Numéro d'ordre : Année 1998

THESE

présentée par

Laurent-Stéphane DIDIER

devant l'Université de Provence

pour l'obtention

du **DIPLOME DE DOCTORAT**

Spécialité : **Mathématiques - Informatique**

Division

et

Multiplication Modulaire

en

Représentation Modulaire

Date de la soutenance : 12 Janvier 1998

Composition du jury :

Rapporteurs :	Jean-Michel Muller
	Jean Vignes
Examinateurs :	Jean-Claude Bajard (directeur de thèse)
	Bertrand Hochet
	Pierre Siegel

Thèse préparée à l'Université de Provence

au sein du Laboratoire d'Informatique de Marseille

LIM, ESA CNRS n° 6077

A mes parent,

A ma famille,

A mes amis.

Je remercie vivement les membres du jury :

- Bertrand Hochet qui a fait le voyage de Suisse pour participer à mon jury et qui m'a fait l'honneur d'en être le président.

- Pierre Siegel qui m'a acceuilli dans son laboratoire

- Jean Vignes qui m'a fait l'honneur d'être rapporteur de ma thèse malgrès un emploi du temps que je sais très chargé.

- Jean-Michel Muller sans qui je n'aurai probablement jamais fait d'arithmétique des ordinateurs. Je le remercie chaleureusement pour tous les conseils qu'il m'a donné et le soutien qu'il m'a apporté à l'étranger. Je le remercie par ailleurs d'avoir accepté d'être rapporteur de ma thèse.

- Enfin j'adresse un grand merci à Jean-Claude Bajard qui a encadré mon travail de thèse au cours de ces dernières années. Pour sa confiance, pour le soutien qu'il m'a donné en toute occasion, pour la science qu'il m'a appris à découvrir, je lui suis reconnaissant.

Je remercie aussi Miloš Ercegovac et Peter Kornerup de leurs précieux conseils sur l'arithmétique des ordinateurs et de l'aide qu'ils m'ont apporté lors de leurs visites et dans leurs laboratoires.

Depuis longtemps je souhaite remercier Roger Anglada de son amitié et de tout le soutien qu'il m'a donné depuis si longtemps. Merci aussi pour ces récits qui' m'ont fait voyager au delà des mers et m'ont beaucoup appris.

Mercie à Karine, la source de bonne humeur du bureau d'à côté.

Ce serait injustice si j'oubliais de remercier le thésard du LIM-Nord pour la bonne ambiance qu'ils ont su apporter. Merci à Fabrice, Laurent et Lionel d'avoir accepté de subir les répétitions de mes exposés et de leur judicieux conseils. Merci à Félix pour l'aide qu'il m'a apporté pour les enseignements au cours de la rédaction de ce mémoire et pour les fromages corse "doux" qui ont agrémenté les plus récents des pique-niques des thésards. Un grand merci enfin à Line ainsi qu'à Romain pour leur chaleureuse amitié.

Merci aussi à Christophe, Hervé, Jean-Yves, Lionel, Pascale et Stéphan de leur amitié sans faille et qui m'ont permis d'arriver au bout de cette thèse.
Enfin merci à mes parents sans qui rien n'aurait été possible.

Table des matières

A Notations **7**

Introduction **9**

1 Arithmétique modulaire, un état de l'art **11**
1.1 Définition . . . 11
1.2 Conversions . . . 13
1.2.1 Conversion des systèmes de numération de position vers les systèmes modulaires . . . 13
1.2.2 Conversion des systèmes modulaires vers les systèmes de numération de position . . . 17
1.2.3 Extension et changement de base . . . 25
1.3 Opérations . . . 27
1.3.1 Addition, Soustraction et Multiplication . . . 27
1.3.2 Implémentations . . . 29
1.3.3 Comparaison . . . 32
1.3.4 Division exacte . . . 41
1.3.5 Bases incomplètes de Gamberger . . . 42

2 Division en représentation modulaire **45**
2.1 Définition de l'arithmétique pseudo-flottante . . . 45
2.1.1 Définition . . . 46
2.1.2 Calculs en arithmétique pseudo-flottante . . . 47
2.1.3 Construction de la représentation pseudo-flottante . . . 47
2.1.4 Implémentation de la construction . . . 51
2.2 Division . . . 53
2.2.1 Algorithme de division en représentation modulaire . . . 53
2.2.2 Analyse de l'algorithme de division . . . 57
2.2.3 Performances . . . 59
2.3 Autres algorithmes de division . . . 60
2.3.1 Division de Gamberger . . . 60
2.3.2 Division de Chiang et Lu . . . 63
2.3.3 Division de Hitz et Kaltofen . . . 65
2.3.4 Division de Chren . . . 67
2.3.5 Division de Hung et Parhami . . . 69
2.4 Conclusion . . . 70

3 Multiplication modulaire RNS **71**
3.1 Applications cryptographiques . . . 71
3.1.1 Algorithme RSA . . . 72
3.1.2 Protocole d'identification de Fiat-Shamir . . . 74
3.2 Multiplication modulaire . . . 74
3.2.1 Multiplication modulaire par accumulations et divisions . . . 75
3.2.2 Multiplication modulaire de Montgomery . . . 76
3.3 Multiplication modulaire RNS . . . 78
3.3.1 Présentation de l'algorithme . . . 78
3.3.2 Preuve de l'algorithme . . . 82
3.3.3 Application . . . 83
3.4 Implémentations . . . 85
3.4.1 Implémentation sur un anneau de processeurs . . . 85
3.4.2 Implémentation sur un bus de processeurs . . . 89
3.5 Autres algorithmes . . . 92
3.5.1 Réduction de Posch et Posch . . . 92
3.5.2 Réduction par division . . . 93
3.5.3 Décryptage de Chren . . . 94
3.6 Conclusion . . . 94

Conclusion **97**

Table des figures

1.1 Conversion pipelinée de la représentation en numération de position vers modulaire 15
1.2 Conversion logarithmique de la représentation en numération de position vers modulaire 16
1.3 Conversions simples de la représentation en numération de position vers une représentation modulaire 17
1.4 Conversion représentation modulaire vers numération de position 19
1.5 Conversion “Mixed Radix” de Szabo et Tanaka 23
1.6 Conversion “Mixed Radix” basée sur le théorème des restes chinois 24
1.7 Multiplication modulaire par table 30
1.8 Multiplication modulaire par index 30
1.9 Addition modulaire avec deux additionneurs 31
1.10 Une architecture de la détermination de la parité d’un nombre par la méthode de Mi Lu et Jen-Shiun Chiang 35
1.11 Arrangement des nombres dans le plan avec les moduli $\{13, 7\}$ 36
1.12 Ordre des diagonales principales dans le plan avec les moduli $\{13, 7\}$ 37
1.13 Implémentation du calcul de la fonction $\mathcal{D}(X)$ 38
1.14 Répartition des entiers relatifs – Comparaison de Hung et Parhami 39

2.1 Architecture de l’implémentation de la conversion en représentation pseudo-flottante 52
2.2 Une première étape de l’encadrement par dichotomie de X 64

3.1 Protocole de chiffrage à clef publique 72
3.2 Représentation binaire et repésentation modulaire pour la représentation de messages 84
3.3 Anneau de six processeurs 85
3.4 Bus de processeurs 89
3.5 Table d’allocation des tâches de la conversion sur un bus de six processeurs 90
3.6 Table d’allocation des tâches de la multiplication modulaire sur un bus de six processeurs 91

Liste des tableaux

1.1 Caractéristiques de méthodes de conversion de numération de position en arithmétique modulaire . . . 16
1.2 Caractéristiques de la conversion d'un nombre en arithmétique modulaire vers une arithmétique de position . . . 18
1.3 Caractéristiques de la conversion "Mixed Radix" de Szabo et Takana . . . 23
1.4 Caractéristiques de l'extension de base de Shenoy et Kumaresan . . . 26
1.5 Caractéristiques de la méthode de comparaison de Mi Lu et Jen-Shiun Chiang 36
1.6 Caractéristiques de la méthode de comparaison de Dimauro, Impedovo et Pirlo . . . 38
1.7 Caractéristiques de la méthode de comparaison de Hung et Parhami . . . 41

3.1 Description des tâches effectuées sur le processeur j . . . 86
3.2 Table d'allocation des tâches sur un anneau de six processeurs . . . 87
3.3 Description des tâches effectuées sur le processeur j pour la conversion "mixed radix" sur un bus de processeurs . . . 90
3.4 Description des tâches effectuées sur le processeur j pour la multiplication modulaire sur un bus de processeurs . . . 91

Annexe A

Notations

$[X]$	Partie fractionnaire de X
$[X]_\alpha$	α premiers chiffres de la partie fractionnaire de X
$\lceil X \rceil$	Partie entière de X par excès
$\lfloor X \rfloor$	Partie entière de X par défaut (partie entière usuelle)
$\|X\|_\alpha$	Reste de la division euclidienne de X par α
$(X)_\alpha^{-1}$	Inverse modulaire de X modulo α
X_{RNS}	Représentation modulaire de l'entier X
x_i	i^{ème} reste de l'entier X
x'_i	i^{ème} chiffre de la représentation MRS de l'entier X
m_i	i^{ème} modulus
$m_{i,j}^{-1}$	Inverse modulaire de m_i modulo m_j
$\mathcal{B}_m^n$	Base modulaire constituée des n moduli m_i
M	Produit des moduli de la base modulaire $\mathcal{B}_m^n$
M_i	Produit des moduli de la base modulaire $\mathcal{B}_m^n$ divisé par m_i
$+_\alpha$	Addition modulaire modulo α
$\times_\alpha$	Multiplication modulaire modulo α
$+_{RNS}$	Addition en arithmétique modulaire
$-_{RNS}$	Soustraction en arithmétique modulaire
$\times_{RNS}$	Multiplication en arithmétique modulaire

Introduction

De nos jours, les microprocesseurs font partie de notre quotidien et nous les retrouvons dans les télécommunications, l'industrie aéronautique et spatiale, les systèmes de réception de télévision numérique. Cette prolifération est essentiellement due à une règle simple : faire des calculs toujours plus rapides ou plus précis, voire exacts avec des composants toujours plus petits. Les moyens d'atteindre ce but ne manquent pas, que ce soit au niveau des techniques d'intégration de circuits ou de la définition de l'architecture des composants. Mais les progrès ne sont pas que technologiques et la recherche dans le domaine de l'arithmétique des ordinateurs permet d'obtenir un gain très appréciable.

En effet, on remarque que la fonction première des systèmes électroniques (microprocesseur, carte à puce, décodeur de télévision payante, calculateur embarqué, etc.) est de faire des calculs. Selon le choix que l'on fait sur la représentation des nombres qu'ils manipulent, ces calculs seront plus ou moins faciles à effectuer. Pour s'en convaincre il suffit de se souvenir de la numération en chiffres romains et de la difficulté que l'on a de calculer avec. Le propos de l'arithmétique des ordinateurs est de trouver des représentations des nombres, des algorithmes et des architectures de circuits permettant de calculer plus efficacement, c'est à dire de manière plus rapide, plus compacte ou bien plus précise. Le travail que nous présentons dans cette thèse se place dans ce cadre.

La recherche en arithmétique des ordinateurs à donné naissance a des représentations aux propriétés interessantes. Ainsi, les représentations redondantes à chiffres signés dues à *Avizienis* sont des notations en numération de position qui permettent, pour certaines, d'effectuer des additions en temps constant. La représentation logarithmique des nombres présentée la première fois par *Swartzlander* et *Alexpoulos* offre l'avantage de pouvoir ramener les multiplications à de simples additions en numération de position, mais ne permet pas d'effectuer des additions efficacement. La représentation modulaire, dont l'origine remonte aux temps anciens, permet de substituer les calculs sur de très grands nombres par des opérations sur de petites valeurs pour la détermination de l'addition et de la multiplication. Ces opération peuvent être effectuées très rapidement en parallèle. Nous nous intéresserons ici à cette représentation des nombres.

Nous proposons dans cette thèse un nouvel algorithme de division dont les performances en font un des plus efficaces trouvés à ce jour. D'autre part nous donnons ici le premier algorithme de multiplication modulaire en représentation modulaire basé sur l'algorithme de *Montgomery*. Ces algorithmes rendent possible l'emploi de cette arithmétique dans des applications qui jusque là lui étaient interdites. Ainsi, certaines applications cryptographiques pour lesquelles elle ne semblait pas adaptée sont désormais exploitables dans cette représentation.

Cette thèse s'articule autour de trois chapitres. Dans le premier nous présentons un état de l'art sur la représentation modulaire. Nous y abordons différentes méthodes de conversion vers les systèmes de numération de position ainsi qu'entre systèmes modulaires. Nous traiterons également de diverses architectures proposées dans la littérature scientifique pour le calcul des additions et multiplications dans cette représentation.

Nous décrivons dans le second chapitre le fonctionnement de notre algorithme de division qui utilise une représentation “pseudo-flottante”. Nous confronterons cet algorithme aux divisions proposées jusqu'à présent.

Enfin, nous traiterons de la multiplication modulaire en représentation modulaire et de ses application cryptographiques dans le dernier chapitre. Nous proposerons quelques implémentations parallèles de notre algorithme destinées aux anneaux de processeurs et aux machines parallèles à bus.

Chapitre 1

Arithmétique modulaire, un état de l'art

L'histoire de l'arithmétique modulaire a près de 18 siècles. Cette représentation des nombres a pour origine les travaux effectués par le mathématicien Chinois *Sun Tsŏ* qui a travaillé sur les systèmes de congruence vraisemblablement entre 280 et 473 ap. JC. [Tay84]. Le théorème proposé par *Sun Tsŏ* a été généralisé par un autre chinois *Shu Shu Chiu Chang* en 1247 [Knu81] et ce n'est qu'en 1734 que la version moderne de ce théorème, connu de nos jours sous le nom de "Théorème des Restes Chinois", a été démontrée par *Leonard Euler* [BEPP97]. L'utilisation de l'arithmétique modulaire pour représenter les nombres a été évoquée bien plus récemment en 1959 par *Garner* dans sa thèse de doctorat [Gar59].

Dans ce chapitre nous abordons les éléments de base de la représentation modulaire des nombres[1]. Afin d'avoir quelques éléments de comparaison des méthodes et algorithmes présentés par la suite nous nous sommes attachés à donner la complexité en temps et en espace de leur implémentation. Dans certains cas, ces complexités sont celles données par les auteurs. Dans d'autres cas, elles sont basées sur une implémentation simple que nous avons déduit de leur algorithme. Dans ce dernier cas elles ne sont pas optimisées et permettent de fournir un aperçu de l'ordre de grandeur de leur complexité.

Ce chapitre est organisé de la façon suivante. Nous définissons la représentation modulaire dans la section 1.1 où nous présentons par ailleurs le théorème des restes chinois. Ensuite, dans la section 1.2 nous exposons quelques unes des méthodes de conversions entre les systèmes modulaires et les représentations en numération de position. Enfin nous montrons dans la section 1.3 les différentes réalisations des opérations d'addition, de multiplication, de comparaison ainsi, que le cas particulier de la division. Nous présenterons aussi dans cette section une extension de la représentation modulaire proposée par *D. Gamberger*.

1.1 Définition

Le principal intérêt de cette arithmétique est de permettre d'effectuer les multiplications et les additions très rapidement. En effet cette représentation des nombres s'affranchit du principal handicap des représentations en numération de position qu'est la

1. Residue Number System ou RNS en anglais

propagation des retenues. De plus étant donné qu'il ne s'agit pas d'une représentation en numération de position, il n'est pas nécessaire de déterminer et d'additionner des produits partiels pour la multiplication.

Pour construire cette représentation, l'idée est de se doter d'un ensemble d'entiers $\{m_1, m_2, \ldots, m_n\}$ premiers entre eux et d'effectuer les calculs non pas sur les nombres eux-mêmes mais sur leur *restes* modulo ces entiers. Ainsi un nombre X est caractérisé par ses restes $X \bmod m_1, X \bmod m_2, \ldots, X \bmod m_n$. Par la suite ces restes seront notés de la façon suivante :

$$\begin{array}{l} x_1 = X \bmod m_1 \\ x_2 = X \bmod m_2 \\ \ldots\ldots\ldots\ldots\ldots \\ x_n = X \bmod m_n \end{array}$$

A priori, cet ensemble de restes x_i ne caractérise pas de façon unique tout entier dans $\mathbb{N}$. Le théorème suivant nous garantit l'unicité de cette représentation sous certaines conditions.

Théorème 1.1 (Théorème des Restes Chinois - TRC) *Soient $m_1, m_2, \ldots, m_n$ des entiers positifs ou nuls premiers entre eux, c'est à dire tels que :*

$$PGCD(m_j, m_k) = 1 \quad pour \quad j \neq k$$

Soit $M = \prod_{i=1}^{n} m_i$, soit un entier relatif a et soient $x_1, x_2, \ldots, x_n$ des entiers positifs tels que $x_i < m_i$. Alors il existe un unique entier X qui vérifie les conditions suivantes :

$$a \leq X < a + M \quad et \quad x_i = X \bmod m_i \quad pour \quad 1 \leq i \leq n$$

Preuve. Soit $M_i = \frac{M}{m_i}$ pour $1 \leq i \leq n$. Puisque m_i et M_i sont premiers entre eux, l'inverse de M_i modulo m_i existe [Knu81]. On le note $(M_i)_i^{-1}$. Alors le nombre X, défini comme suit, vérifie les conditions énoncées dans le théorème :

$$X = a + (x_1(M_1)_1^{-1}M_1 + x_2(M_2)_2^{-1}M_2 + \ldots + x_n(M_n)_n^{-1}M_n) \bmod M \qquad (1.1)$$

■

Ainsi, tout entier X, compris dans un intervalle de largeur M, peut être représenté de façon unique par les petits entiers que sont ses restes. On notera par la suite X_{RNS} la suite d'entiers qui constitue la représentation modulaire de l'entier X. Cette représentation n'a de sens que si l'ensemble des *moduli* $\{m_1, m_2, \cdots, m_n\}$ est défini. On appelle cet ensemble la *base* de la représentation modulaire ou la base du système modulaire. On la note $\mathcal{B}_m^n$. Par la suite nous considérerons, sauf indication contraire, que $a = 0$.

$$X_{RNS} = \{x_1, x_2, \cdots, x_n\}$$

Exemple :

Dans la base $\mathcal{B}_m^3 = \{7, 11, 13\}$ on a :

$W = 1000$	$W = 1111101000_2$	$W = \{6, 10, 12\}_{RNS}$	$W = \{0110_2, 1010_2, 1100_2\}_{RNS}$
$X = 783$	$X = 1100001111_2$	$X = \{6, 2, 3\}_{RNS}$	$X = \{0110_2, 0010_2, 0011_2\}_{RNS}$
$Y = 108$	$Y = 0001101100_2$	$Y = \{3, 9, 4\}_{RNS}$	$Y = \{0011_2, 1001_2, 0100_2\}_{RNS}$
$Z = 12$	$Z = 0000001100_2$	$Z = \{5, 1, 12\}_{RNS}$	$Z = \{0101_2, 0001_2, 1100_2\}_{RNS}$ ■

Remarque : Le nombre de bits nécessaires à la représentation modulaire d'un nombre est du même ordre que le nombre de bits utilisés pour sa représentation binaire. Si l'on considère une base composée de n moduli de p bits chacun, le plus grand nombre représentable dans ce système est majoré par $M = \prod_{i=1}^{n} m_i$. Il est par conséquent composé d'au plus $p \times n$ bits lorsqu'il est représenté en binaire. Or la représentation d'un tel nombre en arithmétique modulaire nécessite n restes de p bits.

Le théorème des restes chinois ne tient pas compte du signe des entiers représentés. Il nous apprend qu'il est possible d'exprimer des entiers compris dans n'importe quel intervalle de longueur M. Par conséquent, en choisissant a de façon appropriée, il est possible de représenter les entiers négatifs. Par exemple on peut représenter un intervalle symétrique ayant autant de nombres positifs que de nombres négatifs. Si la base du système n'est constituée que de nombres impairs ce partage est très facile à obtenir. Si on considére les nombres négatifs X comme étant les nombres tels que :

$$a + 0 \leq X < a + \left\lfloor \frac{M}{2} \right\rfloor$$

les nombres positifs Y tels seront :

$$a + \left\lceil \frac{M}{2} \right\rceil \leq Y < a + M$$

et le nombre $0 = a + \left\lfloor \frac{M}{2} \right\rfloor$. Ainsi l'opposé d'un nombre positif X est défini comme étant $M - X$. Une autre répartition des nombres positifs et négatifs est proposée dans la sous-section 1.3.3.

1.2 Conversions

Les systèmes modulaires sont souvent associés à des représentations en numération de position pour des applications de traitement du signal [Lo91] [SPS92]. En effet, dans ce type d'application demandant beaucoup d'additions et de multiplications, la représentation modulaire permet de réduire significativement les temps de calculs. On l'utilise alors comme arithmétique intermédiaire. Il faut alors être capable de convertir des données entre la représentation modulaire et une représentation de numération de position.

D'autre part, il peut être nécessaire de changer le système modulaire dans lequel sont représentées les données, par exemple pour exprimer certaines valeurs qui ne le pourraient pas comme les inverses modulaires. Il est d'autre part possible d'augmenter le nombre de moduli afin d'agrandir l'intervalle de représentation. Cette opération de changement ou d'extension de base nécessite, jusqu'à présent, une conversion.

Nous présentons dans cette section plusieurs méthodes de conversion entre systèmes modulaires et numération de position, entre systèmes modulaires ainsi que des algorithmes d'extension de base.

1.2.1 Conversion des systèmes de numération de position vers les systèmes modulaires

Le problème de ce type de conversion consiste à déterminer $x_i = X \bmod m_i$ à partir de m_i et de l'écriture en numération de position de X. De toutes les méthodes, il apparaît évident qu'effectuer la division euclidienne de X par m_i et d'en déduire le reste x_i n'est pas la moins coûteuse. Une approche plus élégante consiste à considérer l'expression binaire de X et à calculer x_i à partir de celle-ci. Cette idée a entre autre été évoquée dans [ZSY93], [Par93] et [AM93] :

$$x_i = |X|_{m_i} = \left| \sum_{j=1}^{n} \chi_j \times 2^j \right|_{m_i} = \left| \sum_{j=1}^{n} \chi_j \times \left|2^j\right|_{m_i} \right|_{m_i} \tag{1.2}$$

Les n chiffres χ_j constituent l'expression de X en binaire. Le calcul de x_i se résume à l'addition modulaire d'au plus n valeurs. Ce calcul peut être grandement facilité en tabulant les valeurs de $|2^j|_{m_i}$. En contre-partie, si la valeur à convertir est grande, cette addition est longue et coûteuse.

Afin de réduire le nombre de termes de l'addition 1.2, *B. Parhami* suggère dans [Par93] un algorithme de conversion en grande base. Le nombre à convertir X exprimé en base β est alors constitué de n' chiffres avec $n' < n$. La valeur de la grande base β est choisie de telle sorte que l'expression de X dans cette base puisse facilement se déduire de son écriture binaire. En pratique β est une puissance de deux. L'équation 1.2 se réécrit de la façon suivante :

$$x_i = |X|_{m_i} = \left| \sum_{j=1}^{n'} \chi_j \times \beta^j \right|_{m_i} = \left| \sum_{j=1}^{n'} \left|\chi_j \times \beta^j\right|_{m_i} \right|_{m_i} = \left| \sum_{j=1}^{n'} \chi_j \times \left|\beta^j\right|_{m_i} \right|_{m_i} \tag{1.3}$$

Les valeurs $|\chi_j \times \beta^j|_{m_i}$ sont tabulées, de sorte que le calcul de (1.3) se résume à l'addition de n' termes. Les tables impliquées dans ce calcul sont alors constituées pour chaque modulus m_i de $n' \times \beta$ éléments de $\lceil \log_\beta m_i \rceil$ chiffres. La taille totale T des tables utilisées pour la conversion du nombre X est donc :

$$T = n' \times \beta \sum_{i=1}^{n} \lceil \log_\beta m_i \rceil$$

On remarque que plus la base utilisée est grande, plus la taille des tables est importante. Pour trouver un compromis entre la taille des tables et le nombre d'additions *Parhami* introduit le facteur d'efficacité E d'une implémentation défini comme suit.

$$E = \frac{1 + \alpha}{Ma + \alpha Ad}$$

où Me est la taille de la mémoire utilisée, Ad le nombre d'additions et où le paramètre α représente le rapport entre le coût de l'implémentation d'un additionneur et le coût d'une table de lecture. La valeur de ce facteur dépend de la technologie choisie pour l'implémentation, et varie entre 10 et 1000. Par exemple, pour des tables de très petite taille, α pourra prendre de grandes valeurs.

Il apparaît alors au vu de l'étude de ce facteur d'efficacité que pour des bases légèrement supérieures à 2 l'efficacité croit significativement [Par93]. Toutefois l'utilisation de base plus importantes –de l'ordre de 16– ne se justifie que pour des technologies dont le coût des additions est très supérieur à celui des tables.

Il est possible d'implémenter la somme 1.3 de différentes façons. La première consiste à pipeliner les additions modulo m_i [Par93]. La figure 1.1 illustre une implémentation

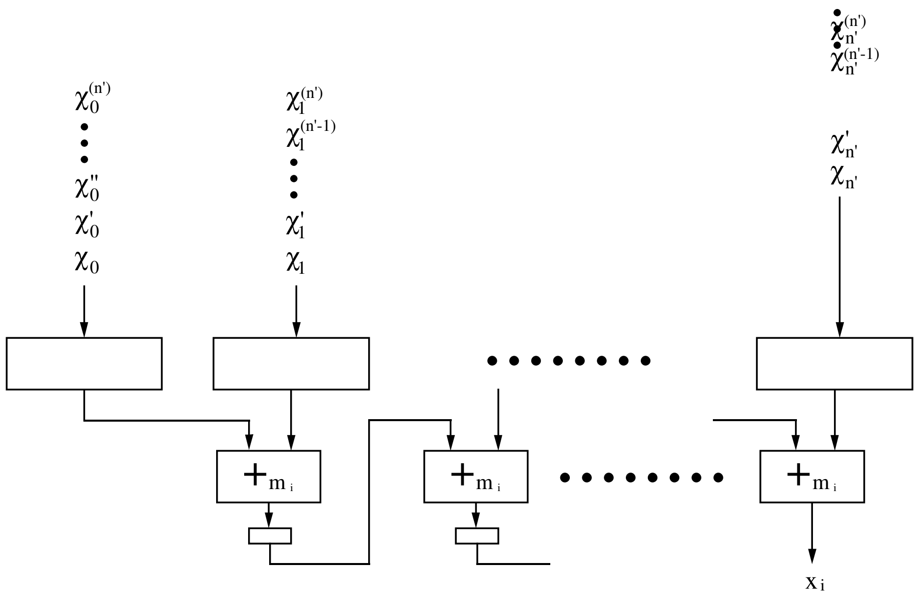


FIG. 1.1 – *Conversion pipelinée de la représentation en numération de position vers modulaire*

possible de cet algorithme. Les chiffres d'un nombre $X = \sum_{j=0}^{n'} \chi_j \times \beta^j$ à convertir sont décalés dans le temps de sorte que le calcul de x_i soit déterminé en n' étapes. Les chiffres χ_j servent d'entrée aux tables stockant $|\chi_j \times \beta^j|_{m_i}$. Il est alors possible de pipeliner ce calcul pour plusieurs nombres $X', X", \cdots, X^{n'}, \cdots$. Le pipeline est plein au bout de n' étapes. Un reste est ensuite obtenu lors de chaque cycle.

Cette implémentation est adaptée aux applications nécessitant beaucoup de conversions comme dans le traitement du signal où l'arithmétique modulaire n'est utilisée que pour accélérer certains calculs. D'après [Par93], l'implémentation de cette méthode nécessite pour chaque modulus n' tables et n' additionneurs modulaires (Tab. 1.1).

Une seconde implémentation [Par93], adaptée aux conversions plus occasionnelles, permet d'effectuer la somme 1.3 en temps logarithmique à l'aide de n' additionneurs modulaires disposés en arbre binaire (Fig. 1.2). Elle nécessite la même quantité de matériel que l'implémentation précédente. Les chiffres du nombre à convertir sont présentés en parallèle. Tout comme dans l'architecture précédente, ils servent d'entrée aux tables de lectures stockant $|\chi_j \times \beta^j|_{m_i}$. Le résultat est ainsi obtenu après $\lceil \log_2 n' \rceil$ cycles.

Enfin l'implémentation présentée dans la figure 1.3.a est plus économique en place et se prête plutôt aux conversions très occasionnelles [Par93]. Les chiffres sont présentés en série au convertisseur. La somme 1.3 est calculée par accumulations successives en n' étapes avec un seul additionneur modulaire. Les chiffres χ_j ainsi que leur indice j servent d'entrée à une table unique stockant les valeurs de $|\chi_j \times \beta^j|_{m_i}$.

Une autre implémentation réduisant la taille des tables utilisées est présentée dans la figure 1.3.b. Dans ce cas on ne tabule plus les valeurs $|\chi_j \times \beta^j|_{m_i}$ mais les puissances de la base $|\beta^i|_{m_i}$ et on effectue une multiplication modulaire supplémentaire (notée $*_{m_i}$ dans la figure 1.3.b). L'indice j est suffisant pour accéder aux valeurs de $|\beta^j|_{m_i}$ stockées dans

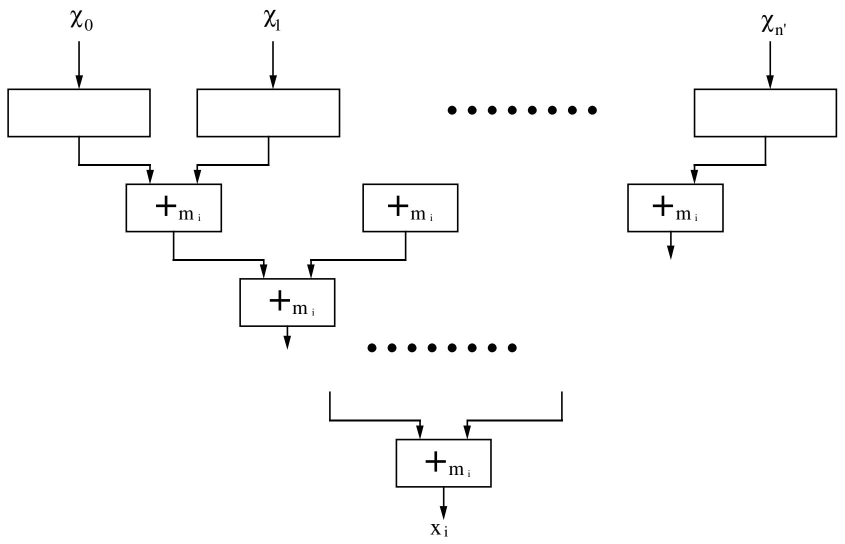


FIG. 1.2 – *Conversion logarithmique de la représentation en numération de position vers modulaire*

la table.

Le choix de cette implémentation peut être avantageux si le coût de l'implémentation du multiplieur modulaire est faible. La taille de la table est réduite de $n' \times \beta \times \lceil \log_\beta m_i \rceil$ à $n' \times \lceil \log_\beta m_i \rceil$ pour chaque modulus.

La table 1.1 résume les caractéristiques des différentes architectures présentées ci-dessus. Nous y présentons la taille des tables, le nombre d'additionneurs modulaires, de multiplieurs modulaires nécessaires pour la conversion d'un nombre vers un système modulaire. La complexité en temps est celle du calcul d'un reste.

	Pipeline	Logarithmique	Simple (a)	Simple (b)
Add. mod.	$n'-1$	n'	1	1
Mult. mod.	0	0	0	1
Table	$n'\beta \sum_{i=1}^{n} \lceil \log_\beta m_i \rceil$	$n'\beta \sum_{i=1}^{n} \lceil \log_\beta m_i \rceil$	$n'\beta \sum_{i=1}^{n} \lceil \log_\beta m_i \rceil$	$n' \sum_{i=1}^{n} \lceil \log_\beta m_i \rceil$
Temps	$\mathcal{O}(n')$ puis $\mathcal{O}(1)$	$\mathcal{O}(\lceil \log_2 n' \rceil)$	$\mathcal{O}(n')$	$\mathcal{O}(n')$

TAB. 1.1 – *Caractéristiques de méthodes de conversion de numération de position en arithmétique modulaire*

Les architectures présentées ci-dessus sont indépendantes du choix des moduli. Les opérations modulaires peuvent être simplifiées pour des moduli du type $2^k, 2^{k+1}$ et $2^k - 1$ (voir section 1.3).

Le choix d'une architecture pour la conversion dépend de l'application à laquelle elle est destinée. Dans le cadre d'une architecture dédiée au traitement du signal où le débit de la conversion doit être élevé, la conversion pipelinée est mieux adaptée. La dernière solution est plus séduisante pour l'implémentation d'un opérateur RNS plus généraliste.

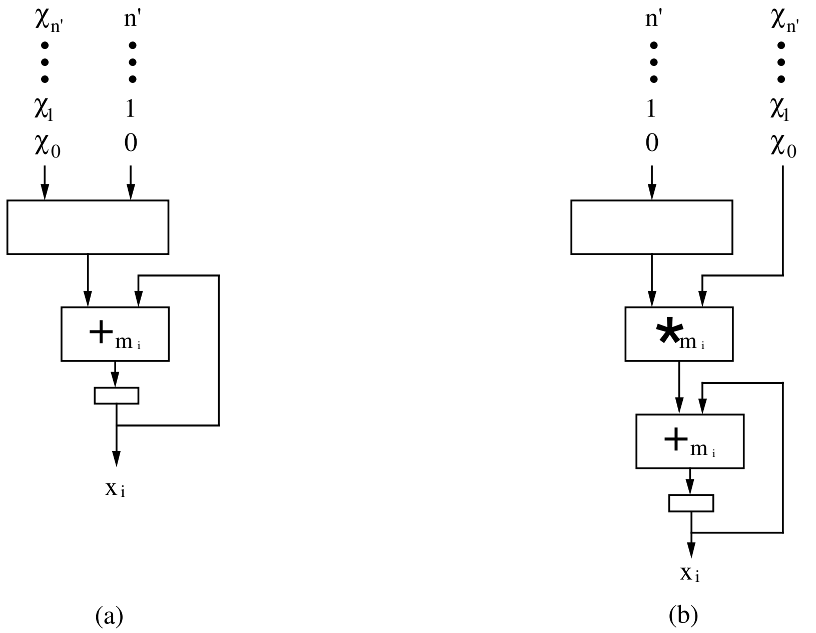


FIG. 1.3 – *Conversions simples de la représentation en numération de position vers une représentation modulaire*

1.2.2 Conversion des systèmes modulaires vers les systèmes de numération de position

Le théorème des restes chinois donne dans sa démonstration un moyen de calculer un nombre dans pratiquement tous systèmes de numération de position à partir de sa représentation modulaire. Nous présentons à la sous-section 1.2.2.1 les méthodes de conversion basées sur ce principe avec leur implémentation. Elles visent à réduire le coût de calcul de la somme

$$X = \sum_{i=1}^{n} \left(x_i (M_i)_i^{-1} M_i \right) \bmod M$$

La seconde partie de cette sous-section présente une méthode de conversion vers les systèmes "mixed radix" [2] ou écritures en bases de Cantor. Cette représentation de position, peu adaptée aux calculs, a l'avantage d'être naturellement associée aux systèmes modulaires et d'être déterminée à l'aide de petites valeurs. Elle permet d'effectuer simplement des comparaisons, ce que ne permettent pas les systèmes modulaires.

1.2.2.1 Conversion simple

C'est une des méthodes de conversion des systèmes modulaires les plus utilisées. Son avantage est de pouvoir être effectuée en temps logarithmique en parallèle. Soit $X_{RNS} = (x_1, x_2, \dots, x_n)$ la représentation modulaire de l'entier $0 \leq X < M$ dans la base B_m^n. Avec les notation définies plus haut on note :

$$\alpha_i = x_i \left(M_i\right)_{m_i}^{-1} \bmod m_i$$

2. Mixed Radix System ou MRS

En remarquant que :

$$(\alpha_i M_i) \bmod m_j = x_i \text{ si } j = i$$

$$(\alpha_i M_i) \bmod m_j = 0 \text{ sinon}$$

l'équation (1.4) définit parfaitement l'entier positif inférieur à M dont la représentation modulaire est X_{RNS}.

$$X = (\sum_{i=1}^{n} \alpha_i M_i) \bmod M \tag{1.4}$$

Exemple :

Soit la base RNS B_m^3=(17, 27, 31), on convertit dans cet exemple l'entier :

$$X_{RNS} = (2, 2, 2)$$

Dans cette base on a :

$$M = 14229, M_1 = 837, M_2 = 527 \text{ et } M_3 = 459$$

$$(M_1)_{17}^{-1} = 13, (M_2)_{27}^{-1} = 2 \text{ et } (M_3)_{31}^{-1} = 5$$

On détermine :

$$\alpha_1 = 9, \alpha_2 = 4, \alpha_3 = 10$$

Alors

$$S = 9 \times 837 + 4 \times 527 + 10 \times 459 = 14231$$

$$X = S \bmod M = 2 \quad \blacksquare$$

Une implémentation adaptée du théorème des restes chinois décrit dans [Knu81] est illustrée par la figure 1.4. Cette conversion offre l'avantage de pouvoir être effectuée en parallèle à l'aide d'un arbre binaire d'additionneurs. Les valeurs $|\alpha_i M_i|_M$ peuvent être tabulées dans des tables de m_i valeurs de $\lceil \log_\beta |\alpha_i M_i|_M \rceil$ pour chaque modulus. Elles sont obtenues à partir des valeurs x_i. La table 1.2 regroupe les caractéristiques de l'implémentation de ce type d'algorithme.

Bien que l'architecture ci-dessus ne soit pas optimisée, elle montre les points faibles de cette conversion : les nombres additionnés dans (1.4) sont de très grands nombres de l'ordre de grandeur de M. Si M est très grand, ces additions modulaires sont coûteuses. Nous verrons par la suite différentes méthodes visant à réduire le temps de calcul de cette addition.

Nombre d'additionneurs	n
Taille additionneurs	$\lceil \log_\beta M \rceil$
Taille tables	$\sum_{i=1}^{n} m_i \lceil \log_\beta \alpha_i M_i \rceil$
Complexité en temps	$\mathcal{O}(\log M \times \log n)$

TAB. 1.2 – *Caractéristiques de la conversion d'un nombre en arithmétique modulaire vers une arithmétique de position*

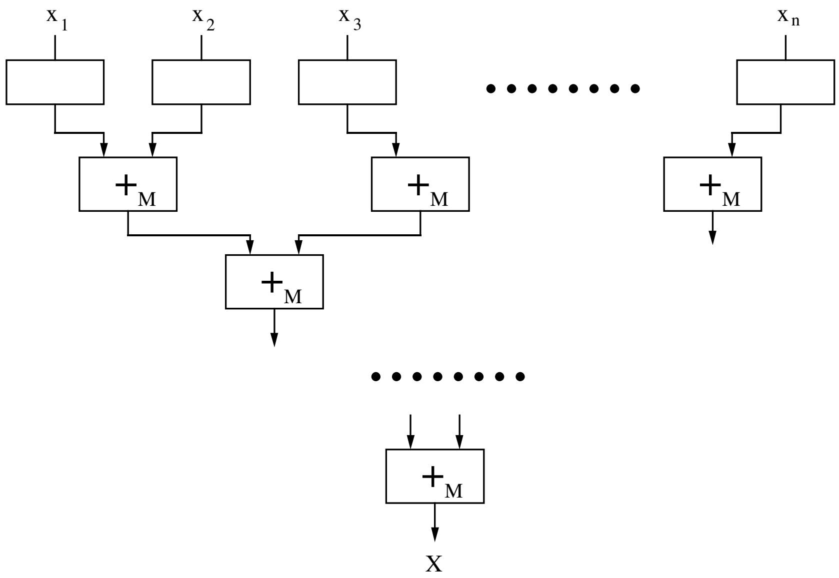


FIG. 1.4 – *Conversion représentation modulaire vers numération de position*

Additions rapides Une implémentation rapide consiste à générer les sommes partielles $\left|\left|x_i\,(M_i)^{-1}_{m_i}\right|_{m_i}\right|_M$ en parallèle puis à les additionner à l'aide d'additionneurs rapides. Cette somme peut être effectuée avec des additionneurs "Carry-Save" disposés en arbre binaire [ZSY93] [EB92]. Le calcul est effectué en temps logarithmique à l'aide de n additionneurs, chacun fonctionnant en temps constant. Une autre approche [Par93] consiste à générer le résultat dans une grande base. De cette façon la somme 1.4 est effectuée sur des nombres ayant moins de chiffres.

Regroupement Une autre approche de cette conversion consiste à réduire le nombre d'arguments de la somme 1.4. Parmi les méthodes de conversion à base de tables décrites par *Parhami* [Par93] on peut choisir de regrouper les moduli m_i en pseudo-moduli $\tilde{m}_i$ plus grands. Ce regroupement doit être effectué de sorte que les $\tilde{m}_i$ aient à peu près le même nombre de chiffres. L'addition 1.4 est alors constituée d'un nombre réduit d'opérandes.

La contrepartie de cette approche est l'augmentation de la taille des tables qu'il devient très vite impossible d'implémenter. Considérons les τ partitions s_j de la base modulaire, de sorte que $s_i \cap s_j = \emptyset$ pour $i \neq j$. Alors la taille T des tables est :

$$T = \sum_{k=1}^{\tau} \prod_{m_i \in s_k} m_i$$

Cette approche convient à l'utilisation de petits systèmes modulaires à cause du coût d'implémentation des grandes tables..

Conversion approchée Plutôt que de chercher à effectuer les additions modulo M le plus rapidement possible, dans [HP94a], [HP95] et [BDM96] les auteurs proposent d'effectuer ces sommes avec des termes approchés plutôt qu'exacts. Les nombres manipulés étant moins longs, la conversion est plus rapide. La conversion n'est plus exacte, mais approchée. Cette méthode permet de comparer deux nombres dans la plupart des cas et se prête bien à différents algorithmes de division [HP94a] [BDM96].

L'article [HP95] donne une analyse de l'erreur générée au cours de ce calcul. Soit $f(X)$ défini comme suit :

$$f(X) = \left[\sum_{i=1}^{n} \frac{\left| x_i \left(M_i \right)_{m_i}^{-1} \right|_{m_i}}{m_i} \right] = \frac{X}{M} \tag{1.5}$$

Une approximation de cette équation peut être faite en approchant ses termes par troncature ou par arrondi au plus près. On note respectivement $f^{T,d}(X)$ et $f^{R,d}(X)$ les valeurs approchées par troncature et arrondi après le $d^{ème}$ chiffre de $f(X)$.

$$f^{T,d}(X) = \left[\sum_{i=1}^{n} \frac{\left[x_i \left(M_i \right)_{m_i}^{-1} \right]_d}{m_i} \right] \tag{1.6}$$

$$f^{R,d}(X) = \left[\sum_{i=1}^{n} \frac{\left[x_i \left(M_i \right)_{m_i}^{-1} \right]_{\tilde{d}}}{m_i} \right] \tag{1.7}$$

Hung et *Parhami* s'intéressent aux erreurs de conversion des deux méthodes lors de la conversion en écriture binaire. On les note :

$$\Delta f^{T,d}(X) = f^{T,d}(X) - f(X)$$

et

$$\Delta f^{R,d}(X) = f^{R,d}(X) - f(X)$$

Ils établissent que :

$$\left\{ \Delta f^{T,d}(X) \mid 0 \leq X \leq M-1 \right\} = \left\{ -2^{-d} \sum_{i=1}^{n} \frac{y_i}{m_i} | 0 \leq y_i \leq m_i - 1, 1 \leq i \leq n \right\}$$

$$\left\{ \Delta f^{R,d}(X) \mid 0 \leq X \leq M-1 \right\} = \left\{ -2^{-d} \sum_{i=1}^{n} \frac{y_i}{m_i} | -\frac{m_i - 1}{2} \leq y_i \leq \frac{m_i - 1}{2}, 1 \leq i \leq n \right\}$$

Cette méthode de conversion est appropriée aux applications ne nécessitant que l'ordre de grandeur des nombres. Une application de cette conversion à la division en représentation modulaire est donnée dans le chapitre 2.

Moduli remarquables La dernière approche de la conversion consiste à trouver des moduli pour lesquels les calculs modulaires sont plus faciles. Dans [SH88] une méthode efficace est présentée pour l'ensemble de moduli $\mathcal{B}_m^3 = \{2^k+1, 2^k, 2^k-1\}$. Avec ce système modulaire l'expression des M_i est très simple ainsi que celle de leurs inverses modulaires $(M_i)_{m_i}^{-1}$.

$$M_1 = 2^k(2^k - 1) \quad (M_1)_{m_1}^{-1} = 2^{k-1} + 1$$

$$M_2 = 2^{2k} + 1 \quad (M_2)_{m_2}^{-1} = 2^k - 1$$

$$M_3 = 2^k(2^k+1) \qquad (M_3)^{-1}_{m_3} = 2^{k-1}$$

Le calcul de X peut alors se ramener à celui de $\lceil X/2^k \rceil$ sachant que :

$$X = \left\lceil \frac{X}{2^k} \right\rceil 2^k + x_2$$

où $x_2 = X \bmod m_2 = x \bmod 2^k$. Le calcul de $\lceil X/2^k \rceil$ se ramène alors à :

$$\left\lceil \frac{X}{2^k} \right\rceil = |(C - x_1) + (B + A)|_{2^{2k}-1} \tag{1.8}$$

avec

$$A = \left|(2^{2k-1} + 2^{k-1})x_3\right|_{2^{2k}-1}$$

$$B = \left|(2^{2k} - 2^k - 1)x_2\right|_{2^{2k}-1}$$

$$C = \left|(2^{2k-1} + 2^{k-1})x_1\right|_{2^{2k}-1}$$

Le calcul de l'équation (1.8), de A, B et C se résume à des opérations simples sur des vecteurs de bits. *Sweidan Andraos* et *Hiasat Ahmad* [SH88] estiment que l'implémentation de leur conversion nécessite quatre additionneurs de $8k$ bits.

Une amélioration de cette méthode a récemment été proposée dans [WJM97] et simplifie le calcul de $|C - x_1 + B|_{2^{2k}-1}$.

Cette conversion fait appel à des opérations arithmétiques et logiques simples. Elle est rapide et peut facilement être implémentée en VLSI [SH88]. D'autre part l'efficacité de l'implémentation des opérations effectuées dans le système modulaire $\{2^n-1, 2^n, 2^n+1\}$ rend ce type de système modulaire populaire dans les applications de traitement du signal [SJJT86]. Toutefois on peut critiquer l'efficacité de ce type de système modulaire pour des calculs sur de très grands entiers. En effet un système comportant un plus grand nombre de moduli semble mieux adapté.

1.2.2.2 Conversion vers MRS

Nous avons vu que les méthodes de conversion basées sur le théorème des restes chinois pêchent par la taille des données utilisées. Une autre approche permettant la manipulation de petits nombres a été proposée par H.L. Garner en 1959 dans [Gar59]. Cette méthode permet de reconstruire l'entier X dans le système de numération de position appelée "Mixed Radix System" (MRS) ou bases de Cantor. Nous présentons en premier lieu dans cette sous-section la représentation "MRS". Ensuite nous abordons deux algorithmes de conversion proposés dans [ST67] et [Hua83].

Pour représenter un entier X dans la notation MRS, on se donne un ensemble de n entiers μ_i constituant la base de la notation. Les chiffres x'_i de la représentation "Mixed Radix" de X vérifient $x'_i < \mu_i$. L'écriture d'un nombre X dans ce système de numération est la suivante :

$$X_{MRS} = x'_0 + x'_1\mu_0 + x'_2\mu_0\mu_1 + \cdots + x'_{n-1}\prod_{i=0}^{n-2}\mu_i$$

On peut remarquer que selon l'ordre des éléments de la base l'écriture MRS est différente. Ainsi pour une base donnée de n éléments on peut associer $n!$ notation MRS différentes selon l'ordre dans lequel sont pris les éléments de la base.

Exemple

Soit l'entier $X = 831$. Il admet les notations suivantes :

$$\begin{array}{rcll} & & & \text{Base MRS} \\ X & = & \mathbf{0} + \mathbf{4} \times 3 + \mathbf{0} \times 3 \times 7 + \mathbf{3} \times 3 \times 7 \times 13 & \{3, 7, 13, 19\} \\ X & = & \mathbf{5} + \mathbf{1} \times 7 + \mathbf{9} \times 7 \times 13 + \mathbf{0} \times 7 \times 13 \times 19 & \{7, 13, 19, 3\} \\ X & = & \mathbf{12} + \mathbf{6} \times 13 + \mathbf{3} \times 13 \times 19 + \mathbf{0} \times 13 \times 19 \times 3 & \{13, 19, 3, 7\} \\ X & = & \mathbf{14} + \mathbf{1} \times 19 + \mathbf{0} \times 19 \times 3 + \mathbf{2} \times 19 \times 3 \times 7 & \{19, 7, 3, 13\} \end{array} \blacksquare$$

On peut remarquer que :

$$x'_i < \mu_i \text{ et } x'_0 + \cdots + x'_{k_1}\mu_0 \cdots \mu_{k-1} < \mu_1\mu_2 \cdots \mu_k$$

Cette notation permet de comparer les nombres par simple comparaison des chiffres de poids le plus fort vers le poids le plus faible.

La représentation MRS est par ailleurs naturellement associée à la représentation modulaire. En effet en considérant que les moduli m_i constituent la base d'une notation MRS il est possible d'obtenir les chiffres x'_i d'une représentation MRS de X de la manière suivante [ST67]. On note $m_{i,j}^{-1}$ est l'inverse de m_i modulo m_j de sorte que :

$$m_i . m_{i,j}^{-1} \equiv 1[m_j]$$

Alors

$$\begin{array}{l} x'_1 = x_1 \bmod m_1 \\ x'_2 = (x_2 - x'_1)m_{1,2}^{-1} \bmod m_2 \\ x'_3 = ((x_3 - x'_1)m_{1,3}^{-1} - x'_2)m_{2,3}^{-1} \bmod m_3 \\ x'_4 = (((x_4 - x'_1)m_{1,4}^{-1} - x'_2)m_{2,4}^{-1}) - x'_3)m_{3,4}^{-1} \bmod m_3 \\ \vdots \\ x'_n = (\cdots(x_n - x'_1)m_{1,n}^{-1} - x'_2)m_{2,n}^{-1}) - \cdots - x'_{n-1})m_{n-1,n}^{-1} \bmod m_n \end{array} \quad (1.9)$$

Et finalement :

$$X = x'_1 + x'_2 m_1 + x'_3 m_1 m_2 + \cdots + x'_n m_1 \cdots m_{n-1} \quad (1.10)$$

Les n nombres $x'_1, x'_2, \ldots, x'_n$ constituent les chiffres de la représentation MRS de X dans la base MRS :

$$(1, m_1, m_1 m_2, \ldots, m_1 m_2 \ldots m_{n-1})$$

Exemple

Soit la base RNS $\mathcal{B}_m^3 = \{17, 27, 31\}$, on convertit :

$$X_{RNS} = (2, 2, 2)$$

$$M = 14229, m_{1,2}^{-1} = 4, m_{1,3}^{-1} = 11 \text{ et } m_{2,3}^{-1} = 23$$

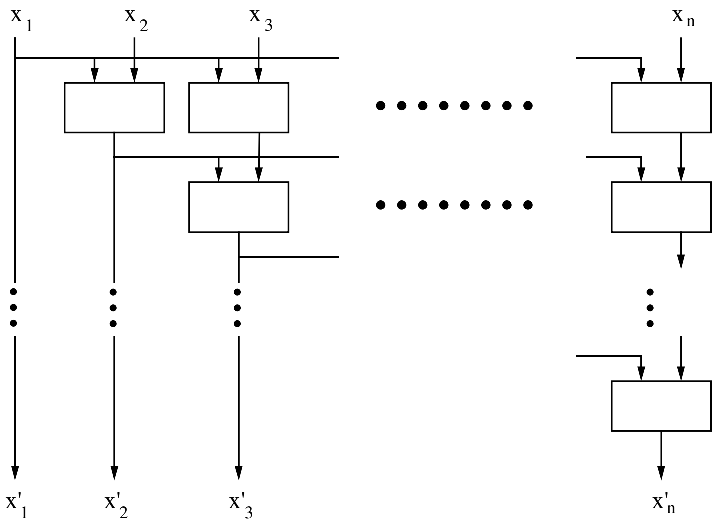


FIG. 1.5 – *Conversion "Mixed Radix" de Szabo et Tanaka*

	Additions et Multiplications	Tables
Taille table	$p \times \dfrac{n(n-1)}{2}$	$p \times \dfrac{n(n-1)}{2} \sum_{i=1}^{n} \sum_{j=i+1}^{n} m_i m_j$
Nombre d'additionneurs	$\dfrac{n(n-1)}{2}$	0
Nombre de multiplieurs	$\dfrac{n(n-1)}{2}$	0
Taille des opérandes	p	p

TAB. 1.3 – *Caractéristiques de la conversion "Mixed Radix" de Szabo et Takana*

Alors

$$a_1 = 2, a_2 = 0, a_3 = 0$$

Par conséquent

$$X = 2 \quad \blacksquare$$

Nous déduisons du système d'équations 1.9 l'architecture illustrée par la figure 1.5. Cette implémentation de la conversion peut être effectuée à l'aide de $n(n-1)/2$ opérateurs. Chacun d'entre eux est constitué d'un soustracteur modulaire, d'un multiplieur modulaire et d'une table stockant une constante $m_{i,j}^{-1}$ sur p chiffres. Le temps nécessaire à la conversion est le temps qu'il faut pour déterminer le chiffre x'_n. C'est à dire le temps nécessaire pour traverser n opérateurs. En considérant que la complexité en temps de chaque opérateur est constante, cette conversion s'effectue avec une complexité en temps $\mathcal{O}(n)$. Ce calcul peut toutefois être pipeliné en n étapes. Dans ce cas les chiffres sont obtenus les un à la suite des autres, un à chaque cycle.

Pour de petits moduli les $n(n-1)/2$ opérateurs peuvent se réduire à de simples tables. Chacune d'entre elles est constituée de $m_i \times m_j$ éléments de p chiffres. La table 1.3 résume les caractéristiques des implémentations de cet algorithme proposées ci-dessus.

Une autre méthode de conversion "MRS" utilisant la formule du théorème des restes chinois en arithmétique "MRS" a été proposée dans [Hua83]. Les opérandes de la somme $\sum_{i=1}^{n}(x_i(M_i)_i^{-1}M_i \bmod M$ sont converties en MRS. Les additions modulo M sont effectuées en arithmétique "MRS" en ne tenant pas compte de la retenue sortante. D'autre part les additions sont simplifiées en remarquant que la représentation modulaire de $\left|M_i \left|x_i(M_i)_i^{-1}\right|_{m_i}\right|_M$ est :

$$\left|M_i \left|x_i(M_i)_i^{-1}\right|_{m_i}\right|_M = \{0, \cdots, 0, x_i, 0, \cdots, 0\}$$

Sa représentation "Mixed Radix" s'écrit alors :

$$\left|M_i \left|x_i(M_i)_i^{-1}\right|_{m_i}\right|_M = \{0, \cdots, 0, \alpha_{ii}, \alpha_{ii+1}, \cdots, \alpha_{in}\}$$

où α_{in} est le chiffre de poids le plus fort.

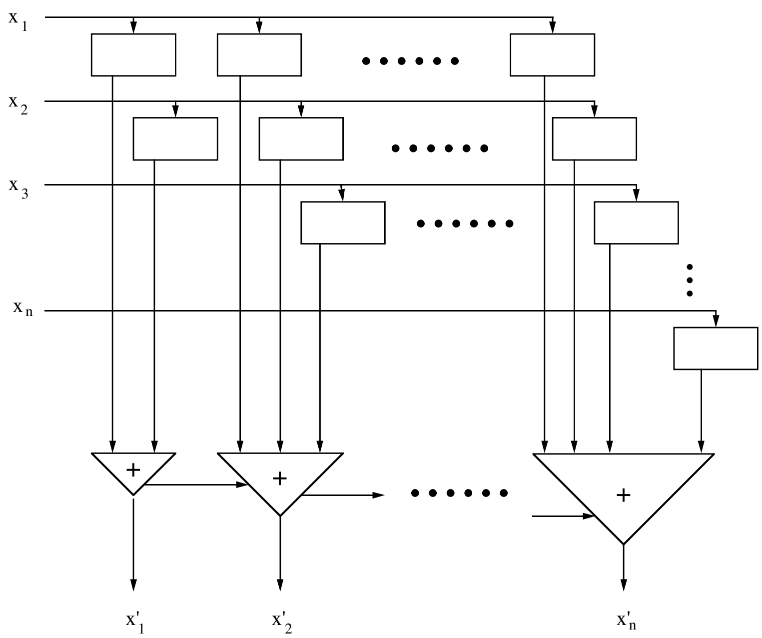


FIG. 1.6 – *Conversion "Mixed Radix" basée sur le théorème des restes chinois*

La figure 1.6 donne une implémentation possible de cet algorithme. Les chiffres de la représentation "Mixed Radix" des $\left|M_i \left|x_i(M_i)_i^{-1}\right|_{m_i}\right|_M$ sont obtenus grâce à $n(n-1)$ tables ayant chacune m_i éléments de p chiffres. Les additions sont effectuées par des arbres d'additionneurs en temps logarithmique de sorte que la conversion est déterminée en temps logarithmique. En résumé cette conversion peut être effectuée avec une complexité en temps de $\mathcal{O}(\log n)$ et une complexité en espace de $\mathcal{O}(n^2)$.

La méthode de conversion de *Hitz* et *Kaltofen* dans [HK95] est très similaire à celle présentée ci-dessus. Leur amélioration consiste en une meilleure gestion de la retenue dans l'addition finale. Toutefois cette conversion reste coûteuse en place.

1.2.3 Extension et changement de base

Il est parfois utile de connaître les restes d'un nombre RNS dans une autre base que celle dans laquelle il est exprimé ou modulo un modulus supplémentaire. L'opération qui consiste à déterminer les restes $|X|_{p_1}, |X|_{p_2}, \ldots, |X|_{p_t}$, d'un nombre X à partir de ses restes $|X|_{m_1}, |X|_{m_2}, \ldots, |X|_{m_n}$ est le changement de base ou l'extension de base. Les bases des deux systèmes modulaires sont les suivantes :

$$\begin{array}{ll} \mathcal{B}_m^r = \{m_1, m_2, \ldots, m_r\} & \mathcal{B}_p^t = \{p_1, p_2, \ldots, p_t\} \\ M = \prod_{i=1}^{r} m_i & P = \prod_{i=1}^{t} p_i \end{array}$$

Le changement de base ou extension de base est une opération qui est utilisée notamment lors de l'extension dynamique de l'intervalle de représentation [Gam89][Gam91], la comparaison et la détection du dépassement de capacité [HK95] ou la détermination de signe [BEPP97]. Nous présentons dans cette sous-section deux approches de ce problème. La première fait appel à la conversion en "Mixed Radix". La seconde, plus efficace, se dérive du théorème des restes chinois.

1.2.3.1 Extension de base de Szabo et Tanaka

Cette extension de base a été présentée par *Szabo* et *Tanaka* dans [ST67]. La conversion d'un nombre modulaire X de $\mathcal{B}_m^n$ vers $\mathcal{B}_p^t$ se divise en deux étapes. La première consiste à effectuer la conversion de X en MRS. Ensuite elle détermine les restes de ce nombre dans le système modulaire ayant pour base $\mathcal{B}_p^t$. Soient $x'_1, x'_2, \ldots, x'_n$ les chiffres de l'expression MRS de X. Les restes $|X|_{p_i}$ s'expriment alors :

$$|X|_{p_i} = \left| x'_1 + x'_2 m_1 + x'_3 m_1 m_2 + \ldots + x'_n \prod_{j=1}^{n-1} m_j \right|_{p_i}$$

$$|X|_{p_i} = \left| |x'_1|_{p_i} + |x'_2|_{p_i} |m_1|_{p_i} + |x'_3|_{p_i} |m_1 m_2|_{p_i} + \ldots + |x'_n|_{p_i} \left| \prod_{j=1}^{n-1} m_j \right|_{p_i} \right|_{p_i} \qquad (1.11)$$

Le calcul des chiffres "Mixed Radix" est effectué par l'un des algorithmes présentés de la sous-section 1.2.2.2. Les valeurs des facteurs $\left|\prod_{j=1}^{n-1} m_j\right|_{p_i}$ apparaissant dans l'équation (1.11) peuvent être tabulées. Dans ce cas, pour chaque modulus de la base B_p^t, il est nécessaire de tabuler $p \times (n-1)$ valeurs de $\lceil \log_\beta m_i \rceil$ chiffres. Le calcul des restes $|X|_{p_i}$ peut être effectué en parallèle. Les chiffres x'_j sont alors diffusés vers tous les opérateurs. Le temps et la quantité de matériel nécessaires à cette extension de base dépendent des choix effectués pour le calcul de la représentation MRS et de la conversion en RNS.

1.2.3.2 Extension de base de Shenoy et Kumaresan

Une autre méthode dérivée du théorème des restes Chinois a été proposée par *Shenoy* et *Kumaresan* [SK89]. D'après la formule du théorème des restes chinois la valeur d'un entier X est obtenue à partir de ses restes dans la base modulaire $\mathcal{B}_m^n$ de la façon suivante :

$$X = \left| \sum_{i=1}^{n} M_i \left| \frac{x_i}{M_i} \right|_{m_i} \right|_M$$

Ce qui revient à écrire qu'il existe un entier α tel que :

$$X = \sum_{i=1}^{n} M_i \left| \frac{x_i}{M_i} \right|_{m_i} - \alpha M \tag{1.12}$$

Lors du calcul des nouveaux restes dans la base B_p^t, le reste relatif au modulus p_j s'écrit alors :

$$|X|_{p_j} = \left| \sum_{i=1}^{n} \left| M_i \left| \frac{x_i}{M_i} \right|_{m_i} \right|_{p_j} - |\alpha M|_{p_j} \right|_{p_j} \tag{1.13}$$

La détermination de $|X|_{p_j}$ dépend de notre capacité à calculer α qui est la seule valeur inconnue dans l'équation 1.13. *Shenoy* et *Kumaresan* suggèrent d'utiliser un modulus redondant m_{n+1} :

$$\alpha = \left| (M)^{-1}_{m_{n+1}} \sum_{i=1}^{n} \left| M_i \left| \frac{x_i}{M_i} \right|_{m_i} \right|_{m_{n+1}} - |X|_{m_{n+1}} \right|_{m_{n+1}} \tag{1.14}$$

Ainsi α peut être calculé à partir de valeurs connues comme les restes de X dans la base B_m^n et l'inverse modulaire de M modulo m_{n+1}. On note ϕ les termes de la somme 1.14 :

$$\phi = \left| (M)^{-1}_{m_{n+1}} \times \left| M_i \left| \frac{x_i}{M_i} \right|_{m_i} \right|_{m_{n+1}} \right|_{m_{n+1}}$$

Les valeurs de ϕ sont stockées dans n tables constituées de m_i éléments de $\lceil \log_\beta m_{n+1} \rceil$ chiffres. La taille totale de ces tables est de $\lceil \log_\beta m_{n+1} \rceil \times \sum_{i=1}^{n} m_i$. Il faut d'autre part remarquer que m_{n+1} est une très petite valeur de l'ordre de grandeur de n.

La somme 1.14 peut être effectuée en temps logarithmique à l'aide de $n+1$ additionneurs modulo m_{n+1}. Ceux-ci peuvent être très simplifiés avec un choix judicieux de ce modulus redondant. Si m_{n+1} est égal à une puissance de la base dans laquelle sont exprimés les moduli, alors les additionneurs modulaires sont des additionneurs traditionnels dont la retenue sortante est ignorée.

Calcul de α	Add. modulo m_{n+1}	$t+1$
	Tables	$\lceil \log_\beta m_{n+1} \rceil \sum_{i=1}^{n} m_i$
Calcul des restes	Add. modulaires	$(n+1)t$
	Tables	$\sum_{j=1}^{t} \lceil \log_\beta p_j \rceil \sum_{i=1}^{n} m_i$
	Complexité en temps	$\mathcal{O}(\log n)$

TAB. 1.4 – *Caractéristiques de l'extension de base de Shenoy et Kumaresan*

Le calcul de chaque reste $|X|_{p_j}$ peut être entrepris en parallèle. Chacun étant déterminé en temps logarithmique à l'aide d'un arbre de t additionneurs. Les valeurs $\left|M_i \left|\frac{x_i}{M_i}\right|_{m_i}\right|_{p_j}$ et $-|\alpha M|_{p_j}$ sont stockées dans des tables dont la taille totale est $\lceil \log_\beta p_j \rceil \sum_{i=1}^{n} m_i$. La table 1.4 résume les caractéristiques de l'implémentation de cet algorithme.

1.3 Opérations

Tout l'intérêt de la représentation modulaire réside dans la vitesse d'exécution de certaines opérations. En effet l'addition, la soustraction et la multiplication peuvent être calculées en parallèle en un cycle. Cette rapidité facilite la manipulation de grands nombres. Toutefois d'autres opérations telles que la comparaison et la division sont relativement plus difficiles.

Nous présentons dans cette section les opérations en représentation modulaire et les principales implémentations qui ont été proposées dans la littérature scientifique. En premier lieu nous présentons les opérations “simples” : l'addition, la soustraction et la multiplication ainsi que quelques-unes de leur implémentation. Nous abordons ensuite le cas plus complexe de la comparaison et les bases incomplètes de Gamberger. Nous terminerons par le cas particulier qu'est la division exacte. Le cas de la division générale est traité dans le chapitre 2.

1.3.1 Addition, Soustraction et Multiplication

Avec un tel système de représentation les additions, les soustractions et les multiplications peuvent être calculées en parallèle en un cycle. En effet chacune de ces opérations peut être effectuée indépendamment sur chacun des restes [Knu81]. Considérons deux nombres X_{RNS} et Y_{RNS} où :

$$X_{RNS} = (x_1, x_2, \ldots, x_{n-1}, x_n)_{RNS}$$

$$Y_{RNS} = (y_1, y_2, \ldots, y_{n-1}, y_n)_{RNS}$$

Alors :

$$\begin{array}{l}(x_1, \ldots, x_n)_{RNS} + (y_1, \ldots, y_n)_{RNS} = ((x_1 + y_1) \bmod m_1, \ldots, (x_n + y_n) \bmod m_n)_{RNS} \\ (x_1, \ldots, x_n)_{RNS} - (y_1, \ldots, y_n)_{RNS} = ((x_1 - y_1) \bmod m_1, \ldots, (x_n - y_n) \bmod m_n)_{RNS} \\ (x_1, \ldots, x_n)_{RNS} \times (y_1, \ldots, y_n)_{RNS} = ((x_1 \times y_1) \bmod m_1, \ldots, (x_n \times y_n) \bmod m_n)_{RNS}\end{array}$$

<u>Exemple</u>

Soit le système modulaire dont la base $\mathcal{B}_m^4$ est $\{3, 7, 13, 19\}$. Soient les entiers $X = 147$ et $Y = 31$. Leur représentation modulaire dans ce système est :

$$\begin{array}{rl} X_{RNS} = & \{0, 0, 4, 14\} \\ Y_{RNS} = & \{1, 3, 5, 12\} \end{array}$$

L'addition, la soustraction et la multiplication de ces deux nombres peuvent être effectuées comme suit :

$$\begin{aligned} X_{RNS} +_{_{RNS}} Y_{RNS} &= \{(0+1) \bmod 3, (0+3) \bmod 7, (4+5) \bmod 13, (14+12) \bmod 19\} \\ &= \{1, 3, 9, 7\} \\ &= 178 \end{aligned}$$

$$\begin{aligned} X_{RNS} -_{_{RNS}} Y_{RNS} &= \{(0-1) \bmod 3, (0-3) \bmod 7, (4-5) \bmod 13, (14-12) \bmod 19\} \\ &= \{2, 4, 12, 2\} \\ &= 116 \end{aligned}$$

$$\begin{aligned} X_{RNS} \times_{_{RNS}} Y_{RNS} &= \{(0\times 1) \bmod 3, (0\times 3) \bmod 7, (4\times 5) \bmod 13, (14\times 12) \bmod 19\} \\ &= \{0, 0, 7, 16\} \\ &= 4557 \ \blacksquare \end{aligned}$$

Quelle que soit la taille des opérandes X_{RNS} et Y_{RNS} utilisées, les opérations se réduisent à des calculs sur des petits entiers. Etant donné que les restes sont indépendants les uns des autres, ces opérations sur de petits nombres peuvent être faites simultanément.

Toutefois, étant donné que les moduli composant un système modulaire doivent être premiers entre eux, il n'est pas possible de représenter des nombres ayant une infinité de chiffres avec des moduli ayant un nombre fini m_e de chiffres. Réciproquement un système modulaire capable de representer des nombres de k chiffre ne peut pas les manipuler en temps constant. En effet, il existe une relation entre le nombre de bits des moduli d'un système et celui du plus grand nombre représentable et le temps de calcul des opérations.

Soit $\Phi(x)$ la somme des logarithmes des nombre premiers p plus petits que x.

$$\Phi(x) = \sum_{\substack{p\leq x \\ p \ premier}} \log p = \log \prod_{\substack{p\leq x \\ p \ premier}} p$$

D'après *Hardy* et *Wright* [HM62] la valeur $\Phi(x)$ vérifie

$$\Phi(x) = \Theta(x)$$

Par conséquent

$$\prod_{\substack{p\leq x \\ p \ premier}} p = \Theta(e^x)$$

Nous pouvons en déduire, d'une part, que si l'on n'utilise que des moduli ayant k bits (c'est à dire inférieurs à 2^k), le système modulaire pourra permettre de représenter des nombres ayant au plus $\Theta(2^k)$ bits.

D'autre part si l'on souhaite définir un système modulaire capable de représenter des nombres inférieurs à $2^{k-1} \leq M < 2^k$ avec le plus de moduli possibles (parallélisation des opérations la plus importante possible), le plus grand modulus aura $\mathcal{O}(\log k)$ bits. Or manipuler des nombres de t bits en arithmétique usuelle pour effectuer des additions et des multiplications demande un temps de $\mathcal{O}(\log t)$. Par conséquent manipuler des nombres de inférieurs à M en représentation modulaire demande un temps de $\mathcal{O}(\log \log k)$

En conclusion la représentation modulaire permet le calcul des additions, soustractions et multiplications en temps $\mathcal{O}(\log \log k)$ et en espace linéaire en parallèle.

Remarque : Les restes d'un nombre ne permettent pas à l'heure actuelle de déterminer simplement son ordre de grandeur. La comparaison de deux nombres en représentation modulaire est encore difficile. D'autre part tous les calculs effectués en représentation modulaire sont déterminés modulo M. Il est donc difficile de détecter les dépassements de capacité. Lors de l'utilisation de cette représentation il est nécessaire de faire attention à ce type de problème.

1.3.2 Implémentations

La rapidité d'une opération en arithmétique modulaire dépend de la rapidité avec laquelle il sera possible d'effectuer cette opération pour chacun des restes. Le calcul de $|x_i+y_i|_{m_i}$, $|x_i-y_i|_{m_i}$ et $|x_i\times y_i|_{m_i}$ est simple si m_i est une puissance de la base dans laquelle sont exprimés x_i et y_i. Mais il en va tout autrement si m_i est un nombre quelconque.

Dans cette partie il sera question des différentes stratégies d'implémentation des additions et multiplications modulo des entiers de quelques bits à plusieurs dizaines de bits.

1.3.2.1 Algorithmes à base de tables

Le plus souvent des tables sont utilisées pour l'implémentation des opérations modulaires et plus particulièrement pour la multiplication modulaire [Jul78][SJJT86][BJM87]. Bien que très rapide cette méthode présente un inconvénient car la taille des tables de lecture croit exponentiellement avec la taille des moduli. Lorsque ceux-ci ont plus de 8 bits il est hors de question d'implémenter les opérations modulaires par des tables de lecture [BJM85] [Bay86] [BJM87].

Toutefois pour de petits moduli tels les systèmes modulaire utilisés dans les applications de traitement du signal [CPO95] [KJM86], les opérations modulaires à base de tables restent intéressantes [BJM83] [BJM87].

Multiplication L'approche la plus simple pour la multiplication modulaire par table est illustrée par la figure 1.7.b. Toutefois ces tables à deux entrés peuvent devenir très grandes [BJM85]. Une approche moins gourmande en place de la multiplication modulaire par lecture de table est de décomposer la multiplication de la façon suivante [SF77] :

$$z_i = |X \times Y|_{m_i} = |x_i \times y_i| = \left| \left| \frac{(x_i+y_i)^2}{4} \right|_{m_i} - \left| \frac{(x_i-y_i)^2}{4} \right|_{m_i} \right|_{m_i}$$

Les valeurs $\left|\frac{(.)^2}{4}\right|_{m_i}$ sont stockées dans des tables de lectures (Fig. 1.7.a) ce qui réduit de moitié la taille des entrées des tables de lecture.

Lorsque le modulus choisi m_i est un nombre premier on peut se ramener à une addition modulaire. En effet il est possible de définir le groupe de Galois $GF(m_i)$ dont tous les éléments peuvent être générés par une racine primitive notée g [Bay86] [RY91]. Ainsi la multiplication peut être effectuée en utilisant un isomorphisme entre le groupe multiplicatif $\{q_k\} = \{1, 2, \cdots, m_i - 1\}$ muni de la multiplication modulo m_i et le groupe additif $\{i_k\} = \{0, 1, \cdots, m_i - 2\}$ muni de l'addition modulo $(m_i - 1)$. La multiplication modulaire de deux nombres de $\{q_k\}$ s'écrit alors :

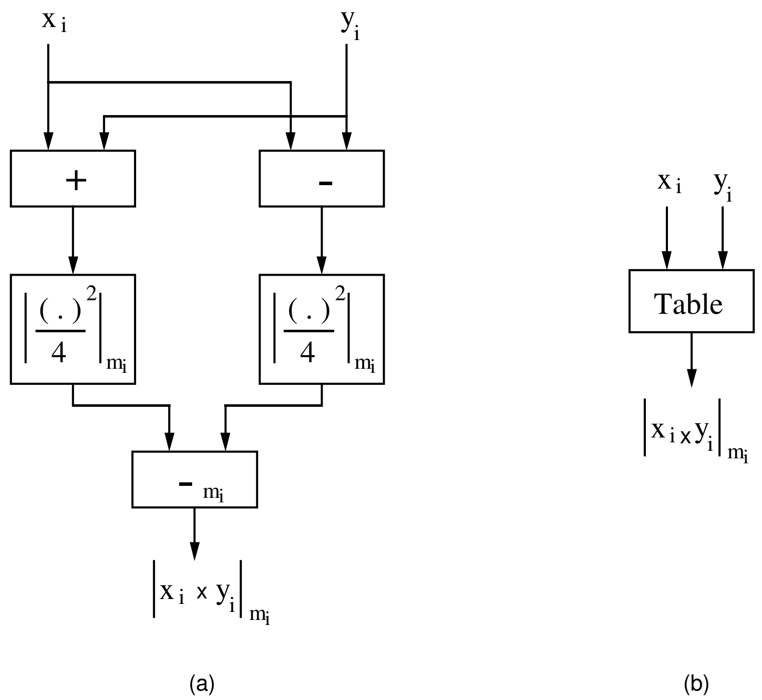


FIG. 1.7 – *Multiplication modulaire par table*

$$|q_j \times q_l|_{m_i} = g^{|i_j+i_l|_{m_i-1}}$$

Les valeurs i_j et i_l sont obtenues à partir de tables de lectures. La somme est effectuée dans un additionneur modulaire et l'exponentielle déterminée à l'aide d'une table (Fig. 1.8).

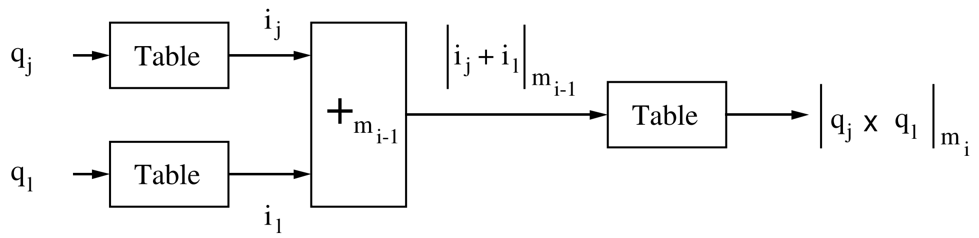


FIG. 1.8 – *Multiplication modulaire par index*

1.3.2.2 Algorithmes pour grands moduli

D'après le théorème des restes chinois, la largeur de l'intervalle de représentation d'un système modulaire est $M = \prod_{i=1}^{n} m_i$. Pour représenter de grands nombres en représentation modulaire nous avons le choix entre un plus grand nombre de moduli et des moduli moins nombreux mais plus gros. Or il y a peu de nombres premiers entre eux de moins de 8 bits. Ils ne sont pas suffisamment nombreux pour pouvoir représenter de grandes valeurs. Nous pouvons être amenés à utiliser de grands moduli pour lesquels l'utilisation de tables est exclue. Dans cette sous-section nous présentons des implémentations issues de la littérature scientifique.

Addition L'addition modulaire s'avère être simple pour certains moduli particulier [BJM85]. L'addition modulo 2^k est équivalente à une addition sur k bits dont le bit de poids supérieur est omis. Une telle addition est aussi simple pour tout modulus qui est une puissance de la base dans laquelle sont exprimés les restes et les moduli.

L'addition modulo $2^k - 1$ quant à elle se ramène à une addition sur k bits dont la retenue sortante est réinjectée au résultat [BJM85].

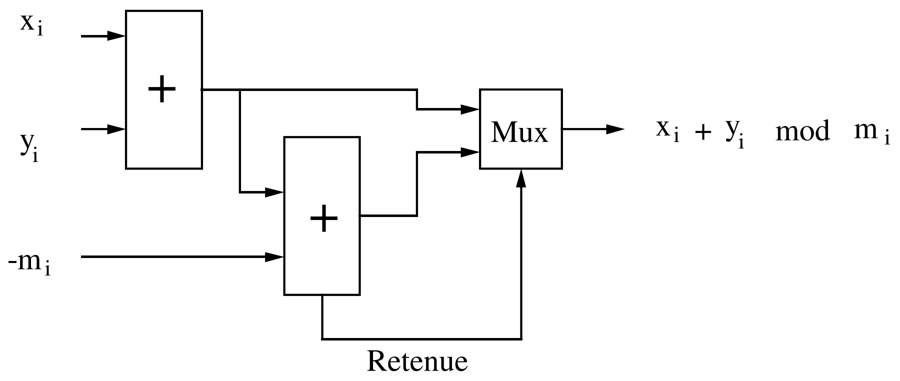


FIG. 1.9 – *Addition modulaire avec deux additionneurs*

Le cas plus général de l'addition peut être implémenté à l'aide de deux additionneurs (Fig. 1.9). On effectue tout d'abord la somme $z_i = x_i + y_i$. Si le résultat z_i est supérieur à m_i alors on soustrait m_i à z_i.

Une méthode permettant de faciliter les additions modulaires est donnée par *Zang*, *Shirozi* et *Yun* dans [ZSY93]. La comparaison entre m_i et z_i est facilitée par l'introduction d'un biais. La comparaison se résume à déterminer un dépassement de capacité. Ainsi l'addition de A et B modulo C exprimés en numération de position en base β peut être obtenue comme suit. On se donne un biais δ de telle sorte qu'il existe un entier k tel que :

$$C + \delta = \beta^k$$

Si le résultat $R = A + B$ est supérieur à ce biais β^k un dépassement de capacité a été produit. Dans ce cas le résultat est $R - \beta^k$ et $R - \delta$ dans le cas contraire. Cette méthode permet de réduire le coût de la comparaison au prix d'une addition supplémentaire.

Multiplication Les algorithmes de multiplication modulaire pour de grands moduli ont surtout été étudiés dans le cadre de la cryptographie où le modulus peut atteindre près de mille bits. Certains des algorithmes les plus courants sont présentés dans la section 3. Nous présentons ici des solutions plus adaptées à la multiplication en représentation modulaire.

Bayoumi propose d'implémenter un multiplieur modulaire à partir d'additionneurs modulaires où les moduli utilisés sont des moduli du type 2^k et $2^k - 1$ [Bay86]. Les produits partiels modulo 2^k et $2^k - 1$ sont additionnés à l'aide de ces additionneurs simplifiés.

Récemment *Wang*, *Jullien* et *Miller* ont proposé dans [WJM97] un algorithme de multiplication modulo $2^k - 1$ dont la structure est similaire au multiplieur de *Wallace* [Wal64].

1.3.3 Comparaison

La comparaison est un des points faibles de la représentation modulaire. En effet, ce n'est pas une représentation en numération simple de position. Il n'est à priori pas possible d'obtenir directement une information sur l'ordre de grandeur d'un nombre [Mul89]. Une solution coûteuse consiste à convertir les nombres à comparer dans un système de numération plus pratique avec un des algorithmes de la section 1.2 et d'y effectuer la comparaison. Une autre méthode de comparaison faisant intervenir une conversion peut être obtenue à l'aide d'un changement de base [HK95].

D'autres techniques de comparaison sont proposées dans [DIP93], [CL92] et [HP94a]. *Lu* et *Chiang* proposent une méthode basée sur la parité et dont l'efficacité est due à la présence de table dont la taille est proportionnelle à celle des nombres représentés. *Dimauro, Impedovo* et *Pirlo* utilisent une technique basée sur les fonction diagonales mais qui nécessite la manipulation de grands nombres. *Hung* et *Parhami* quant à eux proposent un algorithme permettant d'obtenir le signe d'un nombre de façon approchée.

1.3.3.1 Comparaison par changement de base

On se donne un système modulaire principal $\mathcal{B}_m^n$ et une base auxiliaire $\mathcal{B}_p^t$ tels que M et P soient premiers entre eux et $P > M$. Soient deux nombres différents à comparer $X < M$ et $Y < M$ exprimés dans ces deux systèmes. L'idée est de déterminer leur différence $Z = Y - X$ dans les deux systèmes modulaires. Après avoir effectué un changement de base de Z de $\mathcal{B}_m^n$ vers $\mathcal{B}_p^t$ on compare son écriture à celle calculée précédement. Si elles sont différentes, alors un dépassement de capacité s'est produit et Z est négatif. On peut alors conclure que $X > Y$. Dans le cas contraire, vu que $X \neq Y$, on a $Y > X$.

<u>Exemple</u>

Soient $\mathcal{B}_m^3 = \{2, 5, 13\}$ et $\mathcal{B}_p^3 = \{3, 7, 11\}$ les base principale et auxiliaires. On se propose de comparer $X = 4$ et $Y = 3$.

Base principale			Base auxiliaire			Remarque
2	5	13	3	7	11	
0	4	4	1	4	4	$X = 4$
1	3	3	0	3	3	$Y = 3$
1	4	12	2	6	10	$Z = Y - X$
			0	4	8	Changement de base

On constate que les valeurs de Z déterminées dans les deux bases ne correspondent pas. On en conclut que $X > Y$. ■

1.3.3.2 Comparaison de Chiang et Lu

La méthode de *Mi Lu* et *Jen-Shiu Chiang* est utilisée dans l'algorithme de division décrit dans la section (1.3.4). Ils proposent dans leurs articles [CL91] et [CL92] de déterminer le signe de la différence de deux nombres grâce à sa parité. Il est ensuite facile d'ordonner les nombres. Cette méthode de comparaison s'appuie sur les deux théorèmes suivants. On supposera dans cette sous-section que M est un nombre impair.

Le premier théorème permet de comparer deux nombres dont la parité est identique. Le second traite de deux nombres de parités différentes. Ces deux théorèmes se démontrent de la même façon.

Théorème 1.2 *Soient X et Y deux nombres de parité identique et soit $Z = X - Y \bmod M$. X est supérieur ou égal à Y si et seulement si Z est un nombre pair. X est strictement inférieur à Y si et seulement si Z est impair.*

Théorème 1.3 *Soient X et Y deux nombres de parité différentes et soit $Z = X - Y \bmod M$. X est supérieur ou égal à Y si et seulement si Z est un nombre impair. X est strictement inférieur à Y si et seulement si Z est pair.*

L'efficacité avec laquelle il est possible de comparer deux nombres réside donc dans l'efficacité avec laquelle la parité exacte d'un nombre pourra être établie. Le plus rapide serait d'introduire le modulus 2 dans le système de représentation modulaire, mais les théorèmes 1.2 et 1.3 ne seraient plus utiles, la condition sur M n'étant plus vérifiée. Il est alors nécessaire de déterminer cette parité d'une autre façon.

La formule de la démonstration du théorème des restes chinois donne une information sur les nombres manipulés. Celle-ci peut se réécrire de la façon suivante :

$$X = \sum_{i=1}^{n} \left| x_i (M_i)_i^{-1} \right|_{m_i} M_i \quad - kM \tag{1.15}$$

où k est est un entier positif.

Puisque M est impair, tous les moduli m_i sont impairs et par conséquent tous les M_i. La parité de X ne dépend donc que des $x_i(M_i)_i^{-1}$ et de k qui peut être obtenue en récupérant le bit le moins significatif (BMS) de leur expression binaire. Ainsi, s'il y a un nombre pair de nombre impairs –nombres dont le dernier bit est 1– dans la somme (1.15), alors X est pair. Dans le cas contraire X est impair. On note $P(X)$ la parité de X avec $P(X) = 0$ si X est pair et $P(X) = 1$ dans le cas contraire. On note $\oplus$ l'opération booléenne "ou exclusif". $P(X)$ s'exprime de la façon suivante :

$$\begin{aligned} P(X) \quad = \quad & BMS(\left| x_1(M_1)_1^{-1} \right|_{m_1}) \oplus BMS(\left| x_2(M_2)_2^{-1} \right|_{m_2}) \oplus \dots \\ & \oplus BMS(|x_n(M_n)_n^{-1}|_{m_n}) \oplus BMS(k) \end{aligned}$$

Le calcul du bit le moins significatif des termes $\left| x_i \times (M_i)_i^{-1} \right|_{m_i}$ peut être simplifié en tabulant la valeur des $(M_i)_i^{-1}$. La table est alors constituée d'une valeur de $\left\lceil \log_\beta(m_i) \right\rceil$ chiffres pour chaque modulus. La taille totale de cette table est $\sum_{i=1}^{n} \left\lceil \log_\beta m_i \right\rceil$. Une implémentation plus rapide consiste à tabuler le chiffre le moins significatif des produits $\left| x_i(M_i)_i^{-1} \right|_{m_i}$. La table est alors constituée de m_i valeurs de 1 bit pour chaque modulus. La taille totale de cette table est de $\sum_{i=1}^{n} m_i$.

Dans les deux implémentations, la détermination des opérandes se fait en un cycle. La seconde implémentation, bien que plus gourmande en place, évite une multiplication modulaire et a donc un cycle plus court. Le "ou exclusif" de n termes peut être déterminé en temps constant sur une machine PRAM CRCW ou en temps logarithmique avec un arbre de n processeurs [CT93].

La parité de k est plus compliquée à obtenir. L'équation (1.15) donne l'expression de k suivante :

$$k = \frac{\left|x_1(m_1)_1^{-1}\right|_{m_1}}{m_1} + \frac{\left|x_2(M_2)_2^{-1}\right|_{m_2}}{m_2} + \ldots + \frac{\left|x_n(M_n)_n^{-1}\right|_{m_n}}{m_n} - \frac{X}{M} \tag{1.16}$$

La valeur exacte de k ne peut être connue que si la représentation en numération de position des différents termes de la somme (1.16) comporte un nombre infini de chiffres. *Chiang* et *Lu* proposent de ne conserver que les t bits les plus significatifs de $\frac{\left|x_i(M_i)_i^{-1}\right|_{m_i}}{m_i}$.

$$t > \log_2 nM$$

Dans ce cas, la valeur de k se calcule à partir de valeurs approchées. Pour déterminer la parité de k la somme des seuls n termes d'au moins $t+1$ chiffres est nécessaire.

$$k = \left\lceil \frac{\left|x_1(m_1)_1^{-1}\right|_{m_1}}{m_1} \right\rceil_t + \left\lceil \frac{\left|x_2(M_2)_2^{-1}\right|_{m_2}}{m_2} \right\rceil_t + \ldots + \left\lceil \frac{\left|x_n(M_n)_n^{-1}\right|_{m_n}}{m_n} \right\rceil_t$$

<u>Exemple</u>

On se place dans le système modulaire dont la base est $\mathcal{B}_m^4 = \{3, 7, 13, 19\}$. Les entiers exprimés dans ce système sont compris dans l'intervalle $[0, 5187[$. On déduit facilement que les parties fractionnaires devront être approchées sur $t = 16$ bits. Les constantes M_i et leurs inverses modulaires M_i^{-1} sont les suivantes :

$$\begin{array}{llll} M_1 = 1729 & M_2 = 741 & M_3 = 399 & M_4 = 273 \\ M_1^{-1} = 1 & M_2^{-1} = 6 & M_3^{-1} = 3 & M_4^{-1} = 11 \end{array}$$

On considère les entiers $X = 147$, $Y = 31$ et $Z = X - Y = 116$. Leur représentation modulaire dans ce système est la suivante :

$$\begin{array}{rl} X_{RNS} = & \{0, 0, 4, 14\} \\ Y_{RNS} = & \{1, 3, 5, 12\} \\ Z_{RNS} = & \{2, 4, 12, 2\} \end{array}$$

On en déduit ensuite la valeur de k_X, k_Y et k_Z :

$$\begin{array}{l} k_X = \left\lceil 0 + 0 + \left\lceil \frac{|4 \times 3|_{13}}{13} \right\rceil_{16} + \left\lceil \frac{|14 \times 11|_{19}}{19} \right\rceil_{16} \right\rceil = \lceil 1,028\ldots \rceil = 1 \\ k_Y = \left\lceil \left\lceil \frac{|1 \times 1|_3}{3} \right\rceil_{16} + \left\lceil \frac{|3 \times 6|_7}{7} \right\rceil_{16} + \left\lceil \frac{|5 \times 3|_{13}}{13} \right\rceil_{16} + \left\lceil \frac{|12 \times 11|_{19}}{19} \right\rceil_{16} \right\rceil = \lceil 2,005\ldots \rceil = 2 \\ k_Z = \left\lceil \left\lceil \frac{|2 \times 1|_3}{3} \right\rceil_{16} + \left\lceil \frac{|4 \times 6|_7}{7} \right\rceil_{16} + \left\lceil \frac{|12 \times 3|_{13}}{13} \right\rceil_{16} + \left\lceil \frac{|2 \times 11|_{19}}{19} \right\rceil_{16} \right\rceil = \lceil 2,022\ldots \rceil = 2 \end{array}$$

On en déduit les parités de X, Y et Z :

$$\begin{array}{rll} P(X) = & 0 \oplus 0 \oplus BMS(|4 \times 3|_{13}) \oplus BMS(|14 \times 11|_{19}) \oplus BMS(k_X) & = 1 \\ P(Y) = & BMS(|1 \times 1|_3) \oplus BMS(|3 \times 6|_7) \oplus BMS(|5 \times 3|_{13}) \oplus & \\ & BMS(|12 \times 11|_{19}) \oplus BMS(k_Y) & = 1 \\ P(Z) = & BMS(|2 \times 1|_3) \oplus BMS(|4 \times 6|_7) \oplus BMS(|12 \times 3|_{13}) \oplus & \\ & BMS(|2 \times 11|_{19}) \oplus BMS(k_Z) & = 0 \end{array}$$

Par conséquent X et Y sont impairs et Z est pair. On en conclut que $X > Y$ ■

De l'algorithme de *Chiang* et *Lu* nous déduisons une implémentation possible de leur évaluation de parité (Fig. 1.10). Cette implémentation non optimisée permet d'estimer le coût de cet algorithme. Elle se décompose en deux parties. En premier lieu, l'estimation k_X demande un arbre d'additionneurs de $t+1$ bits de large qui peut être réalisé avec n additionneurs. Chacun d'eux est un additionneur à virgule fixe avec t bits pour la partie décimale et 1 bit pour la partie entière. Les retenues sortant étant inutiles sont omises. Ensuite l'évaluation de la parité peut être effectuée par n opérateurs "ou exclusif".

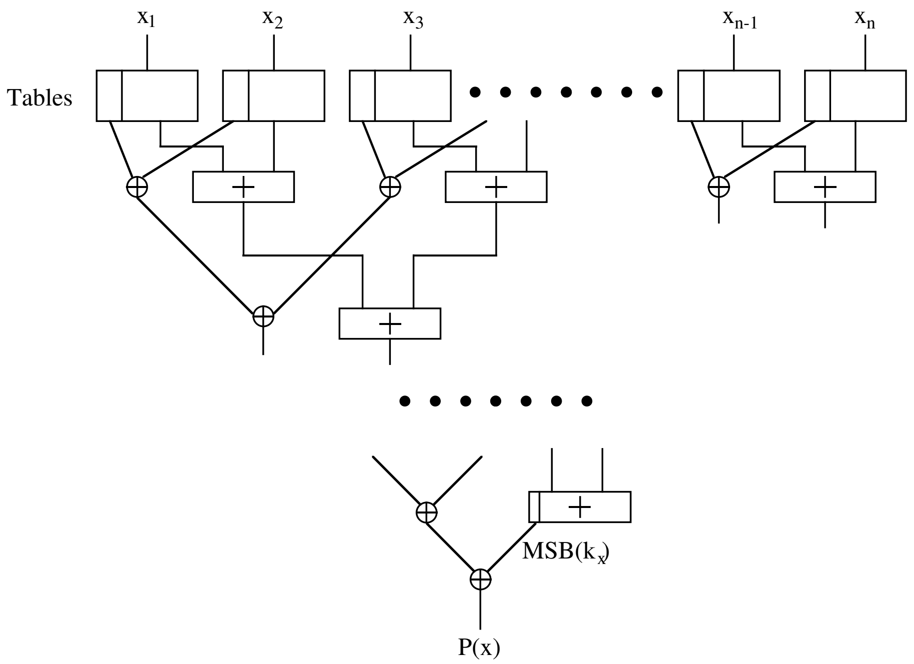


FIG. 1.10 – *Une architecture de la détermination de la parité d'un nombre par la méthode de Mi Lu et Jen-Shiun Chiang*

Les valeurs manipulées par cet algorithme sont tabulées. D'une part, lors du calcul de la parité de k_x, les valeurs de la somme 1.16 sont stockées dans une table de m_i éléments de t bits par modulus. D'autre part, la parité des $\left|x_i(M_i)_i^{-1}\right|_{m_i}$ utilisée dans le calcul du "ou exclusif" est obtenue grâce à une table de m_i éléments de 1 bit pour chaque modulus. La taille totale des tables utilisées est alors de $(t+1)\sum_{i=1}^{n} m_i$.

Le calcul de la parité est effectué avec une complexité en temps de $\mathcal{O}(\log n)$. La table 1.5 résume les caractéristiques de l'implémentation de la méthode de comparaison de *Mi Lu* et *Jen-Shiun Chiang* que nous proposons.

Taille des additionneurs	$\lceil \log_\beta nM \rceil + 1$
Nombre d'additionneurs	n
Nombre d'opérateurs "Xor"	$n+1$
Taille des tables	$\left(\left\lceil \log_\beta nM \right\rceil + 1\right) \sum_{i=1}^{n} m_i$
Complexité en temps	$\log_2 n + 1$

TAB. 1.5 – *Caractéristiques de la méthode de comparaison de Mi Lu et Jen-Shiun Chiang*

L'implémentation, bien que grossière, de la somme telle qu'elle est donnée ici permet de déterminer l'ordre de grandeur du coût de cette architecture. Cette méthode de comparaison est très coûteuse en place. La taille des tables données dans [CL92] et reprises dans l'implémentation ci-dessus montre qu'elle est impraticable pour un grand nombre de moduli. Par exemple pour manipuler des nombres d'environ 1000 bits avec de moduli de 16 bits, il est nécessaire d'avoir une base modulaire de 64 éléments. Avec une telle base les tables comporteraient près de 2^{32} bits.

Dans leur article *Chiang* et *Lu* proposent d'utiliser l'additionneur proposé dans [AO79]. Cet algorithme de comparaison est plutôt adapté à la comparaison dans de petits systèmes modulaires en raison de la taille des tables qu'il utilise.

1.3.3.3 Comparaison de Dimauro, Impedovo et Pirlo

La méthode de comparaison proposée par *Dimauro*, *Impedovo* et *Pirlo* [DIP93] s'appuie sur l'observation suivante : dans l'espace fini à n dimensions défini par les n moduli $\{m_1, m_2, \ldots, m_n\}$, les entiers sont ordonnés selon les diagonales principales de cet espace. La figure 1.11 les met en évidence dans le plan pour les moduli $\{13, 7\}$.

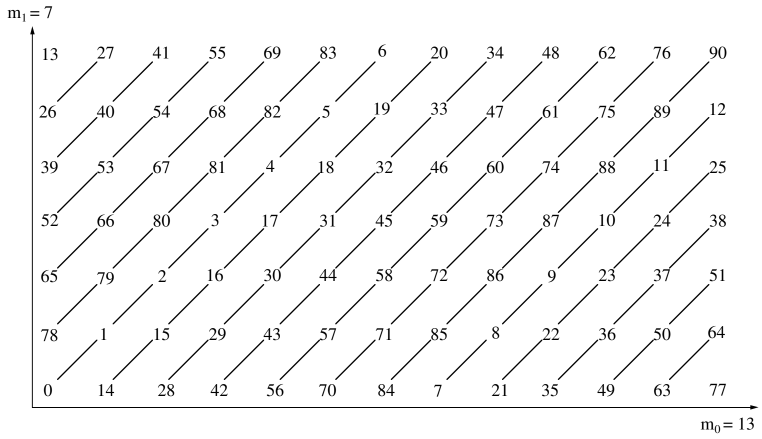


FIG. 1.11 – *Arrangement des nombres dans le plan avec les moduli* $\{13, 7\}$

L'idée est de trouver une fonction $\mathcal{D}$ qui permette d'ordonner ces diagonales. Etant donné un entier X, cette fonction retourne le numéro de la diagonale sur laquelle il se trouve (Fig. 1.12). Ainsi si deux entiers X et Y ne se trouvent pas sur la même diagonale, leur ordre est celui donné par cette fonction. Dans le cas contraire, la relation d'ordre

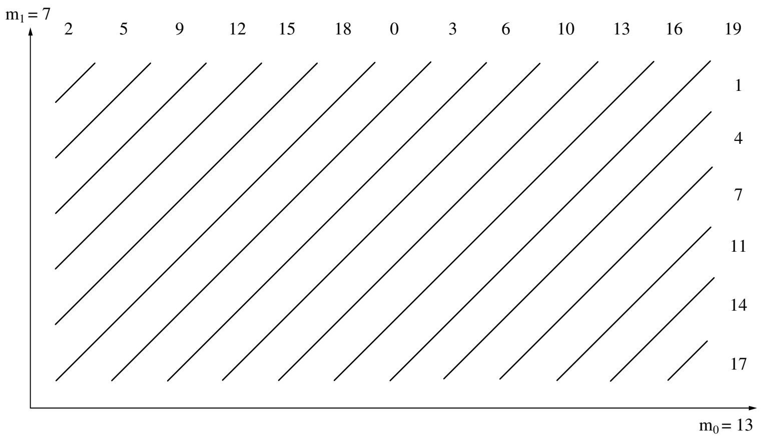


FIG. 1.12 – *Ordre des diagonales principales dans le plan avec les moduli* $\{13, 7\}$

entre X et Y est la même que celle entre x_i et y_i quel que soit i. La fonction $\mathcal{D}(X)$ est définie de la façon suivante :

$$\mathcal{D}(X) = -x_1 m_{1,SQ}^{-1} - x_2 m_{2,SQ}^{-1} - \ldots - x_n m_{n,SQ}^{-1} \bmod SQ \qquad (1.17)$$

où $m_{i,SQ}^{-1}$ est l'inverse multiplicatif de m_i modulo SQ et

$$SQ = M_1 + M_2 + \ldots + M_n$$

Les inverses $m_{i,SQ}^{-1}$ existent car il a été prouvé que, pour tout $0 \leq i < n$, SQ et m_i sont premiers entre eux [DIP93].

Exemple :

On considère le système modulaire dont la base est :

$$\mathcal{B}_m^2 = \{7, 13\}$$

On en déduit que $SQ = 20$ et :

X	$X \bmod m_1$	$X \bmod m_2$	$\mathcal{D}(X)$
10	3	10	1
14	0	1	3
20	6	7	3

Au vu des valeurs de $\mathcal{D}$ on conclut rapidement que $10 < 14$ et $10 < 20$, mais aucune conclusion peut être faite sur la relation d'ordre entre 14 et 20. Dans ce cas l'ordre est obtenu en comparant un des restes de la représentation de chacun des nombres, par exemple ceux du second modulus 13. Il apparaît alors que 14 est bien strictement inférieur à 20 ■

Dimauro, *Impedovo* et *Pirlo* ne donnent pas d'exemple d'implémentation du calcul de la fonction $\mathcal{D}(X)$. Nous présentons ici une implémentation possible déduite de la description de leur algorithme. Elle est illustrée par la figure 1.13. Les n valeurs $-m_{i,SQ}^{-1}$ sont

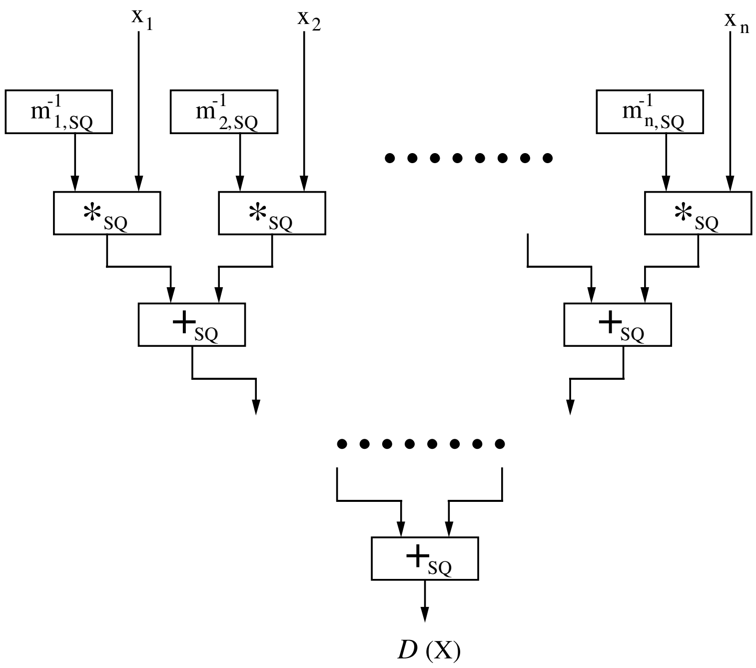


FIG. 1.13 – *Implémentation du calcul de la fonction* $\mathcal{D}(X)$

stockées dans n tables de 1 élément de $\lceil \log_\beta SQ \rceil$ chiffres. Les termes de la somme (1.17) sont ensuite calculés par n multiplieurs modulaire de module SQ notés $*_{SQ}$. La somme est déterminée à l'aide d'un arbre d'additionneurs modulaires de module SQ constitué de n additionneurs notés $+_{SQ}$. Le calcul de cette fonction s'effectue donc en $\mathcal{O}(\log_2 n)$ étapes avec des nombres de $\log_\beta SQ$ chiffres. La table 1.6 présente le coût d'implémentation de la méthode de [DIP93].

Taille des opérandes	$p \times (n-1) + n$
Nombre d'additionneurs	n
Nombre de multiplieurs	n
Taille des tables	$p \times n \times (n-1) + n^2$
Complexité en temps	$\mathcal{O}(\log_2 n)$

TAB. 1.6 – *Caractéristiques de la méthode de comparaison de Dimauro, Impedovo et Pirlo*

Si le système modulaire utilisé comporte un grand nombre de moduli, c'est-à-dire si les nombres manipulés sont de très grands nombres, SQ est un très grand entier et les calculs effectués modulo SQ peuvent être très coûteux. *Dimauro*, *Impedovo* et *Pirlo* proposent de regrouper les moduli en pseudo-moduli moins nombreux mais plus petits. Ainsi SQ est plus petit et les opérations modulo SQ moins coûteuses.

Exemple :

Le système modulaire est constitué de la base suivante :

$$\mathcal{B}_m^4 = \{3, 7, 13, 19\}$$

Cet exemple présente deux choix de regroupements de moduli comparés à la méthode de calcul sans regroupements.

	Bases		
	$\{21, 247\}$	$\{57, 91\}$	$\{3, 7, 13, 19\}$
	SQ = 268	SQ = 148	SQ = 3142
X	$\mathcal{D}(X)$	$\mathcal{D}(X)$	$\mathcal{D}(X)$
31	1	0	17
147	7	3	88
3947	202	112	2388

Le second regroupe d'une part le plus gros modulus avec le plus petit, et d'autre part les deux moduli restant. La base virtuelle est alors $\{57, 91\}$. Le tableau précédent présente les résultats ■

Le calcul de la fonction $\mathcal{D}(X)$ n'évite néanmoins pas la manipulation de nombres dont l'ordre de grandeur est proche de M. Bien que le regroupement des moduli puisse réduire la taille des opérandes, cet algorithme reste inadapté à la manipulation de grands nombres. En effet la détermination de la fonction $\mathcal{D}(X)$ a un coût équivalent à la conversion par le théorème des restes chinois.

1.3.3.4 Comparaison approchée de Hung et Parhami

Cette comparaison approchée qui se base sur l'équation du théorème des restes chinois permet d'estimer le signe d'un nombre rapidement. La méthode présentée dans [HP94a] s'apparente à la méthode de conversion de la représentation modulaire des nombres vers la représentation pseudo-flottante présenté au chapitre 2 et [BDM96].

Hung et *Parhami* organisent les nombres signés de manière différente à celle proposée précédemment. Les nombres inférieurs à $\frac{M}{2}$ sont positifs et les nombres supérieurs à $\frac{M}{2}$ sont négatifs (Fig 1.14).

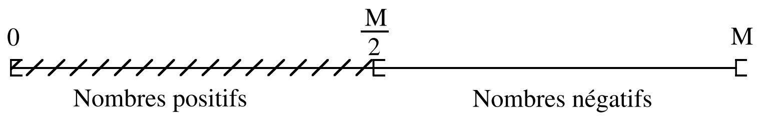


FIG. 1.14 – *Répartition des entiers relatifs – Comparaison de Hung et Parhami*

D'après le théorème des restes chinois, déterminer le signe de X revient à déterminer dans quelle partie de l'intervalle $[0, M-1]$ se situe X.

$$X = \sum_{i=1}^{n} x_i (M_i)_i^{-1} M_i \bmod M$$

Etant donné que les opérations sont effectuées modulo M, cela revient à déterminer où se situe la partie fractionnaire de $F(X) = [\frac{X}{M}]$ dans l'intervalle $[0, 1[$:

$$F(X) = \frac{X}{M} = \left[\sum_{i=1}^{n} \left[\frac{x_i (M_i)_i^{-1}}{m_i} \right] \right] \tag{1.18}$$

Ainsi si $0 \leq F(X) < 1/2$ alors X est un nombre positif et si $1/2 \leq F(X) < 1$ alors X est un nombre négatif. La somme de n éléments (1.18) peut être calculée en parallèle en temps logarithmique avec un arbre d'additionneurs. La valeur de $F(X)$, bien que comprise dans un petit intervalle, nécessite la manipulation de nombres dont la partie fractionnaire peut être infinie. On se propose de calculer la valeur approchées de $F(X)$ notée $EF(X)$ où la partie fractionnaire de chacun des termes de l'équation (1.18) est tronqué au $\gamma^{ème}$ bit :

$$EF(X) = \left[\sum_{i=1}^{n} \left[\frac{x_i (M_i)_i^{-1}}{m_i} \right]_\gamma \right] \tag{1.19}$$

Le signe approché de X est obtenu par simple comparaison de $EF(X)$ à des bornes fixées. Il est possible d'exprimer $F(X)$ en fonction de $EF(X)$, et donc de déterminer le signe de X en fonction des valeurs prise par $EF(X)$.

1. Si $0 \leq EF(X) < \frac{1}{2}$ et si $X \notin [M \times 2^{-\alpha}, M - M \times 2^{-\alpha}]$ avec $\alpha = \gamma - \lceil \log n \rceil$, alors X est un nombre positif.
2. Si $\frac{1}{2} \leq EF(X) < 1 - 2^{-\alpha}$ avec $\alpha = \gamma - \lceil \log n \rceil$, alors X est négatif.
3. Si $\frac{1}{2} - 2^{-\alpha} \leq EF(X) < 1$ alors X est une une valeur proche de 0 qui peut être aussi bien positive que négative. Son signe ne peut donc être déterminé et en outre $-2^{-\alpha} M \leq X \leq 2^{-\alpha} M$.

<u>Exemple</u>

La base modulaire utilisée dans cet exemple est $\mathcal{B}_m^4 = \{5, 7, 9, 11\}$. Avec $\alpha = 4$ et $\gamma = \alpha + \lceil \log n \rceil = 6$, les valeurs $EF_\alpha[i][j]$ sont tronquées au 1/64. Pour la détermination du signe, la somme $EF_\alpha(X)$ est comparée avec les bornes de l'intervalle $[1/2, 1 - 2^{-\alpha}]$ qui correspondent à $[1/2, 60/64]$. Par exemple :

$$EF_\alpha(2) = EF_\alpha(\{2,2,2,2\}) = \left[\frac{51 + 54 + 56 + 29}{64} \right] = \frac{62}{64} \geq \frac{60}{64}$$

Le signe du nombre modulaire $\{2, 2, 2, 2\}$ est alors indéterminé et $ES_\alpha(\{2,2,2,2\}) = \pm$.

Un autre exemple donne :

$$EF_\alpha(100) = EF_\alpha(\{0,2,1,1\}) = \left[\frac{0 + 54 + 28 + 46}{64} \right] = 0$$

On en déduit le signe $ES_\alpha(\{0,2,1,1\}) = +$ ■

Hung et Parhami ne proposent pas d'implémentation de leur algorithme de comparaison. Afin d'estimer le coût d'une telle comparaison nous avons déduit de leur algorithme une implémentation possible. Celle-ci est assez simple et permet de comparer deux nombres en temps logarithmique.

Les valeurs de $\left[\frac{x_i (M_i)_i^{-1}}{m_i} \right]_\gamma$ sont stockées dans n tables ayant pour entrée x_i. La constitution de telles tables nécessite de stocker $\sum_{i=1}^{n} m_i$ valeurs de γ bits. Leur taille peut

Nombre de multiplieurs	n	0
Nombre d'additionneurs	n	n
Taille des tables	$n \times \gamma$	$\gamma \times \sum_{i=1}^{n} m_i$
Complexité en temps	$\mathcal{O}(\log_2 n)$	$\mathcal{O}(\log_2 n)$

TAB. 1.7 – *Caractéristiques de la méthode de comparaison de Hung et Parhami*

être réduite en ne stockant que les $\left[\frac{(M_i)_i^{-1}}{m_i}\right]_\gamma$. Dans ce cas, seules n valeurs de γ bits sont mémorisées mais un multiplieur est nécessaire. Les opérations sont effectuées sur des opérandes de γ bits. La valeur de $EF(X)$ est alors déterminée par un arbre de n additionneurs en temps logarithmique. La table suivante résume les caractéristiques de cette implémentation.

1.3.4 Division exacte

La plupart des algorithmes de division font une large utilisation de la comparaison ce qui en rend l'adaptation difficile à l'arithmétique modulaire. Toutefois il existe un cas particulier pour lequel la division de X par Y en arithmétique modulaire se ramène à une multiplication modulaire. Nous allons traiter ici de ce cas particulier, le cas général de la division étant traité dans le chapitre 2.

Il est possible de déterminer l'inverse de Y modulo M si PGCD$(Y, M) = 1$. La représentation modulaire de celui-ci, notée Y_M^{-1}, peut être obtenu en déterminant l'inverse modulaire de chacun des restes de Y modulo m_i à l'aide de l'algorithme d'Euclide étendu en temps logarithmique [Knu81]. Ceux-ci constituent la représentation modulaire de Y_M^{-1}.

$$Y_M^{-1} = \{y_{1,m_1}^{-1}, y_{2,m_2}^{-1}, \cdots, y_{2,m_2}^{-1}\}$$

où y_{i,m_i}^{-1} est l'inverse modulaire de y_i. En d'autres termes, c'est l'entier qui vérifie :

$$\left| y_i \times y_{i,m_i}^{-1} \right|_{m_i} = 1$$

De plus, si X est un multiple de Y, alors la division de X par Y se ramène à la multiplication de X par l'inverse de Y modulo M :

$$\frac{X}{Y} = XY_M^{-1}$$

Les restes y_{i,m_i}^{-1} pouvant être calculés indépendamment les uns des autres, la division exacte peut être effectuée en temps constant en parallèle.

Exemples :

On considère la base modulaire $\mathcal{B}_m^4 = \{5, 7, 9, 11\}$ et les nombres suivants :

$$X = 78 = \{3, 1, 6, 1\}$$
$$Y_1 = 2 = \{2, 2, 2, 2\}$$
$$Y_2 = 12 = \{2, 5, 3, 1\}$$

Les nombres Y_1 et Y_2 sont premiers avec $M = 3465$. Leurs inverses modulo M existent et :

$$\begin{array}{l} Y_1^{-1} = 1733 = \{3,4,5,6\} \\ Y_2^{-1} = 289 = \{3,4,5,6\} \end{array}$$

Leur produit par X donne :

$$\begin{array}{l} X \times Y_1^{-1} = \{4,4,3,6\} = 39 \\ X \times Y_2^{-1} = \{2,2,6,3\} = 1752 \end{array}$$

La division de X par Y_1 étant exacte, le produit de X par l'inverse modulaire de Y_1 correspond bien au quotient. Ce n'est pas le cas pour Y_2 ■

1.3.5 Bases incomplètes de Gamberger

D. Gamberger [Gam89] introduit la notion de bases incomplètes. Les nombres dont la représentation modulaires est constituée de tous les restes x_i sont des cas particuliers de la représentation en bases incomplètes. Il est en effet possible de définir un entier dont seuls quelques restes sont représentés. Cela équivaut à représenter ce nombre dans un sous-système modulaire constitué de quelques éléments de la base.

Il est à noter que tous les entiers du système modulaire de base ne sont pas représentables de cette façon puisque la largeur de l'intervalle est plus réduite. Pour définir une telle représentation, *Gamberger* considère que chaque reste x_i associé à un modulus m_i peut prendre une valeur indéfinie u en plus de valeur $0, 1, 2, \cdots, m_i - 2, m_i - 1$. L'intervalle des entiers représentables dans cette base incomplète se réduit à $[0, M/M_u[$, avec :

$$M_u = \prod_{i=1}^{n} (m_i)^{\bar{\delta}_i}$$

L'exposant binaire δ_i vaut 1 si le modulus m_i est défini et 0 dans le cas contraire. La notation $\bar{\delta}_i$ est l'inverse logique de δ_i.

Les opérations dans les systèmes incomplets et entre systèmes incomplets sont identiques aux opérations dans les systèmes modulaires complets à ceci près que le résultat d'une opération avec un reste indéfini donne un résultat indéfini. La division exacte est toujours possible dans un tel système. Dans la sous-section 1.3.4 il a été montré que l'inverse d'un entier ne pouvait pas toujours exister. Dans ce système l'inverse multiplicatif de tout reste nul prend la valeur u, ainsi que celui de tout reste indéfini. La division qui se résume à la multiplication de deux nombres est la seule opération qui peut générer un résultat dont la représentation est incomplète à partir d'opérandes complètement définies.

<u>Exemple :</u>

On considère la base $\mathcal{B}_m^4 = \{3,5,7,11\}$. Avec :

$$X = 36 = \{0,1,1,3\} \qquad Y = 6 = \{0,u,6,6\}$$

Alors

$$X + Y = \{0,u,0,9\} = 42$$

$$X - Y = \{0, u, 2, 8\} = 30$$
$$X \times Y = \{0, u, 6, 7\} = 216$$
$$X/Y = \{u, u, 6, 6\} = 6 \quad \blacksquare$$

La conversion d'une représentation incomplète vers une représentation complète se fait grâce à une extension de base. Celle-ci permet de calculer les restes indéterminés à partir des autres. *Gamberger* propose d'utiliser une extension de base similaire à celle de *Szabo* et *Tanaka*.

Les bases incomplètes permettent de tester la divisibilité de deux nombres X et Y. Si le reste y_i est nul et le reste x_i non nul, on conclut à la non divisibilité de X par Y. Dans les autres cas, *Gamberger* propose d'utiliser en plus de la base principale $\mathcal{B}_m^n$, une base auxiliaire $\mathcal{B}_p^k$ telle que $\prod_{i=1}^k p_i > M$. La division directe est ensuite effectuée dans les deux systèmes modulaires. Si les quotients trouvés dans les deux systèmes modulaires sont égaux, alors Y divise exactement X. Ce test peut être obtenu à l'aide d'une extension de base.

Une des propriétés de la représentation modulaire est qu'il est possible de calculer une expression du type $X = A + B + C - D - E$ sans tenir compte du nombre d'opérations et des dépassements de capacités à partir du moment où $X \leq \lceil (M-1)/2 \rceil$ [Gam89]. De la même façon *Gamberger* montre que les opérations du type $X = \frac{A \times B \times C}{D \times E}$ donnent un résultat correct si $X \leq \left\lceil \left(\frac{M}{PGCD(D \times E, M)} - 1 \right) / 2 \right\rceil$.

Chapitre 2

Division en représentation modulaire

On a vu dans le chapitre précédent que la représentation modulaire est une représentation qui permet, d'effectuer des multiplications, des additions et des soustractions en temps constant. Ainsi est-il possible d'effectuer des calculs sur de très grands nombres en les ramenant à des calculs sur de petits nombres. Cette propriété de la représentation modulaire semble intéressante pour de nombreuses applications telles que les applications cryptographiques [CB96] [PP92] ou la géométrie algorithmique [BEPP97].

Toutefois, la détermination de la division en représentation modulaire est moins facile. L'implémentation des algorithmes classiques de division [Mul89] –qui nécessitent de nombreuses comparaisons– reste difficile en représentation modulaire. Dans ce chapitre, nous présentons un nouvel algorithme de division basé sur une arithmétique mixte où nous associons à la représentation modulaire un couple d'entiers donnant l'ordre de grandeur et les premiers chiffres significatifs du nombre. Nous appellerons cette représentation la *représentation pseudo-flottante*. Nous définirons cette arithmétique pseudo-flottante, qui est la borne inférieure d'une représentation en arithmétique d'intervalles, dans la section 2.1.

Nous présentons dans la section 2.2 un algorithme de division qui permet de diviser deux grands nombres exprimés dans un système modulaire que nous avons présenté dans [BDM96] et [BDM], ainsi que quelques exemples. Son principe est similaire à celui d'un algorithme de division en grande base. Ici la représentation pseudo-flottante représente un chiffre.

Le problème de la division en arithmétique RNS a largement été débattu dans la littérature scientifique. Nous présentons dans la section 2.3 quelques-uns de ces algorithmes et quelques éléments permettant de les comparer au notre. Nous nous attacherons également à étudier le comportement de ces algorithmes de division pour de grands systèmes modulaires.

2.1 Définition de l'arithmétique pseudo-flottante

La représentation "pseudo-flottante" (PF) d'un nombre X est déterminée à partir de sa représentation modulaire. Cette conversion se ramène à une conversion de la représentation modulaire vers une représentation de numération de position. Dans le chapitre 1 plusieurs algorithmes de conversion ont été décrits. La conversion vers une représentation "Mixed Radix" a l'avantage de ne manipuler que de petits nombres mais elle est linéaire et trop lente pour une utilisation fréquente. On lui préfèrera donc un algorithme basé sur la démonstration du théorème des restes chinois (TRC) qui, bien que manipulant de très grands nombres, est facilement parallélisable.

L'idée exploitée pour la création de la représentation PF est d'utiliser un algorithme de conversion basé sur le TRC avec des valeurs approchées de façon à réduire le coût de la conversion.

2.1.1 Définition

Nous définissons dans cette section la notation pseudo-flottante. Cette représentation est déterminée à partir de la représentation modulaire ayant pour base $\mathcal{B}_m^n$. La représentation PF est déduite du théorème suivant dont la démonstration est fournie dans la sous-section 2.2.2.

Théorème 2.1 *Soit la base modulaire* $\mathcal{B}_m^n = \{m_1, \ldots, m_n\}$. *Soient* $\beta \geq 2$ *et* m_e *deux entiers tels que :*

$$\beta^{m_e-1} \leq m_i < \beta^{m_e} \quad \text{pour} \quad i = 1, \ldots, n \tag{2.1}$$

Soit X *un entier dont la représentation modulaire dans la base* $\mathcal{B}_m^n$ *est* $\{x_1, \ldots, x_n\}$. *On considère que les restes* x_i *sont représentés en base* β.

Si on connaît X_k *tel que*

$$\beta^{-m_e+l+1} \leq \frac{X \times \beta^{X_k}}{M} < \beta^{-1} \tag{2.2}$$

avec $l = \lceil \log_\beta n \rceil + 1$.

Alors nous pouvons calculer les $m_e - l$ *premiers chiffres de l'approximation* X_f *de* $\frac{X \times \beta^{X_k}}{M} \times \beta^{m_e}$, *en n'utilisant que des entiers d'au plus* $2m_e$ *chiffres. L'approximation* X_f *satisfait l'équation suivante :*

$$X_f \leq \frac{X \times \beta^{X_k+m_e}}{M} < X_f + \beta^l \tag{2.3}$$

■

Dans le théorème 2.1 l'entier X_k est un entier relativement petit ($X_k < n \times m_e$) de telle sorte que $X \times \beta^{X_k}$ soit proche de M. Cet entier indique donc l'ordre de grandeur de X relativement à M. Ainsi, plus X est petit, plus la valeur X_k qui lui est associée est grande. On remarque en outre que X_f est un entier de m_e chiffres. Nous avons :

$$X_f \times \beta^{-X_k} \simeq X\left(\frac{\beta^{m_e}}{M}\right) \tag{2.4}$$

Par conséquent le couple (X_k, X_f) peut être vu comme une représentation en virgule flottante de X ayant un facteur constant.

Definition 1 *Le couple* (X_k, X_f) *est appelé représentation* Pseudo-Flottante *du nombre modulaire* X_{RNS} *;* X_k *est l'"exposant" de cette représentation et* X_f *la mantisse.*

Plusieurs valeurs de (X_k, X_f) peuvent satisfaire la définition précédente. La représentation la plus précise est celle qui a la mantisse ayant la valeur la plus importante – de la même manière que les représentations en virgule flottante – et par conséquent avec la plus grande valeur de l'exposant X_k.

2.1.2 Calculs en arithmétique pseudo-flottante

La représentation PF est manipulée en parallèle avec l'arithmétique RNS. Les opérations effectuées dans l'une le sont en même temps dans l'autre. L'algorithme d'addition est identique à l'algorithme d'addition en arithmétique à virgule flottante [Kor93a]. L'algorithme de multiplication est légèrement différent.

Pour calculer $X \times Y$ en arithmétique PF, nous déterminons en premier lieu $(X_f \times \beta^{-X_k})(Y_f \times \beta^{-X_k})$ en utilisant le même algorithme que la multiplication en virgule flottante. C'est à dire que nous multiplions les mantisses et additionnons les exposants. Ensuite nous multiplions le résultat par une approximation pseudo-flottante de $\frac{M}{\beta^{m_e}}$ de façon à éliminer le terme supplémentaire $\frac{\beta^{m_e}}{M}$ qui apparaît. On remarque que l'élimination de ce terme double l'erreur d'arrondi. Dans le cas où k multiplications successives sont effectuées, il est donc préférable d'éliminer le facteur supplémentaire $(\frac{\beta^{m_e}}{M})^k$ en une seule étape à la fin du calcul.

En règle générale, cette représentation est suffisante pour comparer deux nombres. Toutefois, si les deux nombres sont proches l'un de l'autre, il est nécessaire d'effectuer une conversion vers un système permettant la comparaison. Si, après plusieurs calculs, la représentation PF devient imprécise à cause des erreurs d'arrondi accumulées, il est nécessaire de recalculer la valeur pseudo-flottante des nombres manipulés. On peut gérer l'erreur commise à l'aide de l'arithmétique d'intervalle qui nous donne un encadrement de l'erreur commise lors de chaque calcul, ce qui nous permet de décider quand effectuer la reconstruction. En particulier, si l'erreur commise est de 1 bit lors de chaque calcul, cette reconstruction peut être effectuée toutes les $m_e - l - 1$ opérations [Gol91]. Dans ce cas, la reconstruction d'un nombre X est effectuée en deux étapes : il faut déterminer un X_k correspondant à la définition, c'est à dire l'ordre de grandeur de X relativement à M, puis on reconstruit la mantisse X_f comme précisé dans la sous-section suivante.

2.1.3 Construction de la représentation pseudo-flottante

Le moyen le plus rapide d'obtenir une approximation de X est d'utiliser la formule de la preuve du théorème des restes chinois. En effet cette méthode peut être calculée en parallèle en temps logarithmique. D'après la preuve du théorème des restes chinois nous avons

$$X = \sum_{i=1}^{n} x_i \times (M_i)_{m_i}^{-1} \times M_i \mod M$$

avec $M_i = \dfrac{M}{m_i}$ et $(M_i)_{m_i}^{-1} \times M_i \equiv 1 \mod m_i$. On remarque que :

$$\sum_{i=1}^{n} x_i \times (M_i)_{m_i}^{-1} \times M_i = \left(\sum_{i=1}^{n} x_i \times \frac{(M_i)_{m_i}^{-1}}{m_i} \right) \times M$$

Dans cette reconstruction tous les calculs peuvent être fait avec des entiers. En effet, puisque tous les calculs sont effectués modulo M, il est suffisant de ne considérer que la partie fractionnaire de la somme et de chacun de ses membres. Seuls les chiffres les plus significatifs des parties fractionnaires sont manipulés lors des calculs approchés et nous utiliserons donc l'approximation de la partie fractionnaire à l'aide d'entiers.

2.1.3.1 Construction de la mantisse X_f

Cette sous-section est la preuve par construction du théorème 2.1. Nous désirons construire la valeur X_f à l'aide d'entiers de m_e chiffres. Dans ce calcul, seuls les m_e premiers chiffres de la partie fractionnaire de $x_i \times \frac{(M_i)^{-1}_{m_i}}{m_i}$ et $\sum_{i=1}^{n} x_i \times \frac{(M_i)^{-1}_{m_i}}{m_i}$ sont utilisés.

D'après les conditions du théorème 2.1 nous savons que l'exposant X_k est tel que :

$$\beta^{-m_e+l+1} \leq \frac{X \times \beta^{X_k}}{M} < \beta^{-1}$$

Par conséquent il y a au moins un chiffre significatif dans les $m_e - l$ premiers chiffres de la partie fractionnaire de $(X \times \beta^{X_k})/M$. On note $\widetilde{X} = X \times \beta^{X_k}$ et $\{\tilde{x}_1, \ldots, \tilde{x}_n\}$ la représentation modulaire de $\widetilde{X}$ dans $\mathcal{B}^n_m$.

On remarque tout d'abord que :

$$\beta^{-m_e} < \frac{(M_i)^{-1}_{m_i}}{m_i} < 1$$

Les entiers Ω_i sont constitués par les $2m_i$ premiers chiffres de la partie entière de $\frac{(M_i)^{-1}_{m_i}}{m_i}$:

$$\Omega_i = \left\lfloor \frac{(M_i)^{-1}_{m_i}}{m_i} \times \beta^{2m_e} \right\rfloor$$

$$\frac{(M_i)^{-1}_{m_i}}{m_i} = 0 \bullet \underbrace{\boxed{\text{Nombre de } m_e \text{ chiffres}}\boxed{\text{Nombre de } m_e \text{ chiffres}}}_{\Omega_i \text{ est un nombre de } 2m_e \text{ chiffres}}\ldots$$

Par conséquent, on a :

$$\frac{(M_i)^{-1}_{m_i}}{m_i} \times \beta^{2m_e} - 1 < \Omega_i \leq \frac{(M_i)^{-1}_{m_i}}{m_i} \times \beta^{2m_e} \tag{2.5}$$

Les termes Ω_i sont des constantes précalculées et stockées dans des tables. L'étape suivante est la multiplication de Ω_i par x_i. Le résultat est un nombre de $3m_e$ chiffres. L'entier constitué des m_e premiers chiffres de la partie décimale de $x_i \times \Omega_i \times \beta^{-2m_e}$ est appelé α_i :

$$\alpha_i = \left\lfloor \tilde{x}_i \times \Omega_i \times \beta^{-m_e} \right\rfloor - \left\lfloor \tilde{x}_i \times \Omega_i \times \beta^{-2m_e} \right\rfloor \times \beta^{m_e}$$

$$\tilde{x}_i \times \Omega_i \times \beta^{-2m_e} = \underbrace{\boxed{\text{nombre de } m_e \text{ chiffres}}}_{\text{partie entière}} \bullet \underbrace{\boxed{\text{nombre de } m_e \text{ chiffres}}}_{\alpha_i}\underbrace{\boxed{\text{nombre de } m_e \text{ chiffres}}}_{\text{erreur}}$$

Par conséquent, on a :

$$\tilde{x}_i \times \Omega_i \times \beta^{-m_e} - 1 < \alpha_i + \left\lfloor \tilde{x}_i \times \Omega_i \times \beta^{-2m_e} \right\rfloor \times \beta^{m_e} \leq \tilde{x}_i \times \Omega_i \times \beta^{-m_e}$$

On déduit de l'équation (2.5) :

$$
\begin{aligned}
&\tilde{x}_i \times \frac{(M_i)^{-1}_{m_i}}{m_i} \times \beta^{m_e} - \tilde{x}_i \times \beta^{-m_e} - 1 \\
&< \alpha_i + \lfloor \tilde{x}_i \times \Omega_i \times \beta^{-2m_e} \rfloor \times \beta^{m_e} \\
&\leq \tilde{x}_i \times \frac{(M_i)^{-1}_{m_i}}{m_i} \times \beta^{m_e}
\end{aligned} \tag{2.6}
$$

Finalement, l'entier σ est défini comme étant l'entier constitué des $m_e - l$ premiers chiffres de la partie entière de $\sum_{i=1}^{n} \alpha_i \times \beta^{-m_e}$ suivie de l zéros, à savoir :

$$
\sigma = \left\lfloor \sum_{i=1}^{n} \alpha_i \times \beta^{-l} \right\rfloor \times \beta^{l} - \left\lfloor \sum_{i=1}^{n} \alpha_i \times \beta^{-m_e} \right\rfloor \times \beta^{m_e} \tag{2.7}
$$

Les l derniers chiffres de σ sont mis à zéro de façon à compenser l'erreur d'addition lors du calcul $\sum_{i=1}^{n} \alpha_i$. Cette troncation est suffisante pour que l'inégalité 2.3 soit satisfaite.

$$
\begin{array}{cccc}
 & 0 & \bullet & \boxed{\qquad\alpha_1\qquad} \\
+ & 0 & \bullet & \boxed{\qquad\alpha_2\qquad} \\
+ & & \vdots & \\
+ & 0 & \bullet & \boxed{\qquad\alpha_n\qquad} \\
\hline
 & \boxed{\qquad} & \bullet & \underbrace{\boxed{\qquad\qquad}\ \overset{l}{\overleftrightarrow{0\cdots 0}}}_{\sigma}
\end{array}
$$

Par conséquent :

$$
\sum_{i=1}^{n} \alpha_i - \beta^{l} < \sigma + \left\lfloor \sum_{i=1}^{n} \alpha_i \times \beta^{-m_e} \right\rfloor \times \beta^{m_e} \leq \sum_{i=1}^{n} \alpha_i \tag{2.8}
$$

On déduit alors des équations (2.6) et (2.8) que :

$$
\widetilde{X} \times \beta^{m_e} - \sum_{i=1}^{n} \tilde{x}_i \times \beta^{-m_e} \times M - n \times M < \sigma \times M \leq \widetilde{X} \times \beta^{m_e}
$$

Ainsi,

$$
\widetilde{X} \times \beta^{m_e} - 2 \times n \times M < \sigma \times M \leq \widetilde{X} \times \beta^{m_e}
$$

Avec la définition de l on conclut que :

$$
\widetilde{X} \times \beta^{m_e} - \beta^{l} \times M < \sigma \times M \leq \widetilde{X} \times \beta^{m_e}
$$

Ce qui donne :

$$
\sigma \leq \frac{\widetilde{X}\beta^{m_e}}{M} < \sigma + \beta^{l} \tag{2.9}
$$

Remarque : Suite à la condition (2.2), la valeur de σ doit être dans l'intervalle $[0, \beta^{m_e-1} - \beta^{l}]$. Ainsi si $\sigma > \beta^{m_e-1}$, puisque nous calculons la somme $\sum_{i=1}^{n} \alpha_i$ avec des valeurs par défaut, nous obtenons une valeur qui est juste inférieure à un entier. Cette valeur de σ peut être considérée comme nulle étant donné que $(\sigma + \beta^{l})/\beta^{m_e} > 1$. Donc, d'après les équations (2.3) et (2.9), si $\sigma \neq 0$, alors on peut prendre $X_f = \sigma$. Or si $\sigma = 0$, cela signifie que $\widetilde{X}$ est petit comparé à M. C'est à dire que $\widetilde{X} < M \times \beta^{-m_e+l+1}$. Dans ce cas, nous devons recommencer le calcul de X_f avec une valeur de X_k plus importante de sorte que $\widetilde{X}$ soit plus proche de M.

2.1.3.2 Construction de l'exposant X_k

Si X est petit devant M, σ est trop petit pour être calculé avec précision. Nous devons alors utiliser $X \times \beta^{X_k}$ qui soit plus proche de M afin d'obtenir les premiers chiffres significatifs de X. Etant donné que l'ordre de grandeur de X est inconnu, son estimation se fait pas à pas selon la méthode suivante. On multiplie X par $\beta^{m_e - l}$ en RNS et on tente de calculer σ. Le processus est réitéré tant que la valeur de σ n'est pas significative. Afin de suivre l'augmentation de l'ordre de grandeur de X, X_k est augmenté de m_e à chaque étape. Au final, l'ordre de grandeur de X est connu et nous obtenons une valeur significative pour σ.

Quoiqu'il en soit, même si $\sigma \neq 0$, si σ est petit devant β^{m_e}, il peut être avantageux de multiplier X par β^i en RNS où i est tel que $\beta^{m_e-2} \leq \sigma \times \beta^i < \beta^{m_e-1}$ et de recommencer le processus pour une étape afin d'obtenir une représentation pseudo-flottante plus précise. Dans le pire des cas, (lorsque X est proche de 0), la bonne valeur de X_k est déterminée en $\left\lceil \dfrac{\log_\beta M}{m_e - l} \right\rceil$ itérations.

<u>Exemple</u>:

On considère la base suivante: $\{65437, 65447, 65449, 65479, 65497\}$

$M = 1202097201816031191837293$ et $m_e = 16$ en base 2.

$$M_i = \left\{ \begin{array}{ll} M_0 & 18370298177117398289 \\ M_1 & 18367491280211945419 \\ M_2 & 18366930003759128357 \\ M_3 & 18358514971456973867 \\ M_4 & 18353469652289894069 \end{array} \right. \qquad \Omega_i = \left\{ \begin{array}{ll} \Omega_0 & 3062536088 \\ \Omega_1 & 2923664766 \\ \Omega_2 & 2335264193 \\ \Omega_3 & 3837390181 \\ \Omega_4 & 726046657 \end{array} \right.$$

Soit $X = 1282205882000084821$ et $X_k = 0$

$X_{RNS} = \{40715, 64825, 36443, 60989, 5521\}$

Première étape $X_k = 0$

$$\begin{array}{rrl}
 & \overbrace{}^{\text{Partie entière}} & \overbrace{}^{\alpha_i}\overbrace{}^{\text{Erreur}} \\
x_0 \times \Omega_0 : & 111000101100111 \bullet & 1110110000011011\,1110101110001000 \\
x_1 \times \Omega_1 : & 1010110001011111 \bullet & 1001011111001001\,1100111100001110 \\
x_2 \times \Omega_2 : & 100110101100110 \bullet & 1101001110100111\,1011000110011011 \\
x_3 \times \Omega_3 : & 1101010011011011 \bullet & 0101101100000001\,0111011100010001 \\
x_4 \times \Omega_4 : & 1110100101 \bullet & 0100110101101110\,1100100101010001 \\
\hline
\text{Sum} \quad : & 10 \bullet & \underbrace{111111111111\overbrace{0000}^{\text{Err.}}}_{\sigma}
\end{array}$$

La valeur de σ obtenue est supérieure à β^{m_e-1} Suite à la remarque sur la construction de σ, on obtient $\sigma = 0$ pour $X_k = 0$. Cette valeur n'est pas significative. En effet, X est petit devant M. L'exposant X_k devrait donc être plus grand. On peut remarquer que les valeurs $X_f = 65520$ et $X_k = 0$ correspondent à la représentation pseudo-flottante de $1201803721053869074847098$. X_k est augmenté de $m_e - l = 12$ à chaque étape jusqu'à que σ soit significatif.

Deuxième étape $X_k = 12$

$\tilde{X} = X \times 2^{X_k} = 525191529267234742681 6$
$\tilde{X}_{RNS} = \{35164, 4721, 46808, 8559, 17551\}$

$$
\begin{array}{rr}
\tilde{x}_0 \times \Omega_0 : & \overbrace{110000111110001}^{\text{Partie entière}}\bullet\overbrace{1100010011101110}^{\alpha_i}\overbrace{1101001010100000}^{\text{Erreur}} \\
\tilde{x}_1 \times \Omega_1 : & 110010001101\bullet 10101100010101111101100010011110 \\
\tilde{x}_2 \times \Omega_2 : & 110001101101010\bullet 01111110111000001101000011011000 \\
\tilde{x}_3 \times \Omega_3 : & 11101110111110\bullet 00100100001101111111001111001011 \\
\tilde{x}_4 \times \Omega_4 : & 101110010110\bullet 11101100101111100000100011001111 \\
\hline
\text{Sum} \quad : & 11\bullet\underbrace{000000010001\overbrace{0000}^{\text{Err.}}}_{\sigma}
\end{array}
$$

$\sigma = 272$

Bien que la valeur obtenue soit significative, elle n'est pas assez précise. Une dernière étape est effectuée avec un exposant plus précis : $X_k = 18$.

Troisième étape $X_k = 18$

$\tilde{X} = X \times 2^{X_k} = 33612257873103023531 6224$
$X_{RNS} = \{25638, 40356, 50507, 23944, 9815\}$

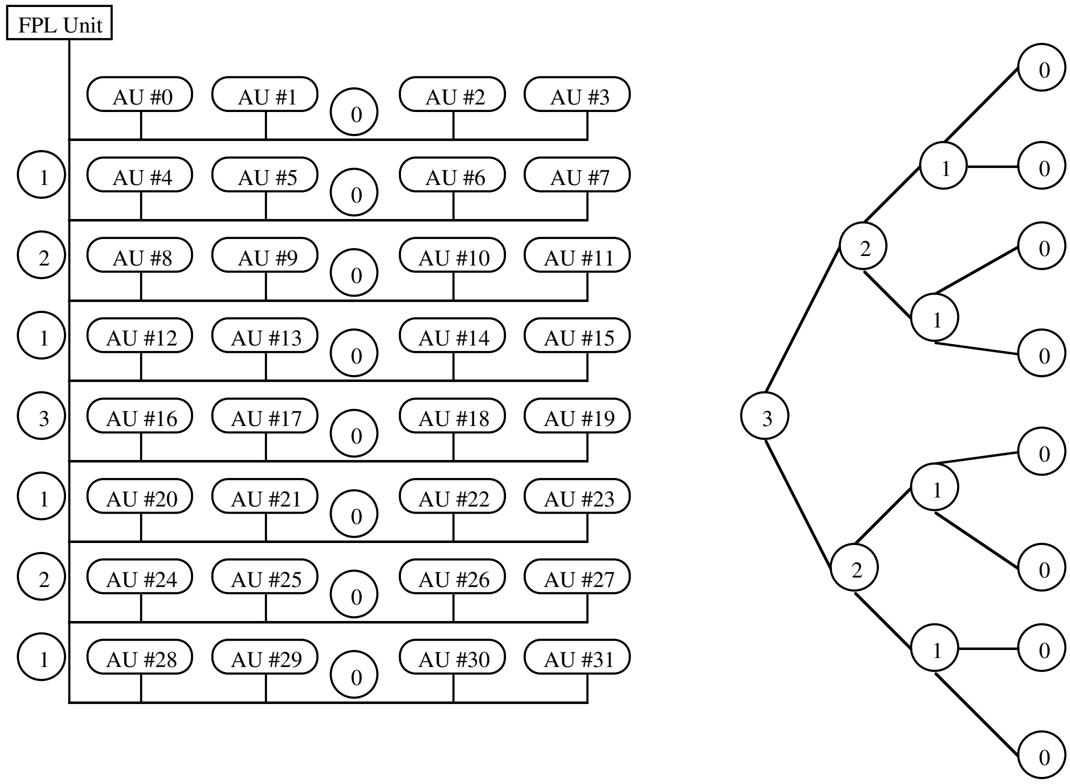
$$
\begin{array}{rr}
\tilde{x}_0 \times \Omega_0 : & \overbrace{100011101101001}^{\text{Pertie entière}}\bullet\overbrace{0011101111001001}^{\alpha_i}\overbrace{1110000010010000}^{\text{Erreur}} \\
\tilde{x}_1 \times \Omega_1 : & 110101101001111\bullet 00010101111110100000101010111000 \\
\tilde{x}_2 \times \Omega_2 : & 110101101000101\bullet 10111000010001110100110010001011 \\
\tilde{x}_3 \times \Omega_3 : & 101001110010001\bullet 00001110000000111100111010101000 \\
\tilde{x}_4 \times \Omega_4 : & 11001111011\bullet 00101111100001000011100010010111 \\
\hline
\text{Sum} \quad : & 1\bullet\underbrace{010001111001\overbrace{0000}^{\text{Err.}}}_{\sigma}
\end{array}
$$

Finalement, on obtient la représentation pseudo-flottante de X : $X_f = 18320$ et $X_k = 18$. On peut remarquer que le premier bit de la mantisse est zéro et que $\sigma < \beta^{m_e - 1}$.

$$\frac{X_f \times M}{\beta^{X_k + m_e}} = 1281873598768707099 \quad \blacksquare$$

2.1.4 Implémentation de la construction

Nous détaillons ici l'implémentation de la construction de la représentation PF. Celle-ci nécessite n unités arithmétiques entières de m_e chiffres (UA) et une unité de calcul pseudo-flottant de m_e chiffres (UPF). Ces unités sont interconnectées sur un bus de m_e chiffres qui est composé de plusieurs sous-bus connectés par des barrières (Fig. 2.1). Cette composition permet de simuler un arbre binaire en programmant judicieusement l'ouverture et la fermeture des barrières connectant les sous-bus. La simulation de cet arbre permet d'effectuer le calcul de σ et de X_f en un temps logarithmique.



Evaluation of σ:

Initialisation
All gates are closed
(no connexion)

Step $= k$
If $k \neq 0$ then
gates $k-1$ open the connexions
AU #$(2n+1)2^k$ sends to AU #$(2n)2^k$
AU #$(2n)2^k$ computes partial σ

FIG. 2.1 – *Architecture de l'implémentation de la conversion en représentation pseudo-flottante*

Les unités arithmétiques entières contiennent leurs propres registres et éléments de mémoires destinés à contenir les constantes m_i, Ω_i et les variables telles que x_i, α_i ou les résultats partiels pour le calcul de σ. Ces unités sont constituées d'un additionneur de m_e chiffres muni d'une partie de contrôle lui permettant d'effectuer des opérations spécifiques pour le calcul de α_i et σ. En effet, la gestion des dépassements de capacité n'est pas utile puisqu'ils correspondent à la partie entière.

Ces unités sont aussi munies d'un additionneur et d'un multiplieur modulaire modulo m_i nécessaires aux calculs en représentation modulaire. Le contrôle de chacune de ces unités est effectué par un simple séquenceur dont les instructions pour les différentes opérations sont stockées dans une mémoire morte. Lors des calculs RNS, ces unités fonctionnement selon le mode SIMD [1] sans aucune communication entre elles.

En résumé, les fonctionnalités minimales des unité UA pour la construction de la

1. Single Instruction, Multiple Data

représentation pseudo-flottante sont :

- 7 registres de m_e chiffres pour le stockage des variables m_i, Ω_i, x_i, α_i et des résultats intermédiaires lors du calcul de σ.
- 1 multiplieur $m_e \times m_e$ capable de calculer la partie de poids fort et la partie de poids faible d'un produit et de placer le résultat dans deux registres de m_e chiffres. Cet opérateur est utilisé lors du calcul de σ
- 1 additionneur de m_e chiffres dont la retenue sortante peut être écartée.

De cette façon nous proposons une architecture des unité UA proche de celle de *Barsi* et *Martinelli* dans [BM91].

Remarque En base 2, les sommes partielles de la multiplication peuvent être facilement obtenues par le décalage d'une des opérandes. Ainsi la multiplication se résume-t-elle à l'addition de ces sommes partielles et peut-elle être simulée avec un simple additionneur. Cette implémentation de la multiplication pourra être choisie dans le cas où la superficie est un paramètre critique.

Afin d'avoir une idée de l'espace requis pour un conversion (qui est de $\mathcal{O}(n \times m_e)$), une implémentation des opérateurs en "Borrow-Save" avec $m_e = 16$ et $n = 64$ (ce qui donne $M \approx 2^{1000}$), utilise environ 50.000 transistors pour la partie arithmétique. Il reste donc assez de place pour implémenter d'autres opérations en hardware, par exemple une multiplication modulaire [AM91].

L'unité de calcul pseudo-flottants contrôle l'ouverture et la fermeture des connections entre les sous-bus. C'est elle qui dirige le calcul de σ.

2.2 Division

La représentation pseudo-flottante permet de connaître rapidement l'ordre de grandeur d'un nombre modulaire. Nous allons utiliser cette représentation dans un algorithme de division euclidienne dérivé d'un algorithme classique de division en grande base. Nous présentons cet algorithme dans la sous-section suivante. Nous analyserons ensuite dans la sous-section 2.2.2 le fonctionnement de cet algorithme. Enfin, une estimation de ses performances sera proposée dans la sous-section 2.2.3.

2.2.1 Algorithme de division en représentation modulaire

On note X_{RNS} la représentation RNS de X : $X_{RNS} = \{x_1, ..., x_n\}$. Nous présentons dans cette section un algorithme de division qui étant donné deux nombres modulaires X_{RNS} et Y_{RNS} détermine le quotient Q_{RNS} et le reste R_{RNS} tels que :

$$X_{RNS} = Q_{RNS} \times Y_{RNS} + R_{RNS}$$

avec $0 \leq R_{RNS} < Y_{RNS}$.

L'algorithme exposé ici est dérivé d'un classique algorithme de division en grande base, où Q et R sont respectivement initialisés à 0 et X. L'itération principale de cet algorithme consiste à calculer l'entier $D = D_m \times \beta^{D_e}$. L'entier D_m peut être vu comme un chiffre du quotient exprimé en grande base. Après chaque étape, D est ajouté au quotient Q et le dividende R est réduit de $D \times Y$ de sorte que $X = Q \times Y + R$. Cette itération est répétée jusqu'à ce que R soit plus petit que le diviseur Y. Au cours de chaque itération, la mise à jour du quotient et de R sont effectués en arithmétique RNS.

Les phases de détermination du chiffre D du quotient ainsi que les comparaison entre R et Y sont effectuées avec la représentation PF des opérandes. Après avoir déterminé les valeurs D_m et D_e, l'unité de calculs pseudo-flottants les diffuse aux unités UA qui construisent la valeur de D_{RNS}. Chaque reste d_i de la représentation modulaire de D est calculé dans l'unité AU correspondante. Ensuite le calcul des nouvelles valeurs de Q_{RNS} et R_{RNS} est effectué en parallèle sur chaque unité AU. Chacune d'entre elles calcule le reste q_i et r_i correspondant. Finalement la dernière comparaison est effectuée avec la représentation "mixed radix" de R et de Y.

L'algorithme suivant calcule la division euclidienne de X par Y en arithmétique modulaire.

Algorithme 2.1

Pré calculs :

Construction of (X_k, X_f) *and* (Y_k, Y_f)

Construction of Y_{MRS}

Initialisation:

$R_{RNS} \leftarrow X_{RNS}$

$(R_k, R_f) \leftarrow (X_k, X_f)$

$Y_{ff} \leftarrow Y_f + \beta^l$

$Q_{RNS} \leftarrow 0$

loop:

while $R_k < Y_k$ *or* $(R_f \geq Y_{ff}$ *and* $R_k = Y_k)$ *(unité PF)*

  if $R_f \neq 0$ **then** *(unité PF)*

    Calcul de (D_m, D_e) *dans l'unité PF*

    L'unité PF diffuse (D_m, D_e) *aux unité UA*

    Les unités UA calculent D_{RNS}

    $R_{RNS} \leftarrow R_{RNS} - Y_{RNS} \times D_{RNS}$ *(unité UA)*

    $Q_{RNS} \leftarrow Q_{RNS} + D_{RNS}$ *(unité UA)*

  if $Y_k - R_k \geq m_e - l - 2$ **then** *(unité PF)*

    $R_k \leftarrow R_k + m_e - l - 2$ *(unité PF)*

  else $R_k \leftarrow Y_k$ *(unité PF)*

  Reconstruction de R_f

end:

if $R_{MRS} \geq Y_{MRS}$ **then**

    $R_{RNS} \leftarrow R_{RNS} - Y_{RNS}$ *(unité UA)*

    $Q_{RNS} \leftarrow Q_{RNS} + 1_{RNS}$ *(unité UA)*

Remarques

- Toutes les opérations en représentation modulaire sont effectuées en parallèle par les unités arithmétiques entières.

- Tous les tests, les calculs sur les ordres de grandeur et les estimations de D, sont effectués par l'unité pseudo-flottante.
- L'ordonnancement des calculs peut être effectué de deux façons. Soit l'unité pseudo-flottante contrôle les opérations effectuées par tous les opérateurs, soit chaque unité ordonnance ses propres calculs à l'aide de son séquenceur.
- Afin d'obtenir D par défaut, nous employons Y_{ff} qui est la majoration de Y_f d'après l'équation (2.3). Par conséquent, à chaque itération, Q est obtenu par défaut et R est toujours positif.
- La reconstruction de la représentation pseudo-flottante de R est déterminée en temps logarithmique selon la méthode présentée dans la section précédente.
- Le dernier test $R_{MRS} \geq Y_{MRS}$ est effectué après la conversion de R_{RNS} avec un des algorithmes présentés dans la section 1.2.2.2. La seconde opérande Y_{MRS} est précalculée avant la division.

Calcul de D_m et D_e dans l'unité pseudo-flottante

L'évaluation des deux entiers D_m et D_e est effectuée par l'unité pseudo-flottante avec l'algorithme suivant qui ne fait intervenir que des opérations entières. Lorsque ce calcul est effectué on a $R_f \neq 0$.

Algorithme 2.2

Calculer R_n *de sorte que* $\beta^{-1} < \dfrac{R_f \times \beta^{R_n}}{Y_{ff}} < \beta$ *(Normalisation)*

If $Y_k - R_k - R_n \geq 0$ **then**

$R_s \leftarrow inf\,(Y_k - R_k - R_n, m_e - 1)$

$D_m \leftarrow \left\lfloor \dfrac{R_f \times \beta^{R_n+R_s}}{Y_{ff}} \right\rfloor$ *(Division des mantisses normalisées)*

$D_e \leftarrow Y_k - R_k - R_n - R_s$ *(Calcul de l'exposant)*

else

$D_m \leftarrow 1$

$D_e \leftarrow 0$

Remarques Tout d'abord, le nombre D_m est un entier de $m_e - 1$ chiffres. Par conséquent $D_m \equiv D_m \bmod m_i$ pour tout $1 \leq i \leq n$. Ensuite D est au moins égal à 1 étant donné que $R > Y$.

Calcul de D modulo m_i dans la $i^{ème}$ unité arithmétique entière

L'unité pseudo-flottante diffuse D_m et D_e à toutes les unités arithmétiques entières (UA). Chacune d'elles calcule d_i de sorte que $D \equiv d_i \bmod m_i$. On remarque que D_e est inférieur à $n \times m_e$. Le reste d_i peut être calculé de différentes façons.

Si $n \times m_e < \beta^{m_e}$, alors D_e peut être représenté par un seul nombre de m_e chiffres. Il n'est pas rare de réunir de telles conditions. Par exemple si $\beta = 2$, un exposant D_e de 16 bits peut être obtenu avec un système modulaire ayant moins de 2^{12} moduli. En outre, le reste de β^{D_e} modulo m_i peut être lu dans une table de lecture si m_e n'est pas trop grand.

Par exemple avec $\beta = 2$ et $m_e = 16$, la taille de la table de lecture est de 2 Ko. Dans ce cas, le calcul de d_i peut être effectué en un seul pas.

Dans le cas contraire, l'exposant D_e peut être décomposé en base β^θ. Nous pouvons alors utiliser $\log_{\beta^\theta}(n \times m_e)$ tables de lecture de taille $\beta^\theta \times m_e$. Le reste de β^{D_e} modulo m_e peut donc être calculé avec au plus $\log_{\beta^\theta}(n \times m_e)$ lectures de table et $\log_{\beta^\theta}(n \times m_e) - 1$ multiplications modulaires. Si β^θ est proche de m_e alors d_i peut être calculé avec au plus $\log_{m_e} n + 1$ lectures de table et $\log_{m_e} n + 1$ multiplications modulaires.

L'exemple suivant illustre le déroulement de l'algorithme.

Exemple :

On considère la base $\mathcal{B}_m^5 = \{65437, 65447, 65449, 65479, 65497\}$ $m_e = 16$ en base 2. Cet exemple illustre la division euclidienne de X par Y en RNS :

$$
\begin{array}{lcrlcr}
X & = & 1282205882000084821 & Y & = & 135 \\
X_f & = & 18320 & Y_f & = & 17376 \\
X_k & = & 18 & Y_k & = & 71 \\
 & & & Y_{ff} & = & 17392
\end{array}
$$

X_f, X_k et Y_f, Y_k sont la représentation de X et Y dans le système pseudo-flottant. La division en grande base est effectuée avec Y_{ff} —la valeur par excès du diviseur— au lieu de Y_f de façon à obtenir à chaque étape un reste positif.

Etape 1

$Y_k - R_k = 53$ et $FD = 1$
$\dfrac{R_f}{Y_{ff}} = 9487804182776850.0$
$D = 9487685836079104$
$Q = 9487685836079104$
$R = 1368294129405781$
$Rk = 28$
$R_f = 20016$

Etape 2

$Y_k - R_k = 43$ et $FD = 1$
$\dfrac{R_f}{Y_{ff}} = 10123194453341.4980$
$D = 10122701045760$
$Q = 9497808537124864$
$R = 1729488228181$
$Rk = 38$
$R_f = 25904$

Etape 3

$Y_k - R_k = 33$ et $FD = 1$
$\dfrac{R_f}{Y_{ff}} = 12794024015.131556$
$D = 12793675776$
$Q = 9497821330800640$
$R = 2341998421$
$Rk = 48$
$R_f = 35936$

Etape 4

$Y_k - R_k = 23$ et $FD = 1$
$\dfrac{R_f}{Y_{ff}} = 17332855.168353$
$D = 17332224$
$Q = 9497821348132864$
$R = 2148181$
$Rk = 58$
$R_f = 33744$

Etape 5

$Y_k - R_k = 13$ et $FD = 1$
$\dfrac{R_f}{Y_{ff}} = 15894.137994$
$D = 15894$
$Q = 9497821348148758$
$R = 2491$
$Rk = 68$
$R_f = 40064$

Etape 6

$Y_k - R_k = 3$ et $FD = 0$
$\dfrac{R_f}{Y_{ff}} = 18.428703$
$D = 18$
$Q = 9497821348148776$
$R = 61$
$Rk = 71$
$R_f = 40064$

En final, nous effectuons la comparaison entre le reste R et le diviseur Y en représentation "mixed radix". On obtient alors $R_{MRS} = 61$ et $Y_{MRS} = 135$ dans la base MRS $\{65437, 65447, 65449, 65479, 65497\}$. Puisque le reste R est strictement plus petit que Y, la dernière étape de correction n'est pas utile. A la sortie de cet algorithme on obtient bien :

$$X = Q \times Y + R$$

avec

$$\begin{array}{rclrcl} X & = & 1282205882000084821 & Y & = & 135 \\ Q & = & 9497821348148776 & R & = & 61 \quad \blacksquare \end{array}$$

2.2.2 Analyse de l'algorithme de division

Cet algorithme s'apparente aux algorithmes de division en grande base du type SRT. En effet, à chaque étape, un nouveau chiffre en base $\beta^{m_e - l - 2}$ du quotient est déterminé. La différence réside dans le fait que le reste R, calculé en représentation modulaire, est converti dans le système pseudo flottant à chaque itération. Le théorème suivant donne une preuve de cet algorithme.

Théorème 2.2 *Si $m_e > l + 2$ alors pour des entiers X_{RNS} et Y_{RNS} donnés, l'algorithme suivant détermine Q_{RNS} et R_{RNS} tels que :*

$$X_{RNS} = Q_{RNS} \times Y_{RNS} + R_{RNS}$$

avec $0 \leq R_{RNS} < Y_{RNS}$

Preuve de la boucle "while". Décroissance de R_e vers Y_e

Soient les notations suivantes :

$$\begin{array}{c} \beta^{X_e} \leq X < \beta^{X_e+1} \\ \beta^{Y_e} \leq Y < \beta^{Y_e+1} \\ \beta^{M_e} \leq M < \beta^{M_e+1} \\ l = \lceil \log_\beta n \rceil + 1 \end{array} \tag{2.10}$$

Nous avons,

$$\frac{R}{M}\beta^{m_e+R_k} - \beta^l < R_f \leq \frac{R}{M}\beta^{m_e+R_k}$$

et,

$$\frac{Y}{M}\beta^{m_e+Y_k} \leq Y_{ff} < \frac{Y}{M}\beta^{m_e+Y_k} + \beta^l$$

Ainsi,

$$\frac{R\beta^{m_e+R_k} - M\beta^l}{Y\beta^{m_e+Y_k} + M\beta^l} < \frac{R_f}{Y_{ff}} \leq \frac{R\beta^{R_k}}{Y\beta^{Y_k}}$$

$$\frac{R - M\beta^{l-m_e-R_k}}{Y + M\beta^{l-m_e-Y_k}} < \frac{R_f}{Y_{ff}}\beta^{Y_k-R_k} \leq \frac{R}{Y} \tag{2.11}$$

puisque $\beta^{M_e} \leq M < \beta^{M_e+1}$ on obtient,

$$\frac{R - \beta^{M_e+l-m_e-R_k}}{Y + \beta^{M_e+l-m_e-Y_k}} < \frac{R_f}{Y_{ff}}\beta^{Y_k-R_k} \leq \frac{R}{Y}$$

et,

$$\frac{R\beta^{M_e-Y_e-1+l-m_e-Y_k} + \beta^{M_e+l-m_e-R_k}}{1 + \beta^{M_e-Y_e-1+l-m_e-Y_k}} > R - \frac{R_f}{Y_{ff}}\beta^{Y_k-R_k} \times Y \geq 0$$

$$\frac{\beta^{R_e+M_e-Y_e-2+l-m_e-Y_k} + \beta^{M_e+l-m_e-R_k}}{1 + \beta^{M_e-Y_e-1+l-m_e-Y_k}} > R - \frac{R_f}{Y_{ff}}\beta^{Y_k-R_k} \times Y \geq 0$$

Comme on a $M_e \geq R_e + R_k + 1$ et si

$$Y_e + Y_k = M_e - 1 \tag{2.12}$$

(cette condition est peut être obtenue avec deux conversions) alors,

$$\frac{\beta^{M_e+l-m_e-R_k} + \beta^{M_e+l-m_e-R_k}}{1 + \beta^{l-m_e}} > R - \frac{R_f}{Y_{ff}}\beta^{Y_k-R_k} \times Y \geq 0$$

Ainsi

$$\beta^{M_e+l-m_e-R_k+1} > R - \frac{R_f}{Y_{ff}}\beta^{Y_k-R_k} \times Y \geq 0 \tag{2.13}$$

Puisque l'on a :

$$\frac{R_f}{Y_{ff}}\beta^{Y_k-R_k} \times (1 - \beta^{-m_e+1}) < D \leq \frac{R_f}{Y_{ff}}\beta^{Y_k-R_k}$$

l'équation 2.13 devient :

$$\beta^{M_e+l-m_e-R_k+1} + \frac{R_f}{Y_{ff}}\beta^{Y_k-R_k-m_e+1} \times Y > R - D \times Y \geq 0$$

$$\beta^{M_e+l-m_e-R_k+1} + R \times \beta^{-m_e+1} > R - D \times Y \geq 0$$

$$\beta^{M_e+l-m_e-R_k+1} + \beta^{R_e-m_e+2} > R - D \times Y \geq 0$$

On obtient :

$$\beta^{M_e+l-m_e-R_k+2} > R - D \times Y \geq 0 \tag{2.14}$$

Il n'y a pas de problèmes de dépassement de capacité dans $R_{RNS} \leftarrow R_{RNS} - Y_{RNS} \times D_{RNS}$ parce que $R - D \times Y \geq 0$. De plus si $R_k + m_e - l - 2 < Y_k$ alors la valeur suivante de R_k peut être $R_k + m_e - l - 2$, sinon $R_k = Y_k$. Donc R_k décroît vers Y_k et l'algorithme converge.

Cas $Y_k = R_k$

Comme il a été vu dans (2.14) et en considérant (2.12), on déduit que l'on a juste avant la dernière itération $R < \beta \times Y$ et $\frac{R}{Y} - 1 < D \leq \frac{R}{Y}$. A présent la dernière itération avec (2.13) donne :

$$\beta^{M_e+l-m_e=R_k+1} > R - \frac{R}{Y} \geq 0$$

et (2.14) se réécrit :

$$\beta^{R_e+l-m_e+2} + Y > R - D \times Y \geq 0$$

En d'autres termes pour $m_e > l+1$:

$$(\beta^{l-m_e+1} + 1) \times Y > R - D \times Y \geq 0$$

$$2 \times Y > R - D \times Y \geq 0 \tag{2.15}$$

Une itération supplémentaire avec une comparaison exacte en "Mixed Radix" est donc nécessaire.

2.2.3 Performances

Evaluons les performances de cet algorithme tant en termes de communications qu'en charge de calculs. Tout d'abord, le calcul de (R_f, R_k) nécessite $\log_2 n$ pas de communications entre unités UA où un entier de m_e chiffres est échangé. Entre chaque pas de communication, une addition sur m_e chiffres est effectuée sur chaque UA. Les valeurs D_m et D_e sont ensuite diffusées de l'unité pseudo-flottante vers les unités UA.

Les tests de la boucle "while" sont effectués par au plus quatre comparaisons de nombres de m_e chiffres dans l'unité pseudo-flottante. La charge de calcul de l'estimation de D_{RNS} se décompose de la façon suivante.

Dans l'unité PF, l'évaluation de R_n est effectuée à l'aide d'au plus m_e décalages et comparaisons sur m_e chiffres. Ensuite le calcul de D_m et D_e est effectué en quatre additions, une multiplication et une division entière sur m_e chiffres. La représentation modulaire de D est calculée par les unité UA en $\log_{m_e} n + 1$ multiplications modulaires de m_e chiffres ou bien $\log_{m_e} n + 1$ lectures de tables.

Ensuite le calcul de la nouvelle valeur de Q_{RNS} et R_{RNS} nécessite deux additions modulaires et une multiplication modulaire de m_e chiffres dans les unités UA. Enfin la reconstruction de R_k demande une comparaison et une addition sur m_e chiffres. Le coût de la reconstruction de la représentation pseudo-flottante de R est détaillé dans la sous-section 2.1.3. Ainsi pouvons-nous en conclure que le coût d'une itération est $\mathcal{O}(\log n)$

Le nombre d'étapes est d'au plus $\frac{M_e}{m_e-l+2}$. En remarquant que $M_e = \mathcal{O}(n)$ et en considérant que m_e est constant, alors le coût de la boucle principale de cet algorithme est $\mathcal{O}(n \log n)$. Finalement le dernier pas est effectué en temps linéaire [Knu81] ce qui justifie le théorème suivant [BDM96] [BDM].

Théorème 2.3 *La complexité en temps de l'algorithme de division en arithmétique modulaire présenté ici est $\mathcal{O}(n \log n)$ et $\mathcal{O}(n)$ en espace.*

2.3 Autres algorithmes de division

La division en arithmétique modulaire a été traitée abondamment dans la littérature scientifique. Cette section présente quelques-unes des solutions proposées et les compare à notre algorithme.

La première division (S.Sect. 2.3.1) est un algorithme sans comparaison présenté par *Gamberger* dont la complexité reste difficile à évaluer correctement. Le second algorithme présenté par *Chiang* et *Lu* est un algorithme qui fait une grande utilisation de la comparaison présentée dans la sous-section 1.3.3.2. Il détermine le quotient de deux entiers par dichotomie (S.Sect. 2.3.2). *Hitz* et *Kaltofen* ont proposé un algorithme de division basé sur l'itération de Newton. Il est décrit dans la sous-section 2.3.3. L'algorithme de la sous-section 2.3.4 dû à *Chren* divise deux nombres modulaires à partir de leur représentation "mixed radix". Enfin nous aborderons les algorithmes de division proposés par *Hung* et *Parhami* (S.Sect. 2.3.5) qui utilisent la comparaison approchée (S.Sect. 1.3.3.4) et sont similaires à l'algorithme SRT [Rob58] [Toc58] [Rob70].

2.3.1 Division de Gamberger

Cet algorithme très habile qui permet de diviser deux nombres sans effectuer de comparaisons a été présenté par *Dragan Gamberger* [Gam91]. Cette sous-section s'articule autour de trois parties. Nous décrirons le principe qui régit cet algorithme, puis nous exposerons son implémentation tel qu'il a été décrit par *Gamberger*. Enfin nous donnerons quelques éléments permettant d'appréhender la difficulté de l'estimation de la complexité de cet algorithme.

2.3.1.1 Approche générale de l'algorithme

Le but de cet algorithme est de trouver l'entier Q tel que :

$$Q = \left\lceil \frac{X}{Y} \right\rceil$$

avec X et Y des entiers.

Dans le cas non trivial où $Y > 1$, l'idée de cet algorithme est de trouver des entiers $X' < X$ et $1 \leq Y' < Y$ tels que :

$$Q = \left\lceil \frac{X}{Y} \right\rceil = \left\lceil \frac{X'}{Y'} \right\rceil$$

En d'autres termes, l'idée de *Gamberger* est d'effectuer des réductions successives des opérandes jusqu'à ce que le diviseur soit égal à 1. Dans ce cas le quotient Q est égal à X'. Cette approche est exploitable dans les systèmes de numération de position mais aussi – ce qui en fait l'intérêt – en RNS. Les réductions se font selon deux itérations :

<u>Itération A :</u>

Si $\mathrm{PGCD}(Y, M) = G > 1$ alors on choisit :

$$X' = \left\lceil \frac{X}{G} \right\rceil$$

$$Y' = \frac{Y}{G}$$

Les valeurs réduites X' et Y' peuvent facilement être obtenues. En remarquant que l'entier G est un produit de certains m_i, la division par G peut être effectuée par la multiplication par l'inverse de G.

Itération B :

Dans le cas où $\mathrm{PGCD}(Y, M) = 1$, c'est à dire si le diviseur est premier avec M, alors les valeur réduites de X et Y peuvent être obtenues de la façon suivante :

$$X' = \left\lceil \frac{XY_M^{-1}}{M} \right\rceil$$

$$Y' = \left\lceil \frac{YY_M^{-1}}{M} \right\rceil = \frac{YY_M^{-1} - 1}{M}$$

où Y_M^{-1} est l'inverse multiplicatif de Y modulo M.

Les itérations A et B ne peuvent pas être implémentées telles qu'elles sont décrites ci-dessus en représentation modulaire. En effet, la réduction par M n'est pas possible dans $\mathcal{B}_m^n$ puisque M_M^{-1} n'existe pas dans cette base. *Gamberger* propose de réécrire les itérations A et B de la façon suivante.

Itération A :

Si $\mathrm{PGCD}(Y, P) = G > 1$ alors

$$X' = \frac{X \times P/G - (X \bmod G) \times P/G}{P} \tag{2.16}$$

$$Y' = \frac{Y \times P/G}{P} \tag{2.17}$$

Itération B :

Si $\mathrm{PGCD}(Y, P) = 1$ alors

$$X' = \frac{XY_P^{-1} - (XY_P^{-1}) \bmod P}{P} \tag{2.18}$$

$$Y' = \frac{YY_P^{-1} - 1}{P} \tag{2.19}$$

avec

$$YY_P^{-1} \bmod P = 1$$

Pour réaliser ces itérations *Gamberger* suggère d'utiliser une base auxiliaire $\mathcal{B}_p^t$ telle que $\prod_{i=1}^{t} p_i = P > M$ et d'effectuer un changement de base. Ainsi la division par P dans $\mathcal{B}_m^n$, si elle est exacte, se résume à la multiplication par l'inverse modulaire de P modulo M.

2.3.1.2 Implémentation

Afin de réduire la quantité de matériel utilisé pour l'exécution de sa division, *Gamberger* propose d'utiliser les mêmes opérateurs pour les itérations A et B. Ainsi l'implémentation de cet algorithme nécessite-t-elle celui d'un changement de base, de la division directe, la multiplication et la soustraction en arithmétique modulaire ainsi que des tables de lecture dont la taille totale est $\mathcal{O}(n \times r)$ [Gam91].

Exemple :

Dans cet exemple, on se propose de diviser $X = 502$ par $Y = 15$ dans le système modulaire principal ayant pour base $\mathcal{B}_m^3 = \{7, 11, 13\}$. On se dote du système modulaire auxiliaire dont la base est $\mathcal{B}_p^3 = \{5, 17, 19\}$. La largeur de l'intervalle de représentation de ces deux systèmes est respectivement $M = 1001$ et $P = 1615$. La représentation modulaire de P dans le système principal est $P_{RNS} = \{5, 9, 3\}$. Son inverse modulaire modulo M est $(P_{RNS})_M^{-1} = \{3, 5, 9\}$. On utilise la notation $D = Y_P^{-1}$ pour l'inverse modulaire de Y modulo P.

Première itération

Base principale $\{7, 11, 13\}$	Base auxiliaire $\{5, 17, 19\}$	Commentaire
$X_{RNS} = \{5, 7, 8\}$ $Y_{RNS} = \{1, 4, 2\}$	$X_{RNS} = \{2, 9, 8\}$ $Y_{RNS} = \{0, 15, 15\}$	 Changement de base
  $X_{RNS} \bmod G = \{2, 2, 2\}$	$G = PGCD(Y_{RNS}, P) = 5$ $X_{RNS} \bmod G = \{2, ., .\}$ $X_{RNS} \bmod G = \{2, 2, 2\}$	Itération A Changement de base
$P_{RNS} \times (G_{RNS})_M^{-1} = \{1, 4, 11\}$ $X'_{RNS} = \{2, 1, 9\}$ $Y'_{RNS} = \{3, 3, 3\}$		 $X' = 100$ $Y' = 3$

Suite à cette première itération on a :

$$Z = \left\lceil \frac{X}{Y} \right\rceil = \left\lceil \frac{502}{15} \right\rceil = \left\lceil \frac{\{5, 7, 8\}}{\{1, 4, 2\}} \right\rceil = \left\lceil \frac{X'}{Y'} \right\rceil = \left\lceil \frac{\{2, 1, 9\}}{\{3, 3, 3\}} \right\rceil = \left\lceil \frac{100}{3} \right\rceil$$

Deuxième itération

Base principale $\{7, 11, 13\}$	Base auxiliaire $\{5, 17, 19\}$	Commentaire
$X_{RNS} = \{2, 1, 9\}$ $Y_{RNS} = \{3, 3, 3\}$	$X_{RNS} = \{0, 15, 5\}$ $Y_{RNS} = \{3, 3, 3\}$	 Changement de base
 $D_{RNS} = \{6, 10, 11\}$ $\|X_{RNS} \times D_{RNS}\|_P = \{4, 10, 5\}$ $X_{RNS} \times D_{RNS} = \{5, 10, 8\}$ $X'_{RNS} = \{3, 0, 1\}$ $Y'_{RNS} = \{2, 2, 2\}$	$G = PGCD(Y_{RNS}, P) = 1$ $D_{RNS} = \{2, 6, 13\}$ $\|X_{RNS} \times D_{RNS}\|_P = \{0, 5, 8\}$	Itération B Changement de base $X' = 66$ $Y' = 2$

A la fin de la seconde itération itération nous obtenons :

$$Z = \left\lceil \frac{X}{Y} \right\rceil = \left\lceil \frac{502}{15} \right\rceil = \left\lceil \frac{100}{3} \right\rceil = \left\lceil \frac{\{2, 1, 9\}}{\{3, 3, 3\}} \right\rceil = \left\lceil \frac{\{3, 0, 1\}}{\{2, 2, 2\}} \right\rceil = \left\lceil \frac{66}{2} \right\rceil$$

Troisième itération

Base principale $\{7,11,13\}$	Base auxiliaire $\{5,17,19\}$	Commentaire
$X_{RNS}=\{3,0,1\}$ $Y_{RNS}=\{2,2,2\}$	$X_{RNS}=\{1,15,9\}$ $Y_{RNS}=\{2,2,2\}$	 Extension de base
 $D_{RNS}=\{3,5,2\}$ $\lvert X_{RNS}\times D_{RNS}\rvert_P=\{5,0,7\}$ $X_{RNS}\times D_{RNS}=\{2,0,2\}$ $X'_{RNS}=\{5,0,7\}$ $Y'_{RNS}=\{1,1,1\}$	$G=PGCD(Y_{RNS},P)=1$ $D_{RNS}=\{3,9,10\}$ $\lvert X_{RNS}\times D_{RNS}\rvert_P=\{3,16,14\}$	Itération B Extension de base $X'=33$

Cette itération est la dernière puisque le diviseur est égal à 1. Suite à quoi nous obtenons :

$$Z=\left\lceil\frac{X}{Y}\right\rceil=\left\lceil\frac{502}{15}\right\rceil=33 \quad \blacksquare$$

2.3.1.3 Performances

Les performances de la division de *Gamberger* sont difficiles à déterminer. Contrairement aux algorithmes de division classiques, la complexité en temps dépend non pas du rapport entre le dividende et le diviseur mais uniquement de l'ordre de grandeur du diviseur. En effet cet algorithme réduit le diviseur à chaque itération jusqu'à ce qu'il soit égal à 1. Plus celui-ci sera grand, plus longue sera la division.

Le pire cas de cet algorithme de division est difficile à déterminer. Afin d'observer le comportement de cet algorithme nous l'avons fait fonctionner dans base modulaire $\{67,71,73,79\}$ sur l'intervalle $[0,27433]$ et nous avons récupéré le nombre d'itérations effectuées en fonction du diviseur. Il apparaît que, dans cet intervalle, le pire cas correspond à $Y=24316$ où la division est effectuée en 16 itérations. En comparaison notre algorithme effectue, dans les mêmes conditions et dans le pire des cas, quatre itérations.

D'autre part, en remarquant que chacune des itérations décrite par *Gamberger* [Gam91] nécessite au moins un changement de base dont la complexité en temps est $\mathcal{O}(n)$ et en espace $\mathcal{O}(n^2)$. En comparaison chaque itération de notre algorithme est déterminée avec une complexité en temps de $\mathcal{O}(\log n)$ et en espace de $\mathcal{O}(n)$.

En conclusion, bien que sans comparaisons, cet algorithme est lourd à mettre en oeuvre et n'offre pas de très bonnes performances [Sal93]. Son coût est supérieur à celui de l'algorithme que nous proposons.

2.3.2 Division de Chiang et Lu

L'algorithme proposé par *Chiang* et *Lu* [CL91] [CL92] s'appuie sur la comparaison de la sous-section 1.3.3.2. Il permet de déterminer le quotient de X par Y en peu d'étapes. Toutefois son implémentation est pénalisée par la comparaison utilisée qui demande de très grandes tables de lecture (S-Sect. 1.3.3.2). La sous-section suivante donne une brève description de leur algorithme de division.

2.3.2.1 Présentation de l'algorithme

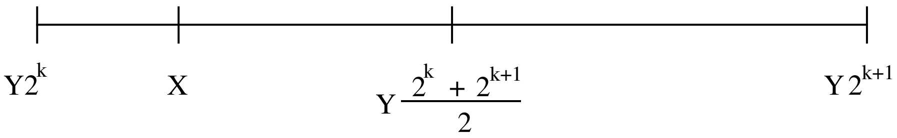
Le principe de cet algorithme est de donner un encadrement de X en fonction de Y et de l'affiner par itérations successives jusqu'à ce que :

$$Y \times Q \leq X < Y \times (Q+1)$$

où Q est le quotient de la division euclidienne de X par Y. L'encadrement initial permet d'encadrer le dividende X en fonction du diviseur Y et de deux puissances de 2 successives. En d'autres termes, elle détermine k de sorte que :

$$Y \times 2^k \leq X < Y \times 2^{k+1}$$

Cette opération est effectuée en k étapes constituées d'une comparaison entre X et $Y \times 2^i$. Noter que $k+1$ est le nombre de bits du quotient. Cet encadrement est ensuite affiné par dichotomie dont la figure suivante illustre la première étape pour un X donné.



$$Y \times 2^k \leq X < Y \times \frac{2^k + 2^{k+1}}{2}$$

FIG. 2.2 – *Une première étape de l'encadrement par dichotomie de X*

Après chaque itération, X est encadré par deux entiers. La dichotomie se poursuit jusqu'à ce que X soit encadré par deux entiers successifs. Cette réduction de l'encadrement de X est effectuée en k itérations où une comparaison entre deux nombres modulaires est déterminée à chaque fois.

Exemple

On se propose dans cet exemple d'effectuer la division entre $X = 783$ et $Y = 10$ avec l'algorithme de *Chiang* et *Lu*.

Une première étape consiste à déterminer l'ordre de grandeur du quotient Q. Celui-ci est obtenu en 6 étapes où une comparaison est déterminée à chaque fois. On a :

$$Y \times 2^6 \leq X < Y \times 2^7$$

Par conséquent $k = 6$. Dans la seconde étape nous déterminons par dichotomie la valeur du quotient qui est comprise dans l'intervalle $[64, 128[$. Cette recherche s'effectue en six itérations. On note BI_i et BS_i les bornes inférieures et supérieures de l'encadrement de X à l'étape i. On a $BI_0 = 2^6$ et $BS_0 = 2^7$. Lors de chaque itération, une comparaison est effectuée entre X et $(BS_i + BI_i)/2$. La table suivante présente les différentes valeurs intermédiaires de cette dichotomie.

Itération	BI_i	BS_i	$(BS_i + BI_i)/2$
0	64	128	96
1	64	96	80
2	64	80	72
3	72	80	76
4	76	80	78
5	78	80	79
6	78	79	

Par conséquent le quotient est $Q = 78$ ∎

2.3.2.2 Performances

La détermination du quotient avec l'algorithme de *Chiang* et *Lu* est déterminé en $2k$ itérations. Outre les comparaisons entre deux nombres modulaires, les autres opérations effectuées dans chaque itération le sont en temps constant. En remarquant que $k = \mathcal{O}(n)$ et que la comparaison s'effectue en $\mathcal{O}(\log n)$ nous pouvons en conclure que cette division a une complexité en temps de $\mathcal{O}(n \times \log n)$. Toutefois, étant donné que la comparaison demande de très grandes tables –leur complexité en espace est de $\mathcal{O}(n^2 \log nM)$–, cette division n'est pas adaptée aux grands nombres et est est moins efficace que la division que nous proposons.

Comme l'algorithme de *Chiang* et *Lu* fonctionne en base 2 et que notre algorithme en base β^{m_e-l-2}, comparer ces algorithmes revient à comparer une itération de notre division à $m_e - l - 2$ itérations de celle de *Chiang* et *Lu*. Or, dans la premier algorithme de division sont effectuées une division entière, quatre additions et une multiplication de m_e chiffres, deux additions et une multiplication en représentation modulaire et $\mathcal{O}(m_e)$ comparaisons sur m_e chiffres. En conséquence, notre algorithme est plus performant que celui de *Chiang* et *Lu*.

2.3.3 Division de Hitz et Kaltofen

L'algorithme proposé par *Hitz* et *Kaltofen* [HK95] s'appuie sur la détermination de l'inverse du diviseur à l'aide de la méthode itérative de Newton. Celle-ci permet de trouver la racine α d'une fonction f continûment dérivable. Dans le cas de la division, si cette fonction est :

$$f(x) = \frac{1}{x} - a$$

alors la suite ci-dessous converge vers $1/a$ avec $x_0 \in [0, 2/a[$ [Mul89].

$$x_{n+1} = x_n \times (2 - a \times x_n)$$

2.3.3.1 Présentation de l'algorithme

Hitz et *Kaltofen* se donnent une base modulaire principale $\mathcal{B}_m^n$ et une base auxiliaire $\mathcal{B}_p^n$ de sorte que $\prod_{i=1}^n m_i < \prod_{i=1}^n p_i$. Les opérations modulaires telles que les additions, les multiplications et les soustractions sont effectuées dans ces deux bases. L'algorithme 2.3

prend en argument les opérandes X et Y et retourne le quotient et le reste de la division entière de X par Y dans le couple $(\lceil X/Y \rceil, X \bmod Y)$.

Algorithme 2.3
$Q \leftarrow \dfrac{X \times \mathbf{RECIP}(Y)}{M}$
$R \leftarrow X - Q \times Y$
if $R < Y$ **then**
    return (Q, R)
else
    return $(Q+1, R-Y)$

La fonction **RECIP** calcule l'inverse de l'entier passé en argument avec la méthode de Newton en représentation modulaire. Elle prend un entier Y tel que $1 \leq Y < M$ et renvoie $\lceil M/Y \rceil$.

Algorithme 2.4 (RECIP)
$Z_1 \leftarrow 0$
$Z_2 \leftarrow 2$
while $Z_1 \neq Z_2$
    $Z_1 \leftarrow Z_2$
    $Z_2 \leftarrow \left\lceil \dfrac{Z_1 \times (2M - Y \times Z_1)}{M} \right\rceil$
if $M - Y \times Z_2 < Y$ **then**
    return Z_2
else
    return $Z_2 + 1$

L'implémentation de ces deux algorithmes est brièvement décrite dans la sous-section suivante.

2.3.3.2 Implémentation

Les additions, multiplications et soustractions en représentation modulaire sont effectuées dans le système modulaire principal et auxiliaire. Les tests de l'algorithme 2.3 et 2.4 sont déterminés de deux façons. Le test d'inégalité entre deux nombres s'effectue sur tous les restes. Les comparaisons sont obtenues grâce à un changement de base (S-Sect. 1.3.3.1). Enfin la réduction par M est déterminée avec le système modulaire auxiliaire et nécessite un changement de base (S-Sect. 1.3.5).

<u>Exemple</u>

On se propose de diviser $X = 67$ par $Y = 12$. On se dote d'un système modulaire principal ayant pour base $\mathcal{B}_m^3 = \{3, 7, 13\}$ et pour base auxiliaire $\mathcal{B}_p^3 = \{5, 11, 17\}$. Ainsi, on a $M = 273$ et $P = 935$. L'inverse modulaire de $(M)_P^{-1}$ dans le système auxiliaire est $\{2,5,1\}$. L'algorithme RECIP se déroule comme suit avec $Y = 12$ comme argument :

Itération	Variable	$\mathcal{B}_m^3 = \{3,7,13\}$	$\mathcal{B}_p^3 = \{5,11,17\}$	Valeur décimale
1	Z_1	$\{2,2,2\}$	$\{2,2,2\}$	2
1	Z_2	$\{0,3,3\}$	$\{3,3,3\}$	3
2	Z_2	$\{2,5,5\}$	$\{0,5,5\}$	5
3	Z_2	$\{2,1,8\}$	$\{3,8,8\}$	8
4	Z_2	$\{1,6,0\}$	$\{3,2,13\}$	13
5	Z_2	$\{0,4,3\}$	$\{3,7,1\}$	18
6	Z_2	$\{0,0,8\}$	$\{1,10,4\}$	21
7	Z_2	$\{1,1,9\}$	$\{2,0,5\}$	22

La table suivante illustre le calcul de Z_2 lors de la première itération.

	$\{3,7,13\}$			$\{5,11,17\}$			Remarques
$2M$	0	0	0	1	7	2	
Y	0	5	12	2	1	12	Calcul dans
$Y \times Z_1$	0	3	11	4	2	7	la base $\mathcal{B}_m^3 \cup \mathcal{B}_p^3$
$2M - YZ_1$	0	4	2	2	5	7	
$T_1 = Z_1(2M - YZ_1)$	0	1	4	4	10	7	
$T_2 = Z_1(2M - YZ_1) \bmod M$	0	1	4	0	5	4	Extension de base
$T_1 - T_2$	0	0	0	4	5	3	Dans $\mathcal{B}_m^3 \cup \mathcal{B}_p^3$
$Z_2 = (T_1 - T_2) \times M_P^{-1}$	•	•	•	3	3	3	
Z_2	3	3	3	3	3	3	Extension de base

On vérifie aisément que $M - Y \times Z_2 < Y$ et donc que RECIP(Y)=$\{1,1,9\}$=22. On obtient ensuite ,dans le système modulaire principal, le quotient et le reste :

$$Q_{RNS} = \{2,5,5\} \quad \text{et} \quad R_{RNS} = \{1,0,7\} \quad \blacksquare$$

2.3.3.3 Performances

La boucle principale de l'algorithme **RECIP** effectue $\mathcal{O}(\log(M/Y)) = \mathcal{O}(n)$ itérations où un changement de base est effectué à chaque fois. L'algorithme utilisé par *Hitz* et *Kaltofen* permet d'effectuer ce changement de base en temps logarithmique [HK95]. En conséquence la complexité en temps de cette division est de $\mathcal{O}(n \log n)$.

L'espace important nécessité par ce changement de base font que la complexité de la division de *Hitz* et *Kaltofen* est de $\mathcal{O}(n^2)$. En comparaison, notre algorithme de division (Sect. 2.2) permet la division de deux nombres avec une complexité de $\mathcal{O}(n \log n)$ en temps et $\mathcal{O}(n)$ en espace. L'algorithme [HK95] n'est donc pas très adapté à la division dans des systèmes modulaires ayant de très nombreux moduli et, par conséquent, à la manipulation de très grands nombres contrairement au notre.

2.3.4 Division de Chren

L'algorithme de division proposé par *Chren* [Chr90] est une amélioration de l'algorithme présenté par *Banerji, Cheung* et *Ganesan* dans [BCG81] qui s'appuie sur la représentation "mixed radix" des opérandes.

2.3.4.1 Présentation de l'algorithme

L'idée de ces deux algorithmes est de construire le quotient en représentation modulaire étape par étape à partir des représentations "mixed radix" des opérandes. Nous présentons dans cette sous-section l'algorithme de *Chren*. A chaque itération, l'entier E_{i+1} est déterminé et :

$$X_{i+1} = X_i - E_{i+1} \times Y$$

avec $X_0 = X$. La condition d'arrêt est $X_r = Y$ ou $E_{r+1} = 0$. Lorsqu'elle est remplie, le quotient est :

$$\left\lfloor \frac{X}{Y} \right\rfloor = \sum_{i=0}^{r-1} E_{i+1} + E' \tag{2.20}$$

où

$$\begin{array}{ll} E' = 0 & \text{si } X_r < Y \\ E' = 1 & \text{si } E_{r+1} = 0 \end{array}$$

La somme 2.20 est déterminée en représentation modulaire. L'estimation des valeurs E_{i+1} est déterminée à partir des représentations "mixed radix" tronquées de X_i et de la représentation MRS de Y. Soient $\{y'_l, y'_{l-1}, \cdots, y'_1\}$ et $\{x'_k, x'_{k-1}, \cdots, x'_1\}$ les représentations MRS de Y et X où y'_l et x'_k sont les chiffres de poids fort non nuls. On utilise dans cette division l'approximation par défaut de X notée $\widetilde{X}$:

$$\widetilde{X} = x'_k m_{k-1} \times \ldots \times m_1 = x'_k, 0, \cdots, 0$$

Ainsi, les valeur E_{i+1} sont déterminées de la façon suivante :

$$\begin{array}{lll} E_{i+1} = \left\lfloor \dfrac{x'_k \times \left\lfloor \dfrac{m_l \times \cdots \times m_1}{Y} \right\rfloor}{m_k} \right\rfloor & & \text{si } k = l \\ E_{i+1} = x'_k \times \left\lfloor \dfrac{m_l \times \cdots \times m_1}{Y} \right\rfloor & & \text{si } k = l+1 \\ E_{i+1} = x'_k \times \left\lfloor \dfrac{m_l \times \cdots \times m_1}{Y} \right\rfloor & \times\, m_{l+1} \times \cdots \times m_{k-1} & \text{si } k > l+1 \end{array}$$

2.3.4.2 Implémentation

Cet algorithme fait une large utilisation des conversions vers le système "mixed radix". L'implémentation de cette conversion peut être faite avec l'un des algorithmes présentés dans la sous-section 1.2.2.2. *Chren* propose que les valeurs $\lfloor m_l \times \cdots \times m_1 / Y \rfloor$ soient tabulées, chaque élément de la table étant accessible en fonction de l'écriture MRS de Y.

<u>Exemple</u>

On se propose de déterminer la division de $X = 2200$ par $Y = 20$ dans le système modulaire ayant pour base $\mathcal{B}_m^3 = \{11, 13, 17\}$. La représentation MRS de Y est $\{0, 1, 9\}$ et celle de $X_0 = X$ est $\{15, 5, 0\}$. On en déduit que $l = 2$ et $k = 3$.

Etape 1 La représentation modulaire de $\lfloor m_2 m_1/Y \rfloor = \{7,7,7\}$. Puisque $k = l+1$ on obtient $E_1 = \{3,1,6\}$ et $X_1 = \{15,9,1\}$.

Etape 2 La représentation MRS de X_1 étant $\{0,9,1\}$ on en déduit $k = 2$. Dans ce cas $k = l = 2$. La réduction par m_2 de $x'_2 \times \lfloor m_2 m_1/Y \rfloor$ donne $\{4,4,\bullet\}$. On obtient alors $E_2 = \{4,4,4\}$ après une extension de base pour déterminer $E_2 \bmod m_2$.

Etape 3 La représentation MRS de X_2 est $\{0,1,9\}$. Par conséquent on a $k = l = 2$. On obtient comme à l'étape 2 $E_3 = \{0,0,0\}$. Donc au final, $Q = E_1+E_2+E_3 = \{8,6,0\} = 110$ ■

2.3.4.3 Performances

Cet algorithme a un coût en espace important à cause de la conversion en représentation "mixed radix". Etant donné qu'il se ramène à un algorithme de division en grande base, il effectue un nombre linéaire d'itérations. Par conséquent, selon l'algorithme de conversion choisi, la complexité en temps de cette division est de $\mathcal{O}(n \log n)$ à $\mathcal{O}(n^2)$ pour une complexité en espace de $\mathcal{O}(n^2)$.

2.3.5 Division de Hung et Parhami

La division proposée par Hung et Parhami dans [HP94a] est une adaptation de l'algorithme de division SRT [Rob58] [Toc58] [Rob70] en représentation modulaire. Dans la division SRT, le quotient est déterminé chiffre après chiffre grâce à des comparaisons simples.

Dans l'algorithme de *Hung* et *Parhami*, à chaque itération, le chiffre du quotient est déterminé en fonction du signe du dividende. Le signe d'un nombre est ici déterminé suite à une conversion approchée effectuée en temps logarithmique (S-Sect. 1.3.3.4). Les nouvelles valeurs du dividende et du quotient sont déterminées en représentation modulaire en temps constant à chaque étape. Tout comme dans la division SRT, le nombre d'itérations est linéaire ce qui lui vaut une complexité en temps de $\mathcal{O}(n \log n)$. En remarquant que le signe peut être obtenu en espace linéaire, la complexité en taille est de $\mathcal{O}(n)$.

Division améliorée *Hung* et *Parhami* proposent, suite à l'algorithme précédent, une amélioration de leur division. Le fonctionnement est identique au détail près que le signe du dividende n'est pas toujours déterminé de la même façon. Celui-ci n'est déterminé par une conversion approchée que toutes les $\log n$ itérations. Le reste du temps, il est estimé à partir des choix faits pour le chiffre du quotient [HP94a] en temps constant. Ainsi la complexité en temps de cette division est réduite d'un facteur $\log n$ l'amenant à $\mathcal{O}(n)$. La complexité en espace reste de $\mathcal{O}(n)$.

Remarques Notre algorithme et celui de *Hung* et *Parhami* sont très proches avec la même complexité en espace. Comme l'algorithme [HP94a] est équivalent à un algorithme de division SRT en base 2, le comparer au notre revient à comparer le coût d'une itération de notre algorithme à $m_e - l - 2$ itérations du leur. On peux remarquer que si $m_e - l - 2 > \log n$, au moins une conversion en temps logarithmique est effectuée toutes les $m_e - l - 2$

itérations de *Hung* et *Parhami*. Ainsi dans les mêmes condition nos deux algorithmes effectuent une conversion et leur coût en temps est similaire. Toutefois la comparaison précise de ces deux algorithmes est délicate. Une étude de leur implémentation en FPGA et VLSI, devrait permettre une comparaison plus détaillée de ces deux algorithmes.

2.4 Conclusion

Nous avons proposé un nouvel algorithme de division en arithmétique modulaire dont le principe se rapproche de celui d'un algorithme de division en grande base en numération de position où chaque chiffre est en représentation pseudo-flottante et est obtenu grâce à une conversion approchée. Ses complexités en temps $\mathcal{O}(n \log n)$ et en espace $\mathcal{O}(n)$ sont semblables à celles des divisions en numération de position les plus utilisées (division restaurante, non restaurante, SRT). La représentation modulaire est donc une arithmétique qui permet le calcul de l'addition et de la multiplication plus vite que les notations en numération de position et celui de la division aussi rapidement. Cette avancée permet de relancer l'intérêt de la représentation modulaire pour effectuer des calculs rapides.

Afin de déterminer les performances de notre division dans le cadre du calcul en représentation modulaire, nous avons confronté ses performances théoriques à celles de cinq autres divisions en arithmétique modulaire. La première [Gam91] fonctionne sans comparaisons, mais semble lente et fait intervenir beaucoup de changements de base. Le second algorithme de division [CL91] [CL92] estime le quotient de deux nombres par dichotomie, mais pêche par la taille des tables nécessaires aux comparaisons qui sont fréquement utilisées. Le troisième [HK95] s'appuie sur la méthode itérative de Newton pour effectuer la division. Toutefois le mécanisme de changement de base employé est très volumineux et reste difficile à implémenter pour de grandes bases. La quatrième division [Chr90] ne reste praticable que sur des nombres de petite taille étant donné la grande quantité de conversion effectuées. Enfin, le dernier algorithme [HP94a] effectue des conversions approchées afin d'estimer le signe du dividende.

Ces algorithmes, sauf celui de *Gamberger* dont il est difficile d'estimer la complexité et celui de *Hung* et *Parhami* qui est linéaire, ont une complexité en temps de $\mathcal{O}(n \log n)$. De ceux-ci, le nôtre est le seul à avoir une complexité en espace linéaire, qui le rend adapté au calcul sur de grands nombres. D'autre part, bien que sa complexité ne soit pas établie, il existe au moins un cas pour lequel l'algorithme de *Gamberger* est moins rapide que le notre et il semble même que ce soit toujours le cas. Une étude de cet algorithme et de ses performances permettra de le vérifier.

L'algorithme dont les performances sont les plus proches du notre est celui de *Hung* et *Parhami* où comme dans le notre, un chiffre du quotient est déterminé suite à une conversion approchée. La différence entre ces algorithmes tient à ce que les itérations de *Hung* et *Parhami* plus simples sont également plus nombreuses que les notres. Une étude de l'implémentation de ces algorithmes en FPGA et VLSI permettrait d'évaluer de façon précise leur performances.

Nous envisageons de prolonger ce travail par l'étude du comportement de tous ces algorithmes pour des sytèmes modulaires relativement petits qui sont utilisés pour le traitement du signal, la détermination de la complexité de la division de *Gamberger* et la définition d'une cellule RNS permettant d'additionner, de soustraire, de multiplier et de diviser.

Chapitre 3

Multiplication modulaire RNS

La multiplication modulaire est une opération utilisée la plupart des cryptosystèmes à clef publique tels que l'algorithme de cryptage et de signature RSA [RSA78], le protocole d'échange de clefs Diffie-Hellman [DH76] ou le protocole d'authentification Fiat-Shamir [FS86] [MS88]. La sécurité de ces algorithmes repose sur la taille des données manipulées et sur la conjecture que la décomposition d'un très grand entier en facteurs premiers est un problème “difficile”. Plus elles sont grandes, plus sûr est l'algorithme. Actuellement les données qui sont manipulées dans ces algorithmes sont de 512 à 1024 bits voire, dans les cryptosystèmes les plus “paranoïaques”, de 2048 bits.

Nous avons vu dans les chapitres précédents que la représentation modulaire est efficace pour le calcul sur de grands entiers en arithmétique RNS. Dans ce chapitre nous nous intéresserons à la multiplication modulaire sur de grands entiers pour des applications de cryptographie. Nous présenterons dans la section 3.1 quelques crypto-algorithmes à clef publique faisant usage de la multiplication modulaire puis l'utilité de ces systèmes et sur quoi repose leur sécurité. Nous présenterons dans la section 3.2 différents algorithmes de multiplication modulaire dont l'algorithme de Montgomery qui fait l'objet d'études intensives.

Nous verrons que la représentation modulaire des nombres semble être intéressante dans les applications cryptographiques dont les données sont de très grands nombres. Nous proposons dans la section 3.3 un algorithme de multiplication modulaire que nous avons élaboré [BDK97] et dont les opérandes sont en représentation modulaire.

3.1 Applications cryptographiques

Les applications privilégiées de la multiplication modulaire sont les applications cryptographiques dont certaines ont pour but de permettre l'échange sécurisé de données entre plusieurs personnes. D'autres visent plutôt à permettre l'authentification ou la vérification de l'identité d'un utilisateur auprès d'un autre.

Cette section présente ces deux applications à travers deux algorithmes basés sur la multiplication modulaire. Le premier qui permet le cryptage de messages et la signature de documents est dû à *Rivest*, *Shamir* et *Adleman* [RSA78]. Le second, présenté par *Fiat* et *Shamir* [FS86], est un moyen très sûr de s'identifier auprès de quelqu'un.

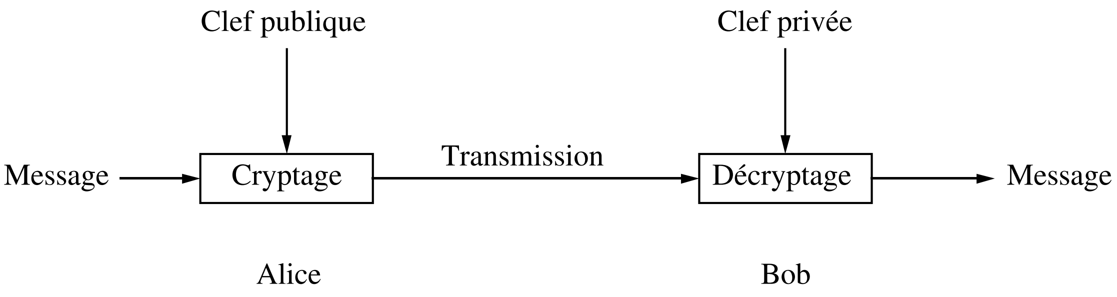


FIG. 3.1 – *Protocole de chiffrage à clef publique*

3.1.1 Algorithme RSA

L'algorithme proposé par *Rivest, Shamir* et *Adleman* [RSA78] est un crypto-algorithme à clef publique. Celui-ci fonctionne de la façon suivante. Lorsqu'un utilisateur (nommons le Bob) désire recevoir d'un autre utilisateur (appelons la Alice) des données confidentielles, il lui envoie sa clef publique. De son côté Alice crypte son message avec la clef que vient de lui envoyer Bob et lui envoie le message chiffré. Enfin, Bob décrypte le message avec sa clef secrète. Dans ce protocole la seule clef qui est communiquée est la clef publique de Bob. Celle-ci a pour seule utilité de chiffrer un message que seul Bob sera capable d'interpréter. La clef secrète de Bob n'est jamais communiquée, la sécurité du message ne peut donc pas être mise en défaut par un tiers.

Cryptage RSA Soit $\mathcal{M}$ le message à crypter. On note P_B la clef publique de Bob et S_B sa clef secrète. On se donne un grand entier noté n qui fait partie de la clef publique. Lorsque Alice souhaite envoyer un message à Bob, elle chiffre le message $\mathcal{M}$ de la façon suivante :

$$\mathcal{C} = \mathcal{M}^{P_B} \bmod n$$

De son côté Bob déchiffre le message crypté comme indiqué ci-dessous :

$$\mathcal{M} = \mathcal{C}^{S_B} \bmod n$$

Signature RSA Le cryptosystème proposé par *Rivest, Shamir* et *Adleman*, outre le cryptage de message, permet aussi la signature de document. La signature a pour but d'authentifier l'émetteur d'un document. Supposons que Bob veuille envoyer un message à Alice et lui prouver qu'il en est bien la source. Il code alors son message avec sa clef secrète,

$$\mathcal{S} = \mathcal{M}^{S_B} \bmod n$$

Alice qui ne peut décoder ce message qu'avec la clef publique de Bob sait ainsi que c'est bien lui qui est l'auteur du message.

$$\mathcal{M} = \mathcal{S}^{P_B} \bmod n$$

Il est clair que, si une personne mal intentionnée veut se faire passer pour Bob, il lui suffit de diffuser une fausse clef publique sous son nom. Il lui est alors possible d'agir sous le nom de Bob. Cet inconvénient peut être contourné par l'utilisation de différentes méthodes dont la description sort du cadre de cette thèse. Voir à ce propos [Sch97].

Calcul des clefs Les clefs sont générées à partir de deux grands nombres premiers p et q que l'on choisit ainsi que de leur produit n. On choisit aléatoirement un grand entier e qui soit premier avec $(p-1)\times(q-1)$. On note d son inverse modulaire modulo $(p-1)\times(q-1)$. Celui-ci peut être déterminé à l'aide de l'algorithme d'Euclide étendu [Knu81]. La clef publique est alors constituée de la paire (e,n). La clef secrète quant a elle est constituée de la paire (d,n).

Le décryptage est valide. En effet puisque d est l'inverse modulaire de e modulo $(p-1)\times(q-1)$, alors il existe un entier k tel que :

$$e\times d = k(p-1)(q-1)+1$$

Par conséquent si $\mathcal{M}\not\equiv 0 \bmod p$, d'après le théorème de Fermat,

$$\begin{array}{rcll} \mathcal{M}^{ed} & \equiv & \mathcal{M}(\mathcal{M}^{p-1})^{k(q-1)} & \mathrm{mod} p \\ & \equiv & \mathcal{M}(1)^{k(q-1)} & \mathrm{mod} p \\ & \equiv & \mathcal{M} & \mathrm{mod} p \end{array}$$

D'autre part, $\mathcal{M}^{ed}\equiv\mathcal{M}$ si $\mathcal{M}\equiv 0 \bmod p$. Donc

$$\mathcal{M}^{ed}\equiv\mathcal{M} \bmod p$$

De la même manière, pour tout $\mathcal{M}$,

$$\mathcal{M}^{ed}\equiv\mathcal{M} \bmod q$$

Par conséquent, puisque $n=pq$ et que p et q sont premiers,

$$\mathcal{M}^{ed}\equiv\mathcal{M} \bmod n$$

La sécurité de l'algorithme RSA repose sur la difficulté qu'une personne mal intentionnée aura pour retrouver la clef privée à partir de la clef publique. En d'autre termes, si cette personne est capable de factoriser n, elle connaîtra p et q et pourra par conséquent en déduire la clef privée. A l'heure actuelle, la factorisation de n est le meilleur moyen de “casser” le crypto-algorithme RSA [Sch97].

Pour que la sécurité de cet algorithme soit bien assurée il est nécessaire de rendre la factorisation de n le plus difficile possible. En effet le problème de la factorisation devient très vite extrêmement difficile pour de très grands entiers [Knu81]. Ainsi cet algorithme n'est sûr que pour des clefs de très grande taille, de l'ordre de 512 à 2048 bits pour les cryptosystèmes les plus “paranoïaques”.

3.1.2 Protocole d'identification de Fiat-Shamir

Lors de l'identification entre deux utilisateurs –nommons les encore Alice et Bob–, Alice peut prouver à Bob qu'elle est bien Alice, mais Bob ne peut pas, au vu des informations que lui a données Alice, se faire passer pour elle auprès de quelqu'un d'autre. Pour cela il est nécessaire de mettre en place un protocole de preuve sans divulgation de l'information. *Amos Fiat* et *Adi Shamir* [FS86] proposent un protocole d'identification.

Alice et Bob conviennent d'un grand entier n produit de deux grands nombres premiers p et q. Alice se donne un nombre v, tel que v soit un résidu quadratique modulo n. En d'autres termes, elle choisit v tel que $x^2 \equiv v \bmod n$ ait une solution et que $v^{-1} \bmod n$ existe. C'est sa clef publique. Ensuite, elle se donne le plus petit entier s tel que $s \equiv \sqrt{v^{-1}} \bmod n$. C'est la clef privée. Lorsqu'elle veut s'identifier auprès de Bob, elle utilise le protocole suivant :

- Alice choisit un nombre aléatoire r et envoie à Bob $x = r^2 \bmod n$.
- Bob répond à l'envoi d'Alice par l'émission d'un bit aléatoire b
- Alice reçoit ce bit.
  - Si $b = 0$ elle renvoie $y = r$
  - Si $b = 1$ elle retourne $y = r \times s \bmod n$
- Bob vérifie alors que
  - Si $b = 0$ alors $x \equiv r^2 \bmod n$, ce qui prouve qu'Alice connaît $\sqrt{x}$
  - Si $b = 1$ alors $x \equiv y^2 \times v \bmod n$, ce qui prouve qu'Alice connaît $\sqrt{v^{-1}}$

Ce protocole permet à Alice de prouver à Bob qu'elle connaît la clef privée s qui correspond à sa clef publique v, sans pour autant la dévoiler à Bob. Ce protocole peut être répété jusqu'à ce que Bob soit convaincu qu'Alice connaît v.

Patricia ne peut pas duper Bob indéfiniment. En effet, si elle ne connaît pas s, elle peut choisir r de manière à tromper Bob soit dans le cas où il lui envoie un 0, soit dans le cas où il lui envoie un 1, mais elle ne peut pas faire les deux en même temps. Ses chances de berner Bob sont donc de 50%. Ses chances de le tromper t fois sont d'une contre 2^t.

3.2 Multiplication modulaire

L'opération de multiplication modulaire consiste à déterminer l'entier C tel que :

$$C = A \times B \bmod N$$

où A, B et C sont des entiers. Pour effectuer ce calcul plusieurs méthodes sont possibles. Une d'elles consiste à effectuer la multiplication $A \times B$ puis la réduction par l'opération modulo. Cette méthode n'est intéressante que si l'on a la possibilité de stocker le résultat intermédiaire $A \times B$ qui peut comporter jusqu'à deux fois plus de chiffres que A ou B. Bien que beaucoup plus coûteuse en place elle offre de bon résultats [VVDJ90]. Toutefois elle est écartée pour toute application où l'espace est critique telle que les implémentations sur carte à puce [Tak93] [Tak95]. Nous ne traiterons pas cette première approche dans ce qui suit.

Une autre méthode consiste à intercaler une étape de réduction modulo N entre chaque étape de la multiplication par additions et décalages [Wal97] [Kor93b] [MY93] [Mon85].

Le gain d'espace de cette méthode pour la multiplication modulaire est évident. Tous les résultats intermédiaires sont de l'ordre de grandeur de N. Toutefois, le nombre d'opérations modulaires est plus important que dans le cas d'une division effectuée en final [Tak95]. Les deux approches de la multiplication modulaire que nous abordons dans cette section sont de ce type. La sous-section 3.2.1 présente un algorithme simple de multiplication modulaire par additions et décalages [Kor93b]. La sous-section suivante traite de l'algorithme de multiplication modulaire de *Montgomery* qui est jusqu'à présent l'algorithme le plus utilisé pour effectuer des multiplications modulaires sur de grands entiers.

3.2.1 Multiplication modulaire par accumulations et divisions

L'algorithme de multiplication par additions et décalages [Mul89] est un algorithme simple qui peut facilement être implémenté pour multiplier deux nombres. La multiplication de deux nombres A et B se décompose en deux phases. La première consiste à générer les produits partiels constitués du produit d'une opérande par le chiffre de l'autre et d'un décalage approprié. La seconde phase consiste en la somme des n produits partiels. On considérera ici que la somme est calculée au fur et à mesure de la production des produits partiels. Ceux-ci sont additionnés au sein d'un accumulateur. On appelle une telle addition une phase d'accumulation.

L'idée maîtresse de la multiplication modulaire par accumulations et réductions successives est d'adapter l'algorithme de multiplication décrit ci-dessus avec des opérations modulaires. Les réductions modulo n sont effectuées lors de chaque accumulation. Ainsi, on intercale entre les phases d'accumulation de la multiplication des phases de réduction où le produit partiel est ajouté à l'accumulateur modulo N. Lorsqu'un produit partiel P_i est calculé, il est ajouté à la valeur accumulée A_{cc}. La réduction modulo N est effectuée en retranchant N à la valeur accumulée autant de fois que nécessaire pour obtenir la valeur du nombre modulo N. Ceci se fait par le calcul de l'entier q_i tel que $A_{cc} = q_i N + R$ avec $R < N$. Par conséquent :

$$A_{cc} \bmod N = A_{cc} - q_i \times N \tag{3.1}$$

Au cours de la multiplication modulaire, ce calcul doit être effectué n fois pour des opérandes de n chiffres.

La rapidité de l'exécution de cette multiplication modulaire tient à deux choses. Premièrement, la détermination du chiffre q_i doit être la plus rapide possible. Plutôt que de l'obtenir grâce à une division, *Peter Kornerup* préfère utiliser une table de lecture. Pour un N donné, les premiers chiffres de Acc sont suffisants pour déterminer q_i [Kor93b].

Plutôt que de tabuler les valeurs de q_i, une autre approche est d'en calculer une valeur approchée par le produit des premiers chiffres significatifs de S par une valeur approchée précalculée de N. Suite à une étude détaillée du cas k=r=2 *Takagi* a établi que les 8 premiers chiffres de S sont suffisant ce qui équivaut à une table de taille 256 [Tak92]. Une méthode similaire a été présentée par *H. Orup* et *P. Kornerup* dans [OK91].

Le calcul efficace de la multiplication modulaire tient ensuite à la vitesse avec laquelle l'équation 3.1 est calculée. Celle-ci peut être déterminée en temps constant avec une arithmétique redondante [Kor93b]. *P. Kornerup* propose en outre d'utiliser une grande base pour ce calcul.

Remarque : Cette approche de la multiplication modulaire nécessite d'une part d'effectuer une division pour obtenir le chiffre q_i et, d'autre part, de manipuler des nombres exprimés en numération de position afin d'obtenir les produits partiels. Une autre méthode due à *Peter Montgomery* permet de substituer des décalages à ces divisions. Nous la présentons dans la sous-section suivante.

3.2.2 Multiplication modulaire de Montgomery

La principale difficulté des algorithmes de la sous-section précédente est la détermination du quotient q_i. *Montgomery* [Mon85] présente un algorithme de multiplication modulaire dont la phase de réduction ne nécessite pas de division mais seulement des décalages. Lors de chaque phase de réduction, il propose de déterminer non pas $U \leftarrow R \bmod N$ mais $U \leftarrow R \times \beta^{-1} \bmod N$ de sorte que la division entière par β et la réduction modulo β soit aisée ce qui revient à déterminer un entier k tel que :

$$R = k \times N + U \times \beta$$

avec $R < \beta N$.

En d'autres termes on cherche $q = -k$ tel que :

$$U = \frac{R + q \times N}{\beta}$$

Or on veut que U soit un entier. Par conséquent, on doit déterminer q de telle sorte que $R + q \times N$ soit multiple de β. Donc :

$$q \times N \equiv -R \bmod \beta$$

et si β est premier avec N, alors :

$$q = R \times (-N)_\beta^{-1} \bmod \beta$$

En définitive, l'algorithme suivant calcule $U \equiv R\beta^{-1} \bmod N$.

Algorithme 3.1

$q \leftarrow (R \bmod \beta)(-N)_\beta^{-1} \bmod \beta$

$U \leftarrow \dfrac{(R + q \times N)}{\beta}$

Si $U \geq N$ ***alors***

  rendre $U - N$

Sinon

  rendre U

En remarquant que $0 \leq R + q \times N < \beta \times N + \beta \times N$, on a $0 \leq U < 2 \times N$ avant la correction finale. Cet algorithme calcule donc bien $U \equiv R\beta^{-1} \bmod N$ [Mon85].

En résumé, l'idée que présente *Montgomery* consiste à simplifier la réduction modulo N par des multiplications et des réduction modulo β, où β est choisi de telle sorte que ces opérations soient simples. En règle générale β est une puissance de la base dans laquelle sont exprimées les opérandes. Ainsi les multiplications et les réductions modulo β se réduisent à de simples décalages de chiffres à gauche et à la récupération des chiffres les moins significatifs.

Tout comme dans la sous-section précédente, les phases de réduction peuvent être insérées entre les phases d'accumulations. Soient A, B et N des entiers codés sur n chiffres en base β. L'algorithme suivant calcule le produit modulaire $R \equiv A \times B \times \beta^{-n} \bmod N$. On note :

$$A = \sum_{i=0}^{n-1} a_i \times \beta^i$$
$$B = \sum_{i=0}^{n-1} b_i \times \beta^i$$
$$C = \sum_{i=0}^{n-1} c_i \times \beta^i$$

Algorithme 3.2

$R \longleftarrow 0$

For $i = 0$ *to* $n - 1$

(1) $q_i \longleftarrow ((r_0 + a_i b_0)(\beta - n_0)_\beta^{-1}) \bmod \beta$

(2) $R \longleftarrow R + a_i B + q_i N$

(3) $R \longleftarrow R \ div\ \beta$

L'algorithme 3.2 détermine un produit modulaire en $\mathcal{O}(n)$ étapes au cours desquelles sont effectuées une addition et une multiplication modulaire sur un chiffre pour la détermination de q_i et deux multiplications d'un nombre de n chiffres par un chiffre, deux additions de deux nombres de n chiffres et un décalage. Si on considère que les opérations sur R sont effectuées en temps constant avec une représentation redondante, la complexité de cet algorithme est $\mathcal{O}(n)$ en temps et $\mathcal{O}(n)$ en espace [Kor93b].

Remarque La multiplication modulaire de Montgomery donne un résultat comportant un facteur supplémentaire β^{-n}. Celui-ci peut être éliminé par une multiplication modulaire supplémentaire par β^{2n} avec le même algorithme.

Exemple :

On se propose ici de multiplier $A = 831$ par $B = 147$ modulo $N = 89$ avec l'algorithme de Montgomery. On prend $\beta = 10$. Ainsi $(\beta - n_0)_\beta^{-1} = 1$.

Itération 1

$$\begin{aligned} q_0 &= (0 + 1 \times 7) \times 1 \bmod 10 \\ &= 7 \\ R &= 0 + 1 \times 147 + 3 \times 89 \text{ div } 10 \\ &= 77 \end{aligned}$$

Itération 2

$$\begin{aligned} q_1 &= (7 + 3 \times 7) \times 1 \bmod 10 \\ &= 8 \\ R &= 77 + 3 \times 147 + 8 \times 89 \text{ div } 10 \\ &= 123 \end{aligned}$$

Itération 3

$$\begin{aligned} q_2 &= (3 + 8 \times 7) \times 1 \bmod 10 \\ &= 9 \\ R &= 123 + 8 \times 147 + 9 \times 89 \text{ div } 10 \\ &= 210 \end{aligned}$$

On vérifie bien que $831 \times 147 \bmod 89 = 210 \times 10^3 \bmod 89 = 49$ ■

En résumé l'algorithme de *Montgomery* permet de calculer le produit modulaire de deux nombres à l'aide d'opérations arithmétiques et logiques très simples ce qui en fait un des plus rapides algorithmes de multiplication modulaire. Particulièrement adapté au calcul sur de grands entiers, il est l'indispensable élément de toute architecture touchant au domaine de la cryptographie. Une abondante littérature traite de son application aux cryptosystèmes [JB97] [KAK96] [YJC96] [CHW96] [SV93] [Wal93] [Kor93b] [EW93].

3.3 Multiplication modulaire RNS

Certaines applications cryptographiques à clefs publiques font usage de la multiplication modulaire avec de grands entiers. Or nous disposons d'une part avec l'algorithme de Montgomery d'un très bon algorithme de multiplication modulaire et d'autre part de la représentation modulaire des nombres qui est est particulièrement adaptée aux calculs sur de grands nombres (Sect. 1.3). Nous avons donc cherché à concilier les avantages de l'algorithme de Montgomery et ceux de la représentation modulaire pour créer un algorithme de multiplication modulaire en représentation modulaire. Cette section présente l'analyse de l'algorithme de Montgomery et son adaptation aux systèmes modulaires.

Nous présenterons l'algorithme de multiplication modulaire en représentation modulaire dans la sous-section 3.3.1. La preuve de cet algorithme est détaillée dans la sous-section 3.3.2. Enfin dans la sous-section 3.3.3 nous justifions l'utilisation de la représentation modulaire pour un usage en cryptographie.

3.3.1 Présentation de l'algorithme

L'algorithme de Montgomery est lié à la représentation en numération de position de ses opérandes. Nous discutons ici de l'adaptation de cet algorithme à la représentation modulaire.

3.3.1.1 Remarques à propos de l'algorithme de Montgomery

L'algorithme de Montgomery permet de multiplier deux nombres de n chiffres A et B modulo N en numération de position. Il se déroule en n itérations, chacune étant décomposée en trois parties (Algo 3.2). Tout d'abord, à la ligne (1), un chiffre q_i est calculé avec le chiffre le moins significatif de B et de C ainsi que le $i^{ème}$ chiffre de A. On note $Q = \sum_{i=0}^{n-1} q_i \beta^i$. Etant donné que la représentation modulaire n'est pas une représentation en numération de position, l'obtention de ces chiffres est délicate. Nous proposons, dans notre adaptation de cet algorithme d'utiliser, l'opérande A dans sa représentation "mixed radix".

Ensuite, à la ligne (2), la nouvelle valeur du reste R est déterminée de sorte que R soit un multiple de la base β. Cette propriété permet de simplifier la réduction de la ligne (3), qui se résume à un décalage à droite.

En représentation modulaire, le calcul de la ligne (2) ne pose pas de difficultés et peut être effectué pourvu que R, B et N soient en RNS. D'autre part, la division effectuée à la ligne (3) étant exacte, nous sommes dans le cas particulier de la division exacte (S-Sect. 1.3.4). Ce calcul peut donc être déterminé rapidement.

3.3.1.2 L'algorithme

On se place dans le système modulaire ayant pour base $\mathcal{B}_m^n = \{m_1, \cdots, m_n\}$. On se donne un système modulaire auxiliaire ayant pour base $\mathcal{B}_{\tilde{m}}^n = \{\tilde{m}_1, \cdots, \tilde{m}_n\}$ de sorte que PGCD($m_i, \tilde{m}_j$)=1 pour tout $1 \leq i \leq n$ et $1 \leq i \leq n$, et que $\widetilde{M} = \prod_{i=1}^n \tilde{m}_i > M$.

Soient A, B et N trois entiers dont la représentation modulaire dans la base $\mathcal{B}_m^n$ et $\mathcal{B}_{\tilde{m}}^n$ est la suivante :

$$\begin{aligned} A &= \{a_1, \cdots, a_n, \tilde{a}_1, \cdots, \tilde{a}_n, \} \\ B &= \{b_1, \cdots, b_n, \tilde{b}_1, \cdots, \tilde{b}_n\} \\ N &= \{n_1, \cdots, n_n, \tilde{n}_1, \cdots, \tilde{n}_n\} \end{aligned}$$

Soient $A_{MRS} = \{a'_n, \cdots, a'_1\}$ et $Q_{MRS} = \{q'_n, \cdots, q'_1\}$ la représentation en "mixed radix" de l'opérande A et celle d'un entier Q. L'algorithme suivant permet de calculer $ABM^{-1} \bmod N$ en représentation modulaire dans le système $\mathcal{B}_{\tilde{m}}^n$.

Algorithme 3.3

$R \longleftarrow_{RNS} 0$

For $i = 0$ *to* $n-1$ *do*

(1) $q'_i \longleftarrow (r_i + a'_i \times b_i)(m_i - n_i)_i^{-1} \bmod m_i$

(2) $R \longleftarrow_{RNS} R +_{RNS} a'_i \times_{RNS} B +_{RNS} q'_i \times_{RNS} N$

(3) $R \longleftarrow_{RNS} R \div_{RNS} m_i$

Les opérations de la ligne (2) et (3) sont effectuées en représentation modulaire. La division de la ligne (3) est une division exacte. Elle est obtenue grâce à la multiplication par l'inverse modulaire de m_i (S-Sect. 1.3.4). L'analyse de cet algorithme est détaillée ci-dessous.

3.3.1.3 Analyse de l'algorithme

L'algorithme que nous proposons a une structure similaire à celui de Montgomery. Comme dans celui-ci (Algo. 3.2), nous calculons à la ligne (1) le $i^{ème}$ chiffre d'un nombre Q. La différence est que q_i est un chiffre MRS, la représentation modulaire des nombres n'étant pas une représentation de numération de position. Pour ce faire, nous utilisons l'opérande A exprimée en arithmétique MRS. Cette ligne peut s'écrire :

(1) $q'_i = (R_{MRS} + a'_i \times B_{MRS})(M_{MRS} - N_{MRS})_i^{-1} \bmod m_i$

D'autre part, puisque nous calculons le $i^{ème}$ chiffre MRS de Q, le calcul de la ligne (1) est effectué modulo m_i. Pour cela nous devons utiliser le chiffre le moins significatif de la représentation MRS de B et N. Celui-ci peut facilement être obtenu en observant que chaque reste x_i de l'écriture RNS d'un nombre X est le chiffre de poids le plus faible x'_1 de son écriture MRS dont le premier élément de la base est le modulus m_i correspondant à ce reste (S-Sect. 1.2.2.2).

Exemple :

Soit l'entier $X = 831$. Sa représentation dans le système modulaire ayant pour base $\mathcal{B}_m^n = \{3, 7, 13, 19\}$ est $\{0, 5, 12, 14\}$. Il admet les notations suivantes :

$$\begin{array}{rcll}
 & & & \text{Base MRS} \\
X & = & \mathbf{0} + \mathbf{4} \times 3 + \mathbf{0} \times 3 \times 7 + \mathbf{3} \times 3 \times 7 \times 13 & \{3, 7, 13, 19\} \\
X & = & \mathbf{5} + \mathbf{1} \times 7 + \mathbf{9} \times 7 \times 13 + \mathbf{0} \times 7 \times 13 \times 19 & \{7, 13, 19, 3\} \\
X & = & \mathbf{12} + \mathbf{6} \times 13 + \mathbf{3} \times 13 \times 19 + \mathbf{0} \times 13 \times 19 \times 3 & \{13, 19, 3, 7\} \\
X & = & \mathbf{14} + \mathbf{1} \times 19 + \mathbf{0} \times 19 \times 3 + \mathbf{2} \times 19 \times 3 \times 7 & \{19, 7, 3, 13\} \quad \blacksquare
\end{array}$$

Pour simplifier, nous considérerons que la détermination de q'_i est effectuée dans un système "mixed radix" dont la base subit une permutation circulaire. Tous les moduli sont ainsi successivement associés au chiffre de poids faible.

A la ligne (2), R est calculé à partir de a'_i et q'_i de sorte que R soit multiple de m_i. Ce calcul est effectué en représentation modulaire.

En dernier lieu, à la ligne (3), nous divisons R par m_i. Cette division est une division exacte et se résume à une multiplication par l'inverse modulaire de m_i. Toutefois l'inverse modulaire de m_i n'existe pas pour tous les moduli. En effet m_i est premier avec tous les $m_{j \neq i}$ mais pas avec lui même. Par conséquent, à chaque itération le résultat est exprimé avec un moduli de moins. Cela signifie qu'en fin de compte le résultat ne peut plus être exprimé dans un système modulaire de n moduli au bout de n itérations. Il est alors nécessaire d'effectuer les calculs sur R dans une base auxiliaire $\mathcal{B}_{\tilde{m}}^{n}$ où $M < \widetilde{M}$. La connaissance des restes de B et N dans cette base peut être obtenue par une extension de base effectuée au préalable.

Etant donnés un nombre A exprimé en arithmétique MRS ayant pour base $\mathcal{B}_m^n$, B et N exprimés en représentation modulaire dans $\mathcal{B}_m^n$ et $\mathcal{B}_{\tilde{m}}^{n}$, l'algorithme 3.3 détermine $A \times B \times M_N^{-1} \bmod N$ en représentation modulaire dans $\mathcal{B}_{\tilde{m}}^{n}$.

Exemple :

On se donne une base modulaire principale $\mathcal{B}^n_m = \{3, 7, 13, 19\}$ et une base auxiliaire $\mathcal{B}^n_{\tilde{m}} = \{5, 11, 17, 23\}$. On se propose de déterminer $A \times B \bmod N$ avec $A = 831$, $B = 147$ et $N = 89$.

La représentation MRS de A dans $\mathcal{B}^n_m$ est :

$$A_{MRS} = \{0, 4, 0, 3\}$$

La représentation modulaire de B et N dans $\mathcal{B}^n_m$ et $\mathcal{B}^n_{\tilde{m}}$ est :

$$B_{RNS} = \{0, 0, 4, 14, 2, 4, 11, 9\}$$
$$N_{RNS} = \{2, 5, 11, 13, 4, 1, 4, 20\}$$

Les valeurs intermédiaires sont présentées ci-dessous :

$\mathcal{B}^n_m$				$\mathcal{B}^n_{\tilde{m}}$					
3	7	13	19	5	11	17	23	Calcul	Base MRS
0								$q'_0 = (0 + 0 \times 0) \times 1 \bmod 3$	3,7,13,19,5,11,17,23
0	0	0	0	0	0	0	0	$R = 0 + +0 \times 147 + 0 \times 89$	
•	0	0	0	0	0	0	0	$R = 0 \text{ div } 3 = 0$	
	0							$q'_1 = 0$	7,13,19,5,11,17,23,3
•	0	3	18	3	5	10	13	$R = 0 + 4 \times 147 + 0 \times 89 = 588$	
•	•	6	8	4	7	16	15	$R = 588 \text{ div } 7 = 84$	
		3						$q'_2 = 3$	13,19,5,11,17,23,3,7
•	•	0	9	1	10	11	6	$R = 84 + 0 \times 147 + 3 \times 89 = 351$	
•	•	•	8	2	5	10	4	$R = 351 \text{ div } 13 = 27$	
			2					$q'_3 = 2$	19,5,11,17,23,3,7,13
•	•	•	0	1	8	0	2	$R = 27 + 3 \times 147 + 0 \times 89 = 646$	
•	•	•	•	4	1	0	11	$R = A \times B \times M_N^{-1} \bmod N = 34$	

■

Le résultat est exprimé dans un système modulaire dont la base est différente du système dans lequel sont exprimées les opérandes. Toutefois cela n'est pas gênant. En effet il est nécessaire d'effectuer une autre multiplication modulaire afin d'éliminer le facteur M_N^{-1}. Cette opération se fait depuis le système modulaire auxiliaire vers le système modulaire principal.

Pour ce faire, le produit R est employé à la place de l'opérande A dans l'algorithme 3.3 et la constante $M^2 \bmod N$ à la place de l'opérande B. Il est donc nécessaire d'effectuer une conversion de R dans un système MRS. Cette conversion s'intercale entre les deux multiplications modulaires. La constante $M^2 \bmod N$ est précalculée et ne nécessite aucun calcul supplémentaire. L'exemple suivant illustre l'élimination du facteur $M^{-1} \bmod N$ par une seconde multiplication modulaire.

Exemple :

On se place dans les mêmes systèmes modulaires que dans l'exemple précédent pour illustrer la seconde multiplication modulaire permettant d'éliminer le facteur supplémentaire M_N^{-1}. Pour ce faire nous multiplions le produit $R = 34$ obtenu précédemment à la constante précalculée $M^2 \bmod N$. On a alors :

$$(M^2 \bmod N)_{RNS} = \{2, 2, 0, 8, 0, 10, 14, 19\}$$
$$R_{MRS} = \{11, 1, 0, 0\} \text{ dans la base } \{23, 17, 11, 5\}$$

Les résultats intermédiaires sont présentés dans la tableau ci-dessous.

$\mathcal{B}_m^n$				$\mathcal{B}_{\tilde{m}}^n$					
3	7	13	19	5	11	17	23	Calcul	Base MRS
								$q'_0 = (0 + 11 \times 19) \times 8 \bmod 23$	23,17,11,5,19,13,7,3
0	4	7	11	4	5	14	0	$R = 0 + 11 \times 65 + 16 \times 89 = 2139$	
0	2	2	17	3	5	8	•	$R = 2139 \text{ div } 23 = 93$	
						3		$q'_1 = 3$	17,11,5,19,13,7,3,23
2	5	9	7	0	7	0	•	$R = (93 + 1 \times 65 + 3 \times 89) = 425$	
1	4	12	6	0	3	•	•	$R = 425 \text{ div } 17 = 25$	
					8			$q'_2 = 8$	11,5,19,13,7,3,23,17
2	2	9	15	2	0	•	•	$R = (8 + 0 \times 65 + 8 \times 89) = 737$	
1	4	2	10	2	•	•	•	$R = 737 \text{ div } 11 = 67$	
			2					$q'_3 = 2$	5,19,13,7,3,23,17,11
1	2	11	6	0	•	•	•	$R = (67 + 0 \times 65 + 2 \times 89) = 245$	
2	6	10	5	•	•	•	•	$R = A \times B \bmod N = 49$	

■

On suppose que la conversion de A en représentation MRS est effectuée au préalable et que les opérandes $B < M$ et $N < M$ sont exprimées en RNS dans $\mathcal{B}_m^n$ et $\mathcal{B}_{\tilde{m}}^n$. La multiplication modulaire que nous présentons n itérations. Dans chacune d'entre elle nous effectuons une addition et une multiplication modulaire sur $\lfloor \log_\beta m_i \rfloor + 1$ chiffres pour le calcul de q'_i, deux additions et trois multiplications en représentation modulaire pour le calcul de R. On considère que les opérations en arithmétique RNS sont effectuées en parallèle sur $2n$ processeurs en temps constant. A chaque étape, nous diffusons la valeur de q'_i et de a'_i auprès de tous les processeurs. Si celle-ci s'effectue en temps constant, alors chaque itération est déterminée en temps constant. Ainsi, la complexité en temps de notre algorithme de multiplication modulaire en représentation modulaire est $\mathcal{O}(n)$.

Cette opération est effectuée en parallèle sur $2n$ processeurs, chacun étant dédié au calcul sur un modulus. Ils contiennent la constante m_i ainsi que des registres destinés au stockage de $(m_i - n_i)$, a'_i, r_i et b_i. On en conclut que la complexité en espace de cet algorithme est $\mathcal{O}(n)$. Nous verrons dans la section 3.4 différentes implémentations de cet algorithme.

3.3.2 Preuve de l'algorithme

Le théorème suivant nous certifie la convergence et la validité de l'algorithme 3.3.

Théorème 3.1 *Soient A, B et N trois entiers tels que :*

$$A < \frac{m_{n-1} - 1}{2} \frac{M}{m_{n-1}}$$

$$B < 2N \qquad et \qquad 0 \leq N < \frac{M}{3 \ \sup_{i \in \{0,\ldots,n-1\}}(m_i)}$$

avec N et M premiers entre eux.

Alors, l'algorithme 3.3 calcule l'entier R de sorte que $R \equiv A * B * (M)^{-1} \mod N$ et $R < 2\,N$.

Preuve

On suppose que N et M sont premiers entre eux. A chaque étape de l'algorithme un chiffre $q_i \longleftarrow (r_i + a_i' * b_i)(m_i - n_i)_i^{-1} \mod m_i$ de la représentation "Mixed Radix" du nombre Q est calculé. Ainsi, la nouvelle valeur obtenue $R \longleftarrow R +_{RNS} a_i' *_{RNS} B +_{RNS} q_i' *_{RNS} N$ est un multiple de m_i. Par conséquent, la division par le modulus m_i peut être faite en multipliant chaque reste par l'inverse modulaire de m_i modulo m_j avec $j \neq i$. En déroulant le fonctionnement de l'algorithme on a :

$$R = \cfrac{\cfrac{\cfrac{\cfrac{a_0' * B + q_0' * N}{m_0} + a_1' * B + q_1' * N}{m_1} + \ddots + a_{n-3}' * B + q_{n-3}' * N}{m_{n-3}} + a_{n-2}' * B + q_{n-2}' * N}{m_{n-2}}$$

Donc,

$$\begin{aligned} R &= \frac{1}{M}((a_0' + a_1' m_0 + \ldots + a_{n-1}' m_0 \ldots m_{n-2}) * B + \\ &\quad (q_0' + q_1' m_0 + \ldots + q_{n-1}' m_0 \ldots m_{n-2}) * N) \\ &= \frac{1}{M}(A * B + Q * N) \end{aligned}$$

Comme $B < 2N$ alors à chaque pas $R < 3N$:

$$R \ + \ a_i' \ * \ B \ + \ q_i' * \ N \leq 3\,N + (m_i - 1)\,2\,N + (m_i - 1)\,N \leq 3\,m_i\,N$$

Avec $A < \dfrac{m_n - 1}{2} \times \dfrac{M}{m_n}$ on obtient qu'après chaque étape $R < 2N$:

$$R \ + \ a_{n-1}' \ * \ B \ + \ q_{n-1}' * \ N \leq 3\,N + (\frac{m_{n-1} - 1}{2} - 1)\,2\,N + (m_{n-1} - 1)\,N \leq 2\,m_{n-1}\,N \quad \blacksquare$$

3.3.3 Application

A priori, l'utilisation d'un algorithme de multiplication modulaire en RNS pour des applications cryptographiques nécessite une conversion depuis une représentation de position vers la représentation modulaire. C'est la conception qu'ont eue *Hung* et *Parhami* de l'utilisation de la représentation modulaire pour des applications cryptographiques [HP94b]. Toutefois, concevoir cette représentation uniquement comme une représentation intermédiaire destinée à accélérer les calculs n'est pas satisfaisant.

Plutôt que de d'interprêter un message comme étant un grand entier en représentation binaire, nous pouvons le voir comme un nombre en représentation modulaire. La figure 3.2 illustre le découpage du message en restes de neuf bits selon une base de six moduli. Ce découpage et son utilisation dans des applications cryptographiques n'est possible

$$\text{Binaire}\quad \overbrace{\boxed{101000100100101110101011111101010111010110100010111001}}^{\text{Message}}$$

$$\text{Modulaire}\quad \underbrace{\boxed{101000100}}_{\text{mod } m_1}\underbrace{\boxed{100101110}}_{\text{mod } m_2}\underbrace{\boxed{101011111}}_{\text{mod } m_3}\underbrace{\boxed{101010111}}_{\text{mod } m_4}\underbrace{\boxed{010110100}}_{\text{mod } m_5}\underbrace{\boxed{010111001}}_{\text{mod } m_6}$$

FIG. 3.2 – *Représentation binaire et repésentation modulaire pour la représentation de messages*

que sous certaines conditions. Tout d'abord, les vecteurs de bits issus de la partition du message en petites parties –notons les x_i– doivent pouvoir être considérées comme des restes moduli m_i. C'est à dire qu'il faut que $0 < x_i < m_i$. Un découpage du message en vecteurs de p bits n'est valide que si les moduli ont au moins $p+1$ bits. Sous cette condition, les vecteurs x_i représentent les restes d'un entier X dans le système modulaire qui a pour base $\mathcal{B}_m^n$.

Ensuite, d'après le théorème 3.1, les opérandes A, B et N doivent vérifier les conditions suivantes pour que la multiplication modulaire en représentation modulaire puisse être réalisée :

$$A < \frac{m_n - 1}{2}\frac{M}{m_n}$$

$$B < 2N \qquad \text{et} \qquad 0 \leq N < \frac{M}{3\ \sup_{i\in\{1,\ldots,n\}}(m_i)}$$

avec N et M premiers entre eux. Afin de réaliser ces conditions, nous proposons d'étendre la base modulaire avec deux moduli supplémentaires m_{n+1} et m_{n+2}. Ainsi, la largeur de l'intervalle représentable avec la base $\mathcal{B}_m^{n+2}$ est de $\prod_{i=1}^{n+2} m_i = M'$.

Les restes associés à ces deux moduli supplémentaires sont calculés grâce à une extension de base. Ainsi, en choisissant judicieusement les deux moduli supplémentaires, les opérandes vérifient les conditions suivantes :

$$A < \frac{m_{n+2} - 1}{2}\frac{M'}{m_{n+2}}$$

$$B < 2N \qquad \text{et} \qquad 0 \leq N < \frac{M'}{3\ \sup_{i\in\{1,\ldots,n+2\}}(m_i)}$$

A priori cette extension de base nécessite un calcul coûteux. Soit X une opérande dont on cherche à étendre l'écriture modulaire de $\mathcal{B}_m^n$ vers $\mathcal{B}_m^{n+2}$. Etant donné que lors du calcul de notre algorithme une extension de base vers le système modulaire $\mathcal{B}_{\tilde{m}}^n$ est effectuée, le calcul des restes x_{n+1} et x_{n+2} peut être effectué sans perte de temps. En effet, une des méthodes d'extension de base consiste à convertir en "mixed radix" un nombre, puis à calculer ses restes dans le sytème modulaire de destination à partir de son écriture MRS (S-Sect. 1.2.3.1). On remarque que cette écriture est identique dans la base $\mathcal{B}_m^n$ et dans dans la base $\mathcal{B}_m^{n+2}$ (le modulus m_{n+2} correspond au chiffre de poids le plus fort). Puisque les opérandes sont inférieures à M, les chiffres les plus significatifs de leur représentation MRS sont nuls. Par conséquent la conversion effectuée lors de l'extension de base de $\mathcal{B}_m^n$ vers $\mathcal{B}_{\tilde{m}}^n$ est suffisante pour déterminer les restes x_{n+1} et x_{n+2}.

En définitive, le message est découpé en n vecteurs de p bits. Chaque vecteur est associé à un modulus codé sur $p+1$ bits. Ceci constitue l'écriture d'un nombre dans la base modulaire $\mathcal{B}_m^n$, mais pour utiliser notre algorithme de multiplication modulaire il est indispensable de remplir les conditions du théorème 3.1. Pour cela nous étendons la base modulaire dans laquelle est exprimé le message. Ainsi notre algorithme fonctionne dans le système modulaire principal $\mathcal{B}_m^{n+2}$ et le système auxiliaire $\mathcal{B}_{\tilde{m}}^{n+2}$

3.4 Implémentations

Nous présentons ici quelques implémentations de notre algorithme de multiplication modulaire. Nous nous intéresserons d'abord à diverses solutions sur un anneau de processeurs (S-Sect. 3.4.1). Ensuite nous donnerons une implémentation parallèle où la diffusion des données se fait en temps constant (S-Sect. 3.4.2).

3.4.1 Implémentation sur un anneau de processeurs

Pour certaines applications il n'est pas possible de communiquer les données en parallèle. C'est notamment le cas des implémentations sur carte à puce. Dans ce cas il peut être plus simple de communiquer les opérandes reste après reste [BdPF84]. D'autre part ce type d'application est limité en taille et il n'est pas possible de placer de gros opérateurs sur de tels circuits. L'architecture que nous présentons [Did97] offre ce double avantage : les données sont communiquées en séries sous forme modulaire reste après reste, et cette architecture modulaire peut se résumer à un seul processeur.

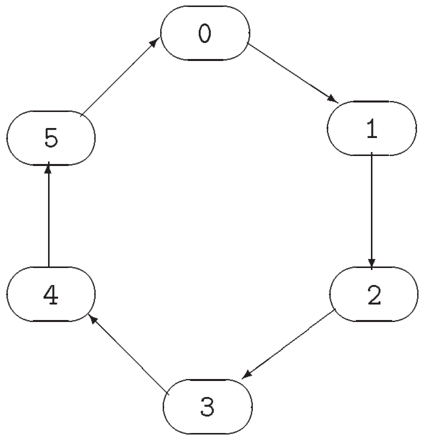


FIG. 3.3 – *Anneau de six processeurs*

On suppose que la représentation modulaire des opérandes est connue à la fois dans la base principale $\mathcal{B}_m^n$ et la base auxiliaire $\mathcal{B}_{\tilde{m}}^n$, et d'autre part, que les restes b_i et n_i des opérandes B et N sont déjà stockées sur chaque processeur. Cette supposition sur l'opérande B se justifie de la façon suivante. Dans le protocole d'identification de *Fiat* et *Shamir* les deux parties vérifient qu'elles connaissent toutes les deux un secret s sans le dévoiler à l'autre partie. Une des composantes des multiplications modulaires effectuées dans ce protocole est ce secret. C'est donc un élément qui n'est pas communiqué et qui est stocké localement. On considère enfin que les restes de l'opérande A sont communiqués reste après reste sur le même port d'entrée.

Une implémentation rapide de la représentation modulaire de n moduli est habituellement constituée de n processeurs indépendants effectuant chacun les opérations sur le modulus qui lui est assigné. une telle implémentation ne comporte aucune communication. Dans le cas qui nous intéresse, il est nécessaire de faire communiquer deux processeurs voisins. La structure d'interconnexion la plus simple est la structure en anneau avec communication unidirectionnelles (Fig. 3.3). L'architecture présentée ici est constituée de n processeur où le processeur j envoie ses données vers le processeur $j+1 \bmod n$ et reçoit ses données du processeur $j-1 \bmod n$. Les restes a_i de l'opérande A sont communiqués au processeur 0. Le processeur j stocke les constantes $m_j, \tilde{m}_j$, les valeur précalculées $(m_j - n_j)^{-1}_{m_j}$ et $(\tilde{m}_j - \tilde{n}_j)^{-1}_{\tilde{m}_j}$ ainsi que b_j et $\tilde{b}_j$ et les variables x_j $\tilde{x}_j$.

A_t: (à cette étape $j = i$ et $k = j$)
  Recevoir a'_j du processeur $(j-1) \bmod n$
  $q'_j \longleftarrow ((r_j + a'_j * b_j)(m_j - n_j)^{-1}_j) \bmod m_j$
  Envoyer la valeur actuelle de a'_j et q'_j au processeur $(j+1) \bmod n$
  $k \longleftarrow k+1$

B_t: **Recevoir** a'_i et q'_i du processeur $(j-1) \bmod n$
  $r_j \longleftarrow (r_j + a'_i * b_j + q'_i * n_j) * (m_i)^{-1}_j \bmod m_j$
  Envoyer la valeur actuelle de a'_i et q'_i au processeur $(j+1) \bmod n$
  $flag \longleftarrow 1$

C_t: $\tilde{r}_j \longleftarrow \left(\tilde{r}_j + a'_i * \tilde{b}_j + q'_i * \tilde{n}_j\right) * (m_i)^{-1}_{\tilde{m}_j} \bmod \tilde{m}_j$
  $flag \longleftarrow 0$
  $\longleftarrow i+1$

$C"_t$: $\tilde{r}_j \longleftarrow \left(\tilde{r}_j + a'_i * \tilde{b}_j + q'_i * \tilde{n}_j\right) * (m_i)^{-1}_{\tilde{m}_j} \bmod \tilde{m}_j$
  $\longleftarrow i+1$

D_t: (à cette étape $k = t - 2j$)
  Recevoir a'_k du processeur $(j-1) \bmod n$
  $a'_k \longleftarrow (a'_k - a'_j) * (m_j)^{-1}_k \bmod m_k$
  Envoyer a'_{k-1} au processeur $(j+1) \bmod n$
  $k \longleftarrow k+1$

D'_t: (à cette étape $k = n$)
  Envoyer a'_{k-1} au processeur $(j+1) \bmod n$

TAB. 3.1 – *Description des tâches effectuées sur le processeur j*

L'implémentation de l'algorithme 3.1 se décompose en plusieurs parties (Tab. 3.1). La première consiste en la conversion de l'opérande A de la représentation modulaire vers le système "Mixed Radix". Ces tâches notées D_t et D'_t par la suite sont intercalées entre les autres. La seconde tâche est le calcul d'un chiffre de la représentation MRS de Q. Nous noterons ces tâches de ce type A_t. Enfin le troisième type de tâches est la détermination du nouveau reste R. C'est dans cette dernière étape qu'est effectuée la division simple. Nous notons ce type de tâche B_t pour les calculs effectués dans $\mathcal{B}^n_m$ et C_t et C''_t pour ceux qui le sont dans $\mathcal{B}^n_{\tilde{m}}$.

Les tâches A, B, C, C'', D et D' sont décrites séparément dans la table 3.1. Les tâches

A reçoivent du processeur précédent un chiffre MRS de l'opérande A. Il faut noter que la tâche A effectuée sur le processeur 0 ne reçoit pas de données du processeur qui le précède sur l'anneau, mais de l'extérieur. En outre, elle n'a pas besoin d'attendre le calcul du premier chiffre $a'_1 = a_1$ qu'elle connaît dès le début.

Les autres chiffres MRS de A sont calculés avant qu'ils soient nécessaires par les tâches D et D'. Les restes a_i sont communiqués au processeur 0 les uns à la suite des autres et sont utilisés par les opérations D et D' qui constituent la décomposition de l'algorithme de conversion présenté dans la sous-section 1.2.2.2. On remarque que les tâches D du processeur 0 reçoivent leur données de l'extérieur et non d'un autre processeur.

Lorsque la valeur de q'_i est connue, le calcul de la nouvelle valeur du reste R est effectué par les tâches B pour tous ses restes r_j avec $i < j < n$ et les tâches C et C'' pour les restes $\tilde{r}_j$ avec $1 \leq j \leq n$. Etant donné que tous les restes sont indépendants les uns des autres, il n'est pas nécessaire d'attendre la fin du calcul de R pour tous ses restes. Les itérations de l'algorithme 3.1 sont donc pipelinés. La table 3.4.1 présente l'implémentation de la multiplication modulaire dans un système modulaire étendu de douze moduli sur un anneau de six processeurs.

Etape\processeur	0	1	2	3	4	5
0	A_t					
1	D_t	B_t				
2	D_t	C_t	B_t			
3	D_t	A_t	C_t	B_t		
4	D_t	D_t	B_t	C_t	B_t	
5	D_t	D_t	C_t	B_t	C_t	B_t
6	D'_t	D_t	A_t	C_t	B_t	C_t
7	$C"_t$	D_t	D_t	B_t	C_t	B_t
8	B_t	D'_t	D_t	C_t	B_t	C_t
9	C_t		D_t	A_t	C_t	B_t
10	B_t	$C"_t$	D'_t	D_t	B_t	C_t
11	C_t	B_t		D_t	C_t	B_t
12	B_t	C_t		D'_t	A_t	C_t
13	C_t	B_t	$C"_t$		D_t	B_t
14	B_t	C_t	B_t		D'_t	C_t
15	C_t	B_t	C_t			A_t
16	B_t	C_t	B_t	$C"_t$		
17	C_t	B_t	C_t	B_t		
18		C_t	B_t	C_t		
19			C_t	B_t	$C"_t$	
20				C_t	B_t	
21					C_t	
22						$C"_t$

TAB. 3.2 – *Table d'allocation des tâches sur un anneau de six processeurs*

L'ordonnancement des tâches sur l'anneau de processeur ainsi que les communications entre les tâches sont détaillées dans l'algorithme 3.4. Cet algorithme fonctionne en $4n-1$ itérations avec une possibilité de pipeliner les calculs toutes les $3n$ itérations [BDK97][Did97].

Interval	Value of i	Value of *flag*
$j \leq t \leq 3j-2$	$\left\lfloor \frac{t-j}{2} \right\rfloor$	$(t-j) \bmod 2$
$t = 3j-1$	$j-1$	0
$3j \leq t \leq 3j+n$	j	0
$t = 3j+n+1$	$j+1$	0
$3j+n+2 \leq t \leq 3n+j-1$	$\left\lfloor \frac{t-(3j+n+2)}{2} \right\rfloor + j$	$(t-(3j+n)) \bmod 2)$

Les valeurs de i et $flag$ sont des fonctions de j et t. Elles sont décrites dans la table suivante. Il est facile de vérifier que les valeurs des variables i et $flag$ sur le processeur j à l'étape t sont égales aux valeurs de i et $flag$ sur le processeur $(j+1) \bmod$ à l'étape $t+1$. En ce qui concerne la valeur de k, il apparaît que pour $3j \leq t \leq 2j+n$ on a $k = t-2j$. On vérifie facilement que la valeur de k sur le processeur j à l'étape t est égale à la valeur de k sur le processeur $(j+1) \bmod n$ à l'étape $j+2$ pour $j < n-1$.

Algorithme 3.4 *Fonctionnement de l'algorithme sur le processeur j*
(Note : Les boucles sont vides si la borne supérieure est négative ou est plus petite que la borne inférieure, i.e. le processeur 0 commence son calcul à l'étape 4). La variable t représente le temps

Initialisation: $i \longleftarrow 0$	$flag \longleftarrow 0$	$k \longleftarrow j$
for $t \longleftarrow 0$	**to** $j-1$	**do** *idle*
for $t \longleftarrow j$	**to** $3j-2$	**do if** $flag = 0$ **then** B **else** C
for $t \longleftarrow 3j-1$		**do** C
for $t \longleftarrow 3j$		**do** A
for $t \longleftarrow 3j+1$	**to** $2j+n-1$	**do** D
for $t \longleftarrow 2j+n$		**do** *D'*
for $t \longleftarrow 2j+n+1$	**to** $3j+n$	**do** *idle*
for $t \longleftarrow 3j+n+1$		**do** *C"*
for $t \longleftarrow 3j+n+2$	**to** $j+3n-2$	**do if** $flag = 0$ **then** B **else** C
for $t \longleftarrow 3n+j-1$		**do** C
for $t \longleftarrow 3n+j$	**to** $4n-2$	**do** *idle*

Suite à ce qui a été dit précédement, on obtient le théorème suivant sur la complexité de cette implémentation.

Théorème 3.2 *L'algorithme 3.4 de multiplication modulaire en représentation modulaire effectue 'n − 1 itérations sur un anneau de processeur n .*

3.4.2 Implémentation sur un bus de processeurs

Nous avons vu, dans l'implémentation précédente, que la diffusion des valeurs a'_i et q'_i nécessitait est effectuée en temps linéaire. Nous présentons ici une implémentation permettant cette diffusion en temps constant. Cette architecture comporte n cellules RNS reliées à un bus grâce à un processeur de communication (Fig. 3.4). Celui-ci est chargé d'envoyer et de réceptionner les données sur le bus. Chaque cellule RNS est chargée des calculs sur un modulus de la base principale et un modulus de la base auxiliaire. Cette implémentation est adaptée aux machines parallèles à bus comme le super calculateur Capitan de Matra ou une carte de calcul spécialisée à base de FPGA.

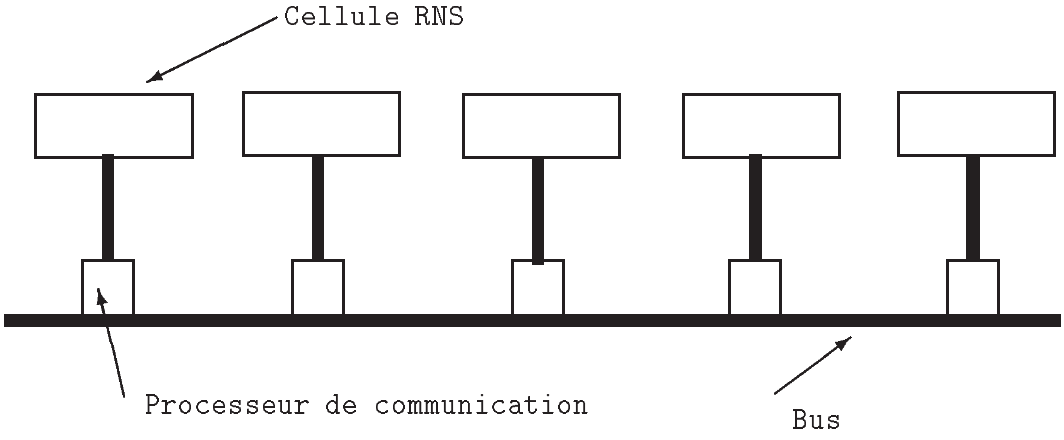


FIG. 3.4 – *Bus de processeurs*

On suppose que la représentation modulaire des opérandes est connue dans la base principale $\mathcal{B}_m^n$ et la base auxiliaire $\mathcal{B}_{\tilde{m}}^n$. Dans cette implémentation on suppose que les restes $a_i, b_i, \tilde{b}_i, n_i$ et $\tilde{n}_i$ sont communiqués en parallèle aux cellules RNS.

Algorithme 3.5 Conversion en "mixed radix" sur un bus de processeurs
Fonctionnement de l'algorithme sur le processeur j
0 *Initialisation* $i \longleftarrow 1; x'_j \longleftarrow x_j$
1 **for** $t < j$ **do** D_t
2 **for** $t = j$ **do** D'_t

Cas particulier sur le processeur 1

2 **for** $t = j$ **do** $D"_t$

L'implémentation que nous proposons se décompose en deux parties : la conversion de A_{RNS} en A_{MRS} sur $\mathcal{B}_m^n$, puis la multiplication proprement dite. La conversion est constituée de trois tâches D_t, D'_t et $D"_t$ (Tab. 3.3). Ce calcul complètement parallélisé est donné par l'algorithme 3.5. La conversion est effectuée en n itérations où une diffusion sur le bus est effectuée à chaque fois.

La figure 3.5 donne l'implémentation de la conversion en "mixed radix" sur un bus de six processeurs. Cette conversion permet de connaître la représentation mixed radix de l'opérande A qui sera utilisée lors de la multiplication modulaire.

D_t: (x_j est présent sur le processeur j)
  Recevoir x'_i du bus
  $x'_j \longleftarrow (x'_j - x'_i) \times m_{i,j}^{-1} \bmod m_j$
  $i \longleftarrow i+1$

D'_t: (x_j est présent sur le processeur j)
  Recevoir x'_i du bus
  $x'_j \longleftarrow (x'_j - x'_i) \times m_{i,j}^{-1} \bmod m_j$
  Diffuser la valeur actuelle de x'_j sur le bus

$D"_t$: **Diffuser** la valeur actuelle de x'_j sur le bus

TAB. 3.3 – *Description des tâches effectuées sur le processeur j pour la conversion "mixed radix" sur un bus de processeurs*

Etape\processeur	1	2	3	4	5	6
1	$D"_t$					
2		D'_t	D_t	D_t	D_t	D_t
3			D'_t	D_t	D_t	D_t
4				D'_t	D_t	D_t
5					D'_t	D_t
6						D_t

FIG. 3.5 – *Table d'allocation des tâches de la conversion sur un bus de six processeurs*

Algorithme 3.6 Multiplication modulaire sur un bus de processeurs
Fonctionnement de l'algorithme sur le processeur j

for $t \longleftarrow 2j-1$ **to** $2n+1$ **do** A_t
for $t \leq j$ *and* (t *is even*) *and* $t < 2n-1$ **do** B_t
for (t *is odd*) *and* $1 < t < 2n$ *and* $t \neq 2j-1$ **do** C_t
for $t = 2j$ *and* $t > 2$ **do** C_t
for $t = 2n$ **do** C_t

Cas particulier sur le processeur n

for $t \longleftarrow 2n+11$ **do** C_t

La multiplication modulaire se décompose en trois tâches A_t, B_t et C_t (Algo. 3.6. La tâche A_t calcule la valeur du nouveau chiffre q'_i) qu'elle diffuse ensuite. Cette valeur est récupérée par les tâches B_t et C_t qui calculent la nouvelle valeur de R. Leur description complète est donnée par la table 3.4. On peut noter que, du fait de la perte d'un modulus dans la base modulaire principale lors de la division par m_i, le calcul de la nouvelle valeur de R pour ces moduli est inutile.

Cet algorithme permet de calculer la multiplication modulaire en représentation mo-

A_t: (a'_j est présent sur le processeur j)
$q'_j \longleftarrow ((r_j + a'_j * b_j)(m_j - n_j)_j^{-1}) \bmod m_j$
Diffuser la valeur actuelle de a'_j et q'_j sur le bus

B_t: **Recevoir** a'_i et q'_i du bus
$r_j \longleftarrow (r_j + a'_i * b_j + q'_i * n_j) * (m_i)_j^{-1} \bmod m_j$

C_t: **Recevoir** a'_i et q'_i du bus
$\tilde{r}_j \longleftarrow \left(\tilde{r}_j + a'_i * \tilde{b}_j + q'_i * \tilde{n}_j\right) * (m_i)_{\tilde{m}_j}^{-1} \bmod \tilde{m}_j$

TAB. 3.4 – *Description des tâches effectuées sur le processeur j pour la multiplication modulaire sur un bus de processeurs*

dulaire en $2n+1$ étape. Le fonctionnement de cet algorithme sur un bus de six processeurs est illustré par la figure 3.6. Le temps total du calcul de la conversion MRS de l'opérande A et le calcul de la multiplication modulaire est effectué en $3n$ étapes.

Etape\processeur	1	2	3	4	5	6
1	A_t					
2		B_t	B_t	B_t	B_t	B_t
3	C_t	A_t	C_t	C_t	C_t	C_t
4		C_t	B_t	B_t	B_t	B_t
5	C_t	C_t	A_t	C_t	C_t	C_t
6			C_t	B_t	B_t	B_t
7	C_t	C_t	C_t	A_t	C_t	C_t
8				C_t	B_t	B_t
9	C_t	C_t	C_t	C_t	A_t	C_t
10					C_t	B_t
11	C_t	C_t	C_t	C_t	C_t	A_t
12	C_t	C_t	C_t	C_t	C_t	C_t
14						C_t

FIG. 3.6 – *Table d'allocation des tâches de la multiplication modulaire sur un bus de six processeurs*

Le théorème suivant résume les résultats précédents.

Théorème 3.3

- *L'algorithme 3.6 de multiplication modulaire en représentation modulaire effectue $2n+1$ itérations sur un bus de n processeurs.*
- *L'algorithme 3.5 de conversion "mixed radix" effectue n itérations sur un bus de n processeurs..*

– *La conversion suivie de la multiplication modulaire est déterminée en* $3n$ *itération.*

3.5 Autres algorithmes

La représentation modulaire des nombres est intéressante pour effectuer des calculs sur de très grands entiers et donc pour des applications cryptographiques. Le problème de la multiplication modulaire et de la réduction modulaire en RNS a déjà été abordé. Nous présentons dans cette section différents travaux parus dans la littérature scientifique. Dans la sous-section 3.5.1 nous décrivons brièvement un algorithme permettant de déterminer la réduction modulaire d'un entier $X \bmod N$. Nous abordons dans la sous-section 3.5.2 une approche de la multiplication modulaire par des divisions. Enfin dans la sous-section 3.5.3 nous présentons une méthode permettant le décryptage de messages chiffrés avec l'algorithme RSA.

3.5.1 Réduction de Posch et Posch

Karl Posch et *Reinhart Posch* décrivent un algorithme de réduction modulaire en représentation modulaire [PP92] [PP95]. Cet algorithme consiste à calculer $X \bmod N$ dans un système modulaire $\mathcal{B}_m^n$ où N et M sont premiers entre eux. L'idée qui est à la base de cet algorithme est la méthode de réduction sans division proposée par *Montgomery* [Mon85]. Elle se résume aux deux congruences suivantes :

$$\frac{X - (X \bmod R) \times (N_R^{-1} \bmod R) \times N}{R} \equiv X \bmod N \tag{3.2}$$

$$X - (X \bmod R) \times (N_R^{-1} \bmod R) \times N \equiv 0 \bmod R \tag{3.3}$$

avec N et R premiers entre eux.

Ces équations peuvent être utilisées pour calculer $X \bmod N$ de façon efficace. On choisira l'entier R de sorte que $X \bmod R$ puisse être calculé facilement. En numération de position, R est une puissance de la base dans laquelle les opérandes sont exprimées. *Karl* et *Reinhart Posch* proposent d'effectuer ce calcul en représentation modulaire, avec $R = M$. Les équations (3.2) et (3.3) se réécrivent de la façon suivante :

$$\frac{X - (X \bmod M) \times (N_M^{-1} \bmod M) \times N}{M} \equiv X \bmod N \tag{3.4}$$

$$X - (X \bmod M) \times (N_M^{-1} \bmod M) \times N \equiv 0 \bmod M \tag{3.5}$$

Les valeurs manipulées sont des entiers [PP92] [PP95]. Afin de pouvoir effectuer la division par M de l'équation 3.4 (qui n'est pas possible dans le système $\mathcal{B}_m^n$) les auteurs font appel à un système modulaire auxiliaire $\mathcal{B}_p^t$ de sorte que $P = \prod_{i=1}^{t} p_i > M$ et PGCD(M, P)=1. Cette division est effectuée après un changement de base en effectuant une multiplication par l'inverse modulaire de M modulo P (S-Sect. 1.3.5). Ils établissent que pour $P > M/2$, $M > 4 \times N$ et $\sqrt{X} < N/2$ l'algorithme suivant détermine :

$$Z = \frac{X}{M} \bmod N$$

Base principale $\mathcal{B}_m^n$	Base auxiliaire $\mathcal{B}_p^t$	Opération
X	X	Extension de base
$U \leftarrow X \times N_M^{-1}$		Multiplication
U	U	Extension de base
	$V \leftarrow U \times N$	Multiplication
	$W \leftarrow X - V$	Soustraction
	$Y \leftarrow W \times M_P^{-1}$	Multiplication (Division par M)
	$Z \leftarrow Y + N$	Addition
Z	Z	Extension de base

Lors de chaque réduction trois multiplications et deux additions en représentation modulaires ainsi que trois extensions de base sont effectuées. Selon l'algorithme d'extension de base utilisé (S-Sect. 1.2.3), la complexité en temps de la réduction de *Posch* et *Posch* est de $\mathcal{O}(\log n)$ à $\mathcal{O}(n)$ pour une complexité en espace de $\mathcal{O}(n^2)$. On remarque que l'implémentation de cette réduction prend beaucoup de place et n'est pas très adaptée à la manipulation de grands nombres et donc aux applications cryptographiques.

En effet, on peut remarquer que dans le cas d'une multiplication modulaire en RNS sur des valeurs inférieures à M, cette réduction doit être effectuée après une multiplication. Afin de pouvoir représenter le résultat intermédiaire, il nous faut un système modulaire comportant deux fois plus de moduli. En conséquence, les changements de base de la réduction ont un coût en espace multiplié par quatre.

En conclusion, l'algorithme de réduction de *Posch* et *Posch* ne peut pas, avec les algorithmes de changement de base actuels, être implémenté efficacement pour des application cryptographiques.

3.5.2 Réduction par division

Hung et *Parhami* [HP94b] proposent d'utiliser leurs algorithmes de division présentés dans [HP94a] pour le chiffrage et le déchiffrage de messages avec l'algorithme de cryptage à clef publique RSA [RSA78]. La réduction effectuée lors d'une multiplication modulaire est effectuée suivant le même entier N. Dans les applications cryptographiques tel que le codage RSA, il s'agit d'une partie de la clef publique. Dans ce dernier cas le codage et le décodage sont des exponentiations modulaires $X^Y \bmod N$ (S-Sect. 3.1.1) qui peuvent se décomposer en $\mathcal{O}(\log N)$ multiplications modulaires [Kor93b].

Hung et *Parhami* ont choisit la représentation modulaire pour la grande rapidité de la multiplication dans cette représentation. Lors du codage et décodage de message ils proposent de convertir les données dans ce système, d'y effectuer le chiffrage ou le déchiffrage, puis de convertir à nouveau le résultat. Pour l'opération de cryptage et de décryptage ils choisissent d'effectuer des multiplications classiques suivies d'une division par N en représentation modulaire au lieu de multiplications modulaires. Afin de pouvoir représenter le produit $X \times Y$ ils suggèrent d'augmenter l'intervalle de représentation du système modulaire et de doubler le nombre de moduli.

Le coût de cet algorithme de multiplication modulaire se détermine de la façon suivante. Tout d'abord les algorithmes de division utilisés nécessitent des précalculs. La complexité en temps de ces précalculs est $\mathcal{O}(n)$ pour l'algorithme de division dont la complexité en temps est $\mathcal{O}(n \log n)$; elle est $\mathcal{O}(n^2)$ pour l'algorithme en $\mathcal{O}(n)$. A ceci il faut rajouter le coût des conversions qui sont de $\mathcal{O}(\log M)$ pour la conversion binaire -

représentation modulaire et $\mathcal{O}((\log n) \times (\log M))$ pour la conversion inverse. D'après *Hung* et *Parhami* leur algorithme est intéressant comparé aux implémentations proposées dans [Bar86] et [KH88].

En comparaison, l'algorithme que nous proposons dans [BDK97] ne nécessite pas une conversion de toutes les opérandes. Notre multiplication modulaire est effectuée sur les opérandes exprimées en représentation modulaire. D'autre part la complexité en espace est moins importante. En effet la division utilisée dans l'algorithme de *Hung* et *Parhami* a une complexité en espace de $\mathcal{O}(n^2)$ (S-Sect. 2.3.5 et [HP94a]) à comparer avec la complexité de notre algorithme qui est de $\mathcal{O}(n)$. En résumé la multiplication modulaire par une division semble complexe à mettre en oeuvre et difficile à implémenter.

3.5.3 Décryptage de Chren

Les multiplications modulaires effectuées dans les crypto-algorithmes présentés dans la section 3.1 sont effectuées modulo N où N est le produite de deux grands nombres premier p et q. *Chren* et *Brogdon* proposent dans leur article [CB96] d'effectuer les multiplications modulaires dans le système modulaire $\mathcal{B}_m^2 = \{p, q\}$. Le résultat est ainsi obtenu modulo N.

Le principal inconvénient de cette approche est qu'elle n'est exploitable uniquement pour le décodage. En effet pour pouvoir effectuer les opérations dans le système modulaire dans $\mathcal{B}_m^2 = \{p, q\}$ un utilisateur devra connaître les valeur p et q et sera par conséquent capable de calculer la clef privée.

D'autre part le gain apporté par la méthode de *Chren* et *Brogdon* n'est pas très important. Seul deux moduli sont utilisés et ne permettent pas d'effectuer des calculs facilement. Par exemple une clef de 1024 bits se décompose en deux moduli de 512 bits et une multiplication modulaire sur 1024 bits se résume à deux multiplications modulaires sur 512 bits.

3.6 Conclusion

Nous avons présenté dans ce chapitre le premier algorithme de multiplication modulaire en représentation modulaire qui est similaire à l'algorithme de *Peter Mongomery*. Cet algorithme peut être utilisé pour des applications cryptographiques sans effectuer de conversion. En effet, nous avons montré qu'un message peut être considéré comme étant un nombre exprimé en représentation modulaire. Ainsi notre algorithme de multiplication modulaire peut être employé sans effectuer de conversion. De plus il peut être utilisé pour le calcul de l'exponentiation modulaire.

Dans le cadre des applications cryptographiques, la multiplication modulaire en représentation modulaire offre un avantage sur les représentations redondantes. La comparaison de deux nombres modulaires est triviale contrairement à celle entre deux nombres en notation redondante. Nous avons vu dans la section 3.1 que certaines applications cryptographiques effectuent des tests d'égalité entre deux produits modulaire. Contrairement représentation redondantes un tel test peut être fait très facilement en représentation modulaire.

La complexité en temps et en espace de notre algorithme est similaire à celle de l'algorithme de Montgomery en arithmétique redondante. Toutefois aucune comparaison de l'efficacité de l'implémentation de la multiplication modulaire en représentation modulaire et en arithmétique redondante n'a été effectuée. Au vu des résultats théoriques aucun argument nous permet de savoir quelle est la meilleure solution. Bien que la comparaison soit facile en notation modulaire nous ne savons pas si ce gain est suffisant. La comparaison entre les deux implémentation est en cours d'étude

Nous allons poursuivre ce travail par la réalisation de l'exponentiation modulaire en représentation modulaire. Ce calcul peut être obtenu grâce à l'adaptation de l'algorithme du calcul de l'exponentielle [CLR94].

Conclusion

La représentation modulaire permet d'effectuer des additions, soustractions et multiplications en ramenant ces calculs à des manipulations de petites valeurs. Cette propriété la rend adaptée au calcul sur de grands entiers. Nous avons présenté dans le premier chapitre un état de l'art en matière de calcul RNS, un nouvel algorithme de division dans le second et un algorithme de multiplication modulaire dans le troisième.

La division que nous avons donnée permet de calculer le quotient de deux nombres en représentation modulaire avec une complexité en temps de $\mathcal{O}(n \log n)$ et une complexité en espace de $\mathcal{O}(n)$. Elle a un coût similaire à celui des divisions dans des représentations à numération de position (SRT, division restaurante). Cette avancée permet de relancer l'intérêt de la représentation modulaire pour effectuer des calculs rapides. En effet, d'une part l'addition et la multiplication, qui sont des opérations fréquement utilisées, sont très performantes et d'autre part, nous pouvons y effectuer une division aussi rapidement que dans des systèmes de numération de position.

La multiplication modulaire en représentation modulaire, que nous avons développée dans le troisième chapitre, reprend les principes de la multiplication de Montgomery. Sa complexité en temps et en espace est comparable à celle d'une multiplication de Montgomery en représentation redondante en grande base. Nous avons montré par ailleurs, que la représentation pouvait être employée pour exprimer un message, la suite de bits le composant pouvant être considérée comme étant un grand nombre exprimé en représentation modulaire. Ainsi notre multiplication modulaire peut être utilisée à des fins cryptographiques avec un avantage sur les représentations redondantes : le test d'égalité peut être effectué rapidement et ne nécessite pas de conversion. Nous avons défini une architecture parallèle pour le calcul de cette multiplication sur un anneaux de processeurs permettant l'entrée d'une des opérandes en série. La nature modulaire de la représentation RNS la rend intéressante pour l'implémentation d'opérateurs sur carte à puce où la taille des opérateurs est un élément critique. Dans ce cas, un seul opérateur peut être employé pour effectuer les opérations modulaires.

Les algorithmes que nous avons donnés permettent d'élargir le champs des applications, jusqu'ici assez restreint, pour lesquelles la représentation modulaire peut être employée. Mais notre travail comporte plusieurs prolongements. Tout d'abord il serait intéressant d'étudier le comportement de notre division sur de petits systèmes modulaires qui sont notamment utilisés en traitement du signal gros consommateurs de calculs. La synthèse d'une cellule élémentaire de calcul modulaire pouvant être assemblée avec d'autres en un opérateur modulaire capable d'effectuer aussi bien des additions que des multiplications et des divisions en représentation modulaire, nous permettrait de simplifier la création d'architectures destinées au traitement du signal ou à des applications plus générales. Ainsi, il semble possible de définir un opérateur modulable pour le calcul en grande précision de l'addition, la soustraction, la multiplication et la division.

Ensuite, nos travaux sur la multiplication modulaire devraient se prolonger par son implémentation hardware pour des applications cryptographiques telles que le codage RSA ou l'identification de *Fiat* et *Shamir*. L'implémentation d'un tel opérateur modulaire dans une carte à puce est une solution qui nous semble intéressante. Les qualités modulaires de cet algorithme nous font envisager son implémentation sur circuits reconfigurables.

Enfin, nous envisageons de poursuivre notre étude de la représentation modulaire par la mise en oeuvre d'un algorithme du simplexe en grande précision avec des divisions exactes, la comparaison détaillée de l'implémentation des différents algorithmes de division en RNS et par l'étude de la complexité de la division de *Gamberger*.

Bibliographie

[AM91] G. Alia and E. Martinelli. A VLSI modulo m multiplier. *IEEE Transactions on Computers*, 40(7):873–878, 1991.

[AM93] G. Alia and E. Martinelli. On the lower bound to the VLSI complexity of number conversion from weighted to residue representation. *IEEE Transactions on Computers*, 42(8):962–967, 1993.

[AO79] D. E. Atkins and S.-C. Ong. Time-component complexity of two approaches to multioperand binary addition. *IEEE Transactions on Computers*, C-28(12):918–926, 1979.

[Bar86] P. Barrett. Implementing the rivest, shamir and adleman public key encryption algorithm on a standard digital processor. In A. M. Odlyzko, editor, *Advances in Cryptology, Proceedings of Crypto'86*, pages 311–323, 1986.

[Bay86] M. A. Bayoumi. Implementation of RNS multiplication in VLSI. In D.E Kirk, editor, *1985-1986 Nineteeth Asilomar Conference on Circuits, Systems and Computers*, pages 48–52, Pacific Grove, USA, 1986. IEEE Computer Society Press.

[BCG81] D.K. Banerji, T.Y. Cheung, and V. Ganesan. A high speed division method in residue arithmetic. In *5th Symposium on Computer Arithmetic*, pages 158–164, Ann Arbor, USA, 1981. IEEE Computer Society Press.

[BDK97] J.-C. Bajard, L.-S. Didier, and P. Kornerup. An "rns" montgomery modular multiplication algorithm. In T. Lang, J.-M. Muller, and N. Takagi, editors, *13rd IEEE Symposium on Computer Arithmetic*, pages 234–239, Asilomar, USA, 1997. IEEE Computer Society Press.

[BDM] J.-C. Bajard, L.-S. Didier, and J.-M. Muller. A new euclidian division algorithm for residue number systems. *Journal of VLSI Signal Processing*. A paraître.

[BDM96] J.-C. Bajard, L.-S. Didier, and J.-M. Muller. A new euclidian division algorithm for residue number systems. In J. Fortes, C. Mongenet, K.K. Parhi, and V.E. Taylor, editors, *IEEE International Conference on Application-Specific Systems, Architectures and Processors: ASAP'96*, pages 45–54, Chicago, USA, 1996. IEEE Computer Society Press.

[BdPF84] O. Brugia, A. di Porto, and P. Filiponi. Un metodo per migliorare l'efficienza degli algorithmi di generazione delle chiavi crittgrafiche basati sull'impegio di grandi numeri primi. *Note Recesioni e Notizie, Minister Poste e Telecommunicazioni*, 33(1-2):15–22, 1984.

[BEPP97] H. Brönnimann, I. Z. Emiris, V. Pan, and S. Pion. Computing exact geometric predicates using modular arithmetic with single precision. In *Proceedings of the 13th Annual Symposium on Computational Geometry*, pages 174,182, Nice, France, 1997. ACM.

[BJM83] M. A. Bayoumi, G. A. Jullien, and W. C. Miller. Models for VLSI implementation of residue number system arithmetic modules. In *6th IEEE Symposium on Computer Arithmetic*, pages 174–182, Aarhus, Denmark, 1983. IEEE Computer Society Press.

[BJM85] M. A. Bayoumi, G. A. Jullien, and W. C. Miller. An efficient VLSI adder for DSP architectures based on RNS. In *IEEE Conference on Acoustics, Speech and Signal Processing (ICASSP'85)*, volume 4, pages 38.2.1–38.2.4, Tampa, USA, 1985. IEEE Computer Society Press.

[BJM87] M. A. Bayoumi, G. A. Jullien, and W. C. Miller. A look-up table VLSI design methodology for RNS structures used in DSP applications. *IEEE Transactions on Circuits and Systems*, CAS-34(6):604–616, 1987.

[BM91] F. Barsi and E. Martinelli. A VLSI architecture for RNS with mi adders. *Integration, The VLSI Journal*, 11:67–83, 1991.

[CB96] W. A. Chren and C. H. Brogdon. RSA decryption using the one-hot residue number system. In A. Jerdee G. Cameron, M. Hassoun and C. Melvin, editors, *39th Midwest Symposium on Circuits and Systems*, Ames, USA, 1996. IEEE Computer Society Press.

[Chr90] W. A. Chren. A new residue number system division algorithm. *Computers and Mathematics with Applications*, 19(7):13–19, 1990.

[CHW96] P.-S. Chen, S.-A. Hwang, and C.-W. Wu. A systolic RSA public key cryptosystem. In *IEEE International Symposium on Circuits and Systems (ISCAS'96)*, volume 4, pages 408–411. IEEE Computer Society Press, 1996.

[CL91] J.-S. Chiang and M. Lu. A general division algorithm for residue number systems. In P. Kornerup and D. W. Matula, editors, *10th IEEE Symposium on Computer Arithmetic*, pages 76–83, Grenoble, France, 1991. IEEE Computer Society Press.

[CL92] J.-S. Chiang and M. Lu. A novel division algorithm for the residue number system. *IEEE Transactions on Computers*, 41(8):1026–1032, Aug. 1992.

[CLR94] T. Cormen, C. Leiserson, and R. Rivest. *Introduction à l'algorithmique*. Dunod, 1994.

[CPO95] E. D. Di Claudio, F. Piazza, and G. Orlandi. Fast combinatorial RNS processors for DSP applications. *IEEE Transactions on Computers*, 44(5):624–633, 1995.

[CT93] M. Cosnard and D. Trystram. *Algorithmes et architectures parallèles.* InterEditions, 1993.

[DH76] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 2:644–654, Nov. 1976.

[Did97] L.-S. Didier. Implémentation d'un algorithme de multiplication modulaire en "rns" sur un anneau de processeurs. In A. Schiper and D. Trystram, editors, *Renpar'9 9eme Rencontre Francophones du Parallélisme*, pages 185–188, Lausanne, Suisse, 1997.

[DIP93] G. Dimauro, S. Impedovo, and G. Pirlo. A new technique for fast number comparison in the residue number system. *IEEE Transactions on Computers*, 42(5):608–612, May 1993.

[EB92] K. M. Elleithy and M. A. Bayoumi. Fast and flexible architectures for RNS arithmetic decoding. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 39(4):226–235, 1992.

[EW93] S. E. Eldridge and C. D. Walter. Hardware implementation of montgomery's modular multiplication algorithm. *IEEE Transaction on Computers*, 42(6):693–699, June 1993.

[FS86] A. Fiat and A. Shamir. How to improve yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology - Proceedings of Crypto'86*, pages 186–194, 1986.

[Gam89] D. Gamberger. Incompletely specified numbers in the residue number system - definition and applications. In M. D. Ercegovac and E. Swartzlander, editors, *9th IEEE Symposium on Computer Arithmetic*, pages 210–215, Santa Monica, U.S.A, 1989. IEEE Computer Society Press.

[Gam91] D. Gamberger. New approach to integer division in residue number systems. In Peter Kornerup and David Matula, editors, *10th IEEE Symposium on Computer Arithmetic*, pages 84–91. IEEE Computer Society Press, 1991.

[Gar59] H. L. Garner. The residue number system. *IRE Transactions on Electronic Computers*, EL-8(6):140–147, June 1959.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23:5–48, March 1991.

[HK95] M.A. Hitz and E. Kaltofen. Integer division in residue number systems. *IEEE Transactions on Computers*, 44(8):983–989, August 1995.

[HM62] Hardy G. H. and Wright E. M. *An introduction to the theory of numbers.* Clarendon Press, Oxford, 4 edition, 1962.

[HP94a] C. Y. Hung and B. Parhami. An approximate sign detection method for residue numbers and its application to RNS division. *Computers and Mathematics with Applications*, 27(4):23–35, Feb. 1994.

[HP94b] C. Y. Hung and B. Parhami. Fast RNS division algorithm for fixed divisors with application to RSA encryption. *Information Processing Letters*, 51(4):163–169, 1994.

[HP95] C. Y. Hung and B. Parhami. Error analysis of approximate chinese-remainder-theorem decoding. *IEEE Transactions on Computers*, 44(11):1344–1348, 1995.

[Hua83] C. H. Huang. A fully parall mixed-radix conversion algorithm for residue number applications. *IEEE Transactions on Computers*, C-32(4):398–402, 1983.

[JB97] Y.-J. Jeong and W. P. Burleson. VLSI array algorithms and architectures for RSA modular multiplication. *IEEE Transactions on Very Large Scale of Integration (VLSI) Systems*, 5(2):211–217, 1997.

[Jul78] G. A. Jullien. Residue number scaling and other operations. *IEEE Transactions on Computers*, C-27(4):325–336, 1978.

[KAK96] Ç. K. Koç, T. Acar, and B. S. Kaliski. Analyzing and comparing montgomery multiplication algorithms. *IEEE Micro*, 16(6):26–33, Jun. 1996.

[KH88] S. Kawamura and K. Hirano. A fast modular arithmetic algorithm using e residue table. In C. G. Günther, editor, *Advances in Cryptology, Proceedings of Crypto'88*, pages 245–250, 1988.

[KJM86] R. Krishnan, G. A. Jullien, and C. Miller. Complex digital signal processing using quadratic residue number systems. *IEEE Transactions on Acoustic, Speech and Signal Processing*, 34(1):166–177, 1986.

[Knu81] Donald Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 2 edition, 1981.

[Kor93a] I. Koren. *Computer Arithmetic Algorithms*. Prenctice Hall, 1993.

[Kor93b] P. Kornerup. High-radix modular multiplication for cryptosystems. In G. Jullien M.J Irwin, E. Swartzlander, editor, *11st IEEE Symposium on Computer Arithmetic*, pages 277–283, Windsor, Canada, 1993. IEEE Computer Society Press.

[Lo91] H.-Y. Lo. Multibit systolic arrays for high-speed signal processing in residue number system. In K. R. Subramanian and E. S. Seumahu, editors, *Computer, Communication and Networking Systems: An Integrated Perspective. Proceedings of the International Conference on Information Engineering - ICIE '91*, volume 2, pages 647–656, Singapore, 1991.

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

[MS88] S. Micali and A. Shamir. An improvement of the fiat-shamir identification and signature scheme. In *Advances in Cryptology - Proceedings of Crypto'88*, pages 244–247, 1988.

[Mul89] Jean-Michel Muller. *Arithmétique des ordinateurs Opérateurs et fonctions élémentaires*. Masson, 1989.

[MY93] H. Morita and C.-H. Yang. A modular-multiplication algorithm using lookahead determination. *IEICE Transactions on Fundamentals of Electronics, Communications and Computers*, E76-A(1):70–76, 1993.

[OK91] H. Orup and P. Kornerup. A high-radix hardware algorithm for calculating the exponential m^e modulo n. In P. Kornerup and D. W. Matula, editors, *10th IEEE Symposium on Computer Arithmetic*, pages 51–56, Grenoble, France, 1991. IEEE Computer Society Press.

[Par93] B. Parhami. Optimal table-lookup schemes for binary-to-residue and residue-to-binary conversions. In *27th Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 812–816, Pacific Grove, USA, 1993. IEEE Computer Society Press.

[PP92] K. C. Posch and R. Posch. Residue number system: A key to parallelism in public key cryptography. In *4th Symposium on Parallel and Distributed Processing*, pages 432–435. IEEE Computer Society Press, 1992.

[PP95] K. C. Posch and R. Posch. Modulo reduction in residue number systems. *IEEE Transaction on Parallel and Distributed Systems*, 6(5):449–454, 1995.

[Rob58] J.E. Robertson. A new class of digital division methods. *IRE Transactions on Electronic Computers*, EC-7:218–222, 1958. Reprinted in E.E. Swartzlander, Computer Arithmetic, Vol. 1, IEEE Computer Society Press Tutorial, 1990.

[Rob70] J.E. Robertson. The correspondence between methods of digital division and multiplier recoding procedures. *IEEE Transactions on Computers*, C-19(8), August 1970.

[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.

[RY91] D. Radhakrishnan and Y. Yuan. Fast and highly compact RNS multipliers. *International Journal on Electronics*, 70(2):281–293, 1991.

[Sal93] Serge Salaneuve. Arithmétique modulaire. Master's thesis, Ecole Normale Supérieure de Lyon, June 1993.

[Sch97] B. Schneider. *Cryptographie Appliquée*. Thomson Publishing, 2 edition, 1997.

[SF77] M. A. Soderstrand and E. L. Fields. Multipliers for residue number arithmetic digital filters. *Electronics Letters*, 13(6):164–166, mar 1977.

[SH88] A. Sweidan and A. Hiasat. A new efficient memoryless residue to binary converter. *IEEE Transactions on Circuits and Systems*, 35(11):1441–1444, 1988.

[SJJT86] M. Soderstrand, W. Jenkins, G. Jullien, and F. Taylor, editors. *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing*, New-York, 1986. IEEE Press.

[SK89] A. P. Shenoy and R. Kumaresan. Fast base extension using a redundant modulus in RNS. *IEEE Transactions on Computer*, 38(2):292–296, 1989.

[SPS92] D. Soudris, V. Paliouras, and T. Stouraitis. Systematic development of architectures for multidimensional DSP using the residue number system. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP92)*, volume 3, pages 397–400, San Francisco, USA, 1992. IEEE Computer Society Press.

[ST67] N. S. Szabo and R. I. Tanaka. *Residue Arithmetic and its Applications to Computer Technology*. McGraw-Hill, 1967.

[SV93] M. Shand and J. Vuillemin. Fast implementations of RSA cryptography. In *11th IEEE Symposium on Computer Arithmetic*, pages 252–259. IEEE Computer Society Press, 1993.

[Tak92] N. Takagi. A radix-4 modular multiplication hardware algorithm for modular exponentiation. *IEEE Transactions on Computers*, C-41(8):949–956, 1992.

[Tak93] N. Takagi. A multiple-precision modular multiplication algorithm with triangle additions. In M. J. Irwin E. Swartzlander and G.A. Jullien, editors, *11th IEEE Symposium on Computer Arithmetic*, pages 272–276, Windsor, Ont., Canada, 1993. IEEE Computer Society Press.

[Tak95] N. Takagi. A multiple-precision modular multiplication algorithm with triangle additions. *IEICE Transactions on Information and Systems*, E78-D(10):1313–1315, 1995.

[Tay84] F. J. Taylor. Residue arithmetic: A tutorial with examples. *IEEE Computer*, 17(5):50–62, 1984.

[Toc58] K.D. Tocher. Techniques of multiplication and division for automatic binary divider. *Quarterly journal of mechanics and applied mathematics*, 11(3):364–384, 1958.

[VVDJ90] A. Vandemeulebroecke, E. Vanzieleghem, T. Denayer, and P. G. A. Jesper. A new carry-free division algorithm and its application to a single-chip 1024-b RSA processor. *IEEE Journla of Solid-State Circuits*, 25:748–756, 1990.

[Wal64] C. S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, EC-13:14–17, 1964.

[Wal93] C. D. Walter. Systolic modular multiplication. *IEEE Transactions on Computers*, 42(3):376–378, 1993.

[Wal97] C. D. Walter. Space/time trade-offs for higher radix modular multiplication using repeated addition. *IEEE Transactions on Computers*, 46(2):139–141, 1997.

[WJM97] Z. Wang, G. A. Jullien, and W. C. Miller. An algorithm for multiplication modulo $2^n - 1$. In G. Cameron, M. Hassoun, A. Jerdee, and C. Melvin, editors, *39th Midwest Symposium on Circuits and Systems*, pages 1301–1304, Ames, USA, 1997. IEEE Computer Society Press.

[YJC96] C.-C. Yang, C.-W. Jen, and T.-S. Chang. The IC design of a high speed RSA processor. In *IEEE Asia Pacific Conference on Circuits and Systems'96 (APCCAS'96)*, pages 33–36, Seoul, Korea, 1996. IEEE Computer Society Press.

[ZSY93] C. N. Zhang, B. Shirazi, and D. Y. Y. Yun. An efficient algorithm and parallel implementations for binary and residue number systems. *Journal of Symbolic Computation*, 15(4):451–462, 1993.