République Algérienne Démocratique et populaire

Ministère de l'enseignement supérieur et du Recherche Scientifique

Université SAAD DAHLEB De BLIDA

Faculté des sciences. Département de l'informatique

Mémoire de Fin d'Etude

Pour l'obtention du diplôme Master en Informatique

Option: Génie des systèmes informatique (GSI)

Thème:

Opérateurs arithmétiques performants pour la cryptographie à clé publique RSA

Réalisé par :

- *Mlle. MIDOUN Khadidja*
- *Mlle. CHERGUI Nadjah*

Soutenu le : 10 Septembre 2013.

Devant le jury composé de :

Dr : **BENNOUAR Djamal, Maître de Conférences A, USDB** Président

Mr : **SIDOUMOU Mohamed Réda, Maître Assistant A, USDB** Examinateur

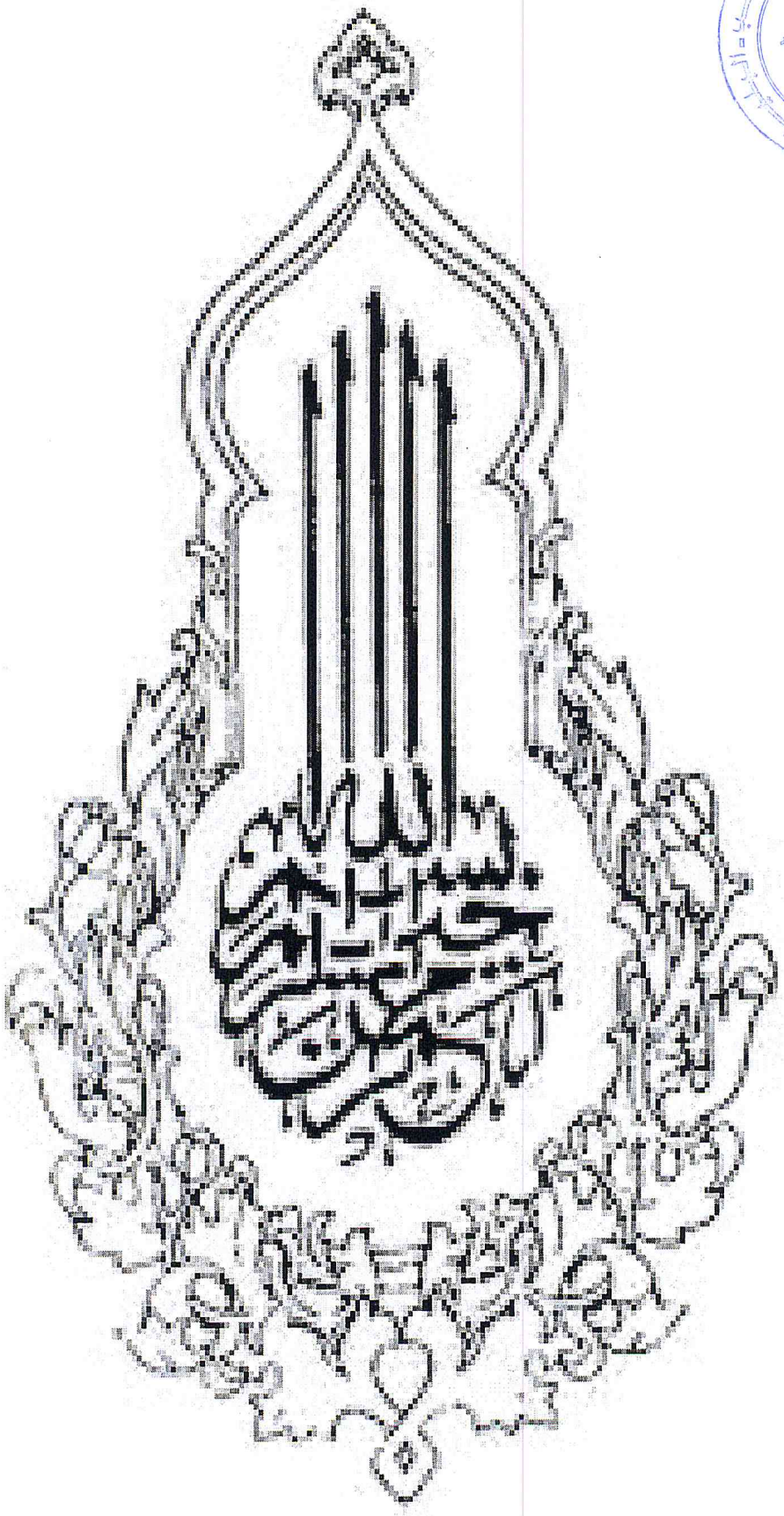
Mr : **BAOUYA Abd Elhakim, Maître Assistante B, USDB** Examinateur

Dr : **ANANE Mohamed, Maître de Conférences B, ESI** Promoteur

Organisme d'accueil :

CDTA – Centre de **D**éveloppement des **T**echnologies **A**vancées

Promotion : 2012/2013



Résumé:

Le cryptosystème asymétrique RSA est basé sur l'opération d'exponentiation modulaire, Celle-ci n'est rien d'autre qu'une suite de multiplications modulaires. Un des moyens d'augmenter les performances de chiffrement/déchiffrement RSA est l'augmentation des performances de la multiplication modulaire qui est au cœur du RSA. Ce PFE est alors consacré à la proposition de solutions matérielle pour l'augmentation des performances de la multiplication modulaire. Dans ce travail les algorithmes de Montgomery et de Barrett sont étudies et des versions sur une grande base sont présentés. Où une réduction du nombre d'itérations est obtenue au détriment d'une augmentation de la complexité dans l'itération. Cette complexité a été contournée par l'utilisation d'une arithmétique redondante.

La méthode bipartite et sa généralisation multipartite sont présentées dans ce travail. Ces méthodes permettent de paralléliser le calcul de la multiplication modulaire. La méthode quadripartite représente un bon compromis (performances/ressource utilisées). Cette dernière a été adaptée aux algorithmes de Montgomery et de Barrett et une architecture performante pour le calcul de la multiplication modulaire 1024×1024 bits en utilisant seulement un datapath de 512 par 512 bits.

Mots-clés: La Cryptographie, RSA, FPGA, la multiplication modulaire, l'exponentiation modulaire, *Parallélisme,* Montgomery, Barrett.

Abstract:

The RSA public-Key cryptosystem is based on the modular exponentiation operation, this is nothing else than a series of modular multiplications. One way to increase the performance of encryption / decryption RSA is the increase performance of the modular multiplication. The PFE is devoted to the proposed hardware solutions for to increase the performance of the modular multiplication. In this work algorithms Montgomery and Barrett are studying and versions on a large database are presented. Where a reduction in the number of iterations is achieved at the expense of increased complexity in the iteration, his complexity has been circumvented by the use of redundant arithmetic.

The bipartite method and multipartite generalization are presented in this work. These methods allow to parallelize the computation of modular multiplication. The quadripartite method represents a good compromise (performance / resource use). The latter was adapted algorithms Montgomery and Barrett and an efficient architecture for computing the modular multiplication 1024 ×1024 bits using only a datapath of 512 by 512.

Key words:

Cryptography, RSA, FPGA, modular multiplication, Modular exponentiation, Parallelism, Montgomery, Barrett.

ملخص

خوارزمية التشفير بواسطة المفتاح العمومي RSA تعتمد على عملية الأس الترديدي و التي تعتبر نتيجة توالي عمليات الضرب الترديدي.

من الطرق المستعملة لرفع أداء خوارزمية RSA هي زيادة فعالية الضرب الترديدي. حيث أن عملنا هذا يندرج في ضمن الأعمال التي تهتم بهذه الطريقة. و من أجل ذلك هو يقترح حل عملي (قابل للتنفيذ على دارة منطقية البرمجة FPGA ل Xilinx) لتحسين أدائها. ومن أجل هذا فقد قمنا بدراسة معمقة لخوارزمية Montgomery و خوارزمية Barrett و هما الأكثر استعمالا في هذا المجال. و لأقد اعتمدنا عليهما من أجل اقتراح خوارزميتنا التي تعتمد على استعمال قاعدة كبيرة التي تعتبر الحل الأمثل لتقليص عدد العمليات ولكنها تقوم بزيادة تعقيد كل عملية و من أجل التخلص من هذا التعقيد قمنا باستعمال الأعداد الممثلة في نظام الحسابيات المكررة (arithmétique redondante).

و لضمان فعالية أكثر لخوارزميتنا قمنا بدراسة طرق متعددة لرفع الأداء و التي منها طريقة ثنائية القطع و متعددة القطع و اللتان تستعملان من أجل الموازاة في عملية الضرب الترديدي. حيث أن من دراستنا لهما وجدنا أن الطريقة التي تستعمل الأربع قطع هي الحل الأنسب الذي يجمع يوافق بين المساحة و الزمن اللازم للتنفيذ حيث تقوم بالموازاة بين خوارزمية Barrett و خوارزمية Montgomery مع استعمالها لبنية عملية حيث تقوم بتقليص طول البيانات من 1024×1024 إلى 512×512 .

مصطلحات أساسية : علم التشفير, التشفير بالمفتاح العام , خوارزمية اراس اي ,الضرب الترديدي , الأس الترديدي , خوارزميات الموازاة , خوارزمية مونغومري, خوارزمية بارات

Remerciements

Nous remercions tout d'abord, notre vénéré Allah, Le tout puissant. à qui nous devons le tout.

Dans un premier temps, nous voudrons remercier Monsieur ***BENNOUAR Djamal,*** *Maître de Conférences à l'université USDB, qui nous fait l'honneur de présider le jury de ce mémoire.*

Nous remercions également Monsieur ***SIDOUMOU Mohamed Réda,*** *et Monsieur* ***BAOUYA Abd Elhakim*** *des Maître Assistant dans l'université USDB d'avoir accepté de juger ce travail de mémoire.*

Nous tenons à exprimer notre gratitude à nos responsables de recherche Monsieur ***ANANE Mohammed*** *maître assistant à l'école supérieur d'informatique et Madame* ***ANANE Nadjia*** *chargé de recherche au CDTA pour avoir dirigés les travaux de recherche.*
On les remercie également pour ses précieux conseils.

Nos sincères remerciements sont adressés à Madame ***CHERID Nacera*** *Directrice des études dans l'école supériure d'informatique ESI, qui nous permet d'accéder à cet école.*

Nous saisissons l'occasion pour remercier tout le corps professoral et administratif de L'USDB.

En fin, Nous remercions, toutes les personnes qui ont contribué de près ou de loin à la réalisation de ce travail.

Dédicaces

D'abord, je voudrais remercier ceux qui m'aiment et que j'aime sans limite, mes chers parents qui étaient toujours à mes cotés et m'avaient tant aidé et soutenu. Qu'ils trouvent ici l'expression de ma sincère gratitude et ma profonde reconnaissance.

Je voudrais le remercier aussi spécialement mon cher frère Ilyes et mes chères sœurs Amina, Sara, Merième, Assia , Merci pour leur amour, leur tendresse et leur confiance en moi.

Je dédie ce modeste travail aux membres de cette famille,

À mes chères amies qui me partagent l'ambition et l'enthousiasme Hoccine Batoul , Tissli Khadidja qui ont sacrifié une partie de leur temps pour m'aider.

À khadidja, Fida et à tous mes camarades du département de l'informatique pour leur bonne humeur et leur sympathie.

À toute personne ayant contribué à ce travail de près ou de lois.

Je ne saurais oublier de remercier chaleureusement mon binôme Chergui Nadjah pour ces années partagées, tant au niveau scientifique qu'au niveau personnel, Elle est vraiment une amie très proche à moi.

MIDOUN Khadidja

Septembre,2013

Dédicaces

C'est avec une immense fierté que je dédie ce modeste

travail fruit de mes études : à ceux qui ont sacrifié leur vie pour ma réussite,

à ceux qui ont fait de moi ce que je suis aujourd'hui, grâce au leurs tendresse et leurs amour : Mes parent : Merci maman, Merci papa.

À les belles filles : mes sœurs :

-Dida, Naima, Saliha, Amel : Merci pour vos aides, vos patients, et vos amours

À mes chères frères : Yacine, Yousef.

À mes amies : Abla, Hamida, Sonia, widad.

Je tiens à remercie ma chère binôme Khadidja pour ses efforts et sa compréhension.

je dédie ce travail aussi à tout la famille Midoun, à ma chère amie Batoul, Sarah, khadidja et toute mes amies de la fac et tout la promotion de 2012-2013.

A toute personne qui m'a aidé de prés ou de loin.

Nadjah Chergui

Liste des acronyms

Lisle des acronyms

AES	*Advanced Encryptions Standard*
ASIC	*Applicatrion-Specific Integrated Circuit*
DES	*Data Encryptions Standard.*
FPGA	*Field Programmable Gate Array*
ISE	*Integrated Software Environment*
LUT	*Look up table*
MMM_SNR	*Multiplication modulaire de Montgomery en System de numération redondant.*
MMB_SNR	*Multiplication modulaire de Barrett en System de numération redondant.*
RSA	*Rivest Shamir Adelman.*
VHDL	*Very High Speed Integrated Circuits Hardware Description Language*
	Virtual Privat Network (Réseaux privés virtuels).
SNR	*System de numération redondant*

Table des matières

Table des matières

Liste des tableaux et figures

1. liste des figures :

2. liste des tableaux

Introduction Générale

Introduction générale

L'apparition des réseaux d'internet permet à tous les utilisateurs d'envoyer et de recevoir des données. Parmi ces données, certaines peuvent présenter une sensibilité importante. Leur perte, leur saisie ou leur vol peut avoir des conséquences importantes sur l'activité de l'émetteur ou le récepteur. C'est le rôle de la cryptographie de garantir la sécurité de ces données.

La plus ancienne cryptographie est la cryptographie symétrique qui repose sur la communication secrète préalable de clé de secrète qui est utilisée dans les deux algorithmes de chiffrement et de déchiffrement. Ce type de cryptographie est connu par le *« problème de distribution de clés »*, La découverte de la cryptographie à clé publique a finalement offert une belle solution pour ce problème en permettant à deux personnes de communiquer confidentiellement sans aucun secret préalable.

Cette solution requiert cependant une grande puissance de calcul puisqu'elle se base sur la difficulté calculatoire d'un certain problème mathématique telle que l'exponentiation modulaire ou le logarithme discret.

La première réalisation concrète d'un cryptosystème asymétrique appelé RSA . Le RSA est basé sur deux problèmes mathématiques difficiles : la factorisation de grands nombres premiers et l'exponentiation modulaire qui se réalise en une suite de multiplications modulaires. Le nombre de ces multiplications modulaires dépend de la taille de la clé publique.

Durant ces dernières années, la recherche de méthodes efficaces dédiées à la factorisation a connu des avancées importantes, à tel point que la taille des clés publique et privée doit augmenter régulièrement afin que l'utilisation du RSA reste sûre. De nos jours, il est plutôt conseillé d'utilise une clé de 1024 bits pour les données commerciales et des clés de 2048 bits (ou plus) pour les données les plus sensibles.

Ce que nécessite l'utilisation de milliers de multiplications modulaires pour un seul opération d'exponentiation modulaire où La complexité de la méthode classique de la multiplication est $O(n^2)$. Par exemple, si on veut multiplier deux entiers de 1000 chiffres, on a besoin de 1 000 000 d'opérations. Il est donc important d'accorder un soin particulier à la réalisation matérielle de cette opération.

C'est la raison pour laquelle les algorithmes de multiplications sont largement et intensivement étudiés pour réduire la complexité de cette opération coûteuse.

La plupart des algorithmes de multiplications modulaires s'exécutent en itérations, l'objectif de ce mémoire est le développement d'opérateurs arithmétiques pour la multiplication modulaire qui permettra d'une part de réduire le nombre de d'itérations et d'autre part réduire la complexité calculatoire dans l'itération.

Pour réaliser ce projet de fin d'étude qui nous a été proposé au sein de l'équipe AC2 (Architecture pour la Compression et la Cryptographie) de la division Architecture des Systèmes et Multimédia (ASM) au Centre de Développement des Technologies Avancées (CDTA), dont le thème est le suivant : « Opérateurs arithmétiques performants pour la cryptographie à clé publique RSA », nous avons organisé notre mémoire en cinq chapitres répartis comme suit :

- ***Chapitre I*** : *la généralité sur la cryptographie*: nous allons présenter dans un premier temps le domaine de la cryptographie et les concepts fondamentaux nécessaires à la compréhension du fonctionnement du cryptosystème RSA
- ***Chapitre II***: " *la multiplication modulaire*"**:** Dans ce chapitre nous allons présenter l'état de l'art de la multiplication modulaire où nous présentons les algorithmes les plus utilisées pour cette opération.
- ***Chapitre III : "****Conception" D*ans ce chapitre nous allons détailler la conception de notre architecture et les blocs qui la composent.
- ***Chapitre IV*** : *"Résultats de simulation et d'implémentation ",* Ce chapitre décrit le détail de simulation et d'implémentation de notre approche, avec quelques tests pour valider notre réalisation. Et nous terminons ce mémoire par une conclusion générale et quelques perspectives.

Chapitre I:

Généralités sur la cryptographie

I.1. Introduction

Aujourd'hui, avec le développement d'internet, transmettre des informations confidentielles de façon sécurisée est devenu un besoin primordial. C'est le rôle de la cryptographie de protéger ces informations et d'assurer leur confidentialité. Aussi, bien qu'il s'agisse d'une science très ancienne, la cryptologie est toujours d'actualité.

Dans ce chapitre, nous présenterons des généralités sur la cryptographie et de ses deux types, à savoir la cryptographie symétrique et asymétrique pour pouvoir comprendre le crypto-système asymétrique RSA en axant sur son opération clé qui est la multiplication modulaire.

I.2. Qu'est-ce la cryptographie

Le mot cryptographie vient des deux mots grecs κρυπτος « Kryptos » et γραφειν « graphien » qui signifient respectivement « caché » et « écrire ».

On peut définir la cryptographie comme l'ensemble des techniques permettant de **chiffrer** des messages, c'est-à-dire permettant de les rendre inintelligibles sans une action spécifique. Puis à partir de ces messages codés, on restitue les messages originaux.

La cryptologie est essentiellement basée sur l'arithmétique où il s'agit de transformer les lettres du texte d'un message en chiffres puis faire des opérations arithmétiques sur ces chiffres pour:

- D'une part les modifier de telle façon à les rendre incompréhensibles. Le résultat de cette modification (le message chiffré) est appelé **cryptogramme** ou (*ciphertext)* en anglais, par opposition au message initial, appelé *message en* **clair** ou (*plaintext) en* anglais.
- Et d'autre part, faire en sorte que le destinataire saura les déchiffrer.

Le codage d'un message de telle sorte à le rendre secret s'appelle *chiffrement*. La méthode inverse, consistant à retrouver le message original, s'appelle *déchiffrement*.
Le chiffrement se fait généralement à l'aide d'une *clé de chiffrement*, le déchiffrement nécessite quant à lui une *clé de déchiffrement*.

On appelle *décryptage* le fait d'essayer de *déchiffrer illégitimement* le message. Lorsque la clé de déchiffrement n'est pas connue de l'attaquant on parle alors de **cryptanalyse**.

La **cryptologie** est la science mathématique qui étudie les aspects scientifiques de ces techniques, c'est-à-dire qu'elle englobe *la cryptographie* et *la cryptanalyse*.

La figure I.1 représente le cryptage, décryptage et cryptanalyse.

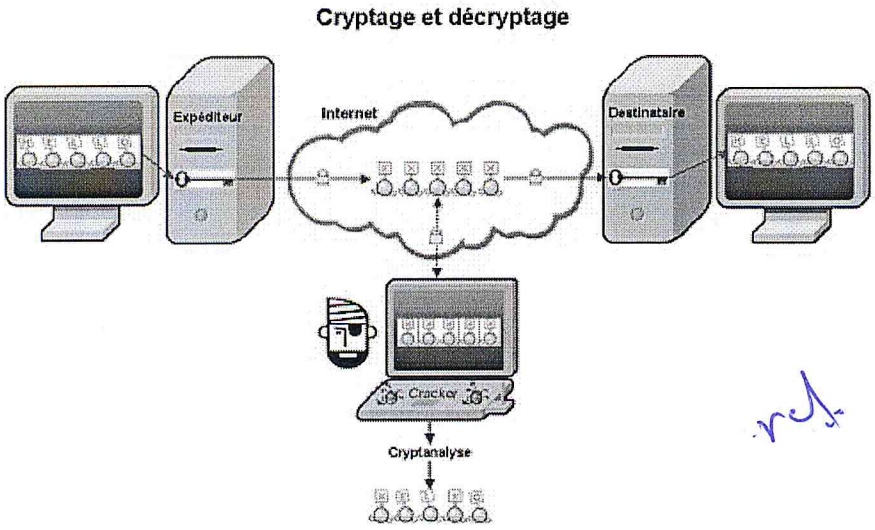


Figure I.1- Cryptage, décryptage et Cryptanalyse

I.3. Les fonctions de la cryptographie

Si le but traditionnel de la cryptographie est d'élaborer des méthodes permettant de transmettre des données de manière confidentielle, la cryptographie moderne s'attaque aux problèmes de sécurité des communications. Son but est d'offrir un certain nombre de services de sécurité comme la confidentialité, l'intégrité et l'authentification des données transmises **[Zab, 12]**.

- **La confidentialité** signifie que seules les personnes ayant droit d'accès à l'information peuvent accéder.
- **L'intégrité** permet de garantir la protection d'un fichier contre toutes modifications par un tiers, c.-à-d. que le fichier reçu est identique à celui qui a été envoyé.
- **L'authentification** permet de prouver l'identité d'une personne ou l'origine d'une donnée.
- **La non-répudiation** assure que l'émetteur ne peut pas nier qu'il ait envoyé un message, c.-à-d. validité de la signature.

I.4. Types de la cryptographie :

Il existe deux types de cryptographie selon l'algorithme utilisé : la cryptographie symétrique et la cryptographie asymétrique.

1. **La cryptographie symétrique :** aussi appelée cryptographie à clé privée (ou à clé secrète) consiste à utiliser la même clé pour le chiffrement que pour le déchiffrement.

La figure I.2 représente le processus de la cryptographie symétrique.

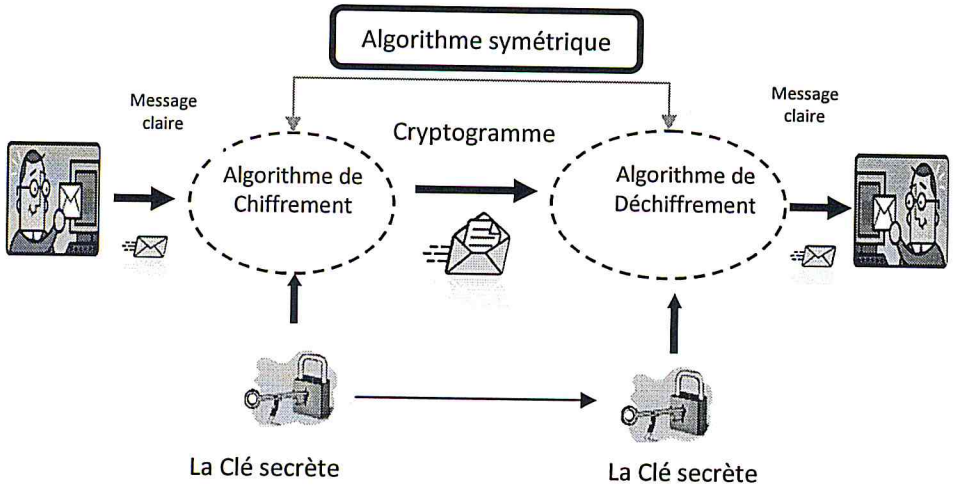


Figure I.2- Le Chiffrement symétrique

Le chiffrement consiste alors à effectuer une opération entre la clé privée et les données à chiffrer afin de rendre ces dernières inintelligibles.

L'avantage de ce type de chiffrement est qu'il est très rapide car il utilise des opérations simples telles que des transformations, des additions et des décalages.

Le protocole de chiffrement symétrique le plus utilisé est l'AES qui a remplacé son prédécesseur le DES qui n'est plus utilisé aujourd'hui, car d'une part il est trop lent et d'autre part, vu la puissance de calcul des ordinateurs actuels, il est devenu relativement facile de être attaqué par recherche de sa clé de 56 bits qui est considérée comme très courte.

En 2000, l'AES (Advanced Encryption standard) remplace le DES avec des blocs de 128 bits et des clés de 128, 192 ou 256 bits. Il est alors plus rapide et très utilisé.

Claude Shannon démontra que pour être totalement sûr, les systèmes à clés privées *doivent utiliser des clés d'une longueur au moins égale à celle du message à chiffrer.*

De plus le chiffrement symétrique impose d'avoir ce qui dégrade sérieusement l'intérêt d'un tel système de chiffrement.

Le principal inconvénient d'un crypto-système à clé secrète provient de d'une part de la disposition d'un *canal sécurisé* pour l'échange de la clé et d'autre part de la gestion des clés pour un un groupe d'utilisateurs utilisant ce crypto-système.

Pour un groupe de n personnes utilisant un crypto-système à clé secrète, il est nécessaire de distribuer $n \times (n-1) / 2$ clés.

Avec l'évolution de réseau et l'augmentation de nombres d'utilisateurs, les développeurs cherche une réponse efficace pour la question suivante :

« *Comment échanger les secrets de manière fiable en particulier à l'heure actuelle où des milliards d'humains cherchent à communiquer ?* ».

2. La cryptographie asymétrique

L'algorithme de chiffrement à clé publique est apparu en 1976, avec la publication d'un l'article *:" New Direction in cryptographie "* par *Whitfield Diffie* et *Martin Hellman.* Ils proposaient d'utiliser deux clés pour chaque entité de communication :

- Une clé publique pour le chiffrement : publiée dans un annuaire.
- Une clé secrète pour le déchiffrement : qu'il est seul à connaître.

La figure I.3 représente le processus de la cryptographie asymétrique

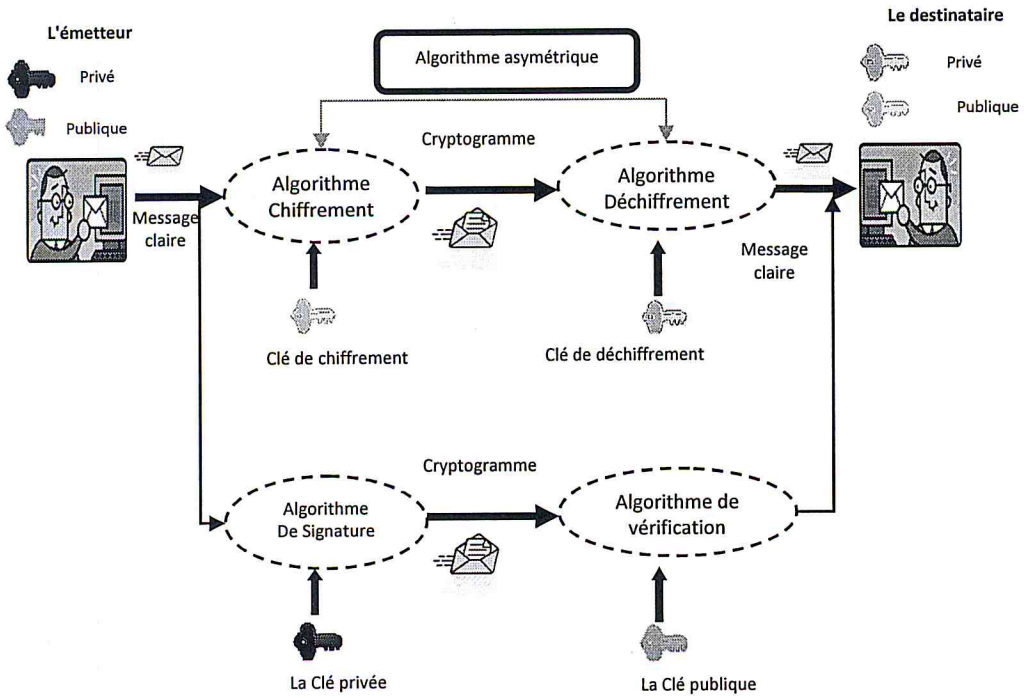


Figure I.3- Le Chiffrement asymétrique

Un chiffrement asymétrique est un cryptage où l'algorithme de chiffrement n'est pas le même que celui de déchiffrement, et où les clés utilisées sont différentes. L'intérêt est énorme : il n'y a plus besoin de transmettre la clé à son destinataire, il suffit de publier librement les clés de cryptage. N'importe qui peut alors crypter un message, mais seul son destinataire, qui possède la clé de décodage, pourra le lire.

Ce système est basé sur une fonction facile à calculer dans un sens (appelée *fonction à trappe à sens unique*), mais qui est mathématiquement très difficile à inverser sans la clé privée (appelée *trappe*).

L'un des principaux avantages de la cryptographie de clé publique est qu'elle offre une méthode d'utilisation des *signatures numériques*. Celles-ci permettent au destinataire de vérifier leur authenticité, leur origine, mais également de s'assurer qu'elles sont intactes. De plus, la signature électronique est créée de manière à ce que n'importe qui puisse en vérifier la validité.

Pour effectuer une signature, le signataire applique au message à signer un algorithme de signature en utilisant sa clé privée. Le résultat de cet algorithme peut être vérifié par n'importe quelle personne possédant la clé publique du signataire en utilisant l'algorithme de vérification correspondant.

Les cryptographies symétriques et asymétriques ont des avantages et des inconvénients. Les systèmes symétriques sont rapides mais nécessitent le partage d'un secret (la clé) de chaque couple d'interlocuteurs via un canal sécurisé. La gestion de ces clés devient vite problématique.

Dans les systèmes asymétriques, le problème d'échange préalable de clé ne se pose plus et le nombre de clés nécessaires est seulement d'une paire de clés par utilisateur soit le double du nombre d'utilisateurs, ce qui rend leur gestion plus aisée. Mais leur inconvénient est la lenteur du calcul du moment que la taille de la clé doit être au mois de 1024 bits.

Actuellement, les cryptographies symétrique et asymétrique sont combinées pour former un système hybride exploitant les points forts de chaque type : la sécurité de l'asymétrique pour échanger de manière sécurisée la clé et la rapidité du symétrique pour chiffrer/déchiffrer les messages de grande taille en un temps raisonnable **[Ccd ,08].**

I.5. Le Crypto système RSA

Suite à l'article de Diffie-Hellman, les trois chercheurs *Ron Rivest, Adi Shamir et Len Adlmen* avaient décidé de travailler ensemble pour établir un nouveau système de cryptographie basé sur l'approche de Diffie-Hellman**[Des, 09].**

En 1978, ils ont publié dans la revue mensuelle de« association of computing Machinery ACM » un article *:* « ***une méthode efficace pour obtenir des signatures électroniques et clé publique de cryptosystème*** ». Elle s'appelle ***RSA 'Rivest Shamir Adlmen'.***

Le RSA est encore le système cryptographique à clé publique le plus utilisé de nos jours. Il sert de base à la sécurité dans beaucoup de systèmes d'information numériques tels que les applications de réseaux comme : l'échange des courriers électroniques, le commerce électronique et les opérations bancaires électroniques qui se fondent sur des services comme les signatures numériques pour assurer la confidentialité, l'authentification et l'intégrité des données.

Il est important de noter que cet algorithme sert encore à protéger les codes nucléaires de l'armée américaine et russe.

L'algorithme RSA fonctionne avec une paire de clé unique à générer : une clé publique *(e, N)* pour le chiffrement et une clé privée *(d, N)* pour le déchiffrement où N est le modulo.

I.5.1. Génération des clés

Pour générer les deux clés publique et privée on doit suivre les étapes suivantes :

a. Prendre 2 nombres premiers très grands : **p** et **q**
b. Calculer **M = p x q**
c. Calculer la fonction indicatrice d'Euler **z = Φ(M) = (p-1) x (q-1)**
d. Prendre un nombre **e** premier avec z tel que **le PGCD (z,e) = 1** avec **1 < e < z**
e. Calculer **d** tel que **d x e = 1 mod z**
f. La clé *publique* correspond au couple : **{e,M}**
g. La clé *privé* correspond au couple : **{d, M}**

On considère un message A à transmettre. Ce message A est un nombre entier, par exemple un texte codé à l'aide du code ASCII.

I.5.2. Chiffrement

Pour crypter un message A, l'émetteur cherche dans l'annuaire la clé publique du destinataire. Il découpe son message chiffré en blocs de même longueur représentant chacun un nombre plus petit que M.

Un bloc M_i est chiffré par la formule :$\mathbf{C_i = A_i^e \bmod M}$ **où** C est un bloc du message chiffré. Comme le modulo *N* et l'exposant *e* sont connus, alors tout le monde peu tchiffrer un message.

I.5.3. Déchiffrement

Pour retrouver le message d'origine M (décrypter), on utilise la même opération, mais en mettant à la puissance **d** (*la clé privée*):$\mathbf{A_i = C_i^d \bmod M}$

Seule, la personne possédant la clé privée de déchiffrement d peut donc déchiffrer le message.

Le principe de fonctionnement du protocole RSA est représenté sur la figure I.4.

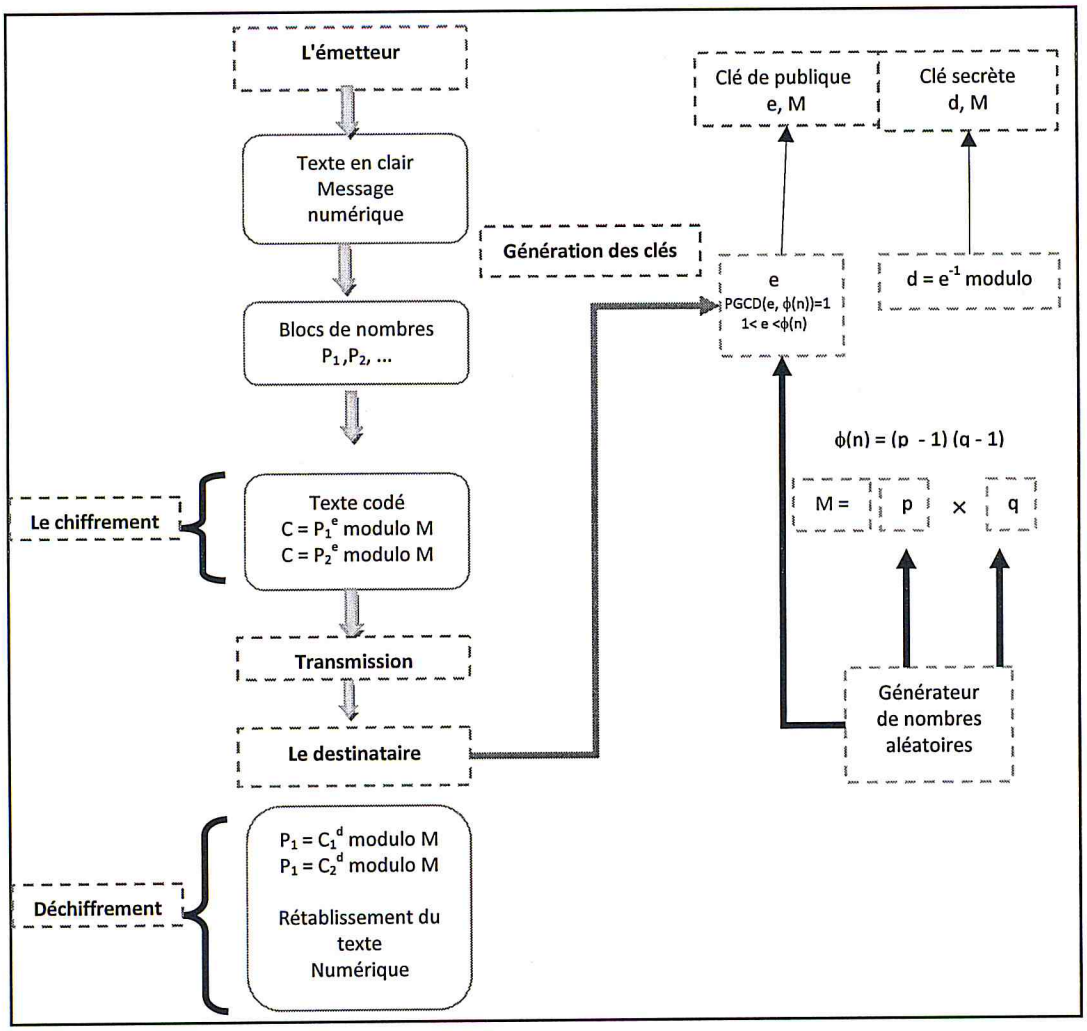


Figure 1.4 : Le principe de Fonctionnement du RSA

I.5.4. La Sécurité du RSA

La confiance dans la sécurité du RSA n'est pas due à la démonstration théorique que ce système est sûr, car une telle démonstration n'existe pas. La confiance affichée provient de l'échec, répété pendant 35 ans**[Nit, 09]** , de toutes les tentatives entreprises pour casser ce système, tentatives qui n'ont conduit qu'à la formulation de quelques recommandations pour le choix des paramètres p, q, e, d.

D'autres systèmes concurrents du RSA peuvent être aussi bons ou même meilleurs, mais aucun ne bénéficie de la validation par l'usage et la robustesse aux attaques dont le RSA peut se prévaloir. Le fait d'avoir été l'un des premiers lui a donné un avantage qui, puisqu'il résiste aux tentatives d'effraction, le maintien en tête.

La solidité du RAS repose uniquement sur la difficulté de décomposer le nombre M (public) en deux nombres premiers. Il faut donc, pour garantir une certaine sécurité, choisir des clés plus grandes : les experts recommandent des clés de 768 bits pour un usage privé et des clés de 1024, voire 2048 bits, pour un usage sensible. Si l'on admet que la puissance des ordinateurs double tous les 18 mois (loi de Moore), une clé de 2048 bits devrait tenir jusqu'en 2079.

I.5.5. Mise en œuvre du RSA :

Même si le protocole RSA est assez simple, sa mise en œuvre pose toutefois quelques problèmes notamment le problème de l'élévation de façon efficace d'un gros nombre à une grosse puissance modulo M. C'est-à-dire calculer M^e mod N pour le chiffrage et C^e mod M pour le déchiffrage connu sous : *l'exponentiation modulaire* qui est une suite de multiplications modulaires suivi par une division qui est une opération très couteuse en terme arithmétique **[Del,00].**

En effet pour assurer un haut niveau de sécurité la longueur de M doit être d'au moins 1024 bits. Avec l'exponentiation modulaire classique le calcul de $C=A^e$ mod M nécessite (e-1) multiplications suivie d'une division, ce qui est infaisable pour des clés de l'ordre de 1024 bits.

Cette méthode produit beaucoup de calculs intermédiaires qui donnent des résultats de calculs de plus en plus longs et qui doivent être stockés. Néanmoins, l'espace mémoire requis pour le stockage de ces résultats et du nombre binaire A^e est énorme.

Une autre complexité est dans la réalisation de la multiplication elle-même celle-ci est liée à la taille du message A à crypter, donc à la taille du modulo, pour des clés de 1024 bits on aura des multiplications de 1024 bits par 1024 bits. L'implémentation parallèle de cette multiplication est hors de portée pour les supports matériels actuels.

Augmenter les performances d'un crypto-système RSA revient donc à réduire d'une part le temps consenti à la réalisation de la multiplication modulaire et d'autre part à réduire le nombre de multiplications modulaires requises par l'exponentiation modulaire.

I.6. Conclusion

Dans ce chapitre, nous avons présenté des généralités sur la cryptographie ainsi que les différents types de cryptographie, à savoir symétrique et asymétrique avec leurs avantages et inconvénients.

D'un autre côté, le RSA qui est l'algorithme de cryptographie à clé publique le plus répandu a été détaillé. Les opérations arithmétiques nécessaires à sa mise en œuvre sont basées essentiellement sur *l'exponentiation modulaire* qui est réalisée par une suite répétée de multiplications modulaires. Alors toute amélioration de la complexité de la multiplication modulaire aura des répercussions positives sur la complexité de l'exponentiation et par conséquent sur le crypto-système RSA.

Dans le prochain chapitre, nous aborderons différentes méthodes pour l'optimisation de la multiplication modulaire.

Chapitre II:
La multiplication modulaire

II.1. Introduction

La multiplication modulaire est l'opération de base du cryptosystème RSA, celle-ci s'exécute plusieurs fois lors du chiffrement ou du déchiffrement d'un message. Cette opération nécessite deux opérations complexes qui sont la multiplication et la division qui est l'opération la plus coûteuse en arithmétique. Et comme la plus part des algorithmes de la multiplication modulaire sont des algorithmes séquentiels qui s'exécutent en itérations, l'augmentation des performances de cette opération passe forcément par soit la réduction du nombre d'itérations ou par l'amélioration du temps d'exécution de l'itération. Ces améliorations sont plus appréciées lors de l'exécution de l'exponentiation modulaire qui est le RSA.

Dans ce chapitre nous allons passer en revue les algorithmes matériels dédiés à cette opération et nous terminons par une comparaison de ces algorithmes.

II.2. La multiplication modulaire

II.2.1. Définition de la multiplication modulaire

La multiplication modulaire consiste à effectuer la multiplication de deux nombres donnés et à prendre le reste (S) du produit obtenu aprés la division par un troisième nombre appelé module. Cette opération est donnée par l'expression :

$$S = X \times Y \pmod{M}. \quad \text{Où } 0 < X, Y < M.$$

Cette opération se décompose donc à priori en deux phases :

- **Une phase de multiplication:** la multiplication entre deux nombres de n bits conduits à un résultat codé sur 2n bits.
- **Une phase de réduction modulaire:** c'est équivalent à une division euclidienne d'un entier P par une constante M connue (appelé modulo) dans laquelle on ne désire récupérer que le reste S : $P = q\,M + S$ avec $0 \leq S < M$.

L'algorithme de la division est le plus compliqué et le plus coûteux en temps de calcul, il consiste à déterminer des quotients partiels, cette opération élémentaire est plus longue que la détermination des produits partiels dans l'algorithme de la multiplication.

En cryptographie RSA, un usage intensif de la réduction modulaire est nécessaire, il est alors globalement trop coûteux de réaliser cette réduction par division.

II.2.2. Les catégories de la multiplication modulaire

Deux approches sont possibles sur la façon de calculer une multiplication modulaire : la multiplication modulaire y est décomposée en une multiplication suivie d'une réduction modulaire. Dans l'algorithme de RSA : $0 < X,Y \leq 2^{n-1} < M < 2^n$. Soit p le produit de ces nombres : $p = X \times Y \; avec \;\; 0 < p < 2^{2n}$. Cela requiert une grande capacité de stockage pour pouvoir conserver les résultats partiels. Il est impossible d'implémenter en hardware des multiplieurs de très grandes tailles. Si une division suffit pour calculer ce quotient, son coût élevé en temps de calcul par rapport à la multiplication modulaire de la deuxième catégorie. C'est pour cela que la majorité des algorithmes de multiplications modulaires appartiennent à la seconde catégorie qui intègre la réduction à la multiplication **[Ber, 07]**.

II.2.3. Les algorithmes de la multiplication modulaire

Les algorithmes de la multiplication modulaire sont largement et intensivement étudiés pour réduire la complexité de cette opération coûteuse. Nous présentons ici les principaux ainsi que leurs complexités.

Dans les algorithmes suivants, on cherche à calculer $S = X \times Y \; mod \; M$ avec : $0 \leq X,Y < M \; et \; 2^{n-1} < M < 2^n$ où n : la taille de modulo M

II.2.3.1. L'algorithme Classique

La multiplication modulaire selon cette méthode est calculée en trois étapes : La première calcule le produit des deux opérandes puis la deuxième calcule le quotient par une division euclidienne, en fin le résultat est obtenu après la soustraction entre le produit et le quotient multiple de M. L'algorithme 2.1 représente cet algorithme.

Algorithme 2.1 : Multiplication Modulaire Classique

Entrées : $X,Y,M \; avec \; 0 \leq X,Y < M, 2^{n-1} < M < 2^n$

Sortie : $S \; avec \; S = X \times Y \; mod \; M$

début

$P = X \times Y$

$q = \frac{P}{M}$

$S = P - q \times M.$

Fin

Complexité:

La complexité de cette opération est égale à deux multiplications de $n \times n$ $bits$ et une seule division de $\frac{2n}{n}$ bit. À cause du coût élevé de la division entière, cet algorithme est rarement utilisé en pratique au niveau logiciel.

II.2.3.2. L'algorithme de Taylor avec mémorisation

Taylor propose un algorithme **[Pla, 05]** qui utilise une décomposition de la multiplication en carrés : $XY \ mod \ M = \left(\frac{(X+Y)^2}{4} - \frac{(X-Y)^2}{4}\right) mod \ M$

$$= \frac{(X+Y)^2}{4} mod \ M - \frac{(X-Y)^2}{4} \ modM$$

Cet algorithme utilise une table mémoire MEM qui effectue cette opération coûteuse. $0 \leq P < M, \quad MEM(P) \leftarrow 4^{-1}P^2(mod \ M)$

Cette décomposition est intéressante uniquement si deux mises au carré modulaire $(P^2 \ mod \ M)$ sont moins couteuses qu'une multiplication modulaire.

L'algorithme 2.2 représente cet algorithme.

Algorithme 2.2 : Multiplication Modulaire de Taylor

Entrées : X, Y, M avec $0 \leq X, Y < M$

Donnés : MEM avec $MEM(x) = (4^{-1} \times x^2) \bmod M$

Sortie : S avec $S = X \times Y \bmod M$

début

$u = |a + b - M|$

$v = |a - b|$

$s = MEM(u) - MEM(v)$

Si $s < 0$ *alors* $s = s + M$

Fin

Évaluation : Cette multiplication modulaire nécessite cinq additions et deux appels mémoires. Mais celle-ci est utilisée pour des petits modulos, le contrecoup de cette méthode est la taille de la table mémoire qui égale $2^n \times$ n bits.

II.2.3.3. L'algorithme de Blakley

Blakley **[Bla, 83]** adapte un "*double and add* " classique en y intégrant des réductions après chaque étape de calcul. Cet algorithme utilise une représentation classique des nombres. Il est représenté dans l'algorithme 2.3.

Algorithme 2.3 : Multiplication Modulaire de Blakley

Entrées: X, Y, M *avec* $0 \leq X, Y < M$

Sortie: $S = XY \bmod M$

début

$S = 0$

pour i ***de*** $n - 1$ à 0 ***faire***

$S \leftarrow 2 \times S$;

Si $S \geq M$ *alors* $S = S - M$ *fin Si*;

$S = S + X_i \times Y \bmod M$;

Si $S \geq M$ *alors* $S = S - M$ *fin Si*;

Fin pour ;

Fin ;

Évaluation :

L'algorithme de Blakley a une complexité en temps de $\mathbf{3} \times \boldsymbol{n}$ additions de $\boldsymbol{n}$ bits : le doublement en base 2 n'est pas compté (C'est un simple décalage).

Il nécessite $\boldsymbol{n}$ itérations, et à chaque itération il intègre la réduction au modulo à l'intérieur de la multiplication, mais il utilise toujours la division qui est l'opération la plus coûteuse en arithmétique. Donc, il n'est pas performant pour la cryptographie RSA.

II.2.3.4. La multiplication modulaire de Montgomery

En 1985, Peter Montgomery a introduit une méthode efficace pour la multiplication modulaire. L'algorithme **[Mir, 09]** remplace la division par M par une division de puissance de 2^n (n est la taille du modulo M en base 2) qui est implémenté tout simplement par des décalages à droite facile à implémenter et moins coûteux en terme de calcul.

La multiplication modulaire de Montgomery MMM ne calcule pas $S = X \times Y \bmod M$, mais une réduction avec un facteur supplémentaire, c'est-à-dire $S = X \times Y \times R^{-1} \bmod M$ où $R=2^n$ et PGCD(R, N)=1.

L'algorithme 2.4 représente cet algorithme.

Algorithme 2.4: Multiplication modulaire de Montgomery

Entrée: X, Y, M **avec** $0 \leq X, Y < M$
Données: M', R^{-1} **et** R **avec** : $R \geq r^n$;
$((-M) \times M') \bmod R = 1$, $R^{-1} \times R \bmod M = 1$
Variables : C, q
Sortie : S **tel que** $S = (X \times Y \times R^{-1}) \bmod M$

Début
$C = X \times Y$
$q = C \times M' \bmod R$
$S = (X + q \times M) \times R^{-1}$
Si $(S \geq M)$ **alors** $S = S - M$
Retourner S;
Fin

II.2.3.4.1. La Représentation de Montgomery

L'entrée dans le domaine de Montgomery utilise *la représentation de Montgomery* $\bar{X}$ qui n'est rien d'autre qu'une multiplication modulaire avec le facteur R.

$\bar{X} = MMM(X, R^2) = X \times R^2 \times R^{-1} \bmod M = X \times R \bmod M$;

Cela peut être réalisé en supposant que $R^2 \bmod M$ soit pré calculé et sauvegardé.

II.2.3.4.2. La Réduction de Montgomery

La Réduction de Montgomery (RM) est la transformation inverse de la représentation de Montgomery dont la définition est donnée par la formule : $RM(u) = u \,.\, r^{-1} \bmod M$ où u est un m-résidu. Cette réduction permettant d'éviter la division par M et qui est le principale avantage de cet algorithme.

La multiplication modulaire $S = X \times Y \bmod M$ est calculée avec cet algorithme en deux étapes : La première calcule La multiplication modulaire de Montgomery des images $\bar{X}$ et $\bar{Y}$ qui donne le résultat :

$$\bar{S} = MMM(\bar{X}, \bar{Y}) = X \times R \times Y \times R \times R^{-1} \bmod M = X \times Y \times R \bmod M$$

La deuxième étape permet de sortir du domaine de Montgomery il utilise l'opération suivante :

$$S = MMM(\overline{S}, 1) = \overline{S} \times 1 \times R^{-1} mod\ M = X \times Y \times R \times 1 \times R^{-1}\ mod\ M = X \times Y\ mod\ M$$

Évaluation : cet algorithme fonctionne pour tous les modulos. Il nécessite, pour le calcul d'une seule multiplication modulaire de faire trois multiplications modulaires. Pour cette raison, il est utilisé dans les opérations qui nécessitent un grand nombre de multiplications modulaires comme l'opération de l'exponentiation modulaire. Il est efficace pour les grands modulos.

II.2.3.5. L'algorithme de Barrett

Barrett propose un algorithme **[Bar, 86]** qui n'utilise pas une représentation particulière, les entiers sont simplement représentés en base r avec r = 2^k. Il approche le quotient $q = \left\lfloor \frac{xy}{M} \right\rfloor$ par un calcul sans division avec l'équation : $q = \left\lfloor \frac{X \times Y}{M} \right\rfloor \approx \left\lfloor \frac{\left\lfloor \frac{X \times Y}{r^{n-1}} \right\rfloor \times \left\lfloor \frac{r^{2n}}{M} \right\rfloor}{r^{n+1}} \right\rfloor$ où

$$q - 2 \leq \left\lfloor \frac{\left\lfloor \frac{xy}{r^{n-1}} \right\rfloor \times \left\lfloor \frac{r^{2n}}{M} \right\rfloor}{r^{n+1}} \right\rfloor \leq q$$

Deux soustractions au plus permettent de corriger le résultat final. L'algorithme 2.5 représente cet algorithme.

Algorithme 2.5: Multiplication Modulaire de Barrett

Entrées: X, Y, M *avec* $0 \leq X, Y < M$

pré-calculé : $\mu = \left\lfloor \frac{r^{2n}}{M} \right\rfloor$

Sortie: S *avec* $S = X \times Y\ mod\ M$

début

$P = X \times Y$

$q = \left\lfloor \frac{\lfloor P/r^n \rfloor \times \mu}{r^n} \right\rfloor$

$S = c - q \times M$

Tant que $S \geq M\ faire\ S = S - M;$ fin

Fin

Évaluation : L'algorithme du Barrett avec ses versions est plus adapté pour des grands modulos, Il utilise une représentation classique des nombres pour faire la soustraction, une valeur pré calculée, l'algorithme ne contient que des multiplications ou des décalages, et évite les divisions coûteuses.

II.2.3.6. Comparaison

Chaque algorithme possède des avantages et des inconvénients qui les rendent tous intéressants mais dans des contextes différents.

Lorsque les applications de la cryptographie RSA requiert des multiplications des grandes opérandes et qui s'exécutent plusieurs fois, il fallait trouver l'algorithme le plus adapté à ces applications, et comme la complexité de la multiplication modulaire dépend du nombre d'itérations et du temps d'exécution d'une itération, on peut comparer les algorithmes de multiplication modulaire selon ces critères.

L'algorithme de Taylor est dédie aux applications qui nécessitent un petit modulo, il utilise une table de grande taille.

L'algorithme de Blakley nécessite n itérations, dont chaque itération, il utilise la réduction modulo en exécutant l'opération de la division qui est l'opération la plus couteuse en arithmétique, ça implique que chaque itération prend un temps important, donc il est dédie aux applications avec de petits modulos.

L'algorithme de Montgomery utilise une représentation particulière, une valeur pré calculée qui prend un certain temps mais constitue un avantage si l'algorithme s'exécute plusieurs fois comme dans l'exponentiation modulaire. L'algorithme de Montgomery réduit le temps d'une itération car il élimine l'opération de division par le modulo M (grand taille) et la remplace par un simple décalage, donc il est dédie aux applications avec de grands modulos.

L'algorithme de Barrett nécessite n itérations avec une représentation des nombres classiques, il n'utilise que des multiplications et des décalages et des soustractions, il élimine la division couteuse.

L'algorithme de Barrett et celui de Montgomery sont en concurrence direct : ce sont tous deux des algorithmes de réduction généralistes avec pré-calculs. L'algorithme de Montgomery est certes plus rapide mais il nécessite l'utilisation d'une représentation particulière. Il parait judicieux de conseiller l'algorithme de Barrett dans le cadre d'une

simple réduction et de proposer l'algorithme de Montgomery, plus rapide, dans le cadre de calcul plus nombreux (exponentiation modulaire par exemple).

Le surcoût d'un pré-calcul est moins important que le surcoût d'une représentation. Dans le cas d'une exponentiation modulaire par exemple, la représentation donne un surcoût à chaque exponentiation alors que le pré-calcul ne coûte qu'une seule fois pour toutes les exponentiations sur le même modulo.

Dans la suite de ce chapitre nous présentons les améliorations qui touchent ces algorithmes et qui améliorent bien sûr la multiplication modulaire.

II.3. La multiplication modulaire de Montgomery

L'algorithme de multiplication de Montgomery est considéré pour être l'algorithme le plus rapide pour compter $X \times Y \; mod \; M$, lorsque les valeurs de X, Y et M sont très grandes. Dans cette partie, on présentera les versions les plus récentes de l'algorithme de Montgomery.

II.3.1. L'algorithme de Montgomery en base 2

Dans cet algorithme les opérandes X, Y, M sont présentés sous format binaire (base 2), il consiste à lire les bits de Y : bit par bit et les multiplier par X, en ajoutant le résultat précédent $(S_i + Y_i \times X)$, l'avantage de cette variante est que le produit de $(Y_i \times X)$ est effectué par l'opérateur logique *AND*. Pour une implémentation matérielle cette opération est moins coûteuse en termes de temps d'exécution qu'une multiplication de deux mots.

L'algorithme 2.6 représente cet algorithme.

Algorithme 2.6 : l'algorithme de Montgomery en base 2

Entrée: $X = (X_{n-1}, \dots, X_0)_2, Y = (Y_{n-1}, \dots, Y_0)_2, \; M = (M_{n-1}, \dots, M_0)_2$

Sortie: $S = X \times Y \times R^{-1} mod \; M$

Début $S_0 = 0$;

Pour i **de** 0 **à** $n-1$ ***faire***

$q_i = (S_i + Y_i \times X) \; mod \; 2$;

$S_{i+1} = (S_i + q_i \times M)/2$;

fin **pour ;**

Si $S \geq N$ ***alors*** $S = S - M$ ***fin si;***

Fin;

Mais, l'utilisation de systèmes de numération binaires pose des problèmes en terme de temps de calcul : pour n=1024 on a besoin 1024 itérations $Nmb_{iteration} = 1024$ où chacun de ce dernier exécute deux opérations d'addition entre un nombre et un produit de deux nombres. Donc le temps d'exécution d'une itération noté $T_{iteration}$égale :

$T_{iteration} = 2 \times T_{addition} + T_{multiplication}$ où $T_{multiplication} = T_{AND}$

alors $T_{iteration} = 2 \times T_{addition}$

Le temps d'exécution d'une seule multiplication modulaire « T_{MM} » par l'algorithme 2.6 est égal: $T_{MM} = 1024\ T_{cycle\ d'horloge} = 2048\ T_{addition}$.

Augmenter la rapidité de cet algorithme revient donc à réduire d'une part le temps consenti à la réalisation d'une opération à l'aide de *la représentation redondante* et d'autre part à réduire le nombre de multiplications modulaires : diminuer le nombre d'itérations à l'aide de la *représentation grande base.* Où plusieurs méthodes sont proposées dans la littérature. On commence par la première approche :

II.3.2. Réduire le temps d'opération (représentation redondants)

L'algorithme 2.6 montre que la multiplication modulaire pouvant être assimilée à une série d'additions et de décalages. Donc son opération de base est *l'addition*, le problème de cette opération est la *propagation de la retenue (complexité O(n))* qui est directement proportionnelle à la longueur des opérandes.

La représentation redondante des nombres permet d'éliminer le problème de la propagation de retenu survenant lors de l'addition. Cette classe de notations autorise plusieurs représentations possibles pour certains nombres, ce qui peut permettre d'accélérer le temps de calcul de l'addition car cette dernière peut s'exécuter à temps constant. Cette propriété est d'autant plus intéressante lorsqu'il est calculé plusieurs additions successives.

La solution matérielle de ce problème est l'utilisation L'additionneur carry save, c'est simplement un ensemble de **n** *Full-Adder* exécutent en parallèle. Des versions de l'algorithme de Montgomery adapté à la représentation redondante ont été déjà proposées

II.3.2.1. L'Algorithme de Montgomery avec carry save adder (fast)

Cet algorithme **[Koo,08]** consiste à utiliser l'additionneur a retenu conservé *(en anglais Carry Save Adder)* : les entiers en base 2 sont représentés en utilisant les chiffres 0, 1 et 2. Chacun de ces chiffres est représenté par deux bits dont il est la somme.

Par exemple : (1,1) (1,0) (0,0) (1,0) = (1+1) × 2^3+ (1+0) × 2^2+ (0+0) × 2^1+ (1+0) × 2^0

Cette représentation nécessite un doublement du matériel, mais elle rend l'addition plus rapide. L'algorithme 2.7 représente cet algorithme.

Algorithme 2.7: Faste Montgomery multiplication

Entrées: X, Y, M où $0 \leq X, Y < M$

Sortie: P avec $P = X \times Y \times 2^{-1} \ mod \ M$

début

S =0; C =0;

Pour i **de** 0 **à** $n-1$ ***faire***

$S, C = S + C + Y_i \times X;$

$S, C = S + C + S_0 \times M;$

$S = \frac{S}{2}; \quad C = \frac{C}{2};$

Fin pour;

$P = S + C;$

Si ($P \geq M$) alors $P = P - M$; fin Si ;

Fin;

L'addition classique de l'algorithme 2.6 dont la complexité est O(n) a été remplacée par une addition de Carry Save Adder de la complexité O(1).

Dans cet algorithme S et C représentent la somme et la retenue respectivement. Bien sûr, les additions qui ne sont pas dans la boucle (**pour**) sont des additions classiques. Mais comme ils ne sont exécutés qu'une seule fois tandis que les additions dans la boucle sont exécutées **n** fois permet de réduire le temps d'exécution.

Cet algorithme utilise 2 Carry Save Adder dans chaque itération, pour des grands opérandes (n=1024 bit) on a besoin de beaucoup de ressources matérielles pour implémenter cette architecture, l'optimisation de cet algorithme a un but de réduire le nombre d'additions.

Une version plus développée permet de réduire le nombre d'additions dans la boucle.

II.3.2.2. algorithme de Montgomery plus rapide (Faster Montgomery Multiplication)

Cet Algorithme (comme illustre l'algorithme 2.8) consiste à éliminer la deuxième addition dans la boucle ce que nécessite de trouver tous les valeurs possibilités de $S_0 \times M$. (on note *val* $= S_0 \times M$ et *som* $= S + C$):

Si som est *pair* et $Y_i = 0 \rightarrow val = 0$.

Si som est *impair* et le $Y_i = 0 \rightarrow val = M$.

Si som est *pair* et le $Y_i = 1, X_0 = 1 \rightarrow val = Y$;

Si som est *pair* et le $Y_i = 1, X_0 = 0 \rightarrow val = Y + M$;

Algorithme 2.8: algorithme de Montgomery plus rapide

Entrées: X, Y, M où $0 \leq X, Y < M$

Sortie: P avec $P = X \times Y \times 2^{-n} \bmod M$

début

$E = X + M$; $S = 0$; $C = 0$;

Pour i **de** 0 **à** $n - 1$ ***faire***

Si $(S_0 = C_0)\ ET\ not(Y_i))$ ***alors*** $I = 0$; ***fin Si***

Si $(S_0 \neq C_0)\ ET\ \overline{Y_i})$ ***alors*** $I = M$; ***fin Si***;

Si $(\ not(S_0 \oplus C_0 \oplus X_0)\ ET\ Y_i\ )$ **alors** $I = X$; ***fin Si***;

Si $(\ (S_0 \oplus C_0 \oplus X_0)\ ET\ Y_i\ )$ ***alors*** $I = E$; ***fin Si***;

$S, C = S + C + I$;

$S = \frac{S}{2}$; $C = \frac{C}{2}$;

fin pour;

$P = S + C$;

Si $(P \geq M)$ **alors** $P = P - M$; **fin Si ;**

Fin;

Cet Algorithme a l'avantage d'utiliser une seule addition au lieu de deux avec un stockage de 4 valeurs 0, M, X et E.

II.3.3. Multiplication modulaire sur une grande base

La représentation dans une grande base est une généralisation de la méthode binaire. Elle consiste à étendre la base 2 à une grande base plus grande qui est multiple de 2. Elle remplace le travail avec des bits par le travail avec des blocs de taille $\boldsymbol{k}$ bits. .

Le principal avantage d'utilisation de cette représentation est que le nombre d'itérations diminue néanmoins la complexité calculatoire de l'itération augmente. Il est important d'assurer que cette complexité ne ramène pas un délai total plus important que celui en base 2.

Un algorithme utilisant une grande base est plus rapide qu'un algorithme en base 2 si et seulement si : $\Delta_k = (n/k) \times T_k < \Delta_2 = n \times T_2$. Cet algorithme offre une façon d'augmenter les performances de la multiplication modulaire, il diminue le nombre d'itérations néanmoins la complexité de l'itération augmente par l'introduction d'une opération de multiplication au lieu d'un AND dans la version en base 2. Ceci permet d'effectuer des multiplications modulaires sur des grands nombres (1024 bits) en utilisant des opérateurs sur 18 ou 32 bits par exemple (Mul32 et Add32). L'algorithme 2.9 utilise des mots de taille $\boldsymbol{k} : \boldsymbol{r} = \boldsymbol{2^k}$ avec un nombre d'itérations d : $\boldsymbol{d = n/k}$.

Algorithme 2.9 : L'algorithme de la Multiplication Modulaire de Montgomery sur une grande base

Entrées : X, Y, M avec $0 \leq X, Y < M$, $R = r^n$

Pré calculé : $M' avec\ (-M) \times M' = 1\ mod\ r$

Sortie : $S = X \times Y \times R^{-1} mod\ M$

Début $S_0 = 0$ // le mot le moins significatif.

pour i **de** 0 à d $-$ 1 **faire**

$q_i = (S_0 + Y_i \times X) \times M' mod\ r$

$S = (S + Y_i \times X + q_i \times M)/r$

fin **pour ;**

Si $(S \geq M)$ ***alors*** $S = S - M$

fin

Dans cet algorithme, la complexité dépend du nombre d'itérations « **d** » et de la base k.

Base	Nbr d'itérations	Complexité d'itération	Délai total
Base 2	1024	$T_2 = T_{add}$	$\Delta_2 = 1024 \times T_2$
Base 32	32	$T_{32} = 2 \times (T_{add} + T_{mul\_32})$	$\Delta_{32} = 32 \times T_{32}$
Base 64	16	$T_{64} = 2 \times (T_{add} + T_{mul\_64})$	$\Delta_{64} = 16 \times T_{64}$
Base k	k	$T_k = 2 \times (T_{add} + T_{mul\_64})$	$\Delta_k = 1024/k \times T_k$

Tableau 2.1- La complexité des algorithmes selon la base utilisée

D'après ce tableau on constate que le délai total de l'algorithme dépend de la base utilisée. L'augmentation de k implique :

➔ Diminution du nombre d'itérations « d »

➔ Augmentation de la complexité de la multiplication qu'est une opération séquentielle, elle a une complexité très importante car elle consiste à calculer les produits partiels puis faire leurs additions.

Pour une seule multiplication de $\boldsymbol{n \times n}$ bits (en grande base) on a besoin de « d » itérations (chacune génère un produit partiel et fait l'addition avec le résultat précédant où chaque produit partiel est une multiplication de $n \times k$ bits).

Pour un algorithme rapide il est intéressent de réduire le temps de la multiplication le plus que possible, le système de numération redondant en base 2^k est une bonne solution.

II.3.4. Système de représentation des nombres redondants en base 2^k

Ce système **[Ken, 08]** utilise une représentation redondante mais très facile à implémenter sur matériel. Par cette représentation le nombre est décomposé en d chiffres où chaque chiffre a une taille de **k+2** bits (*k : les bits principaux* et *2: les bits de retenue*).

Par cette représentation le nombre est décomposé en **d** chiffres $(X_{d-1}, X_{d-2}, \dots, X_0)$ chacun de taille (k+2) bits (les k bits principaux et les 2 bits de retenue). Un nombre de taille **d× (k+2)** bits, il est composé par d chiffres tels que.

X_i $[k-1,0]$: Bits principaux. X_i [k+1, k] : Bits redondants

Un nombre redondant est représenté sur la figure **2.1**.

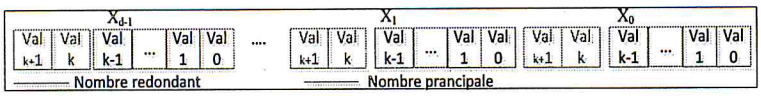


Figure 2.1- représentation d'un nombre en **SNR_d_2^k**

Cette présentation redondante est plus basée sur le travail présenté dans **[Nak, 09]** qui permet l'utilisation des multiplieurs embarqués (18×18) des circuits FPGA de Xilinx. La taille du multiplieur est de 18bits donc il faut k+2 =18 ➔ k=16. Pour la multiplication de deux nombres on a besoin de **d** multiplieurs (18×18) et de **d** additionneurs de (16, 16 ,4).

Un multiplieur $18 \times 18bits$ permet de multiplier deux nombres de 18 bits, si la taille est supérieure à 18 les opérandes sont découper en **s** mots où le nombre de multiplieur égal $\mathbf{2}^{s}$ multiplieurs. La figure I.2 représente la multiplication par les multiplieurs 18×18 de FPGA.

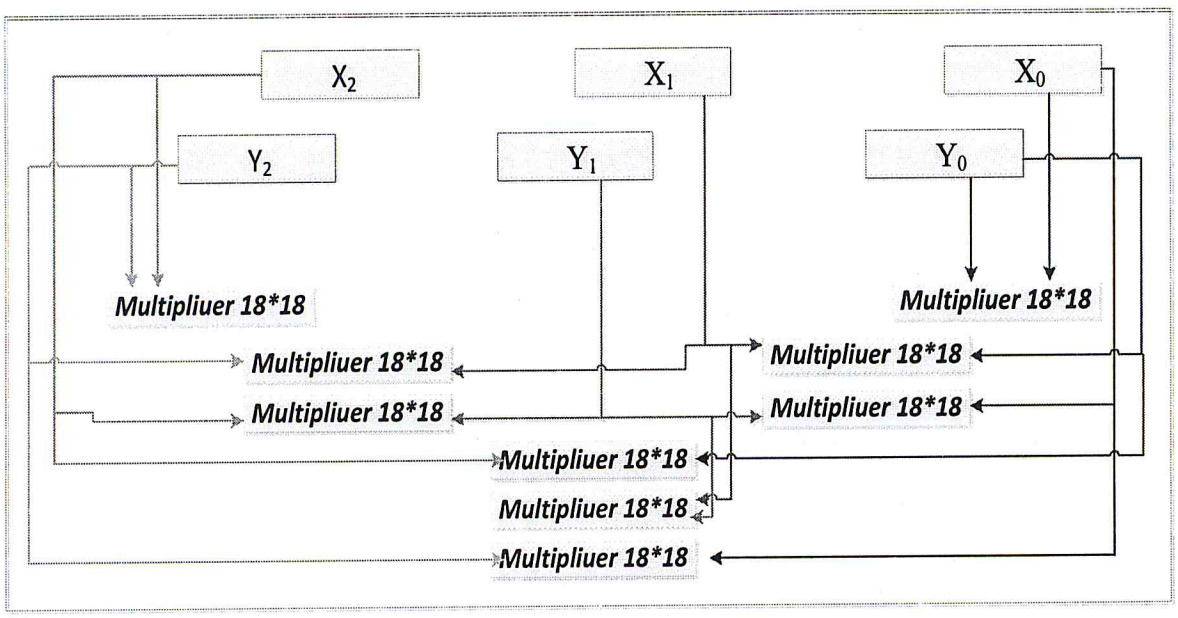


Figure 2.2- la multiplication de deux opérandes de trois mots

La figure 2.3 montre que pour une taille : $36 < T_{op} \leq 54$, C'est-à-dire pour 3 mots de 18 bits, le nombre de multiplieur égale $\mathbf{2^3 = 9}$.

La minimisation de nombre des ressources a un grand avantage sur la diminution de temps d'exécution, pour cela on va utiliser une taille de $k + 2 = 18$ ➔ $k = 16\ bits$.

II.4. La multiplication modulaire avec l'algorithme de Barrett

Pour réduire le nombre d'étapes, Dhem introduit une généralisation de l'algorithme de Barrett **[Mir, 09]** présenté dans l'algorithme 2.10. Cette généralisation permet la réduction de l'erreur sur le quotient à 1. Cet algorithme n'est pas approprié au système embarqué comme il exige l'utilisation de grands multiplieurs. Il est représenté par l'algorithme 2.10

Algorithme 2.10 : l'algorithme de la réduction de Barrett en base 2.

Entrées : $X = (X_{n-1} \dots X_0)_2$, $Y = (Y_{n-1} \dots Y_0)_2$, $M = (M_{n-1} \dots M_0)_2$, $\mu = \left\lfloor \frac{2^{n+\alpha}}{M} \right\rfloor$ où $0 \leq X, Y < M$, $2^{n-1} \leq M < 2^n$ et $\beta = -2$, $\alpha = k + 3$.

Sortie : $S = X \times Y \bmod M$.

début $S = 0$;

pour i *do* $n - 1$ à 0 ***faire***

$S = 2 \times S + X \times Y_i$

$$\hat{q} = \left\lfloor \frac{\left\lfloor \frac{S}{2^{n+\beta}} \right\rfloor \mu}{2^{\alpha-\beta}} \right\rfloor$$

$S = S - \hat{q} \times M$

Fin pour

Si $S \geq M$ *alors* $S = S - M$ ***fin Si*** ;

Retourner S ;

Fin Si ;

II.4.1. Multiplication modulaire sur une grande base avec l'algorithme de Barrett

Pour rendre la multiplication modulaire plus efficace, la multiplication modulaire série-bloc entrelacé avec la généralisation de la réduction du Barrett est donnée dans l'algorithme suivant, le quotient est évalué de la même façon que dans la généralisation du Barrett.

L'algorithme 2.11 représente cet algorithme.

Algorithme 2.11: Multiplication modulaire série-bloc entrelacé avec la réduction généralisée Barrett

Entrées : $X = (X_{d-1} \dots X_0)_r$, $Y = (Y_{d-1} \dots Y_0)_r$, $M = (M_{d-1} \dots M_0)_r$, $\mu = \left\lfloor \frac{2^{n+k+3}}{M} \right\rfloor$

où $0 \leq X, Y < M$, $2^{n-1} \leq M < 2^n$, $r = 2^k$ et $d = \lceil n/k \rceil$

Sortie : $S = XY \bmod M$.

Début $S \leftarrow 0$

pour $i = d - 1$ à 0 ***faire***

$S = S \times r + XY_i$

$$\hat{q} = \left\lfloor \frac{\left\lfloor \frac{S}{2^{n-2}} \right\rfloor \mu}{2^{k+5}} \right\rfloor$$

$S = S - \hat{q}M$

Fin pour

Si $S \geq M$ ***alors*** $S - M$ ***fin Si***

Retourner S*;*

Fin;

Évaluation :

L'algorithme du Barrett utilise une représentation classique des nombres, une valeur pré calculée, l'algorithme ne contient que des multiplications et des décalages, et évite les divisions coûteuses. Sa complexité égale : $T_{\text{cycle d'horloge}} = (T_{\text{add}} + T_{\text{soustraction}} + 3T_{\text{mul\_k}})$.

Le système des nombres redondants en base 2^k n'est pas adapté (car il utilise la soustraction). La multiplication de Barrett est plus complexe et nécessite trois multiplications au lieu d'une dans l'algorithme de Montgomery.

Remarque Importante :

Pour un seul calcul de l'opération X× Y mod M et pour un modulo M inférieur à 768 bits, l'algorithme Barrett serait plus rapide que celui de Montgomery.

L'algorithme de Montgomery serait un meilleur choix si le modulo est sur une taille supérieure à 768 où pour une application qui nécessite le calcul de la multiplication modulaire plusieurs fois comme l'exponentiation modulaire.

Dans ce qui suit, on propose une nouvelle version de l'algorithme de Barrett où le système des nombres redondants en base 2^k est avantageusement utilisé. Ceci augmente les performances d'exécution de l'algorithme de Barrett avec une complexité proche de celle de l'algorithme de Montgomery. Cette proposition sera détaillée dans ce qui suit.

II.5. Le parallélisme dans les algorithmes de multiplication modulaire

Souvent pour l'exécution performante d'un algorithme, il est nécessaire de l'implémenter dans un matériel dédie et le parallélisme doit être exploité autant que possible pour atteindre les meilleurs performances.

Le parallélisme a pour but d'augmenter la vitesse d'application en multipliant les ressources de calcul d'un système. Pour en bénéficier, l'application doit être pensée en termes de tâches indépendantes qui pourront être exécutés de manière concurrente par les différentes unités de calcul **[Iza, 11].**

On a déjà vu précédemment, le parallélisme à l'intérieur de l'itération c'est-à-dire parallélisé les opérations arithmétiques comme l'addition et la multiplication, par l'utilisation de la représentation redondante.

La représentation de grande base permet de diminuer le nombre de cycles, mais avec une augmentation de la complexité de l'itération.

Kaihara et Takagi ont proposés une nouvelle méthode **[Kai, 08]** pour réduire le nombre d'itérations sans augmenter le temps requis pour chaque itération, en outre, la conception peut rester simple par rapport à d'autres conceptions de la multiplication modulaire à grande base pour une performance similaire. Cette méthode est également adaptée pour une mise en œuvre logicielle dans un environnement multiprocesseurs, Il est à noter que cette dernière n'a pas fait objet d'implémentation matérielle sur FPGA ou ASIC.

Dans ce qui suit on va présenter cette méthode et ses différentes versions. Une adaptation de celle-ci à une implémentation sur FPGA sera développée tout au long de ce travail.

II.5.1 la multiplication modulaire Bipartite

La méthode de multiplication bipartite MMB comme son nom indique cherche à calculer $X \times Y \bmod M$ de manière à découper cette opération en deux sous multiplications modulaires qui seront exécutées en parallèle.

L'idée de base est de diviser le multiplicateur Y en deux parties de même taille Y_H et Y_L :

$Y = Y_H\, r^{n/2} + Y_L$, $|Y_H| < r^{n/2}$ et $|Y_L| < r^{n/2}$ n : la taille de M, X, Y.

$$\boldsymbol{X \times Y \bmod M} = X\left(Y_H r^{\frac{n}{2}} + Y_L\right) \bmod M = \left(XY_H r^{\frac{n}{2}} \bmod M + XY_L \bmod M\right) \bmod M$$

La MMB utilise la représentation de Montgomery donc on cherche à calculer

$$\boldsymbol{X \times Y \times R^{-1}\ mod\ M} = \left(XY_H r^{\frac{n}{2}}\ mod\ M + XY_L\ mod\ M\right) \times \boldsymbol{R^{-1}} mod\ M$$

$$= \left(XY_H r^{\frac{n}{2}} \times \boldsymbol{R^{-1}} mod\ M + XY_L \times \boldsymbol{R^{-1}} mod\ M\right) mod\ M$$

Pour éliminer le facteur $r^{n/2}$ de $XY_H r^{\frac{n}{2}}\ mod\ M$ on prend $R^{-1} = r^{-n/2}$ donc :

$$\boldsymbol{X \times Y \times R^{-1} mod\ M} = \left(XY_H\ mod\ M + XY_L\ r^{-\frac{n}{2}}\ mod\ M\right) mod\ M$$

La multiplication a été décomposée en deux multiplications modulaires $XY_H\ mod\ M$ et $XY_L\ r^{-\frac{n}{2}}\ mod\ M$ qui sont calculés en parallèle par l'algorithme classique et l'algorithme de Montgomery respectivement.

Selon les études précédentes pour effectuer la multiplication modulaire à double dimension on utilise deux multiplicateurs modulaires, elle a une taille unique. Son délai égal au délai de la multiplication la plus lente.

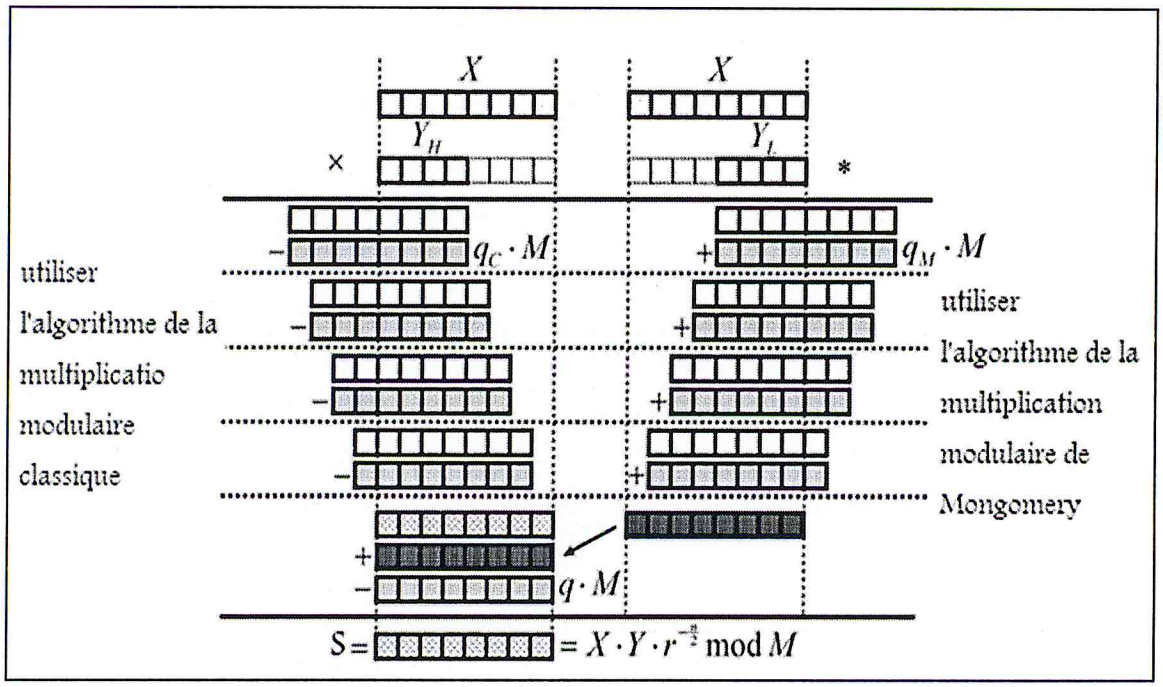


Figure 2.3- Schéma de fonctionnement de la méthode

Remarque :

Le résultat calculé par la méthode Bipartite est $\boldsymbol{XY\ r^{-\frac{n}{2}}\ mod\ M}$ qui est dans le domaine de Montgomery et pas $XY\ mod\ M$ donc pour revenir au résultat original de $XY\ mod\ M$ il faut sortir de domaine de Montgomery par $S = Montgomery\ (C, R^2, M)$.

La méthode Bipartite démunie la complexité ($1024 \times 1024\ bits$) à une complexité de deux multiplications de ($\mathbf{512\ bits \times 1024\ bits}$) qui sont exécutés en parallèle. Mais l'utilisation de l'algorithme de la multiplication modulaire classique rende l'implémentation matérielle de cette méthode impossible.

Deux versions ont été inventées **[Gio, 13]** consistent à généraliser MMB pour réduire d'avantage la complexité calculatoire, cette méthode a été généralisée à d'autre découpages :

La quadripartite : Elle a une complexité de 512×512 bits.

La multipartite : Elle a une complexité de $\frac{1024}{k} \times \frac{1024}{k}\ bits$.

L'idée est de générer plus des sous-multiplications modulaires de taille inférieure à la multiplication modulaire originale avec l'utilisation de nouvelles versions de l'algorithme de Barrett et l'algorithme de Montgomery qui sont, l'algorithme partiel de Barrett et l'algorithme partiel de Montgomery respectivement.

II.5.2. La multiplication modulaire Multipartite

La multiplication modulaire Multipartite consiste à diviser les opérateurs X et Y en k blocs tel que : $X = \sum_{i=0}^{k-1} X_i \times r^{-i \times n/k}$, $Y = \sum_{j=0}^{k-1} Y_j \times r^{-j \times n/k}$. et r représente la base.

$X \times Y \times r^{-n/2}\ mod\ M = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} X_i \times Y_j \times r^{d_{i,j}}\ mod\ M$ Où :

$-\frac{n}{2} \leq d_{i,j} = \frac{n(i+j)}{k} - \frac{n}{2} \leq \frac{3n}{2} - \frac{2n}{k}$

Cette méthode génère k^2 produits partiels : $X_i Y_j r^{d_{i,j}}$ qui sont calculés en parallèle et ils sont devisés en trois types :

Les produits bas: Si $d_{i,j} < 0$, le produit $X_i Y_j r^{d_{i,j}}$ est réduit par l'algorithme partiel de Montgomery.

Les produits haut: Si $d_{i,j} > n - \frac{2n}{k}$, le produit $X_i Y_j r^{d_{i,j}}$ est réduit par l'algorithme partiel de Barrett.

Pour $0 \leq d_{i,j} \leq n - \frac{2n}{k}$, on a $X_i Y_j r^{d_{i,j}} < r^n$ donc sont seulement des multiplications.

Pour trouver le résultat de l'opération $\boldsymbol{XY\ r^{-\frac{n}{2}}}$ la multiplication modulaire multipartite utilise l'arbre de réduction pour calculer la somme modulaire de ces sous multiplications modulaires.

Le nombre de produits partiales (N_p) dépend de k choisi, lorsque k augment N_p va augmenter. L'intérêt de son implémentation sur FPGA n'est pas justifié car elle a besoin énormément de ressources matérielles.

La figure I.2 représente le schéma de fonctionnement de la méthode Multipartite avec une valeur k=5.

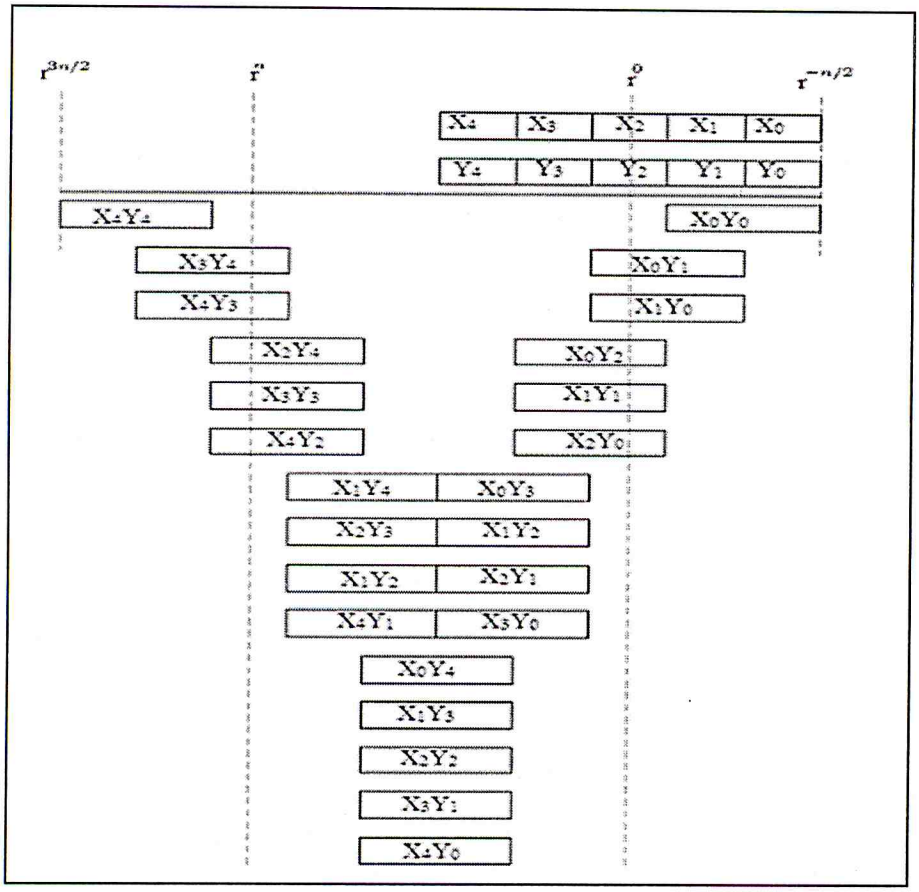


Figure 2.4- Schéma de fonctionnement de la méthode Multipartite k=5

II.5. Conclusion

La multiplication modulaire est l'opération de base de la cryptographie RSA, donc chaque amélioration au niveau de cette opération se répercute directement sur les performances du crypto-système RSA.

Dans ce chapitre nous avons présenté les différentes méthodes de calcul de la multiplication modulaire et nous avons focalisé sur les méthodes de Montgomery et celle de Barrett. Nous avons aussi présenté les versions sur une grande base de numération de ces méthodes pour diminuer le nombre d'itérations. Celles-ci font augmenter la complexité

calculatoire de l'itération, pour ce faire nous avons proposé une méthode, basée sur la représentation redondante des nombres où tous les calculs dans l'itération sont réalisés dans ce format.

Ceci évite la propagation de la retenue et réduit sensiblement de délai de l'itération. Pour augmenter d'avantage les performances de notre multiplication modulaire, nous avons adapté une méthode, dédiée aux calculs sur multiprocesseurs, pour paralléliser le calcul de notre multiplication modulaire. Cette dernière est basée sur le découpage des opérandes sur k blocs et une multiplication modulaire de n×n bits se trouve réduit à une multiplication de $\frac{n}{k} \times \frac{n}{k}$ bits

Dans le chapitre suivant nous allons détailler notre proposition d'algorithme matériel pour la multiplication modulaire qui sera dédié au crypto-système RSA en se basant sur les principes développés dans ce chapitre.

Chapitre III:
Conception

III.1. Introduction :

Les performances en matière de temps de calcul du protocole RSA dépendent de l'efficacité des algorithmes utilisés à chaque niveau arithmétique. On dispose généralement de plusieurs algorithmes pour réaliser ces opérations de base comme l'exponentiation modulaire et la multiplication modulaire varient en fonction d'un certain nombre de paramètres, comme la taille des opérandes, l'utilisation de pré-calcul,... etc. Mais ils dépendent également du type des opérations utilisées.

L'algorithme qui utilise la représentation binaire des nombres a l'avantage d'voir un temps d'exécution, de la multiplication d'un bit par un opérande, presque négligeable, égale au délai d'une porte AND. Le désavantage est qu'il nécessite un nombre d'itérations égales à la taille des opérandes en bits, qui est de l'ordre 1024 à 2048 bits.

L'augmentation de la base est intéressante où les opérandes sont découpés en paquet de k bits, ce qui nous permet d'une part réduire le nombre d'itérations et par conséquent nous permet de diminuer le temps d'exécution. Néanmoins le réseau de portes AND sera remplacé par un multiplieur, du fait qu'à chaque itération on fait une multiplication d'un bloc de k bits par un opérande.

En effet, l'augmentation de la base réduire le nombre d'itérations mais, elle introduit une complexité calculatoire dans l'itération et par conséquent le délai de l'itération augmente. Comme nous avons vu dans le chapitre précédent la représentation redondante des nombres permet de réduire ce délai.

Pour la multiplication modulaire. Il existe deux types d'algorithmes selon le sens de traitement de données qui sont :

Les algorithmes qui commencent par les chiffres les moins significatifs comme l'algorithme de Montgomery où à chaque itération il ajoute un multiple du modulo tel que le chiffre le mois significatif du résultat sera égale à zéro. C'est le plus utilisé pour des applications matérielles.

Les algorithmes qui commencent par les chiffres les plus significatifs comme l'algorithme de Barrett où à chaque itération, on calcule le quotient sans faire la division puis calculer la soustraction entre le nombre et le quotient multiplié par le modulo. Il est à noter que la soustraction nécessite l'utilisation de la représentation classique des nombres ce qui diminue la rapidité, une

implémentation matérielle de cet algorithme induit une consommation excessive des ressources en plus d'une grande complexité de routage qui induit un délai d'exécution important, C'est pour cela que cet algorithme n'est pas conseillé pour les applications cryptographiques qui nécessitent une taille de modulo supérieur à 712 bits.

Comme on a déjà vu précédemment, la méthode multipartite permet de paralléliser les calculs et de travailler avec un data path plus petit que la taille des opérandes. Elle divise, en fait, les opérandes en des parties égales où chaque partie est traitée par un des deux algorithmes soit Montgomery ou celui de Barrett. Il est à noter que toutes les parties sont exécutées en parallèle.

Notre travail s'articule autour de la définition d'opérateurs arithmétiques performants pour le protocole RSA permettant d'augmenter les performances de ce dernier et comme le parallélisme au niveau arithmétique le plus bas permet des gains modérés en matière de temps de calcul, nous proposons d'utiliser la méthode quadripartite, mais comme on a l'indique dans le chapitre précédent, les performances d'exécution de cette méthode dépendent étroitement des performances de deux algorithmes, l'algorithme de Montgomery et l'algorithme de Barrett où le temps d'exécution égale au temps d'exécution de l'algorithme le moins rapide.

En effet, le temps d'exécution d'une multiplication modulaire de Barrett nécessite un temps d'exécution plus important que celui de Montgomery grâce à l'utilisation de la représentation classique des nombres.

Pour atteindre notre objectif, nous proposons un algorithme permettant un parallélisme au niveau de l'opération arithmétique la plus bas dans ce protocole, nous introduisons une version rapide de l'algorithme de Barrett. Cette dernière utilise une représentation redondante que nous appelons « système de numération redondant signés en base 2^k » qui permet de diminuer le temps d'exécution de ce dernier à un temps qui sera proche de celui l'algorithme de Montgomery. On appelle notre algorithme « *algorithme parallèle pour la multiplication modulaire* » qui sera présenté dans ce qui suit.

III.2. La multiplication modulaire quadripartite

Parmi les versions de la méthode multipartite et après plusieurs tests on a trouvé que la méthode quadripartite MMQ est la plus performante pour notre architecture, elle permet de diminuer la complexité de la multiplication modulaire de 1024×1024 bits à une complexité de deux multiplications modulaires de 512×512 bits.

Cette méthode **[Gio, 13]** consiste à découper les deux opérandes en deux parties égales :

$$XY2^{-n/2} mod\ M = (\mathrm{X1} \times 2^{\frac{n}{2}} + \mathrm{X0})(\mathrm{Y1} \times 2^{n/2} + \mathrm{Y0})2^{-n/2} mod\ M$$

$$XY.2^{-n/2} mod\ M \quad = (\mathrm{X1} \times \mathrm{Y1}.2^{\frac{n}{2}} + \mathrm{X1} \times \mathrm{Y0} + \mathrm{X0} \times \mathrm{Y1} + \mathrm{X0} \times \mathrm{Y0}.2^{-\frac{n}{2}}) mod\ M$$

En effet, comme $X < M = M1\ 2^{n/2} + M0$, *on a* $X1\ 2^{n/2} < M1\ 2^{n/2} < M$ car M est impair. On a de plus $\mathrm{Y0} < 2^{n/2}$ donc $\mathrm{X1} \times \mathrm{Y0} < X_1 2^{n/2} <$ M. Le même raisonnement conduit à $\mathrm{X0} \times \mathrm{Y1} <$ M. Alors l'expression devient alors :

$$XY2^{-n/2} mod\ M = (\mathrm{X1} \times \mathrm{Y1}\ 2^{\frac{n}{2}} mod\ M + \mathrm{X1} \times \mathrm{Y0} + \mathrm{X0} \times \mathrm{Y1} + \mathrm{X0} \times \mathrm{Y0} \times 2^{-\frac{n}{2}} mod\ M)\ mod\ M$$

Cette multiplication quadripartite requiert donc deux multiplications et deux sous-multiplications modulaires d'une taille égale la moitié ($\frac{n}{2}$) de la taille initiale qui sont :

- La réduction du produit bas $\mathrm{X0} \times \mathrm{Y0}2^{-n/2}\ mod\ M$ est calculée par l'algorithme de Montgomery.
- La réduction du produit haut$\mathrm{X1} \times \mathrm{Y1}\ 2^{n/2}\ mod\ M$ est calculée par l'algorithme de Barrett.
- Les produits médians $\mathrm{X1} \times \mathrm{Y0}$ et $\mathrm{X0} \times \mathrm{Y1}$ sont des multiplications

Où la taille de tous les sous opérandes X1, X0, Y0 et Y1 sont égale à $\frac{n}{2}$

La figure 3.1 présente le fonctionnement de cette méthode.

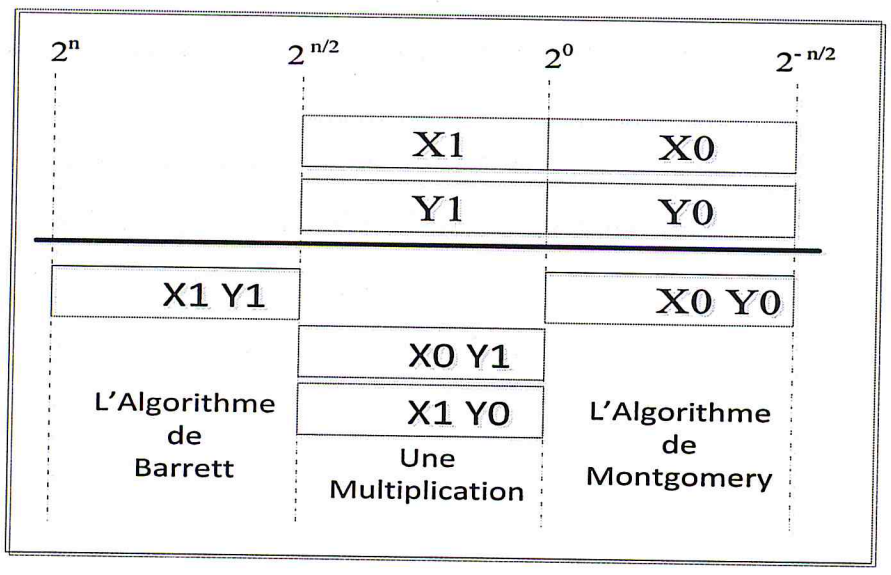


Figure 3.1- Schéma de fonctionnement de la MMQ

Dans ce que suite on va détailler nos propositions pour les architectures de Montgomery et de Barrett puis l'architecture globale pour le calcul de la multiplication modulaire en parallèle par la méthode MMQ.

III.3. La multiplication modulaire de Montgomery

III.3.1. Algorithme de la MMMGB

Dans l'algorithme de Montgomery dans une grande base, les opérandes sont définis comme suit : Soit les opérandes X et Y et le modulo M représentés en base 2^k par :

$$X = \sum_{j=0}^{d-1} X_j \times 2^{j\times k},\quad Y = \sum_{i=0}^{d-1} Y_i \times 2^{i\times k},\quad M = \sum_{j=0}^{d-1} M_j \times 2^{j\times k},$$

$$R = 2^{d\times k} \quad et \quad R^{-1} = 2^{-(d\times k)}$$

Montgomery calcule S selon l'expression suivante :

$$S = (X \times Y \times 2^{-(d\times k)}) mod\ M$$

$$= \left(\left(\sum_{i=0}^{d-1} Y_i \times 2^{i\times k}\right) \times X \times 2^{-(d\times k)}\right) mod\ M$$

$$= ((\dots (Y_0 \times X)2^{-k} \bmod M + Y_1 \times X)2^{-k} \bmod M + \cdots + Y_i \times X)2^{-k} \bmod M + \cdots + Y_{d-1} \times X) \times 2^{-k} \bmod M$$

Ce qui signifié que la multiplication de Montgomery peut être calculée par l'utilisation de la formule de récurrence suivante :

$$\begin{cases} S_0 = 0 \\ S_{i+1} = (S_i + (Y_i \times X)) \times 2^{-k} mod\ M, \text{ où } i = 0, \dots, d-2. \end{cases}$$

Cette formule signifiée qu'à chaque itération un produit partiel est généré, puis ajouté au résultat précédent. Le tout est multiplié par le facteur $\mathbf{2^{-k}}$, qui n'est rien d'autre qu'un décalage de k bits. Ce dernier est effectué après avoir normalisé le résultat intermédiaire S_i. Cette normalisation est réalisée par l'addition à S_i d'un certain nombre de modulos $(q_i \times M)$ de manière à avoir les k bits les moins significatifs de S_i à zéro ce qui permet de le diviser par $\mathbf{2^{-k}}$. C'est le principe de l'algorithme de Montgomery :

$$\text{Si } q_i = \left((S_i + (Y_i \times X)) \times M'\right) \bmod \mathbf{2^k} \text{ avec } M' = -M_0^{-1} mod\ 2^k \ et\ pgcd(M, 2^k) = 1$$

Il est à noter que q_i est codé sur k bits et M_0 est le mot le moins significatif de M. alors :

$$S_{i+1} = (S_i + (Y_i \times X) + (q_i \times M)) \times 2^{-k} \quad avec\ S_{i+1} < 2M.$$

Dans cette expression l'opérande Y est considéré par bloc de k bits alors que X et le modulo M sont considérés dans leur totalité. L'implémentation matérielle de cette récurrence présente une grande complexité de routage qui induit un délai considérable et c'est pourquoi une implémentation de cette récurrence avec le système de numération redondant en base 2^k est intéressante.

III.3.2 Algorithme de la MMMSNR-2^k

L'algorithme de la Multiplication Modulaire de Montgomery avec le Système de Numération Redondante de **d** chiffres en base 2^k est la généralisation de l'algorithme MMMGB (Multiplication Modulaire de Montgomery sur une grande base) à l'utilisation des opérandes X, Y ainsi que le modulo M et le résultat S_i dans la représentation redondante des nombres en base 2^k(où les bits redondants de tous les mots de X, Y et M sont des zéros).

$$\begin{cases} S_0 = 0 \\ S_{i+1} = \left(\left(\sum_{j=0}^{d-1} S_{i,j} \times 2^{j\times k}\right) + \left(Y_i \times \sum_{j=0}^{d-1} X_j \times 2^{j\times k}\right) + \left(q_i \times \sum_{j=0}^{d-1} M_j \times 2^{j\times k}\right)\right) \times 2^{-k} \end{cases}$$

Cette dernière expression est constituée par deux opérations élémentaires qui sont l'addition et la multiplication. Le système de numération redondant en base 2^k permet de calculer la multiplication et l'addition dans un temps constant indépendant de la taille du data path. L'exécution de cet algorithme est basée sur deux indices $i\ et\ j$:

- L'indice (i) désigne d'une part, la $i^{ème}$ itération de l'algorithme et d'autre part le $i^{ème}$ mot de l'opérande Y.
- L'indice (j) désigne le $j^{ème}$ mot de l'opérande X, du modulo M et des valeurs intermédiaires Z, SP et S.
- Le déroulement d'une itération (i) est décrit comme suit :

À la fin de l'itération (i-1), l'exécution de l'itération (i) commence à partir de la lecture de Y_i.

On peut considérer que l'itération (i) peut être exécutée en deux étapes successives :

$$\begin{cases} Z_i = \sum_{j=0}^{d} Z_{i,j} \times 2^{j\times k} = S_i + Y_i \times X = \left(\sum_{j=0}^{d-1} S_{i,j} \times 2^{j\times k}\right) + \left(Y_i \times \sum_{j=0}^{d-1} X_j \times 2^{j\times k}\right) \\ S_{i+1} = (Z_i + (q_i \times M)) \times 2^{-k} = \left(\left(\sum_{j=0}^{d} Z_{i,j} \times 2^{j\times k}\right) + \left(q_i \times \sum_{j=0}^{d-1} M_j \times 2^{j\times k}\right)\right) \times 2^{-k} \\ où \quad q_i = (Z_{i,0} \times M')mod\ 2^k \end{cases}$$

On calcul en premier lieu Z à partir de S_i , Y_i et X_j .

$Z_i = \sum_{j=0}^{d-1} S_{i,j} \times 2^{j \times k} + Y_i \times \sum_{j=0}^{d-1} X_j \times 2^{j \times k}$.

Puisque $S_{i,j}$ est codé sur k+2, On peut considère que :

$$S_{i,j} = C1_{i,j} \times 2^k + S1_{i,j} \text{ où } C1_{i,j} = S_{i,j}[k+1, k] \text{ et } S1_{i,j} = S_{i,j}[k-1,0]$$

Et comme la multiplication de deux nombres codés sur k bits donne comme résultat un nombre codé sur 2k bits. On peut considère que :

$SP_{i,j} = Y_i \times X_j = SP2_{i,j} \times 2^k + SP1_{i,j}$. où $SP2_{i,j} = SP_{i,j}[2k-1, k]$ et $SP1_{i,j} = SP_{i,j}[k-1,0]$

Après l'additionne de tous les mots de même poids sont regroupés ensemble, l'expression de Z_i sera alors:

$$Z_i = (S1_{i,0} + SP1_{i,0}) + (S1_{i,1} + SP1_{i,1} + C1_{i,0} + SP2_{i,0}) \times 2^k + \cdots + (S1_{i,j} + SP1_{i,j} + C1i,j-1 + SP2i,j-1 \times 2j \times k + .. + C1i,d-1 + SP2i,d-1 \times 2d \times k .$$

La méthode quadripatrite qui est utilisé dans notre architecture consiste à calculer le résultat de l'opération : $X0 \times Y0 \times 2^{-\frac{n}{2}} \bmod M$. Où les opérandes X0, Y0 ont$d = \frac{n}{k}$ mots et le modulo M a $\frac{d}{2} = \frac{n}{2k}$ mots pour le modulo M. alors le produit de X, Y est calculé comme suit :

$SP_i = Y_i \sum_{j=0}^{\frac{d}{2}} X_j = \sum_{j=0}^{\frac{d}{2}} SP_{i,j} \times 2^{j \times k}$.

Ce que signifie que Z_i peut être calculé en utilisant la formule de récurrence suivante :

$$\begin{cases} SP2_{i,-1} = SP2_{i,\frac{d}{2}+1} = S1_{i,d+1} = C1_{i,-1} = 0; \\ Z_{i,j} = (SP2_{i,j-1} + SP1_{i,j} + C1_{i,j-1} + S1_{i,j}) \ \forall j, \ 0 \le j \le \frac{d}{2} \\ Z_{i,j} = (C1_{i,j-1} + S1_{i,j}) \ \forall j, \ \frac{d}{2} + 1 \le j \le d + 1 \end{cases}$$

On conclura que Z est un nombre redondant de d+1 mots en base 2^k.

En effet, la multiplication suivie par une addition dans le système de numération redondant en base 2^k est calculée en deux étapes, la première étape consiste à calculer tous les produits partiels en parallèle et la deuxième consiste à calculer la somme de ces produits avec le troisième nombre où tous les mots sont calculés en parallèle, le résultat final est un nombre redondant de $d + 1$ mots en base2^k.

Le déroulement de cette opération est représenté sur la figure 3.2 :

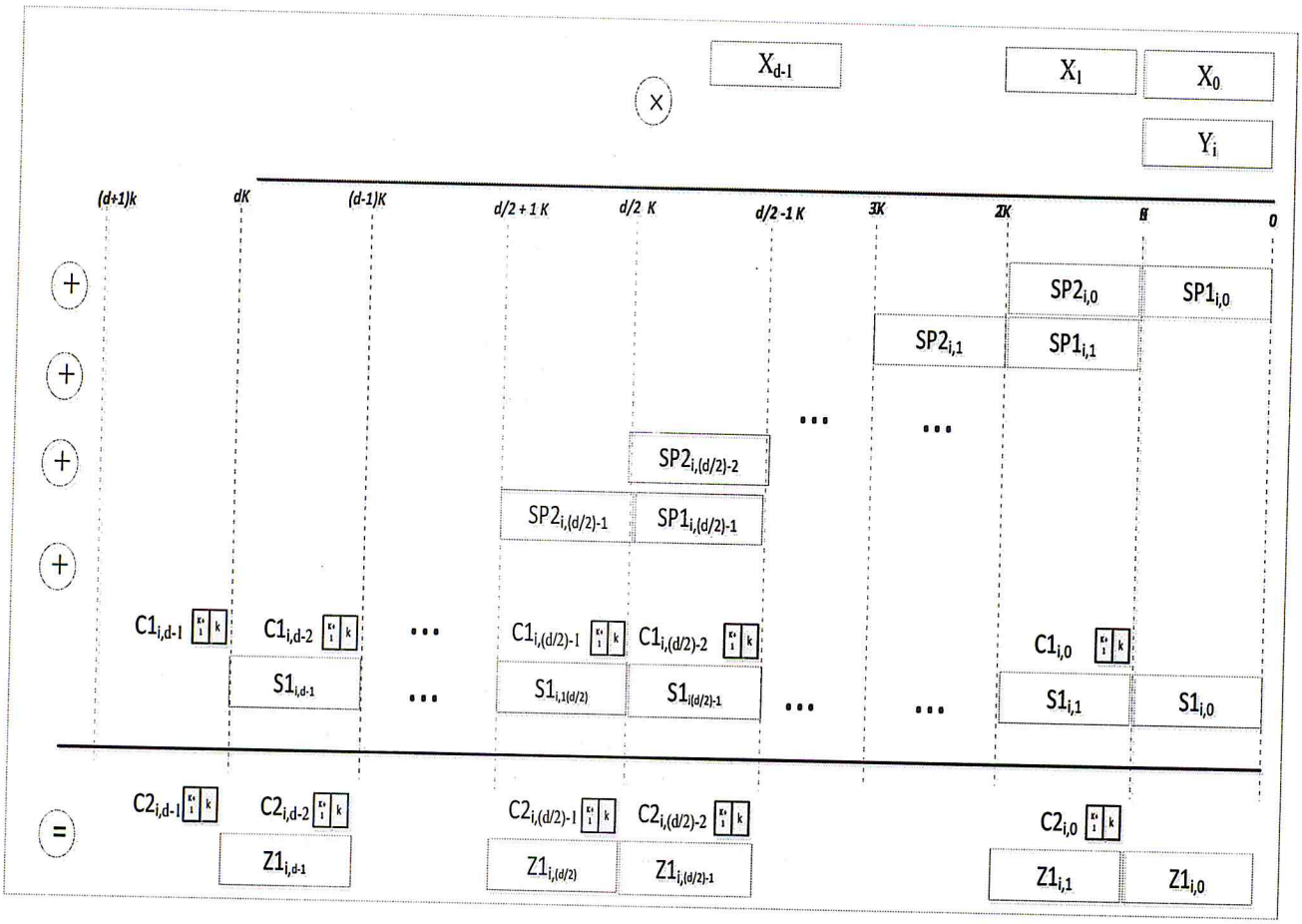


Figure 3.2 : L'exécution d'une multiplication suivie par une addition.

Dans la seconde étape de l'opération (i) on peut considérer que l'obtention de S_{i+1} est identique au calcule de Z_i et est constituée par les mêmes types d'opérations (une multiplication suivie par une addition). La seule différence est dans le nombre de produits partiels de $q_i \times M$ qui est égale à d produits ce que signifie:

$$\begin{cases} P_i = \sum_{j=0}^{d-1} P_{i,j} \times 2^{j \times k} = q_i \times \sum_{j=0}^{d-1} M_j \times 2^{j \times k} \text{ où } P_{i,j} \text{ est codé sur 2k bits} \\ S_{i+1} = \sum_{j=0}^{d} S_{i+1,j} \times 2^{j \times k} = (\sum_{j=0}^{d} Z_{i,j} \times 2^{j \times k} + \sum_{j=0}^{d-1} P_{i,j} \times 2^{j \times k}) \end{cases}$$

Comme q_i et M_j sont respectivement sur k bits leur produit $q_i \times M_j$ est sur 2k bits, une nouvelle représentation est introduite après l'accumulation de tous ces produits partiels qu'est P_i. Ce dernier est représenté dans sa forme en digits puis additionné à Z_i pour trouver S_{i+1}.

On a $Z_{i,j}$ est codé sur $k+2$ bits ce qui permet de le représenter comme suite :

$Z_{i,j} = C2_{i,j} \times 2^k + Z1_{i,j}$ où $C2_{i,j} = Z_{i,j}[k+1, k]$ et $Z1_{i,j} = Z_{i,j}[k-1,0]$.

$P_{i,j} = P2_{i,j} \times 2^k + P1_{i,j}$ où $P2_{i,j} = P_{i,j}[2k-1, k]$ et $P1_{i,j} = P_{i,j}[k-1,0]$.

$S_{i+1} = (Z1_{i,0} + P1_{i,0}) + (Z1_{i,1} + C2_{i,0} + P2_{i,0} + P1_{i,1}) \times 2^k + \cdots + (Z1_{i,j} + C2_{i,j-1} + P2_{i,j-1} +$ $P1_{i,j} \times 2j \times k + .. + Z1i,d + C2i,d-1 \times 2d \times j$.

Ce que signifié que tous les mots de S_{i+1} peuvent être calculés en parallèle en utilisant la formule de récurrence suivante :

$$\begin{cases} P2_{i,-1} = P1_{i,d+1} = Z1_{i,d+1} = 0 \\ S_{i+1,j} = (Z1_{i,j} + C2_{i,j-1} + P2_{i,j-1} + P1_{i,j}) \quad \forall j,\ 0 \le j \le d+1 \end{cases}$$

Finalement, $S_{i+1} = S_{i+1} \times 2^{-k}$, Ce qui élimine les k bits les moins significatifs de S_{i+1}. Il reste donc les deux bits redondants qui sont sauvegardés dans le variable ***reg*** pour les additionner avec le premier mot dans l'itération suivante, donc la formule de Z_i devient alors :

$$\begin{cases} SP_{i,-1} = SP_{i,\frac{d}{2}} = S1_{i,d+1} = 0 \\ Z_{i,0} = (S1_{i,0} + reg + SP1_{i,0}) \\ Z_{i,j} = (S1_{i,j} + C1_{i,j-1} + SP2_{i,j-1} + SP1_{i,j}) \quad \forall j,\ 1 \le j \le \frac{d}{2} \\ Z_{i,j} = (S1_{i,j} + C1_{i,j-1}) \quad \forall j,\ \frac{d}{2} + 1 \le j \le d+1 \end{cases}$$

L'exécution de l'étape 2 de cet algorithme est montrée dans la figure **3.3**.

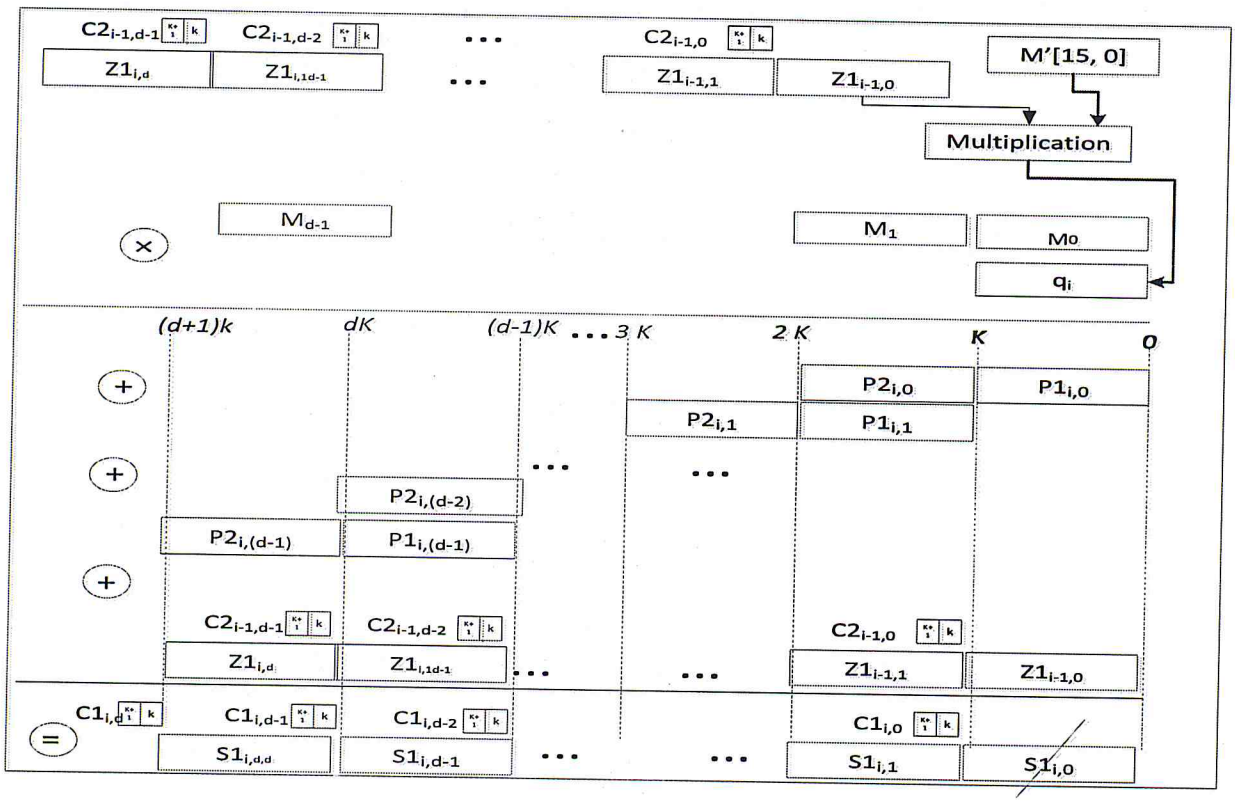


Figure 3.3 : L'exécution de la 2 ème étape de l'algorithme MMMSNR_2^k pour l'itération i

III.3.2 l'algorithme de la MMMSNR-2^k avec une mémoire

Le circuit qui réalise cette opération utilise d multiplieurs (18×18) pour calculer $q_i \times M$ en parallèle donc un grand nombre de multiplieurs. On peut le remplacer par une mémoire qui stocke la valeur de $q_i \times M$.

On a $q_i = (Z1_{i,0} \times M') \bmod 2^k$ ➔ $q_i \times M = ((Z1_{i,0} \times M') \bmod 2^k) \times M$

Soit F une fonction telle que : $F(Z1_{i,0}) = ((Z1_{i,0} \times (-M^{-1})) \bmod 2^k) \times M$.

La fonction F est calculée en utilisant 2^k mots de $(d+1) \times k$ bits mémoire où la valeur de F(i) $(0 \leq i \leq 2^k - 1)$ est stockée dans l'adresse i dans la mémoire à l'avance. Avec la lecture de l'adresse de Z [k-1,0] de la mémoire on obtient la valeur de F(Z) dans un cycle d'horloge.

Dans notre architecture k=16 et d×k=1024 bits, donc pour stoker $q_i \times M$ dans une mémoire, il nécessite 8Mo, C'est une grande capacité mémoire.

Pour éviter ce problème, la division de q_i en deux parties de $\frac{k}{2}$ bits est une solution parfaite pour réduire la taille de mémoire **[Nak, 09]**.

On peut calculer $\overline{q_i}$ *et* $\underline{q_i}$ en parallèle : $\overline{q_i} = q_i\left[k-1,\frac{k}{2}\right]$ et $\underline{q_i} = q_i\left[\frac{k}{2}-1,0\right]$.

Soit $(-M^{-1})$ le nombre minimum non négatif:

$$(-M^{-1}) \times M \equiv -1 \bmod 2^{\frac{k}{2}} \rightarrow M' = (-M^{-1}) \bmod 2^{\frac{k}{2}}$$

On pose $\begin{cases} \overline{Z1_i} = Z1_{i,0}[k-1,\frac{k}{2}] \\ \underline{Z1_i} = Z1_{i,0}[\frac{k}{2}-1,0] \end{cases}$

Si on prend $\underline{q_i} = \underline{Z1_i} \times M' \bmod 2^{\frac{k}{2}}$, alors les $\frac{k}{2}$ bits les moins significatifs de $Z_i + \underline{q_i} \times M$ sont des zéros, Il reste les $\frac{k}{2}$ bits les plus significatifs. Soit G une fonction telle que :

$$G(Z_i) = \left(\left(\left(Z_i\left[\frac{k}{2}-1,0\right] \times M'\right) \times M\right)\left[k-1,\frac{k}{2}\right]\right) + c \quad \text{où} \quad \begin{cases} \text{si } (Z_i \times M)\left[\frac{k}{2}-1,0\right] = 0 \rightarrow c = 0 \\ \text{sinon } c = 1 \end{cases}$$

Alors : $\overline{q_i} = \left(\overline{Z1_i} + G\left(\underline{Z1_i}\right)\right)\left[\frac{k}{2} - 1{,}0\right]$ où $(Z_i + \underline{q_i} \times M + \overline{q_i} \times M \times 2^{\frac{k}{2}})[k\text{-}1{,}0]=0$.

Pour calculer $\underline{q_i} \times M$ et $\overline{q_i} \times M$ avec l'utilisation de la mémoire on suppose une fonction H telle que : $H(X) = \left(X\left[\frac{k}{2} - 1{,}0\right] \times M'\right)\left[\frac{k}{2} - 1{,}0\right]$.

$$\begin{cases} \underline{MemQ_i} = \underline{q_i} \times M = Mem\left(H\left(Z_{i,0}\left[\frac{k}{2} - 1{,}0\right]\right)\right) \\ \overline{MemQ_i} = \overline{q_i} \times M = Mem\left(H\left(Z_{i,0}\left[k - 1, \frac{k}{2}\right] + G(Z_{i,1}\left[\frac{k}{2} - 1{,}0\right]\right)\right) \times 2^{k/2} \end{cases}$$

Toutes les valeurs possibles de la fonction Mem(q)=q×M sont stockées dans la mémoire avec l'utilisation de $2^{\frac{k}{2}}$ mots de (d+1) ×k bits mémoires. On obtient $\underline{q_i} \times M$ en 1 cycle d'horloge et 2 cycles d'horloge pour $\overline{q_i} \times M$.

Dans cet algorithme tous les mots de S_i sont calculés en parallèle par l'expression :

$$\{S_{i,j} = \overline{MemQ}_{i,j} + \underline{MemQ_{i,j}} + Z1_{i,j} + C2_{i,j-1} \qquad \forall j,\ 0 \le j \le d+1$$

Cet algorithme est présenté dans l'algorithme **3.1**.

Algorithme 3.1 : l'algorithme de la MMMSNR-2^k avec une mémoire

Entrée : $X = \sum_{j=0}^{d-1} X_j \times 2^{j \times k}, Y = \sum_{i=0}^{d-1} Y_i \times 2^{i \times k}, M = \sum_{j=0}^{d-1} M_j \times 2^{j \times k}, \boldsymbol{avec\ pgcd(M, 2^k) = 1}$

Pré calculés : $M' = \boldsymbol{M_0^{-1} mod\ 2^k}$

Variables Intermédiaires :

q_i

$$Z_i = \sum_{j=0}^{d-1} Z_{i,j} \times 2^{j \times k} = \sum_{j=0}^{d-1} (C2_{i,j} \times 2^k + Z1_{i,j}) \times 2^{j \times k}$$

$$\text{où } C2_{i,j} = Z_{i,j}[k+1, k],\ Z1_{i,j} = Z_{i,j}[k-1, 0]$$

$$SP_i = \sum_{j=0}^{d-1} SP_{i,j} \times 2^{j \times k} = \sum_{j=0}^{d-1} (SP2_{i,j} \times 2^k + SP1_{i,j}) \times 2^{j \times k}$$

$$\text{où } SP2_{i,j} = SP_{i,j}[2k-1, k] \text{ et } SP1_{i,j} = SP_{i,j}[k-1, 0]$$

$$S_i = \sum_{j=0}^{d} S_{i,j} \times 2^{j \times k} = \sum_{j=0}^{d} (C1_{i,j} \times 2^k + S1_{i,j}) \times 2^{j \times k}$$

$$\text{Où } S1_{i,j} = S_{i,j}[2k-1, k];\ C1_{i,j} = S_{i,j}[k-1, 0]$$

$$\overline{MemQ_i} = \sum_{j=0}^{d-1} \overline{MemQ}_{i,j} \times 2^{j \times k},\ \underline{MemQ_i} = \sum_{j=0}^{d-1} \underline{MemQ_{i,j}} \times 2^{j \times k}$$

Fonctions : *H, G, Mem;*

Sortie : $S = \sum_{j=0}^{d-1} S_{d-1,j} \times 2^{j \times k} = \boldsymbol{(X \times Y \times 2^{d-1}) mod\ M}$

Début

pour i ***de*** 0 à d ***faire***

pour j de 0 à $\frac{d}{2}$*-1 faire*

$SP_{i,j} = Y_i \times X_j$ *//en parallèle*

fin pour;

$SP1_{i,-1} = SP1_{i,\frac{d}{2}} = S1_{i,d+1} = 0$

$Z_{i,0} = SP1_{i,0} + S1_{i-1,0} + reg$;

pour j de 1 à $\frac{d}{2}$ *faire*

$Z_{i,j} = SP2_{i,j-1} + SP1_{i,j} + C1_{i,j-1} + S1_{i,j}$ *//en parallèle*

fin pour;

pour j de $\frac{d}{2} + 1$ *à d faire*

$Z_{i,j} = C1_{i,j-1} + S1_{i,j}$ *//en parallèle*

fin pour;

if $\left(\left(Z_{i,0} \times M_j\left[\frac{k}{2} - 1{,}0\right]\right)\left[\frac{k}{2} - 1{,}0\right] == 0\right)$ $c = 0$ ***else*** $c = 1$; ***end if;***

$q_{sup} = Z_{i,0}\left[k - 1, \frac{k}{2}\right] + \left(\left(\left(Z_{i,0}\left[\frac{k}{2} - 1{,}0\right] \times M'\left[k - 1, \frac{k}{2}\right]\right) \times M_0\left[k - 1, \frac{k}{2}\right]\right)\left[k - 1, \frac{k}{2}\right]\right) + c$

$\overline{MemQ_i} := H(q_{sup})$; $\underline{MemQ_i} := H(Z_{i,0}\left[\frac{k}{2} - 1{,}0\right])$;

pour j de 0 à d-1 faire

$S_{i+1,j} = \overline{MemQ_{i,j}} + \underline{MemQ_{i,j}} + Z1_{i,j} + C2_{i,j-1}$; *//en parallèle*

fin pour;

$S_{i+1} = S_{i+1} \times 2^{-k}$;

fin pour ;

Retourne $S=S_{i+1}$;

fin

La figure suivante représente l'exécution d'une itération dans MMM_SNR_2^k dans le cas d'utilisation d'une mémoire.

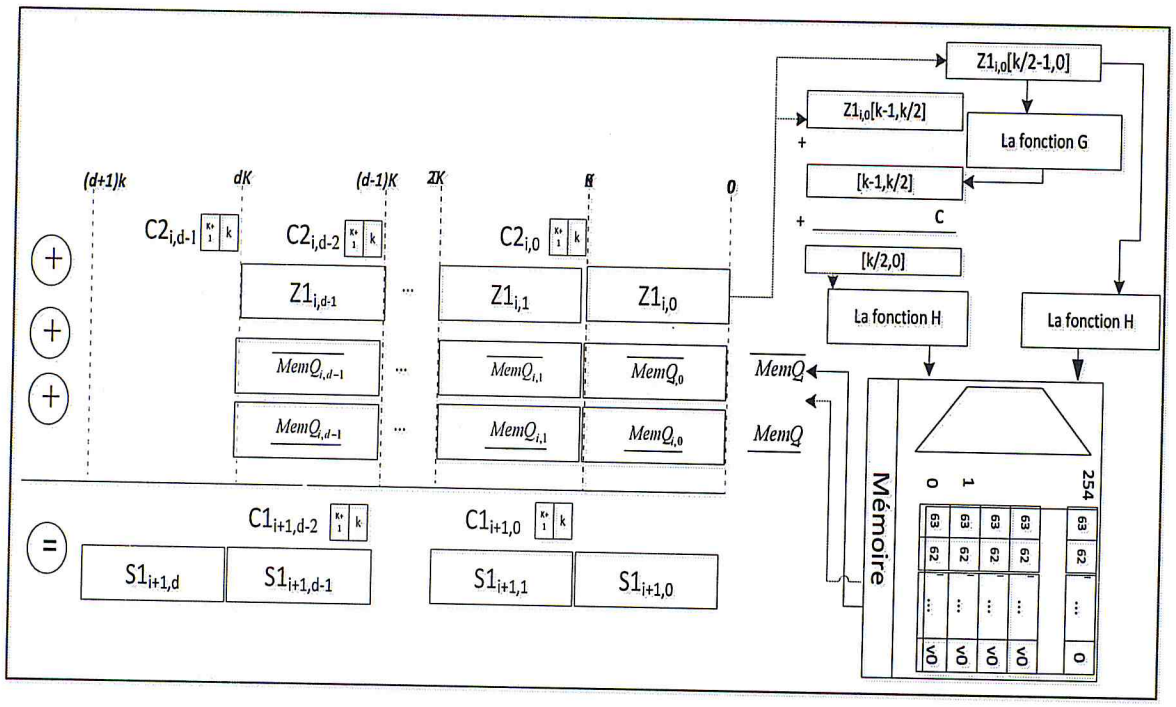


Figure 3.4 : Exécution de la 2ème étape d'une itération i par l'algorithme MMMSNR_2^k avec une mémoire

III.4. Algorithme de la multiplication modulaire de Barrett

III.4.1.Système de numération redondante signée en base 2^k :

Un nombre redondant signé de d chiffres en base 2^k est un nombre de bits égal $d \times (k+2)$ bits et il est composé par $\boldsymbol{d}$ mots chacun de taille $(k+2+1)$ $bits$ qui représente successivement les k $bits$ principaux, les deux bits de retenu et le bit de signe vaut (0 si le mot est positif et 1 si le mot est négatif), chaque nombre (X) est décomposé en deux nombres, un positif X^+ et un négatif X^-.

$X_i[k-1,0]$: $Les\ bits$ principaux , $X_i[k+1,k]$: $Les\ bits$ redondants .

La figure 3.5 représente un nombre redondant signé.

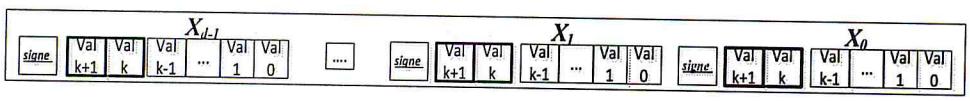


Figure 3.5- représentation d'un nombre redondant signé en base 2^k

Pour transformer un nombre de ce système redondant à la représentation classique on fait la soustraction entre la partie positive et la partie négative $(X = X^+ - X^-)$.

Dans le cas contraire, c'est-à-dire pour transformer un nombre positif non redondant à la représentation redondant signée en base 2^k, On découpe ce résultat à un ensemble des mots de k bits, on ajoute pour chacun deux bits de retenu égale à 0 et un bit de signe qui vaut 0.

L'utilisation de ce bit est plus importante pour calculer la soustraction entre les mots de deux nombres redondants en parallèle.

Exemple

Soit deux nombres redondants A et B qui sont

$A = 10\ 1101\ \ 11\ 1010 \qquad B = 01\ 1001\ \ 10\ 0100$

nous calculons $P := A - B\ et\ N = B - A$.

La représentation redondante de ces nombres en base 2^4 est la suivante :

Les figures **3.6** et **3.7** représentent respectivement les opérations $P = A - B\ \ et\ N = B - A$

- Calculer $P = A - B = 310$

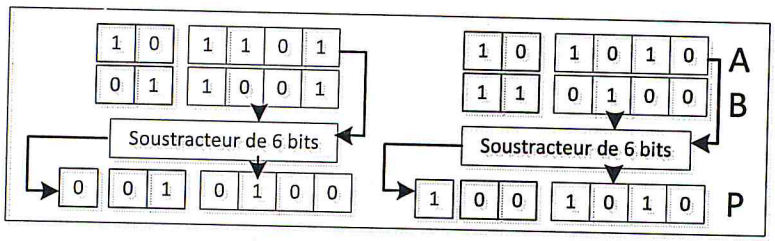


Figure 3.6- Calculer le résultat P.

- Calculer $N = B - A = -310$

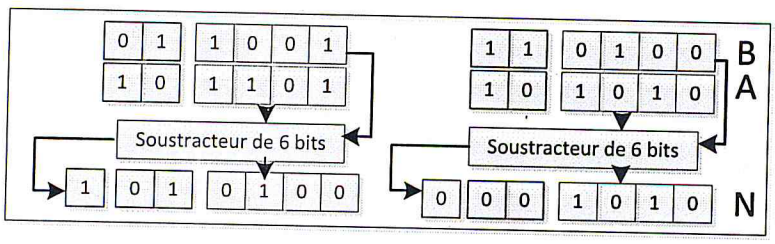


Figure 3.7- Calculer le résultat N.

Le résultat P (N dans la deuxième cas) est un nombre redondant de deux mots en base 2^k composé par deux nombres , un nombre positif P^+ (N^+) et un nombre négatif P^- (N^-), Il peut être représenté par l'expression suivant $P = P^+ - P^-$ ($N = N^+ - N^-$)

P= 0 01 0100 1 00 1010 et P^+= 01 0100 00 0000 et P^-= 00 0000 00 1010

P = $P^+ - P^-$ = (01 0100 00 0000) - (00 0000 00 1010) = 100110110

III.4.2. Algorithme de MMB_SNR _2^k général :

L'algorithme général de Barrett SNR en base 2^k consiste à calculer le résultat $(Y \times X) mod\ M$ où les opérandes X, Y et le modulo M ont la même taille et ils sont transformés à la représentation redondante signée en base 2^k :

$$X = \sum_{j=0}^{d-1} X_j \times 2^{j\times k} = \sum_{j=0}^{d-1} X_j^+ \times 2^{j\times k}, \quad Y = \sum_{i=0}^{d-1} Y_i \times 2^{i\times k} = \sum_{i=0}^{d-1} Y_i^+ \times 2^{i\times k}$$

$$M = \sum_{i=0}^{d-1} M_i \times 2^{k\times i} = \sum_{i=0}^{d-1} M_i^+ \times 2^{k\times i}, \qquad R = 2^{d\times k} \ et\ R^{-1} = 2^{-(d\times k)}$$

Aussi le résultat de chaque itération S_i est représenté dans ce système :

$$S_i = \sum_{j=0}^{d-1} S_{i,j} \times 2^{j\times k} = \sum_{j=0}^{d-1} S_j^+ \times 2^{j\times k} - \sum_{j=0}^{d-1} S_j^- \times 2^{j\times k}$$

Barrett calcule S selon l'expression suivante :

$$S = (X \times Y) mod\ M$$

$$= \left(\left(\sum_{i=0}^{d-1} Y_i \times 2^{i\times k}\right) \times X\right) mod\ M$$

$$= (..((Y_{d-1} \times X)2^k + Y_{d-2} \times X)2^k + \cdots + Y_i \times X)2^k + \cdots) + Y_0 \times X)\ mod\ M$$

$$S = (...((Y_{d-1} \times X\ mod\ M) \times 2^k + Y_{d-2} \times X) mod\ M) \times 2^k ... + Y_i \times X) mod\ M) \times 2^k + \cdots) mod\ M) \times 2^k + Y_0 \times X)\ mod\ M$$

Ce qui signifie que la multiplication de Barrett peut être calculée en utilisant la formule de récurrence suivante :

$$\begin{cases} S_d = 0 \\ S_{i+1} = \left(S_i \times 2^k + (Y_i \times X)\right) mod\ M, \ où\ i = d-1, ..., 0. \end{cases}$$

Cette formule signifie qu'à chaque itération un produit partiel est généré, puis ajouter au résultat précédent qui est multiplie par le facteur 2^k, puis calculer le quotient approximé sans calculer la division de $\frac{S_i}{M}$. En fin, pour trouver le reste de la division par M le calcule de la soustraction entre S_i et le résultat de $\widehat{q_i}$×M est nécessaire. C'est le principe de l'algorithme de Barrett :

$$S_{i+1} = \left(\left(S_i \times 2^k + (Y_i \times X)\right) - (\widehat{q_i} \times M)\right) \quad avec\ S_{i+1} < 2^k M.$$

Dans cette équation les nombres $X, Y_i\ et\ M$, $\widehat{q_i}$ sont des nombres redondants signés où leur partie négative égale 0. Mais le résultat de l'itération précédente $S_i$est un nombre redondant signé à deux parties : positifs et négatif.

L'équation devient alors

$$S_{i+1} = \left(\left(\left(\sum_{j=0}^{d-1} S_{i,j}^{+} \times 2^{j\times k} - \sum_{j=0}^{d-1} S_{i,j}^{-} \times 2^{j\times k}\right) \times 2^{k} + \left(Y_i \times \sum_{j=0}^{d-1} X_j \times 2^{j\times k}\right)\right) - \left(\widehat{q_1} \times \sum_{j=0}^{d-1} M_j \times 2^{j\times k}\right)\right)$$

Ce résultat est calculé en deux étapes :

La première permet de calcule le résultat intermédiaire Z_i à partir de l'équation suivante :

$$Z_i = S_i \times 2^k + Y_i \times X$$
$$= \left(\left(\sum_{j=0}^{d-1} S_{i,j}^{+} \times 2^{j\times k}\right) \times 2^{k} - \left(\sum_{j=0}^{d-1} S_{i,j}^{-} \times 2^{j\times k}\right) \times 2^{k} + \left(Y_i \times \sum_{j=0}^{d-1} X_j \times 2^{j\times k}\right)\right)$$

La seconde permet de déterminer S_{i+1} à partir de l'équation : $S_{i+1} = (Z_i - (\widehat{q_1} \times M))$

En effet $\widehat{q_1} \cong \left\lfloor \frac{Z_i}{M} \right\rfloor$ est calculé à partir de l'équation $\left\lfloor \frac{\left\lfloor \frac{Z_i}{2^n} \right\rfloor \mu}{2^{k+2}} \right\rfloor$ avec $\mu = \left\lfloor \frac{2^{n+k+2}}{M} \right\rfloor$ est une valeur pré-calculée, Ce qui signifie qu'il est trouvé à partir de deux chiffres les plus significatifs de Z_i, en conséquence ce dernier doit être transformé à la représentation classique ce qui nécessite un temps supplémentaire important.

Pour éviter ce problème, notre approche consiste à calculer les quotients de deux parties positif et négatif de Z_i en parallèle, puis faire la réduction vers le modulo M:

$$S_{i+1} = (Z_i - \widehat{q_1} \times M)$$
$$= \left(\left(S_i^{+} \times 2^{k}\right) + \left(Y_i \times X\right)\right) - \left(S_i^{-} \times 2^{k}\right) - \left(\widehat{q_1} \times M\right)$$
$$= \left(\left(S_i^{+} \times 2^{k}\right) + \left(Y_i \times X\right)\right) - \left(S_i^{-} \times 2^{k}\right) - \left(\left\lfloor \frac{\left(\left(S_i^{+} \times 2^{k}\right) + \left(Y_i \times X\right)\right)}{M} \right\rfloor - \left\lfloor \frac{\left(S_i^{-} \times 2^{k}\right)}{M} \right\rfloor\right) \times M$$
$$= \left(\left(S_i^{+} \times 2^{k}\right) + \left(Y_i \times X\right)\right) + \left\lfloor \frac{\left(S_i^{-} \times 2^{k}\right)}{M} \right\rfloor \times M - \left(\left(S_i^{-} \times 2^{k}\right) + \left\lfloor \frac{\left(\left(S_i^{+} \times 2^{k}\right) + \left(Y_i \times X\right)\right)}{M} \right\rfloor \times M\right)$$

On peut noter : $q_i^{-} = \left\lfloor \frac{\left(S_i^{-} \times 2^{k}\right)}{M} \right\rfloor$ et $q_i^{+} = \left\lfloor \frac{\left(\left(S_i^{+} \times 2^{k}\right) + \left(Y_i \times X\right)\right)}{M} \right\rfloor$ ce que signifie :

$$S_i = \left(\left(S_i^{+} \times 2^{k}\right) + \left(Y_i \times X\right)\right) + q_i^{-} M - \left(\left(S_i^{-} \times 2^{k}\right) + q_i^{+} M\right) \text{ Où } S_{i+1} \leq M$$

On a déjà montré dans le chapitre précédant que le quotient calculé avec l'algorithme de Barrett selon l'approche de *Dhem*[**Mir, 09**] est un quotient approximé varie entre : $q - 1 \leq \hat{q} \leq q$ selon cette propriété : $q_i^{-} - 1 \leq \hat{q}_i^{-} \leq q_i^{-}$ Et $q_i^{+} - 1 \leq \hat{q}_i^{+} \leq q_i^{+}$.

L'utilisation de ces deux quotients dans notre expression introduit quatre possibilités qui sont :

$\hat{q}_i^- = q_i^- - 1$ Et $\hat{q}_i^+ = q_i^+ - 1$ ➔ $0 < S_{i+1} < M$

$\hat{q}_i^- = q_i^-$ Et $\hat{q}_i^+ = q_i^+$ ➔ $0 < S_{i+1} < M$

$\hat{q}_i^- = q_i^-$ et $\hat{q}_i^+ = q_i^+ - 1$ ➔ $0 < S_{i+1} < 2^k M$

$\hat{q}_i^- = q_i^- - 1$ Et $\hat{q}_i^+ = q_i^+$ ➔ $-M < S_{i+1} < M$.

Pour éviter le dernier cas, on a besoin d'ajouter M à l'expression :

$$S_{i+1} = \left((S_i^+ \times 2^k) + (Y_i \times X)\right) + \hat{q}_i^- M - \left((S_i^- \times 2^k) + \hat{q}_i^+ M\right) + M$$

$$= \left((S_i^+ \times 2^k) + (Y_i \times X)\right) + (\hat{q}_i^- + 1) \times M - \left((S_i^- \times 2^k) + \hat{q}_i^+ M\right) \text{bits.}$$

L'expression de Z_i sera alors,:

$$Z_i = \left(S1_{i,0}^+ + SP1_{i,0}\right) + \left(S1_{i,1}^+ + CS1_{i,0}^+ + SP2_{i,0} + SP1_{i,1}\right) \times 2^k + \cdots + \left(S1_{i,j}^+ + CS1_{i,j-1}^+ + SP2_{i,j-1} + SP1_{i,j} \times 2j \times k + .. + S1_{i,d-1} + + CS1_{i,d-2} + + SP2_{i,d-1} \times 2d \times j + CS1_{i,d-1} + \times 2d + 1 \times j\right.$$

Ce que signifié que Z_i peut être calculé en utilisant la formule de récurrence suivante :

$$\begin{cases} SP_{i,-1} = SP_{i,-2} = S1_{i,d} = 0, \\ Z_{i,j} = C2_{i,j} \times 2^k + Z1_{i,j} = \left(S1_{i,j}^+ + CS1_{i,j-1}^+ + SP2_{i,j-1} + SP1_{i,j}\right) \times 2^{j \times k} \quad \forall j, \ 0 \le j \le d \end{cases}$$

Après l'obtention de Z_i, on passe au calcul de S_i, ce dernier est calculé en fonction du q_i qu'est codé sur k+2 bits.

$$S_{i+1} = (Z_i + \hat{q}_i'^- \times M) - \left(S_i^- + (\hat{q}_i^+ \times M)\right)$$

Cette expression est constituée par deux opérations $(Z_i + \hat{q}_i'^- \times M)$ et $\left(S_i^- + (\hat{q}_i^+ \times M)\right)$ chaque opération est une multiplication suivie par une addition, ces deux opérations sont calculées en parallèle, la soustraction entre ses résultats représente le résultat S_i.

Pour une architecture utilise un nombre optimum de ressource avec un temps d'exécution minimum, l'utilisation d'une mémoire qui stocke la valeur de $(\hat{q}_i^+ \times M)$ *et* $(\widehat{q'}_i^- \times M)$ Avec la lecture de l'adresse $\hat{q}_i^+, \widehat{q'}_i^-$ de la mémoire on obtient la valeur de $(\hat{q}_i^+ \times M)$ *et* $(\hat{q}_i'^- \times M)$en 1 cycle d'horloge.

Il est à noter que $\hat{q}_i^+$ et $\hat{q}_i'^-$ sont codé sur k+2, dans notre architecture k+2=18 bits.

Pour stoker les valeurs $(\hat{q}_i^+ \times M)$ *et* $(\hat{q}_i'^- \times M)$ dans une mémoire, on a besoin d'utiliser 133 Mbits. Pour éviter ce problème la division de $\hat{q}_i^+$et $\hat{q}_i'^-$ en deux parties de $\frac{k+2}{2}$ est une solution parfaite pour réduire la taille de mémoire.

$$\begin{cases} \hat{q}_i^+ = \overline{\hat{q}_i^+} \times 2^{\frac{k+2}{2}} + \underline{\hat{q}_i^+} \text{ où } \overline{\hat{q}_i^+} = \hat{q}_i^+ \left[k+2, \frac{k}{2}+2\right] \text{ et } \underline{\hat{q}_i^+} = \hat{q}_i^+ \left[\frac{k}{2}+1{,}0\right] \\ \hat{q}_i'^- = \overline{\hat{q}_i'^-} \times 2^{\frac{k+2}{2}} + \underline{\hat{q}_i'^-} \text{ où } \overline{\hat{q}_i'^-} = \hat{q}_i'^- \left[k+2, \frac{k}{2}+2\right] \text{ et } \underline{\hat{q}_i'^-} = \hat{q}_i'^- \left[\frac{k}{2}+1{,}0\right] \end{cases}$$

On peut calculer $\overline{\hat{q}_i^+}$, $\underline{\hat{q}_i^+}$, $\overline{\hat{q}_i'^-}$ *et* $\underline{\hat{q}_i'^-}$ en parallèle .

Cette méthode permet de minimiser la capacité mémoire nécessaire à $133kbits$ dans une mémoire double ports. De cette manière nous pouvons utiliser une seule mémoire pour stocker $\underline{\hat{q}_i^+} \times M$ et $\underline{\hat{q}_i'^-} \times M$ de Barrett et $\overline{q} \times M$ et $\underline{q} \times M$ de Montgomery

On suppose que :

$$\begin{cases} \overline{MemQ_i^+} = \overline{\hat{q}_i^+} \times M \times 2^{\frac{k+2}{2}} & \text{et} \quad \underline{MemQ_i^+} = \underline{\hat{q}_i^+} \times \mathrm{M} \\ \overline{MemQ_i^-} = \overline{\hat{q}_i'^-} \times M \times 2^{\frac{k+2}{2}} & \text{et} \quad \underline{MemQ_i^-} = \underline{\hat{q}_i'^-} \times \mathrm{M} \end{cases}$$

L'utilisation de deux variables Spos_i, Sneg_i qui prennent les valeurs suivants :

$$\begin{cases} \mathrm{Z1_{i,d}} = 0 \\ \mathrm{Spos}_{i,j} = \left(\mathrm{Z1_{i,j}} + \mathrm{C2_{i,j-1}} + \underline{MemQ_{i,j}^-} + \overline{MemQ_{i,J}^-}\right) \ \forall \mathrm{j}, \ 0 \le \mathrm{j} \le \mathrm{d} \end{cases}$$

$$\begin{cases} \mathrm{S1_{i,d}^-} = 0 \\ \mathrm{Sneg}_{i,j} = \left(\mathrm{S1_{i,j}^-} + \mathrm{CS1_{i,j-1}^-} + \underline{MemQ_{i,j}^+} + \overline{MemQ_{i,J}^+}\right) \ \forall \mathrm{j}, \ 0 \le \mathrm{j} \le \mathrm{d} \end{cases}$$

Permet de calculer le résultat final de l'itération (i) dans un temps constant :

$$\mathbf{S_{i+1,j}} = \mathrm{Spos}_{i,j} - \mathrm{Sneg}_{i,j} \ \forall \mathbf{j}, \ \mathbf{0 \le j \le d}$$

L'algorithme 3.2 représente cet algorithme.

Algorithme 3.2 : l'algorithme Général de la MMBSNRS-$\mathbf{2^k}$ avec l'utilisation de la mémoire

Entrée : $\boldsymbol{X = \sum_{j=0}^{d-1} X_j \times 2^{j\times k}}$, $Y = \sum_{i=0}^{d-1} Y_i \times 2^{i\times k}$, $M = \sum_{j=0}^{d-1} M_j \times 2^{j\times k}$.

Pré calculés : $\mu = \left\lfloor \frac{2^{n+k+3}}{M} \right\rfloor$ *où* $\lfloor A \rfloor$: *La partie entière inférieure de A*

Variables Intermédiaire :

$$\mathrm{Z_i} = \sum_{\mathrm{j=0}}^{\mathrm{d-1}} \mathrm{Z_{i,j}} \times 2^{\mathrm{j\times k}} = \sum_{\mathrm{j=0}}^{\mathrm{d-1}} \left(\mathrm{C2_{i,j}} \times 2^{\mathrm{k}} + \mathrm{Z1_{i,j}}\right) \times 2^{\mathrm{j\times k}}$$

$$\text{où} \quad \mathrm{C2_{i,j}} = \mathrm{Z_{i,j}}[k+1, k], \ \mathrm{Z1_{i,j}} = \mathrm{Z_{i,j}}[k-1{,}0]$$

$$SP_{\mathrm{i}} = \sum_{\mathrm{j=0}}^{\mathrm{d-1}} SP_{\mathrm{i,j}} \times 2^{\mathrm{j\times k}} = \sum_{\mathrm{j=0}}^{\mathrm{d-1}} \left(SP2_{\mathrm{i,j}} \times 2^{\mathrm{k}} + SP1_{\mathrm{i,j}}\right) \times 2^{\mathrm{j\times k}}$$

$$\text{où} \quad SP2_{\mathrm{i,j}} = SP_{\mathrm{i,j}}[2k-1, k] \text{et} \ SP1_{\mathrm{i,j}} = SP_{i,j}[k-1, 0]$$

$$\mathrm{S_{i-1}^+} = \sum_{j=0}^{d-1} \mathrm{S_{i-1,j}^+} \times 2^{j\times k} = \sum_{j=0}^{d-1} \left(\mathrm{CS1_{i,j}^+} \times 2^{\mathrm{k}} + \mathrm{S1_{i-1,j}^+}\right) \times 2^{j\times k}$$

$$\text{Où} \quad \mathrm{CS1_{i,j}^+} = \mathrm{S_{i-1,j}^+}[2k-1, k]; \quad \mathrm{S1_{i-1,j}^+} = \mathrm{S_{i-1,j}^+}[k-1{,}0]$$

$$S^{-}_{i-1} = \sum_{j=0}^{d-1} S^{-}_{i-1,j} \times 2^{j\times k} = \sum_{j=0}^{d-1} (CS1^{-}_{i,j} \times 2^{k} + S1^{-}_{i-1,j}) \times 2^{j\times k}$$

$$\text{Où } CS1^{-}_{i,j} = S^{-}_{i-1,j}[2k-1,k];\ S1^{-}_{i-1,j} = S^{-}_{i-1,j}[k-1,0]$$

$$S_{i-1} = \sum_{j=0}^{d-1} S_{i-1,j} \times 2^{j\times k} = \sum_{j=0}^{d-1} (C1_{i,j} \times 2^{k} + S1_{i-1,j}) \times 2^{j\times k}$$

$$\text{Où } C1_{i,j} = S_{i-1,j}[2k-1,k];\ S1_{i-1,j} = S_{i-1,j}[k-1,0]$$

$Spos_i = \sum_{j=0}^{d-1} Spos_{i,j} \times 2^{j\times k}$ où $\forall j, Spos_{i,j} < 2^{k+2}$

$Sneg_i = \sum_{j=0}^{d-1} Sneg_{i,j} \times 2^{j\times k}$ où $\forall j, Sneg_{i,j} < 2^{k+2}$

$$\overline{MemQ^{+}_{i}} = \sum_{j=0}^{d-1} \overline{MemQ^{+}_{i,j}} \times 2^{j\times k},\ \underline{MemQ^{+}_{i}} = \sum_{j=0}^{d-1} \underline{MemQ^{+}_{i,j}} \times 2^{j\times k}$$

$$\overline{MemQ^{-}_{i}} = \sum_{j=0}^{d-1} \overline{MemQ^{-}_{i,j}} \times 2^{j\times k},\ \underline{MemQ^{-}_{i}} = \sum_{j=0}^{d-1} \underline{MemQ^{-}_{i,j}} \times 2^{j\times k}$$

Sortie : $S=\sum_{j=0}^{d-1} S_{d-1,j} \times 2^{j\times k} = (X \times Y) mod\ (3M)$.

Début

pour *i* ***de*** $d-1$ à 0 ***faire***

pour j de 0 à d-1 faire

$SP_{i,j} = Y_i \times X_j$ *//en parallèle*

fin pour

$SP_{i,-1} = SP_{i,-2} = S1_{i,d} = 0$

pour j de 0 à d-1 faire

$Z_{i,j} = (S1^{+}_{i,j} + CS1^{+}_{i,j-1} + SP2_{i,j-1} + SP1_{i,j})$ *//en parallèle*

fin pour

$q^{+}_{i} = \left\lfloor \frac{(Z1_{i,d-1} + C2_{i,d-2}) \times \mu}{2^{k+2}} \right\rfloor$; $q^{-}_{i} = \left\lfloor \frac{(S1^{-}_{i,d-1} + CS1^{-}_{i-1,d-2}) \times \mu}{2^{k+2}} \right\rfloor$;

$\overline{MemQ^{+}_{i}} = H\left(q^{+}_{i}\left[k-1, \frac{k}{2}\right]\right) \times 2^{\frac{k+2}{2}}$; $\overline{MemQ^{-}_{i}} = H\left(q^{-}_{i}\left[k-1, \frac{k}{2}\right]\right) \times 2^{\frac{k+2}{2}}$;

$\underline{MemQ^{-}_{i}} := H\left(q^{-}_{i}\left[\frac{k}{2}-1,0\right]\right)$; $\underline{MemQ^{+}_{i}} := H(q^{+}_{i}\left[\frac{k}{2}-1,0\right])$;

pour j de 0 à d-1 faire

$Spos_{i,j} = \left(Z1_{i,j} + C2_{i,j-1} + \underline{MemQ^{-}_{i,j}} + \overline{MemQ^{-}_{i,j}}\right)$

$Sneg_{i,j} = \left(S1^{-}_{i,j} + CS1^{-}_{i,j-1} + \underline{MemQ^{+}_{i,j}} + \overline{MemQ^{+}_{i,j}}\right)$ *//en parallèle*

fin pour

pour j de 0 à d-1 faire

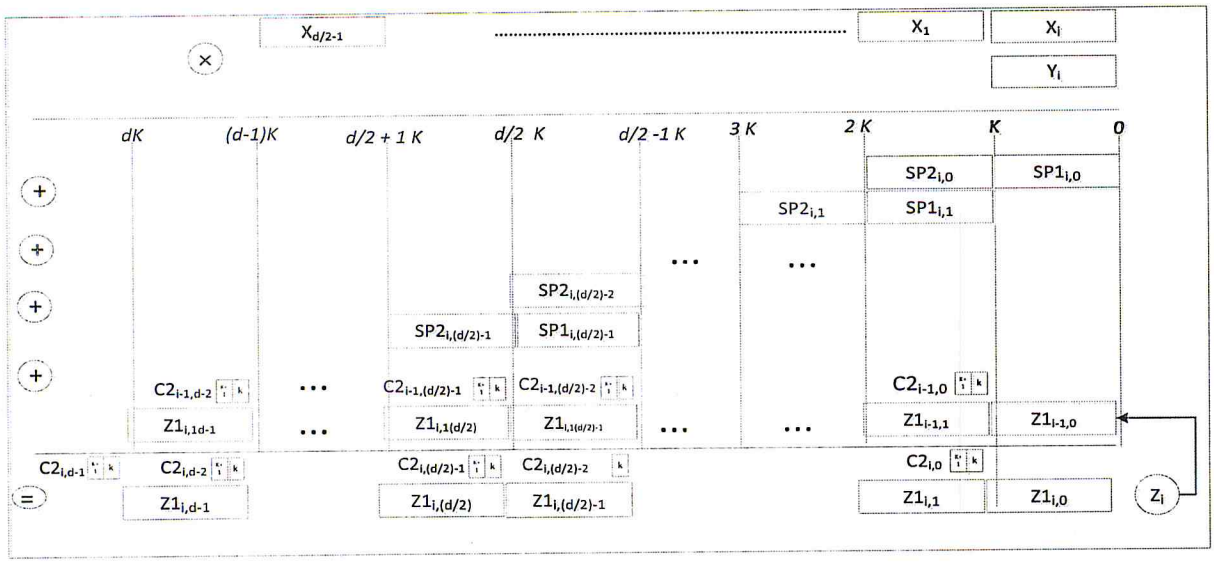


Figure 3.8- LA multiplication en SNR_2^k

Et dans la deuxième étape, on calcule tous les mots Spos et Sneg en parallèle par l'équation suivante :

$$\begin{cases} \mathrm{Spos}_{i,j} = \left(\mathrm{Z1}^{+}_{\mathrm{i,j}} + \mathrm{CZ1}^{+}_{\mathrm{i,j-1}} + \overline{MemQ^{-}_{i,j}} + \underline{MemQ^{-}_{i,j}}\right) \\ \mathrm{Sneg}_{i,j} = \left(\mathrm{Z1}^{-}_{\mathrm{i,j}} + \mathrm{CZ1}^{-}_{\mathrm{i,j-1}} + \overline{MemQ^{+}_{i,j}} + \underline{MemQ^{+}_{i,j}}\right) \end{cases}$$

La dernière étape consiste à calculer la soustraction suivante :

$$S_{\mathrm{i,j}} = \mathrm{Spos}_{i,j} - \mathrm{Sneg}_{i,j} .$$

L'algorithme MMB_SNR_2^k est représenté dans l'algorithme 3.3.

Algorithme 3.3 : l'algorithme Spécial de la MMBSNRS-2^k avec l'utilisation de la mémoire

Entrée : $X = \sum_{j=0}^{d-1} X_j \times 2^{j\times k}$, $Y = \sum_{i=0}^{d-1} Y_i \times 2^{i\times k}$, $M = \sum_{j=0}^{d-1} M_j \times 2^{j\times k}$.

Pré calculés : $\mu = \left\lfloor \frac{2^{n+k+3}}{M} \right\rfloor$ *où* $\lfloor A \rfloor$: *La partie entière inférieure de A*

Variables Intermédiaire :

$$\mathrm{Z_i} = \sum_{\mathrm{j=0}}^{\mathrm{d-1}} \mathrm{Z_{i,j}} \times 2^{\mathrm{j\times k}} = \sum_{\mathrm{j=0}}^{\mathrm{d-1}} \left(\mathrm{C2_{i,j}} \times 2^{\mathrm{k}} + \mathrm{Z1_{i,j}}\right) \times 2^{\mathrm{j\times k}}$$

$$SP_{\mathrm{i}} = \sum_{\mathrm{j=0}}^{\mathrm{d-1}} SP_{\mathrm{i,j}} \times 2^{\mathrm{j\times k}} = \sum_{\mathrm{j=0}}^{\mathrm{d-1}} \left(SP2_{\mathrm{i,j}} \times 2^{\mathrm{k}} + SP1_{\mathrm{i,j}}\right) \times 2^{\mathrm{j\times k}}$$

$$\mathrm{S_i} = \sum_{j=0}^{d-1} \mathrm{S_{i,j}} \times 2^{j\times k} = \sum_{j=0}^{d-1} (\mathrm{C1_{i,j}} \times 2^{\mathrm{k}} + \mathrm{S1_{i,j}}) \times 2^{j\times k}$$

$$Spos_{\mathrm{i}} = \sum_{j=0}^{d-1} Spos_{\mathrm{i},j} \times 2^{j\times k} \ , Sneg_{\mathrm{i}} = \sum_{j=0}^{d-1} Sneg_{\mathrm{i},j} \times 2^{j\times k}$$

$$\overline{MemQ^{+}_{i}} = \sum_{j=0}^{d-1} \overline{MemQ^{+}_{i,j}} \times 2^{j\times k} \ , \underline{MemQ^{+}_{i}} = \sum_{j=0}^{d-1} \underline{MemQ^{+}_{i,j}} \times 2^{j\times k}$$

$\overline{MemQ_i^-} = \sum_{j=0}^{d-1} \overline{MemQ_{i,j}^-} \times 2^{j\times k}$, $\underline{MemQ_i^-} = \sum_{j=0}^{d-1} \underline{MemQ_{i,j}^-} \times 2^{j\times k}$

Sortie : $SB = \sum_{j=0}^{d-1} S_{d-1,j} \times 2^{j\times k} = (X \times Y) mod\ (3M)$.

Début

$Z_{-1} = \sum_{j=0}^{d-1} Z_{-1,j} \times 2^{j\times k} = 0,$

pour i ***de*** 0 à $d-1$ ***faire***

Pour j de 0 à $\frac{d}{2} - 1$ *faire*

$SP_{i,j} = Y_i \times X_j$ *//en parallèle*

fin pour

$SP_{i,-1} = Z1_{i,d} = 0$

pour j de 0 à d-1 faire

$Z_{i,j} = \left(Z1^+_{i-1,j} + C2^+_{i-1,j-1} + SP2_{i,j-1} + SP1_{i,j}\right)$ *//en parallèle*

fin pour

Fin pour ;

pour i ***de*** 0 à $\frac{d}{2} - 1$ ***faire***

$q_i^+ = \left\lfloor \frac{(Z1^+_{i,d-1} + C2^+_{i,d-2}) \times \mu}{2^{k+2}} \right\rfloor$; $q_i^- = \left\lfloor \frac{(Z1^-_{i,d-1} + C2^-_{i,d-2}) \times \mu}{2^{k+2}} \right\rfloor$;

$\overline{MemQ_i^+} = H\left(q_i^+ \left[k-1, \frac{k}{2}\right]\right) \times 2^{\frac{k+2}{2}}$; $\overline{MemQ_i^-} = H\left(q_i^- \left[k-1, \frac{k}{2}\right]\right) \times 2^{\frac{k+2}{2}}$;

$\underline{MemQ_i^-} := H\left(q_i^- \left[\frac{k}{2} - 1,0\right]\right)$; $\underline{MemQ_i^+} := H(q_i^+ \left[\frac{k}{2} - 1,0\right])$;

pour j de 0 à d-1 faire

$Spos_{i,j} = \left(Z1^+_{i,j} + C2^+_{i,j-1} + \underline{MemQ_{i,j}^-} + \overline{MemQ_{i,j}^-}\right)$

$Sneg_{i,j} = \left(Z1^-_{i,j} + C2^-_{i,j-1} + \underline{MemQ_{i,j}^+} + \overline{MemQ_{i,j}^+}\right)$ *//en parallèle*

fin pour

pour j de 0 à d-1 faire

$S_{i,j} = Spos_{i,j} - Sneg_{i,j}$ *//en parallèle*

fin pour ;

Fin pour,

Retourner

$SB = S_{d-1}$

Fin ;

Comme on a déjà vu les deux opérations $X0 \times Y1$ et $X1 \times Y0$ sont des multiplications. Ils sont calculés dans un temps constant de la même manière que $X1 \times Y1$ Une fois l'exécution des algorithmes est terminée, on obtient les résultats suivants :

SB=($X1 \times Y1 \times 2^{\frac{n}{2}}) mod\ 3M$

SM=$(X0 \times Y0 \times 2^{-\frac{n}{2}}) mod\ 2M$

$P1$ =$X1 \times Y0$ où $P1 < M$

$P2$ =$X0 \times Y1$. où $P2 < M$

Ils sont des nombres redondants en base 2^k tel que SM, P1 et P2 sont des nombres positifs et SB est composé par deux nombres un positif ($Spos$) et l'autre est négatif ($Sneg$).

La somme de ces résultats est

$$\text{Resultat}_{\text{Finale}} = X \times Y \times 2^{(-n/2)}\ mod\ M = (SB + SM + P1 + P2)\ mod\ M$$

$$= \big((Spos + SM + P1 + P2) - sneg\big) mod\ M$$

$$Resultat_{Finale} = \left(\sum_{j=0}^{j=d-1} \left((Spos1_j + CSpos1_j \times 2^k) + (SM1_j + CSM1_j \times 2^k) + P1j + CP1j \times 2k + P2j + CP2j \times 2k - Sneg1j + CSneg1j \times 2k mod\ M\right.\right.$$

III.5 L'architecture de la multiplication modulaire

Dans notre architecteur on a calculé la multiplication modulaire $X \times Y\ mod\ M$ où la taille des deux opérandes et le modulo est égale 1024, dans la représentation redondante en base 2^{16} cette taille nécessite 64 chiffres pour la représenter.

L'architecture que nous avons développée pour l'algorithme parallèle de la multiplication modulaire en représentation redondante est montrée sur la figure **3.9**. Celle-ci constituée d'une unité de stockage et Cinque parties opératives chacun exécute les opérations arithmétiques d'un algorithme qui sont :

- **la partie opérative de Barrett :** Elle calcule SB=$\left(X1 \times Y1 \times 2^{\frac{n}{2}}\right) mod\ (2^k + 1)M$ avec l'algorithme de Barrett.

- **la partie opérative de multiplication 1 :** Elle calcule $P1=X1 \times Y0$ avec l'algorithme de la multiplication en représentation redondante.
- **la partie opérative de multiplication 2 :** Elle calcule $P2=X0 \times Y1$ avec l'algorithme de la multiplication en représentation redondant
- **la** partie **opérative de Montgomery :** Elle calcule $SM = (X0 \times Y0 \times 2^{-\frac{n}{2}}) mod\ M$ avec l'algorithme de Montgomery
- **La dernière partie :** consiste à calculer le résultat final.

Cette architecture est consacrée au calcul la multiplication modulaire en représentation redondante en 4 étapes :

La première consiste au chargement des opérandes $X, Y\ et\ M$ dans la mémoire, la seconde est l'exécution de tous les algorithmes (Montgomery et Barrett et l'algorithme de la multiplication en représentation redondant) et la troisième étape consiste à calculer l'addition puis la réduction en modulo M .

Dans la suite on va présenter les architecteurs de tous les algorithmes utilisés dans notre architecture.

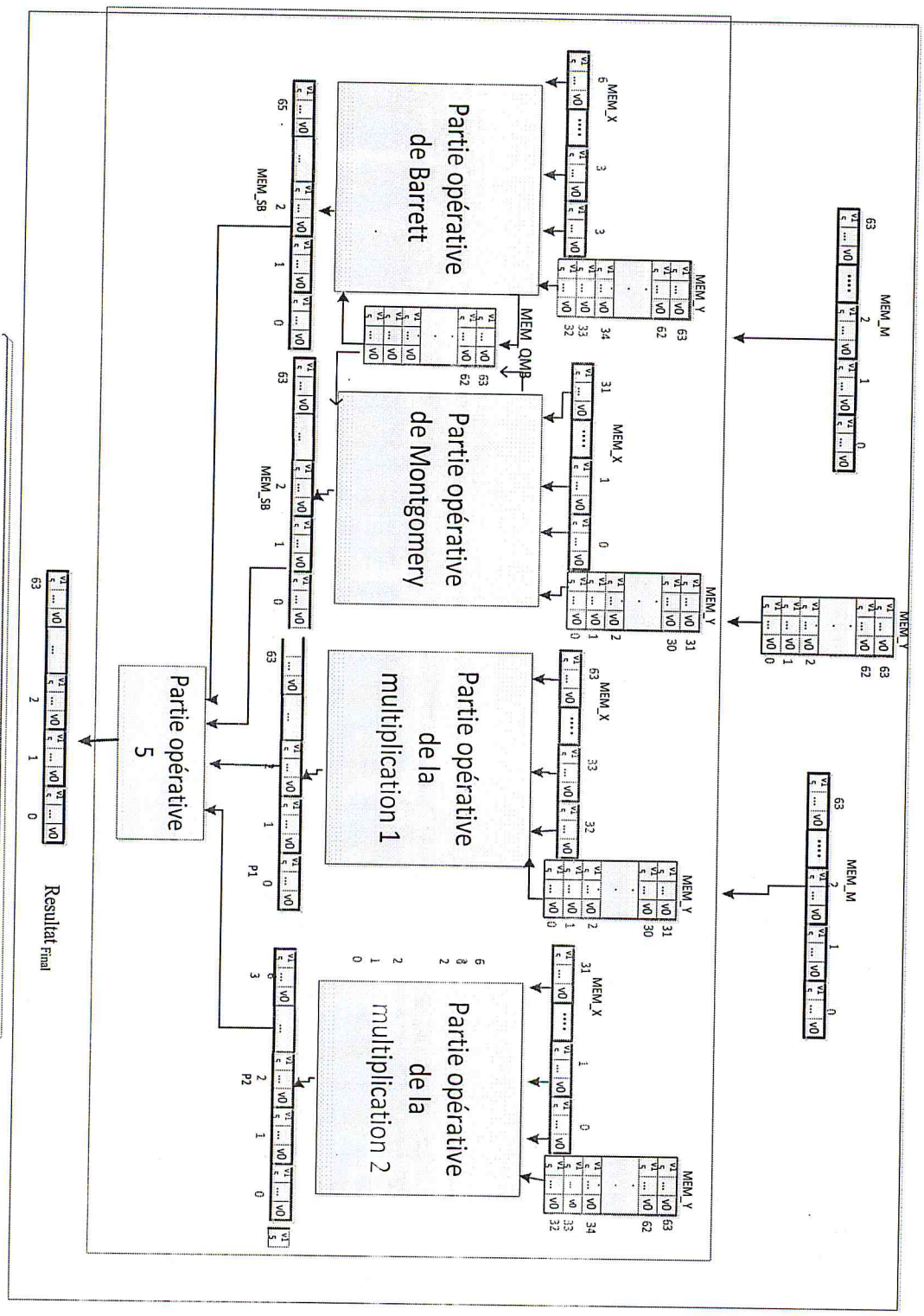


Figure 3.9 : l'algorithme parallèle de la multiplication modulaire en représentation redondante

III.5.1 L'architecture de $MMMSNR\_2^{16}$:

La partie opérative exécute les opérations arithmétiques de $MMMSNR - 2^{16}$, celle-ci conçut de sorte que chaque itération (i) est exécutée en deux étapes :

Dans la première on calcule Z_i à partir de S_{i-1} et Y_i, il calcule tous les mots $\boldsymbol{Z_i}$ en parallèle, Cette opération nécessite 32 multiplieurs 18×18 et 64 additionneurs (2, 16, 16, 16).

Son architecture est représentée dans la figure 3.10

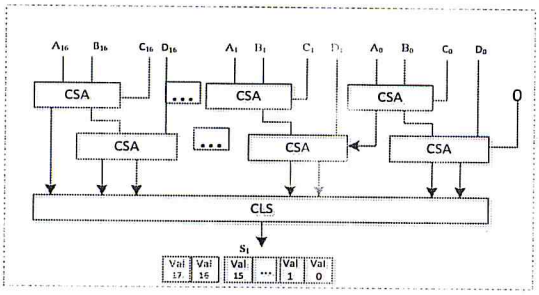


Figure 3.10 – L'additionneur (2,16,16,16)

Et dans la deuxième étape (Cette étape est représentée sur la figure **3.4**), il calcule tout les mots $\boldsymbol{S_{i+1}}$ en parallèle à partir de Z_i et $q_i \times M$. Cette opération nécessite 65 additionneurs (2, 16, 16, 16), qui est présenté au auparavant.

À la fin de l'itération, reg = $C1_{i,0}$ et S_{i+1} , alors $S_{i+1} = S_{i+1} \times 2^{-k}$

III.5.2 L'architecture de $MMBSNRS\_2^{16}$:

La partie opérative exécute les opérations arithmétiques de **MMBSNRS_2^{16}**, l'exécution de cet algorithme est fait en deux boucles principaux :

- Dans la première boucle, l'algorithme calcule la multiplication $Z = X1 \times Y1$, C'est comme la figure 3.12 . Elle nécessite 32 itérations pour le calculer.
- Dans la deuxième boucle, il calcule S=$Z \times 2^{512} mod\ 3M$. Elle nécessite 32 itérations où chaque itération exécute en 3 étapes :

La première consiste à calculer les deux quotients $q^+ et\ q^-$ à partir de $S^+ et\ S^-$ respectivement. La deuxième calcule $S\_pos\ et\ S\_neg$. La dernière consiste à déterminer la soustraction entre ces deux nombres $Spos_{i,j}$ et $Sneg_{i,j}$. La figure 3.12 représente ces étapes.

Le calcule de $Spos\ et\ Sneg$ nécessite l'utilisation des additionneurs de (2, 2, 16, 16) leur architecture est comme l'architecture de l'additionneur (2, 16, 16, 16).

Pour calculer la soustraction on a utilisé notre soustracteur de 18×18, leur architecture est représenter dans la figure 3.11.

La table de vérité est représentée dans le tableau III.1

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Tableau III.1– Table de vérité du soustracteur.

Le soustracteur a 3 entrées, les bits de l'opérandes et un bit de C_{in} et deux sortie S et C_{out}.

$$S = C_{in} \oplus (A \oplus B)$$

$$C_{out} = C_{in}\overline{(A \oplus B)} + \overline{A}B$$

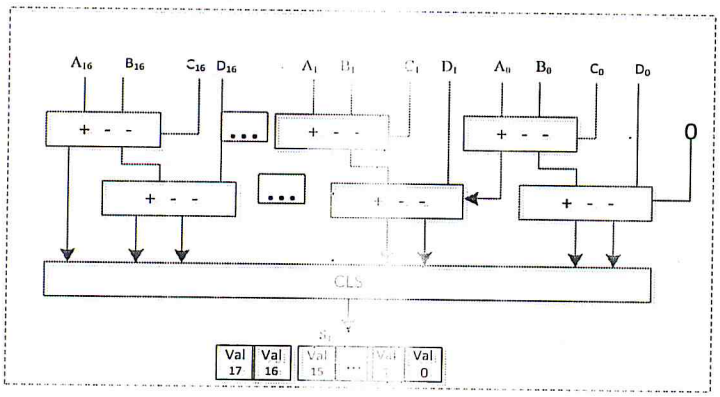


Figure 3.11 – Schéma bloc de soustracteur

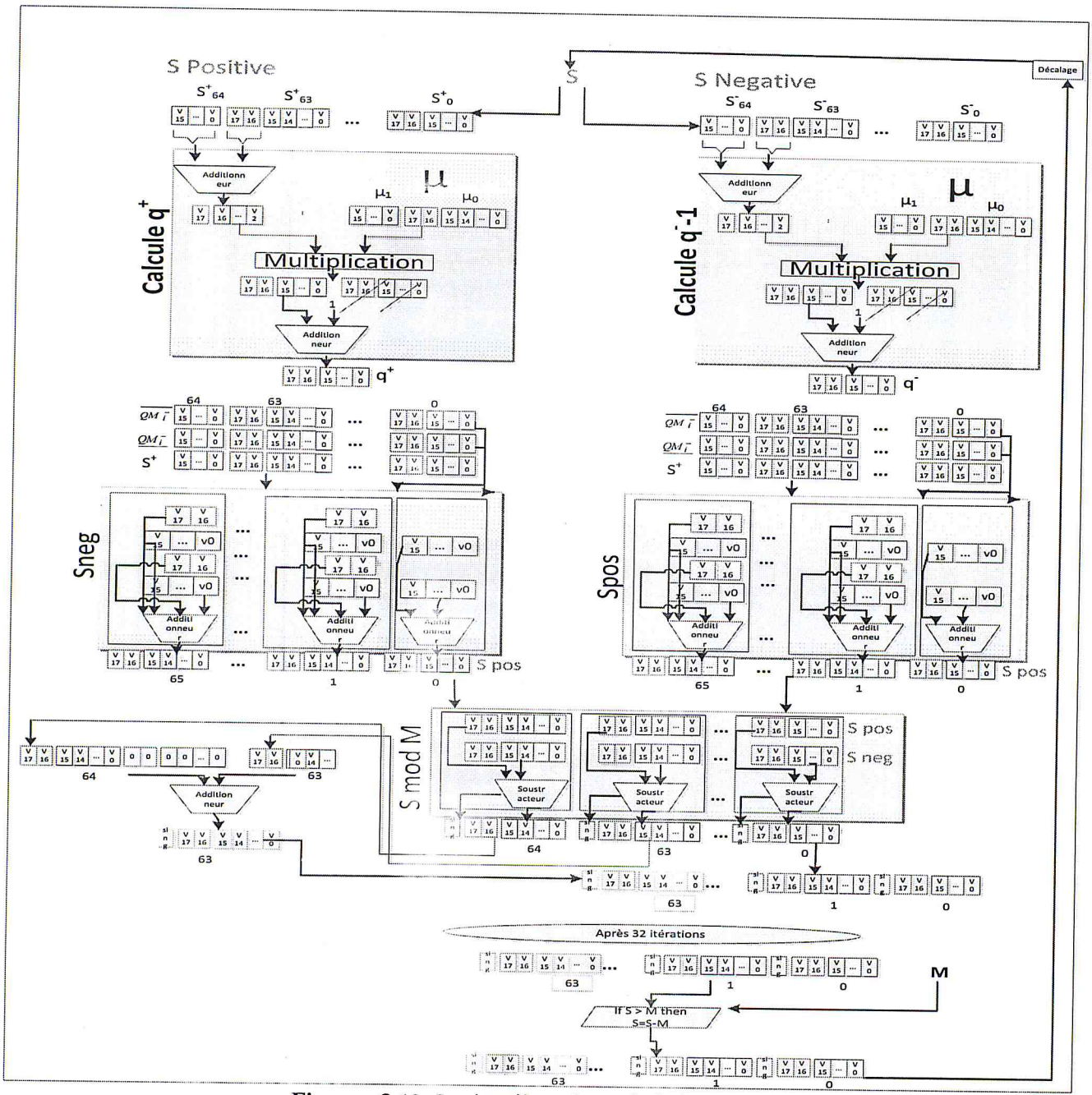


Figure : 3.12- La deuxième étape de l'algorithme

$MMBSNR\_2^{16}$

Les partie opérative de la multiplication 1 et 2 sont les même que dans la partie une de l'algorithme de Barrett.

La dernière partie : consiste à calculer le résultat final qui représente :

$$\text{Resultat}_{\text{Finale}} = \left(\sum_{j=0}^{j=d-1} \left((Spos1_j + CSpos1_j \times 2^k) + (SM1_j + CSM1_j \times 2^k) \right.\right.$$
$$\left. + (P1_j + CP1_j \times 2^k) + (P2_j + CP2_j \times 2^k) \right)$$
$$\left. - (Sneg1_j + CSneg1_j \times 2^k) \right) mod\ M$$

Cette opération est exécutée en deux étapes :

La première calcule la somme de tous les mots de poids fort en parallèle par l'équation suivant :

$$Som = (Spos1_j + CSpos1_{j-1} + SM1_j + CSM1_{j-1} + P1_j + CP1_{j-1} + P2_j + CP2_{j-1})$$

Elle utilise l'additionneur $(2, 2, 2, 2, 16, 16, 16, 16)$.

La deuxième calcule la soustraction entre le résultat précédent et le $Sneg$ par l'équation suivant :

$$\text{Resultat}_{\text{finale j}} = Som_{\text{j}} - Sneg_{\text{j}}$$

Ce résultat peut être supérieur à $5*M$ pour cela il est nécessaire de faire 5 itérations où chaque itération compare ce résultat avec M, si elle est supérieure à M on fait la soustraction avec M, sinon l'opération est terminée et le résultat est le résultat final.

III.6. Conclusion

Dans ce chapitre nous avons présenté notre architecture pour la multiplication modulaire, qui est basée sur deux algorithmes les plus adaptés à la cryptographie RSA, cette architecture a introduit une façon de parallélisme à l'intérieur des opérations de base comme l'addition et la multiplication. Dans le prochaine chapitre nous allons essayer de valider cette architecture en utilisant des outils spécifiques et le validé en implémentation matérielle.

Chapitre IV :

Résultats de simulation et d'implémentation

IV.1. Introduction

Après avoir présenté dans le chapitre précédent les concepts fondamentaux de notre proposition, nous arrivons dans ce chapitre à présenter les détails des résultats de simulation et d'implémentation de notre architecture sur le circuit FPGA de la famille Virtex-5 qui est le Xc5vlx50t-3ff665.

Pour vérifier le bon fonctionnement de notre algorithme matériel, nous avons utilisés l'outil de validation formelle ***Maple [Nic, 09].***

La description de tous les blocs composant notre architecture a été réalisé par le langage VHDL **[wil,11]** qui est intégré dans l'environnement ***ISE* 9.1*i*** (*Integrated Software Environnement*) de ***XILINX*** pour la conception, la simulation et l'implémentation de notre architecture.

Nous commençons dans un premier temps par la présentation de notre environnement de travail, ensuite nous passons aux détails de la démarche d'implantation et les résultats de simulation pour valider noter approche.

IV.2 Ethodologie de conception

La méthodologie de conception est basée sur le langage de description matériel VHDL, l'outil ISE Simulator (VHDL/ Verilog) a été utilisé pour la simulation et la synthèse au niveau RTL a été faite par XST (VHDL / Verilog).

Cette méthodologie est utilisée pour tester le bon fonctionnement de notre architecture et pour extraire les résultats de l'implémentation matérielle sur circuit FPGA de la famille Virtex-5, lexc5vlx330t-2ff1738.

Elle doit répondre à la fois aux objectifs de la description architecturale utilisée comme spécification et aux contraintes de réalisation (les performances).

IV.2.1. Description de l'ISE «Integrated Software Environnement »

C'est le logiciel de programmation des produits Xilinx (CPLD, FPGA Spartan et Virtex…). Cet outil permet de créer des projets comportant plusieurs types de fichiers (HDL, schématique, UCF, EDIF, etc.), de compiler, de créer des contraintes d'implémentation avec des contrainte de timings sur les horloges, de déterminer l'emplacement des broches et de créer des fichiers d'essai de simulation (Test Bench).

Le Navigateur de projet ISE offre un environnement de conception regroupe tous les outils nécessaires à la conception, la simulation et à l'implémentation d'un projet, Il comporte :

- ✓ Un éditeur de textes, de schémas et diagrammes d'états.
- ✓ D'un compilateur VHDL et Verilog.
- ✓ D'un simulateur.
- ✓ D'outils pour la gestion des contraintes temporelles
- ✓ D'outils pour la synthèse.
- ✓ D'outils pour la vérification.
- ✓ D'outils pour l'implémentation sur FPGA et CPLD.

Les étapes pour l'implémentation d'une spécification HDL sur un FPGA sont illustrées sur la figure 4.1.

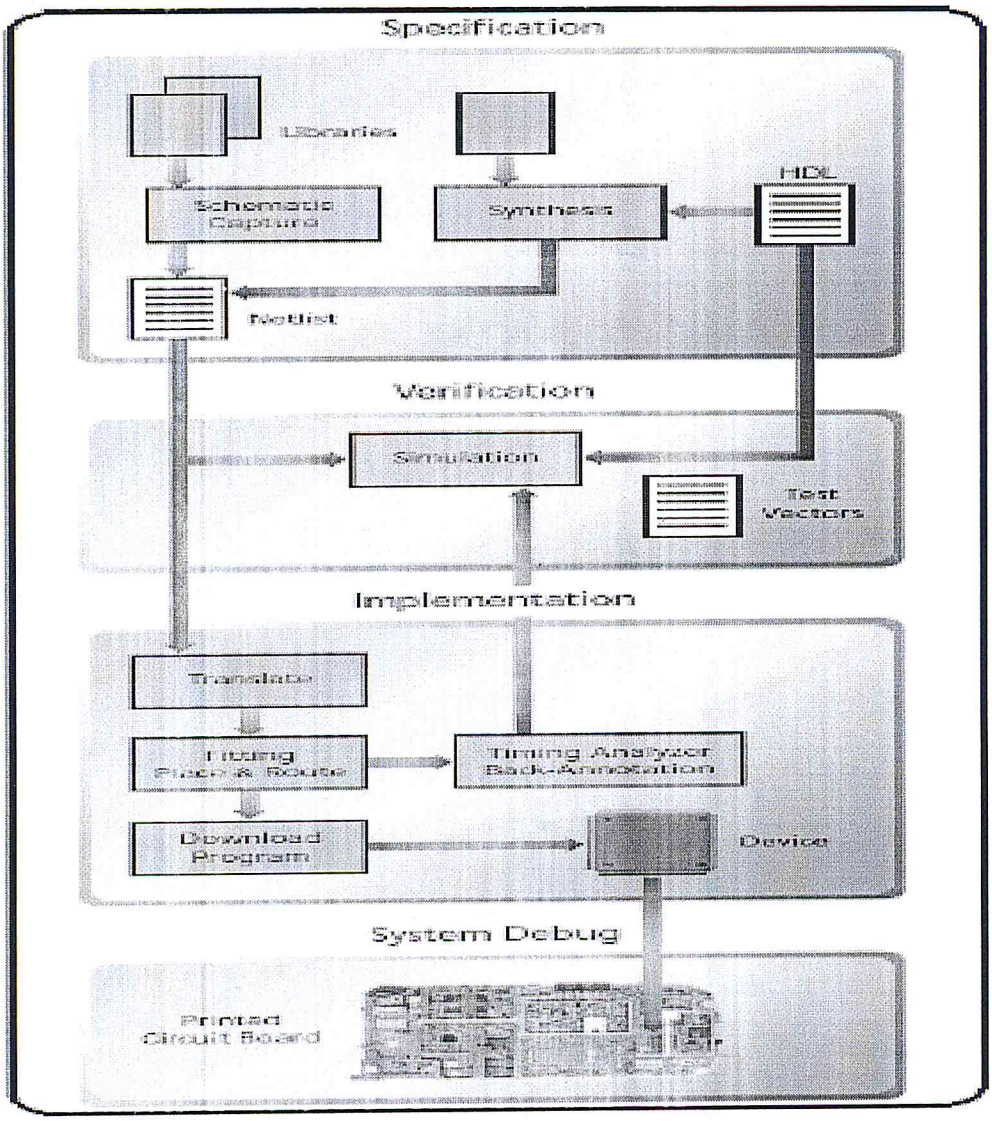


Figure 4.1. Les étapes d'implémentation d'un circuit sur un circuit logique programmable Xilinx.

IV.2.1.1.Spécification

La spécification HDL regroupe les trois modes de création d'un circuit (schématic, diagrammes d'états ou HDL). Elle est synthétisée pour générer un fichier appelé NETLIST qui décrit les interconnexions entre les registres.

IV.2.1.2. Vérification :

La vérification du design est une étape parallèle où le concepteur observe le comportement du code et s'il se comporte tel qu'il est supposé. Un simulateur simule le circuit par l'utilisation des vecteurs de test. Les vecteurs de test peuvent se présenter sous plusieurs formes, la plus courante est les **TESTBENCHS** écrits dans un langage de description matériel comme le VHDL pour entrer les instructions au simulateur. En appliquant les vecteurs de test sur le code pour que le simulateur fournit les sorties du circuit.

IV.2.1.3.Implémentation

Une fois la vérification est terminée, le circuit est implémenté sur le composant en spécifiant les références exactes de celui-ci à savoir : la carte utilisée, la fréquence de travail et les autres options spécifiques à chaque composant. Cette étape se termine par un rapport de tous les sous-programmes exécutés(les erreurs, les I/O utilisées et des données qui permettent de savoir si le composant choisi est le mieux adapté pour l'application ciblée).

IV.2.2. Langage de description VHDL

Le VHDL (**V**ery High Speed Integrated Circuits **H**ardware **D**escription **L**anguage) est un langage de description matériel HDL portable et synthétisable.

IV.2.2.1. Structure d'un programme VHDL

a) Entity: La partie déclarative de l'entité d'un circuit est décrite à travers les entrées et le sorties.

b) Architecture : l'architecture décrit le comportement que doit effectuer le circuit. Une architecture se doit toujours d'être attachée à une entité. C'est dans cette section que le programme est rédigé. Un programme comporte essentiellement les éléments suivants :
(Les signaux internes, opérateurs logiques (synchrone ou les process).

IV.3. Implémentation sur circuit FPGA

Dans cette étape, il est question de concevoir la partie matérielle de l'IP AES de notre système.

Notre implémentation de cet *IP* été basée sur les composants qui permettent d'utiliser et de fournir un ensemble de modules. Sachant que, nous avons implémenté trois architectures pour le chiffrement AES. Ces architectures sont : Exécution série-série, Exécution parallèle-série et Exécution parallèle-pipeline, qui se diffèrent entre eux dans la taille du chemin de données et la manière d'exécution série ou parallèle.

IV.3.1. Résultats de simulation

Les différents blocs fonctionnels de l'architecture proposée ont été testés et vérifiés par simulation avant de les implémenter dans un circuit FPGA. La simulation consiste à envoyer, via un fichier Test-Bench décrit en VHDL, des stimuli aux entrées du système et à observer le comportement de ses sorties. L'outil de simulation utilisé est ISE Simulator permettant de visualiser la variation des signaux de sortie et par la suite effectuer des modifications au niveau de la description en cas de résultats insatisfaisants.

IV.3.1.1. Résultats de simulation des modules de l'architecture

pour tester notre architecture, nous avons choisie des valeurs pour les opérandes X,Y et le modulo M ces valeurs sont choisies par l'outil *Maple* dans la représentation binaire.

IV.3.1.1.1. Résultats de simulation du module Montgomery

Les résultats de simulation du module Montgomery pour une itération est illustré dans la figure 4.2

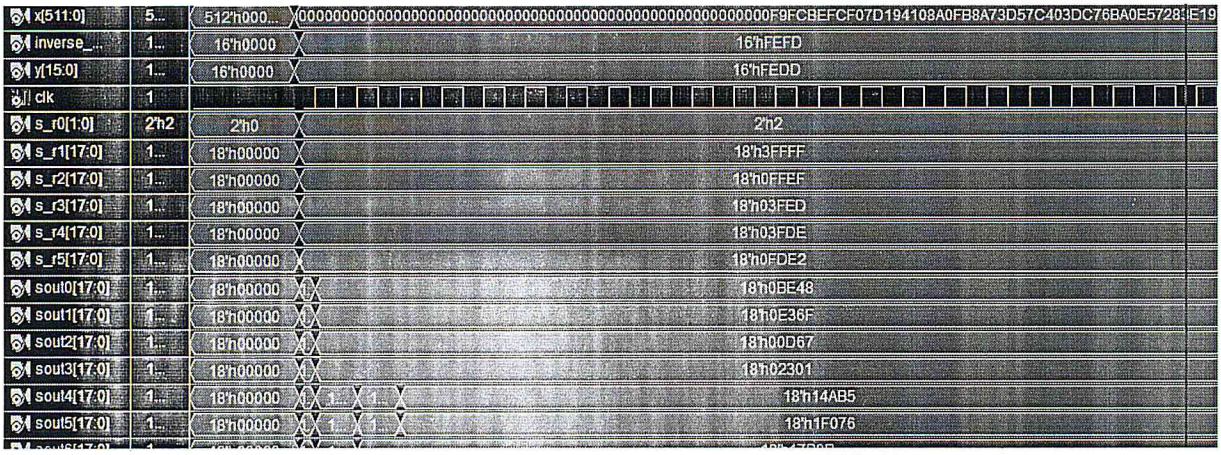


Figure4.2. Une partie de la simulation d'une itération de Montgomery.

Cette figure représente la simulation de module de Montgomery pour une itération, nous prenons X de 0 à 511

Les résultats obtenues sont représentés en 65 blocs de 18 bit comporte les bits principaux et les bits de retenue, dans cette figure nous avons montré une partie de résultats. Cette itération calcul S_{i+1} grâce à des modules intégrés.

Le composant MONTGOMERY est composé de plusieurs autres composants selon son architecture que nous avons développée dans le chapitre 3, parmi ces composants on trouve le multiplieur 18×18 ,nous avons utilisé le composant multiplieur 18×18 fournit par le Core-generator de *XILINX* qui est implémenté sur silicium.

Le but d'utiliser ce composant est en premier lieu pour gagner un espace d'implémentation car ce composant est implémenté sur silicium et en deuxième lieu la diminution du temps de conception, le routage des données à multiplié par les composants du circuit de Montgomery.

L'addition des résultats intermédiaires de la multiplication qui sont des blocs, nécessite la conservation des deux bits de retenu. Pour la phase où on calcule la multiplication de XY_i on nécessite des additionneurs de quatre entrées *(16, 16, 16, 2)*, donc nous avons réalisé ces additionneurs à partir de l'additionneur en parallèle *Carry Save Adder*

La figure suivante représente plus de détaille pour le circuit de Montgomery

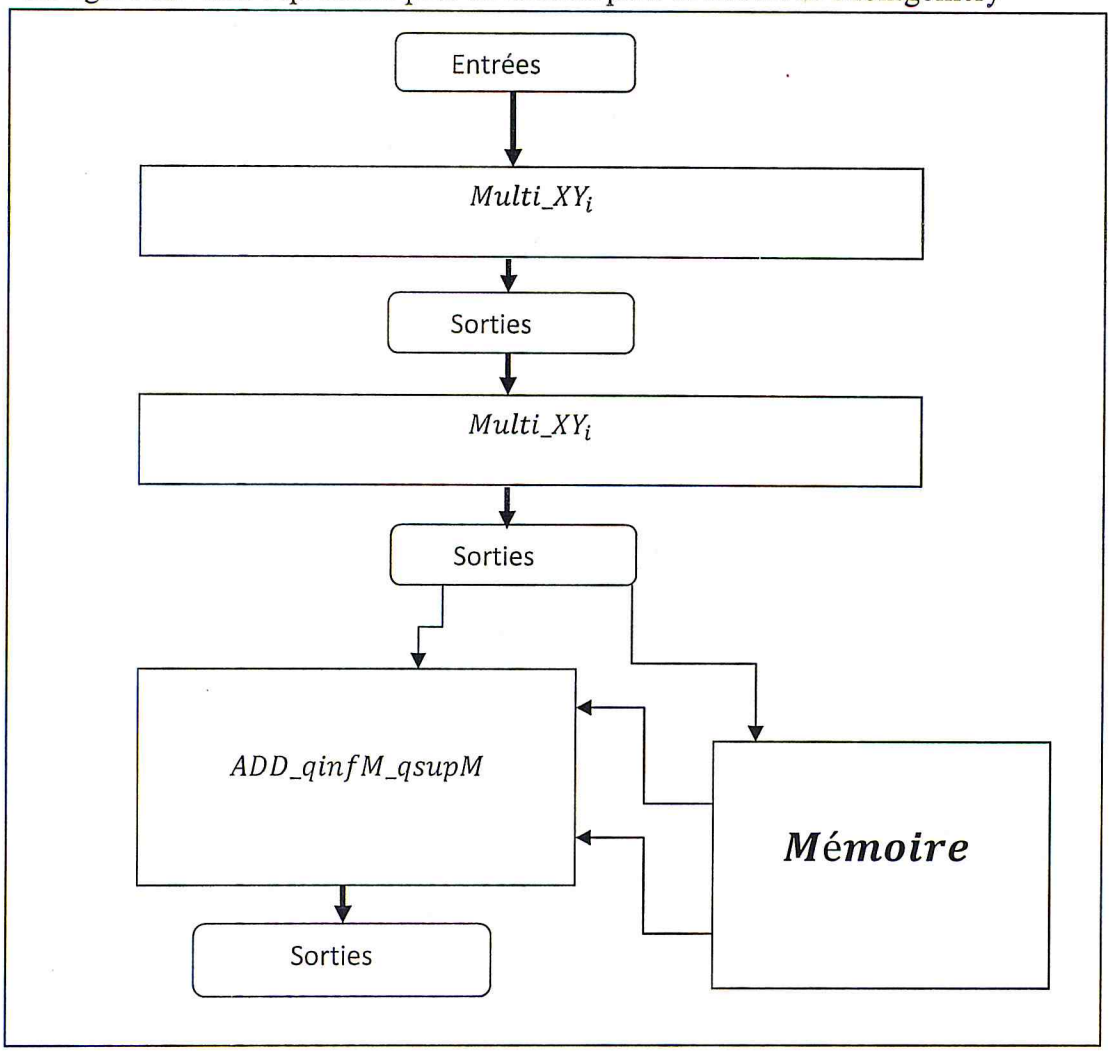


Figure 4.3 : schéma de bloc du circuit Montgomery.

IV.3.1.1.2. Résultats de simulation du module calcul_xy

La figure 4.3 représente la simulation de module calcul_xy pour une itération, ce module calcul les produits de X_0Y_1 et X_1Y_0.

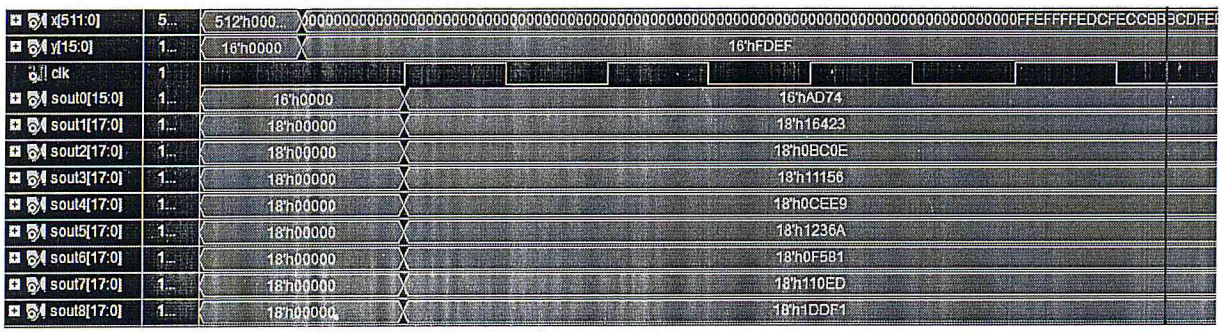


Figure 4.4. Une partie de la simulation de module calcul_xy.

Pour cette simulation nous avons pris le X de 0 à 511 et Y de 512 à 1023 pour tester le produit X_0Y_1 et X de 512 à 1023 et le Y de 0 à 511 pour simuler X_1Y_0.

Ce module donne les résultats dans la représentation redondante en 64 blocs de 18 bits chaque un.

Les résultats de simulation sont données pour les valeurs suivantes : X= » », Y= » », M= » »,

IV.3.1.1.3. Résultats de simulation du module Barrett

La figure 4.5 représente une partie de la simulation le module Barrett qui calcul une itération de Barrett.

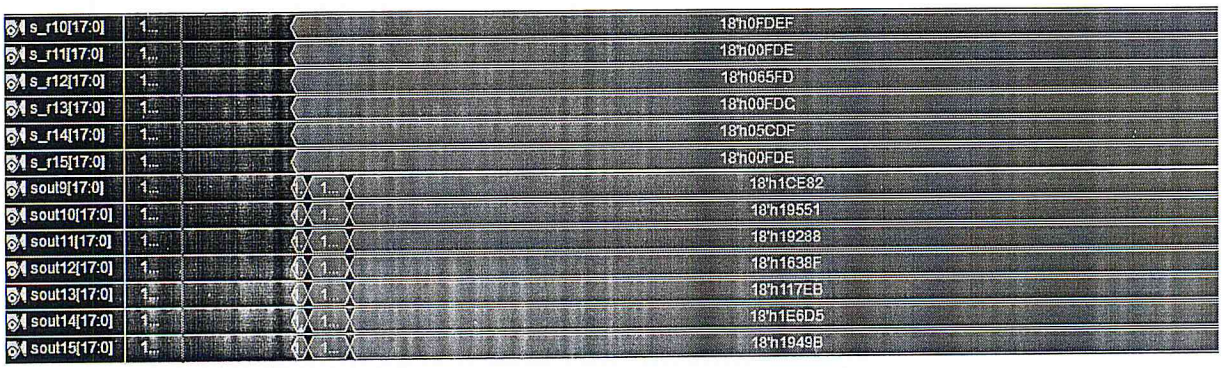


Figure 4.5. La simulation de module de Barrett pour une itération.

Pour la simulation nous avons choisie le X de 512 à 1023 et Y de 512 à 1023.

Après la compilation et la simulation du code et son Test-Bench, les valeurs obtenues de ces comme résultats sont apparues sur les signaux de sorties (*Sout)*

Le temps nécessaire pour une itération de Barrett est calculé comme suite :

On note que

$T_{itération}$: C'est le temps nécessaire pour une itération de l'algorithme qui calcule les quotients (q^+ *et* q^-) de Barrett.

T_B : C'est le temps nécessaire pour l'algorithme de Barrett.

T_{XYi} : C'est le temps nécessaire pour itération de l'algorithme qui calcule le produit XY

n : Le nombre d'itération.

$$T_B = (T_{XY_i} \times n) + (T_{itération} \times n)$$

$$T_B = (9.684 \times 32) + (4.156 \times 32) = 442.512ns.$$

La figure suivante représente un schéma de bloc pour ce circuit.

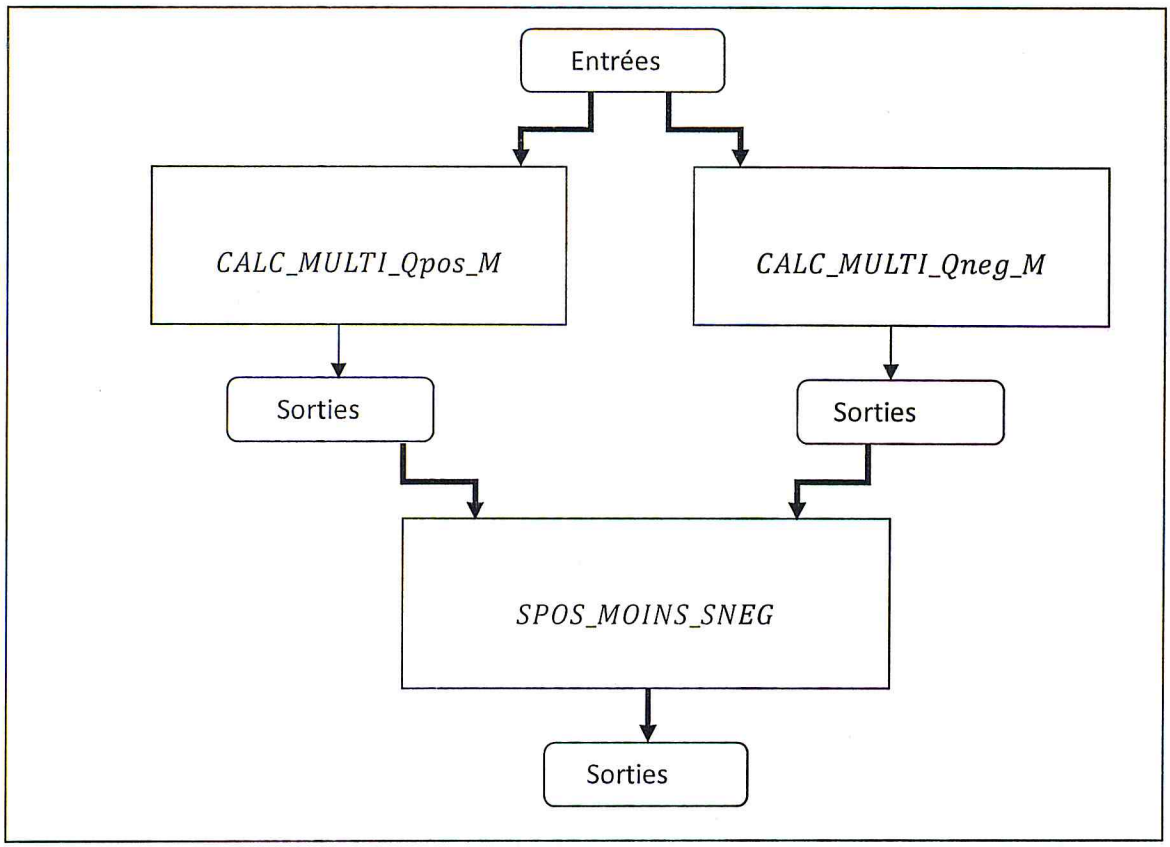


Figure 4.6 : Schéma de bloc pour le circuit Barrett.

IV.3.1.1.4. Résultats de simulation du module Quadripartite

Le module Quadripartite a pour but l'assemblage des résultats de Montgomery, Barrett et les deux produits de (X_0Y_1, X_1Y_0) , la figure 4.6 représente le schéma bloc de ce module.

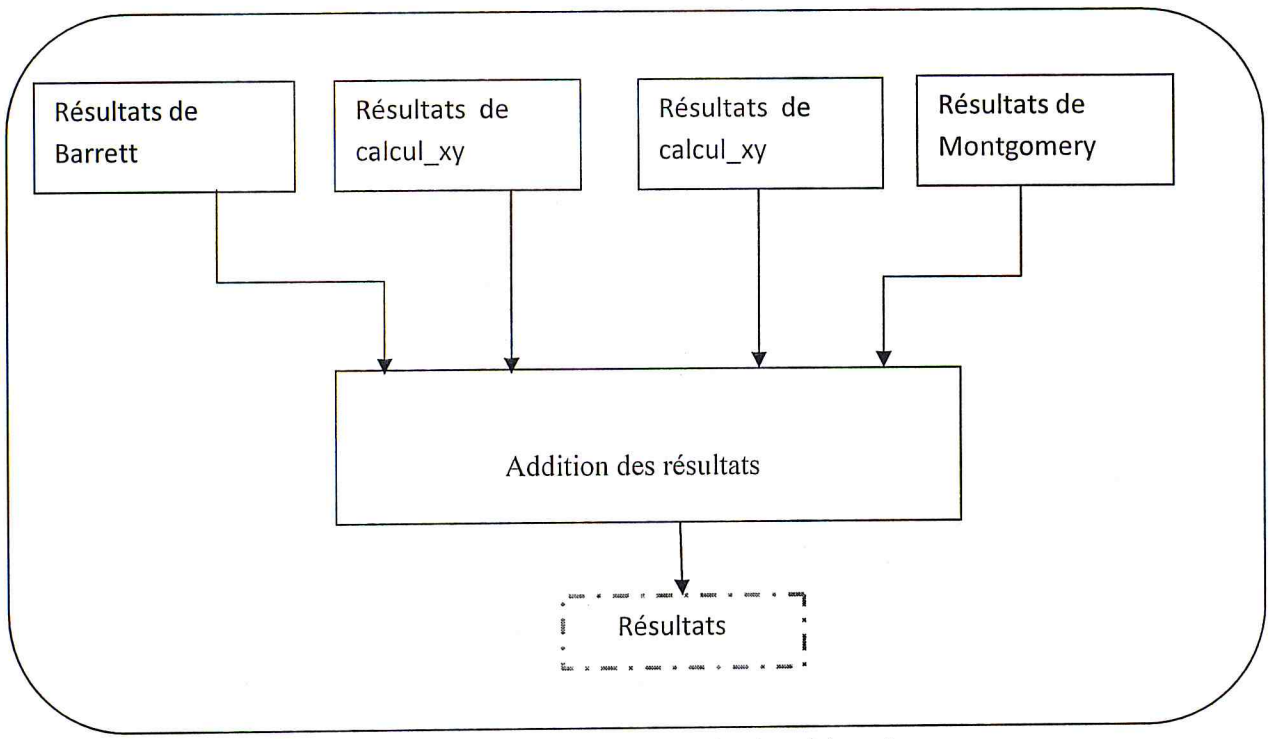


Figure4.7 : Le schéma bloc de module Quadripartite

Après avoir les résultats de simulation de Montgomery, Barrett et les deux produits pour la dernière itération, on les utilisé comme entrées à ce module.

IV.3.2. Résultats de synthèse et d'implémentation

Dans ce travail, la synthèse a été effectuée en utilisant l'outil XST (VHDL /Verilog) (Xilinx Simulator Tools VHDL/Verilog). La cible matérielle est le FPGA, Virtex-5Target device xc5vlx50t-3ff665 pour toutes nos implémentations.

Les résultats de l'implémentation des différents modules de cette architecture sont regroupés dans le tableau 4.1 suivant :

Ressources / Modules	Slice Registers	Slice LUTs	temps(ns)
Montgomery	/	13 (0%)	433.696
Calcul_xy	/	8 (0%)	9.673
Barrett	12%	27 (0%)	4.156
Quadripartite	/	40 (0%)	456,621

Tableau 4.1 : Résultats d'implémentation des modules de l'architecture proposée.

Nous avons intéressé au temps d'exécution, d'après les résultats montrés dans le tableau précédent, cette architecture donne un temps d'exécution plus optimisé et plus rapide, à cause de la représentation redondante qui provoque un parallélisme à l'intérieur des opérations de base comme l'addition et la multiplication.

IV.4. Conclusion

Notre proposition présente un temps de multiplication modulaire plus performant et plus rapide grâce à le parallélisme utilisé et la représentation redondante, et comme chaque méthode a des avantages et des inconvénients, notre proposition a comme inconvénient le doublement de matérielle.

Dans ce chapitre nous avons présenté comment nous avons implémenté notre architecteur qu'on a proposé dans le chapitre 03.

Pour ce la nous avons décrit chaque composant indépendamment pour être intégré dans un autre circuit pour arrivé an circuit final, nous avons pressant aussi les éléments de base qui on a implémenté comme additionneur, soustracteur, multiplieur.

La simulation de ces composants est aussi présentée avec les résultats de synthèse générés par *ISE*.

Conclusion générale

Conclusion et perspectives

Le but dans ce projet est le développement d'un opérateur arithmétique performant pour la cryptographie asymétrique RSA.

Pour ce faire nous avons fait en premier temps, une étude sur le protocole de chiffrement RSA que nous avons présenté dans le chapitre 1. Le RSA est basé sur l'exponentiation modulaire qui est une suite de multiplications modulaires, pour se faire nous avons présenté dans le chapitre 2 une étude détaillée sur les différentes méthodes matérielles pour le calcul de la multiplication modulaire pour en choisir les méthodes de Montgomery et Barrett. Ces derniers ont été choisis pour leurs avantages et adaptabilités aux implémentations matérielles. Il est à noter que les traitements se font du poids fort au poids faible pour l'algorithme de Barrett et inversement pour celui de Montgomery, ce qui offre un grand avantage dans la parallélisation du calcul de la multiplication modulaire. Pour ce faire nous avons présenté la méthode Bipartite et sa généralisation la Multipartite.

Dans un premier temps on s'est penché sur l'augmentation des performances des deux algorithmes Barrett et Montgomery par l'utilisation de ces algorithmes dans une grande base de numération. L'augmentation de la base diminue certes le nombre d'itérations, néanmoins celle-ci augmente la complexité calculatoire dans l'itération. L'introduction de la représentation redondante SNR conjugué à la disponibilité de blocs multiplieurs 18×18 bits dans les circuits FPGA de Xilinx, nous ont permis de contourner cette complexité calculatoire de la ramener même au niveau de la complexité de la multiplication modulaire en base 2. Dans ce travail notre choix a été porté sur la base 2^{16} qui s'adapte bien aux ressources disponibles dans les circuits FPGA de la famille Virtex5.

D'avantages de performances ont été obtenues par l'utilisation de la méthode quadripartite qui permet de calculer une multiplication modulaire n×n bits en deux multiplications modulaires de n/2×n/2 bits. En plus de permettre la parallélisation de ces deux opérations.

Dans le chapitre 3, les détails de nos architectures ont été exposés. Alors que dans le chapitre 4, nous avons présenté les étapes de conception, les résultats de simulation et d'implémentation de notre architecture. Notre architecture a fait objet de plusieurs simulations pour valider son fonctionnement. Cette vérification s'est faite d'une manière hiérarchique du

composant de base jusqu'au l'architecture globale. Celle-ci a été faite pour l'additionneur, le multiplieur et le soustracteur pour arriver en fin au circuit final de la multiplication modulaire. Dans ce chapitre nous avons aussi présenté les résultats de synthèse et d'implémentation où nous avons défini le chemin critique de notre architecture .

Comme perspectives, l'utilisation de la multipartite permettrait certes de diminué d'avantage la taille du datapath et par conséquent d'augmenter le parallélisme dans le calcul de la multiplication modulaire, néanmoins celle-ci souffre du dédoublement du matériel et de la complexité du routage. Cette piste reste à explorer et la diminution de cette complexité reste un challenge pour les améliorations futures des performances de cette opération.

Références Bibliographique

- **[Bar, 86]:** P. Barrett. « *Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor* ». In Advances in Cryptology – CRYPTO'86, volume 263/1987 of Lecture Notes in Computer Science, pages 311–323. Springer-Verlag (1986).
- **[Ber, 07]:** F. Bernadr. « *Étude des algorithmes arithmétiques et leur implémentation matérielle*». Thèse de doctorat : Informatique. L'université de Saint Denis Paris 8. Paris. 155p. (2007).
- **[Bla, 83]:** G.R. **B**lakely. « *A computer algorithm for calculating the product AB modulo M* ». **C**omputers, IEEE Transactions on, 32(5). 497–500p. (1983).
- **[Ccd, 08] :** V. COLIN, P. COLLIN et A. DUMAINE « Travaux Personnels Encadrés: *Rupture et continuité :* L'évolution de la cryptographie a-t-elle permise l'émergence de techniques inviolables? ». 1èreS4 - Lycée David d'Angers, Travaux .p20.(2008)
- **[Des, 09]:** L. DESTREE, M. MARCHAL , «*Mini-RSA :Programme d'initiation au chiffrement RSA* », – P2 gr B – Projet MPI n°1,p20. (2006)
- **[Gio, 13]:** P. Giorgi, L. Imbert and T. Izard . «*Multipartite modular multiplication*». IEEE Symposium on Computer Arithmetic, Austin, Texas : États-Unis (2013). hal-00805242, version 1. (2013)
- **[Iza, 11]:** T. Izard. « *Opérateurs arithmétiques parallèles pour la cryptographie asymétrique* » Thèse de doctorat : Informatique. Université de Montpellier 2. Montpellier. 132 p. (2011).
- **[Kai, 08]:** Kaihara, M. E. and Takagi, N. « *Bipartire modular multiplication method*», IEEE Transactions On Computers, VOL. 57, NO. 2, FEBRUARY (2008).
- **[Ken, 08]:** K. Kawakami, K. Nakano and K. Shigemoto. « *Redundant Radix-2^r Number System for Accelerating Arithmetic Operations on the FPGAs*». Hiroshima University: Information Engineering, JAPAN. Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies. (2008).
- [**Man, 08**]: K. Manochehri and S. Pourmozafari. « *Fast Montgomery Modular Multiplication by Pipelined CSA Architectwet* ». University of TechnoZogy, Tehran, Iran. (2008)
- **[Mir, 09]** : K. Miroslav, et V. Ingrid.« *Speeding Up Barrett and Montgomery Modular Multiplications*».(2009).

Références Bibliographique

- **[Mon, 85]:** P. Montgomery, Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, (1985).

- **[Nak, 09]:** K. Nakano, K. Kawakami, and K. Shigemoto. *« RSA Encryption and Decryption using the Redundant Number System on the FPGA »*. Hiroshima University: Information Engineering, JAPAN. (2009).
- **[Nit, 09] :** A. Nitaj. *« Cryptanalyse de RSA»*. . Laboratoire de Mathématiques Nicolas Oresme, Université de Caen, France, Version du 28 juin 2009.
- **[Pla, 05]:** T. Plantard. *« Arithmétique modulaire pour la cryptographie »*. Thèse de doctorat : Informatique. Université de Montpellier 2. Montpellier. 132 p. (2005).
- **[Riv, 78] :** R. Rivest, A. Shamir, L. Adleman « A method for obtaining digital signatures and public-key cryptosystems», Communications of the ACM, Vol. 21 (2), 120|126. (1978).
- **[Tis, 10]** : A. Tisserand. *« Étude et conception d'opérateurs arithmétiques »*. Thèse de doctorat : Informatique. Université de Rennes 1 .Rennes. 163p. (2010).
- **[Pla, 05]:** T. Plantard. *« Arithmétique modulaire pour la cryptographie »*. Thèse de doctorat : Informatique. Université de Montpellier 2. Montpellier. 132 p. (2005).
- **[Wlc, 02]:** J-F. Wang, P-C. Lin and P-K. Chiu. *«A Staged Carry-Save-Adder Array forMontgomery Modular Multiplication»*, Department Of Electrical Engineering, National Cheng Kung University.(2002).

[Toy]

Annexe : de Maple.

Etude comparative