Université de Montréal

Implémentation de la multiplication des grands nombres par FFT dans le contexte des algorithmes cryptographiques.

par
Kassem Kalach

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

août, 2005

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

Implémentation de la multiplication des grands nombres par FFT dans le contexte des algorithmes cryptographiques.

présenté par:

Kassem Kalach

a été évalué par un jury composé des personnes suivantes:

Alain Tapp
président-rapporteur

Jean-Pierre David
directeur de recherche

Gilles Brassard
codirecteur

Michel Boyer
membre du jury

Mémoire accepté le

13 décembre 2005

RÉSUMÉ

La multiplication de grands entiers est la base de la plupart des cryptosystèmes à clé publique, notamment RSA, ElGamal et les cryptosystèmes basés sur les courbes elliptiques. Ainsi, les algorithmes de multiplication ont été étudiés largement et intensivement.

La plupart des travaux sont basés sur la méthode de multiplication de Montgomery et ses variantes. Les chercheurs ont toujours évité l'algorithme de la transformée de Fourier rapide (FFT) croyant qu'il est toujours inintéressant pour les systèmes contemporains malgré son faible ordre de complexité ($O(n \log n)$).

Dans cette recherche, nous présentons une étude approfondie des algorithmes de multiplication et leur implémentations dans le contexte des algorithmes cryptographiques. Ensuite, nous proposons une architecture et son implémentation matérielle de l'algorithme de la FFT en utilisant l'arithmétique modulaire pour calculer efficacement la multiplication de très grands entiers.

L'algorithme a été mis en application en utilisant CASM, un langage de description du matériel (HDL) qui a été développé dans notre laboratoire. La technologie cible est un FPGA. L'algorithme est paramétrable et peut facilement être adapté à n'importe quelle taille d'opérande.

Nos résultats indiquent qu'une telle implémentation de l'algorithme commence à être utile pour les opérandes de 4096 bits et plus.

Keywords: Cryptographie, Multiplication Modulaire, Multiplication de Grands Entiers, FPGA, Transformée de Fourier Rapide (FFT).

ABSTRACT

Multiplication of large integers is the foundation of most public-key cryptosystems, specifically RSA, ElGamal and the Elliptic Curve cryptosystems. Thus modular multiplication algorithms have been studied widely and extensively.

Most of works are based on the well known Montgomery Multiplication Method and its variants. Authors have always avoided the Fast Fourier Transform (FFT) method believing that it is impractical for present systems despite its smaller complexity order ($O(n \log n)$).

In this research, we present a thorough study of the algorithms of multiplication and their implementations in the context of the cryptographic algorithms. Then, we propose the design and hardware implementation of a FFT-based algorithm using modular arithmetic to efficiently compute very large number multiplications.

The algorithm has been implemented in CASM, an HDL developed in our laboratory. The target architecture is a FPGA. The algorithm is scalable and can easily be mapped to any operand size.

Results show that such algorithm implementation starts to be useful for 4096-bit operands and beyond.

Keywords: Cryptography, Modular Multiplication, Big Integer Multiplication, FPGA, Fast Fourier Transform (FFT).

TABLE DES MATIÈRES

RÉSUMÉ iii

ABSTRACT iv

TABLE DES MATIÈRES v

LISTE DES TABLEAUX x

LISTE DES FIGURES xi

LISTE DES APPENDICES xiv

LISTE DES SIGLES xv

NOTATION xvii

DÉDICACE xviii

REMERCIEMENTS xix

CHAPITRE 1 : INTRODUCTION 1

1.1 Aperçu historique 1

1.2 Implémentation de la multiplication 11

1.3 Notre approche 20

CHAPITRE 2 : ÉTAT DE L'ART 26

2.1 Algorithmes de multiplication 26

2.1.1 L'algorithme classique de multiplication 27

2.1.2 L'algorithme de multiplication à la russe 29

2.1.3 L'algorithme de Horner 30

2.1.4 L'algorithme de Karatsuba 32

2.1.5 La transformée de Fourier rapide (FFT) 36

Procédure de la multiplication de deux entiers par la FFT . 38
2.1.6 La méthode de multiplication de Montgomery (MMM) . . . 39
Représentation de Montgomery 39
Réduction de Montgomery 40
Principe de réduction de Montgomery 41
Algorithme de réduction de Montgomery (RM) 42
2.2 Représentation de données 45
2.2.1 Systèmes de congruences (RNS) 46
2.2.2 Théorème du reste chinois (CRT) 47
2.2.3 Les systèmes de numération redondants (RNR) 49
Les systèmes de numération à chiffres signés 50
2.3 Technologies 51
2.3.1 Processeur 51
2.3.2 Microcontrôleur 52
2.3.3 DSP 53
2.3.4 Réseaux systoliques 54
2.3.5 ASIC 56
Les réseaux prédiffusés (GA) 57
Les cellules standards (SC) 58
Les cellules personnalisées ou à la demande (FC) 58
Avantages et inconvénients 59
2.3.6 Les circuits logiques reconfigurables 60
PLD 61
CPLD 64
2.4 La technologie FPGA 64
2.4.1 FPGA fabriqué par Altera 66
2.4.2 Programmation ou configuration des FPGA 68
2.4.3 Avantages et limites des FPGA : 73

CHAPITRE 3 : TRANSFORMÉE DE FOURIER RAPIDE 76

3.1 Polynômes 77

3.1.1 Définition 77

3.1.2 Opérations sur les polynômes 77

3.1.3 Représentations des polynômes 78

Représentation par coefficients 79

Représentation par valeurs 79

Interpolation 81

Multiplication rapide de polynômes 82

3.2 Transformée de Fourier 83

3.3 Transformée de Fourier discrète (DFT) 83

3.3.1 Interprétation polynomiale de la DFT 83

3.3.2 Interprétation matricielle de la DFT 84

3.3.3 Exemples de la DFT 85

Exemple : DFT à deux points ($n = 2$) 85

Exemple : DFT à quatre points ($n = 4$) 85

3.3.4 DFT inverse 87

3.4 La transformée de Fourier rapide (FFT) 88

3.4.1 Racines $n^{\text{ièmes}}$ de l'unité 88

Exemple : Racines $n^{\text{ièmes}}$ de l'unité dans $\mathbf{C}$ 90

Exemple : Racines $n^{\text{ièmes}}$ de l'unité dans $\mathbf{Z}_m$ 90

3.4.2 Algorithme de la FFT (Cooley-Tukey) 91

3.4.3 Complexité de la FFT récursive 94

3.5 Architectures efficaces de la FFT 94

3.5.1 La FFT itérative 94

Algorithme de la FFT itérative 96

Opération papillon (*Butterfly operation*) 97

Permutation du vecteur (*bit inverison*) 98

Complexité 99

3.5.2 La FFT Parallèle 99

3.6 La FFT en $\mathbf{Z}_m$ 99
3.6.1 Procédure générale de la FFT en $\mathbf{Z}_m$ 102
3.6.2 Application : multiplication des polynômes 102
3.6.3 Application : multiplication des grands entiers 104
Principe général 105
Paramètres optimaux pour la FFT en $\mathbf{Z}_m$ 106
Exemple 107
Complexité 108

CHAPITRE 4 : ARCHITECTURES ET IMPLÉMENTATIONS . . 110
4.1 Méthodologie 110
4.2 CASM 111
4.3 Détail de l'algorithme 112
4.3.1 Ressources de calcul 112
4.3.2 Ressources de mémoire 114
4.3.3 Structure de contrôle 115
4.4 Implémentation 115
4.4.1 Java 116
Calcul des paramètres de la FFT 116
Ressources de calcul 118
Opération papillon 118
Boucle principale de la FFT 119
Autres calculs 119
4.4.2 CASM, architecture I 120
4.4.3 Opérateurs de calcul 125
Addition modulaire 125
Propriétés des modulos $2^n + 1$ 126
Multiplication modulaire 126
4.4.4 CASM, architecture II 128
4.5 Résulats 132

4.5.1 Tests de validité 133
4.5.2 Résultats de synthèse 135
4.5.3 Résultats de simulation 135

CHAPITRE 5 : CONCLUSION ET PERSPECTIVES 137
5.1 La cryptographie et la multiplication des entiers 137
5.2 Algorithmes de multiplication 138
5.3 Architectures et techniques 140
5.4 La technologie FPGA 141
5.5 La transformée de Fourier rapide et la multiplication des entiers . . 142
5.6 Cryptographie du futur 144
5.7 Perspectives 144

BIBLIOGRAPHIE 146
A.1 Chiffrement de César clxxiii
A.2 Chiffrement mono-alphabetique clxxiii
A.3 Matrice ou carré de Vigenère clxxiii

LISTE DES TABLEAUX

2.1 Multiplication classique 27
2.2 Exemple de multiplication à la russe 29
2.3 Exemple de multiplication de Montgomery 45
2.4 Système de numération redondant 50
2.5 Système de numération à chiffres signés. 50
2.6 Représentation de systèmes redondants 51

3.1 Complexité de multiplication des polynômes 82
3.2 Réorganisation des coefficients du vecteur initial 98

4.1 Résultats de synthèse 135

A.1 Chiffrement de César clxxiii
A.2 Chiffrement mono-alphabetique clxxiii

LISTE DES FIGURES

1.1 Une scytale 2
1.2 Nombre d'opérations selon les paramètres 22

2.1 Schéma de la multiplication de polynômes par la FFT 37
2.2 Les architectures systoliques les plus utilisées 55
2.3 Vue d'ensemble de différentes technologies 57
2.4 Architecture d'un PLA 62
2.5 Programmation d'un PLA 62
2.6 Architecture d'un PAL 63
2.7 Architecture simplifiée d'un FPGA 65
2.8 Architecture MAX simplifiée 67
2.9 Architecture d'un LUT 68
2.10 Étapes de programmation d'un FPGA 69
2.11 Compilation d'un circuit 71
2.12 Simulation d'un circuit 72
2.13 Comparaison de différentes technologies 75

3.1 Schéma de la multiplication de polynômes par la FFT 82
3.2 Interprétation polynomiale de la FFT 84
3.3 Schéma du théorème de la convolution par la FFT 87
3.4 Arbre de récursivité de la FFT 95
3.5 Simulation de la FFT itérative 96
3.6 L'opération papillon 97
3.7 Nombre d'opérations selon les paramètres 107

4.1 Architecture globale 121
4.2 Interface du circuit FFT 123
4.3 Interface du circuit de multiplication 127
4.4 Architecture II 128

4.5 Architecture interne de la FFT 129
4.6 Chiffres de poids faible du nombre produit 136
4.7 Chiffres de poids fort du nombre produit 136

A.1 Matrice ou carré de Vigenère clxxiv

LISTE DES ALGORITHMES

1 Algorithme Classique de Multiplication 28
2 Algorithme de Multiplication à la Russe 30
3 Algorithme de Horner pour calculer $p(\alpha)$ 31
4 Algorithme de Multiplication de Horner 32
5 Algorithme Classique Récursif de Multiplication (ACRM) 33
6 Algorithme de Multiplication de Karatsuba (AMK) 35
7 Réduction de Montgomery (RM) 42
8 Méthode de Multiplication de Montgomery (MMM) 43
9 Méthode de Multiplication de Montgomery (optimisée) 43
10 Méthode de Multiplication de Montgomery (binaire) 44
11 FFT Récursive (FFTR) 93
12 FFT Itérative (FFTI) 97
13 Permutation par inversion de bits 98
14 FFT Itérative 113
15 Algorithme de la fonction Mirroir clxxv

LISTE DES APPENDICES

Annexe A : Systèmes de chiffrement clxxiii

Annexe B : Permutation par inversion de bits (miroir) . . . clxxv

Annexe C : Exécution de la procédure de la multiplication des entiers de 1024 bits par la FFT clxxvi

LISTE DES SIGLES

AES	**A**dvanced **E**ncryption **S**tandard
ARM	**A**dvanced **RISC** Machine
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
CASM	**C**hannel-based **A**lgorithmic **S**tate **M**achine
CPLD	**C**omplexe **PLD**
CRT	**C**hinese **R**emainder **T**heorem
CSA	**C**arry **S**ave **A**dder
DES	**D**ata **E**ncryption **S**tandard
DFT	**D**iscrete **F**ourier **T**ransform
DSP	**D**igital **S**ignal **P**rocessing
DSP	**D**igital **S**ignal **P**rocessor
ECC	**E**lliptic **C**urve **C**ryptosystems
EM	**E**xponentiation **M**odulaire
FC	**F**ull **C**ustom
FFT	**F**ast **F**ourier **T**ransform
FIPS	**F**ederal **I**nformation **P**rocessing **S**tandard
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FPLD	**F**ield **P**rogrammable **L**ogic **D**evice
GA	**G**ate **A**rray
GAL	**G**eneric **A**rray **L**ogic

HDL	**H**ardware **D**escription **L**anguage
LUT	**L**ook-**U**p **T**able
MM	**M**ultiplication **M**odulaire
MMM	**M**éthode de **M**ultiplication de **M**ontgomery
NIST	**N**ational **I**nstitute of **S**tandards and **T**echnology
OTP	**O**ne **T**ime **P**ad
PAL	**P**rogrammable-**A**nd **L**ogic
PKC	**P**ublic **K**ey **C**ryptosystems
PLA	**P**rogrammable **L**ogic **A**rray
PLD	**P**rogrammable **L**ogic **D**evice
RISC	**R**educed **I**nstruction **S**et **C**omputer
RNR	**R**esidue **N**umber **R**epresentation
RNS	**R**esidue **N**umber **S**ystem
RSA	**R**ivest, **S**hamir, et **A**dleman
SC	**S**tandard **C**ell
SDR	**S**igned **D**igit **R**epresentation
SET	**S**ign **E**stimation **T**echnique
SoB	**S**ystem **o**n **B**oard
SoC	**S**ystem **o**n **C**hip
VHDL	**V**HSIC (Very High Speed Integrated Circuits) **HDL**

NOTATION

$\mathbf{Z}$	L'ensemble des nombres entiers
$\mathbf{Z}_m$	L'ensemble des nombres entiers modulo un entier m
$\mathbf{F}_{p^n}$	Le corps d'entiers dont le nombre d'éléments est p^n
$\mathbf{R}$	L'ensemble des nombres réels
$\mathbf{R}^+$	L'ensemble des nombres réels positifs
$\mathbf{C}$	L'ensemble des nombres complexes
i	Un "nombre imaginaire" tel que $i^2 = -1$

À celui dont j'ai honte de mentionner le nom dans ce modeste travail !

REMERCIEMENTS

Mes remerciements vont à toutes les personnes qui ont contribué d'une manière ou d'une autre à me faire progresser et faire progresser mes connaissances.

Je tiens à remercier mon directeur de recherche, Jean-Pierre David, pour avoir accepté de diriger mon travail. Merci à lui de m'avoir indiqué la voie, ainsi que pour sa disponibilité et ses conseils qui m'ont permis de mener à bien cette recherche.

J'exprime ma plus profonde reconnaissance et gratitude à mon codirecteur de recherche, Gilles Brassard, pour m'avoir accueilli dans son laboratoire de recherche, le LITQ (Laboratoire d'Informatique Théorique et Quantique). Merci pour ses précieux conseils, ainsi que pour son soutien et pour m'avoir donné l'occasion de participer à la conférence Eurocrypt'04. Je voudrais le remercier aussi spécialement pour son livre d'algorithmique *Brassard et Bratley*, la principale référence qui m'a facilité le chemin tout au long de cette recherche.

Merci à tous ceux qui m'ont aidé à rédiger ce mémoire, spécialement, mes collègues du laboratoire LITQ (Hugue, Sébastien, et Serge) et Julie du laboratoire LISA. Un merci spécial à tous ceux qui ont sacrifié une partie de leur temps pour m'aider durant l'apprentissage de LaTeX, en particulier, Anne. Jean-Pierre mérite encore un très bon merci dans ce sens pour m'avoir encouragé à rédiger mon mémoire par l'intermédiaire de cet outil professionnel.

Merci à mes amis qui me partagent l'ambition et l'enthousiasme. Mes remerciements vont aussi au personnel de la bibliothèque et du support technique. Finalement, je voudrais remercier ceux qui m'aiment et que j'aime sans limite, mes parents. Merci pour leur amour, leur tendresse et leur confiance en moi.

CHAPITRE 1

INTRODUCTION

Les cryptographes ainsi que les militaires disent souvent que *si les alliés n'avaient pas pu briser la machine de chiffrement de l'armée allemande, Enigma, le cours de la deuxième guerre mondiale aurait été changé.* Cette affirmation montre clairement à quel point la confidentialité de l'information est vitale dans le domaine militaire. La protection des données est aussi fondamentale pour le grand public, surtout depuis la venue au monde d'Internet et l'apparition du commerce électronique. Des pertes de plusieurs millions de dollars ont lieu chaque année à cause d'opérations financières illégales se faisant via Internet. Ainsi, le rôle de la cryptographie, solution possible à ces questions cruciales, se révèle capital.

1.1 Aperçu historique

Cryptographie de l'antiquité : Le terme *cryptographie* vient de deux mots grecs *Kruptos* et *Graphein* qui signifient respectivement *caché* et *écrire*. Néanmoins, l'histoire de la cryptographie remonterait aussi loin que l'origine de l'homme et le moment où il apprit à communiquer. C'est ce qui a fait de la cryptographie une science intimement liée à l'histoire de l'homme et qui a mis des siècles pour prendre forme. Des découvertes archéologiques ont montré que le point de départ fut l'ancienne Égypte.

Il y a environ 4 000 ans (1 900 av. J.C.) dans une ville appelée *Menet Khnufu* de l'ancienne Egypte, un scribe dessina des *hiéroglyphes* indiquant l'histoire de la vie de son seigneur. Ceci fut le premier évènement documenté dans l'histoire de la cryptographie [Kah66]. Il s'agissait d'un système élémentaire qui remplaçait quelques symboles hiéroglyphiques standards par des symboles peu communs. L'intention n'était pas de rendre un texte secret mais plutôt de lui attribuer une dignité et une autorité. Le scribe fonda ainsi le principe cryptographique de la transformation de

l'écriture.

Tout au long de ses 3000 premières années, la cryptographie ne connut pas de grand développement. Elle est apparue de façon isolée dans plusieurs civilisations telles que la civilisation grecque, perse, et arabe mais elle s'est pratiquement éteinte avec la fin de chacune d'elles.

Au $V^{\text{ième}}$ siècle av. J.C., Histaiaeus écrivit à Aristagoras sur le crâne rasé d'un esclave, attendit la repousse de ses cheveux puis l'envoya. L'esclave, à son arrivée, déclara seulement : "Rase moi le crâne". Aristagoras, après qu'un barbier eut rasé la tête de l'émissaire, trouva une phrase tatouée l'invitant à se révolter contre les Perses. Percer un trou à l'aide d'une aiguille très fine au-dessus de caractères judicieusement choisis d'un texte anodin fut une autre technique utilisée pour la dissimulation d'un message confidentiel. Nous précisons que ces deux techniques *stéganographiques*, destinées à dissimuler le message lui-même, doivent être distinguées des techniques *cryptographiques*, destinées à cacher le sens du message.

Le premier témoignage attestant de l'utilisation des moyens techniques permettant de chiffrer les messages vint de la Grèce, vers le $V^{\text{ième}}$ siècle av. J.C. où les spartiates mirent au point le premier dispositif cryptographique militaire nommé *scytale*. La scytale était un bâton de bois autour duquel l'expéditeur enroulait une lanière de cuir ou une bandelette de papyrus. Il écrivait longitudinalement sur le bâton, puis il déroulait la bandelette et l'expédiait au destinataire. La bandelette, une fois déroulée, semble couverte d'une suite de lettres incohérente. Sans la connaissance du diamètre du bâton qui joue le rôle de clé, il était impossible de déchiffrer le message [Beu94].

FIG. 1.1 – Une scytale

Chiffrement de César : Le premier vrai système cryptographique qui se servit des mathématiques fut utilisé par Jules César pour fins militaires [Yas02]. Ce

chiffrement, qui remonte au premier siècle av. J.C., consiste en un décalage de l'alphabet de trois rangs vers la droite (cf Annexe A). Par exemple, le chiffrement du message "ESPION" est "HVSLRQ".

Néanmoins, la sécurité offerte par ce système de cryptage se révéla très limitée puisqu'il n'offre que 25 manières possibles[1] pour crypter un message. Toutefois, il est toujours un bon exemple pédagogique pour introduire les principes cryptographiques.

Chiffrement mono-alphabétique : Le chiffrement de César fut amélioré et généralisé par le cryptage par *substitution monoalphabétique*. Il s'agit d'une technique rudimentaire qui consiste à définir un alphabet crypté généré à partir d'une permutation de l'alphabet clair (cf Annexe A). Le nombre de clefs d'une substitution mono alphabétique s'élève à $26! - 1 \approx 4 \cdot 10^{26}$, rendant le décryptage a priori impossible. Ce procédé résista plusieurs siècles à la perspicacité des cryptanalystes avant qu'il ne fut brisé au début du $IX^{\text{ième}}$ siècle par un mathématicien arabe, Al-Khalil ibn Ahmad al Farahidi, lorsqu'un cryptogramme (en Grec) lui fut envoyé par l'empereur byzantin.

L'âge d'or de la civilisation arabe débuta en 750 avec l'avènement des Abbassides. Les savants arabes étudièrent les méthodes mono-alphabétiques et découvrirent des procédés permettant de les briser, devenant ainsi, comme le souligne David Kahn, les premiers cryptanalystes de l'histoire.

> *"Cryptology was born among the Arabs. They were the first to discover and write down the methods of cryptanalysis"* [Kah66].

Les travaux effectués par le linguiste, mathématicien et philosophe al-Farahidi constituèrent les fondements de la cryptanalyse [Beu01]. Ce savant développa des outils tels que l'analyse de fréquences ainsi que le calcul des combinaisons, des substitutions et des permutations. Auteur de 290 ouvrages scientifiques, Abu Yusuf Al-Kindi exploita les découvertes d'al-Farahidi afin de proposer une méthode de décryptage révolutionnaire qui se base l'analyse fréquentielle du message crypté.

[1] Le cas où les deux alphabets sont identiques est évidemment dépourvu d'intérêt.

Chiffrement de Vigenère : Après la découverte d'al-Kindi, tous les chiffrements devinrent vulnérables à la cryptanalyse par l'étude des fréquences jusqu'à l'invention du cryptage polyalphabétique par Leon Battista Alberti vers 1465. Alberti suggéra d'utiliser plusieurs alphabets cryptés pour chiffrer un message. Blaise de Vigenère, diplomate français, découvrit les écrits d'Alberti, les étudia en détail et publia en 1586, dans son *Traité des chiffres*, une nouvelle méthode de cryptage. Il s'agit d'une matrice de 26×26 cases, appelée tableau de Vigenère, dont chaque ligne contient l'alphabet décalé d'une lettre par rapport à la ligne précédente [JS89]. Le texte chiffré s'obtient en prenant l'intersection de la ligne qui commence par la lettre à coder avec la colonne qui commence par la lettre de la clé. Dès que l'on atteint la fin de la clé, on recommence à la première lettre. Par exemple, "TJB FTVO" est le message chiffré de "IBN SINA" avec la clé "LION" (cf Annexe A).

L'avantage de cette méthode est qu'une lettre du texte crypté correspond à plusieurs lettres du texte en clair. L'analyse fréquentielle se révèle ainsi inefficace et les failles du cryptage mono-alphabétique sont palliées. Ce chiffrement résista plus de deux siècles jusqu'à la première attaque réussie publiée en 1863 par le major allemand Friedrich Kasiski [Beu94].

Principe de Kerckhoffs : En 1883, Auguste Kerckhoffs publia un ouvrage fondamental [Ker83], intitulé *la cryptographie militaire*, où il formula quelques principes indispensables des cryptosystèmes modernes. Notamment, que *"la sécurité d'un système cryptographique ne doit pas reposer sur la sécurité de l'algorithme, mais seulement sur la sécurité d'un paramètre aisément modifiable, la clé"*. En revanche, l'algorithme lui doit être largement diffusé afin de pouvoir être étudié en profondeur par des spécialistes. Ses faiblesses seront ainsi mises potentiellement en évidence et il pourra être amélioré ou abandonné. Depuis, la conception des systèmes cryptographiques devrait répondre à cette règle d'or.

Le masque jetable : En 1917, Gilbert Vernam conçut une méthode de chiffrement sécuritaire, nommée *masque jetable* (**O**ne **T**ime **P**ad). En fait, il s'agit d'une forme de chiffrement pratique de Vigenère qui consiste à faire le ou exclusif (XOR) entre le message et la clé. Bien qu'étant très simple, cette méthode est tellement sûre

qu'elle était utilisée pour les communications entre Moscou et Washington par le fameux téléphone rouge pendant la guerre froide. La clé privée était alors échangée grâce à une valise diplomatique jouant le rôle de canal sécurisé. Cependant, cette méthode est irréaliste à cause de ses inconvénients que sont la génération et le transport des clés. En effet, une clé doit être parfaitement aléatoire, au moins aussi longue que le message qui va être chiffré, et tout cela pour une seule utilisation.

Enigma : Enigma, mise au point en 1923, fut la première machine technique de chiffrement. Il s'agissait d'un dispositif électromécanique qui réalisait une substitution poly-alphabétique. Cette machine est composée, entre autre, d'un clavier, d'un jeu de disques rotatifs adjacents appelés roues, arrangés le long d'un axe, et d'un mécanisme entraînant en rotation un ou plusieurs des roues chaque fois qu'une touche est pressée. Le mécanisme fonctionne à la manière d'un compteur de vitesse automobile. Chaque roue étant câblée de manière qu'elle représente une substitution possible de 26 lettres de l'alphabet, le nombre de roues possibles est ainsi 26!, ce qui est énorme à l'époque. Une roue utilisée toute seule ne réalise en fait qu'un chiffrement très simple, le chiffre mono-alphabétique. La complexité de la machine Enigma provient de l'utilisation de plusieurs roues en série, généralement trois ou plus, et leurs mouvements réguliers. Cela augmente énormément l'espace des clés et offre, en principe, un système de chiffrement très fort.

En 1926, la marine allemande reprend le projet d'Enigma. En 1929, Enigma version M3 fut adoptée par la Wehrmacht, l'armée allemande, et utilisée pendant la deuxième guerre mondiale [TW02]. Avant même le début la guerre, Enigma fut brisée par un groupe de trois cryptologistes polonais. Leurs techniques ont été ensuite passées aux Anglais en 1939. Les Anglais ont amélioré les techniques polonaises et déchiffré avec succès les messages allemands pendant toute la deuxième guerre mondiale. La faiblesse majeure d'Enigma était son algorithme de chiffrement ; il ne se basait pas sur le principe de Kerckhoffs.

Théorie de Shannon : Tous les systèmes antérieurs de chiffrement manquaient d'une fondation mathématique permettant de mesurer et de quantifier leur résistance à d'éventuelles attaques. En 1948 et 1949, deux articles fondamentaux de

Claude Shannon, *A mathematical theory of communication* [Sha48] et *The communication theory of secrecy system* [Sha49], donnèrent des assises mathématiques à la cryptographie. Shannon prouva aussi que le masque jetable est l'unique méthode qui permet d'assurer la confidentialité parfaite[2]. Depuis, on parle de sécurité calculatoire ou de sécurité inconditionnelle au sens de la théorie de l'information de Shannon. Un système cryptographique moderne devrait vérifier les trois principes fondamentaux suivants :

- Principe de Kerckhoffs
- Clés de taille raisonnable
- Calculatoirement ou inconditionnellement sûr

DES : Après la deuxième guerre mondiale, la recherche en cryptologie fut limitée au domaine militaire. Au début des années 70, surtout après l'apparition des réseaux, les besoins des civils pour sécuriser leurs informations telles que les banques, les universités, les ministères ont considérablement augmenté. Le 15 mai 1973, le NBS (**N**ational **B**ureau of **S**tandards) des États-Unis (connu maintenant NIST pour **N**ational **I**nstitute of **S**tandards and **T**echnology) lança un appel d'offre de système cryptographique dans le journal officiel américain (*Federal Register*) [Sti03].

Fin 1974, IBM proposa un système appelé *Lucifer*. IBM modifia Lucifer et le publia, le 17 mars 1975, sous le nom DES (**D**ata **E**ncryption **S**tandard). Après un nombre considérable de débats publics, DES fut adopté comme standard en 1977. Depuis son adoption, DES fut réévalué par le NIST tous les 5 ans. DES est un chiffrement en bloc qui opère à l'aide d'une clé secrète de 56 bits. Il consiste en 16 itérations, formées de permutations et de substitutions, où chaque itération est contrôlée par une clé partielle de 48 bits extraite de la clé initiale. Avec une clé de 56 bits, environ 72 millions de milliards (2^{56}) de clés sont possibles. Ceci peut sembler énorme, pourtant la critique de DES la plus sérieuse porta sur la taille trop petite de l'espace de clés !

Triple DES ou 3DES : Le perfectionnement de la cryptanalyse et la fabuleuse

[2]Notons que Shannon n'a pas mentionné la mécanique quantique dans son énoncé.

croissance de la puissance des ordinateurs mirent fin au DES. Le 17 juin 1997, DES fut brisé en 3 semaines par un regroupement de petites machines sur Internet. En outre, une machine de recherche exhaustive coûtant 250,000 euros fut réalisée en 1998. Cette machine, baptisé *DES Cracker* peut chercher 88 milliards de clefs par seconde. Elle gagna le concours de RSA, DES Challenge II-2, en trouvant une clé DES en 56 heures [Sti03]. Ainsi, remplacer DES était devenu incontournable. La solution fut dans un premier temps l'adoption de 3DES, trois applications de DES à la suite avec 2 clés DES différentes.

Si le 3DES semble largement suffisant à l'heure actuelle, il est malheureusement trois fois plus lent que le DES. DES est aussi faible contre deux autres types d'attaques plus sophistiquées que la recherche exhaustive : la cryptanalyse *différentielle* et la cryptanalyse *linéaire* [Hey02]. Quand DES ne fut plus capable de répondre aux besoins de sécurité, le NIST lança, le 2 janvier 1997, un nouvel appel à contribution mondiale pour remplacer DES. Le successeur de DES sera alors l'AES (**A**dvanced **E**ncryption **S**tandard).

Rijndael ou AES : Suite à l'appel du NIST, des vingt-et-un cryptosystèmes soumis, seulement quinze remplissaient tous les critères nécessaires et ont été acceptés en tant que candidats AES [Sti03]. Le 20 août 1998, le NIST présenta ces quinze candidats lors de la première conférence pour le candidat de AES (*First AES Candidate Conference*). En août 1999, cinq candidats furent acceptés comme finalistes : *MARS, RC6, Rijndael, Serpent et Twofish.* Le 2 octobre 2000, Rijndael fut choisi en tant que standard avancé. Après une dernière période d'appel à commentaires, il fut officialisé le 26 mai 2002 comme le standard FIPS-197 (**F**ederal **I**nformation **P**rocessing **S**tandard). Il est notamment utilisé par le gouvernement américain ainsi que par beaucoup d'autres organisations dans le monde pour protéger les informations sensibles.

Rijndael fut inventé par deux belges, Joan Daemen et Vincent Rijmen, d'où le nom du système. Il s'agit d'un chiffrement par bloc de 128 bits mais dont la longueur de clés est variable, à savoir 128, 192 et 256 bits. Comme DES, AES est un algorithme de chiffrement *itéré*. Le nombre d'itérations N dépend de la

longueur de la clé. Pour chaque itération, on fait une substitution selon une boite mystérieuse appelée *S-Box* (Boîte-S). Stinson présente une belle illustration de AES [Sti03]. AES a résisté à tous les types connus d'attaques jusqu'aujourd'hui.

Limites de la Cryptographie Symétrique : Tous les systèmes présentés jusqu'ici sont des systèmes *cryptographiques symétriques (ou à clé privée)* reposant sur la communication secrète préalable d'une clé K entre deux personnes, nommées souvent par les cryptographes Alice et Bob, avant de commencer à transmettre les messages. Ainsi, se pose le problème de distribution de clés lorsqu'il y a un nombre raisonnable d'utilisateurs. Imaginez un groupe de n personnes utilisant un cryptosystème à clés secrètes. À cette fin, il est nécessaire de distribuer et gérer confidentiellement $n(n-1)/2 = O(n^2)$ clés différentes. C'est un problème majeur qui a limité l'essor de la cryptographie symétrique et même de la cryptographie de manière générale jusqu'à la découverte de la *cryptographie asymétrique (ou à clé publique*)

Cryptographie à clé publique : *Le besoin est la mère de l'invention.* L'objectif de la cryptographie à clé publique est de permettre à deux personnes, Alice et Bob, de communiquer confidentiellement sans aucun secret préalable. En 1976, Diffie et Hellman proposèrent l'idée d'un système cryptographique à clé publique [DH76]. Nous soulignons que Ralph Merkle y avait pensé avant eux, lui-même précédé les services secrets britaniques. L'idée de la cryptographie à clé publique ne mena pas seulement à une modification fondamentale de la cryptographie mais aussi à la naissance et au développement solide de nouvelles branches dans les mathématiques. La notion d'une fonction à sens unique, sur laquelle se fonde leur idée, fut introduite [Yas02]. Dans un cryptosystème à clé publique, Alice (ou toute autre personne) peut envoyer un message à Bob en utilisant une clé publique, qui sera publiée par exemple dans un annuaire. Bob est la seule personne capable de déchiffrer le message en utilisant sa clé privée. Notons que la clé privée et la clé publique sont reliées par une fonction à sens unique qui rend calculatoirement très difficile de retrouver la clé privée à partir de la clé publique. Depuis lors, nombreux sont les cryptosystèmes qui ont été proposés et implémentés selon ce

principe. RSA (**R**ivest, **S**hamir et **A**dleman) [RSA78], le système ElGamal [ElG85], le protocole d'échange de clé de Diffie-Hellman [DH76] et les cryptosystèmes basés sur les courbes elliptiques (ECC pour **E**lliptic **C**urve **C**ryptosystems) sont parmi les plus connus au monde.

La sécurité de tous ces systèmes est basée sur la difficulté calculatoire d'un certain problème mathématique. RSA est fondé d'une part sur la la difficulté de la factorisation des grands entiers et d'autre part sur l'exponentiation modulaire qui n'est quant à elle qu'une série de multiplications modulaires. Les protocoles de Diffie-Hellman et ElGamal sont fondés sur le problème du logarithme discret dont l'opération de base est la multiplication modulaire. Pareillement, la multiplication modulaire est l'opération essentielle dans les ECC. Ainsi, la multiplication de grands entiers est la pierre angulaire de tous ces cryptosystèmes modernes.

La complexité de la méthode classique de la multiplication est $O(n^2)$. Par exemple, si on veut multiplier deux entiers de 1000 chiffres (≈ 4096 bits), on a besoin de 1 000 000 opérations. C'est la raison pour laquelle les algorithmes de multiplication sont étudiés largement et intensivement pour réduire la complexité de cette opération coûteuse.

Cryptographie quantique : Les cryptosystèmes à clé publique (PKC pour **P**ublic **K**ey **C**ryptosystems) bien que très utiles en pratique souffrent de deux défauts principaux. Le premier défaut est qu'ils sont rendus plus faibles par la progression de la technologie puisqu'ils n'offrent qu'une sécurité calculatoire. Rien n'empêche en effet avec les ressources de calcul nécessaires de pouvoir briser n'importe quel PKC.

Peter Shor a conçu en 1994 un algorithme quantique[3] qui permet de factoriser un entier de n bits efficacement [Spi05], soit en temps asymptotiquement proportionnel à $n^{2+\epsilon}$ pour un quelconque petit ϵ [Bra95]. Cet algorithme se base sur la version quantique de la transformée de Fourier discrete et sur la méthode classique de

[3]Un algorithme quantique est un algorithme qui fonctionnerait sur un ordinateur quantique. Les ordinateurs quantiques sont des ordinateurs s'appuyant sur les lois de la mécanique quantique pour traiter l'information. Dû au défi technologique que représente la construction d'un tel ordinateur, ils font toujours partie pour l'instant du domaine de la recherche.

factorisation *"factoring via order finding"*. La réalisation physique d'un ordinateur quantique de taille suffisante mettra par conséquent directement la sécurité des PKC en cause. Brassard, dans son article intitulé "*The Impeding Demise of RSA* ?", souligne en particulier l'effet dramatique que cela aurait sur RSA [Bra95].

Le deuxième défaut des PKC est qu'on n'est jamais arrivé à prouver leur sécurité d'une façon mathématique rigoureuse. En particulier, les mathématiciens essayent depuis longtemps de prouver formellement cette sécurité sans y parvenir. Toutefois, personne n'a encore révélé l'existence d'un algorithme classique qui permettrait d'inverser une fonction à sens unique. La découverte d'un tel algorithme mettrait en péril la sécurité des PKC. Dans le pire cas, il est possible qu'un tel algorithme soit déjà découvert, par une agence gouvernementale par exemple, mais gardé secret. Ainsi, historiquement, le fait que Enigma avait été brisée est demeuré un secret pendant presque 30 ans après la fin de la guerre, en partie parce que les Anglais avaient vendu des machines Enigma à d'anciennes colonies mais sans leur dire que le système avait été brisé [TW02]. Ces deux défauts illustrent le fait que les PKC ne pourront plus garantir dans le futur l'échange de clés cryptographiques d'une manière sûre [IDQ05].

C. Bennett et G. Brassard ont introduit en 1984 une idée révolutionnaire, celle de la cryptographie quantique [BB84] qui permet d'accomplir la distribution de clés avec une sécurité inconditionnelle. Cette sécurité inconditionnelle est garantie non pas par les hypothèses mathématiques mais par les lois de la mécanique quantique. La cryptographie quantique repose sur le principe fondamental suivant : l'observation d'un système quantique perturbe ce système. Autrement dit, si un espion tente d'intercepter les communications et d'apprendre de l'information, le message sera automatiquement modifié. Ceci est la principale différence avec les PKC où si le canal de communication est espionné, il est impossible de s'en rendre compte à cause de la nature classique de l'information qui peut toujours être observée sans être modifiée.

Le principe de base est que si on représente (encode) un bit par un système quantique, toute tentative de l'espion pour apprendre de l'information sur ce bit

le perturbera nécessairement. Cette perturbation causera ainsi des erreurs dans les bits transmis et les deux participants pourront se rendre compte qu'ils ont été espionnés.

En pratique, la cryptographie quantique utilise les photons, particules de la lumière, pour transporter l'information. Chaque photon est polarisé, c-à-d que l'on donne une direction à son champ électrique. La polarisation est mesurée par un angle qui peut varier de 0° à 180°. BB84, le protocole de Bennett et Brassard, est le protocole le plus communément utilisé pour la distribution d'une clé quantique (**Q**uantum **K**ey **D**istribution). Dans celui-ci, la polarisation peut prendre 4 valeurs : $0°, 45°, 90°$ et $135°$ [BB84].

La cryptographie quantique n'est pas seulement une possibilité théorique ; elle a été implémentée dans plusieurs laboratoires dans le monde, entre autres à l'Université de Genève qui a réalisé le processus de la distribution d'une clé quantique via une fibre optique de 70 km de long [Spi05]. Bien que devenue réalité, il est difficile de prévoir à l'heure actuelle quand elle sera utilisée à grande échelle à cause des difficultés technologiques qu'il reste à surmonter. La cryptographie à clé publique reste donc pour l'instant la seule alternative la plus fiable en pratique et le besoin d'algorithmes de multiplication plus efficaces reste pertinent.

1.2 Implémentation de la multiplication

Peu après l'invention de la cryptographie à clé publique et tout au long des deux dernières décennies, une vaste variété d'architectures pour la multiplication des grands entiers est apparue. Ces architectures englobent aussi bien des réalisations matérielles que logicielles pour les gros ordinateurs comme pour les petites puces. La plupart de ces approches sont basées sur la **M**éthode de **M**ultiplication (modulaire) de **M**ontgomery (MMM) [Mon85] combinée avec une ou plusieurs techniques d'amélioration de performance. Citons par exemple les réseaux systoliques [Atr65, Eve90, DL92, Wal93, IMI94, Kor94, TK99, BP01, BP99, Wal00, WHW01, BOPV03, Kim01, WCSG03, FK03, TK03], les systèmes de

numération redondant (en anglais RNR pour **R**edundant **N**umber **R**epresentation) [EW93, Oru95], les systèmes de congruences (RNS pour **R**esidue **N**umber **S**ystem) [PP95, BDK98, BDK01, BI04, CNPQ03] et le théorème du reste chinois (CRT pour **C**hinese **R**emainder **T**heorem) [WHW01, QC82, Gro00].

L'algorithme de Montgomery est communément utilisé dans les implémentations matérielles puisqu'il est efficace et ne requiert pas beaucoup de ressources. Il est surtout employé pour implémenter des crypto-coprocesseurs sur les cartes à puces [NM96, HP00, CAM+02], notamment celles de Motorola et Thomson [NM96].

Comme la MMM a démontré une bonne efficacité matérielle, les autres algorithmes ont été abandonnés. Nous n'avons trouvé que deux implémentations matérielles basées sur l'algorithme de Horner [JB97, BM04] et aucune implémentation matérielle sur l'algorithme de Karatsuba [KO63].

Quant aux travaux logiciels, ils sont rares en ce qui concerne les algorithmes de multiplication puisque l'approche matérielle est plus efficace[4] et plus sécuritaire[5]. L'algorithme de Karatsuba a été utilisé dans trois implémentations logicielles [LHL03, Moe76, SV93] dont une très récente datant de 2003. Concernant l'algorithme de Montgomery, nous avons seulement trouvé deux implémentations logicielles qui tournent sur un processeur générique, dont la plus récente remonte à 1996 [BGV94, KAK96]. Par contre, il est très utilisé dans les cartes à puces basées sur un processeur RISC (**R**educed **I**nstruction **S**et **C**omputer) telle que la carte GempX-

[4]Un matériel dédié (cryptoprocesseur), qui est optimisé pour accomplir une tâche précise, est considérablement plus rapide qu'un processeur générique (voir section ??).

[5]Dans une carte à puce par exemple, le microprocesseur et la mémoire sont contenus dans la même puce. Cette structure permet une sécurité maximale puisqu'on ne peut avoir accès à la mémoire reprogrammable sans passer par le processeur, qui assure les services de sécurité (identification, authentification, cryptage et gestion des modes d'accès). D'autre part, cette carte utilise une grande partie (jusqu'à 1/3) de son système d'exploitation, qui est déjà optimisé, à sa propre protection (mécanismes de contrôle d'accès aux données et de protection contre des attaques externes). Cette carte offre ainsi une résistance au piratage notablement plus élevée que les systèmes informatiques purement logiciels. Muni de ses propres ressources de calculs, cette carte est également capable d'accomplir la majorité de ses traitements avec ses données confidentielles d'une façon complètement indépendante du système auquel elle est reliée. Peu importe qu'on travaille hors réseau ou en réseau avec accès sécurisé, elle n'est pas en conséquence susceptible d'attaques par virus, contrairement aux PC qui en sont victimes systématiquement.

presso 2.0 de Gemplus qui utilise le processeur ARM7M [Dhe98, HQ00, DF01].

Nous allons maintenant récapituler les travaux pertinents qui couvrent une bonne partie des techniques et algorithmes proposés dans la littérature en rapport avec la multiplication des entiers.

Réseaux systoliques basés sur la MMM : L'architecture d'un réseau systolique est une solution attirante pour les systèmes informatiques exigeant des calculs massifs. Elle fut mise au point par A. Atrubin en 1965 [Atr65]. Pourtant, elle ne fut étudiée intensivement du point de vue théorique et pratique qu'après la contribution de S. Even [Eve90]. Ce dernier a combiné la MMM et un réseau systolique en se basant sur le multiplieur d'Atrubin. Le nombre de cycles d'horloge pour une **M**ultiplication **M**odulaire (MM) d'opérandes de n bits était $3n$ et le piplinage y était impossible.

Le réseau systolique présenté par B. Dixon et A. Lenstra [DL92] fut implémenté pour factoriser les courbes elliptiques. Il consiste en un réseau de 128×128 éléments (cellules) de traitement (**P**rocessing **E**lement) ayant chacun 64 KB de mémoire. Néanmoins, il a besoin, entre autres, de 1 GBytes de mémoire.

C. Walter [Wal93] considéra la MMM dans un réseau systolique à deux dimensions comme étant une solution idéale mais peu pratique pour RSA puisque son réseau exige énormément de portes logiques, ce qui constitue un circuit de taille énorme. Dans [Kor94], P. Kornerup proposa une architecture systolique linéaire qui requiert n cellules de traitement et $5n$ bascules (Flip-Flops). Le temps d'exécution d'une **E**xponentiation **M**odulaire (EM) de n bits est $2n^2$ cycles d'horloge, l'exposant étant de n bits également. L'espace exigé a constitué l'obstacle principal à la réalisation d'une implémentation matérielle. La première instance d'un réseau systolique de taille paramétrable a été introduite par A. Tenca et Ç. Koç [TK99]. Le temps d'exécution d'une MMM de deux entiers ayant une précision donnée dépend alors de la profondeur de piplinage choisi.

T. Blum et C. Paar [BP01] ont proposé une architecture systolique et son implémentation en VHDL sur un FPGA (**F**ield **P**rogrammable **G**ate **A**rray) qui est le circuit intégré reconfigurable le plus évolué. Les auteurs ont utilisé la MMM à

base élevée (16 bits) pour implémenter RSA. Une EM complète (la base, l'exposant et le modulo ayant tous la même taille) de 1 kb prend maintenant 29.84% moins de temps comparé avec son travail antérieur [BP99]. Pourtant, l'utilisation d'une base élevée exige une augmentation de 36.34% d'espace.

C.-H. Wu et al. [WHW01] ont combiné un réseau systolique avec la MMM et le CRT pour réaliser un circuit RSA à 512 bits . Une meilleure performance a été observée sur d'autres travaux au prix de 50% d'espace supplémentaire. Le réseau systolique linéaire proposé par S. Örs et al. [BOPV03] est considéré comme étant particulièrement approprié pour RSA ainsi que pour les ECC. D'autres variétés d'architectures sont aussi présentées dans la littérature [WCSG03, FK03, TK03].

Réseaux systoliques non basés sur la MMM :

Comme l'algorithme de Montgomery se révéla être le meilleur choix matériel, il n'existe pas beaucoup d'autres propositions. Deux réalisations matérielles de RSA n'utilisant pas la MMM sont proposées par K. Iwamura et T. Matsumoto [IM92]. Les auteurs essayèrent de créer une architecture flexible en permettant de changer le nombre des cellules de traitement. Pourtant, dans un travail postérieur [IMI94], ils avaient conclu que ce réseau n'est pas aussi efficace que celui basé sur la MMM. Dans [JB97], Y.-J. Jeong et W. Burleson présente deux algorithmes, basés sur la règle itérative de Horner, ainsi que leurs structures systoliques pour la MM de grands entiers. Comparé avec le réseau de Walter [Wal00], ces réseaux exigent non seulement 50% de cycles supplémentaires par multiplication, mais ils sont plus lents et plus complexes. Trois nouvelles architectures visant un réseau de taille paramétrable sans tirer profit de la MMM sont proposées par W. Freking et K. Parhi [FP00]. Les auteurs ont choisit un algorithme qu'ils ont inventé eux-mêmes [FP99] croyant que la MMM n'était pas le meilleur choix pour les calculs systoliques à base élevée.

L'implémentation coûteuse de l'approche systolique en a fait une solution impraticable. Aucune implémentation ne fut publiée jusqu'à 1999 [Blu99] et très peu jusqu'à 2002 [BOPV03]. La recherche s'est ensuite intensifiée afin de trouver d'autres alternatives. Le RNR, y compris l'arithmétique en logique signée, le RNS, le CRT,

la technique de basée élevée ainsi que les additionneurs sans propagation de retenue ont été tous proposés comme remplaçants de l'approche systolique. Cela ne signifie pas que ces solutions soient incompatibles. On peux parfois les trouver, séparées ou combinées, utilisées dans l'architecture systolique.

Architectures non systoliques basés sur la MMM :

Les travaux non systoliques se basant sur la MMM sont très rares également. Dans une architecture non systolique, S. Eldridge et C. Walter décrivent comment implémenter une MMM rapide [EW93]. Le gain de vitesse est dû à une horloge plus rapide grâce à une logique combinatoire simple. La détermination du quotient est le problème le plus complexe dans la MMM à base élevée. Orup en introduit une solution au détriment d'une phase de pré-calcul, pour chaque nouveau modulo, et cela avec une basse fréquence [Oru95].

Système de numération redondant (RNR) :

Le RNR est utilisé pour réduire la propagation de retenues dans les milliers d'opérations d'additions auxquelles se réduisent la MM et l'EM comme c'est le cas de la MMM.

L'arithmétique en logique signée (SDR pour **S**igned **D**igit **R**epresentation), un RNR, fut employée pour la première fois par A. Vandemeulebroecke pour implémenter un simple processeur RSA de 1024 bits [VVDJ89]. Les opérations de multiplications sont réalisées en utilisant à plusieurs reprises l'addition. Aussi sans tirer profit de la MMM, deux algorithmes ont été proposés par N. Takagi et S. Yajima [TY92], un à base 2 et l'autre à base 4, pour la MM en matérielle. Ces deux algorithmes sont exclusivement basés sur la MM dans le RNR et effectuent l'addition sans propagation de retenues. Cependant, ils ne conviennent que pour les applications effectuant des multiplications itératives.

K.-S. Cho et al. [CRC01] ont combiné la technique d'estimation de signe (SET pour **S**ign **E**stimation **T**echnique) avec, entre autres, la méthode binaire d'exponentiation (Binary Method ou Square and Multiply) pour obtenir une réalisation matérielle d'un processeur RSA. Pour un modulo de n bits, une MM nécessite $(\frac{n}{2}+3)$ cycles d'horloge et une opération RSA nécessite $n(\frac{n}{2}+3)$ cycles.

S. Wei et al. [WCS02] ont proposé une implémentation d'un nouvel algorithme de MM en série utilisant cette fois la méthode de codage de Booth et le SDR. Les résultats des multiplications ont été seulement reportés pour des petits modulos, 16 bits.

M. Niimura et Y. Fuwa [NF03] ont proposé une gamme des techniques et des théorèmes pour le calcul modulaire à base 2^k dans le RNR. Toutefois, le RNR qui sert essentiellement à résoudre le problème de propagation de retenues augmente le temps d'exécution, la taille du circuit ainsi que la complexité de la logique du circuit.

Système de congruences (RNS) :

L'arithmétique de RNS, en conjonction avec le CRT, fournit un excellent moyen pour l'arithmétique de très grands entiers. Le parallélisme d'exécution qu'offre le CRT permet d'augmenter la performance d'un facteur de 3.5 [WCS02, BOPV03] voire 4 [Kob94].

La combinaison du RNS avec la MMM fut introduite par K. Posch et R. Posch [PP95] pour surmonter certains inconvénients du RNS tels que la difficulté de la comparaison, la détection de signe et la détection du dépassement de capacité. L'extension de base, technique pour effectuer ces opérations, prend 3/5 de la durée total du traitement. Un nouvel algorithme basé sur la MMM dans RNS et utilisant un anneau de n processeurs simples est proposé par J. Bajard et al. [BDK98], le temps d'exécution est alors en $O(n)$. Ils ont proposés d'utiliser 33 processeurs à 32 bits pour RSA de 1024 bits. Cette solution améliore considérablement la performance du circuit mais la sécurité du RSA est mise en cause puisque le nombre de modulos pour RSA de 1024 bits ne devrait pas dépasser 3. Les mêmes auteurs ont étendu leur travail en abordant un problème complexe du RNS, la conversion de base [BDK01].

Une implémentation RSA basée sur la MMM et le RNS qui n'utilise pas la conversion de base a été proposée par J.-C. Bajard et L. Imbert [BI04]. Le message à chiffrer doit être divisé en blocs qui correspondent à des nombres valides dans le RNS. Outre cette étape supplémentaire, les participants doivent préalablement se

mettre en accord sur l'ensemble de paramètres du RNS, ce qui pourrait compliquer le processus de la communication. Hélas, aucun résultat n'a été reporté.

Récemment, J.-J. Quisquater et al. [CNPQ03] ont présenté une architecture parallèle permettant d'éviter certains types d'attaques analytiques telles que l'attaque temporelle, l'analyse de puissance et l'analyse électromagnétique. Ils ont utilisée la MMM basée sur le RNS pour implémenter sur un FPGA un système capable d'exécuter une signature RSA en parallèle sur un ensemble de coprocesseurs identiques. Cette architecture consomme pourtant d'espace mémoire supplémentaire.

Hélas, l'efficacité du RNS est détérioré par le majeur défaut que pose l'opération de conversion de base entre le système binaire usuel et le RNS. Cette opération, appelée l'extension de base, consomme la plupart de la durée de traitement dans l'implémentation basée sur le RNS. Une variété des travaux s'attaque à cette problématique [IS98, HS03, Pie95, VR94, Wan00] ainsi qu'à l'arithmétique modulaire et la conversion entre le SDR et RNS [WS00, WS01, LNBO03].

Théorème du reste chinois (CRT) :

Cette technique fut proposée pour la première fois par J.-J. Quisquater et C. Couvreur [QC82]. En avril 2000, Compaq et RSA security inc. ont annoncé une nouvelle technologie comme standard du système RSA [BOPV03]. Au lieu du modulo traditionnel $N = pq$, N est devenu un produit de trois nombres premiers (distincts) ou plus. L'idée était d'améliorer la performance en augmentant le nombre de facteurs et en utilisant ensuite le CRT pour des exponentiations parallélisées. Une autre approche avait été introduite par T. Takagi [Tak98] où N est de la forme $p^n q$. Le choix de la taille des facteurs est ainsi critique pour préserver la sécurité de RSA.

L'implémentation de RSA en mode CRT réalisée par J. Grosschädl [Gro00] a permis d'augmenter le taux de déchiffrage par un facteur de 3.5 . L'algorithme de MM, ou l'algorithme de réduction modulaire, de Barret [MaSV97] et la méthode binaire d'exponentiation (square-and-multiply) y ont été utilisés respectivement pour la MM et l'EM. Un prototype a été optimisé dans le cas d'un modulo de 1024 bits et un multiplieur de 16 bits.

L'extension de base prend la majeure partie de la durée de la MMM dans le

RNS. Dans [KKSS00], S. Kawamura et al. ont utilisé l'architecture Cox-Rower parallélisée pour la MMM. Ils ont aussi réalisé un prototype LSI qui implémente une transaction RSA de 1024 bits avec et sans CRT. L'utilisation du CRT améliore la performance d'un facteur de 1.75.

Les articles [BDL97,JLQ99,UBF+03,BOS03,Wag04] présentent les attaques par défaut des algorithmes de signature RSA utilisant le CRT ainsi que leurs contres-mesures. Il faut noter que ce type d'attaque est très répandu pour les cartes à puces.

Travaux basés sur l'algorithme de Horner et de Karatsuba :

Une variante d'algorithmes de MM originaire de C. Koç et C. Hung [KH98], qui est particulièrement appropriés pour l'implémentation sur un FPGA, est présentée par J.-L. Beuchat et J.-M. Muller [BM04]. En particulier, les auteurs utilisent l'algorithme de Horner dans plusieurs implémentations. Cependant, les résultats rapportés ne concernent que la multiplication à petits modulo (n^8, n^{16}, n^{32}). Nous avons déjà mentionné précédemment les travaux qui utilisent la règle de Horner dans une architecture systolique ; ceux-ci sont souvent lents et complexes. Dans un article relativement ancien mais qui reste pertinent [SV93], M. Shand et J. Vuillemin analysent plusieurs techniques et algorithmes pour réaliser un cryptosystème matériel RSA performant. Celui-ci se base entre autres sur le CRT et la méthode de division de Hensel. En outre, pour la partie logicielle, l'algorithme de Karatsuba est utilisé.

La nature itérative l'algorithme de Horner et celui de Karatsuba n'en fait pas de bons candidats pour l'implémentation matérielle.

Autres travaux :

L'approche logicielle/matérielle (Co-Design) a été utilisée par M. Šimka [Š03] pour implementer RSA embarqué sur FPGA. Le fait d'utiliser un co-processeur RSA augmente considérablement la performance du système. Cette implémentation a été faite en utilisant le processeur Nios et le FPGA APEX d'Altera. Un chiffrement RSA de 2048 bits avec l'approche co-design s'accomplit 5 fois plus rapidement que celui avec l'approche logicielle.

H. Sedlak [Sed87] a présenté un processeur cryptographique RSA où l'EM et la MM sont implémentées respectivement comme une série des multiplications et additions. Dans [DQ00], J.-F. Dhem et J.-J. QuisquaterLa présente une alternative à la MMM qui permet d'obtenir des calculs très efficaces sur les nouveaux processeurs RISC et qui sont utilisables dans les cartes à puces.

De récents travaux s'attaquent au problème de la MM et de l'EM de grands entiers, avec ou sans la MMM. Ils utilisent des additionneurs sans propagation de retenues (CSA pour **C**arry **S**ave **A**dder) [KH98, MMM+03, MMM04] et des LUT (**L**ook-**U**p **T**able) [BST02,BS03b,BS03a,BS04a,BS04b] pour mémoriser des valeurs pré-calculées. Les CSA augmentent la complexité ainsi que l'espace du circuit. Les LUT exigent un espace mémoire supplémentaire et du temps additionnel pour pré-calculer certaines valeurs de chaque nouveau modulo.

Citons encore trois autres travaux de références. Dans un de ses articles [Bri90], E. Brickel présente une liste très complète d'implémentations matérielles de RSA. Aussi Ç. Koç [Koç94] détaille une variété d'algorithmes et de techniques ayant trait à la multiplication et l'exponentiation modulaire. Récemment, Örs et al. [BOPV03] présente un sommaire des différentes architectures pour la cryptographie à clé publique .

Travaux logiciels :

Il existe peu d'articles parlant de contributions logicielles. Tel que nous l'avons déjà expliqué, l'approche matérielle est beaucoup plus efficace et souvent plus sécuritaire. Une combinaison entre la méthode classique et l'algorithme de Karatsuba a été utilisée récemment par C.-B. Liu [LHL03] pour la multiplication d'entiers. Cet algorithme hybride consiste à exécuter l'algorithme de Karatsuba récursivement jusqu'à ce que l'on arrive à des nombres de taille 2^k bits, qu'on multiplie ensuite par la méthode classique. La taille 2^k bits, appelée le seuil (*cutoff number*), est critique pour la performance de cet algorithme. L'algorithme de Karatsuba présente sa meilleure performance pour un seuil de $2^4 = 16$ bits indépendamment de la taille du problème. La méthode classique est plus performante à la condition que la taille originale des entiers ne dépasse 128 bits. Par

contre elle est beaucoup moins performante lorsque la taille du problème est plus grande ; la multiplication de deux entiers de 2048 bits se fait 3.57 plus rapidement par la meilleure version de l'algorithme de Karatsuba.

Des implémentations pour comparer l'algorithme classique formulé par D. Knuth [Knu81], la MMM et l'algorithme de Barret [Bar87] ont été effectuées par A. Bosselaers et al. [BGV94]. L'algorithme classique semble être le meilleur choix pour une seule multiplication modulaire, la performance de la MMM étant très proche de celle de l'algorithme de Barret et même de l'algorithme classique. La MMM est pourtant la méthode la plus performante pour effectuer l'exponentiation modulaire. Différentes versions de la MMM ont été implémentées en langage *C* et en assembleur par Ç. Koç et al. [KAK96].

1.3 Notre approche

Tout au long de cette introduction, nous avons parlé de la multiplication des entiers ne dépassant pas 1024 bits. D'après le principe de Kerckhoffs, pour assurer plus de sécurité, des clés de grandes tailles sont de plus en plus exigées surtout avec la progression rapide de la technologie. Cela impose de chercher des algorithmes de multiplication capables de répondre aux besoins de calculs de cryptosystèmes dont les tailles de clés sont sans cesse croissantes.

L'algorithme classique est asymptotiquement le plus lent parmi les algorithmes de multiplication [BGV94, LHL03, Moe76], sa complexité est de $O(n^2)$. En d'autres termes, pour les entiers de grande taille, d'autres algorithmes plus sophistiqués sont plus rapides, et cette supériorité augmente lorsque n augmente.

L'algorithme de Horner s'exécute également en temps $O(n^2)$ et n'est pas un bon candidat lui non plus pour l'implémentation matérielle [JB97, BM04]. D'ordre $O(n^2)$, l'algorithme à la Russe est très simple à implémenter mais il n'y a aucune raison de le préférer à l'algorithme classique du côté performance [BB88]. Karatsuba est asymptotiquement plus performant que tous les algorithmes mentionnés précédemment pour la multiplication des entiers mais, en raison de sa

nature récursive, il n'est pas pratique dans une approche matérielle.

L'algorithme de Montgomery requiert $2n^2$ multiplications et $2n(n-1)+1$ additions pour effectuer la multiplication des entiers [NEG04]. Ultérieurement, il a été adapté pour la multiplication modulaire de manière à le rendre plus efficace. Néanmoins, la conversion entre la représentation de Montgomery et la représentation binaire classique et le pré-calcul de certains paramètres, qui sont nécessaires au fonctionnement de cet algorithme, prennent un temps si important que cela rend l'utilisation de cet algorithme avantageuse seulement dans les applications exigeant des multiplications répétées plusieurs fois tels que les cryptosystèmes à clés publiques. Sinon, la performance de l'algorithme de Montgomery est très proche de celle de l'algorithme classique [BGV94] comme l'indique sa complexité. C'est pour cette raison que l'algorithme de Montgomery est souvent utilisé pour effectuer l'exponentiation modulaire [BGV94].

D'autre part, le fait que l'algorithme de Montgomery consiste à réduire la MM et l'EM en une série d'additions modulaires (milliers), cela implique une énorme propagation de retenues lors de la manipulation des clés de 4096 et plus. Ceci augmente la complexité du circuit et réduit son efficacité.

La transformée de Fourier rapide (FFT pour **F**ast **F**ourier **T**ransform) est un algorithme qui permet d'effectuer la multiplication des entiers d'une façon très efficace, en temps de $O(n \log n)$ [BB88], mais qui devient intéressant surtout pour des entiers relativement grands[6]. La méthode classique prend un temps en $O(n^2)$ tandis que l'algorithme de Karatsuba requiert seulement un temps en $O(n^{1.59})$. La différence entre $O(n^2)$ et $O(n \log n)$ est beaucoup plus que celle entre $O(n^2)$ et $O(n^{1.59})$.

L'utilisation de la FFT fut constamment évitée dans le contexte de la multiplication des entiers à cause de l'impression qu'elle est toujours impraticable pour les problèmes calculatoires actuels notamment les cryptosystèmes modernes et les

[6]Notons que cette complexité ne s'applique que dans le cas où les opérations élémentaires portent sur des scalaires de ℓ bits, où ℓ est la taille d'un bloc (base=2^ℓ). Si on voulait s'astreindre à ne compter comme élémentaire que les opérations au niveau du bit, cette compléxité passerait à $O(n \log n \log\log n)$.

problèmes connexes. Plus précisément, la FFT n'offrirait pas l'efficacité théorique prétendue puisqu'elle est efficace seulement lorsque ces problèmes sont de taille suffisamment grande. C'est pour cela que la FFT était exclusivement utilisée pour calculer par exemple la valeur de π jusqu'à $4,194,293$ de chiffres [BB88, TK83] et même jusqu'à 51.5 milliards de chiffres [SG03].

Cette hypothèse est cependant basée sur des travaux qui datent de l'époque où les clés changeaient de 128 à 256 puis à 512 bits. Pourtant, la taille minimale d'une clé RSA est présentement de 1024 bits et celle recommandée est de 2048 bits. On commence même à parler de 4096 bits dans les prochaines années. Les résultats pratiques (Figure 1.2) ont montré que pour les opérandes de 1024 bits, le nombre de multiplications est le même pour FFT et la méthode classique. Le nombre d'additions est même plus grand d'ailleurs. Cependant, l'algorithme de la FFT offre déjà une meilleure performance pour les opérandes de 4096 bits et plus. La FFT requiert 512 multiplications à 129 bits tandis que la méthode classique nécessite 1024 à 128 bits. Ceci indique que cette méthode serait préférable pour les cryptosystèmes futurs où la taille principale atteindra 4096 bits ou plus.

		k = 256	k=1024	k=4096
Classique	Mult	64 x (32x32)	256 x (64x64)	1024 x (128x128)
	Add	56 (64 bits)	240 (128 bits)	992 (256 bits)
FFT	l	8	16	32
	n	32	64	128
	w	2	2	2
	m (bits)	33	65	129
	Mult	128 x (33x33)	256 x (65x65)	512 x (129x129)
	Papillon	576	1344	3072
	Add/Sub	1152 (33 bits)	2688 (65 bits)	6144 (129 bits)

FIG. 1.2 – Nombre d'opérations selon les paramètres

Cela est aussi conforme au travail de R. Moenck [Moe76] qui analyse et compare la performance des différentes méthodes pour la multiplication des polynômes telles que la méthode classique, la FFT dans un corps fini, l'algorithme de Karatsuba ainsi qu'une méthode hybride. Il conclut que la FFT ne commence à offrir la performance attendue qu'après un seuil. Nous voulons dire par seuil la borne mini-

male que les polynômes à multiplier doivent avoir pour que la FFT soir préférable, sinon la méthode classique est plus performante. Dans ce travail, la borne était 32. Pour les polynômes de borne 64 ou plus (seuil=64), la FFT est presque aussi performant que l'algortihme de Karatsuba le plus rapide et deux fois plus rapide que la méthode classique. Si on considère la multiplications des entiers de 4096 bits, représentés en base 16, on obtient 256 coefficients. Ainsi, l'utilisation de la FFT pour la multiplication des entiers de 4096 est déjà un choix judicieux. De ce fait, nous croyons que les algorithmes de multiplication basés sur la FFT commenceront bientôt à offrir une bonne efficacité dans les cryptosystèmes du proche futur.

La FFT est parmi les 10 meilleurs algorithmes du $XX^{\text{ème}}$ siècle et décrite comme *"The FFT-An Algorithm the Whole Family Can Use"* [Roc00]. Grâce au travail de J. Cooley et J. Tukey en 1965 [CT65], il est peut-être l'algorithme le plus utilisé actuellement dans l'analyse et le traitement de données numériques. La FFT est largement utilisée dans différentes branches scientifiques notamment l'analyse des systèmes linéaires, l'optique, la théorie des probabilités, la multiplication de grands entiers et même dans l'informatique quantique. Depuis, plusieurs architectures et implémentations de la FFT ont été proposées pour les ordinateurs personnels, les processeurs spécialisés, les circuits VLSI ainsi que pour les dispositifs reconfigurables.

L'idée de la conception d'un processeur FFT spécialisé a été introduite par B. Gold and T. Bially [GB73]. Un processeur FFT spécialisé augmente considérablement la performance. Différentes architectures FFT spécialisées ont été ensuite mises en application. Xilinx a mis en application un processeur FFT à nombre complexe sur un FPGA [FFT05]. Altera a également mis en application son propre FFT Megacore [FFT01]. L'algorithme ainsi que l'architecture à deux modules FFT pipelinés ont été proposés par A. El-Khashab et E. Swartzlander [EKS03]. Un nouveau processeur FFT à basse consommation d'énergie conçu pour les applications dans un LAN (**L**ocal **A**rea **N**etwork) sans fil est proposé par M. Hasan et al. [HAT03]. Pourtant, les travaux utilisant la FFT pour la multiplication de grands entiers sont quasi absents dans la littérature. Nous n'avons trouvé

que cinq travaux logiciels et un seul travail matériel très récent [CPA04] même après une recherche étendue.

W. Gentleman and G. Sande [GS66] pourraient être les premiers à avoir introduit l'utilisation de la FFT pour la multiplication des polynômes de très haute degré. Plusieurs techniques sont décrites par J. Hartwell [Har71] pour améliorer la performance de l'algorithme de Cooley-Tukey sur les petits ordinateurs.

C. Yap et C. Li [YL00] ont implémenté une librairie C++ gratuite dédiée à l'arithmétique modulaire. Aussi B. Haible a écrit une librairie C++ pour l'arithmétique à précision quelconque en utilisant l'algorithme de Schönhage et Strassen [SS71]. Gnu GMP et Gambit ont pour leur part implémenté respectivement un package C++ et Scheme pour le même but [CLN00, Gam87].

L'unique implémentation matérielle, que nous avons trouvée vers la fin de cette recherche, est un travail très récent qui vise à réaliser la multiplication des entiers ultra grands (12 millions de chiffres). On a rapporté une très haute performance, mais au prix d'une augmentation des ressources très coûteuse.

Pour la performance comme pour des raisons physiques de sécurité, on préfère souvent des réalisations matérielles des algorithmes cryptographiques. Les solutions traditionnelles d'ASIC (**A**pplication-**S**pecific **I**ntegrated **C**ircuit) ont l'inconvénient bien connu d'une flexibilité réduite, d'un temps de développement et d'un coût élevé de fabrication comparé à celui du logiciel. L'implémentation d'un ASIC requiert en moyenne trois mois de conception et un mois de test au coût de plusieurs milliers de dollars. Le prototype du circuit fonctionne dans environ 60% des cas seulement [LS89] même avec toutes les précautions prises pendant la phase de développement. D'autre part, si les spécifications d'un circuit logique changeaient ou si une erreur avait échappé à la simulation, le circuit serait complètement remplacé et il faudra par la suite tout refaire. Ceci est catastrophique du côté économique. Pour parer ces inconvénients, les dispositifs logiques reconfigurables ont été développés, notamment les FPGA. C'est une solution brillante qui combine presque la flexibilité d'un logiciel et la vitesse d'un ASIC.

Un FPGA, introduit en 1990, est un circuit logique manufacturé à l'usine en

tant que dispositif générique et plus tard programmé (ou reconfiguré) sur place pour une application spécifique sans aucune étape technologique supplémentaire. La reconfiguration de tels circuits signifie que de nouveaux circuits peuvent être facilement examinés et changés à plusieurs reprises sans risquer les coûts que demandent les ASIC. Ceci permet ainsi de mettre à jour ou même de remplacer les produits mis déjà en application sans remplacer les anciennes puces.

Tout comme les ASIC, la taille et la vitesse des FPGA augmentent constamment selon la loi de Moore, au point où ils sont maintenant capables de concurrencer les ASIC dans maintes applications. Un avantage fondamental d'un FPGA est qu'il est possible de prendre son code et en faire un ASIC. En fait, il semble que de plus en plus les concepteurs de circuits ASIC préfèrent passer par l'étape intermédiaire d'un FPGA ce qui est moins risqué économiquement. Puis, une fois que le modèle FPGA est au point, il est alors relativement aisé de le retranscrire dans une architecture de type ASIC. Il est même possible de réaliser une partie du code sur un ASIC et garder une autre sur un FPGA. Ceci semble idéal pour un cryptosystème dont l'algorithme est fixe mais les clés sont variables.

Dans ce mémoire, nous présentons la multiplication des grands nombres par la FFT à nombres entiers plutôt qu'à nombres complexes. Nous allons également réaliser une implémentation matérielle sur un FPGA.

Le reste de ce mémoire est organisé comme suit : le chapitre 2 survole les architectures et concepts mathématiques des différents algorithmes et techniques mentionnés dans cette introduction. Dans le chapitre 3, nous présentons en détails la transformée de Fourier discrète ainsi que sa version la plus rapide, la transformée de Fourier rapide. Nous y discutons surtout de la version en arithmétique modulaire utilisée pour la multiplication des entiers. Le chapitre 4 présente deux architectures pour la multiplication de grands entiers ainsi que leurs implémentations matérielles. Finalement, dans le chapitre 5, nous comparons notre travail à des travaux précédents, évaluons notre contribution et terminons par une conclusion.

CHAPITRE 2

ÉTAT DE L'ART

Nous introduisons dans ce chapitre les algorithmes de multiplication les plus utilisés dans la multiplication des grands entiers ainsi que les différentes technologies adoptées dans les implémentations matérielles.

2.1 Algorithmes de multiplication

La multiplication est un problème calculatoire fondamental d'une très grande importance pratique. En fait, l'ordinateur a été à l'origine construit pour automatiser les opérations arithmétiques dont la multiplication est une opération vitale. Avec l'avènement de la cryptographie à clé publique, les travaux théoriques et pratiques sur les algorithmes de multiplication ont beaucoup augmenté. Un grand intérêt a été mis en particulier sur les algorithmes de multiplication de grands entiers qui avaient été mis de côté depuis longtemps à cause de leur inefficacité pour les problèmes de l'époque.

Tout au long de ce travail, nous nous intéressons à la multiplication de grands entiers de k bits. Quand on parle de grands entiers, on parle souvent de k plus grand que 256 bits, un nombre qui équivaut donc à plusieurs mots mémoires. Ceci exige l'établissement d'une nouvelle structure de données pour manipuler ces grands nombres. Cette structure de données est construite comme suit. Nous divisons les nombres de k bits en n blocs de ℓ bits. Pour une implémentation efficace, ℓ est toujours choisi sous la forme de puissance de 2 qui ne dépasse pas la taille d'un mot mémoire. D'après cette structure, la valeur $\beta = 2^\ell$ dénotera alors la base de cette structure et tout entier a de k bits peut être écrit sous la forme polynomiale suivante :

$$a = (a_{n-1}a_{n-2}\dots a_0) = \sum_{i=0}^{n-1} a_i\beta^i$$

où les a_i sont des entiers naturels dans l'intervalle $[0, \beta - 1]$.

2.1.1 L'algorithme classique de multiplication

Soient a et b deux nombres de k bits, exprimés en base $\beta = 2^\ell$:

$$a = (a_{n-1}a_{n-2}\dots a_0) = \sum_{i=0}^{n-1} a_i\beta^i,$$

$$b = (b_{n-1}b_{n-2}\dots b_0) = \sum_{i=0}^{n-1} b_i\beta^i,$$

où les a_i et b_i sont dans l'intervalle $[0, \beta - 1]$. L'algorithme classique pour la multiplication de a et b consiste à calculer des produits partiels en multipliant les b_i du multiplieur b par le nombre a tout entier puis additionner ces produits partiels afin d'obtenir le produit final p qui est un nombre de $2k$ bits.

Notons par p_{ij} la paire (**R**etenue, **S**omme) obtenue du produit partiel a_ib_j. Par exemple, en système décimal $\beta = 10$. Si $a_i = 3$ et $b_j = 8$, alors $p_{ij} = (2, 4)$. La table 2.1 illustre les p_{ij} obtenus de la multiplication de a et b à 3 chiffres.

			a_2	a_1	a_0
$\times$			b_2	b_1	b_0
			p_{02}	p_{01}	p_{00}
		p_{12}	p_{11}	p_{10}	
$+$	p_{22}	p_{21}	p_{20}		
p_5	p_4	p_3	p_2	p_1	p_0

TAB. 2.1 – Multiplication classique

Le dernier rang dénote la somme totale des produits partiels qui est aussi le produit de a par b représenté par un nombre de $2k$ bits. L'essentiel de l'algorithme classique est qu'il effectue chiffre par chiffre les opérations de multiplications et d'additions illustrées ci-dessus. Le produit partiel p est initialisé à 0. Par la suite, chaque chiffre b_i de b se multiplie par a tout entier et s'ajoute au produit partiel p. À la fin du calcul, p aura le produit final ab. Cette méthode classique de multiplication est décrite par l'algorithme 1. L'opération principale de cet algorithme est à l'étape 7, communément connue par l'opération du produit interne.

Algorithme 1 Algorithme Classique de Multiplication

1: Entrée : a, b
2: Sortie : $p = ab$
3: Initialement $p_i := 0$ pour tout $i = 0, 1, \cdots, 2n-1$
4: **for** $i = 0$ to $n-1$ **do**
5: R :=0
6: **for** $j := 0$ to $n-1$ **do**
7: (R,S) :=$p_{i+j} + b_i a_j + R$
8: $p_{i+j} := S$
9: **end for**
10: $p_{i+n} := R$
11: **end for**
12: Retourner $(p_{2n-1} p_{2n-2} \cdots p_0)$

Cette opération consiste à multiplier deux nombres de k bits, ajouter ce produit à la retenue précédente et puis ajouter ce dernier résultat au produit partiel p_{i+j}. Comme ce produit se trouve dans la boucle la plus interne, il prendra la durée la plus longue de l'exécution de l'algorithme et doit ainsi être accompli le plus rapidement possible.

La complexité de l'algorithme classique est ainsi déterminée à partir de cette opération interne qui comporte deux boucles imbriquées allant de 0 à n. Ceci implique que le nombre total des produits internes est égal à n^2. Puisque $k = n \cdot \ell$ bits, ℓ est une constante, il exige ainsi $O(k^2)$ opérations (niveau bits) pour multiplier deux nombres de k bits.

Cet algorithme standard de multiplication est asymptotiquement le plus lent parmi les algorithmes naturels de multiplication. Nous verrons plus tard d'autres algorithmes plus sophistiqués et bien plus efficaces. Néanmoins, il est très aisé à implémenter et, pour des nombres de petites tailles (moins que 128 bits dans certaines implémentations), offre une meilleure performance sur les algorithmes cités [BB88]. Il est même souvent utilisé par ces algorithmes sophistiqués pour calculer les multiplications de petites tailles [BGV94, LHL03, Moe76].

2.1.2 L'algorithme de multiplication à la russe

Cet algorithme fonctionne selon la procédure suivante. On écrit le multiplieur et le multiplicande côte à côte. On fait deux colonnes, une sous chaque opérande, en répétant la règle suivante jusqu'à ce que le nombre sous le multiplieur soit 1 : on divise le nombre sous le multiplieur par 2, en ignorant toutes les fractions, et on double le nombre sous le multiplicande en l'additionnant à lui-même. Finalement, on élimine chaque rangée dans laquelle le nombre sous le multiplieur est pair, et on ajoute les nombres qui demeurent dans la colonne sous le multiplicande. Par exemple, la multiplication de 9 par 35 se fait comme indiqué dans la table 2.2. Notez que le produit $9 \cdot 35 = 315$ est bien la somme cumulée $9 + 18 + 288 = 315$.

Multiplieur	Multiplicande	Somme cumulée
35	9	9
17	18	18
8	36	–
4	72	–
2	144	–
1	288	288
		315

TAB. 2.2 – Exemple de multiplication à la russe

Bien que cet algorithme puisse sembler évident au début, c'est essentiellement la méthode employée dans beaucoup d'ordinateurs [BB88]. Il est très simple à implémenter et il n'y a aucun besoin de mémoriser aucune table de multiplication, comme il apparaît dans le pseudo-code algorithme (2). Tout ce que nous avons besoin de savoir est comment ajouter, doubler, ou diviser un nombre par deux, des opérations assez simples sur un ordinateur classique.

Le temps requis par la multiplication à la est d'ordre $O(k^2)$, à condition que nous choisissions l'opérande le plus petit comme le multiplieur et le plus grand comme le multiplicande, sinon c'est moins bon. Ainsi, il n'y a aucune raison de la préférer à l'algorithme classique [BB88].

Algorithme 2 Algorithme de Multiplication à la Russe

```
1: Entrée : a, b
2: Sortie : p = ab
3: if (a > b) then
4:    Echanger (a, b) {Mettre le multiplieur le plus petit}
5: end if
6: p := 0
7: while (a > 0) do
8:    if (a est impair) then
9:       p := p + b
10:   end if
11:   a := a/2        {simple décalage à droite}
12:   b := b * 2      {simple décalage à gauche}
13: end while
14: Retourner p
```

2.1.3 L'algorithme de Horner

L'algorithme ou la règle itérative de Horner fut introduite à l'origine pour évaluer efficacement la valeur d'un polynôme $p(x) = \sum_{i=0}^{n} a_i x^i$ pour une valeur α donnée. Il est basé sur la réécriture suivante :

$$p(x) = a_0 + a_1 x + \ldots + a_n x^n \tag{2.1}$$

$$= a_0 + x(a_1 + x(a_2 + \ldots + x(a_{n-1} + x(a_n)) \ldots)) \tag{2.2}$$

Le schéma de Horner (2.2) fait n multiplications et n additions pour calculer $p(\alpha)$ alors que $2n-1$ multiplications sont nécessaires pour une évaluation classique (2.1) puisqu'il faut :

- $n-1$ multiplications pour calculer successivement $\alpha^2, \ldots, \alpha^n$.
- n multiplications pour calculer $a_1\alpha, \ldots, a_n\alpha^n$

L'algorithme 3 permet d'évaluer un polynôme $p(x)$ en un point donné. Bien que développé pour l'évaluation des polynômes, l'algorithme de Horner peut également être employé pour la multiplication de longs entiers [JB97, BM04]. Nous savons qu'un nombre entier peut être écrit sous la forme d'un polynôme à évaluer dans la base de représentation, appelée parfois représentation positionnelle. Par exemple

Algorithme 3 Algorithme de Horner pour calculer $p(\alpha)$

1: Entrée : $\alpha, a_0, a_1, \cdots, a_{n-1}$
2: Sortie : $p(\alpha)$
3: $p_n := 0$
4: **for** $i = n-1$ to 0 **do**
5: $\quad p_i := a_i + \alpha p_{i+1}$
6: **end for**
7: Retourner p {p contient $p(\alpha)$ en sortie}

$325 = 3x^2+2x+5$ à $x = 10$. De même, $25 = 2x+5$ à $x = 10$. Afin de multiplier 325 par 25, nous multiplions tout d'abord les deux polynômes $(3x^2+2x+5)(2x+5) = 6x^3 + 19x^2 + 20x + 25$ puis évaluons le polynôme produit à la valeur 10, soit : $6(10)^3+19(10)^2+20(10)+25 = 8125$, pour obtenir en effet le produit $325 \cdot 25 = 8125$.

D'une façon générale, soient a et b deux entiers de k bits, la multiplication de a et b par l'algorithme de Horner est définie par la relation itérative ci-dessous. Afin de faciliter l'explication, nous allons utiliser cette fois la base binaire ($x = 2$), ce qui implique que $k = n$. D'après l'equation 2.2, on peut écrire

$$\begin{aligned} ab &= a \cdot \sum_{i=0}^{n-1} b_i 2^i, \\ &= ab_0 + 2(ab_1 + 2(ab_2 + \ldots + 2(ab_{n-2} + 2(ab_{n-1}))\ldots)) \end{aligned} \tag{2.3}$$

Cette dernière (2.3) peut être écrite sous la forme récursive suivante

$$p_0 = 0 \tag{2.4}$$

$$p_i = 2p_{i-1} + b_{i-1}a \tag{2.5}$$

qui est à la base de l'algorithme de Horner (Algorithme 4) pour la multiplication des entiers.

L'algorithme de Horner requiert $O(k^2)$ pour multiplier deux entiers de k bits, puisque dans l'équation 2.5 chaque coefficient du vecteur b doit être multiplié par chaque coefficient du vecteur a. Ceci est traduit par la ligne 5 dans l'algorithme. La multiplication des polynômes est beaucoup plus complexe que l'évaluation d'un

Algorithme 4 Algorithme de Multiplication de Horner

1: Entrée : $a_0, a_1, \cdots, a_{n-1}$ et $b_0, b_1, \cdots, b_{n-1}$
2: Sortie : $p = ab$
3: $p_0 := 0$
4: **for** $i = n - 1$ to 0 **do**
5: $\quad p_i := 2p_{i-1} + b_{i-1}a$
6: **end for**
7: Retourner p

polynôme. D'autre part, la nature itérative de l'algorithme de Horner n'en fait pas un bon candidat pour les implémentations matérielles.

2.1.4 L'algorithme de Karatsuba

L'algorithme de Karatsuba est un algorithme récursif introduit par le mathématicien russe Karatsuba en 1962. Un traitement détaillé se trouve dans [BB88, PB85, Koç94]. Cet algorithme exige $O(k^{\log_2 3})$ pour multiplier deux nombres de k bits, à comparer avec le temps en $O(k^2)$ requis par la méthode classique. La complexité est réduite grâce à la méthode *diviser-pour-régner* qui utilise moins de multiplications que dans l'algorithme classique.

Soient a et b deux entiers de k bits et soit $\ell = \lceil k/2 \rceil$. Karatsuba décompose initialement a et b en deux parties égales :

$$a = 2^{\ell} a_1 + a_0, \quad b = 2^{\ell} b_1 + b_0,$$

tels que a_1 est les ℓ bits de poids forts de a et a_0 est les ℓ bits de poids faibles de a. Notons que la valeur 2^{ℓ} constitue ainsi la base β de la représentation. Cet algorithme ramène la multiplication de a et b à la multiplication de leurs composantes a_0, a_1, b_0 et b_1 dont la taille est la moitié de la taille des entiers initiaux comme le montre

l'équation suivante :

$$\begin{aligned} p &= a \cdot b \\ &= (2^\ell a_1 + a_0)(2^\ell b_1 + b_0) \\ &= 2^{2\ell}(a_1 b_1) + 2^\ell(a_1 b_0 + a_0 b_1) + a_0 b_0 \\ &= 2^{2\ell} p_2 + 2^\ell p_1 + p_0 \end{aligned}$$

Cette formulation nous révèle que la multiplication de deux nombres de k bits exige 4 multiplications de $\ell = \frac{k}{2}$ bits. Ceci n'est pas une bonne nouvelle en ce qui concerne l'amélioration de l'algorithme classique puisqu'elle n'en est qu'une version récursive. Ceci est montré dans l'algorithme 5 que nous appelons **A**lgorithme **C**lassique **R**écursif de **M**ultiplication (ACRM).

Algorithme 5 Algorithme Classique Récursif de Multiplication (ACRM)

1: Entrée : a, b, k
2: Sortie : $p = a \cdot b$
3: **if** (k est petit) **then**
4: Retourner ACM(a,b) {Appliquer l'algorithme classique}
5: **end if**
6: $\ell := k/2$ {Diviser l'entier en deux parties égales}
7: $a_0 := a/2^\ell$
8: $a_1 := a \mod 2^\ell$
9: $b_0 := a/2^\ell$
10: $b_1 := a \mod 2^\ell$
11: $p_0 := \text{ACRM}(a_0, b_0)$
12: $p_2 := \text{ACRM}(a_1, b_1)$
13: $t_0 := \text{ACRM}(a_0, b_1)$
14: $t_1 := \text{ACRM}(a_1, b_0)$
15: $p_1 := t_0 + t_1$
16: Retourner $2^{2\ell} p_2 + 2^\ell p_1 + p_0 = \beta^2 p_2 + \beta p_1 + p_0$

Les quatre appels récursifs (lignes 11–14) déterminent la complexité de cet algorithme. Supposant que $T(k)$ est le nombre d'opérations nécessaires pour multiplier

deux nombres de k bits par cet algorithme, alors

$$T(k) = 4T(\frac{k}{2}) + \alpha k \tag{2.6}$$

La valeur αk est le temps exigé pour effectuer les opérations d'addition, de soustraction et de décalage. Partons de la condition $T(1) = 1$, la solution de cette recurrence indique que l'exécution de cet algorithme requiert $O(k^2)$ opérations [Koç94].

L'idée de Karatsuba sert à améliorer la performance de cet algorithme en réduisant le nombre de multiplication à trois plutôt que quatre au prix d'additions supplémentaires. Ceci reste très avantageux puisque l'addition est beaucoup plus rapide surtout avec les grands entiers. L'algorithme de Karatsuba est essentiellement basé sur l'observation suivante. En réarrangeant les termes du produit $p = ab$, nous obtenons :

$$\begin{aligned} p_0 &= a_0 \cdot b_0 \\ p_2 &= a_1 \cdot b_1 \\ p_1 &= (a_0 + a_1) \cdot (b_0 + b_1) - p_0 - p_2 \end{aligned}$$

Trois multiplications, deux à n bits et une à $n + 1$ bits, sont ainsi suffisantes dans la formulation ci-haut pour multiplier deux nombres de k bits, plutôt que quatre. Cette observation est la base de l'**A**lgorithme de **M**ultiplication de **K**aratsuba (AMK) dont le pseudo code est montré dans l'algorithme 6. La complexité de l'algorithme de Karatsuba est donnée par la relation suivante

$$T(k) = 2T(\frac{k}{2}) + T(\frac{k}{2} + 1) + \alpha k \approx 3T(\frac{k}{2}) + \alpha k. \tag{2.7}$$

La valeur αk est le temps exigé pour effectuer les opérations d'addition, de soustraction et de décalage. Supposons que $T(1) = 1$, nous pouvons résoudre la récursivité 2.7 pour obtenir que l'algorithme de Karatsuba requiert

$$O(k^{\log_2 3}) \approx O(k^{1.59})$$

Algorithme 6 Algorithme de Multiplication de Karatsuba (AMK)

1: Entrée : a, b, k
2: Sortie : $p = a \cdot b$
3: **if** (k est petit) **then**
4: Retourner ACM(a, b) {Appeler l'algorithme classique}
5: **end if**
6: $\ell := k/2$
7: $a_0 := a/2^\ell$
8: $a_1 := a \mod 2^\ell$
9: $b_0 := a/2^\ell$
10: $b_1 := a \mod 2^\ell$
11: $p_0 :=$ AMK(a_0, b_0)
12: $p_2 :=$ AMK(a_1, b_1)
13: $temp :=$ AMK$(a_0 + a_1, b_0 + b_1)$
14: $p_1 := temp - p_0 - p_2$
15: Retourner $2^{2\ell}p_2 + 2^\ell p_1 + p_0 = \beta^2 p_2 + \beta p_1 + p_0$

opérations en bits [Koç94]. Karatsuba est en conséquence asymptotiquement plus performant que l'algorithme classique. Pourtant, ce dernier est plus performant pour les problèmes de petites tailles, moins de 128 bits par exemple [LHL03]. Pour les problèmes au-dessus de cette taille, il est toujours possible d'arrêter la récursivité à n'importe quel point et recourir à la méthode classique (ligne 3). On parle ainsi d'un algorithme hybride. Par exemple, pour multiplier deux entiers à 1024 bits, on peut appliquer plusieurs niveaux la récursivité de Karatsuba jusqu'à ce qu'on obtienne des entiers de petite taille puis calculer les trois autres multiplications par la méthode classique. En général, on applique Karatsuba jusqu'à ce que la taille des nombres arrive à un seuil sous lequel il n'est plus efficace. Notons que le seuil diffère d'une implémentation à une autre et surtout entre une implémentation logicielle et matérielle. Ce seuil est de 128 bits dans [LHL03] tandis qu'il est de 256 bits dans d'autres implémentations. Tout comme l'algorithme de Horner, en raison de sa nature récursive, il n'est pas pratique dans l'approche matérielle.

2.1.5 La transformée de Fourier rapide (FFT)

Tout d'abord, nous avons besoin de définir la transformée de Fourier discrète (DFT pour **D**iscrete **F**ourier **T**ransform).

La **DFT** d'un vecteur de n composants $a = \{a_0, a_1, \cdots, a_{n-1}\}$ par rapport à une constante ω, notée $DFT(a)$, est un nouveau vecteur $A = \{A_0, A_1, \cdots, A_{n-1}\}$ tel que :

$$A_j = \sum_{k=0}^{n-1} a_k \omega^{jk},\ 0 \leq j \leq n-1 \tag{2.8}$$

où $\omega = e^{i\frac{2\pi}{n}}$, appelée *la racine $n^{ième}$ primitive de l'unité*, et les valeurs ω^k pour $k = 0, \ldots, n-1$ sont appelées *les racines $n^{ièmes}$ de l'unité.*

Pensant au vecteur a en tant que représentation par coefficients d'un polynôme $p(x) = \sum_{i=0}^{n-1} a_i x^i$, le calcul de $DFT(a)$ est équivalent à évaluer $p(x)$ aux points $x_i = \omega^i$ pour $0 \leq i \leq n-1$, plus précisément le vecteur

$$A = DFT(a) = \{p(1), p(\omega), p(\omega^2) \cdots, p(\omega^{n-1})\}$$

Pour calculer un seul A_i en utilisant l'équation 2.8, il nous faut n multiplications et $n-1$ additions. Alors, pour calculer tous les A_i un à la fois, pour $i = 0 \ldots n-1$, ceci requiert n^2 multiplications et $n(n-1)$ additions. Il semble, à première vue, que le calcul de la DFT s'exécute en $O(n^2)$. Cependant, ce temps peut être ramené à $O(n \lg n)$ grâce à la transformée de Fourier rapide (FFT).

La FFT est un algorithme qui permet de réduire le calcul de la DFT de $O(n^2)$ à $O(n \lg n)$ en se basant sur la *stratégie diviser-pour-régner* et tire profit des propriétés particulières des racines $n^{ièmes}$ de l'unité. C'est une méthode qui peut servir à réaliser avec efficacité la multiplication des polynômes comme le montre la figure 2.1. Il s'agit d'évaluer les deux polynômes aux racines $n^{ièmes}$ de l'unité en appliquant à chacun d'eux la FFT, multiplier ces valeurs terme à terme et finalement appliquer la FFT inverse à ces valeurs finales pour obtenir le polynôme produit (interpolation). Notons que la FFT inverse utilise le même algorithme que celui de la FFT mais avec des paramètres différents. L'évaluation de deux polynômes prend

un temps en $O(n \log n)$ grâce à la FFT, la multiplication point à point nécessite $O(n)$ et l'interpolation s'exécute en $O(n \log n)$ également.

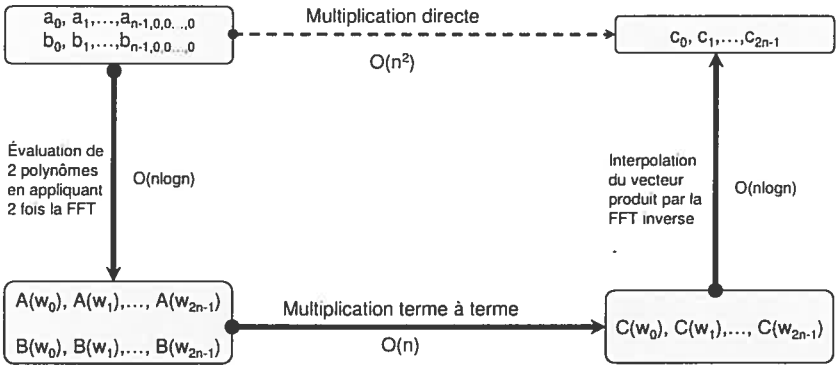

FIG. 2.1 – Schéma de la multiplication de polynômes par la FFT

Développée à l'origine pour le problème de la convolution, qui revient à la multiplication des polynômes, la FFT est utilisée maintenant dans maints domaines y compris la multiplication de grands entiers. Les entiers peuvent être représentés par la représentation positionnelle équivalente aux polynômes à évaluer en un point donné ; par exemple $325 = 3x^2+2x+5$ en $x = 10$. De même, $25 = 2x+5$ en $x = 10$. Afin de multiplier 325 par 25, nous multiplions tout d'abord les deux polynômes

$$(3x^2 + 2x + 5)(2x + 5) = 6x^3 + 19x^2 + 20x + 25$$

puis évaluons le polynôme produit à la base utilisée (10 dans notre cas),

$$6(10)^3 + 19(10)^2 + 20(10) + 25 = 8125,$$

pour obtenir finalement le produit $325 \cdot 25 = 8125$. Par conséquent, si nous pouvons multiplier des polynômes rapidement, nous pourrons multiplier de grands entiers rapidement. Une fois que l'algorithme de la FFT est réalisé, on peut ensuite multiplier des polynômes ou des entiers selon la procédure suivante.

Procédure de la multiplication de deux entiers par la FFT

Soient $A(x) = \sum_{i=0}^{n-1} a_i\beta^i$ et $B(x) = \sum_{i=0}^{n-1} b_i\beta^i$ la représentation polynomiale en base β de deux entiers a et b, $0 \leq a_i, b_i \leq \beta - 1$. Soient $t = 2n$ une puissance de 2 et ω la racine $t^{ième}$ primitive de l'unité. Alors, la procédure suivante permet de calculer le produit de A et B ainsi que celui de a et b :

1. Doublage de borne : on représente $A(x)$ et $B(x)$ par leurs coefficients sous la forme des polynômes de borne t en ajoutant à chacun n coefficients nuls dans les rangs supérieurs :

$$\begin{aligned} A(x) &= (a_0, \cdots, a_{n-1}, 0, 0, \cdots, 0) \\ B(x) &= (b_0, \cdots, b_{n-1}, 0, 0, \cdots, 0) \end{aligned}$$

2. Représentation par valeurs : on effectue les représentation par valeurs de $A(x)$ et $B(x)$ en appliquant à chacun d'eux la FFT d'ordre t. On obtiendra t couples (point-valeur) représentant les valeurs de deux polynômes aux abscisses racines $t^{ièmes}$ de l'unité.

3. Multiplication point à point : la représentation par valeurs du polynôme $C(x) = A(x)B(x)$ s'obtient en multipliant terme à terme $A(x)$ et $B(x)$. Cette représentation n'est que l'évaluation de $C(x)$ aux abscisses racines $t^{ièmes}$ de l'unité :

$$C(x) = \{A(\omega^0)B(\omega^0), A(\omega^1)B(\omega^1), \cdots, A(\omega^{t-1})B(\omega^{t-1})\}$$

4. Interpolation : on récupère la représentation par coefficients du polynôme $C(x)$ en appliquant une seule fois la FFT^{-1} sur t coordonnées.

5. Évaluation en β : cette étape, qui concerne uniquement la multiplication des entiers, consiste à évaluer le polynôme produit c en base β afin d'obtenir le produit ab.

Les étapes 1 et 3 nécessitent $O(n)$ opérations et les étapes 2 et 4 sont réalisées en $O(n \log n)$. Ainsi, la multiplication de deux polynômes de borne n pourrait se faire en $O(n \log n)$. Cette complexité est vraie aussi pour la multiplication des grands entiers à condition que les opérations élémentaires portent sur des scalaires de ℓ bits. Néanmoins, la FFT n'est efficace que pour les problèmes de grande taille. La taille demandée dans FFT est même plus grande que celle dans Karatsuba.

2.1.6 La méthode de multiplication de Montgomery (MMM)

En 1985, P. Montgomery a introduit une méthode efficace pour la multiplication modulaire par l'intermédiaire d'une transformation des entiers en un système de congruence (RNS). En conséquence, il faut penser en termes des classes modulo m ou bien des entiers représentants.

L'algorithme ou la Méthode de Multiplication de Montgomery (MMM) effectue la multiplication modulaire sans diviser par le modulo (m). Cet algorithme remplace la division par m, l'opération le plus coûteuse dans la reduction modulaire, par une division de puissance de 2 qui est implémentée tout simplement par des décalages à droite. Ceci est pourtant au prix d'une transformation des entiers au RNS avant l'opération et d'une re-transformation du résultat au système original après l'opération.

À l'origine, c'était un algorithme pour la multiplication des entiers. Néanmoins, la complexité totale était de $2n^2$ multiplications et de $2n(n-1)+1$ additions [NEG04]. Ultérieurement, il a été adapté pour la multiplication modulaire dans les corps $\mathbf{F}_{p^n}$. Dans ce qui suit, nous récapitulons l'idée fondamentale derrière l'algorithme de réduction de Montgomery en introduisant la représentation dans le RNS de Montgomery.

Représentation de Montgomery

Soit m un modulo de k bits, alors $2^{k-1} \leq m < 2^k$. On choisit aussi un entier r tel que $r = 2^k > m$. En outre, m et r doivent être premiers entre eux, $gcd(r, m) = 1$, pour

qu'ils puissent avoir des inverses (r^{-1}, m^{-1}) uniques dans $\mathbf{Z_m}$. Cette condition, qui est nécessaire pour que la MMM fonctionne correctement, est satisfaite pour tout m impair puisque r est une puissance de 2. Nous soulignons qu'il suffit uniquement que r et m soient premiers entre eux. Cette forme particulière de r, puissance de deux, est toujours adoptée pour des fins d'efficacité.

Étant donné un entier $0 \leq a < m$, on définie sa représentation de Montgomery par la formule

$$\bar{a} = a \cdot r \mod m$$

Cette représentation de a dans le nouveau domaine $(\bar{a})$est appelée le m-résidu de a par rapport à r. On l'appelle parfois la classe modulo m ou le représentant de a. Ainsi, l'ensemble $\{i \cdot r \mod m | 0 \leq i < m\}$, appelé système de congruence (RNS) de Montgomery, contient toutes les classes représentatives modulo m (ou tous les représentants) entre 0 et $m-1$ qui représentent à leur tour tous les entiers du domaine original qui sont également entre 0 et $m-1$. En fait, il existe un isomorphisme entre le domaine original et le domaine du RNS. Par exemple, prenons $m = 5$ et $r = 8$. On a alors $r^{-1} = 2$. Les valeurs de l'ensemble $\{i \cdot r \mod m | 0 \leq i < m\}$ sont donc $0, 3, 1, 4, 2$.

Montgomery a profité de cette propriété (isomorphisme) en introduisant une routine de multiplication très rapide appelée réduction ou produit de Montgomery.

Réduction de Montgomery

La **R**éduction ou le produit de **M**ontgomery (RM) est la transformation inverse de la représentation de Montgomery dont la définition est donnée par la formule

$$RM(u) = u \cdot r^{-1} \mod m \tag{2.9}$$

où u est un m-residue et r^{-1} est l'unique inverse de r dans $\mathbf{Z_m}$, puisque m et r sont premiers entre eux. Nous avons utilisés la notation u plutôt que $\bar{u}$ pour faciliter l'explication dans la section qui vient juste après. On peut donc toujours penser à un entier a tel que $u = \bar{a}$. Cette formule, qui sera décrit sous la forme d'une routine

dans la section 2.1.6, permet de calculer efficacement le m-résidu du produit de deux entiers dont les m-résidus sont donnés. La représentation dans le domaine originale s'obtient ainsi en multipliant le m-résidu (le résultat dans le nouveau domaine) par r^{-1}.

Principe de réduction de Montgomery

La réduction de Montgomery est l'idée de base qui permet de remplacer la division modulo m par une réduction modulo r. Montrons donc comment calculer $RM(u)$ sans réduction modulo m.

- On suppose que $0 \leq u < mr$ et on cherche une valeur v telle que $u + vm$ soit un multiple de r, on obtient ainsi

$$(u + vm)/r \equiv u \cdot r^{-1} \mod m$$

  puisque $vm \equiv 0 \mod m$.

- D'autre part, $u + vm \equiv 0 \mod r$ puisque $u + vm$ est un multiple de r, ce qui implique que

$$v \equiv u \cdot (-m^{-1}) \mod r$$

  On obtient par la suite que $vm < mr$. Notons que la nouvelle valeur introduite $(1 \leq m^{-1} < m)$ est un pré-requis dans l'algorithme de Montgomery. Il s'agit de l'inverse unique de m dans $\mathbf{Z_m}$ $(m^{-1}m \equiv 1 \mod m)$, puisque m et r sont premiers entre eux.

- On déduit que

$$(u + vm)/r < (mr + mr)/r = 2m$$

  Finalement,

$$RM(u) = \begin{cases} (u + vm)/r & \text{si } (u + vm)/r < m \\ (u + vm)/r - m & \text{autrement} \end{cases}$$

Voilà le principe de réduction de Montgomery qui permet de contourner la division par m par une méthode plus rapide, soit en divisant par une puissance de deux (r). Ceci est la base de la méthode de multiplication de Montgomery.

Algorithme de réduction de Montgomery (RM)

La réduction de Montgomery de deux m-résidus $\bar{a}$ et $\bar{b}$ est défini par :

$$\bar{c} = RM(\bar{a}, \bar{b}) = \bar{a} \cdot \bar{b} \cdot r^{-1} \mod m \tag{2.10}$$

L'entier $\bar{c}$ est en effet le m-résidu du produit $c = a \cdot b$ puisque

$$\begin{aligned} \bar{c} &= RM(\bar{a}, \bar{b}) \\ &= \bar{a} \cdot \bar{b} \cdot r^{-1} \mod m \\ &= (a \cdot r) \cdot (b \cdot r) \cdot r^{-1} \mod m \\ &= a \cdot b \cdot r \mod m \\ &= c \cdot r \mod m \end{aligned}$$

Le produit c s'obtient simplement en multipliant $\bar{c}$ par r^{-1} :

$$c = a \cdot b = \bar{c} \cdot r^{-1} \mod m \tag{2.11}$$

La fonction RM($\bar{a}, \bar{b}$) est illustré dans l'algorithme 7.

Algorithme 7 Réduction de Montgomery (RM)

1: Entrée : $\bar{a}, \bar{b}$,
2: Sortie : $\bar{c} = \bar{a} \cdot \bar{b} \mod m$ {sans diviser par m }
3: $u := \bar{a} \cdot \bar{b}$
4: $v := u \cdot (-m^{-1}) \mod r$
5: $\bar{c} := (u + v \cdot m)/r$
6: **if** $\bar{c} \geq m$ **then**
7: $\quad \bar{c} = \bar{c} - m$
8: **end if**
9: Retourner $\bar{c}$

Le profit le plus important de la RM est que les opérations impliquées, 4 et 5, sont des multiplications modulo r et des divisions par r, les deux opérations sont intrinsèquement rapides puisque r est une puissance 2.

Nous avons maintenant à notre disposition la fonction RM qui calcule la multiplication modulaire dans la représentation de Montgomery. Pourtant, nous avons besoin d'une méthode qui ramène ce calcul au domaine d'entiers original. La méthode qui effectue la multiplication des entiers à l'aide de la réduction (ou routine de multiplication modulaire) de Montgomery (RM), est appelée méthode multiplication (modulaire) de Montgomery (MMM) dont le pseudocode est présente dans l'algorithme 8.

Algorithme 8 Méthode de Multiplication de Montgomery (MMM)

1: Entrée : a, b, m {m est impair}
2: Sortie : $c = a \cdot b$ {Le produit de a et b dans $\mathbf{Z}$}
3: $\bar{a} = a \cdot r \mod m$
4: $\bar{b} = b \cdot r \mod m$
5: $\bar{c} := RM(\bar{a}, \bar{b})$
6: $c := RM(\bar{c}, 1)$
7: Retourner c

En remarquant que $RM(\bar{a}, b) = a \cdot b \mod$, cet algorithme peut ainsi avoir une version plus simple et plus efficace comme il apparaît dans l'algorithme 9.

Algorithme 9 Méthode de Multiplication de Montgomery (optimisée)

1: Entrée : a, b, m {m est impair}
2: Sortie : $c = a \cdot b$ {Le produit de a et b}
3: $\bar{a} = a \cdot r \mod m$
4: $c := RM(\bar{a}, b)$
5: Retourner c

Nous citons encore les deux propriétés suivantes qui profitent du constante pré-calculé r^2 ainsi que de la fonction RM pour la transformation entre les deux

domaines :

$$\begin{aligned} \bar{a} &= RM(a, r^2) \\ &= ar^2 r \mod m \\ &= ar \mod m \\ a &= RM(\bar{a}, 1) \\ &= arr^{-1} \mod m \end{aligned}$$

Nous avons présenté une version de haut niveau de la RM pour faciliter l'explication de son fonctionnement. Pourtant, elle est toujours implémentée en matériel, ce qui vaut la peine de présenter sa version générique en base b pour la multiplication deux entiers x et y dont le pseudocode est illustré dans l'algorithme 10.

Algorithme 10 Méthode de Multiplication de Montgomery (binaire)

1: Entrée : $m = (m_{n-1} \cdots m_0)_b, x = (x_{n-1} \cdots x_0)_b, y = (y_{n-1} \cdots y_0)_b$, avec $0 < x, y < m$, $r = b^n$, $gcd(m, b) = 1$ et $m' = (-m_0^{-1}) \mod b$
2: Sortie : $xyr^{-1} \mod m$
3: $p := 0$
4: **for** $i = 0$ to $k - 1$ **do**
5: $\quad t_i := (p_0 + x_i y_0) m' \mod b$
6: $\quad p := (p + x_i y + t_i m)/b$
7: **end for**
8: **if** $(p \geq m)$ **then**
9: $\quad p := p - m$
10: **end if**
11: Retourner p {Le produit de x, y}

L'exemple présenté dans la table 2.3, repris de [MaSV97], illustre la simulation de la fonction RM pour $m = 72\,639, b = 10, r = 10^5, \bar{x} = 5\,792$ et $\bar{y} = 1\,229$ avec $m' = (-m_0^{-1}) \mod b$. Alors,

- $m_0 = 9, m_0^{-1} \mod b = 9$ et $(-m_0^{-1}) \mod b = 1$
- $r^{-1} \mod m = 33\,589$
- $\bar{p} = \bar{x}\bar{y}r^{-1} \mod m = 39\,796$

i	x_i	$x_i y_0$	t_i	x_i y	$t_i m$	p
0	2	18	8	2 458	581 112	58 357
1	9	81	8	11 061	581 112	65 053
2	7	63	6	8 603	435 834	50 949
3	5	45	4	6 145	290 556	34 765
4	0	0	5	0	363 195	39 796

TAB. 2.3 – Exemple de multiplication de Montgomery

La conversion entre la représentation de Montgomery et la représentation binaire classique est nécessaire pour cet algorithme. Outre cette opération coûteuse, les opérations de pré-calcul, surtout celui de m' et r^{-1}, prennent également du temps. Ceci rend l'utilisation de la MMM avantageuse seulement dans les applications exigeant des multiplications répétées plusieurs fois, notamment l'exponentiation dans le RSA. Sinon, la méthode classique est préférable.

2.2 Représentation de données

La représentation binaire traditionnelle a plusieurs inconvénients. Considérons par exemple la somme de deux entiers de n bits, $x = \sum_{i=0}^{n-1} x_i \cdot 2^i$ et $y = \sum_{i=0}^{n-1} y_i \cdot 2^i$. Nous commençons par calculer $x_0 + y_0$ pour obtenir la somme partielle s_0 et une retenue c_1. Nous additionnons ensuite les bits x_1,y_1 et c_1. Ainsi, bien que tous les bits de x et y soient souvent disponibles simultanément, il faut attendre le calcul des c_i avant de traiter $x_i + y_i$. L'opération s'effectue sériellement, le parallélisme est ainsi impossible dû à la propagation de retenues, et le temps de calcul est proportionnel à la taille des opérandes.

Un autre inconvénient dans la représentation traditionnelle est la manipulation des entiers dépassant plusieurs mot mémoires. Ces deux constatations suggèrent la conception des nouvelles techniques et structures de données permettant de réaliser le calcul plus rapidement.

2.2.1 Systèmes de congruences (RNS)

On dit que deux entiers a et b sont *congrus* modulo m ($m > 1$) si m divise $(a-b)$ et on écrit $a \equiv b \mod m$, notation introduite par Gauss. C'est donc une autre façon de parler de la divisibilité. Par exemple, $x \equiv 1 \mod 2$ signifie que 2 divise $x - 1$ ou que le reste de la division de x par 2 est 1. Ainsi, la solution de cette équation est l'ensemble des nombres impairs $x = 2k + 1$.

Un système de congruence (RNS pour **R**esidue **N**umber **S**ystem) est défini par un ensemble des modulos, $m = \{m_1, m_2, \ldots, m_k\}$, tels que m est la base du système et les m_i sont premiers entre eux[1] deux à deux. Le produit $M = \prod_{i=1}^{k} m_i$ définit le domaine de définition de ce système, $[0, M-1]$. Nous soulignons que si les entiers négatifs sont permis, ce domaine sera $[-M/2, M/2-1]$.

Tout entier x plus petit que M est représenté par un ensemble ordonné de k résidus, $\{x_1, x_2, \ldots, x_k\}$, tel que $x_i = x \mod m_i$. Le théorème du reste chinois (CRT pour **C**hinese **R**emainder **T**heorem) garantit l'unicité de cette représentation en base m à condition que x soit dans $[0, M-1]$ et les m_i soient premiers deux à deux.

Un système de congruences pourrait être décrit comme suit :

$$\begin{cases} x & \equiv x_1 \ (mod\ m_1) \\ x & \equiv x_2 \ (mod\ m_2) \\ & \vdots \\ x & \equiv x_k \ (mod\ m_r) \end{cases} \tag{2.12}$$

Les opérations arithmétiques sur $[0, M-1]$ sont ainsi définies de la façon suivante :

$$(x \pm y)_m = \{(x_1 \pm y_1)_{m_1}, (x_2 \pm y_2)_{m_2}, \cdots, (x_k \pm y_k)_{m_k}\} \tag{2.13}$$

$$(x \times y)_m = \{(x_1 \times y_1)_{m_1}, (x_2 \times y_2)_{m_2}, \cdots, (x_k \times y_k)_{m_k}\} \tag{2.14}$$

Ces équations illustrent clairement la nature parallèle et sans propagation de re-

[1] Deux nombres sont premiers entre eux si leur diviseur commun est 1.

tenues de l'arithmétique dans le RNS. Le RNS ramène l'arithmétique de grands nombres x, y à l'arithmétique des ses composants x_i, y_i qui est beaucoup plus simple et plus efficace. Les opérations s'exécutent indépendamment en un temps constant sur une architecture parallèle. Ainsi, le RNS, contrairement à la représentation de base usuelle, ne souffre pas de la propagation de retenue.

Pourtant, les opérations de la comparaison, de la détection de signe et du dépassement de capacité se font en utilisant une opération appelée l'extension de base, c'est l'opération qui prend la majeure partie de la durée de traitement dans le RNS.

Il reste à trouver une solution, si elle existe, à l'équation 2.12. Ceci est équivalent à récupérer x à partir de ses résidus. Le CRT est réputé être le moyen le plus efficace pour résoudre ce système.

2.2.2 Théorème du reste chinois (CRT)

Le théorème du reste chinois fait essentiellement correspondre un système d'équations modulo un ensemble d'entiers premiers entre eux deux à deux (Équation 2.12) à une équation modulo leur produit.

Étant donné $m = \prod_{i=1}^{k} m_i$, où les facteurs m_i sont premiers entre eux deux à deux, Le CRT possède deux intérêts fondamentaux. Tout d'abord, il est un théorème de structure. Autrement dit, il décrit la structure $\mathbf{Z}_m$ comme étant identique à celle du produit $\mathbf{Z}_{m_1} \times \mathbf{Z}_{m_2} \times \cdots \times \mathbf{Z}_{m_k}$ en associant l'addition et la multiplication modulo m_i au $i^{\text{ième}}$ composant du produit.

D'autre part, cette description rend le calcul beaucoup plus efficace, en termes d'opérations sur les bits, en transformant le travail dans chacun des systèmes $\mathbf{Z}_{m_i}$. Ceci permet de définir également des algorithmes efficaces grâce au parallélisme hérité. Notamment, un système RSA utilisant le CRT est trois fois et demi [QC82, Tak98, BOPV03, Gro00] plus rapide voire quatre fois [Kob94]. Pour cela le CRT est presque omniprésent dans les implémentations de RSA [BDL97, JLQ99, UBF+03, BOS03, Wag04].

Théorème 2.2.2.1 (Théorème du Reste Chinois). *Soient* $m = \prod_{i=1}^{k} m_i$, *où les* m_i *sont premiers deux à deux. On considère l'application*

$$x \iff (x_1, x_2, \cdots, x_k), \tag{2.15}$$

où $x \in \mathbf{Z}_m$, $x_i \in \mathbf{Z}_{m_i}$, *et* $x_i = x \pmod{m_i}$ *pour* $i = 1, 2, \cdots, k$. *Alors, l'application 2.15 est un isomorphisme d'anneaux entre* $\mathbf{Z}_m$ *et* $\mathbf{Z}_{m_1} \times \mathbf{Z}_{m_2} \times \cdots \times \mathbf{Z}_{m_k}$.

Les opérations sur les éléments de $\mathbf{Z}_m$ peuvent ainsi être effectuées de manière équivalente sur les $k-$tuples correspondant en les appliquant indépendemment et parallèlement à chaque élément dans le système $\mathbf{Z}_{m_i}$ approprié. Autrement dit, si $x \iff (x_1, x_2, \cdots, x_k)$ et $y \iff (y_1, y_2, \cdots, y_k)$, alors les équations 2.13 et 2.14 sont valides. Il nous teste à trouver la solution de ce système, ce qui est le but du corollaire suivant.

Corollaire 2.2.2.1.1 (Résoudre un système de congruences). *Soient* $m = \prod_{i=1}^{k} m_i$, *où les* m_i *sont premiers deux à deux. Alors, le système 2.12 a une solution unique*

$$x = \sum_{i=1}^{k} x_i M_i (M_i^{-1})_{m_i}, \tag{2.16}$$

où $M_i = \frac{m}{m_i}$ *et* $(M_i^{-1})_{m_i}$ *est l'inverse de* M_i *modulo* m_i.

Exemple : À titre d'exemple d'application du théorème du reste chinois, considérons le système de congruences suivant

$$\begin{cases} x & \equiv 2 \ (mod\ 5) \\ x & \equiv 3 \ (mod\ 13) \end{cases}$$

En respectant toujours la même notation que dans le corollaire, nous obtenons que $x_1 = 2, m_1 = 5, x_2 = 3, m_2 = 13$ et $m = 65$. Notre problème est le calcul de x mod 65. Nous commençons par calculer les M_i et leurs inverses :
$M_1 = 13, M_1^{-1} \equiv 2(mod\ 5)$ et $M_1 M_1^{-1} = 26$

$M_2 = 5, M_2^{-1} \equiv 8(mod\ 13)$ et $M_2 M_2^{-1} = 40$ Alors,

$$\begin{cases} x & \equiv 2 \cdot 26 + 3 \cdot 40 \quad (mod\ 65) \\ & \equiv 52 + 120 \quad (mod\ 65) \\ & \equiv 42 \quad (mod\ 65) \end{cases}$$

2.2.3 Les systèmes de numération redondants (RNR)

Les systèmes de numération redondants (RNR pour **R**edundant **N**umber **R**epresentation) permettent de supprimer la propagation de retenue intervenant lors de l'addition, réduisant ainsi le chemin critique. Montrons ceci par un exemple en supposant les nombres codés dans une base dix particulière telle que

$$x = \sum_{i=0}^{n-1} x_i 10^i,\ x_i \in D = \{-9, -8, \ldots, 8, 9\} \tag{2.17}$$

Certains nombres possèdent par la suite plusieurs représentations. Si nous travaillons avec des nombres d'un seul chiffre nous pouvons écrire

$$\begin{aligned} 1 &= 1 \cdot 10^0 = 1 \cdot 10^1 + (-9) \cdot 10^0 \\ 2 &= 2 \cdot 10^0 = 1 \cdot 10^1 + (-8) \cdot 10^0 \\ 3 &= 3 \cdot 10^0 = 1 \cdot 10^1 + (-7) \cdot 10^0 \\ 4 &= 4 \cdot 10^0 = 1 \cdot 10^1 + (-6) \cdot 10^0 \\ 5 &= 5 \cdot 10^0 = 1 \cdot 10^1 + (-5) \cdot 10^0 \end{aligned}$$

Afin d'éviter toute confusion entre le signe d'un chiffre et l'opérateur de soustraction, nous représentons les chiffres négatifs par leur valeur absolue surmontée d'une barre. Choisissons $x = 7\,447$ et $y = 2\,673$ et additionnons-les en commençant par le chiffre de poids fort comme décrit dans la table 2.4. L'astuce consiste à exprimer $7 \cdot 10^3 + 2 \cdot 10^3$ par $1 \cdot 10^4 + \bar{1} \cdot 10^3$. Si une retenue provient de la colonne des centaines, elle sera ainsi annulée par $\bar{1} \cdot 10^3$ et ne pourra pas se propager. Le même principe régit le traitement des autres colonnes. Dans la colonne de centaines $4 \cdot 10^2 + 6 \cdot 10^2$

s'exprime par $1 \cdot 10^3 + 0 \cdot 10^2$ et ainsi de suite.

	7	4	4	7		7	4	4	7		7	4	4	7		7	4	4	7
+	2	6	7	5	+	2	6	7	5	+	2	6	7	5	+	7	4	4	7
1					1	1				1	1	1			1	1	1	1	
	$\bar{1}$					$\bar{1}$	0				$\bar{1}$	0	1			$\bar{1}$	0	1	2
1					1	0				1	0	1			1	0	1	2	2

TAB. 2.4 – Système de numération redondant

Les systèmes de numération à chiffres signés

A. Avizienis suggéra en 1961 l'utilisation de systèmes de numération à chiffres signés [8], tel l'exemple étudié à la section précédente. Le principe consiste à coder les nombres en base r à l'aide de chiffres appartenant à l'ensemble $E_\rho = \{-\rho, \cdots, \rho\}$, où $\rho \leq r-1$ et $2\rho+1 \geq r$. Cette dernière condition garantit que tous les nombres possèdent au moins une représentation. Si $r = 10$ et $E_\rho = \{-4, \cdots, 4\}$, la condition $2\rho + 1 \geq r$ n'est pas satisfaite et les nombres $5, 15, 25, 35, 45, \cdots, 55$ n'admettent par exemple aucune représentation. Avizienis a démontré que si $2\rho+1 = r$, chaque nombre possède une écriture unique. Finalement, lorsque $2\rho + 1 > r$, le système de numération devient redondant et permet, sous certaines conditions, l'addition parallèle en temps constant. Mettons cette fois $r = 5$ et choisissons un ensemble de chiffres $E_\rho = \{-3, \cdots, 3\}$ tel que $2\rho + 1 > r$. Certains nombres possèdent maintenant plusieurs représentations.

Base dix	Base cinq	Base dix	Base cinq
0	0	5	10
1	1	6	11
2	2	7	12
3	$1\bar{2}$	8	$2\bar{2}$
4	$1\bar{1}$	9	$2\bar{1}$

TAB. 2.5 – Système de numération à chiffres signés.

Il faut encore établir une représentation des chiffres d'un ensemble E_ρ adapté aux circuits numériques, autrement dit, au système binaire. Nous décrivons la

Chiffre dans $E_\rho = \{-1, 0, 1\}$	Représentation borrow-save (x_i^+, x_i^-)
-1	(0,1)
0	(0,0) et (1,1)
1	(1,0)

TAB. 2.6 – Représentation de systèmes redondants

représentation borrow-save qui est bien décrite dans [LNBO03]. Ce système consiste en un ensemble $E_\rho = \{-1, 0, 1\}$. Un chiffre x_i appartenant à $E_\rho = \{-1, 0, 1\}$ est codé à l'aide de deux bits x_i^+ (bit positif) et x_i^- (bit négatif) tels que $x_i = x_i^+ - x_i^-$ table 2.6. Un nombre X de n bits, s'exprime alors sous la forme :

$$x = x^+ - x^- = \sum_{i=0}^{n-1} x_i \cdot 2^i - \sum_{i=0}^{n-1} x_i \cdot 2^i, \quad x_i \in \{-1, 0, 1\} \tag{2.18}$$

Puisque $2\rho + 1 > r$, $\rho = 2$ et $r = 2$, alors certains nombres auront plusieurs représentations. L'entier 6 peut par exemple être représenté comme (0110), $(1\bar{1}10)$ ou $(10\bar{1}0)$. Notons que ce système représente un nombre négatif sans aucun bit de signe additionnel. En outre, la négation d'un entier x est très simple, $x = (x_i^+, x_i^-)$ devient $-x = (x_i^-, x_i^+)$.

2.3 Technologies

Dans le but de montrer l'intérêt de la technologie que nous allons utiliser dans l'implémentation de notre approche, nous présentons dans cette section les différentes technologies, leurs domaines d'application, leurs avantages et leurs limites. La comparaison entre elles sera par la suite plus simple et plus claire.

2.3.1 Processeur

Un processeur est un circuit intégré standard, conçu pour des *fins universelles.* Autrement dit, *c'est une composante générique susceptible de faire n'importe quel calcul, mais qui n'est optimisée pour rien de particulier.* Ce petit morceau de silicium (tout simplement du sable raffiné et cristallisé) a typiquement une surface

de 1 cm^2 mais consiste en des millions de transistors. Il s'agit du circuit intégré le plus avancé qui représente un véritable cerveau au sein de l'ordinateur. Il est chargé d'effectuer particulièrement toute sorte de calcul, en plus d'y gérer les flux d'informations. On lui adjoint désormais de plus en plus de coprocesseurs pour l'aider dans les calculs massifs. On parle indifféremment de processeur, de microprocesseur et plus souvent, de CPU (**C**entral **P**rocessing **U**nit) ; la distinction est devenue de plus en plus floue. Puisqu'un processeur est conçu pour un usage universel, il contient un jeu d'instructions ne dépendant d'aucune application. Outre la mémoire principale, il englobe un ensemble de registres généraux, une mémoire cache et une unité arithmétique et logique.

Les processeurs sont idéaux pour les applications génériques telles que les applications financières, statistiques, médicales, éducationnelles et bureautiques. Par contre, ils ne sont pas convenables aux applications exigeant du calcul massif telles que la convolution des vecteurs, les applications vidéo, la multiplication de grands entiers et la transformée de Fourier rapide. Une autre solution est d'utiliser plutôt des processeurs spécialisés conçus pour accomplir une seule tâche. Une approche que nous allons aborder un peu plus loin.

Présentement, même les microprocesseurs d'usage universel ont également des idées et des influences des processeurs spécialisés, tels que le processeur Pentium MMX (**M**ulti**M**edia e**X**tensions) dans l'architecture d'Intel. Le processeur MMX contient un ensemble de 57 instructions additionnelles conçues particulièrement pour améliorer l'audio, la vidéo, les graphiques et les opérations de modem.

2.3.2 Microcontrôleur

Un microcontrôleur pourrait être défini comme étant un *ordinateur-sur-puce* optimisé pour réaliser une tâche bien précise. Un microcontrôleur est plus adapté aux applications embarquées, car il comporte sur sa puce toutes les mémoires et les interfaces d'Entrée/Sortie requises qui n'existent pas sur un microprocesseur à usage universel. D'ailleurs, il est généralement moins puissant en termes de rapidité, et sa taille mémoire adressable est petite et plus souvent cantonnée aux données de 8

ou 16 bits.

Les microcontrôleurs font partie de la majorité des systèmes embarqués tels que les automobiles, les avions, les appareils-photo, les téléphones, les feux de signalisation ainsi que les robots et les jouets. Ils se retrouvent dans la majorité des puces de processeurs vendus. Un exemple d'un microcontrôleur est le PIC (**P**eripheral **I**nterface **C**ontroller) qui tourne à une horloge relativement lente (l'horloge est divisée par 4 à l'intérieur du PIC), ce qui est peu rapide pour des applications complexes.

2.3.3 DSP

L'acronyme DSP est utilisé pour indiquer deux domaines différents mais complémentaires : DSP (**D**igital **S**ignal **P**rocessing) et DSP (**D**igital **S**ignal **P**rocessor).

Digital Signal Processing : Le terme DSP, ou le traitement de signaux numériques, comprend toute technique qui sert à traiter les signaux venant des sources telles que le son, les images, les satellites météo, les moniteurs de tremblement de terre, les tensions produites par le cœur et cerveau, les vibrations séismiques. Le DSP est la science qui utilise des ordinateurs pour comprendre ces types de données.

Habituellement, la première étape dans le DSP est de convertir le signal analogique en format numérique en employant un convertisseur analogique-numérique. Ces signaux sont par la suite analysés par différents algorithmes, notamment, la transformée de Fourier. Le DSP a deux champs d'application principaux : traitement des signaux audio (traitement de la parole) et traitement d'images.

Digital Signal Processor : Les processeurs d'usage universel intégrés dans les ordinateurs personnels répondent aux besoins communs du public tels que les applications bureautiques. Néanmoins, ces processeurs ne sont pas optimisés pour des algorithmes tels que le filtrage numérique ou l'analyse de Fourier. De là est venue l'idée d'exécuter ces algorithmes DSP sur des processeurs spécialisés qu'on appelle processeurs de signaux numériques, abrégé aussi DSP (**D**igital **S**ignal **P**rocessor).

Les DSP sont des processeurs conçus particulièrement pour exécuter des opérations coûteuses exigées dans le traitement de signal numérique et généralement en temps réel. Un DSP, par opposition à un processeur, a son propre jeu d'instructions RISC (**R**educed **I**nstruction **S**et **C**omputer) conçues et optimisées pour un problème bien défini. Les DSP sont extrêmement rapides et puissants afin de pouvoir accomplir des séquences d'opérations dans le même cycle d'horloge telles que décaler, ajouter et multiplier. Plutôt que de se limiter aux calculs généraux, les DSP ont habituellement un ensemble d'instructions optimisées employant les techniques suivantes :

- Les opérations **M**ultiply-**AC**cumulate (MAC) : une opération Multiplier-Accumuler achève le calcul d'une multiplication et addition en un cycle d'horloge, appelée le cycle MAC. On trouve cette opération dans quasiment tous les DSP et elle est surtout appréciée dans la convolution et la multiplication de matrices.
- Pipelinage à plusieurs niveaux (*Deep pipiline*)
- Mémoires de programme et de données séparées
- Virgule Flottante : La plupart de DSP utilisent l'arithmétique à précision fixe. Cependant, ceux à virgule flottante sont communs pour les applications scientifiques où la précision est exigée.
- Instructions spécialisées pour les modes d'adressage : adressage modulaire et adressage par inversion de bits particulièrement utilisées dans la FFT.

Ces dispositifs se sont déployés abondamment pendant la dernière décennie, trouvant l'utilisation dans tous les systèmes embarqués notamment les téléphones cellulaires, les télécopieurs, les cartes audio et les TV numériques. Ils sont aussi une solution attirante dans les systèmes informatiques temps réel.

2.3.4 Réseaux systoliques

Un réseau systolique est typiquement défini comme un réseau de processeurs élémentaires (PE pour **P**rocessing **E**lement) identiques qui calculent et échangent des données régulièrement. "L'analogie est faite avec la régularité de la contraction

cardiaque qui propage le sang dans le système circulatoire du corps. Chaque processeur d'un réseau systolique peut être vu comme un cœur jouant un rôle de pompe sur plusieurs flots le traversant. Le rythme régulier de ces processeurs maintient un flot de données constant à travers tout le réseau" [Gar03].

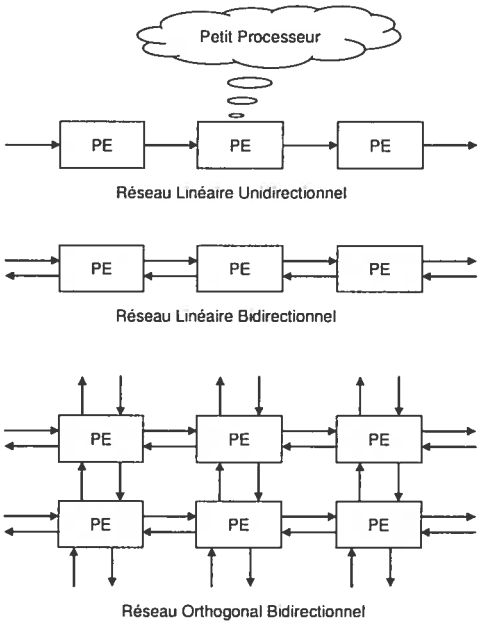

FIG. 2.2 – Les architectures systoliques les plus utilisées

La popularité de ce modèle vient en grande partie de la nécessité accrue de calculateurs extrêmement rapides et sa potentialité à réduire significativement le temps d'exécution des algorithmes séquentiels au prix d'un accroissement de la complexité du matériel par clonage (réplication) des structures élémentaires simples et régulières. Une donnée introduite dans le réseau est propagée d'un PE à un autre voisin et peut ainsi être utilisée maintes fois. Cette propriété fournit un débit très élevé même si la cadence des entrées-sorties reste faible. C'est ce qui fait que le modèle systolique est une solution adéquate pour un grand nombre de problèmes dont le nombre de calculs sur une même donnée est largement supérieur à son nombre d'entrée-sortie (compute-bound problems). Les réseaux systoliques les plus communément utilisés sont les réseaux systoliques linéaires unidirectionnels, les

réseaux systoliques linéaires bidirectionnels, les réseaux orthogonaux bidirectionnels comme le montre la figure 2.2.

2.3.5 ASIC

Les circuits ASIC (**A**pplication **S**pecific **I**ntegrated **C**ircuit) ou circuits intégrés spécialisés ont vu le jour au début des années 80. *Les ASIC regroupent tous les circuits dont la fonction peut être personnalisée, d'une manière ou d'une autre, en vue d'une application spécifique plutôt qu'un circuit adapté à maints buts tel qu'un processeur.* Étant optimisé et câblé de façon à accomplir un seul travail, un ASIC n'a pas ainsi à encourir le surchargement de chercher et d'interpréter les instructions à exécuter comme le fait un processeur. C'est pourquoi un ASIC pourrait être 10 à 100 fois plus performant qu'un processeur. Les cartes à puces sont un exemple typique d'un ASIC. Le premier ordinateur électronique de l'histoire, appelé *Colossus* et établi en 1943-4 [Wal99], s'est basé sur la notion d'ASIC bien que cette dernière est apparue 4 décennies après. Colossus était un processeur spécialisé pour cryptanalyser le code d'état major des Allemands durant la deuxième guerre mondiale. Par contre la deuxième machine connue sous le nom *ENIAC* était à usage universel.

L'apparition de la technologie ASIC a introduit une nouvelle notion de conception et fabrication dans l'industrie électronique. Avant l'ère des ASIC, les fabriquants ne construisaient que des microprocesseurs et des circuits intégrés standards notamment les mémoires. Les compagnies en concevaient ensuite de systèmes informatiques complets fonctionnels ; c'est un processus d'établir des systèmes à partir des puces (SoB pour **S**ystem **o**n **B**oard). Tandis que la conception d'ASIC, est un processus d'intégrer des systèmes sur les puces (SoC pour **S**ystem **o**n **C**hip) [And05]. Un ASIC standard pourrait inclure un ou plusieurs noyaux de microprocesseurs ainsi que les logiciels embarqués.

À la lumière de cette définition, les ASIC couvrent les circuits personnalisés ou à la demande (FC pour **F**ull **C**ustom) et les circuits semi-personnalisés. Ces derniers comportent également les réseaux logiques programmables (PLD pour

Programmable Logic Device), les prédiffusés (GA pour Gate Array) et les cellules standard(SC pour Standard Cell). La distinction principale entre ces circuits est le degré de la flexibilité dans la mise en application. Les FC et SC dans les masques, GA dans l'interconnexion en métal, et PLD dans des fusibles. Toutefois, l'interprétation dominante de ASIC se rapporte aux circuits GA et SC [LS89]. La figure 2.3 illustre graphiquement l'arborescence de ces différentes technologies.

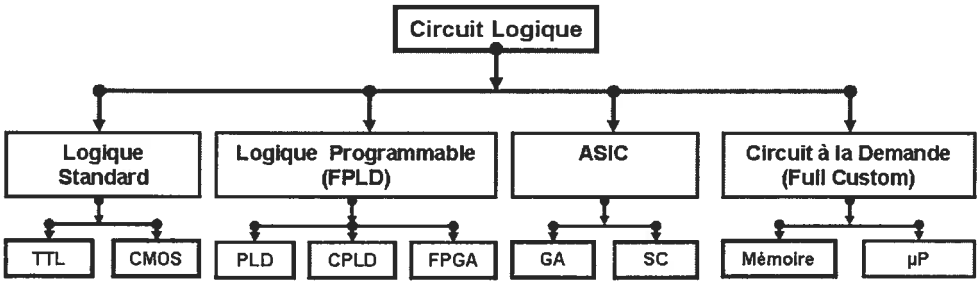

FIG. 2.3 – Vue d'ensemble de différentes technologies

Les réseaux prédiffusés (GA)

Un GA est un circuit partiellement fini ou préfabriqué de rangées des transistors et résistances, groupés en cellules et incorporés sur une puce suivant une certaine topologie, mais sans être connectés entre eux. Le circuit est accompli plus tard en ajoutant une couche en métal qui représente les fils d'interconnexion. La conception d'un masque d'interconnexions d'une application est la tâche du concepteur (client) alors que la réalisation de ce masque est exclusive au fabriquant. Pour cette raison, le fabriquant met à la disposition du concepteur un outil de développement accompagné d'une librairie standard de cellules (portes, registres, etc...) et macrocellules (RAM, ROM, etc...) afin de produire le masque de son circuit. A partir de ce masque, c'est au tour du fabriquant de parachever le circuit en ajoutant les fils métalliques d'interconnexion.

Cette dernière étape est moins coûteuse que la conception d'un circuit personnalisé FC qui exige un nouveau photo-masque pour chaque couche de transistor et d'interconnexion. Les GA sont habituellement préfabriquées et stockées en grande

quantité, ce qui réduit le temps de fabrication comparé au circuit standard ou sur commande. Pourtant, les GA ont une basse densité due aux cellules gaspillées lors de la phase d'interconnexion. Cette topologie est intéressante sur le plan de la conception et de la fabrication, par contre elle présente l'inconvénient de basse densité et performance.

Les cellules standards (SC)

Le circuit SC est la technologie de développement d'ASIC la plus commune. Le concepteur des SC a à sa disposition une librairie standard de cellules prédéfinies qui s'étendent des portes logiques primitives à des fonctions plus complexes telles qu'une mémoire ou un noyau de microprocesseur. Cette librairie constitue un véritable catalogue dont le concepteur se sert pour constituer son schéma. Basé sur la conception du client, les circuits désirés sont placés sur une puce et connectés par un logiciel de placement et routage "place-and-route". Par opposition aux GA qui sont partiellement fabriqués, les SC sont créés sur des puces de silicium vierges, ce qui nécessite une fabrication des masques personnalisés pour chacune des couches diffusées et des métallisations. Par contre, deux avantages évidents en découlent : alors qu'il est impossible avec les GA d'utiliser à 100% la surface du circuit (ce qui se traduit par une perte de silicium), les SC permettent de l'exploiter complètement. L'autre avantage est que les SC sont plus performants puisqu'ils sont plus optimisés.

Les cellules personnalisées ou à la demande (FC)

Tout comme le circuit standard SC, le circuit personnalisé FC emploie un masque personnalisé pour chaque couche dans la puce. À la différence des SC, les concepteurs des FC ont le contrôle total sur le circuit, que ce soit la taille du composant, le nombre de broches, l'emplacement et la taille de tout transistor. Ils peuvent ainsi raffiner la taille formant chaque transistor pour une performance optimale. Ils disposent pour cela d'une librairie de modèles mathématiques et d'un compilateur plus sophistiqué que celui utilisés dans la conception d'autres types de circuits.

Aujourd'hui, les circuits FC représentent un petit pourcentage du marché d'ASIC parce que les GA et les SC permettent de mettre en application les masques des circuits sur des puces fonctionnelles beaucoup plus rapidement et à bien moindre coût. Bien qu'étant le plus rapide et optimisé, les phases de mise au point d'un circuit FC sont longues et onéreuses. L'avantage de vitesse qu'offre un FC n'est plus ce qu'il était avant. Il est employé principalement pour construire les microprocesseurs et les mémoires qui doivent fonctionner aussi rapidement que possible et seront produits en grande quantité. D'ailleurs, le concepteur doit avoir l'expertise de la technologie en plus de l'expertise des aspects algorithmiques, architectures et implémentations.

Avantages et inconvénients

L'utilisation d'un ASIC apporte généralement de nombreux avantages :

- Optimisation maximale, côté performance et espace silicium, du circuit à réaliser vu qu'il est tellement personalisé.
- Réduction considérable de la consommation et de l'encombrement du circuit dû à l'optimisation du nombre de composants.
- Personnalisation du circuit qui offre une confidentialité au concepteur et une protection industrielle.
- Très haute complexité, vitesse de fonctionnement et fiabilité.
- Personnalisation du fonctionnement et augmentation de la densité d'intégration.

Toutesfois, les ASIC présentent les inconvénients suivants :

- Une flexibilité réduite puisque la moindre modification dans l'application tournant sur le circuit exige le remplacement de ce dernier.
- Un autre inconvénient majeur réside dans le passage obligatoire chez le fabriquant. Ceci implique des surcoûts de développement.
- Les ASIC nécessitent qu'on soit très rigoureux lors de la phase de développement, de telle sorte que le circuit prototype fonctionne dès les premiers essais pour

réduire les surcoûts. Le prototype fonctionne dans environ 60% des cas seulement.

- La conception d'un ASIC nécessite la connaissance et l'expertise d'un mélange des aspects : l'approche de conception, l'architecture et la technologie d'implémentation.
- Les ASIC complexes nécessitent souvent des programmes de testabilité qui augmentent le coût de développement.

2.3.6 Les circuits logiques reconfigurables

Les spécifications dans un système numérique sont en perpétuel changement et par conséquent, la présence des failles y est très probable. D'ailleurs, la probabilité que le circuit fonctionne dès la première fois est 60% [LS89] même avec toutes les précautions et le travail rigoureux. D'autre part, si les spécifications d'un circuit logique changeaient ou si une erreur avait échappé à la simulation, le circuit serait remplacé vu que ce dernier est figé à l'usine. Pour parer à ces inconvénients, les réseaux logiques programmables (FPLD pour **F**ield **P**rogrammable **L**ogic **D**evice) ont été développés.

Un FPLD se rapporte à tout circuit logique manufacturé à l'usine en tant que dispositif générique et plus tard programmé pour une application spécifique sans aucune étape technologique supplémentaire. Autrement dit, on n'a pas besoin de passer à l'usine une fois le circuit est fabriqué.

La programmabilité de tels circuits signifie que de nouvelles conceptions des circuits peuvent être facilement examinées et changées sans engendrer les coûts que demandent les ASIC et les circuits personnalisés. Il peut ainsi être reprogrammé à plusieurs reprises. Ceci permet de mettre à jour, et sur place, les produits mis déjà en application sans remplacer les anciennes puces.

Un FPLD typique consiste en une matrice de blocs ou cellules logiques configurables entourées de blocs d'Entrée/Sortie configurables. L'ensemble est relié par des ressources d'interconnexions configurables également. Une cellule logique d'un

FPLD est typiquement capable de réaliser une fonction logique combinatoire ou séquentielle de différents niveaux de complexité. Les FPLD actuels incluent un grand nombre d'architectures dont les cellules logiques sont toutes basées sur un ou plusieurs circuits logiques :

- Les fonctions logiques de base : NAND et XOR à deux entrées
- Les multiplexeurs
- Les LUT (**L**ook-**U**p **T**able)
- Les structures AND-OR à large entrée

La méthodologie de programmation des FPLD, dont nous allons parler en détail dans la section 2.4, est similaire à celle utilisée dans les différents types de mémoires ROM (PROM, EPROM, EEPROM). En fait, les PROM et EPROM sont considérées comme des FPLD lorsqu'elles contiennent du code de programmes plutôt que des données uniquement. Cette programmation, qui se fait grâce à différents outils de développement, consiste à établir des connexions en imposant un courant supérieur aux courants de fonctionnement normaux (claquage de fusibles ou de jonctions).

Les FPLD comportent les PLD (**P**rogrammable **L**ogic **D**evice), les CPLD (**C**omplex PLD) et les FPGA (**F**ield **P**rogrammable **G**ate **A**rray) [Lee00]. Notons que l'acronyme PLD a été utilisé parfois par certains fabriquants pour signifier le même sens que le FPLD, pourtant il est typiquement utilisé pour se rapporter aux PLA (**P**rogrammable **L**ogic **A**rray), PAL (**P**rogrammable-**A**nd **L**ogic) et GAL (**G**eneric **A**rray **L**ogic).

PLD

Les efforts de recherche des constructeurs portent plus sur les circuits à plus forte densité d'intégration que sont les CPLD et les FPGA. Le fait que les PLD soient à la base de la conception des CPLD, très en vogue aujourd'hui, justifie cependant leur étude.

PLA : Toute fonction logique peut être codée par une somme de produits (SDP). Ceci est le fondement du PLA, le premier dispositif développé spécifiquement pour

l'usage comme un circuit configurable.

Un PLA dispose d'une matrice ET programmable et d'une matrice OU programmable. Comme montre la figure 2.4 qui est prise de [Dav02], ce circuit prend m entrées et leurs complémentaires qui arrivent sur p portes ET à m entrées. La matrice ET produit aussi p sorties qui arrivent à leur tour sur n portes OU à p entrées. Un PLA $m \times p \times n$ peut ainsi implémenter toute fonction logique de n SDP à deux niveaux . L'architecture d'interconnexion est simple, avec chaque

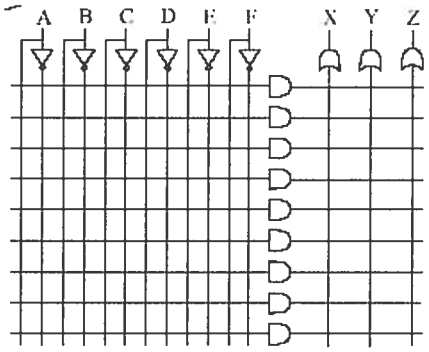

FIG. 2.4 – Architecture d'un PLA

entrée connectée à chaque porte ET qui, à son tour, connectée à chaque porte OU par une seule connexion programmable (un fusible) comme montre la figure 2.5. Le principe d'interconnexion consiste à détruire un fusible conducteur par passage

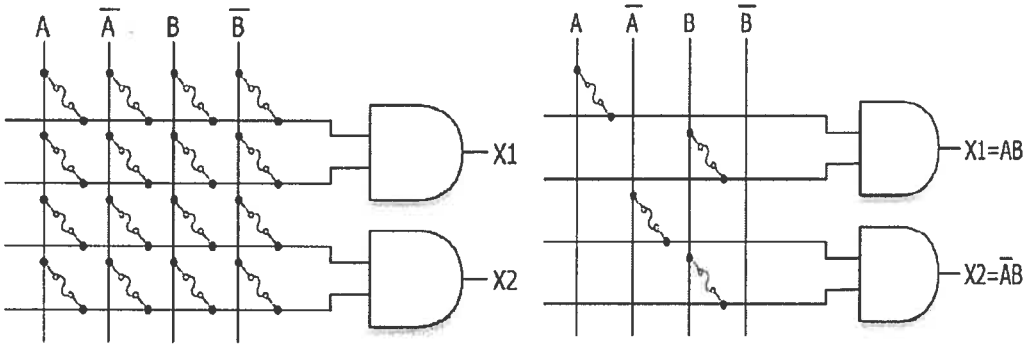

FIG. 2.5 – Programmation d'un PLA

d'un courant fourni par une tension supérieure à l'alimentation (12 à 25 V). Lors-

qu'un fusible est brûlé, la connexion entre la variable et la porte correspondante disparaît. La programmation du système se fait en choisissant les fusibles que l'on laisse ou que l'on brûle. Une fois brûlé, un fusible ne peut plus être utilisé. Les PLD à fusibles ne sont donc pas reprogrammables (ou effaçables). Ces circuits ont aujourd'hui quasiment disparu au profit de technologies plus performantes.

PAL : La structure du PLA est utilisée dans certains circuits ASIC qui demandent une densité d'intégration importante. En effet, pour n variables en entrées, il faut 2^n fonctions ET à $2n$ entrées et au moins un OU à 2^n entrées. La plupart des applications n'exigent pas une telle complexité et on peut se contenter d'une matrice ET programmable et d'une matrice OU figée. De même, puisqu'il est peu probable d'utiliser tous les termes produits, on peut alors limiter le nombre d'entrées de la fonction OU. Cette architecture économique est appelée PAL qui n'est qu'une restriction de son ancêtre PLA. Un PAL dispose ainsi d'une matrice ET programmable et d'une matrice OU figée (Figure 2.6). C'est un circuit plus simple à programmer

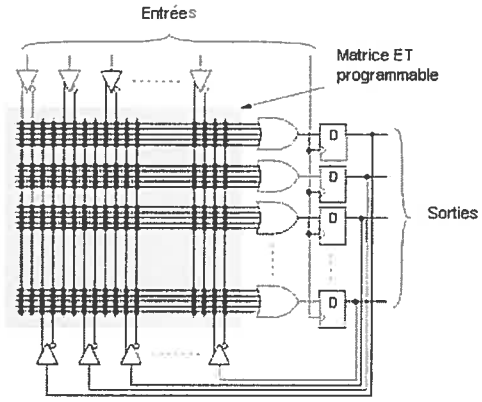

FIG. 2.6 – Architecture d'un PAL

mais moins flexible que le PAL. Certains modèles permettent de réaliser des circuits logiques séquentiels en intercalant un registre entre les sorties des portes OU et les entrées des portes ET. Habituellement, les PAL sont définis par leur nombre d'entrées et de sorties ; par exemple, un PAL 22v10 signifie un PAL à 22 entrées et 10 sorties. Tout comme le PLA, le principe d'interconnexion de PAL est basé sur la destruction des fusibles. Néanmoins, d'autres solutions technologiques rem-

placent aujourd'hui les fusibles, permettant l'effacement (électrique ou par UV) et la reprogrammation des PLD à plusieurs reprises.
GAL : Le GAL est une évolution de PAL et la dernière génération de PLD. Ce circuit est plus générique, reconfigurable et offre plusieurs autres fonctionnalités.

CPLD

La nécessité de réaliser de plus en plus des systèmes numériques complexes sur un même circuit a conduit tout naturellement à intégrer plusieurs PLD simples (blocs logiques) sur une même puce, reliés entre eux par une matrice d'interconnexion reconfigurable. Depuis, la taille d'un circuit n'est plus limitée que par la technologie [Dav02] et des applications extrêmement complexes se réalisent et se modifient aisément. Les CPLD utilisent typiquement la EEPROM, mémoire instantanée (Flash memory) ou SRAM pour tenir les interconnexions de la conception.

2.4 La technologie FPGA

Un FPGA, ou réseaux de portes programmables, est simplement une évolution de CPLD qui combine un grand nombre de PLD tels que les EPROM, les PAL et les GAL [Lee00]. Un FPGA, inscrit au sommet de l'évolution des composants logiques programmables [Dav02], est le circuit le plus complexe qui peut être programmé sur place pour réaliser un circuit logique quelconque. Il est constitué d'une matrice de blocs ou cellules logiques identiques entourées des blocs d'Entrée/Sortie. L'ensemble est relié par des canaux de routage entre les colonnes adjacentes et les rangées adjacentes. Les cellules logiques, les blocs d'Entrée/Sortie et les canaux de routage sont tous reconfigurables.
Une autre vue d'un FPGA est, comme indique son nom, une version évoluée des réseaux logiques (GA) dont nous avons discutés dans la section précédente. Contrairement aux circuits GA conventionnels, les FPGA ne demandent pas de fabrication spéciale en usine, ni de systèmes de développement coûteux. Il peut être configuré sur place et utilisé quelques minutes après. Ceci nous évite le passage chez le fabri-

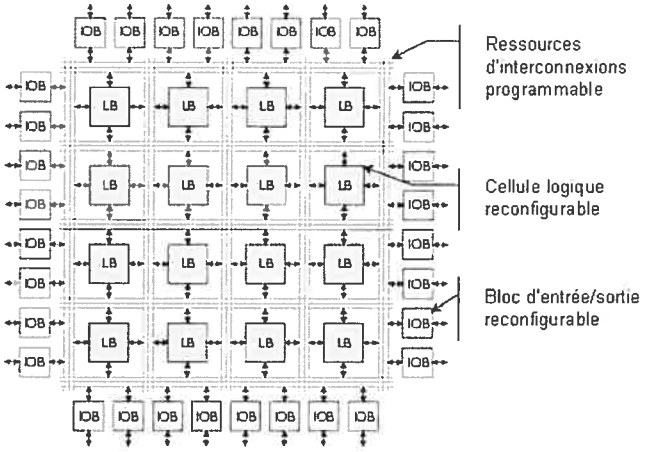

FIG. 2.7 – Architecture simplifiée d'un FPGA

quant et tous les inconvénients qui en découlent.

Une carte FPGA est accompagnée d'un outil de développement très convivial permettant de décrire, compiler, simuler et synthétiser un circuit. Grace à cet outil, nous pouvons étudier le comportement d'un circuit dans des conditions réelles en quelques minutes. Les FPGA constituent ainsi un formidable outil de prototypage [Beu01] en offrant toutes les possibilités de personnalisation matérielle, un prix abordable au public et une réutilisation illimitée.

Les caractéristiques des nouvelles générations de FPGA élargissent encore leurs champs d'application. Ils sont utilisés dans maintes applications : DSP, aérospatiales et de défense, prototypage d'ASIC, imagerie médicale, reconnaissance de parole, cryptographie, et une gamme d'autres secteurs. L'ajout d'une carte FPGA à un ordinateur personnel permet le développement de coprocesseurs spécifiques à chaque application. Les utilisateurs d'ordinateurs ont ainsi le pouvoir de concevoir leurs propres systèmes numériques. Les FPGA sont même employés par les ingénieurs dans la conception des ASIC, évitant par la suite les pertes dues aux erreurs probables et spécifications insatisfaisantes. Les FPGA concurrencent aujourd'hui même les circuits ASIC dans certaines applications. Ils sont moins onéreux que l'ASIC pour de petites séries et bénéficient des meilleures technologies de gravure. La conception de systèmes performants nécessite toutefois une

exploitation judicieuse des ressources d'un FPGA.

Le progrès rapide de ces technologies permet de faire des composants toujours plus rapides et à plus haute intégration, ce qui permet de programmer des applications assez complexes. Les FPGA récents sont configurables en une centaine de millisecondes et contiennent des millions de portes logiques. Certains sont si sophistiqués qu'ils englobent également des mémoires entières, RAM et ROM, des multiplieurs et même des noyaux de processeurs. Avec l'avènement du FPGA, il est aussi possible de connecter un ensemble des FPGA afin de réaliser des circuits de plus en plus performants et complexes. L'approche multi-FPGA est particulièrement adoptée dans la simulation et prototypage rapides des ASIC afin d'éviter de simuler pendant de longues heures avec un simulateur logiciel [Dav02].

Ces caractéristiques (reconfiguration, haute intégration, performance) ont fait des FPGA un outil idéal pour l'implémentation matérielle d'innombrables applications numériques. Notamment, les applications cryptographiques qui nécessitent fréquemment le changement des paramètres en temps réel, des crypto-processeurs assez complexes et un débit de chiffrement en très court délai.

Le marché des FPGA est dominé par de nombreux fabricants internationaux dont Altera et Xilinx. Dans la section suivante, nous passerons en revue la technologie proposée par Altera puisque nous avons leur technologie dans notre laboratoire.

2.4.1 FPGA fabriqué par Altera

Altera adopte deux types d'architectures différents basés sur des sommes de produits et LUT (LookUp Table).

Architecture sommes de produits : Cette architecture est une génération améliorée des PAL auxquels on a intégré un réseau hiérarchique de connexions. C'est ce qui explique leur nom de famille MAX (**M**ultiple **A**rray matri**X**) dont l'architecture est illustré par la figure 2.8 prise du thèse de J.-P. David [Dav02].

Le circuit MAX 7 000S par exemple contient de 600 à 20 000 portes logiques. Il est configuré à l'aide d'une EEPROM qui tient la configuration même après l'abscence du courant. Cette famille consistant en 32 à 256 cellules logiques ressemble

au premier PAL. Chaque cellule contient 5 portes ET reconfigurable à n entrées (n est très grand) qui arrivent à une porte OU avec une inversion programmable à sa sortie. La sortie d'un réseau ET/OU peut être introduite à une bascule (Flip-Flop) configurable permettant ainsi de réaliser des circuits logiques séquentiels.

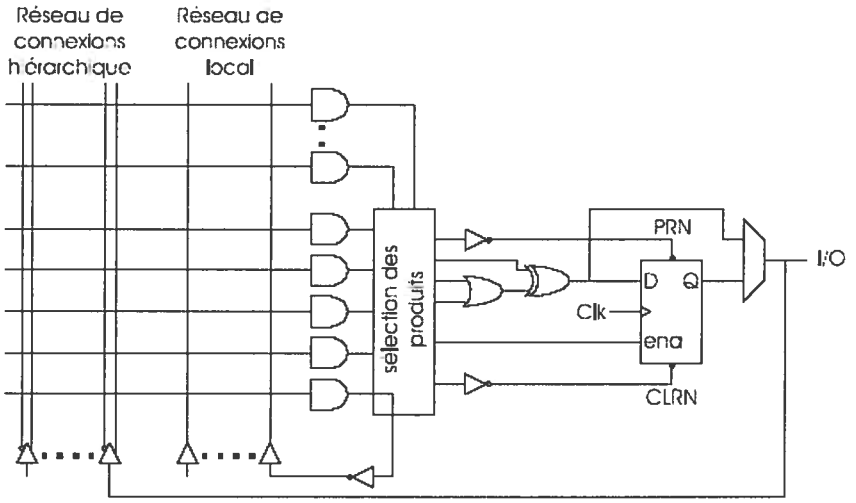

FIG. 2.8 – Architecture MAX simplifiée

Architecture LUT : Le principe de cette architecture est complètement différent. Le LUT est une structure de données, habituellement une matrice, employée pour remplacer un calcul (en temps réel) par une opération de consultation beaucoup plus simple. Le gain de vitesse peut être très significatif puisque rechercher une valeur de la mémoire est souvent plus rapide que réaliser un calcul coûteux. Un exemple classique est la table trigonométrique. Le calcul du sinus d'une valeur donnée peut être très coûteux dans quelques applications. Pour éviter ceci, l'application peut prendre quelques secondes quand elle commence pour pré calculer le sinus d'un certain nombre de valeurs de haute fréquence. Plus tard, quand l'application veut le sinus d'une valeur, elle emploie cette table pour le rechercher au lieu de le calculer par une formule mathématique complexe.

Ce principe est utilisé dans la famille FLEX (**F**lexible **L**ogic **E**lement matri**X**) où chaque cellule dispose de quatre entrées. Elle dispose donc de $2^4 = 16$ combinaisons

différentes de ces entrées. L'idée est de mémoriser la sortie correspondant à chaque combinaison d'entrée dans une petite table de 16 bits appelé le LUT. Ce dernier permet ainsi d'implémenter n'importe quelle fonction logique de 4 entrées. Tout comme dans la famille MAX, un registre peut être intercalé entre la fonction logique et la sortie qui lui correspond pour réaliser des circuits séquentiels. La figure 2.9, prise de [Dav02], montre la structure interne d'un LUT proposé par Altera.

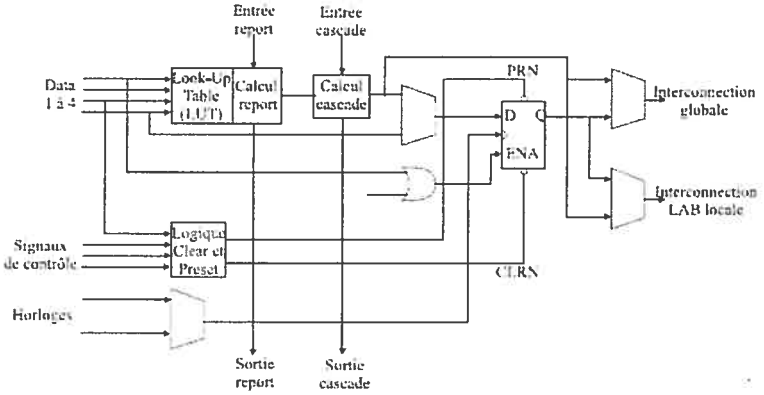

FIG. 2.9 – Architecture d'un LUT

2.4.2 Programmation ou configuration des FPGA

Bien que chaque fabriquant ait ses propres outils de développement, l'implémentation d'un circuit logique à l'aide d'un FPGA consiste en plusieurs étapes communes comme le montre la figure 2.10 prise de [FPG01]. À titre d'application de la programmation d'un FPGA, nous considérons un exemple simple que nous avons testé au début de cette recherche. Il s'agit de la multiplication modulaire de deux entiers à 5 bits.

1. **Description du circuit :** La description d'un circuit logique peut être faite à travers un éditeur graphique, textuel ou même les deux. Typiquement, un circuit logique est entré via un éditeur textuel en utilisant un langage de description de matériel notamment VHDL ou Verilog. Il est aussi possible d'utiliser une description plus simple grâce à un langage intermédiaire permettant de générer

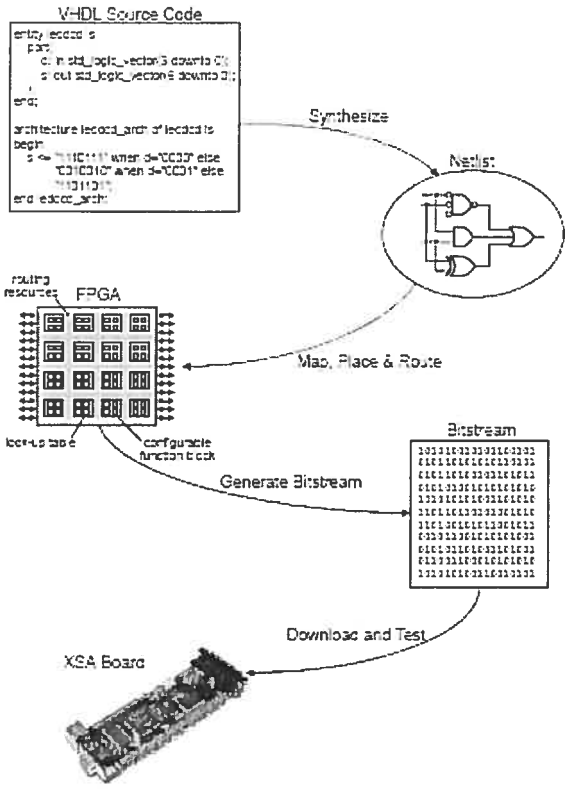

FIG. 2.10 – Étapes de programmation d'un FPGA

le code VHDL ou Verilog correspondant. Détaillé plus tard, le CASM est une instance de tel langage qui a été développé dans notre laboratoire. Vous trouvez ci-après le code CASM d'un module permettant de calculer la multiplication modolaire de deux entiers.

À l'aide de l'interface graphique, on dessine le schéma du circuit à l'aide de connexions sur des cellules de base à partir d'une librairie standard. Cette méthode rend difficile la réalisation de circuits complexes où chaque changement ou amélioration remet en cause toute la description.

Le module ASM ci-dessous a trois entrées (ina,inb, modulo) de 5 bits, une sortie (result) de 5 bits, un registre temporaire (temp) de 6 bits ainsi que trois états finis (START, N0, N1). L'exécution commence évidemment à l'état START qui

effectue l'addition sur 6 bits. À la fin de cet état, le résultat sera prêt et le contrôle sera donné à l'état N0 par l'instruction goto-N0. Ce dernier, effectue un test si le résultat obtenu est plus grand que le modulo ou non. Si la condition est vraie, il faut retrancher le modulo du temp avant d'aller à l'état final N1, sinon il faut simplement aller à l'état N1 pour afficher le résultat.

```
input   ina[5],inb[5],modulo[5];
output  result[5];
ASM A1 on clk {
    register    temp[6];
    START:  do  temp=uxt{6}((ina + inb));   goto N0;
    N0: if(temp~> (uxt{6}(modulo))) temp=uxt{6}((temp-modulo));
            goto N1;
        else goto N1;
    N1: do  result is A1->temp.[4..0]; goto START;
}
```

2. **Compilation et synthèse :** La compilation du circuit commence par la vérification de la cohérence de la description et la syntaxe du langage utilisé. Cette étape est aussi chargée de réaliser la synthèse du circuit. La *synthèse* est une transformation de la description du circuit en un "réseau d'interconnexion" (en anglais *netlist*). Un netlist est simplement une description des différentes portes logiques et ainsi que la manière dont elles sont interconnectées. Une compilation sans erreur est suivie d'un rapport (Figure 2.11) portant sur toutes les ressources exigés par le circuit en question (nombre des registers, nombre des broches, mémoires). Cette étape est réalisable à condition de se limiter à un sous ensemble du langage qui soit strictement synthétisable.
3. **Simulation fonctionnelle :** Le circuit doit être normalement simulé avant de le télécharger au FPGA pour s'assurer qu'il fonctionne selon le cahier de charges. C'est une simulation initiale qui met en relief le comportement idéal du circuit sans tenir compte des délais dus au routage entre les différentes cellules.

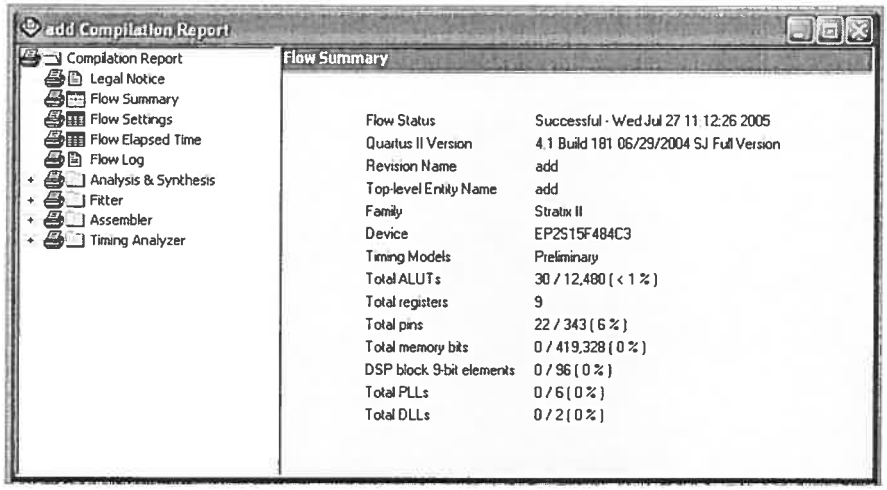

FIG. 2.11 – Compilation d'un circuit

Elle permet donc de vérifier uniquement la validité du circuit par rapport à la spécification d'un point de vue fonctionnel et non d'un point de vue temporel.

L'outil de simulation permet de déboguer et contrôler chaque variable ou signal dans le circuit. La simulation se fait à partir d'une interface graphique montrée dans la figure 2.12. Au départ, on ouvre un fichier de simulation puis on détermine les traces de simulation (variables, signaux, registres...). À ce niveau, le principe est le même que celui du débogueur C++ ou Java. La simulation nécessite toutefois des données d'entrée externes, on parle souvent d'un vecteur d'entrée ou stimulus. Dans notre exemple, c'est le cas de nreset, clk, ina et inb. Le simulateur va ensuite générer des signaux représentant le fonctionnement réel du circuit permettant ainsi de visualiser la forme des signaux et les valeurs attribuées aux variables du circuit.

À titre d'exemple, observons le fonctionnement du circuit pour $a = 6$ et $b = 15$. D'abord, observons qu'il y a un résultat prêt tous les trois cycles d'horloge ; nous avons ajouté la trace des états pour plus de clarté. À l'état START, la somme de a et b est effectuée (21) et le contrôle est donné à l'état N0. Dans ce dernier, 21 est plus grand que le modulo 17, alors il faut en soustraire 17 avant de l'afficher pour obtenir finalement 4. Le résultat final est déjà prêt à l'état N1, il est ainsi

affiché. Notons que dans START et N0 le résultat n'est pas prêt, ce qui justifie le fait que ce dernier y est nul.

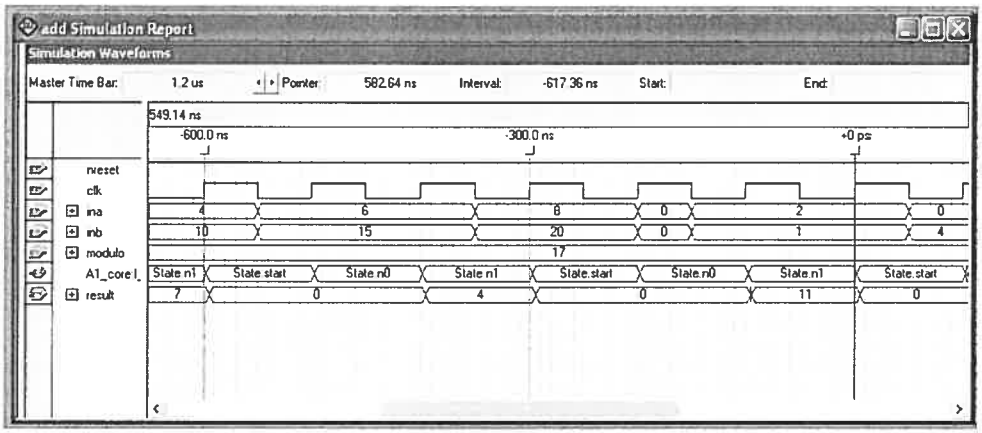

FIG. 2.12 – Simulation d'un circuit

4. **Projection, placement et routage :** La phase de projection adapte le netlist au FPGA cible. Nous savons qu'un FPGA consiste en blocs logiques reconfigurables (CLB). Ceux-ci peuvent être encore décomposés en LUT qui effectuent des opérations logiques. Les CLB et les LUT sont entrelacés avec des différentes ressources d'interconnexion. L'outil de synthèse rassemble les portes logiques de netlist en groupes s'adaptant dans les LUT et l'outil de placement/routage attribue à ces groupes des cellules spécifiques (CLB) tout en ouvrant ou fermant les commutateurs (en anglais *switches*) dans les matrices d'interconnexion pour relier les portes ensembles.
5. **Simulation temporelle :** Le circuit doit être simulé encore une fois pour vérifier sa fonctionnalité en prenant en compte cette fois des délais introduits par les longueurs d'interconnexion, les cellules utilisées ainsi que le placement et routage. La simulation temporelle vérifie que la fonctionnalité n'a pas été modifiée par l'introduction des délais de propagation et reste conforme au cahier de charges.
6. **Génération du fichier de configuration :** Une fois que la phase d'implémentation est achevée, un programme spécifique extrait les états des

commutateurs dans les matrices d'interconnexion et genère par la suite le fichier de configuration du FPGA, appelé en anglais *Bitstream*. Dans ce dernier, les 0 respectivement les 1 correspondent aux commutateurs fermés et ouverts. Il s'agit du fichier de configuration (ou de programmation) à charger sur le FPGA et qui correspond évidemment à la spécification du circuit.

7. **Programmation et test :** L'outil de programmation charge le Bitstream à une carte FPGA dont les commutateurs s'ouvrent ou se ferment selon les valuers binaires de ce fichier. Une fois ce chargement est terminé, le FPGA commence ensuite à effectuer les opérations selon la description tout comme un circuit intégré usuel.

2.4.3 Avantages et limites des FPGA :

Les FPGA actuels ont plusieurs avantages et n'ont plus rien à envier à ceux des autres circuits logiques :

Reconfiguration : L'argument fort d'un FPGA est évidemment la possibilité de reconfiguration même après qu'il ait été utilisé pour réaliser un système numérique. Ceci permet une meilleure exploitation du composant, une réduction considérable du temps de conception et une évolutivité assurant la possibilité de couvrir à long terme des nouveaux besoins sans nécessairement repenser l'architecture dans sa totalité ou changer la carte comme c'est le cas des ASIC. Par conséquent, un FPGA offre une diminution de coût considérable.

Reconfiguration dynamique : Une autre richesse d'un FPGA est la possibilité de reconfiguration dynamique partielle ou totale d'un circuit. L'intérêt de la reconfiguration dynamique est effectivement de permettre de modifier la fonctionnalité en quelques microsecondes, c.à.d, en temps quasi réel. Ainsi, le même CLB pourra à un instant donné être intégré dans un processus de cryptage AES à 256 bits et l'instant d'après être utilisé pour gérer le cryptosystème RSA à 1024 bits. On dispose donc totalement de la souplesse d'un système informatique qui peut exploiter des programmes différents, mais avec la différence fondamentale qu'ici il ne s'agit pas de logiciel mais de configuration matérielle, ce qui est beaucoup plus rapide.

Haute complexité : La complexité des FPGA a augmenté considérablement. Les familles les plus récentes permettent d'obtenir des densités pouvant atteindre plus de 10 millions de portes logiques configurables.

Large domaine d'application : Les domaines d'application se sont élargis au fur et à mesure que la taille et la vitesse des circuits augmentaient, au point où ils sont maintenant capables de concurrencer les ASIC. En outre, il semble que de plus en plus fréquemment les concepteurs de circuits ASIC préfèrent passer par l'étape intermédiaire d'un FPGA ce qui est moins risqué économiquement. Puis, une fois que le modèle FPGA est au point, il est alors relativement aisé de le retranscrire dans une architecture de type prédiffusé ou précaractérisé.

Outil pédagogique idéal : Le FPGA représente un outil sans précédant pour des fins pédagogiques puisqu'il est possible de réaliser un circuit logique sur place sans aucun coût une fois que la carte est disponible.

Bas prix : Bien que les FPGA aient connu une baisse significative de leur prix, une carte FPGA pourrait coûter entre 150 $ et 10 000 $ dépendamment de ses caractéristiques. Par contre, ceci sera beaucoup moins cher en grande quantité.

Performance et consommation : Les FPGA restent limités en performance de calcul et consommation d'énergie. À ce haut niveau de complexité, la conception n'est plus aussi simple et rapide qu'avant car ils doivent intégrer toutes les ressources nécessaires au caractère reconfigurable de chacune des petites cellules logiques. D'ailleurs, l'interconnexion prend beaucoup d'espace, ce qui résulte d'une puce à basse densité et par la suite d'une consommation plus élevée.

Circuit non confidentiel : Le FPGA n'est pas un circuit très sécurisé sur le plan de la confidentialité puisqu'il suffit d'analyser le contenu de la ROM associée pour remonter à la schématique imaginée. Néanmoins, dans les nouvelles générations des FPGA, telle que Virtex-II Pro, ce problème a été attaqué en utilisant le 3-DES pour chiffrer le fichier de configuration (bitstream).

Notons enfin que ces circuits n'ont pas vocation à concurrencer les super calculateurs, mais plutôt à offrir une alternative en fonction de critères comme l'encombrement, les performances et le prix, et sont de ce fait bien adaptés à des

applications de qualité dans le domaine des systèmes changeants.

Nous terminons ce chapitre par la figure 2.13 qui schématise les avantages et les limites de différentes technologies rencontrées précédemment en matières de plusieurs caractéristiques (densité, performance, coût de conception, etc.).

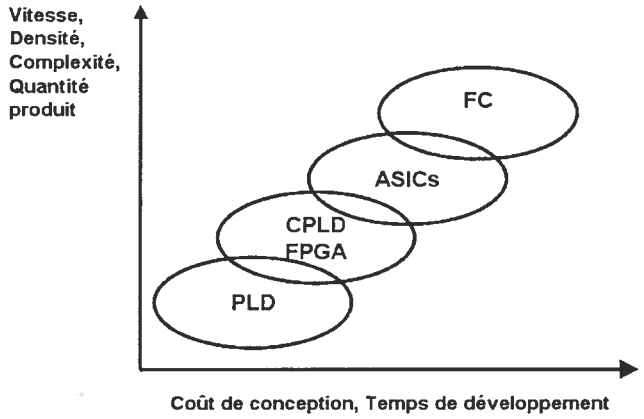

FIG. 2.13 – Comparaison de différentes technologies

CHAPITRE 3

TRANSFORMÉE DE FOURIER RAPIDE

Il est parfois plus efficace de transformer un problème avant de le résoudre. Par exemple, si vous multipliez deux entiers représentés en chiffres romains, il vous serait plus efficace de les convertir tout d'abord en notation arabe [BB88]. Ce processus, appelé transformation ou transformation de domaine, est utilisé pour faciliter considérablement divers problèmes. La fonction logarithmique en est un exemple très important ; elle remplace la multiplication dans $\mathbf{R}^+$ par l'addition dans $\mathbf{R}$. La transformée de Fourier rapide est l'algorithme le plus utilisé actuellement dans l'analyse et le traitement de données numériques. Cette importance vient du fait qu'elle réduit remarquablement le temps de la multiplication de $O(n^2)$ (méthode classique) à $O(n \log n)$[1]. Notons qu'une telle transformation n'est avantageuse que si le calcul dans le nouveau domaine et la conversion entre ce dernier et le domaine original sont plus efficaces que le calcul dans le domaine original.

Une transformation de domaine exige évidemment la connaissance de deux domaines ainsi qu'une technique de conversion entre eux. La FFT se base sur la représentation par valeur des polynômes, les propriétés des racines $n^{\text{ièmes}}$ de l'unité et la stratégie "*diviser-pour-régner*". Pour cela, nous commencerons ce chapitre par la présentation de certaines notions mathématiques inspirées du livre de T. Cormen et al. [CLR94]. Ensuite, nous décrirons les différents types de la transformée de Fourier et montrerons que le calcul dans le domaine transformé est plus rapide que celui dans le domaine original. Nous présenterons surtout la FFT en arithmétique modulaire inspirée du livre de G. Brassard et P. Bratley [BB88], une version plus rapide pour la multiplications des grands entiers, et donnerons un exemple à titre d'une application pratique de notre approche.

[1]Nous avons mentionné que cette complexité ne s'applique que dans le cas où les opérations élémentaires portent sur des scalaires de ℓ bits, où ℓ est la taille d'un bloc. Si on voulait s'astreindre à ne compter comme élémentaire que les opérations au niveau du bit, cette compléxité passerait à $O(n \log n \log\log n)$.

3.1 Polynômes

La transformée de Fourrier repose, entre autres, sur certaines propriétés de polynômes. C'est pour cela nous commençons ce chapitre par leur étude. En fait, la transformée de Fourrier a été conçue à l'origine pour calculer la convolution des polynômes.

3.1.1 Définition

Un **polynôme** en x, défini sur un anneau commutatif $\mathbf{A}$, est la somme formelle :

$$p(x) = \sum_{i=0}^{n-1} a_i x_i = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} \tag{3.1}$$

Les valeurs $a_0, a_1, \cdots, a_{n-1}$ sont appelées les *coefficients* du polynôme qui appartiennent toujours à l'anneau $\mathbf{A}$ et souvent au corps des nombres complexe $\mathbf{C}$. Un polynôme $p(x)$ est dit de *degré* k $(0 \leq k < n)$, si k est le plus grand entier tel que a_k n'est pas nul. Nous parlerons plutôt tout au long de ce mémoire d'un polynôme de *borne* n [CLR94]. Le degré d'un polynôme de borne n peut être n'importe quel entier strictement plus petit que n. L'intérêt de cette notion vient du fait que nous avons besoin de savoir le nombre exact des éléments d'un vecteur indépendamment de leurs valeurs. Par exemple,

- $3x^2 - 5x + 7$ est un polynôme de degré 2 et de borne 3
- $2x^{-1} + x + 1$ n'est pas un polynôme

Dorénavant, tous nos calculs seront effectuées en général dans un anneau commutatif $\mathbf{A}$, ou bien dans l'anneau $\mathbf{Z_m}$, à moins d'une indication contraire.

3.1.2 Opérations sur les polynômes

Une grande variété d'opérations peut s'effectuer sur les polynômes. Néanmoins, nous ne décrivons que l'addition et la multiplication qui seront utilisées plus tard. Dorénavant, soient $A(x) = \sum_{i=0}^{n-1} a_i x^i$ et $B(x) = \sum_{i=0}^{n-1} b_i x^i$ deux polynômes de borne n.

Addition des polynômes : La somme de $A(x)$ et $B(x)$ est un polynôme $C(x)$ de borne n également, tel que :

$$C(x) = \sum_{i=0}^{n-1} c_i x^i, \quad c_i = (a_i + b_i) \tag{3.2}$$

Multiplication des polynômes : La multiplication de $A(x)$ et $B(x)$ est le polynôme produit $C(x)$ de borne $2n-1$ tel que :

$$C(x) = \sum_{i=0}^{2n-2} c_i x^i, \tag{3.3}$$

où

$$c_i = \sum_{k=0}^{i} a_k b_{i-k}. \tag{3.4}$$

Notons que degré(C)= degré(A) + degré(B), ce qui implique que, si $n_a - 1$ et $n_b - 1$ sont respectivement les degrés de $A(x)$ et $B(x)$, alors $C(x)$ est un polynôme de degré $n_a + n_b - 2$ et on dit normalement que le polynôme produit $C(x)$ est de borne $n_a + n_b - 1$. Nous parlerons toutefois dans ce mémoire de la borne de C comme étant la somme des bornes de A et B, soit $n_a + n_b$.

Définition 3.1.2.1 (Définition de Convolution). *Le vecteur de coefficients résultant c, donné par l'équation 3.3, est souvent appelé* **convolution** *des vecteurs d'entrée a et b, ce qu'on note par* $c = a \otimes b$.

3.1.3 Représentations des polynômes

Un polynôme peut être représenté de deux façons équivalentes : par coefficients ou par valeurs. La première est très commode pour l'évaluation d'un polynôme tandis que la deuxième est idéale pour la convolution. Bien qu'elles soient équivalentes, nous montrerons que le choix propice de la représentation peut considérablement réduire le calcul.

Représentation par coefficients

La ***représentation par coefficient*** d'un polynôme $A(x) = \sum_{i=0}^{n-1} a_i x^i$ est le vecteur de coefficients $a = (a_0, a_1, \cdots, a_{n-1})$. Il s'agit d'un vecteur colonne dans une matrice. Cette représentation est commode pour certaines opérations sur les polynômes telle que l'*évaluation* de $A(x)$ en un point x_0 qui pend un temps en $O(n)$, en particulier si l'on utilise la règle de Horner :

$$p(x) = a_0 + a_1 x + \ldots + a_n x^n \tag{3.5}$$

$$= a_0 + x(a_1 + x(a_2 + \ldots + x(a_{n-1} + x(a_n)) \ldots)) \tag{3.6}$$

De même, l'addition de deux polynômes représentés par leurs vecteurs de coefficients $a = (a_0, a_1, \cdots, a_{n-1})$ et $b = (b_0, b_1, \cdots, b_{n-1})$ s'accomplit en $O(n)$ en appliquant l'équation 3.20. Par contre, la multiplication de deux polynômes $A(x)$ et $B(x)$ prend $O(n^2)$ puisque chaque coefficient de a doit être multiplié par chaque coefficient de b comme illustré dans l'équation 3.4. Ainsi, la représentation par coefficient semble inapropriée pour la multiplication des polynômes qui est vraisemblablement une opération beaucoup plus complexe que l'évaluation ou l'addition de polynômes.

Représentation par valeurs

La ***représentation par valeurs*** d'un polynôme $A(x) = \sum_{i=0}^{n-1} a_i x^i$ est un ensemble de n points du plan, représentés par leurs coordonnées

$$\{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})\} \tag{3.7}$$

tels que les x_k sont tous *distincts* et $y_k = A(x_k)$ pour $k = 0, 1, \cdots, n-1$. Nous soulignons qu'un même polynôme peut avoir plusieurs représentations par valeurs puisqu'il suffit de choisir un ensemble de n points distincts $\{x_0, x_1, \cdots, x_{n-1}\}$ et de calculer ensuite les $A(x_k)$ correspondants. Cette évaluation requiert $O(n^2)$ en utilisant la méthode de Horner. Néanmoins, nous montrerons un peu plus loin que la combinaison de cette représentation avec un choix pertinent des x_k peut réduire

considérablement ce temps à $O(n \lg n)$. Cette représentation est efficace pour des nombreuses opérations sur les polynômes, y compris la multiplication. Supposons que les représentations par valeurs de A et B sont respectivement

$$\{(x_0, y_0^a), (x_1, y_1^a), \cdots, (x_{n-1}, y_{n-1}^a)\},$$

$$\{(x_0, y_0^b), (x_1, y_1^b), \cdots, (x_{n-1}, y_{n-1}^b)\}$$

Si $C(x)$ est le polynôme produit de $A(x)$ et $B(x)$, alors $C(x_k) = A(x_k)B(x_k)$ pour tous les x_k. Pourtant, nous rencontrons le problème de la borne de C, qui est la somme des bornes de A et B comme convenu dans la section 3.1.2. En multipliant A et B, on obtient n coordonnées pour C, mais puisque la borne de C est $2n$, alors $2n$ coordonnées sont nécessaires pour la représentation par valeurs de C, d'après le théorème 3.1.3.1 décrit un peu plus loin. Il faut donc partir des représentations *étendues*, composées de $2n$ coordonnées, pour A et B. Considérons à present la représentation étendue de A et de B

$$\{(x_0, y_0^a), (x_1, y_1^a), \cdots, (x_{2n-1}, y_{2n-1}^a)\}, \tag{3.8}$$

$$\{(x_0, y_0^b), (x_1, y_1^b), \cdots, (x_{2n-1}, y_{2n-1}^b)\}. \tag{3.9}$$

Notons que A et B sont évaluées pour les mêmes $2n$ *abscisses*. La représentation étendue par valeurs de C sera alors

$$\{(x_0, y_0^a y_0^b), (x_1, y_1^a y_1^b), \cdots, (x_{2n-1}, y_{2n-1}^a y_{2n-1}^b)\} \tag{3.10}$$

Ainsi, la multiplication des polynômes données sous leur forme étendue par valeurs se fait en temps linéaire $O(n)$ seulement comparativement à un temps en $O(n^2)$ requis pour les polynômes donnés par leurs coefficients. Néanmoins, cette représentation n'a pas de méthode efficace pour évaluer un polynôme en un nouveau point. La méthode la plus simple consiste à calculer les coefficients du polynôme, par interpolation, puis l'évaluer au point voulu. Hélas, cette méthode peut s'effec-

tuer en un temps $O(n^2)$ comme nous le verrons dans la section suivante. Toutefois, des algorithmes d'interpolation plus efficaces existent.

Interpolation

L'interpolation d'un polynôme est le processus inverse de l'évaluation. Elle consiste à déterminer les coefficients d'un polynôme à partir de sa représentation par valeurs.

Le théorème ci-dessous, énoncé formellement dans le livre de Cormen et al. [CLR94], garantit l'existence et l'unicité du polynôme d'interpolation. Bien que ce théorème ne s'applique pas nécessairement à $\mathbf{Z_m}$, il peut être utilisé dans notre contexte de la FFT comme Brassard et Bratley l'ont fait dans la section 9.1 de leur ouvrage [BB88] en autant que les conditions générales soient respectées. Ces conditions seront rappelées dans la section 3.4.1.

Théorème 3.1.3.1 (Unicité d'un polynôme d'interpolation). *Un ensemble x_i distincts de n coordonnées, $\{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})\}$, a un et un seul polynôme $A(x)$ de borne n tel que $y_k = A(x_k)$ pour $k = 0, 1, \cdots, n-1$.*

L'interprétation matricielle de l'interpolation sur n points est la solution du systèmes $a = V^{-1}y$, où V est une matrice carrée inversible appelée matrice de *Vandermonde*. L'algorithme le plus rapide pour l'interpolation sur n points est basée sur la formule de *Lagrange* :

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k}(x - x_j)}{\prod_{j \neq k}(x_k - x_j)} \tag{3.11}$$

Cet algorithme peut calculer les coefficients de A en $O(n^2)$. Ainsi, l'évaluation et l'interpolation pour n points sont des opérations inverses bien définies et les algorithmes décrits jusqu'à présent les calculent en $O(n^2)$. La question naturelle qui se pose maintenant est : *est-il possible de profiter de la multiplication en temps linéaire, lorsque les polynômes sont représentés par valeurs, pour multiplier des polynômes représentés par coefficients ?* Nous répondrons à cette question dans la section suivante.

Multiplication rapide de polynômes

Dans la représentation par coefficients, l'évaluation d'un polynôme requiert un temps en $O(n)$ mais la multiplication de polynômes prend un temps en $O(n^2)$, comme illustré dans la table 3.1. Par contre, la représentation par valeurs offre la possibilité de réaliser cette multiplication en $O(n)$, mais l'évaluation d'un polynôme requiert un temps en $O(n^2)$ si on se sert de l'interpolation de *Lagrange*. En conclu-

Représentation par	**Multiplication**	**Évaluation**	**Total**
Coefficient	$O(n^2)$	$O(n)$	$O(n^2)$
Valeur	$O(n)$	$O(n^2)$	$O(n^2)$

TAB. 3.1 – Complexité de multiplication des polynômes

sion, la possibilité de profiter de la multiplication en temps linéaire, lorsque les polynômes représentés par valeurs, pour multiplier des polynômes représentés par coefficients, est déterminée par la méthode de conversion de la représentation par coefficients à la forme par valeurs (évaluation) et vice versa (interpolation). Heureusement, la FFT offre l'outil efficace pour accomplir cette conversion en $O(n \lg n)$ seulement en choisissant les racines $n^{\text{ièmes}}$ de l'unité comme points d'évaluations. La figure 3.1 montre le processus de la multiplication des polynômes en utilisant la FFT. L'évaluation d'un polynôme n'est que la FFT de son vecteur de coefficients a. Par contre, l'interpolation est la FFT inverse de son vecteur de coordonnées.

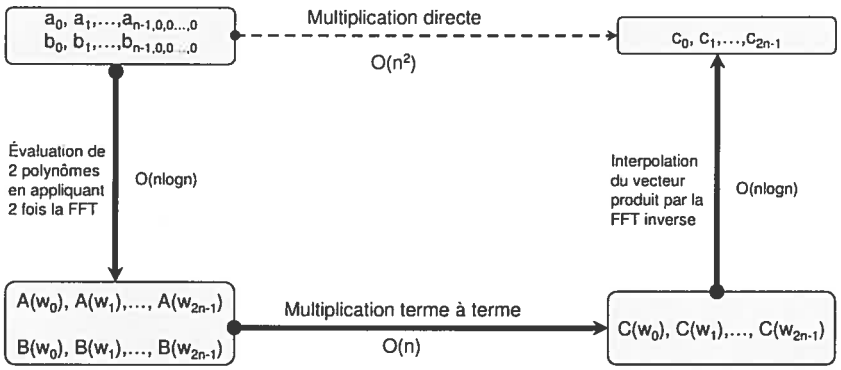

FIG. 3.1 – Schéma de la multiplication de polynômes par la FFT

3.2 Transformée de Fourier

La transformée de Fourier d'une fonction intégrable $f(t)$ est la fonction $F(\omega)$ tel que :

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{i\omega t}\, dt$$

et la transformée inverse de Fourier est

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)e^{-i\omega t}\, d\omega$$

Notons que $i^2 = -1$ et $e^{i\theta} = \cos\theta + i\sin\theta$. Si $f(t)$ est considéré comme étant un signal (données d'entrée) alors $F(\omega)$ est appelée le spectre du signal.

3.3 Transformée de Fourier discrète (DFT)

Dans le monde des systèmes numériques où les signaux sont discrets et périodiques nous n'avons pas besoin de la transformée de Fourier continue. Nous employons plutôt la transformée de Fourier discrète (DFT pour **D**iscrete **F**ourier **T**ransform). La **DFT** d'un vecteur de n composants $a = \{a_0, a_1, \cdots, a_{n-1}\}$ par rapport à une constante ω, notée $DFT(a)$, est un nouveau vecteur $A = \{A_0, A_1, \cdots, A_{n-1}\}$ tel que :

$$A_j = \sum_{k=0}^{n-1} a_k \omega^{jk},\ 0 \leq j \leq n-1 \tag{3.12}$$

où $\omega = e^{i\frac{2\pi}{n}}$, appélée *la racine $n^{ième}$ primitive (ou principale) de l'unité*, et les valeurs ω^k pour $k = 0, \ldots, n-1$ sont appélées *les racines $n^{ièmes}$ de l'unité* dont nous parlerons en détails dans la section 3.4.1.

3.3.1 Interprétation polynomiale de la DFT

Pensant au vecteur a en tant que représentation par coefficients d'un polynôme $p(x) = \sum_{i=0}^{n-1} a_i x^i$, le calcul de $DFT(a)$ est équivalent à évaluer $p(x)$ aux points

$x_i = \omega^i$ pour $0 \leq i \leq n-1$, plus précisément le vecteur

$$A = DFT(a) = \{p(1), p(\omega), p(\omega^2) \cdots, p(\omega^{n-1})\} \tag{3.13}$$

Pour calculer un seul A_i en utilisant l'équation 3.12, il nous faut n multiplications et $n-1$ additions. Alors, pour calculer tous les A_i pour $i = 0 \ldots n-1$, ceci semble requérir n^2 multiplications et $n(n-1)$ additions. Il semble, à première vue, que le calcul de la DFT s'éxecute en $O(n^2)$. Cependant, ce temps peut être ramené à $O(n \lg n)$ grâce à la transformée de Fourier rapide qui se base sur la stratégie "*diviser-pour-régner*" et tire profit des propriétés des racines $n^{\text{ièmes}}$ de l'unité.

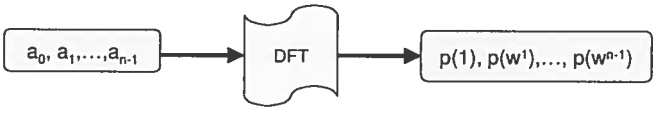

FIG. 3.2 – Interprétation polynomiale de la FFT

3.3.2 Interprétation matricielle de la DFT

La DFT peut être interprétée comme étant une transformation linéaire

$$A = M_n(\omega) \cdot a$$

où les deux vecteurs A et a sont deux matrices colonnes et $M_n(\omega)$ est la matrice carrée $n \times n$ dont l'entrée i, j est ω^{ij}.

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ \vdots \\ A_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

Lorsque ω et n vérifient certaines générales que nous allons en discuter dans la section 3.4.1, nous pouvons montrer que cette transformation linéaire est inversible

et que

$$(M_n(\omega))^{-1} = \frac{1}{n} \cdot M_n(\omega^{-1}).$$

Ceci permettra de calculer la DFT inverse définie prochainement. Nous soulignons que les ω^{ij} se simplifient tellement, comme le montre l'exemple 3.3.3, grâce aux propriétés suivantes : $\omega^{\frac{n}{2}} = -1$ et $\omega^{n+k} = \omega^k$ $(k \geq 0)$.

3.3.3 Exemples de la DFT

Exemple : DFT à deux points ($n = 2$)

Pour $n = 2$, $\omega = e^{i\frac{2\pi}{n}} = e^{i\pi} = -1$, et

$$A_j = \sum_{k=0}^{1} a_k(-1)^{jk} = (-1)^{k\cdot 0}a_0 + (-1)^{k\cdot 1}a_1 = a_0 + (-1)^k a_1$$

Alors,

$$A_0 = a_0 + a_1$$
$$A_1 = a_0 - a_1$$

$$A = DFT(a) = (A_0, A_1)$$

Exemple : DFT à quatre points ($n = 4$)

Pour $n = 4$, $\omega = e^{i\frac{2\pi}{n}} = e^{i\frac{\pi}{2}} = -i$, et

$$\begin{aligned} A_j &= \sum_{k=0}^{3} a_k(-i)^{jk} \\ &= a_0 + (-i)^k a_1 + (-i)^{2k} a_2 + (-i)^{3k} a_3 \\ &= a_0 + (-i)^k a_1 + (-1)^k a_2 + (i)^k a_3 \end{aligned}$$

Alors,

$$\begin{aligned} A_0 &= a_0 + a_1 + a_2 + a_3 \\ A_1 &= a_0 - ia_1 - a_2 + ia_3 \\ A_2 &= a_0 - a_1 + a_2 - a_3 \\ A_3 &= a_0 + ia_1 - a_2 - ia_3 \end{aligned}$$

Ceci est équivalent à la multiplication des matrices suivante

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

Observons que les ω^{ij} se sont simplifiés à deux valeurs simples très faciles à gérer, il n'y a que des i et -1 partout. Ceci offre toujours la possibilité de regrouper ces équations afin d'obtenir des expressions plus simples :

$$\begin{aligned} A_0 &= (a_0 + a_2) + (a_1 + a_3) \\ A_1 &= (a_0 - a_2) - i(a_1 - a_3) \\ A_2 &= (a_0 + a_2) - (a_1 + a_3) \\ A_3 &= (a_0 - a_2) + i(a_1 - a_3) \end{aligned}$$

Ceci pourrait réduire jusqu'à 50% les opérations d'additions, comme c'est le cas dans cet exemple ; nous avons 8 additions qui peuvent être réduites à 4 seulement. Une autre avantage de ce regroupement est la possibilité de calculer A encore plus rapidement en pré-calculant ces expressions communes mais cette fois au prix d'un petit espace mémoire additionnel.

3.3.4 DFT inverse

Étant donné un vecteur de n composants A tel que $A = DFT(a)$, le vecteur original a peut être récupéré par la DFT inverse. La DFT inverse d'un vecteur de n composants A par rapport à une constante ω, notée DFT^{-1}, est un vecteur $a = \{a_0, a_1, \cdots, a_{n-1}\}$ tel que :

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} A_k \omega^{-jk},\ 0 \leq j \leq n-1 \tag{3.14}$$

Ainsi

$$DFT^{-1}(DFT(a)) = a \tag{3.15}$$

Puisque ω est une racine $n^{\text{ième}}$ de l'unité, ω^{-1} l'est aussi. Ainsi, l'équation 3.14 est justement une DFT excepté la division par n. Donc, l'algorithme de la DFT^{-1} est le même que celui de la DFT. Il suffit de remplacer ω^k par son inverse ω^{-k} et diviser tous les éléments du tableau par n lorsque le calcul est terminé.

Théorème 3.3.4.1 (Théorème de Convolution). *Soient a et b deux vecteurs de n éléments où n est une puissance de 2. La convolution de a et b est la DFT inverse du produit de leurs transformées de Fourier.*

$$a \otimes b = DFT_{2n}^{-1}(DFT_{2n}(a) \cdot DFT_{2n}(b)) \tag{3.16}$$

Notons que l'opérateur $(\cdot)$ signifie le produit terme à terme de deux vecteurs tandis que $(\otimes)$ représente leur convolution. Nous rappelons aussi que les vecteurs a et b

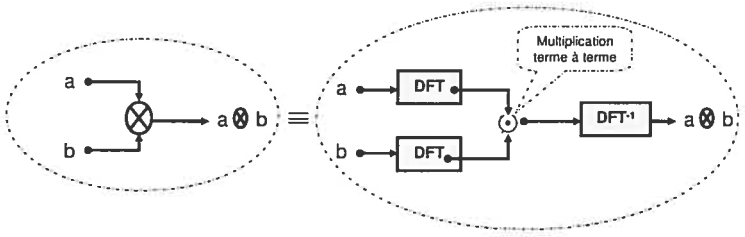

FIG. 3.3 – Schéma du théorème de la convolution par la FFT

doivent être complétés avec des 0 jusqu'à la longueur $2n$ avant de commencer le calcul. Ce théorème peut être schématisé par la figure 3.3 qui récapitule la procédure de la convolution par la FFT.

3.4 La transformée de Fourier rapide (FFT)

La transformée de Fourier rapide (FFT pour **F**ast **F**ourier **T**ransform) est simplement un algorithme permettant de réduire considérablement le nombre d'opérations pour calculer la DFT. Cet algorithme se base sur la stratégie *diviser-pour-régner* et tire profit des propriétés particulières des racines $n^{\text{ièmes}}$ de l'unité (périodicité et symétrie).

C'est en 1965 que James Cooley et John Tukey publient cette méthode. Pourtant, il a été découvert par la suite que l'algorithme avait déjà été inventé par Carl Friedrich Gauss en 1805 et adapté à plusieurs reprises sous des formes différentes. Jusqu'à cette date, la complexité de la DFT de n points était basée sur l'équation 3.12. C'est une multiplication de matrices qui exigeait n^2 multiplications complexes et $n(n-1)$ additions complexes même avec l'algorithme le plus rapide, plus un plus petit nombre d'opérations pour calculer les puissances de ω. Néanmoins, avec la FFT de Cooley-Tukey, le nombre d'opérations peut être réduit à $\frac{n}{2}\log n$. La différence entre $\frac{n}{2}\log n$ et n^2 est immense pour un n assez grand.

Nous soulignons que d'autres versions de la FFT existent et requièrent d'autres exigences (n premier, n n'est pas une puissance de 2), mais nous ne considérerons ici que celle de l'algorithme original de Cooley-Tukey sur lequel celles-ci se basent.

3.4.1 Racines $n^{\text{ièmes}}$ de l'unité

Définition 3.4.1.1. *Une constante ω est dite racine $n^{\text{ième}}$ de l'unité si $\omega^n = 1$. En outre, ω est dite racine $n^{\text{ième}}$ principale (ou primitive) de l'unité si ω satisfait les deux conditions supplémentaires suivantes :*

1. $\omega^k \neq 1$ *pour* $0 < k < n$
2. $\sum_{j=0}^{n-1} \omega^{jk} = 0$ *pour* $0 < k < n$

Lorsque n est une puissance de deux, il en résulte que ces conditions sont automatiquement vérifiées pour tout $\omega^{\frac{n}{2}} = -1$ [BB88]. Le fait que les racines $n^{\text{ièmes}}$ de l'unité sont à la base de la transformée de Fourier, nous allons citer le théorème ci-dessous, énoncé dans le livre de G. Brassard et P. Bratley, afin que le lecteur puisse facilement trouver tout ce qui est nécessaire à la compréhension de notre travail.

Théorème 3.4.1.2 (Racines $n^{\text{ièmes}}$ de l'unité). *On considère un anneau commutatif quelconque. Soit $n > 1$ un entier de puissance de deux et soit ω un élément dans cet anneau tel que $\omega^{\frac{n}{2}} = -1$. Alors*

1. *ω est une racine $n^{ième}$ principale de l'unité.*
2. *ω^{-1} est l'inverse multiplicatif de ω dans l'anneau.*
3. *ω^{-1} est aussi une racine $n^{ième}$ principale de l'unité.*
4. *$1 = \omega^0, \omega^1, \cdots, \omega^{n-1}$, appelés les racines $n^{ièmes}$ de l'unité, sont tous distincts.*
5. *Supposant qu'il existe un inverse multiplicatif n^{-1} de n dans notre anneau, alors ω^{-1} est conséquence du fait que ω est une racine $n^{ième}$ principale de l'unité.*

Il existe ainsi exactement n racines $n^{\text{ièmes}}$ de l'unité, $\{\omega^0, \omega^1, \cdots, \omega^{n-1}\}$, qui sont toutes des puissances de ω. L'essentiel est alors de trouver une racine $n^{\text{ième}}$ principale de l'unité à partir de laquelle seront générées les n racines. Nous présentons tout de suite les propriétés essentielles des racines $n^{\text{ièmes}}$ de l'unité, tiré du livre de T. Cormen et al., qui nous servirons un peu plus loin.

Propriété 3.4.1.3 (Propriété de Périodicité). *Pour tous entiers $n \geq 0$ et $k \geq 0$,*

$$\omega^{n+k} = \omega^k.$$

Propriété 3.4.1.4 (Propriété de Symétrie). *Pour tout entier n pair,*

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

Propriété 3.4.1.5 (Propriété de bipartition). *Pour tout n pair, les carrés des n racines $n^{ièmes}$ de l'unité sont les $\frac{n}{2}$ racines $(\frac{n}{2})^{ièmes}$ de l'unité.*

Exemple : Racines $n^{\text{ièmes}}$ de l'unité dans C

L'exemple pratique le plus utilisé est celui des nombres complexes. Pourquoi les nombres complexes ? Parce que dans les nombres réels, il existe une seule racine $n^{\text{ième}}$ réelle ($\omega = +1$) si n est impair et deux racines si n est pair ($\omega = \pm 1$).

Contrairement au **R**, il existe dans **C** exactement n racines $n^{\text{ièmes}}$ distinctes de l'unité, dont $+1$. D'après la définition, les racines $n^{\text{ièmes}}$ de l'unité sont toutes de module 1, mais d'angles différents.

Considérons le nombre complexe ω dont le module est 1 et l'angle est $\frac{2\pi}{n}$. La forme exponentielle de ω est $e^{i\frac{2\pi}{n}}$. Ainsi, $\omega^n = (e^{i\frac{2\pi}{n}})^n = e^{i2\pi} = 1$. Pareillement, nous pouvons trouver les autres racines. Pour un entier $0 < k < n$, $(\omega^k)^n = \omega^{kn} = (\omega^n)^k = (1)^k = 1$. D'autre part, ω est une racine $n^{\text{ième}}$ principale de l'unité ($\omega^{\frac{n}{2}} = -1$). En effet, pour cette valeur de ω ($e^{i\frac{2\pi}{n}}$), les points $\omega^0, \omega^1, \cdots, \omega^{n-1}$ sont tous différents. Ils sont de module 1, mais ont un angle $\frac{2\pi k}{n}$. Il existe ainsi exactement n racines $n^{\text{ièmes}}$ de l'unité qui sont des puissances de ω et valent $\omega = e^{i\frac{2\pi k}{n}}$ pour $k = 0, 1, \cdots, n-1$. Géométriquement, ces racines représentent les sommets d'un polygone régulier à n côtés inscrit dans le cercle de rayon unitaire centré à l'origine du plan complexe.

Exemple : Racines $n^{\text{ièmes}}$ de l'unité dans Z_m

Ce travail porte sur la multiplication des grands entiers en utilisant la FFT. Dans **C**, chaque opération "*papillon*"[2] consiste en une multiplication et deux additions complexes, c'est équivalent à quatre multiplications et six additions réelles. Dans notre approche, cela ne serait pas un bon choix puisque le calcul en **C** exige la manipulation de nombres rationnels en précision finie alors que la multiplication de grands nombres est en précision infinie. Une autre possibilité consiste à utiliser

[2] C'est l'opération de base de la FFT. Nous allons en parler en détails dans la section 3.5.1.

l'arithmétique modulaire ($\mathbf{Z}_m$). Un papillon se restreint simplement à une multiplication et deux additions modulaires qui s'effectuent efficacement grâce à la forme *deux-à-la-puissance*. Néanmoins, la FFT en arithmétique modulaire requiert des racines $n^{\text{ièmes}}$ de l'unité dans $\mathbf{Z}_m$.

Afin de concrétiser l'idée de ce paragraphe, nous nous satisfaisons ici d'un simple exemple qui vérifie les conditions générales permettant de calculer telles racines, et laissons les détails à la section concernée. Notons que cet exemple a été testé et validé comme le montre la section 3.6.3.

Mettons que $\omega = 8$ et $m = 4\,097$, alors ω est la racine $8^{\text{ième}}$ principale de l'unité dans $\mathbf{Z}_{4097}$ puisque $\omega^{\frac{n}{2}} = \omega^4 \equiv -1 \mod 4\,097$ d'après le théorème 9.3.1 [BB88]. En effet, ceci est montré par le calcul qui suit :

ω	$\equiv 8$	$\mod 4\,097$	ω^2	$\equiv 64$	$\mod 4\,097$
ω^3	$\equiv 512$	$\mod 4\,097$	ω^4	$\equiv 4096$	$\mod 4\,097$
ω^5	$\equiv 4089$	$\mod 4\,097$	ω^6	$\equiv 4033$	$\mod 4\,097$
ω^7	$\equiv 3585$	$\mod 4\,097$	ω^8	$\equiv 1$	$\mod 4\,097$

3.4.2 Algorithme de la FFT (Cooley-Tukey)

L'algorithme de la FFT le plus connu est celui de Cooley-Tukey. Cet algorithme requiert un nombre d'échantillons qui soit une puissance de 2. Nous supposons tout au long de ce travail que n est une puissance de deux sans perte de généralité puisqu'il est toujours possible d'ajouter de nouveaux coefficients de valeur nulle si nécessaire.

L'algorithme de la FFT se base sur la décomposition successive d'un vecteur d'entrée de taille n, en deux autres vecteurs identiques de plus petite taille, ce qui correspond à la stratégie "diviser-pour-régner". C'est la raison pour laquelle que l'algorithme de Cooley-Tukey est appelé parfois algorithme de réduction à base binaire dans le domaine temporel (*decimation-in-time*).

Cette décomposition est aussi combinée avec certaines propriétés des racines $n^{\text{ièmes}}$ de l'unité pour offrir une très bonne efficacité.

Stratégie diviser-pour-régner :

Considérons les deux nouveaux polynômes $p^0(x)$ et $p^1(x)$ de degré $(\frac{n}{2}-1)$ en se servant séparément des coefficients d'indices pairs et impairs de $p(x)$.

$$p^0(x) = a_0 + a_2x^2 + \cdots + a_{n-2}x^{\frac{n}{2}-1} \tag{3.17}$$

$$p^1(x) = a_1 + a_3x^3 + \cdots + a_{n-1}x^{\frac{n}{2}-1} \tag{3.18}$$

Remarquons que p^0 contient tous les coefficients de $p(x)$ dont le code binaire de l'indice se termine par 0 (indice pair) et que p^1 contient tous les coefficients de $p(x)$ dont le code binaire de l'indice se termine par 1 (indice impair). Cette décomposition permet la re-écriture de $p(x)$ sous la forme

$$\begin{aligned} p(x) &= (a_0 + a_2x^2 + \cdots + a_{n-2}x^{n-2}) + \\ &\quad (a_1 + a_3x^2 + \cdots + a_{n-1}x^{n-2})x \\ &= p^0(x^2) + xp^1(x^2) \end{aligned} \tag{3.19}$$

de sorte que le problème d'évaluer $p(x)$ en $\{\omega^0, \omega^1, \cdots, \omega^{n-1}\}$ est ramené à

1. évaluer les deux polynômes $p^0(x)$ et $p^1(x)$ aux points

$$(\omega^0)^2, (\omega^1)^2, \cdots, (\omega^{n-1})^2,$$

2. combiner les deux résultats selon l'équation 3.19

Cette nouvelle écriture n'apporte rien de nouveau jusqu'à présent. Pourtant, le choix judicieux des valeurs auxquelles sera évalué $p(x)$ pourrait changer complètement l'image et apporter par la suie une réduction immense du calcul. Par exemple, l'évaluation de $p(x)$ en $x = \pm 1$ nécessite seulement le calcul de $p^0(1)$ et $p^1(1)$ et non pas $p^0(\pm 1)$ et $p^1(\pm 1)$. C'est alors la recherche de nouvelles valeurs ayant telle propriété, que vient l'intérêt des racines $n^{\text{ièmes}}$ de l'unité.

Profit dû aux racines $n^{\text{ièmes}}$ de l'unité :

D'après la propriété de symétrie, $\omega^{\frac{n}{2}} = -1$. Donc, $\omega^{\frac{n}{2}+j} = -\omega^j$ pour tout j. Cette

équation implique qu'il y a $\frac{n}{2}$ paires d'éléments symétriques par rapport à $\omega^{\frac{n}{2}}$. Autrement dit, les valeurs $\{(\omega^0)^2, (\omega^1)^2, \cdots, (\omega^{n-1})^2\}$ ne sont pas n racines distinctes, mais plutôt $\frac{n}{2}$ racines $(\frac{n}{2})^{\text{ièmes}}$ de l'unité où chaque racine apparaît exactement deux fois (*bipartition*). Ainsi, il suffit d'évaluer les deux polynômes $p^0(x)$ et $p^1(x)$ seulement pour les $\frac{n}{2}$ racines $(\frac{n}{2})^{\text{ièmes}}$ de l'unité. Nous avons obtenu ainsi deux sous-problèmes à $\frac{n}{2}$ éléments ayant la même forme que le problème initial, mais qui sont moitié moins grands. Cette décomposition est la base de l'algorithme récursif de la FFT (FFTR) illustré dans l'algorithme 11 qui montre d'une façon assez simple l'idée de la FFT.

Algorithme 11 FFT Récursive (FFTR)

1: Entrée : a, n
2: Sortie : A $\{A = \{p(\omega^0), p(\omega^1), \cdots, p(\omega^{n-1})\}\}$
3: **if** (n=1) **then**
4: Retourner a {La DFT d'un seul élément est l'élément lui-même}
5: **end if**
6: $\omega \leftarrow e^{\frac{2\pi}{n}}$
7: $a^0 \leftarrow \{a_0, a_2, \cdots, a_{n-2}\}$
8: $a^1 \leftarrow \{a_1, a_3, \cdots, a_{n-1}\}$
9: $A^0 \leftarrow FFTR(a^0, \frac{n}{2})$
10: $A^1 \leftarrow FFTR(a^1, \frac{n}{2})$
11: $\alpha \leftarrow 1$
12: $t \leftarrow \frac{n}{2}$
13: **for** $k = 0$ to $t - 1$ **do**
14: $A_k \leftarrow A_k^0 + wA_k^1$
15: $A_{k+t} \leftarrow A_k^0 - wA_k^1$
16: $\alpha \leftarrow \alpha\omega$
17: **end for**
18: Retourner A

Remarquons que la boucle *for* (lignes 13-14) dans l'algorithme 11 calcule la valeur wA_k^o à deux reprises. Cette boucle peut être modifiée légèrement pour la calculer une seule fois, en la sauvegardant dans une variable temporaire. L'action principale de cette boucle est connue sous le nom *opération papillon* qui sera étudiée un peu plus loin.

3.4.3 Complexité de la FFT récursive

La **complexité temporelle** de cet algorithme est donnée par la rélation de récurrence :

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

Le calcul de la FFT pour n éléments nécessite le calcul de 2 FFT pour $\frac{n}{2}$ éléments plus un temps proportionnel à n pour séparer le tableau initial en deux sous tableaux et pour réunir les deux résultats. La résolution de cette récurrence fournit la solution $T(n) = O(n \lg n)$. Donc, l'évaluation d'un polynôme de borne n pourrait être exécute en temps $O(n \lg n)$ à l'aide la transformée de Fourier rapide.

3.5 Architectures efficaces de la FFT

La structure récursive est souvent évitée, surtout dans les implémentations matérielles, pour de raisons de performance et de taille du circuit. Un autre inconvénient de la structure recursive est que le parallélisme y est impossible. Nous étudions ici la procédure pour rendre la structure de l'algorithme FFT iterative.

3.5.1 La FFT itérative

L'idée de la structure itérative de la FFT est justement inspirée de la FFT recursive (FFTR). Plus précisement, elle est basée sur la réorganisation des coefficients du vecteur initial due aux appels récursifs. L'arbre binaire dans la figure 3.4 simule les appels de la FFTR pour un échantillon de 8 points. Chaque appel FFTR est représenté dans l'arbre par un nœud étiqueté avec le vecteur d'entrée correspondant, et fait à son tour deux autres appels avec un vecteur d'entrée de $\frac{n}{2}$ coefficients tant que $n > 1$ (condition d'arrêt). L'analyse de cet algorithme pourrait être divisée en deux observations essentielles :

1. En observant cet arbre, on remarque que les appels récursifs qui démarrent l'algorithme de la FFT ne servent qu'à réorganiser le vecteur des coefficients à chaque niveau dans l'arbre. En effet, avant la condition d'arrêt et avant de

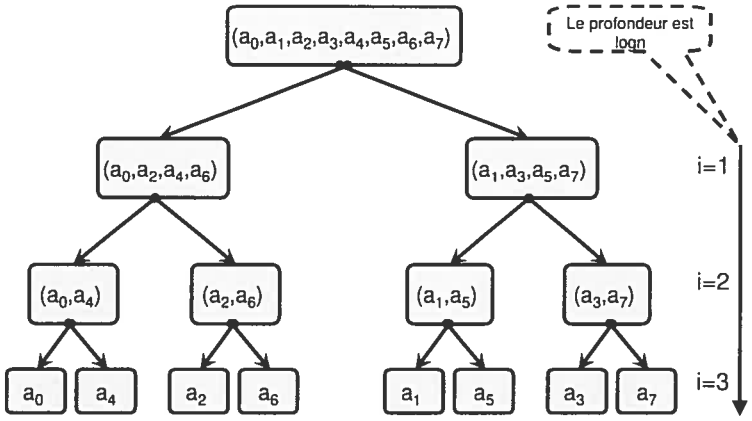

FIG. 3.4 – Arbre de récursivité de la FFT

passer à la boucle **for**, nous aurons obtenu une permutation des coefficients du vecteur initial selon l'ordre d'apparition aux feuilles de l'arbre. Donc, si on peut réorganiser les coefficients du vecteur initial a dans cet ordre, il est possible de calculer la FFT d'une façon itérative. Nous montrerons plus tard la manière de faire cette réorganisation.

2. L'action qui suit les appels récursifs est une action de recombinaison consistant à reconstituer le vecteur résultat à partir des sous-vecteurs obtenus. Il s'agit du parcours de l'arbre précédent des feuilles vers la racine niveau par niveau. À chaque niveau i ($i = 1, \cdots, \log n$) il y a $s = \frac{n}{2^i}$ recombinaisons DFT comme le montre la figure 3.5. Au départ, s vaut $\frac{n}{2}$, on prend les éléments deux par deux, on calcule la DFT de chaque paire en se servant de l'opération papillon et on la remplace par sa DFT. Par exemple a_0, a_4 seront remplacés par $\text{DFT}(a_0, a_4)$, a_2, a_6 seront remplacé par $\text{DFT}(a_2, a_6)$ et ainsi de suite. Le vecteur contient alors $\frac{n}{2}$ DFT à deux éléments. Ensuite, on prend ces $\frac{n}{2}$ combinaisons DFT deux par deux et on fait de même pour obtenir un vecteur de $\frac{n}{4}$ DFT à 4 éléments. On continue de cette manière jusqu'à ce que le vecteur contienne 2 DFT à $\frac{n}{2}$ éléments qui seront combinés à l'aide de $\frac{n}{2}$ opérations papillons pour retourner finalement le DFT à n éléments.

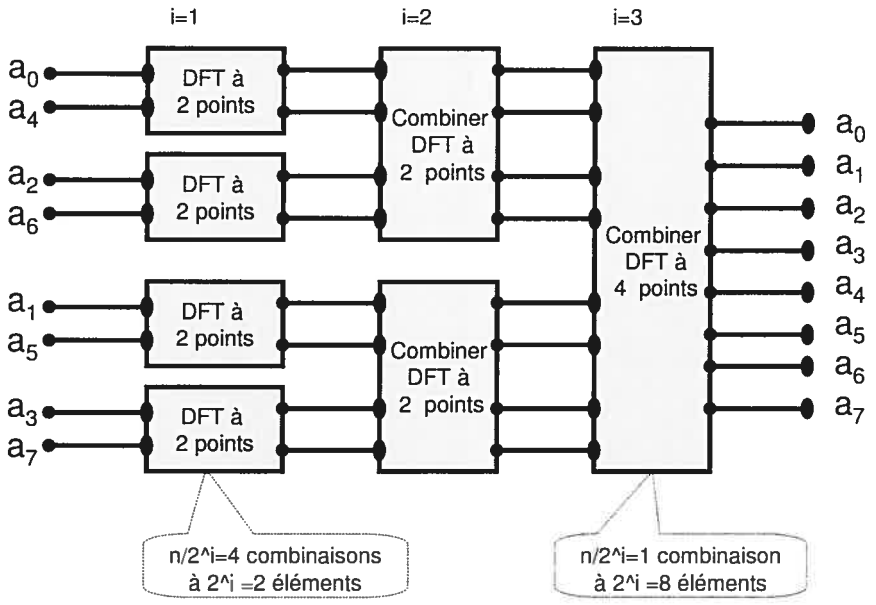

FIG. 3.5 – Simulation de la FFT itérative

Algorithme de la FFT itérative

L'implémentation de la FFT itérative nécessite plusieurs variables. Un tableau $a[0..n]$ qui contient le vecteur d'entrée et un tableau $A[0..n]$ qui contient initialement les éléments permutés du vecteur a ainsi que le résultat final. Le tableau A n'est introduit que pour une meilleure clarté. Un compteur de niveau i est nécessaire puisqu'à chaque niveau il y aura un nombre de combinaisons correspondant. Le compteur i va de 1 (combinaison de $\frac{n}{2}$ DFT à deux éléments) à $\lg n$ (combinaison de 2 DFT à $\frac{n}{2}$ éléments pour retourner le résultat final).

On introduit aussi une variable α qui garantie la mise à jour de la valeur de la racine $n^{\text{ième}}$ pour chaque valeur de i . Cette variable, initialisée à 1 pour chaque niveau, est mise à jour à chaque opération papillon, c.à.d, à chaque itération dans la boucle la plus interne. On utilise aussi une variable ω qui contient la racine $n^{\text{ième}}$ primitive de l'unité ainsi qu'une variable s en vue d'une meilleure lisibilité.

Algorithme 12 FFT Itérative (FFTI)

```
1: Entrée : a, n
2: Sortie : A
3: Mirroir (a, A)
4: for i = 1 to log n do
5:    s ← 2^ℓ    {pas d'incrémentation }
6:    ω ← e^{2πi/s} {racine sième de l'unité}
7:    α ← 1
8:    for j = 0 to s/2 − 1 do
9:       for k = j to n − 1 ; pas de s do
10:         v ← αA[k + s/2]
11:         u ← A[k]
12:         A[k] ← u + v
13:         A[k + s/2] ← u − v
14:         α ← αω
15:      end for
16:   end for
17: end for
18: RetournerA
```

Opération papillon (*Butterfly operation*)

L'opération papillon (Figure 3.6) consiste à prendre deux entrées complexes, p et q, et calculer les deux sorties $p+\alpha q$ et $p-\alpha q$, α étant une constante. Cette opération requiert ainsi une multiplication et deux additions complexes, ce qui est équivalent à 4 multiplications et 6 additions réelles. C'est une opération si fondamentale qu'elle

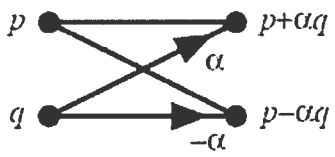

FIG. 3.6 – L'opération papillon

influence la performance de la FFT tout entière puisque cette dernière divise la DFT en $\log_2 n$ étapes chacune consistant en $\frac{n}{2}$ opérations papillons. Un échantillon de 128 points nécessite ainsi 3 072 opérations papillons.

Permutation du vecteur (*bit inverison*)

Il s'agit de la réorganisation des coefficients du vecteur initial a dans l'ordre d'apparition aux feuilles de l'arbre afin que la FFT itérative soit valide. Pour connaître la fonction mystère derrière ce réarrangement, observons la table 3.2. Cette table montre l'ordre initial et final des coefficients du vecteur pour $n = 8$ ainsi que la representation binaire de leur indice. Étant donné que A est le vecteur qui contient

Coefficients	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
	a_0	a_4	a_2	a_6	a_1	a_5	a_3	a_7
Indice i (Binaire)	000	001	010	011	100	101	110	111
mir(i)	000	100	010	110	001	101	011	111

TAB. 3.2 – Réorganisation des coefficients du vecteur initial

initialement les éléments permutés du vecteur a, observons la propriété qui relie les deux tabeaux : $a[i] = A[mir(i)]$ où $mir(i)$ est l'inverse de la représentation binaire de i. On l'appelle le miroir de la représentation binaire de i sur $\log n$ bits. Supposons que $mir(k)$ est la fonction qui retourne l'entier sur $\log n$ bits, en inversant les bits de k, alors il faut permuter l'élément a_k avec l'élément $a_{mir(k)}$. Dans notre exemple, les coefficients sont rangées initialement par ordre croissant de 0 à 7, soient en binaire $000, 001, 010, 011, 100, 101, 110, 111$. Si l'on inverse les bits de cette suite, on obtient la suite $000, 100, 010, 110, 001, 101, 011, 111$; soit en décimal la suite $0, 4, 2, 6, 1, 5, 3, 7$ qui en fait l'ordre dans lequel apparaissent les feuilles de l'arbre. Cette permutation peut être réalisée par la simple procédure suivante. Le pseudo-code de cette procédure se trouve dans l'annexe B.

Algorithme 13 Permutation par inversion de bits

1: Entrée : a, n
2: Sortie : A
3: **for** $k = 0$ to $n - 1$ **do**
4: $\quad A[mir(k)] = a_k$
5: **end for**
6: Retourner A

Complexité

La complexité de la FFT itérative est déterminé à partir du nombre d'itarations $T(n)$ de la boucle la plus interne (lignes 10-13) :

$$\begin{aligned} T(n) &= \sum_{i=1}^{\lg n} \sum_{j=0}^{2^{i-1}-1} \frac{n}{2^i} = \sum_{i=1}^{\lg n} \frac{n}{2^i} \cdot 2^{i-1} \\ &= \sum_{i=1}^{\lg n} \frac{n}{2} = O(n \lg n) \end{aligned}$$

3.5.2 La FFT Parallèle

Outre que la FFT itérative soit une alternative pratique et efficace à la FFT récursive, elle est aussi à la base d'une structure parallèle encore plus efficace. Tout comme le circuit de la FFTI, le circuit parallèle commence avec une étape de permutation des entrées par inversion de bits comme décrit auparavant. Cette permutation est ensuite suivie de $\log n$ étapes dont chacune est constituée de $\frac{n}{2}$ opérations papillons exécutées en parallèle. Ce circuit imite exactement la FFTI. La seule différence est que chaque itération de la boucle **for** la plus externe effectue $\frac{n}{2}$ opérations papillons independentes exécutées en parallèle. La valeur de i dans chaque itération de cette boucle correspond à une étape (palier) d'opérations papillon. En outre, à l'intérieur d'un même palier i ($i = 1, \cdots, \log n$), on a $\frac{n}{2^i}$ groupes de papillons, chacun étant de 2^{i-1} papillons. Notons que le nombre de groupes correspond à la valeur de k dans la FFTI et le nombre de papillons dans chaque groupe correspond à la valeur de j. Par exemple, au départ ($i = 1$) on a 4 groupes à un papillon, puis ($i = 2$), on a 2 groupes à deux papillons et ainsi de suite.

3.6 La FFT en $\mathbf{Z}_m$

Nous sommes finalement arrivées au point essentiel de cette recherche, l'utilisation de la FFT en arithmétique modulaire comme étant une alternative plus performante et moins coûteuse côté ressources matérielles, pour la multiplication des grands

entiers.

La FFT que nous avons vue jusqu'à présent impose l'utilisation des nombres complexes qui souffrent de deux défauts principaux pour la multiplication des entiers. Le premier est que ces calculs pourraient avoir des valeurs inexactes dues aux erreurs d'arrondissement. Quoiqu'en pratique l'erreur soit trop petite, ceci ne convient à pas certains problèmes qui exigent des valeurs exactes. Par exemple, la multiplication des polynômes à coefficients entiers exige un polynôme produit à coefficients entiers. La multiplication de deux entiers impose que la réponse soit un entier. Il est toujours possible d'utiliser l'approche FFT à racines complexes donnée par la formule 3.12 et arrondir la réponse finale afin d'obtenir la valeur exacte. Néanmoins, ceci requiert un grand effort en déterminant la précision appropriée. Trop peu d'exactitude mènera à une réponse erronée et trop d'exactitude induit un temps de calcul excessif.

Le deuxième défaut de l'utilisation des nombres complexes pour la multiplication des entiers est qu'ils exigent des calculs additionnels considérables. La multiplication de deux entiers implique 3 FFT, sans compter les 2 multiplications additionnels que requirent la FFT inverse. Chaque FFT requiert $\frac{n}{2}\log n$ opérations papillons. En $\mathbf{C}$, chaque papillon consiste en une multiplication et deux additions complexes, ce qui est équivalent à quatre multiplications et six additions réelles. Tandis qu'en $\mathbf{Z}_m$, un papillon se restreint simplement à une multiplication et deux additions modulaires qui s'effectuent efficacement grâce à la forme *deux-à-la puissance*. Considérons à titre d'exemple le cas de la multiplication de deux entiers de 4 096 bits. Si nous utilisons la base 32, nous obtenons ainsi 128 coefficients, ce qui requiert 3 072 opérations papillons par la FFT. Ceci implique un total de 12 288 multiplications et 18 432 additions réelles en $\mathbf{C}$ et seulement 3 072 multiplications et 6 144 additions modulaires dans $\mathbf{Z}_m$. Cette réduction remarquable de calcul est déjà un avantage important pour la FFT en arithmétique modulaire.

Souvent, on se sert de la FFT comme d'une étape intermédiaire pour calculer la convolution ; la FFT est un outil et non pas un but. Ce qui permet d'employer n'importe quelle définition de ω à condition que le ω choisi ait certaines des pro-

priétés des racines $n^{\text{ièmes}}$ de l'unité. De là est venue l'idée d'utiliser la FFT basée sur l'arithmétique modulaire qui donne des valeurs exactes, demande moins de ressources et calcule plus rapidement. Cette méthode a été introduite par Schönhage et Strassen en 1972 [SS71]. Notons que l'algorithme de la FFT modulaire est le même que celui de la FFT complexe, l'un faisant tous ses calculs dans $\mathbf{Z}_m$ et l'autre dans $\mathbf{C}$. Il nous suffit ainsi de décrire comment calculer le modulo m et les racines $n^{\text{ièmes}}$ de l'unité dans $\mathbf{Z}_m$, ceci étant à la base de la FFT modulaire.

Calcul du modulo m :

Le modulo m doit satisfaire les deux conditions suivantes :

- m doit être impair. Cette condition est nécessaire pour la FFT inverse. Une division par n dans l'algorithme original sera remplacée par une multiplication par son inverse modulaire. Pour que n ait une inverse dans $\mathbf{Z}_m$, il faut que m et n soient premiers entre eux. Il est nécessaire et suffisant que m soit impair puisque n est une puissance de deux.
- m doit être assez grand pour que la réponse à notre problème soit égale à la réponse modulo m. Par exemple, le calcul de la convolution des deux suites de n entiers positifs exige un $m > nM^2 = L$, où M est la valeur maximale que peuvent atteindre les éléments de deux suites. D'autre part, m ne doit pas être trop grand, car plus m est grand, plus le calcul prend de temps, en particulier si m est si grand qu'il remplisse plusieurs mots machine. Par exemple, pour calculer la convolution de deux suites $a = \{1, 2, 3, 4\}$ et $b = \{5, 6, 7, 8\}$, $m = 257 > 4 \cdot 8^2$ est déjà suffisant.

Racines $n^{\text{ièmes}}$ de l'unité dans $\mathbf{Z}_m$:

Pour compléter la discussion à propos de la FFT modulaire, on a besoin de considérer ω, la racine $n^{\text{ième}}$ primitive de l'unité, et montrer comment la calculer. On peut attribuer à ω n'importe quel entier de puissance de deux à condition qu'on attribue $\omega^{\frac{n}{2}} + 1$ à m, ce qui garantie la condition $\omega^{\frac{n}{2}} \equiv -1 \mod m$. Si $\omega = 2$, par exemple, cela assure une valeur assez grande pour m, on l'accepte sinon on essaye une valeur plus grande telle que $4, 8, \cdots, 2^\ell$. Cette valeur pour m vérifie toutes les propriétés nécessaires au bon fonctionnement de a FFT :

- $m = \omega^{\frac{n}{2}} + 1 \Rightarrow \omega^{\frac{n}{2}} + 1 \equiv 0 \mod m \Rightarrow \omega^{\frac{n}{2}} \equiv -1 \mod m$.
- $\omega^n = \omega^{\frac{n}{2}}\omega^{\frac{n}{2}} \equiv (-1)(-1) \mod m \equiv 1 \mod m \Rightarrow \omega^n \equiv 1 \mod m$.
- l'inverse de ω est ω^{n-1} puisque $ww^{n-1} \equiv \omega^n \equiv 1 \mod m$.
- Puisque n est pair (une puissance de deux) et m est impair, alors m et n sont premiers entre eux. Ainsi, n est inversible dans $\mathbf{Z}_m$.

3.6.1 Procédure générale de la FFT en $\mathbf{Z}_m$

La procédure de la FFT en $\mathbf{Z}_m$ consiste à :

- Trouver une borne supérieur L des calculs du problème, par exemple nM^2 pour les vecteurs.
- Choisir ω pour être la plus petite puissance de 2 telle que $\omega^{\frac{n}{2}} + 1 > L$
- Choisir $m = \omega^{\frac{n}{2}} + 1$.
- Calculer ω^{-1} et n^{-1}, qui seront utilisés dans la DFT inverse.

Maintenant, on peut appliquer la FFT pour obtenir la représentation par valeur (évaluation) et appliquer la FFT inverse pour récupérer la représentation par coefficient (interpolation).

3.6.2 Application : multiplication des polynômes

Soient $A(x) = \sum_{i=0}^{n-1} a_i x^i$ et $B(x) = \sum_{i=0}^{n-1} b_i x^i$ deux polynômes de borne n à coefficients entiers. Leur produit $C(x)$ un polynôme de borne $2n$ tel que :

$$C(x) = \sum_{i=0}^{2n-1} c_i x^i, \tag{3.20}$$

où $c_i = \sum_{k=0}^{i} a_k b_{i-k}$, et $c = a \otimes b$. Ainsi, calculer le produit de deux polynômes implique la convolution de leurs vecteurs de coefficients qui se fait en $O(n \lg n)$ en appliquant la procédure suivante, nous l'appellerons procédure de la multiplication des polynômes donnés par leurs coefficients :

1. Doubler la taille : Cette première étape consiste à créer des représentations par coefficients de $A(x)$ et $B(x)$ comme étant des polynômes de borne $t = 2n$, en

ajoutant à chacun n coefficients nuls dans aux supérieurs (padding with zeros) :

$$\begin{aligned} a &= \{a_0, a_1, \cdots, a_{n-1}, 0, 0, \cdots, 0\} \\ b &= \{b_0, b_1, \cdots, b_{n-1}, 0, 0, \cdots, 0\} \end{aligned}$$

2. Trouver une borne supérieur L des calculs : Soient u et v les maximums des valeurs absolues des coefficients de A et B repectivement. Soit d le maximum des degrés de deux polynômes. Alors, aucun coefficient du polynôme produit C dépasse $uv(d+1)$ en valeur absolue. Dans l'exemple précedent, $u = 4$, $v = 8$, et $d = 3$, aucun coefficient du polynôme produit C dépasse $L = 4 \cdot 8 \cdot 5 = 160$ en valeur absolue. D'une façon générale, en base β, la valeur maximale en valeur absolue de chaque coefficient est $\beta - 1$. Alors, la multiplication de deux polynômes de degré maximal d produit des coefficient inférieur à $(d+1)(\beta-1)^2$. Ceci peut être prouvé facilement par recurrence.
3. Choisir ω : la plus petite puissance de 2 telle que $\omega^{t/2} + 1 > L$
4. Choisir m : $m = \omega^{t/2} + 1$.
5. Calculer les inverses : Il nous faut les inverses modulaires de ω et de n pour prévoir le calcul la FFT inverse. Notons que $\omega^{-1} = \omega^{n-1} \mod m$ alors n^{-1} peut être calculé en utilisant l'algorithme d'Euclid étendu.
6. Évaluation : Cette étape a pour but de créer les représentations par valeurs de $A(x)$ et $B(x)$, et ce, en calculant FFT(a) et FFT(b). Autrement dit, on doit évaluer $A(x)$ et $B(x)$ aux racines $(2n)^{\text{ièmes}}$ de l'unité.

$$\begin{aligned} FFT(a) &= \{(\omega^0, A(\omega^0)), (\omega^1, A(\omega^1)), \cdots, (\omega^{2n-1}, A(\omega^{2n-1}))\} \\ FFT(b) &= \{(\omega^0, B(\omega^0)), (\omega^1, B(\omega^1)), \cdots, (\omega^{2n-1}, B(\omega^{2n-1}))\} \end{aligned}$$

7. Multiplication terme à terme : On calcule maintenant la représentation par valeurs du polynôme C(x)=A(x)B(x), en multipliant terme à terme les valeurs de $A(x)$ et $B(x)$. Notons que cette représentation n'est que l'évaluation de C(x) aux même racines $(2n)^{\text{ièmes}}$ de l'unité et implicitement la FFT du vecteur recherché

$c = a \otimes b$.

$$\{(\omega^0, A(\omega^0)B(\omega^0)), \cdots, (\omega^{2n-1}, A(\omega^{2n-1})B(\omega^{2n-1}))\} = FFT(c)$$

8. Interpolation : Cette dernièere étape, consiste à recuperer la représentation par coefficients du polynôme $C(x)$ en appliquant une seule fois la FFT inverse sur la représentation par valeur de $C(x)$ obtenue dans l'étape précédente.

$$c = FFT^{-1}(FFT(c)) = \{c_0, c_1, \cdots, c_{2n-1}\} = a \otimes b$$

3.6.3 Application : multiplication des grands entiers

Considérons maintenant le problème de la multiplication de deux grands entiers, a et b, de k bits où k est une puissance de 2. D'abord, les entiers sont transformés en polynômes représentés par leurs coefficients. Le problème est ensuite équivalent à multiplier deux polynômes en utilisant exactement la procédure précédente. Une étape additionnelle consiste simplement à évaluer le polynôme produit en base choisie.

En vue d'une explication plus facile, commençons par un exemple en décimal plutôt qu'en binaire. Nous voulons calculer le produit de $a = 125$ et $b = 75$. Soit $p_a(x)$ le polynôme dont les coefficients sont les chiffres décimaux successifs d'un entier a tel que $p_a(10) = a$. Alors, $p_a(x) = x^2 + 2x + 5$ et $p_b(x) = 7x + 5$. En appliquant la procédure de la multiplication de deux polynômes, on obtient le polynôme produit dont la représentation par coefficient est

$$p_c(x) = 7x^3 + 19x^2 + 45x + 25$$

Finalement, le produit $c = ab$ est obtenu en calculant $p_c(10) = 9\,375$. Le même principe s'applique pour calculer en binaire sauf que l'évaluation se fait bien entendu en base 2. On denote cette fois par $p_a(x)$ le polynôme dont les coefficients sont les bits successifs d'un entier a tel que $p_a(2) = a$. Si $a = 5 = (101)_2$ et $b = 3 = (011)_2$,

alors $p_a(x) = x^2 + 1$ et $p_b(x) = x + 1$. De même, en appliquant la même procédure, on obtient la représentation par coefficient du polynôme produit

$$p_c(x) = x^3 + x^2 + x + 1$$

et l'entier produit $c = ab$ est obtenu en calculant $p_c(2) = 15$. Nous passons maintenant au principe général de la multiplication des entiers.

Principe général

Suivant la procédure précédente, mais cette fois-ci en base 2, la multiplication de deux polynômes de taille n bits consiste en $t = 2n$ multiplications des entiers inférieurs à $\omega^{t/2} + 1$, t étant la borne du polynôme produit. Hélas, ces entiers sont de taille $1 + \frac{1}{2}\lg\omega = n + 1$ même si on prend $\omega = 2$, la plus petite valeur possible. Ainsi, la multiplication des entiers de taille n requiert $2n$ multiplications d'entiers un peu plus grand [BB88]. Pour corriger ce problème, on devrait réduire le degré des polynômes représentant les entiers à multiplier en augmentant la taille de leurs coefficients. Ce qui exige d'établir une nouvelle structure de données pour manipuler ces grands nombres. Cette structure de données est construite comme suit. Nous divisons les nombres de k bits en $n = \frac{k}{\ell}$ blocs de ℓ bits. Pour une implémentation efficace, ℓ est toujours choisi sous la forme de puissance de 2. D'après cette structure, la valeur $\beta = 2^\ell$ dénotera alors la base de cette structure et tout entier a de k bits peut être écrit par la suite sous la forme polynômiale suivante :

$$a = (a_{n-1}a_{n-2}\ldots a_0) = \sum_{i=0}^{n-1} a_i\beta^i$$

où les a_i sont dans l'intervalle $[0, \beta^\ell - 1]$. De même, $b = (b_0, b_1, \cdots, b_{n-1})$. Nous pouvons maintenant calculer le polynôme produit $c = a \otimes b$ à l'aide de la FFT,

puis évaluer $C(x)$ en $\beta = 2^\ell$ par la formule

$$c = ab = \sum_{i=0}^{2n-1} c_i \beta^i$$

pour obtenir l'entier produit de a et b. Toutefois, la FFT n'est efficace que si les polynômes considérés ont un degré suffisamment élevé. De là vient l'intérêt du choix judicieux des paramètres de la FFT.

Paramètres optimaux pour la FFT en $\mathbf{Z}_m$

Notre souci principal cette version de la FFT est le choix optimal de ℓ puisqu'à partir de lui, on peut determiner le degré des polynômes, le modulo ainsi que la racine $n^{\text{ième}}$ de l'unité. Une petite valeur de ℓ implique une grande valeur pour n, ce qui entraîne plus d'opérations papillons. Par contre, une grande valeur de ℓ implique une grande valeur de m qui implique à son tour plus de ressources pour chaque opération de base, notamment l'addition, la soustraction et la multiplications (mod m). Ce dernier choix est encore pire. Nous présentons le choix optimal théorique de ℓ et passons ensuite à une remarque pratique. Théoriquement[3], on choisit

$$\ell = \begin{cases} \sqrt{n} & \text{si } \log n \text{ est pair} \\ \sqrt{2n} & \text{si } \log n \text{ est impair} \end{cases}$$

Étant donnée que $n = \frac{k}{\ell}$ tels que k et ℓ sont de puissance de deux, nous avons ainsi transformé des entiers de k bits en polynômes de borne n dont les coefficients sont les n blocs de ℓ bits. Calculons maintenant la valeur de la racine $t^{\text{ième}}$ de l'unité où $t = 2n$ est la borne du polynôme produit. Puisque les coefficients de polynômes sont compris entre 0 et $2^\ell - 1$, et leur borne est n, le plus grand coefficient peut

[3]Notons que cette valeur théorique de ℓ est prise de la version recursive de la FFT présenté par Brassard et Bratley. Par contre, nous n'avons pas rencontré aucune référence qui parle de la valeur de ℓ dépendamment de la verison de l'algorithme (recursive ou itérative). Toutefois, nous nous sommes basés sur des valeurs expérimentales de ℓ.

atteindre $n(2^\ell - 1)^2$. Il suffit ainsi de choisir ω tel que

$$n \cdot 2^{2\ell} < m = \omega^{t/2} + 1 \tag{3.21}$$

Ceci implique que $\log \omega \geq (2\ell + \log k)(t/2)$. Par conséquent, $\omega = 8$ est suffisant pour garantir la validité du calcul lorsque $\log n$ est pair, et $\omega = 32$ est suffisant lorsque $\log n$ est impair.

En pratique, ces valeurs pour ℓ, ω et m ne sont pas toujours efficaces. Par exemple, $\omega = 8$ et $\omega = 32$ implique une grande valeur de m, ce qui implique à son tour plus de ressources et moins d'efficacité que pour le problème de super taille. D'autre part, la taille de ℓ ne devrait pas dépasser celle du mot mémoire. Nous avons trouvé que la solution optimale consiste à tester plusieurs valeurs de ℓ et calculer tous les autres paramètres correspondants. À la lumière des résultats obtenus, on pourra décider quelle combinaison des valeurs convient à notre problème. La figure 3.7, qui a été répétée pour plus d'éclaircissement, montre quelques exemples de calcul dépendamment des paramètres choisis ainsi qu'une comparaison avec la méthode classique.

		k = 256	k=1024	k=4096
Classique	Mult	64 x (32x32)	256 x (64x64)	1024 x (128x128)
	Add	56 (64 bits)	240 (128 bits)	992 (256 bits)
FFT	l	8	16	32
	n	32	64	128
	w	2	2	2
	m (bits)	33	65	129
	Mult	128 x (33x33)	256 x (65x65)	512 x (129x129)
	Papillon	576	1344	3072
	Add/Sub	1152 (33 bits)	2688 (65 bits)	6144 (129 bits)

FIG. 3.7 – Nombre d'opérations selon les paramètres

Exemple

Considérons maintenant un exemple simple qui consiste à multiplier de deux entiers, $a = 4\,321$ et $b = 8\,765$, en utilisant la FFT modulaire. Pour la simplicité, nous ferons tous nos calculs en base 10. Soient $p_a(x) = 1 + 2x + 3x^2 + 4x^3$ et

$p_b(x) = 5 + 6x + 7x^2 + 8x^3$ respectivement la représentation polynomiale de a et b. Puisque leur produit est de borne 7, 8 est la plus petite puissance de deux pour t. D'autre part, les coefficients de polynômes sont compris entre 0 et 9, et leur borne est 4, alors la valeur de ω est, d'après l'équation 3.21,

$$4 \cdot 9^2 = 324 < m = \omega^4 + 1.$$

La valeur minimal de ω vérifiant cette équation est 8, ce qui implique que $m = 4\,097 > 324$ est suffisant pour effectuer nos calculs. Un fois les paramètres sont calculés, nous pouvons obtenir les calculs suivant en applicant la procédure de la multiplication des entiers.

- $a = 1, 2, 3, 4, 0, 0, 0, 0$ $b = 5, 6, 7, 8, 0, 0, 0, 0$
- $FFT(a) = (10, 2\,257, 3\,967, 865, 4\,095, 2\,226, 126, 2\,850)$
  $FFT(b) = (26, 500, 3\,967, 2\,693, 4\,095, 406, 126, 518)$
- $C = (260, 1\,825, 512, 2\,349, 4, 2\,416, 3\,585, 1\,380)$
- $FFT^{-1}(C) = (5, 16, 34, 60, 61, 52, 32, 0)$
- $c = 5 + 16.10 + \ldots + 32.106 = 37\,873\,565$

Remarquons que pour ce problème, $m = 257$ est suffisant pour effectuer correctement notre calcul puisque la valeur maximale de coefficient est en fait 8 et non pas 9. Ceci implique qu'il suffit que $m > 4 \cdot 8^2 = 256$. Le fait d'avoir des paramètres optimisés réduit considérablement la calcul. Nous avons effectué le même calcul avec ce petit m pour montrer que le calcul ainsi que les ressources pour représenter les résultats peuvent être bien réduit à :

$$FFT(a) = (10, 56, 223, 97, 255, 42, 30, 66) \tag{3.22}$$

$$FFT(b) = (26, 139, 223, 52, 255, 95, 30, 248) \tag{3.23}$$

Complexité

La FFT en arithmétique modulaire s'exécute bien en temps $O(n \log n)$, en prenant comme opération élémentaire (à coût unitaire) les additions et multiplications de

scalaires, puisque nous avons utilisé le même algorithme. En outre, cette version de la FFT est même plus efficace dû à la forme spéciale de m d'une part et à l'absence es nombres complexes d'autre part. Quant à la complexité spatiale, elle a été remarquablement réduite puisque les nombres complexes, qui interviennent l'utilisation de deux réels, ont été remplacés tout simplement par des entiers.

CHAPITRE 4

ARCHITECTURES ET IMPLÉMENTATIONS

Dans ce chapitre, nous commençons par la méthodologie adoptée dans cette recherche suivie d'une petite introduction au langage de description de matériel utilisé. Nous y présenterons également deux architectures différentes de la FFT ainsi que leurs implémentations matérielles sur un FPGA.

4.1 Méthodologie

Notre recherche porte sur la multiplication des grands entiers en utilisant la FFT. Chaque opération "*papillon*" consiste en une multiplication et deux additions complexes, c'est équivalent à quatre multiplications et six additions réelles. Dans notre approche, le corps des nombres complexes $\mathbf{C}$ ne serait pas un bon choix puisque le calcul en $\mathbf{C}$ exige la manipulation de nombres rationnels en précision finie alors que la multiplication de grands nombres est en précision infinie. Une autre possibilité consiste à utiliser l'arithmétique modulaire ($\mathbf{Z}_m$). Un papillon se restreint simplement à une multiplication et deux additions modulaires qui s'effectuent efficacement grâce à la forme *deux-à-la-puissance*. Nous allons donc plutôt utiliser la FFT en arithmétique modulaire qui est plus efficace et plus économique. La FFT en $\mathbf{Z}_m$ utilise toujours le même algorithme mais elle a besoin de certains paramètres (ℓ, ω et m) pour garantir sa validité. Le choix théorique de ces paramètres ainsi que leurs valeurs optimales sont bien couverts dans livre de G. Brassard et P. Bratley [BB88], notre référence principale.

Nous commençons par une simulation Java pour calculer les valeurs optimales de ℓ, ω et m pour un certain problème. À la lumière des résultats obtenus, on décide quelle combinaison des valeurs convient à notre problème. Nous réalisons ensuite une première implémentation Java de la FFT pour vérifier que l'algorithme fonctionne correctement selon les contraintes mentionnées précédemment. Il s'agit de

l'architecture I qui est la version itérative séquentielle et qui consiste à multiplier deux entiers de 1024 bits. Le code complet se trouve dans l'annexe C. Puis, nous traduisons ce code Java en un langage matériel de niveau intermédiaire, appelé CASM, que nous allons introduire dans la section suivante. Ce langage est muni d'un compilateur qui génère automatiquement le code VHDL synthétisable correspondant. Nous nous servons de ce code VHDL pour effectuer la simulation et la synthèse dans l'environnement d'Altera et finalement réaliser l'implémentation sur un FPGA.

À ce stade, nous aurons à notre disposition un circuit électronique fonctionnel pour la multiplication des grands entiers de 1024 bits à l'aide de la FFT en $\mathbf{Z}_m$. Notre souci dans cette architecture est d'avoir un algorithme correctement fonctionnel en laissant de côté la question de la performance. L'architecture II prendra en considération cette dernière et sera en fait une optimisation de la première architecture. La même méthodologie sera suivie pour l'implémenter : nous commençons par une simulation Java, une traduction en CASM suivie d'une génération automatique du code VHDL correspondant, ensuite une simulation et une synthèse, et finalement l'implémentation sur un FPGA.

4.2 CASM

La description d'un circuit logique est typiquement saisie via un éditeur textuel en utilisant un langage de description de matériel (HDL pour **H**ardware **D**escription **L**anguage) dont VHDL et Verilog sont les plus célèbres. Pourtant, ces langages emploient une description de bas niveau (souvent des équations logiques) et sont seulement accessibles aux concepteurs ayant une bonne connaissance dans le domaine du matériel. D'ailleurs, ces langages exigent un travail rigoureux et plus de temps qu'une solution logicielle puisqu'il faut prendre en considération les contraintes de performance et de taille du circuit.

Le matériel et le logiciel ont toujours été considérés comme étant deux domaines d'expertise différents mais complémentaires. Pour la majorité des ingénieurs infor-

maticiens, le matériel est un monde tout à fait étrange et difficile à manipuler. L'introduction du co-design fût le premier pont entre ces deux mondes [OD04]. Néanmoins, l'environnement utilisé était souvent destiné pour réaliser la simulation et non pas la synthèse complète d'un circuit électronique. Plusieurs langages de programmation, inspirés souvent de C, ont été proposés à cette fin, notamment SystemC [Sys].

Une autre approche a été proposée il y a quelques années. Cette approche consiste à développer des outils permettant de générer automatiquement le code matériel (VHDL par exemple), à partir d'une spécification de haut niveau, (C ou Java par exemple). Le CASM (**C**hannel-based **A**lgorithmic **S**tate **M**achine) est une instance d'une telle approche qui a été développée dans notre laboratoire. Il s'agit d'un HDL de niveau intermédiaire qui permet la description d'un algorithme d'une manière similaire au langage C tout en gardant la précision au niveau des cycles d'horloge. Ce langage semble convenir aux gens des deux communautés logicielles et matérielles. Il est surtout utile pour les protypages rapides puisque le transfert et la synchronisation de données entre les différents composants se font automatiquement. Étant toute une première version, ce langage a besoin encore beaucoup d'optimisation ainsi que des fonctionnalités à intégrer. Un traitement détaillé de ce langage se trouvé dans l'article [OD04] publié à cette fin.

4.3 Détail de l'algorithme

L'analyse de l'algorithme 14 montré ci-dessous, celui de la FFT itérative, révèle les ressources de calcul, les ressources de mémoire ainsi que les structures de contrôle nécessaires au bon fonctionnement.

4.3.1 Ressources de calcul

La partie critique de cet algorithme est l'opération papillon (lignes 10–13). En effet, cet algorithme pourrait être vu comme étant une opération papillon qui s'exécute à l'intérieur de trois boucles imbriquées comme on peut l'observer dans l'algorithme.

Algorithme 14 FFT Itérative

```
1: Entrée : A, n
2: Sortie : A
3: Mirroir (A, n)
4: for i = 1 to log n do
5:    s ← 2^ℓ   {pas d'incrémentation }
6:    ω ← e^{2πi/s} {s^ième racine n^ième de l'unité}
7:    α ← 1
8:    for j = 0 to s/2 − 1 do
9:       for k = j to n − 1 ; pas de s do
10:         v ← αA[k + s/2]
11:         u ← A[k]
12:         A[k] ← u + v
13:         A[k + s/2] ← u − v
14:         α ← αω
15:      end for
16:   end for
17: end for
18: Retourner A
```

Le coût total d'exécution de la FFT est directement lié au coût d'exécution de cette opération puisque les principaux calculs y sont localisés. Chaque opération papillon fait intervenir une multiplication, une addition et une soustraction. Le calcul en arithmétique modulaire nécessite intrinsèquement des opérateurs qui permettent de calculer les opérations d'addition, de soustraction et de multiplication modulaires. En plus, nous avons besoin de l'opération d'addition et de décalage qui sont vitales pour gérer les trois boucles.

D'autre part, le chapitre 3 présente deux versions itératives : une simple et l'autre parallèle. La première s'exécute de manière séquentielle puisqu'elle ne dispose que d'une opération papillon. La seconde s'exécute de manière parallèle comme décrit dans la section 3.5.2. Plutôt que d'avoir un seul papillon, cette version permet d'exécuter $\frac{n}{2}$ papillons en parallèle offrant ainsi une rapidité de calcul maximale au prix d'un espace maximal. Entre ces deux extrêmes, on peut toujours réaliser une solution adaptée aux besoins de l'application en faisant un compromis espace/temps. La version itérative simple offre un temps d'exécution en $O(n \log n)$

avec un espace minimal d'un seul papillon alors que la version parallèle offre un temps d'exécution en $O(\log n)$ au prix d'un espace maximal de $\frac{n}{2}$ papillons, $O(\log n)$ étant la profondeur du circuit.

Dans notre étude, nous considérons la FFT de 64 points à 16 bits. Si on vise une vitesse maximale, il faut alors 64 papillons, c.à.d, 128 additionneurs et 64 multiplieurs. En ce qui concerne nos architectures, elles utilisent un seul papillon. La première est une implémentation directe de la version itérative séquentielle de cet algorithme tandis que la deuxième est une version pipelinable qui a été optimisée en modifiant le flux de données.

4.3.2 Ressources de mémoire

Commençons par le cas général où on considère la FFT d'un vecteur a de 2 coefficients. Partant de l'algorithme, nous aurons à priori besoin des ressources suivantes :

- un tableau a de taille n
- quatre variables pour gérer les trois boucles : i, j, k, s
- quatre variables pour gérer les racines de l'unité et le papillon : u, v, α, ω
- un tableau de $\frac{n}{2}$ éléments si on veut pré-calculer les racines de l'unité et éviter par la suite la multiplication qui apparaît dans la ligne 14 de l'algorithme
- il faut aussi prévoir les ressources mémoires requises pour réaliser les ressources de calcul

Pour concrétiser les choses, prenons à titre d'exemple le problème de la multiplication des entiers à 1024 bits qui implique une FFT de 128 coefficients à 64 bits. D'après le paragraphe précédent, les ressources sont :

- un tableau a de taille 128 données de 64 bits, soit un total de 1 *k octet* ;
- quatre variables de 8 bits pour gérer les trois boucles
- quatre variables de 64 bits pour gérer le calcul des racines de l'unité et de papillon
- huit variables de 64 bits et une de 128 bits pour réaliser le papillon
- un tableau de 7 éléments pour pré-calculer les racines de l'unité

Les ressources nécessaires présentées dans cette section concernent une seule FFT. Néanmoins, la multiplication des entiers requiert environ le triple des ces ressources

comme le montre l'architecture de la figure 4.1.

4.3.3 Structure de contrôle

La boucle *for* est la seule structure de contrôle dont nous avons besoin pour implémenter la FFT. Comme le montre le code algorithmique suivant, trois boucles *for* imbriquées sont nécessaires.

for $i = 1$ to $\log n$ **do**
  for $j = 0$ to $s/2 - 1$ **do**
    for $k = j$ to $n - 1$; pas de s **do**
      $\vdots$
    end for
  end for
end for

4.4 Implémentation

Dans cette section, nous allons appliquer ce que nous avons vu sur la FFT dans $\mathbf{Z}_m$. Il s'agit bien entendu de l'implémentation de multiplication des grands entiers par la FFT. Nous considérons la multiplication des entiers de 1024 bits puisque c'est la taille standard actuelle d'une clé RSA.

Le langage Java est toujours l'outil de simulation dans un premier temps tout au cours de la phase d'implémentation. La deuxième implémentation est réalisée à l'aide du CASM, qui sera traduit à son tour automatiquement en VHDL.

Deux architectures différentes (I et II) ont été effectuées en CASM pour réaliser la multiplication des entiers de 1024 bits. La multiplication des entiers de plus de 1024 bits est toujours possible en modifiant un peu la version actuelle qui n'est pas paramétrée. Ceci pourrait se faire dans un travail futur. Par contre, l'implémentation en Java est paramétrée.

4.4.1 Java

Il s'est avéré judicieux de commencer l'implémentation de son circuit (algorithme) en un langage de haut niveau. C'est plus facile et plus rapide qu'un langage de bas niveau ou même de niveau intermédiaire, surtout lors des premiers pas d'implémentation où on a besoin fréquemment d'effectuer des modifications, des simulation et des tests. La section suivante concerne uniquement les ressources de calcul ainsi que les parties principales de la FFT, le code complet sera mis dans les annexes concernées.

Calcul des paramètres de la FFT

Nous avons montré dans la section 3.6.3 que les valeurs attribuées aux paramètres de la FFT (k, ℓ, m et ω) jouent un rôle fondamental dans la détermination de la performance et des ressources nécessaires. Puisque ces paramètres dérivent en fait de la valeur attribuée à ℓ, nous avons réalisé une fonction qui permet de calculer sa valeur théorique optimale ($\ell = \sqrt{n}$ si $\log n$ est pair et $\ell = \sqrt{2n}$ autrement) pour plusieurs valeurs de n comme l'indique le code suivant.

```
while(n>=1024){
    if(log2(n)%2==0){
        l=(int)Math.sqrt((int)n);
        System.out.println("Pour n=" + n + " l=" + l + " et k=" + n/l");
    }
    else{l=(int)Math.sqrt((int) 2*n);
        System.out.println("Pour n=" + n + " l=" + l + " et k=" + n/l");
    }
    n=n/2;
}
Exécution partielle
Pour n=4096 l=64 et k=64
Pour n=2048 l=64 et k=32
Pour n=1024 l=32 et k=32
```

À la lumière de ces résultats de ℓ, nous avons constaté qu'en pratique les valeurs correspondantes des autres paramètres ne sont pas toujours efficaces. En particulier, la valeur de ℓ devrait être plus petite que k afin d'avoir un polynôme de degré suffisamment élevé et avantager par la suite l'utilisation de la FFT.

C'est la raison pour laquelle nous avons implémenté spécifiquement une fonction qui permet de calculer automatiquement toutes les combinaisons possibles de ces paramètres. Ces derniers donnent une bonne idée de toutes les dimensions de l'implémentation et permettent de décider le choix qui répond aux contraintes de l'application cible. Afin de mieux éclaircir le but de cette section, nous avons préféré de présenter, ci-dessous, une partie de l'exécution de cette fonction plutôt que son code.

```
-------------------------------------------------------
| Pour n = 1024 =2^10 les paramètres de la FFT sont   |
-------------------------------------------------------
1) Si la base l=8 alors le nb de coefficients k=128 et:
    w= 2
    Nb de Papillon : 1024
    Nb de bits de m: 129
2) Si la base l=16 alors le nb de coefficients k=64 et:
    w= 2
    Nb de Papillon : 448
    Nb de bits de m: 65
3) Si la base l=32 alors le nb de coefficients k=32 et:
    w= 8
    Nb de Papillon : 192
    Nb de bits de m: 97
    .....
```

Remarquons dans le premier cas qu'une petite valeur pour $\ell = 8$, implique une grande valeur pour $k = 128$, ce qui entraîne plus d'opérations papillons (1024). Par contre, une grande valeur de $\ell = 32$ implique moins d'opérations papillons (192) au detriment de ressources additionnelles pour chaque opération de base, notamment

l'addition, la soustraction et la multiplications modulaire (*mod m*). D'ailleurs, cela implique une valeur relativement grande de m.

Ressources de calcul

L'implémentation de ces ressources est simple puisque Java dispose déjà de sa propre classe *BigInteger* qui permet d'effectuer des calculs arithmétiques modulaires des grands entiers. Il nous fallait une petite adaptation avec les méthodes de cette classe (*add, subtract, remainder*) pour avoir le code Java suivant :

```
static BigInteger add(BigInteger x, BigInteger y, BigInteger modulo){
        return (x.add(y)).remainder(modulo);
}
static BigInteger sub(BigInteger x, BigInteger y, BigInteger modulo){
        return ((x.subtract(y)).add(modulo)).remainder(modulo);
}
static BigInteger mult(BigInteger x, BigInteger y, BigInteger modulo){
    return (x.multiply(y)).remainder(modulo);
}
```

Opération papillon

Nous avons parlé suffisamment de cette opération et la clarté du code permet d'expliquer aisément son fonctionnement.

```
BigInteger u,v;
u= a[j];
v = mult(alpha,a[k],modulo);
a[j] = add(u,v,modulo);
a[k] = sub(u,v,modulo);
```

Boucle principale de la FFT

Il s'agit de la boucle la plus externe qui est chargée de gérer les calculs de la FFT. Cette boucle se répète $\ell = \log n$ [1] fois, et à chaque itération il y a deux boucles imbriquées i et j comme indiqué ci-dessous :

```
for(l = h-1, step = 1; l >=0; l--, step == 2){
    BigInteger alpha = new BigInteger("1");
    BigInteger u,v; int k;
    for (i = 0; i < step; i++){
        for (j = i; j < taille; j += 2=step){
            k=j+step;
            butterfly(a, j, k, alpha,modulo);
        }
        alpha = mult(alpha,racine[l],modulo);
    }
}
```

Autres calculs

Nous mentionnons surtout les trois étapes additionnelles que nécessitent la FFT inverse :

- multiplication terme à terme
- multiplication par l'inverse de taille
- évaluation en la base.

La multiplication terme à terme est réalisée tout simplement par la boucle suivante :

```
for (int ind = 0; ind < a.length; ind++)
    result[ind]= mult(a[ind],b[ind],modulo);
```

La multiplication par l'inverse de la taille consiste en une boucle qui parcourt tous les éléments du tableau produit en les multipliant par cette valeur constante :

[1] Il se peut que vous rencontrez, un peu plus loin dans ce chapitre, dans le code une variable *level*. En fait, c'est le même que ce ℓ.

```
for (int ind = 0; ind < result.length; ind++)
    result[ind]= mult(result[ind],tailleInverse,modulo);
```

L'étape d'évaluation s'effectue par une boucle relativement simple :

```
for (int i = 0; i < a.length; i++){
    temp=p.multiply(a[i]);
    s=s.add(temp);
    p=p.multiply(base);
}
return s;
```

où la base *base* et le vecteur a sont envoyés comme paramètres, et p et s sont initialisés respectivement à 1 et 0.
Pour optimiser le temps d'exécution, on peut pré-calculer certaines valeurs qui resteront constantes tout au long de de l'exécution du programme, surtout s'il s'agit de grands entier. Cela améliore la performance au prix d'un petit espace additionnel. Ceci s'applique aux racines $n^{\text{ièmes}}$ de l'unité qui peuvent être pré-calculées comme suit, ω étant la racine $n^{\text{ième}}$ principale et $\ell = \log n$:

```
racine[0] = w;
for(l = 1; l< h; l++)
    racine[l] = mult(racine[l-1],racine[l-1],modulo);
```

4.4.2 CASM, architecture I

Cette première architecture est tout simplement une implémentation directe de l'algorithme 14. La figure 4.1 montre le schéma global de cette architecture qui est composée de trois modules ASM (**A**lgorithmic **S**tate **M**achine). Chaque ASM a accès à une mémoire locale pour sauvegarder les résultats intermédiaires. Les deux premiers modules (FFT) ainsi que la deuxième étape du troisième module (FFT) sont semblables. Le module 3, module de convolution, est divisé en quatre étapes, la première étape consiste à effectuer la multiplication terme à terme des données envoyées parallèlement par les deux premiers modules. Une fois cette étape

terminée, l'étape FFT est ensuite appliquée. La troisième étape est la division de ces résultats par $\frac{1}{n}$. La quatrième et la dernière étape est la représentation du polynôme produit sous forme d'un entier en l'évaluant en 2^ℓ, où 2^ℓ est la base convenue. On pourrait récapituler ceci par les points suivants :

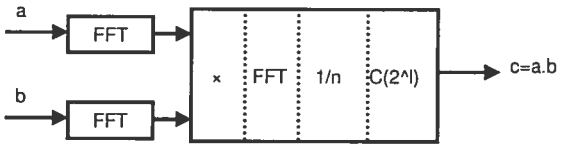

FIG. 4.1 – Architecture globale

1. Au départ, deux vecteur de données a et b, représentant les coefficients des polynômes, sont envoyés en parallèle aux deux modules FFT (1 et 2) et stockés dans leurs mémoires locales M. Ceci est accompli à l'état START et INIT :

```
START:  do  j=0; goto INIT;
INIT:   if(j<128)
            M:=x;
            M.add:=j.[0..6];
            j:{final}=j+1;
            goto INIT;
        else
            level:=6;
            step:=1;
            taille:=128;
            goto DOLOOP;
```

   Notons que les coefficients ont été rangés dès le début dans la mémoire selon l'ordre d'inversion de bits qui se réalise simplement en matériel (M.add :=j.[0..6]). Nous avons utilisé cette technique pour éviter l'étape de permutation selon cet ordre qui aurait pris quelques centaines d'accès mémoire.

2. Les deux modules FFT commencent leurs traitements en parallèle lorsque les vecteurs d'entrées sont entièrement dans leurs mémoires, ce qui est traduit par le

else de la boucle INIT. Les résultats finaux sont tout de suite envoyés au module 3. Si nous avions sauvegardé ces résultats et qu'ensuite nous les avions envoyés au module de convolution, ceci aurait requis deux accès mémoire en surplus. Nous avons réalisé ceci par un test selon que ℓ (*level* dans le code ci-dessous) a atteint sa dernière itération ($\ell = 0$) ou non :

```
FLY2:   if(level==0)
        AddX->operanda:=u;
        AddX->operandb:=v;
        Conv->operandx:=AddX->result;
        Conv->ind:=j.[0..6];
```

3. Le module 3 est divisé en quatre étapes :

(a) multiplication terme à terme : cette première étape peut commencer dès qu'elle reçoit des résultats de modules 1 et 2. En CASM, cette multiplication peut être traduite de la façon suivante :

```
T2T:    if(count<128)
    MultZ->operanda:=a;
    MultZ->operandb:=b;
    M:=MultZ->result;
    M.add:=n.[6..0];
    goto START;
```

(b) application de la FFT au polynôme produit à l'étape précédente en utilisant cette fois les inverse des racines $n^{\text{ièmes}}$ de l'unité.

(c) division par $\frac{1}{n}$ les résultats de l'étape précédente pour obtenir la représentation par coefficients du polynôme produit. Cette étape a été intégrée à l'intérieur de la boucle la plus interne pour éviter des accès mémoires supplémentaires. La dernière itération de cette boucle consiste à calculer l'opération papillon suivie d'une écriture mémoire. Nous avons pu ainsi économisé deux accès mémoire en multipliant cette valeur par l'inverse modulaire avant de la sauvegarder comme indiqué dans le code suivant :

```
FLY4:   do  MultP->operanda:=SubP->result;
            MultP->operandb:=tailleInv;
            M[k]:=MultP->result;
```

(d) évaluation : cette dernière étape dans le module *convolution* a pour but de représenter le polynôme produit sous forme d'un entier en l'évaluant en 2^{ℓ}.

Le circuit qui correspond à cette architecture est montré dans la figure 4.2. En fait, c'est le même circuit que la deuxième architecture puisqu'il s'agit de l'interface externe. Il consiste en trois entrées de 65 bits et une sortie de 16 bits comme l'indique le code ASM suivant. Une entrée pour le modulo et deux pour recevoir 128 coefficients. Un sortie de 16 bits pour afficher sériellement le produit qui consiste en 2048 bits.

```
input  modulo[65];
input  x{type="HS"}[65];
input  y{type="HS"}[65];
output produit{type="HS"}[16];
```

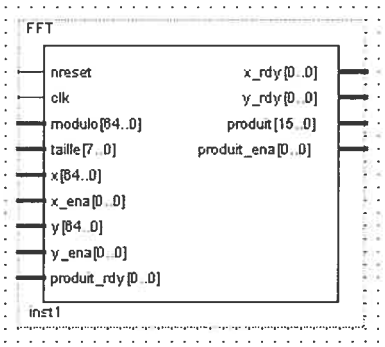

FIG. 4.2 – Interface du circuit FFT

Il nous reste à parler des parties principales du module FFT dont le code complet se trouve à l'annexe A. Chaque module FFT a trois entrées de 65 bits : une pour le modulo et deux pour recevoir 128 coefficients. Une fois les coefficients prêts, un ensemble de registres est initialisé avant de commencer la boucle principale DOLOOP qui se répète 7 fois.

```
DOLOOP: if(level~>=0)
            i:=0;alpha:=1;
            goto DOLOOP_i;
        else
            i:=0;
            goto START;
```

Nous présentons encore le code simplifié de la boucle la plus interne qui effectue essentiellement l'opération *papillon*. Elle est divisée en plusieurs états dûs aux accès mémoire multiples nécessaires. Une première lecture mémoire est faite pour préparer le premier opérande (u). La deuxième ne peut être lue que dans un autre état (FLY). Une fois cette donnée prête, les deux opérandes sont ensuite envoyées à l'opérateur de multiplication (Mult) qui renvoie à son tour le résultat qui sera rangé dans v. Les deux états FLY2 et FLY6 effectuent l'opération papillon dont chacun exige une écriture mémoire. Observons qu'il est impossible de commencer une nouvelle lecture (DOLOOP-j) avant de terminer l'écriture en FLY6 et vice versa. Ceci exige plusieurs cycles d'horloges en attendant que l'accès mémoire soit possible. Ce problème sera corrigé dans l'architecture II.

```
DOLOOP_j: if(j<taille)
            M.add:=j.[0..6];     //Lecture mémoire: u=M[j];
            u:=M;
            goto FLY;
FLY:      do MultX->operanda:=alpha;
            MultX->operandb:=M; //Lecture mémoire: v=M[k];
            M.add:=k.[0..6];
            v:=MultX->result;
            goto FLY2;
FLY2:       ...
            M:=AddX->result;     //Écriture mémoire (M[j]=u+v)
            M.add:=j.[0..6];
```

```
            goto FLY6;
FLY6:   do  ...
            M:=SubX->result;     //Écriture mémoire (M[k]=u-v)
            M.add:=k.[0..6];
            goto DOLOOP_j;
```

4.4.3 Opérateurs de calcul

Avant de passer à l'architecture II, nous devons discuter un peu des opérateurs de calculs que l'architecture I a utilisés. Le fait que les deux architectures utilisent les mêmes opérateurs, nous avons préféré de consacrer à cette fin une section indépendante mais utiles pour les deux.

Addition modulaire

Ce composant a deux entrées (opérandes) et une sortie (somme) de 65 bits comme on peut le voir dans le code ci-dessous. Tout comme n'importe quel circuit électronique, il commence à l'état initial (START). Il reste à cet état jusqu'à ce qu'il reçoive deux opérandes qu'il met dans deux registres a et b, et passe ensuite à l'état N0 où il calcule la somme (TMP). Le résultat final n'est pas encore prêt puisqu'il faut faire le test selon que TMP est plus grand que le modulo ou non. Ce test est fait à l'état N1. Si la condition est vraie, il faut ainsi soustraire le modulo, sinon il faut tout simplement envoyer le résultat au composant émetteur. Le circuit qui effectue la soustraction est très similaire. Il faut d'abord remplacer l'addition par soustraction. En conséquent, il faut tester si TMP est négative. Dans le cas positif, il faudra ainsi ajouter la modulo m pour corriger le résultat.

```
ASM Add on clk{
    post operanda{type="HS"}[65];
    post operandb{type="HS"}[65];
    post result{type="HS"}[65];
    register a[65],b[65],TMP[66];
    START:  do  a:=operanda;
```

```
        b:=operandb;
        TMP=0;
        goto N0;
N0: do  TMP=uxt{66}((a + b));
        goto N1;
N1: if(TMP>= (uxt{66}(modulo)))
        result:=uxt{65}((TMP-modulo));
        goto START;
    else    result:=TMP.[64..0];
        goto START;
}
```

Propriétés des modulos $2^n + 1$

La réduction d'un entier a modulo un entier m ($a \mod m$) peut être effectuée soit par division par m soit par des soustractions itératives de m jusqu'à ce que a devient plus petit que m. Pour les modulos m de la forme 2^n+1, ce calcul pour un entier de de $2n$ bits au maximum s'effectue simplement par une addition ou une soustraction. En effet, a peut être écrit sous la forme $a = a_1 2^n + a_0$, où a_1 est le n bits de poids fort et a_0 le n bits de poids faible. Puisque $2^n \mod (2^n + 1) = 2^n - (2^n + 1) = -1$, alors

$$\begin{aligned} a \mod (2^n + 1) &= (a_1 2^n + a_0) \mod (2^n + 1) \\ &= (a_0 - a_1) \mod (2^n + 1) \end{aligned}$$

Ainsi, la réduction modulo $(2^n + 1)$ est réduite simplement à soustraite les n bits de poids faible de n bits de poids fort et ajouter $(2^n + 1)$ si le résultat est négatif.

Multiplication modulaire

Tout comme l'opérateur d'addition, cet opérateur a deux entrées et une sortie de 65 bits comme le montre la figure suivante 4.3 qui a été générée par Quar-

tus. Un registre temporaire TMP de 130 bits est introduit pour sauvegarder le résultat intermédiaire avant d'effectuer la réduction modulaire. La multiplication

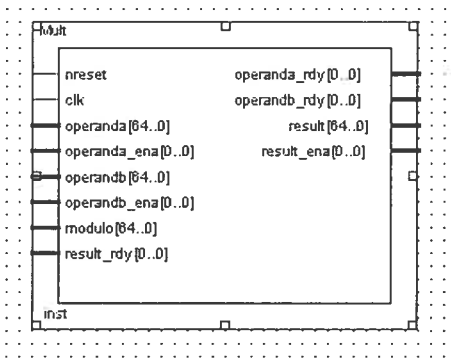

FIG. 4.3 – Interface du circuit de multiplication

de deux entiers de 64 bits est effectuée selon la méthode classique discutée dans le chapitre 3. La base 16 étant utilisée, nous aurons donc besoin d'une boucle de 4 itérations pour multiplier chaque 16 bits de b (b_i) par a, suivie d'une addition et d'un décalage nécessaires comme montré dans l'état N0-else et N1. Dans l'état N0, Le test $(a.[64] > 0)$ est une application de la propriété de $m(= 2^n + 1)$ décrite dans la section précédente. En effet, si c'est le cas alors tous les autres bits de a sont nuls puisque nous travaillons dans Z_m. Le produit $p = ab$ est ainsi obtenu par un simple décalage. Il s'agit d'un entier de $2n = 130$ bits tels que les n bits de poids faibles sont nuls et le reste n'est que b. D'après la section précédente, on obtient que $p = a - b = -b$ auquel on ajoute la valeur de m pour éliminer la valeur négative.

```
START:  do  a:=operanda;
            b:=operandb;
            TMP=0; I=4;
            goto N0;
N0:     if (a.[64]>0)
            result=uxt{65}((modulo-b));
            goto START;
        elsif (b.[64]>0)
```

```
            result=uxt{65}((modulo-a));
            goto START;
        else    TMP=uxt{130}((TMP<<16)+(b.[63..48]*a));
            b=b<<16;I:=uxt{4}(I-1);
            goto N1;
N1:     if (I>0)
            TMP=uxt{130}((TMP<<16)+(b.[63..48]*a));
            b=b<<16;I:=uxt{4}(I-1);
            goto N1;
        else    TMP=uxt{130}(TMP-TMP.[127..64,127..64]);
            goto N2;
```

4.4.4 CASM, architecture II

Cette architecture est une optimisation de l'architecture I dont la limitation principale est l'accès mémoire que nécessite très fréquemment l'opération papillon. Elle a toujours le même fonctionnement globale montrée dans la schéma 4.1. La seule différence entre les deux architectures est la structure interne du module FFT dont le nombre d'opérations papillon est directement lié à son coût d'exécution comme décrit dans la section 4.3.1.

La figure 4.4 montre cette architecture optimisée dont la boucle principale, compteur ℓ, est implémentée comme étant un jeu de pong entre deux ASM A et B. Lorsque ℓ est pair, A envoie les données à B via l'ASM de l'opération papillon. À la

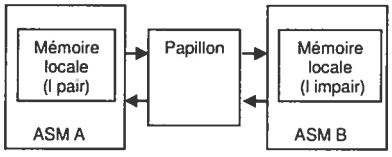

FIG. 4.4 – Architecture II

fin de l'itération, ℓ devient impair et B renvoie les données à A via l'opération papillon également. La lecture et l'écriture en mémoire se font ainsi en parallèle, ce qui permettra plus tard de réaliser le pipelinage qui n'a pas été introduit présentement.

Détaillons un peu ceci : le module FFT, qui consiste dans l'architecture I en un seul ASM à une seule mémoire de données, a été divisé en cinq ASM interagissant ensemble comme le montre la figure 4.5 : A, B, Encodeur, Papillon et Decodeur. Par analogie au langage de haut niveau, c'est équivalent à subdiviser une longue méthode en plusieurs courtes méthodes indépendantes.

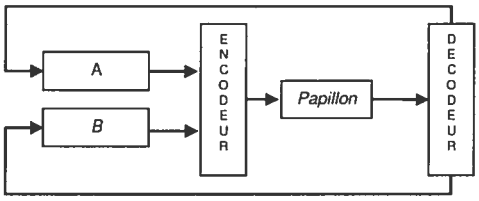

FIG. 4.5 – Architecture interne de la FFT

Les deux ASM, A et B, sont semblables et contiennent la boucle principale de la FFT qui gère la transmission de données entre les différents ASM. Encodeur est chargé d'acheminer les données au papillon. Il reçoit les données venant alternativement de A et B (toutes les 64 données pour chaque valeur de ℓ), et les renvoie au papillon via trois signaux de sorties de données (outa, outb, outAlpha) dont le papillon a besoin pour accomplir le calcul. Observons que (Encodeur) reçoit toujours ses premières 64 données de A (N0-else) et ses deuxièmes 64 données de B (N0-if).

```
ASM Encodeur on clk{
    post outa{type="HS"}[65];
    post outb{type="HS"}[65];
    post outAlpha{type="HS"}[65];
    register i[7];
    START:  do  i=0;
            goto N0;
    N0: if (i.[6])          //Si (i>=64), recevoir des données de B
            outa:=B->outa;
            outb:=B->outb;
            outAlpha:=B->outAlpha;
            i:{final}=uxt{7}(i+1);
            goto N0;
```

```
else                    //Si (i<64), recevoir des données de A
    outa:=A->outa;
    outb:=A->outb;
    outAlpha:=A->outAlpha;
    i:{final}=uxt{7}(i+1);
    goto N0;
```

Le papillon a pour but d'accomplir l'opération papillon et renvoie le résultat à Decodeur qui fonctionne à l'inverse de Encodeur, en d'autres termes, il achemine les données vers A ou B dépendamment de la valeur de ℓ. À la différence de Encodeur, Decodeur a deux couples identiques de signaux de sortie (outaB, outbB, outaA, outbA) puisqu'il envoie les résultats (deux coefficients) déjà calculés par papillon.

```
ASM Decodeur on clk{
post outaB{type="HS"}[65];
post outbB{type="HS"}[65];
post outaA{type="HS"}[65];
post outbA{type="HS"}[65];
register i[7];
START:  do  i=0;
        goto N0;
N0: if (i.[6])
        outaA:=Papillon->outa;
        outbA:=Papillon->outb;
        i:{final}=uxt{7}(i+1);
        goto N0;
    else
        outaB:=Papillon->outa;
        outbB:=Papillon->outb;
        i:{final}=uxt{7}(i+1);
        goto N0;
```

Cette architecture a permis de réduire le temps d'exécution : dans la première architecture, une donnée est lue en mémoire, traitée par l'opération papillon et

finalement réécrite dans la même mémoire, comme indiqué aux états DOLOOP-j, FLY, FLY2 et FLY6. Une nouvelle lecture ne peut commencer qu'à la fin de cette réécriture, ce qui entraîne à une perte de plusieurs cycles d'horloge à chaque opération papillon. Ceci dégrade remarquablement la performance de la FFT.

Ce problème n'existe plus dans l'architecture II puisque la lecture et l'écriture se font dans deux ASM séparés s'exécutant en parallèle. Par exemple, pour $\ell = 1$, le rôle de A est la lecture en mémoire (64 valeurs) et celui de B est l'écriture en mémoire (64 valeurs). Autrement dit, A lit une donnée et l'envoie au papillon via Encodeur. A peut commencer tout de suite une nouvelle lecture puisque Decodeur va écrire le résultat du papillon dans la mémoire de B comme on peut l'observer dans les deux boucles principles de A et B qui s'exécutent en parallèle comme indique le montre le code suivant :

```
ASM A on clk {
...
LOOP_j: if(j<taille)
            outa:=M[j];          //Lecture mémoire en ASM A
            outAlpha:=beta;
            k:=uxt{8}(j+step);
            goto FLY;
        else
            i:=uxt{8}(i+1);
            goto LOOP_i;
FLY:    do  outb:=M[k];
            j:=uxt{8}(j+(step<<1));
            goto LOOP_j;

ASM B on clk {
...
SAVEj: if(j<taille)
            M[j]:=DecodeX->outaodd; //Écriture mémoire en ASM B
            k:=uxt{8}(j+step);
            goto SAVE;
```

```
        else
            i:=uxt{8}(i+1);
            goto SAVEi;
SAVE:   do  M[k]:=DecodeX->outBodd;
            j:=uxt{8}(j+(step<<1));
            goto SAVEj;
```

À la fin de cette itération, ces deux ASM alternent les rôles : B lit une donnée, l'envoie au papillon qui la renvoie à son tour à A pour être mémorisée.

En résumé, pour une valeur donnée de ℓ, le circuit fonctionne de la façon suivante : au moment où A lit une nouvelle donnée en mémoire, le papillon traite la donnée courante. B à son tour écrit l'ancienne valeur calculée par papillon. À la prochaine valeur de ℓ, A et B changent de rôle.

4.5 Résulats

Cette section sera consacrée à la présentation de nos résultats de simulation dans deux environnements différents : Java et Quartus d'Altera. L'architecture II a été simulée en Quartus seulement mais l'architecture I a été simulée en Java et Quartus. Les deux architectures ont été automatiquement transformées de CASM en VHDL, synthétisées dans un FPGA fabriqué par Altera et simulées.

Malheureusement, nous n'avons pas eu l'occasion de réaliser ces deux architectures sur un FPGA pour deux raisons. Premièrement, nous n'avions pas encore à notre disposition de FPGA pour faire tourner notre architecture. Deuxièmement, pour réaliser un système numérique sur un FPGA, il faut également intégrer une interface entre de celui-ci un moniteur LCD. C'est cette interface qui permet de saisir les données au FPGA, pour les traiter, et d'afficher le résultat obtenu à l'écran pour le visualiser.

Les résultats sont conformes aux espérances et ont été validés par l'implémentation Java réalisées auparavant. La version actuelle du CASM ne permettait d'implementer qu'une version séquentielle de l'opération papillon qui exige

14 cycles d'horloge. Néanmoins, cette opération peut être accomplie en seulement un cycle d'horloge dans une version pipelinée qui sera possible avec la nouvelle version du compilateur CASM. La valeur théorique minimale accessible sera alors de 448 cycles pour un FFT simple avec un opérateur simple de papillon.

4.5.1 Tests de validité

Plusieurs simulations ont été accomplies correctement en Java, notamment la multiplication des entiers de 1024, 2048 et 4096 bits. Nous présentons dans cette section en bref la simulation concernant la multiplication des entiers de 1024 bits dont les étapes d'exécution détaillées se trouvent dans l'annexe C. Nous avons commencé par créer deux tableaux différents de 128 coefficients, a et b, dont 64 de poids fort sont nuls comme le montre le code suivant. La représentation en hexadécimal a été utilisée afin de faciliter la validation des résultats de simulation ultérieurement.

```
for(int i=0;i<nbCoef;i++){
      a[i]= i+1;
      b[i]= i+3;
}
/* Résultats d'exécution
a[0]=0001, b[0]=0003
a[1]=0002, b[1]=0004
a[2]=0003, b[2]=0005
...
a[62]=3F, b[62]=41
a[63]=40, b[63]=42 */
/* Ajout des coefficients nuls pour */
for(int i=nbCoef;i<a.length;i++){
    a[i]=0;
    b[i]=0;
}
```

Nous avons cré une fonction, dont la boucle principale est ci-dessous, pour évaluer un polynôme représenté par sa valeur. Pourtant, nous n'en aurons pas besoin dans

cet exemple.

```
s=temp=0; p=1;
for (int i = 0; i < a.length; i++){
    temp=p.multiply(a[i]);
    s=s.add(temp);
    p=p.multiply(base);
}
```

Les deux grands entiers obtenus sont donnés ci-dessous en hexadécimal. Puisque nous utilisons la base 16, l'obtention de l'entier en question est simplement la concaténation des coefficients ensemble selon leurs poids.

40003F003E003D003C003B003A00390038003700360035003400330032003100300002F002
E002D002C002B002A00290028002700260025002400230022002100200001F001E001D001C
001B001A00190018001700160015001400130012001100100000F000E000D000C000B000A0
00900080007000600050004000300020001

4200410040003F003E003D003C003B003A00390038003700360035003400330032003100300
0002F002E002D002C002B002A00290028002700260025002400230022002100200001F001E
001D001C001B001A00190018001700160015001400130012001100100000F000E000D000C0
00B000A00090008000700060005000400003

Ensuite, le produit a été calculé de deux manières différentes : à l'aide de la classe BigInteger de Java et à l'aide de la FFT. Les deux méthodes ont donné le résultat suivant :

1080207E2FFB3EF84D765B7668F97600828C8E9E9A37A558B002BA36C3F5CD40D618DE7EE
673EDF8F50EFBB701F207C10D25121F16B01AD91E9B21F724EE278129B12B7F2CEC2DF92E
A72EF72EEA2E812DBD2C9F2B282959273324B721E61EC11B49177F13640EF90A3F0536FFE
1FA40F454EE1EE79FE0D8D9CAD276CADDC300BA60B202A9E5A2089A6A930A8BE785007E54
77E271A96BA865DE604A5AEB55C050C84C02476D43083ED23ACA36EF33402FBC2C6229312
6282346208A1DF31B801930170214F51308113A0F8A0DF70C800B2409E208B907A806AE05
CA04FB044003980302027D020801A2014A00FF00C0008C00620041002800160000A0003

4.5.2 Résultats de synthèse

La table 4.1 montre les résultats de synthèse des deux architectures discutées dans ce chapitre sur un FPGA de type Stratix.

	Architecture I	Architecture II
Mémoire (octets)	3,315	12,480
Fréquence (MHz)	50.58	44.06
Cellules logiques	7,377	24,146

TAB. 4.1 – Résultats de synthèse

Bien que l'architecture II soit plus rapide que l'architecture I, au detriment d'une augmentation de ressources, cette implémentation n'est pas optimisée pour concurrencer les travaux d'autrui. Essentiellement, elle réalise la multiplication des grands entiers. Quant à la performance, elle serait l'objectif d'un autre travail futur. En plus que nous savons déjà les limites de cet algorithme, il y aura une version CASM plus optimisée. Il est toujours possible d'utiliser un langage comme VHDL.

4.5.3 Résultats de simulation

Le même exemple de simulation en Java a été adapté à l'environnement Quartus pour faciliter la comparaison. Après avoir préparé le fichier de simulation, le résultat obtenu a été cohérent avec celui obtenu en Java. Afin de faciliter la comparaison, nous avons utilisé la représentation en base 16. À défaut d'espace, nous avons choisi seulement deux parties de la simulation, le début et la fin de l'affichage. La figure 4.6 montre les quatre premiers chiffres hexadécimaux du produit (0028, 0016, $000A$ et 0003). Le premier affichage (0003) commence à l'instant 1.6195 ms, le deuxième ($000A$) à 1.6198 ms et ainsi de suite. Ces chiffres sont effectivement ceux le plus à droite du produit obtenu par la simulation en Java précédemment.

La figure 4.7 montre les chiffres le plus à droite (de poids faible) du produit obtenu. Remarqueons que le chiffre 1080 est suivit de 5 chiffres nuls. C'est pour cette raison qu'ils n'apparaissent pas à gauche du produit obtenu par Java.

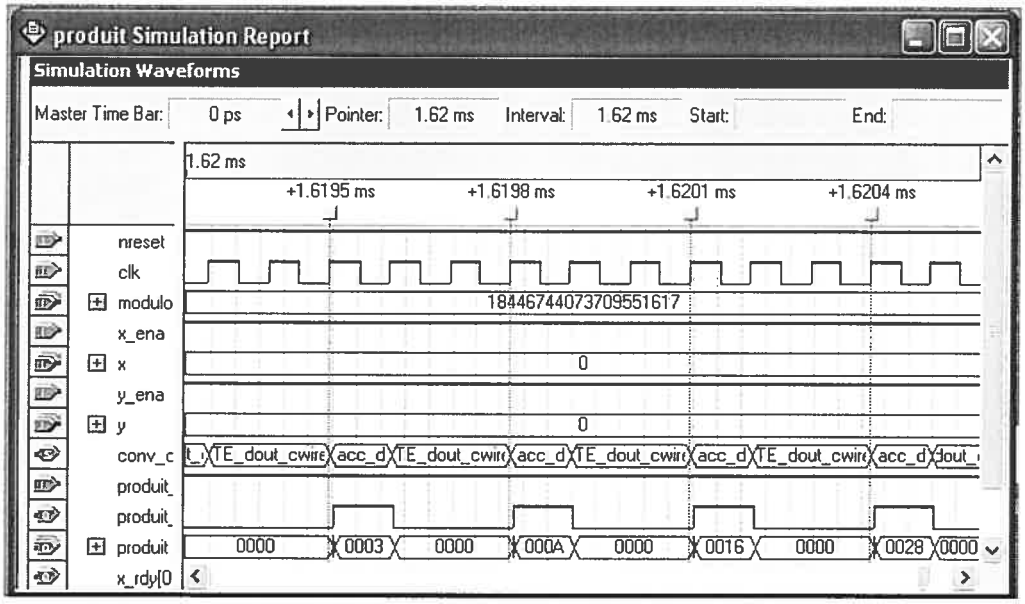

FIG. 4.6 – Chiffres de poids faible du nombre produit

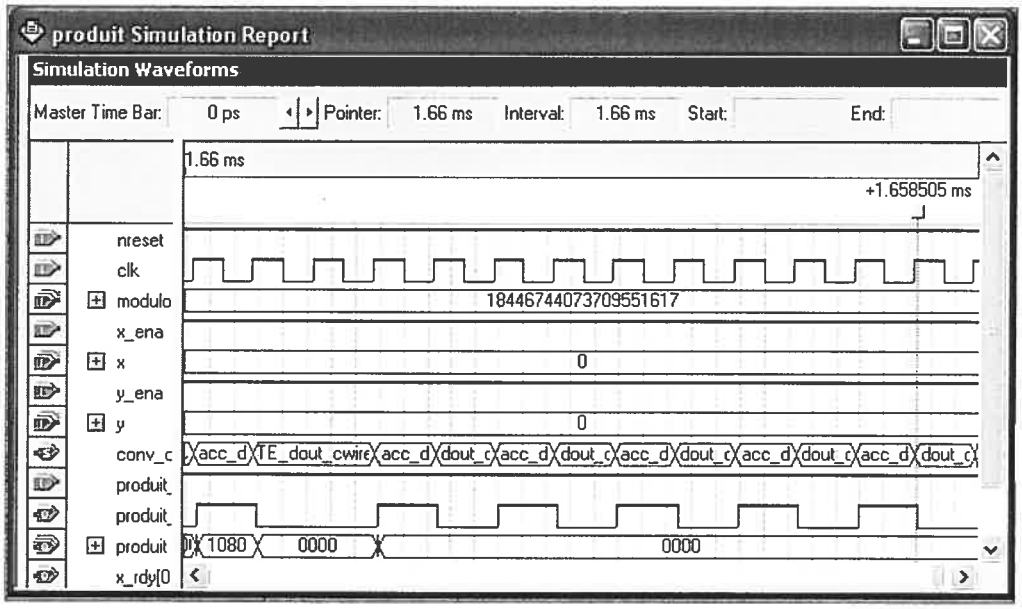

FIG. 4.7 – Chiffres de poids fort du nombre produit

CHAPITRE 5

CONCLUSION ET PERSPECTIVES

Lorsque cette recherche a commencé, nous voulions répondre à la question "*Est-ce que la FFT est présentement pratique pour la multiplication des entiers en général et les cryptosystèmes à clé publique en particulier ?*". Pourtant, nous nous sommes rendu compte qu'elle s'est élargie un peu pour être également une étude approfondie de l'état de l'art de la cryptographie, des algorithmes de multiplication ainsi que des architectures et des technologies utilisées dans une approche matérielle.

5.1 La cryptographie et la multiplication des entiers

Nous avons débuté ce mémoire par le rôle vital que joue la cryptographie aussi bien dans le domaine militaire que civil. Nous avons ensuite passé en revue le développement isolé de la cryptographie dans plusieurs civilisations, du premier évènement historique documenté jusqu'à l'algorithme symétrique le plus sophistiqué et le plus sécuritaire pour le moment. Cette revue historique a montré comment la cryptographie a mis des siècles pour finalement formuler des critères scientifiques, établis par Kerckhoffs et Shannon, auxquels doit répondre tout système cryptographique :

- La sécurité d'un système cryptographique doit reposer seulement sur la sécurité de la clé.
- La sécurité d'un système cryptographique, non quantique, est uniquement calculatoire et dépend de la taille de cette clé.

Ces contraintes ont amplifié le défaut majeur de la cryptographie symétrique qui repose sur la communication secrète préalable de clés. Ce défaut, connu comme *"problème de distribution de clés"*, a limité durant des années l'essor de la cryptographie. La découverte de la cryptographie à clé publique a finalement offert une belle solution pour ce problème en permettant à deux personnes de communiquer

confidentiellement sans aucun secret préalable. Cette solution requiert cependant une grande puissance de calcul puisqu'elle se base sur la difficulté calculatoire d'un certain problème mathématique telle que l'exponentiation modulaire ou le logarithme discret. L'opération de base de ces problèmes est la multiplication des grands entiers qui s'effectue en temps $O(n^2)$ par la méthode classique. Les algorithmes de multiplications furent ensuite étudiés largement et intensivement des points de vue théorique et pratique.

5.2 Algorithmes de multiplication

Nous avons discuté les algorithmes les plus utilisés dans la littérature en fondant notre discussion sur des travaux pratiques dont les résultats sont conformes aux résultats théoriques. En particulier, les algorithmes asymptotiquement rapides ont été présentés comme étant une solution possible pour les cryptosystèmes futurs.

Caractérisé par sa facilité d'implémentation, l'algorithme classique est le plus intéressant tant que la taille des entiers ne dépasse pas un certain seuil (assez petit). Par contre, c'est l'algorithme le plus lent pour la multiplication des entiers au-dessus de ce seuil.

Outre qu'il s'exécute en temps $O(n^2)$, l'algorithme de Horner n'est pas un bon candidat pour les implémentations matérielles [JB97, BM04] en raison de sa nature itérative. Par contre, il est très efficace dans l'évaluation des polynômes qui est l'application pour laquelle il a été découvert à l'origine.

L'algorithme de Montgomery requiert $2n^2$ multiplications et $2n(n-1)+1$ additions pour calculer la multiplication des entiers [NEG04]. Ultérieurement, il a été adapté pour la multiplication modulaire de manière à le rendre plus efficace. Néanmoins, la conversion entre la représentation de Montgomery et la représentation binaire classique et le pré-calcul de certains paramètres prennent un temps si important que cela rend l'utilisation de cet algorithme avantageux seulement dans les applications exigeant des multiplications répétées plusieurs fois telle l'exponentiation, justement ce qui est approprié à la cryptographie. Sinon, la per-

formance de l'algorithme de Montgomery est très proche de celle de l'algorithme classique [BGV94].

L'algorithme de Karatsuba et celui de la FFT sont communément connus pour leurs efficacités asymptotiques. Autrement dit, ils sont plus rapides que la méthode classique lorsque n est relativement grand et cette supériorité augmente lorsque n augmente. Le défaut commun de ces algorithmes rapides est que le seuil à partir duquel ils offrent une efficacité avantageuse est assez grand. C'est la raison pour laquelle la méthode classique est souvent utilisée par ces algorithmes sophistiqués pour calculer les multiplications des entiers inférieurs à ce seuil (multiplications scalaires) [BB88, BGV94, LHL03, Moe76].

L'algorithme de Karatsuba est un algorithme récursif plus performant que l'algorithme classique après un certain seuil. Ainsi, un algorithme hybride combinant les deux méthodes est souvent souhaitable. Cet algorithme hybride, qui consiste à appliquer récursivement l'algorithme de Karatsuba jusqu'à ce que la taille des entiers atteignent le seuil, présente une meilleure performance sur celui de Karatsuba seul [LHL03]. Identiquement à l'algorithme de Horner, en raison de sa nature récursive, il n'est pas pratique dans une approche matérielle.

La FFT est asymptotiquement l'algorithme le plus performant dans la littérature en réduisant le temps de la multiplication à $O(n \log n)$ dans le cas où les opérations élémentaires portent sur des scalaires de ℓ bits. Si on voulait s'astreindre à ne compter comme élémentaire que les opérations au niveau du bit, cette compléxité passerait à $O(n \log n \log \log n)$. Tout comme l'algorithme de Karatsuba, elle commence à devenir intéressante après un certain seuil. Le seuil demandé dans FFT est même plus grand que celui dans l'algorithme de Karatsuba. Pourtant, son avantage fondamental sur ce dernier est sa nature itérative qui permet de réaliser une version parallélisée capable de calculer la multiplication en $O(\log n)$. Ceci le rend un excellent candidat pour les implémentations matérielles.

5.3 Architectures et techniques

Suite à l'invention de la cryptographie à clé publique et tout au long des deux dernières décennies, une vaste variété d'architectures pour la multiplication des grands entiers est apparue. Citons essentiellement les réseaux systoliques [Atr65, Wal93, IMI94, Kor94, BP01, BOPV03, TK03], les systèmes de numération redondant (RNR) [EW93, Oru95], les systèmes de congruences (RNS) [PP95, BDK98, BDK01, BI04, CNPQ03] et le théorème du reste chinois (CRT) [WHW01, QC82, Gro00].

L'architecture systolique est une solution attirante pour les systèmes informatiques exigeant des calculs massifs. Elle fut étudiée intensivement du point de vue théorique et pratique et s'est révélée être la meilleure solution du coté performance [DL92, Wal93, Kor94, TK99, BP01, IMI94, BOPV03, FK03]. Pourtant, l'implémentation coûteuse de cette approche en a fait une solution impraticable.

La technique de multiplication à base élevée n'est pas souhaitable puisqu'elle augmente la complexité du circuit et mène à une basse fréquence.

Le RNR est utilisé pour réduire la propagation de retenues dans les milliers d'opérations d'additions auxquelles se réduisent la multiplication modulaire et l'exponentiation modulaire. L'arithmétique en logique signée (**S**igned **D**igit **R**epresentation) et la technique d'estimation de signe (**S**ign **E**stimation **T**echnique) sont deux instances du RNR. Pourtant, cela requiert un temps de calcul additionnel et augmente la complexité du circuit.

L'arithmétique en RNS, en conjonction avec le CRT, fournit un excellent moyen pour l'arithmétique de très grands entiers. Le parallélisme s'y concrétise parfaitement en offrant une augmentation de performance d'un facteur de 3.5 [WCS02, BOPV03] voire 4 [Kob94]. Cette approche est pourtant un peu détériorée par le temps que nécessite la conversion entre le système binaire et le RNS. D'ailleurs, la difficulté de comparaison, de détection de signe et de détection de dépassement de capacité sont également des opérations coûteuses dans le RNS. Elle semble quand même être le plus attirant du côté espace et performance.

Certains récents travaux utilisent des additionneurs sans propagation de retenues CSA (**C**arry **S**ave **A**dder) [KH98, MMM+03, MMM04] et des LUT (**L**ook-**U**p **T**able) [BST02, BS03b, BS03a, BS04a, BS04b] pour mémoriser des valeurs précalculées. Les CSA augmentent la complexité ainsi que l'espace du circuit. Les LUT exigent un espace mémoire supplémentaire et du temps additionnel pour précalculer certaines valeurs de chaque nouveau modulo.

5.4 La technologie FPGA

Nous avons présenté presque toutes les technologies qui peuvent être utilisées dans une implémentation matérielle, leurs avantages, leurs limites ainsi que leurs domaines d'applications. Grâce à la caractéristique de reconfiguration, la technologie des circuits reconfigurables (FPGA) a révolutionné la méthodologie de l'ingénierie des systèmes numériques. Elle est effectivement une solution qui combine presque la flexibilité d'un logiciel et la vitesse d'un ASIC.

Cette technologie a considérablement évolué au cours des dix dernières années. Partant de quelques cellules logiques, il existe maintenant des circuits contenant des centaines de milliers des cellules. Les FPGA récents sont configurables en une centaine de millisecondes, contiennent des millions de portes logiques et englobent jusqu'à 500 blocs mémoires de 20Kb (RAM et ROM). En outre, certains sont si sophistiqués qu'ils contiennent des multiplieurs élémentaires, des noyaux de processeurs et permettent des fréquences d'horloge allant jusqu'à 500 Mhz [Xil05]. Les grands FPGA peuvent contenir 512 de ces multiplieurs qui travaillent typiquement sur des opérandes de 18 bits.

Les domaines d'application se sont eux-aussi élargis au fur et à mesure que la taille et la vitesse des circuits augmentaient. Ils sont maintenant capables de réaliser des fonctions logiques précédemment réservées aux ASIC seuls, diminuant ainsi considérablement le temps de développement. Les concepteurs d'ASIC eux-même préfèrent de plus en plus passer par l'étape intermédiaire d'un FPGA, évitant par la suite les pertes dues aux erreurs probables et spécifications insatisfaisantes. Il

est aussi possible de connecter un ensemble de FPGA (multi-FPGA) afin de réaliser des circuits de plus en plus performants et complexes, approche particulièrement adoptée dans la simulation et prototypage rapides des ASIC plutôt que de simuler pendant de longues heures avec un simulateur logiciel.

Ces caractéristiques (reconfiguration, haute intégration, performance) ont fait des FPGA un outil idéal pour l'implémentation matérielle d'innombrables applications numériques. Le FPGA semble en particulier être idéal pour les implémentations des algorithmes cryptographiques qui nécessitent fréquemment des changements des paramètres en temps réel, des crypto-processeurs assez complexes, et un débit de chiffrement élevé avec très court délais.

La conception de systèmes performants nécessite toutefois une exploitation judicieuse des ressources d'un FPGA. D'autre part, ces circuits n'ont pas vocation à concurrencer les super calculateurs ou remplacer les ASIC, mais plutôt à offrir une alternative en fonction de critères comme l'encombrement, les performances et le prix, et sont de ce fait bien adaptés à des applications de qualité dans le domaine des systèmes changeants.

5.5 La transformée de Fourier rapide et la multiplication des entiers

Considéré parmi les 10 meilleurs algorithmes dans le $20^{\text{ième}}$ siècle [Roc00], l'algorithme de la FFT est déjà omniprésent dans la plupart des branches scientifiques. Maintes applications logicielles et matérielles, des processeurs dédiés ainsi que de librairies gratuites utilisant la FFT sont disponibles sur marché. Néanmoins, cette FFT se sert des nombres complexes qui ne conviennent pas à la multiplication des entiers. Cet algorithme a été évité dans la multiplication des entiers parce que la taille du problème (seuil) à partir de laquelle il commence à être meilleure que la méthode classique est assez haut. Cette taille est si élevée qu'elle était plus grande que la plupart des problèmes arithmétiques rencontrés dans le passé. Même une trentaine d'années après la venue de cryptographie à clé publique, la taille de clés était toujours plus petite que la taille à partir de laquelle la FFT offre une effica-

cité avantageuse sur tous les autres algorithmes connus. Nous avons essayé dans ce mémoire de répondre à la question suivante : *est-ce que la FFT est toujours inintéressante pour les problèmes actuels ?*

Cette question fut répondue par une implémentation matérielle de la FFT en arithmétique modulaire. Nous avons visé à présenter cette approche comme étant une méthode efficace pour la multiplication de grands entiers pour les problèmes du futur proche, si pas présents, l'opération fondamentale de tous les cryptosystèmes à clés publiques contemporains. Les résultats ont montré que pour les opérandes de 1024 bits, le nombre de multiplications est le même pour FFT et la méthode classique. Le nombre d'additions est même plus grand d'ailleurs. Cependant, l'algorithme de la FFT offre une meilleure performance pour les opérandes de 4096 bits et plus. Nous pensons que cette méthode sera un choix approprié aux cryptosystèmes futurs où la taille principale atteindra 4096 bits ou plus. Cela est basé sur deux faits. Le premier est que l'histoire des cryptosystèmes nous a révélé que la taille de clés est sans cesse croissante et a presque atteint le niveau où la FFT devient efficace. Le deuxième fait est que plus la taille est grande plus la FFT est préférable à tous les autres algorithmes dans la littérature.

Pourtant, une évaluation adéquate de performance de la FFT n'était pas possible puisque la version prototype du CASM ne permettait d'implémenter qu'une version séquentielle de l'opération papillon qui exige 14 cycles d'horloge. Cette opération peut cependant être accomplie en seulement un cycle d'horloge dans une architecture pipelinée qui sera possible avec la nouvelle version du CASM, ce qui est équivalent à une amélioration de facteur de 14 à priori. En outre, de meilleures performances sont aussi réalisables avec plusieurs unités de papillon fonctionnant en parallèle.

Toutefois, la FFT souffre de certains défauts à prendre en considération. Tous les entiers appartenant à un intervalle donné de n bits sont traités de la même manière. Autrement dit, la multiplication à l'intérieur d'un tel intervalle s'effectue en temps constant. Par exemple, si $1024 < n < 2048$, la multiplication d'entiers ne prend pas plus de temps ou de ressources pour $n = 1025$ bits qu'il en faut

pour $n = 2047$ bits. Ceci a pour conséquence qu'à la limite de chaque intervalle, la FFT fait un saut considérable, en temps d'exécution et en ressources de calcul. Une solution possible à ce problème a été proposée dans la section 5.7.

D'autre part, la FFT requiert une implémentation très rigoureuse et judicieuse. Des tests devraient être faits pour choisir les paramètres ainsi que le seuil au-dessus duquel il faut recourir à une autre méthode plus performante.

5.6 Cryptographie du futur

C. Bennett et G. Brassard ont introduit en 1984 une idée révolutionnaire, celle la cryptographie quantique, qui permet d'accomplir la distribution de clés avec une sécurité inconditionnelle [BB84] garantie non pas par les hypothèses mathématiques mais par les lois de la mécanique quantique.

Bien que devenue réalité, il est difficile de prévoir à l'heure actuelle quand elle sera utilisée à grande échelle à cause des difficultés technologiques qu'il reste à surmonter. La cryptographie à clé publique reste donc pour l'instant la seule alternative universelle et le besoin d'algorithmes de multiplication plus efficaces reste pertinent.

5.7 Perspectives

La version actuelle du CASM ne permettait d'implementer qu'une version séquentielle de l'opération papillon qui exige 14 cycles d'horloge. Cette opération peut être accomplie en seulement un cycle d'horloge dans une version pipelinée qui serait possible avec la nouvelle version du compilateur CASM dans le futur proche.

La valeur du modulo m a été choisie selon la formule $m^{n/2} \geq n(\ell - 1)^2$ où ℓ est la base calculée comme décrit par, entre autres, G. Brassard et P. Brateley [BB88]. En pratique, l'application directe de cette approche implique plus de ressources qu'il faut pour calculer le produit en question. Un moyen d'améliorer la situation est de jouer sur la valeur de ℓ en procédant dans le sens inverse de cette formule. Pour concrétiser l'idée, considérons l'application abordée dans cette recherche qui

concerne la multiplication des entiers de 1024 bits. Ainsi, la valeur de ℓ est telle que $2^{64} \geq 128(\ell - 1)^2$, ce qui implique que $\ell \leq 2^{28}$. Par conséquent, nous pourrons effectuer la multiplication des entiers de taille jusqu'à $28 \times 64 = 1792$ bits au lieu de 1024 bits seulement en utilisant toujours les mêmes ressources.

Montgomery a montré la meilleure efficacité pour implémenter l'exponentiation modulaire dans une approche matérielle. Découvrir l'intérêt de la FFT dans les algorithmes cryptographiques consiste à la comparer avec celui-ci pour les entiers de taille 2048 bits par exemple ou plus. Cette comparaison consisterait à implémenter la FFT et l'intégrer dans une application cryptographique telle RSA où l'algorithme de Montgomery est souvent utilisé.

Une approche matérielle hybride combinant la FFT avec d'autres algorithmes, tels que l'algorithme de Karatsuba ou celui de la méthode classique, semble être attirante à explorer.

Il est en général intéressant d'étudier des algorithmes de multiplication et leurs implémentations en cryptographie, en particulier celui de la FFT pour la multiplication des grands nombres. Ceci exige nécessairement un HDL bien optimisé afin de pouvoir mieux concrétiser l'intérêt de ces algorithmes.

BIBLIOGRAPHIE

[And05] Jason R. Andrews. *Co-Verification of Hardware and Software for ARM SoC Design.* Elsevier, 2005.

[Atr65] A. J. Atrubin. A one-dimensional real-time iterative multiplier. *IEEE Trans. Elec. Comput.*, 14 :394–399, 1965.

[Bar87] P. Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology, Proc. Crypto'86, LNCS 263*, pages 311–323. Springer-Verlag, 1987.

[BB84] Charles Bennett and Gilles Brassard. Quantum cryptography : Public key distribution and coin tossing. In *Proccedings of IEEE International Conference on computers, Sytems and Signal Processing*, pages 175–179, 1984.

[BB88] Gilles Brassard and Paul Bratley. *Algorithmics Theory : Practice & Practice.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[BDK98] J. Bajard, L. Didier, and P. Kornerup. An RNS Montgomery's Modular Multiplication Algorithm. *IEEE Transactions on Computers*, 19(2) :167–178, 1998.

[BDK01] J. Bajard, L. Didier, and P. Kornerup. Modular Multiplication and Base Extensions in Residue Number Systems. In *ARITH '01 : Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pages 59–65, Washington, DC, USA, 2001. IEEE Computer Society.

[BDL97] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). *Proceedings of EUROCRYPT'97, Lecture Notes in Computer Science*, 1233 :37–51, 1997.

[Beu94] Albrecht Beutelspacher. *Cryptology.* Mathematical Association of America, 1994.

[Beu01] Jean-Luc Beuchat. *Étude et conception d'opérateurs arithmétiques optimisés pour circuits programmables.* PhD thesis, École Polytechnique Fédérale de Lausanne, Swisse, 2001.

[BGV94] A. Bosselaers, R. Govaerts, and J. Vandewalle. A fast and flexible software library for large number arithmetic. In *15th Symp. on Information Theory in the Benelux*, Louvain-la-Neuve (B), 1994. Werkgemeenschap Informatie-en Communicatietheorie, Enschede (NL).

[BI04] J.-C. Bajard and L. Imbert. A Full RNS Implementation of RSA. *IEEE Computer Society*, 53(6) :769–774, 2004.

[Blu99] Thomas Blum. Modular Exponentiation on Reconfigurable Hardware. Master's thesis, Worcester Polytechnic Institute, April 1999.

[BM04] J.-L. Beuchat and J.-M. Muller. Modulo m multiplication-addition : algorithms and FPGA implementation. *ELECTRONICS LETTERS 27th*, 40(11), 2004.

[BOPV03] L. Batina, S. B. Örs, B. Preneel, and J. Vandewalle. Hardware architectures for public key cryptography. *INTEGRATION, the VLSI journal*, 34(1-2) :1–64, 2003.

[BOS03] Johannes Blömer, Martin Otto, and Jean-Pierre Seifert. A new CRT-RSA algorithm secure against bellcore attacks. In *CCS '03 : Proceedings of the 10th ACM conference on Computer and communications security*, pages 311–320, New York, NY, USA, 2003. ACM Press.

[BP99] Thomas Blum and Christof Paar. Montgomery Modular Exponentiation on Reconfigurable Hardware. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 70–77. IEEE Computer Society Press, 1999.

[BP01] Thomas Blum and Christof Paar. High-Radix Montgomery Modular Exponentiation on Reconfigurable Hardware. *IEEE Transactions on Computers*, 50 :759–764, 2001.

[Bra95] Gilles Brassard. The Impending Demise of RSA ? *RSA Laboratories' CryptoBytes*, 1 :1–4, 1995.

[Bri90] E. F. Brickel. A Survey of Hardware Implementations of RSA. In *Advances in Cryptology–CRYPTO'89*, pages 368–70. Springer-Verlag, 1990.

[BS03a] V. Bunimov and M. Schimmler. Efficient Parallel Multiplication Algorithm for Large Integers. In *Euro-Par Proc. of the 15th IASTED International Conference PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS*. Springer-Verlag, August 2003.

[BS03b] V. Bunimov and M. Schimmler. A Simple Algorithm For Time Optimal Modular Multiplication and Exponentiation. In *Proc. of the 15th IASTED International Conference PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS*, November 2003.

[BS04a] V. Bunimov and M. Schimmler. Fast Modular Multiplication by Operand Changing. In *International Conference on Information Technology : Coding and Computing (ITCC'04)*, volume 2, 2004.

[BS04b] V. Bunimov and M. Schimmler. High Radix Modular Multiplication of Large Integers Optimised with Respect to Area and Time. In *In The 2004 International Conference on VLSI*, volume 2, June 2004.

[BST02] V. Bunimov, M. Schimmler, and B. Tolg. A Complexity-Effective Version of Montgomery's Algorithm. In *ISCA'02, Workshop on Complexity Effective Designs*, 2002.

[CAM+02] Paul Cunningham, Ross Anderson, Robert Mullins, George Taylor, and Simon Moore. Improving Smart Card Security Using Self-Timed Circuits. In *ASYNC'02 : Proceedings of the 8th International Symposium on Asynchronus Circuits and Systems*, page 211, Washington, DC, USA, 2002. IEEE Computer Society.

[CLN00] CLN is a C++ library for arbitrary precision arithmetic and other elementary functions. In implements that Schönhage-

Strassen multiplication algorithm. CLN Homepage, 2000. http ://clisp.cons.org/haible/packages-cln.html/.

[CLR94] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction à l'algorithmique.* DUNOD, 1994.

[CNPQ03] Mathieu Ciet, Michael Neve, Eric Peeters, and Jean-Jacques Quisquater. Parallel FPGA Implementation of RSA with residue number systems – can side-channel threats be avoided ? *IEEE Midwest International Symposium on Circuits and Systems*, 2003.

[CPA04] Stephen Craven, Cameron Patterson, and Peter Athanas. Super-sized Multiplies : How Do FPGAs Fare in Extended Digit Multipliers ? In *Proceedings of the 7th Annual Conference on Military and Aerospace Programmable Logic Devices*, Washington, DC, 2004. MAPLD 2005.

[CRC01] Koon-Shik Cho, Je-Hyuk Ryu, and Jun-Dong Cho. High-speed modular multiplication algorithm for RSA cryptosystem. *Industrial Electronics Society, IECON '01. The 27th Annual Conference of the IEEE*, 1 :479–483, 2001.

[CT65] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.*, 19 :297–301, 1965.

[Dav02] Jean-Pierre David. *Architecture synchronisée par les données pour système reconfigurable.* Université Catholique de Louvain (UCL), Belgique, Thèse doctorale, 2002.

[DF01] Jean-Francois Dhem and Nathalie Feyt. Hardware and Software Symbiosis Helps Smart Card Evolution. *IEEE Micro*, 21(6) :14–25, 2001.

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

[Dhe98] Jean-François Dhem. *Design of an Efficient Public-Key Cryptographic Library for RISC-based Smart Cards.* PhD thesis, May 1998.

[DL92] B. Dixon and A. K. Lenstra. Massively parallel elliptic curve factoring. *Proc. of EUROCRYPT'92, Springer, Berlin*, 658 :183–193, 1992.

[DQ00] Jean-François Dhem and Jean-Jacques Quisquater. Recent Results on Modular Multiplications for Smart Cards. In *CARDIS '98 : Proceedings of the The International Conference on Smart Card Research and Applications*, pages 336–352. Springer-Verlag, 2000.

[EKS03] A. El-Khashab and E. Swartzlander. The Modular Pipeline Fast Fourier Transform Algorithm and Architecture. In *IEEE Proc. of the 37th Asilomar Conference on Signals, Systems, and Computers*, California, USA, 2003.

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *Advances in Cryptology : Proc. CRYPTO'84, Springer, Berlin*, 196 :10–18, 1985.

[Eve90] Shimon Even. Systolic modular multiplication. *Proc. CRYPTO'90*, 537 :619–624, 1990.

[EW93] S. E. Eldridge and C. D. Walter. Hardware Implementation of Montgomery's Modular Multiplication Algorithm. *IEEE Transactions on Computers*, 42(6) :693–699, 1993.

[FFT01] FFT MegaCore Function User Guide. Altera Corporation, 2001. http ://www.altera.com/literature/ug/®t ug.pdf.

[FFT05] RFEL, IP Cores–HiSpeed Pipelined FFT's (Vectis Range). Xilinx, 2005. http ://www.rfel.com/products/Products_FFT_Hispeed.asp.

[FK03] A. Fournaris and O. Koufopavlou. Montgomery Modular Multiplier Architectures and Hardware Implementations for an RSA Cryptosystem. *proc. of 46th IEEE Midwest Symposium on Circuits & Systems '03 (MWSCAS 2003)*, 2003.

[FP99] William L. Freking and Keshab K. Parhi. A Unified Method for Iterative Computation of Modular Multiplication and Reduction Operations. In *ICCD '99 : Proceedings of the 1999 IEEE International Conference on Computer Design*, pages 80–87, 1999.

[FP00] William L. Freking and Keshab K. Parhi. Performance-Scalable Array Architectures for Modular Multiplication. In *ASAP '00 : Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pages 149–160, 2000.

[FPG01] Using Xilinx WebPACK Software to Create FPGA Designs for the XSA Board., Oct 2001. Introduction to WebPACK 4.1 for FPGAs.

[Gam87] GAMBIT SCHEME SYSTEM. Gambit Homepage, 1987.

[Gar03] Thierry Garcia. *Algorithnique parallèle du texte : du modèle systolique au modèle CGM.* PhD thesis, Université de Picardie Jules Vernne - Amiens, Faculté de Mathématiques et d'Informatique, 2003.

[GB73] B. Gold and T. Bially. Parallelism in fast Fourier transform hardware. *IEEE Trans. on Audio Electroacoustics*, AU-21 :5–16, 1973.

[Gro00] J. Grosschädl. The Chinese remainder theorem and its application in a high-speed RSA crypto chip. *Proceedings of the 16th Annual Computer Security Application Conference,*, page 384–393, 2000.

[GS66] W. Gentleman and G. Sande. Fast Fourier Transforms - For Fun and Profit. In *AFIPS Proceedings*, volume 29, pages 563–578, 1966.

[Har71] J. W. Hartwell. *A Procedure for Implementing the Fast Fourier Transform on Small Computers.* PhD thesis, Department of Electrical Engineering, Duke University, Durham, North Carolin, 1971.

[HAT03] M. Hasan, T. Arslan, and J.S. Thompson. A Novel Coefficient Ordering based Low Power Pipelined Radix-4 FFT Processor for Wireless LAN Applications. *IEEE Trans. on Consumer Electronics*, 49, 2003.

[Hey02] Howard M. Heys. A tutorial on linear and differential cryptanalysis. *Cryptologia*, XXVI(3) :189–221, 2002.

[HP00] Helena Handschuh and Pascal Paillier. Smart Card Crypto-Coprocessors for Public-Key Cryptography. In *CARDIS '98 : Proceedings of the The International Conference on Smart Card Research and Applications*, pages 372–379, London, UK, 2000. Springer-Verlag.

[HQ00] Gaël Hachez and Jean-Jacques Quisquater. Montgomery Exponentiation with no Final Subtractions : Improved Results. In *CHES '00 : Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, pages 293–301, London, UK, 2000. Springer-Verlag.

[HS03] A. Hiasat and A. Sweidan. Residue number system to binary converter for the moduli set $(2^{n-1}, 2^n-1, 2^n+1)$. *J. Syst. Archit.*, 49(1-2) :53–58, 2003.

[IDQ05] Understanding Quantum Cryptography. id Quantique, Switzerland, April 2005. Version 1.0.

[IM92] Keiichi Iwamura and Tsutomu Matsumoto. High-Speed Implementation Methods for RSA scheme. *Advances in Cryptology : Proc. of EUROCRYPT'92, Lecture Notes in Computer Science*, 658, 1992.

[IMI94] K. Iwamura, T. Matsumoto, and H. Imai. Montgomery modular multiplication method and systolic arrays suitable for modular exponentiation. *Elec. Comm., Japan*, 77(3) :40–50, 1994.

[IS98] K. M. Ibrahim and S. N. Saloum. An efficient residue to binary converter design. *IEEE Trans. Circuits Syst.*, 35 :1156–1158, 1998.

[JB97] Yong-Jin Jeong and Wayne P. Burleson. Vlsi array algorithms and architectures for RSA modular multiplication. *IEEE Transactions on Very Large Scale Integration (VLSI) systems*, 5(2) :211–217, 1997.

[JLQ99] M. Joye, A. K. Lenstra, and J.-J. Quisquater. Chinese remaindering based cryptosystem in the presence of faults. *J. Cryptol*, 4(12) :241–245, 1999.

[JS89] Christophe Jan and Guy Sabatier. *La sécurité informatique.* EYROLLES, 1989.

[Kah66] David Kahn. *The codebreakers : the story of secret writing.* New York : Scribner, 1966.

[KAK96] Ç. K. Koç, T. Acar, and B. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *Micro, IEEE*, 16(3) :26–33, 1996.

[Ker83] Auguste Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, IX :5–38, Jan 1883.

[KH98] C. K. Koç and C. Y. Hung. A Fast Algorithm for Modular Reduction. In *IEEE Proceedings : Computers and Digital Techniques*, volume 145, pages 265–271, 1998.

[Kim01] C. Kim. VHDL Implementation of Systolic Modular Multiplications on RSA Cryptosystem. Master's thesis, Department of Computer Science, The City College, City University of New York, 2001.

[KKSS00] S. Kawamura, M. Koike, F. Sano, and A. Shimbo. Cox-rower architecture for fast parallel Montgomery multiplication. *Advances in Cryptology : Proceedings of EUROCRYPT'00*, 1807 :523–538, 2000.

[Knu81] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 1981. Seminumerical Algorithms.

[KO63] A. Karatsuba and Y. Ofman. Multiplication of multi-digit numbers by automata. In *Soviet Physics-Doklady*, volume 7, page 595–596, 1963.

[Koç94] Ç. K. Koç. High-Speed RSA Implementation. In *RSA Laboratories*, Parkway, Redwood City, CA 94065-1031, November 1994. Version 2.0.

[Kob94] Neal Koblitz. A Course in Number Theory and Cryptography. In *Graduate Text in Mathematics, 2nd Edition*, volume 114, Berlin, Germany, 1994. Springer.

[Kor94] Peter Kornerup. A Systolic Linear-Array Multiplier for a Class of Right-Shift Algorithms. *IEEE Trans. on Computers*, 43(8) :892–898, 1994.

[Lee00] Sunggu Lee. *Design of Computers and Other Complex Digital Devices.* Prentice Hall, 2000.

[LHL03] Chin-Bou Liu, Chua-Huang Huang, and Chin-Laung Lei. Design and Implementation of Long-Digit Karatsuba's Multiplication Algorithm

Using Tensor Product Formulation. Master's thesis, Department of Information Engineering and Computer Science, Feng Chia University, Taiwan, May 2003.

[LNBO03] A. Lindström, M. Nordseth, L. Bengtsson, and A. Omondi. Arithmetic Circuits Combining Residue and Signed-Digit Representations. *Proc. of the Eighth Asia-Pacific Computer Systems Architecture Conference*, 2823, 2003.

[LS89] Stephen S. Leung and Michael A. Shanblatt. *ASIC system design with VHDL : a paradigm.* Kluwer Academic Publishers, 1989.

[MaSV97] A. Menezes and P. van Oorschot anc S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, Boca Raton, FL, USA, 1997.

[MMM+03] C. McIvor, M. McLoone, J. V. McCanny, A. Daly, and W. Marnane. Fast Montgomery Modular Multiplication and RSA Cryptographic Processor Architectures. In *37th Annual Asilomar Conference on Signals, Systems and Computers*, 2003.

[MMM04] C. McIvor, M. McLoone, and J. V. McCanny. Modified Montgomery Modular Multiplication and RSA Exponentiation. In *Techniques IEE Proceedings–Computers & Digital Techniques*, volume 151, pages 402–408, 2004.

[Moe76] Robert T. Moenck. Practical Fast Polynomial Multiplication. In *SYMSAC '76 : Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, pages 136–148, 1976.

[Mon85] Peter Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170) :519–21, 1985.

[NEG04] Christophe NEGRE. *Opérateurs Arithmétiques pour la Cryptographie Basée sur les Courbes Elliptiques.* PhD thesis, Université Montpellier II, France, 2004.

[NF03] M. Niimura and Y. Fuwa. High Speed Modulo Calculation Algorithm with, Radix-2^k SD Number. *JOURNAL OF FORMALIZED MATHEMATICS*, 15, 2003.

[NM96] David Naccache and David M'Raïhi. Arithmetic co-processors for public-key cryptography : The state of the art. Second Smart Card Research and Applications Conference (CARDIS '96), 1996.

[OD04] Etienne Ogoubi and Jean-Pierre David. Automatic synthesis from high level ASM to VHDL : a case study. In *North East Workshop on Circuits and Systems (NewCAS)*, Montreal, Canada, 2004.

[Oru95] H. Orup. Simplifying Quotient Determination in High-Radix Modular Multiplication. In *ARITH '95 : Proceedings of the 12th Symposium on Computer Arithmetic*, pages 193–199, Washington, USA, 1995. IEEE Computer Society.

[PB85] P. W. Purdom and C. A. Brown. *The Analysis Of Algorithms.* Holt, Rinehart and Winston, 1985.

[Pie95] S. J. Piestrak. A high speed realization of residue to binary number system conversion. *IEEE Trans. Circuits Syst. II*, 41 :661–663, 1995.

[PP95] K. Posch and R. Posch. Modulo Reduction in Residue Number Systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5) :449–454, 1995.

[QC82] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosyste. *Electronics Letters*, 18 :905–907, 1982.

[Roc00] D. N. Rockemore. The FFT–An Algorithm the Whole Family Can Use. *Computing in Science and Engineering*, 2(1) :60–64, 2000.

[RSA78] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2) :120–126, 1978.

[Sed87] H. Sedlak. The RSA cryptography processor. volume 304, page 95–105, Berlin, 1987. Springer.

[SG03] P. Sebah and X. Gourdon. π and its computation through the ages. Technical report, 2003.

[Sha48] Claude E. Shannon. A Mathematical Theory of Communication. *Belle Systems Technical Journal*, 27, July, October 1948.

[Sha49] Claude E. Shannon. Communication Theory of Secrecy Systems. *Belle Systems Technical Jornal*, 28 :656–715, Oct 1949.

[Spi05] Richard J. Spillman. *Classical and Contemporary Cryptology.* Prentice Hall, 2005.

[SS71] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7 :281–292, 1971.

[Sti03] Douglas Stinson. *Cryptographie : théorie et pratique.* Vuibert, 2003.

[SV93] M. Shand and J. Vuillemin. Fast implementations of RSA cryptography. In *Proc. of the 11th IEEE Symposium on Computer Arithmetic*, pages 252–259, 1993.

[Sys] SystemC. http ://www.systemc.org.

[Tak98] T. Takagi. Fast RSA-type cryptosystem modulo $p^k q$. *Proceedings of CRYPTO'98, Lecture Notes in Computer Science*, 1462 :318–326, 1998.

[TK83] Y. Tamura and Y. Kanada. Calculation of π to 4, 194, 293 Decimals Based on Gauss-Legendre Algorithm. Technical report, Computer Center, University of Tokyo, 1983.

[TK99] A. F. Tenca and Ç. K. Koç. A scalable architecture for Montgomery multiplication. *Proc. of Cryptographic Hardware and Embedded Systems (CHES 1999), Springer, Berlin*, 1717 :94–108, 1999.

[TK03] A. F. Tenca and C. K. Koc. A scalable architecture for modular multiplication based on montgomery's algorithm. *IEEE Trans. on Computers*, 52(9) :1215–1221, 2003.

[TW02] Wade Trappe and Lawrence C. Washington. *Introduction to cryptography : with coding theory.* Prentice Hall, 2002.

[TY92] N. Takagi and S. Yajima. Modular multiplication hardware algorithms with a redundant representation and their application to RSA cryptosystem. *IEEE Transactions on Computers*, 41(7) :887–891, 1992.

[UBF+03] C. A. Uller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert. Fault Attacks on RSA with CRT : Concrete Results and Practical Countermeasures. In *CHES '02 : Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 260–275. Springer-Verlag, 2003.

[VR94] B. Vinnakota and V. V. B. Rao. Fast conversion techniques for binary to RNS. *IEEE Trans. Circuits Syst. I*, 41(927–929), 1994.

[Š03] Martin Šimka. RSA Implementation on Reconfigurable Hardware. In *Proceedings of the III. PhD students conference*, pages 81–82, 2003.

[VVDJ89] A. Vandemeulebroecke, E. Vanzieleghem, T. Denayer, and P. G. A. Jespers. A single chip 1024 bits RSA processor. In *Advances in Cryptology : Proceedings of EUROCRYPT'89, Lecture Notes in Computer Science*, volume 434, page 219–236, Springer,Berlin, 1989. Springer.

[Wag04] David Wagner. Cryptanalysis of a provably secure CRT-RSA algorithm. In *CCS '04 : Proceedings of the 11th ACM conference on Computer and communications security*, pages 92–97, New York, NY, USA, 2004. ACM Press.

[Wal93] Colin. D. Walter. Systolic Modular Multiplication. *IEEE Trans. on Computers*, 42 :376–378, 1993.

[Wal99] Colin D. Walter. Montgomery's Multiplication Technique : How to Make It Smaller and Faster. *Proc. of the First International Workshop on Cryptographic Hardware and Embedded Systems*, 1717 :80–93, 1999.

[Wal00] Colin. D. Walter. An improved linear systolic array for fast modular exponentiation. *IEEE Comput. Digital Tech*, 147(5), 2000.

[Wan00] Yuke Wang. Residue-to-Binary Converters Based on New Chinese Remainder Theorems. *Trans. Circuits Syst. II*, pages 197–206, 2000.

[WCS02] S. Wei, S. Chen, and K. Shimizu. Fast Modular Multiplication Using Booth Recoding Based on Signed-Digit Number Arithmetic. In *Proc. of 2002 IEEE Asia Pacific Conference On Circuits and Systems*, volume 2, pages 31–36, 2002.

[WCSG03] X. Wu, H. Chen, Y. Sun, and W. Gai. A Fully-Pipeline Linear Systolic Architecture for Modular Multiplier in Public-Key Crypto-Systems. *Journal of VLSI Signal Processing Syst.*, 33(1) :191–7, 2003.

[WHW01] C.-H. Wu, J.-H. Hong, and C.-W. Wu. RSA cryptosystem design based on the Chinese Remainder Theorem. In *ASP-DAC '01 : Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 391–395, 2001.

[WS00] S. Wei and K. Shimizu. A novel residue arithmetic hardware algorithm using a signed-digit number representation. *IEICE Trans. inf. & syst.*, E83-D(12) :2056–2064, 2000.

[WS01] S. Wei and K. Shimizu. Fast residue arithmetic multipliers based on signed digit number system. *IEEE 8th Conf. Electronics, Circuits and Systems*, 1 :263–266, 2001.

[Xil05] Xilinx products, Mars 2005. http ://www.xilinx.com/products/fpgas.

[Yas02] V. V. Yaschenko. *Cryptography : An Introduction.* American Mathematical Society, 2002.

[YL00] C. Yap and C. Li. Quickmul : Practical FFT-based Integer Multiplication. Technical report, Department of Computer Science, Courant Institute, New York University, NY, USA, October 2000.

Annexe A

Systèmes de chiffrement

A.1 Chiffrement de César

Le chiffrement de César consiste simplement à décaler les lettres de l'alphabet de quelques rangs vers la droite ou vers la gauche. Par exemple, décalons les lettres de 3 rangs vers la gauche, comme le faisait Jules César (d'où le nom de ce chiffre) :

Clair	A	B	C	D	E	F	G	H	I	J	K	L	M
Chiffré	d	e	f	g	h	i	j	k	l	m	n	o	p
Clair	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Chiffré	q	r	s	t	u	v	w	x	y	z	a	b	c

TAB. A.1 – Chiffrement de César

A.2 Chiffrement mono-alphabetique

Une autre manière très classique de chiffrer des messages consiste à remplacer une lettre par une autre, en utilisant un alphabet désordonné. C'est un chiffrement mono-alphabétique ou de substitution. On pourrait, par exemple, utiliser la grille de chiffrement ci-dessous :

Clair	A	B	C	D	E	F	G	H	I	J	K	L	M
Chiffré	b	t	u	e	q	v	z	a	r	w	g	o	n
Clair	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Chiffré	c	l	k	j	s	x	d	m	h	p	i	f	y

TAB. A.2 – Chiffrement mono-alphabetique

A.3 Matrice ou carré de Vigenère

La lettre de la clé est dans la ligne tout en haut, la lettre du message clair est dans la colonne la plus à gauche. Le texte chiffré s'obtient en prenant l'intersection de

la ligne qui commence par la lettre à coder avec la colonne qui commence par la lettre de la clé. Dès que l'on atteint la fin de la clé, on recommence à la première lettre. Cette table montre que "TJB FTVO" est le message chiffré de "IBN SINA" avec la clé "LION".

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	**O**	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	**J**	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	**T**	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	**V**	W	X	Y	Z	A	**B**	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	**F**	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

FIG. A.1 – Matrice ou carré de Vigenère

Annexe B

Permutation par inversion de bits (miroir)

Algorithme 15 Algorithme de la fonction Mirroir

1: Entrée : i
2: Sortie : j {Miroir de i}
3: $h = \log n$ {Le nombre de bits qui représente n}
4: $j = 0$
5: **while** $(h > 0)$ **do**
6: $\quad j = j \ll 1$ {Préparation d'un nouveau bit à droite}
7: $\quad j = j|(n\&1)$ {Positionnement de ce bit}
8: $\quad n = n \gg 1$ {Effacement du bit de droite de n}
9: $\quad h = h - 1$
10: **end while**
11: Retourner j

Annexe C

Exécution de la procédure de la multiplication des entiers de 1024 bits par la FFT

```
Pour n=1024 bits, on aura k= 64 coefficients de  l= 16 bits

modulo=w^k +1 =  18446744073709551617

a[0]=1, b[0]=3
a[1]=2, b[1]=4
a[2]=3, b[2]=5
...

a[62]=63, b[62]=65
a[63]=64, b[63]=66

Le polynôme a en Hex est:
40003F003E003D003C003B003A00390038003700360035003400330032003100300002F002
E002D002C002B002A00290028002700260025002400230022002100200001F001E001D001C
001B001A00190018001700160015001400130012001100100000F000E000D000C000B000A0
00900080007000600050004000300020001

Le polynôme b en Hex est:
4200410040003F003E003D003C003B003A00390038003700360035003400330032003100300
0002F002E002D002C002B002A00290028002700260025002400230022002100200001F001E
001D001C001B001A00190018001700160015001400130012001100100000F000E000D000C0
00B000A000900080007000600050004000300

racineInv:9223372036854775809
```

```
tailleInv:18302628885633695745

La FFT de a est telle que:
a[0]=2080
a[1]=18446744073709551585
a[2]=18446743936270598113
...

a[124]=1475739641345485098
a[125]=17708874195312448688
a[126]=10248191152060862053
a[127]=136
============================

La FFT de b est telle que:
b[0]=2208
b[1]=18446744073709551585
b[2]=18446743936270598113
...

b[124]=12543786092443163745
b[125]=2951478929472859724
b[126]=4099276460824344850
b[127]=144
============================

Après la multiplication point à point:
c[0]=4592640
c[1]=1024
c[2]=8796093022208
...
```

```
c[124]=15082070726152195119
c[125]=5371679102776959958
c[126]=16624843424454289273
c[127]=19584
============================

Les racines nièmes de l'unité:
powerRacine[0]=8000000000000001
powerRacine[1]=c000000000000001
powerRacine[2]=f000000000000001
powerRacine[3]=ff00000000000001
powerRacine[4]=ffff000000000001
powerRacine[5]=ffffffff00000001
powerRacine[6]=10000000000000000
==============================

Après la multiplication par tailleInv
c[0]=3    c[1]=10    c[2]=22    c[3]=40
c[4]=65    c[5]=98    c[6]=140
c[7]=192    c[8]=255    c[9]=330
c[10]=418    c[11]=520    c[12]=637
c[13]=770    c[14]=920    c[15]=1088
c[16]=1275    c[17]=1482    c[18]=1710
c[19]=1960    c[20]=2233    c[21]=2530
c[22]=2852    c[23]=3200    c[24]=3575
c[25]=3978    c[26]=4410    c[27]=4872
c[28]=5365    c[29]=5890    c[30]=6448
c[31]=7040    c[32]=7667    c[33]=8330
c[34]=9030    c[35]=9768    c[36]=10545
c[37]=11362    c[38]=12220    c[39]=13120
c[40]=14063    c[41]=15050    c[42]=16082
```

```
c[43]=17160    c[44]=18285    c[45]=19458
c[46]=20680    c[47]=21952    c[48]=23275
c[49]=24650    c[50]=26078    c[51]=27560
c[52]=29097    c[53]=30690    c[54]=32340
c[55]=34048    c[56]=35815    c[57]=37642
c[58]=39530    c[59]=41480    c[60]=43493
c[61]=45570    c[62]=47712    c[63]=49920
c[64]=51933    c[65]=53878    c[66]=55754
c[67]=57560    c[68]=59295    c[69]=60958
c[70]=62548    c[71]=64064    c[72]=65505
c[73]=66870    c[74]=68158    c[75]=69368
c[76]=70499    c[77]=71550    c[78]=72520
c[79]=73408    c[80]=74213    c[81]=74934
c[82]=75570    c[83]=76120    c[84]=76583
c[85]=76958    c[86]=77244    c[87]=77440
c[88]=77545    c[89]=77558    c[90]=77478
c[91]=77304    c[92]=77035    c[93]=76670
c[94]=76208    c[95]=75648    c[96]=74989
c[97]=74230    c[98]=73370    c[99]=72408
c[100]=71343   c[101]=70174   c[102]=68900
c[103]=67520   c[104]=66033   c[105]=64438
c[106]=62734   c[107]=60920   c[108]=58995
c[109]=56958   c[110]=54808   c[111]=52544
c[112]=50165   c[113]=47670   c[114]=45058
c[115]=42328   c[116]=39479   c[117]=36510
c[118]=33420   c[119]=30208   c[120]=26873
c[121]=23414   c[122]=19830   c[123]=16120
c[124]=12283   c[125]=8318    c[126]=4224
c[127]=0
```

Après l'évaluation, le produit de a et b est enfin:

1080207E2FFB3EF84D765B7668F97600828C8E9E9A37A558B002BA36C3F5CD40D618DE7EE
673EDF8F50EFBB701F207C10D25121F16B01AD91E9B21F724EE278129B12B7F2CEC2DF92E
A72EF72EEA2E812DBD2C9F2B282959273324B721E61EC11B49177F13640EF90A3F0536FFE
1FA40F454EE1EE79FE0D8D9CAD276CADDC300BA60B202A9E5A2089A6A930A8BE785007E54
77E271A96BA865DE604A5AEB55C050C84C02476D43083ED23ACA36EF33402FBC2C6229312
6282346208A1DF31B801930170214F51308113A0F8A0DF70C800B2409E208B907A806AE05
CA04FB044003980302027D020801A2014A00FF00C0008C0062004100280016000A0003