MA-004-106-1

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Ministère de L'Enseignement Supérieur et de la Recherche Scientifique

UNIVERSITE SAAD DAHLAB BLIDA

Département INFORMATIQUE

Mémoire de Projet de Fin d'Etudes

de Master en Informatique

Option : Ingénierie Logiciel

Thème :

Contribution à la conception d'un crypto système embarqué sur FPGA

Proposé par :

Mr. Mohamed ANANE

Présenté par :

Mr. Bellemou Mohamed Ahmed

Mr. Zebouchi Mohammed

Encadré par :

Mr. Mohamed ANANE

Mr. Mohamed ISSAD

MA-004-106-1

Soutenu le : 02.07.2012

Devant le jury compose de :

President: Mme.Bensettiti S

Examinateur: Mr.Cherif zahar

Promoteur: Mr.Mohamed ANANE

Promotion : 2011 - 2012

Remerciements

Au terme de ce travail, nous tenons tout d'abord à rendre grace à ALLAH, le Clément et le Miséricordieux, qui nous a permis d'aboutir à cette fin tant souhaitée, en nous insuflant la patience et le courage nécessaires pour surmonter les difficultés que nous avons rencontrées au cours de notre cursus.

Nos remerciements à Mr B.BOUZOUIA, directeur du Centre de Développement des Technologiques avancées (CTDA)

Nous tenons aussi à exprimer nos remerciements les plus chaleureux à nos promoteurs, Mr. Mohamed ANANE et Mr.Mohamed ISSAD, pour leur précieuse aide, pour leur suivi rigoureux et leur disponibilté pendant notre travail.

Nous adressons aussi nos remerciements à Mme. Anane pour son aide.

Que les membres du jury trouvent ici le témoignage de notre reconnaissance pour avoir bien accepté d'évaluer notre travail.

Nous remercions enfin toute personne qui nous a aidé de près ou de loin à l'élaboration de ce travail.

A ma très chère Maman qui n'a cessé de m'entourer de sa tendresse et de me prodiguer ses encouragements

A mon brave Père à qui je dois sincèrement beaucoup de choses dans ce travail.et qui ne cesse, pour sa part, de m'encourager à me surpasser et de faire toujours mieux.

A mes frères Omar et A.Wadoud.

A mes sœurs Khadidja , Amina et Widad.

A tous ceux qui m'ont encouragé pour l'accomplissement de ce projet.

A mon binôme Mohamed pour son apport dans ce travail.

A tous mes amis et toute la promotion Master 2012,

Je dédie ce modeste travail.

Anouar.

Table de matières

Introduction générale.. 1

Chapitre 1 : Généralités sur la cryptographie

1.1 Introduction.. 4

1.2 Définition de la cryptographie.. 4

1.3 Types d'algorithmes de cryptographie.. 6

1.3.1 Cryptographie symétrique.. 6

1.3.2 Cryptographie asymétrique.. 7

1.4 Le crypto système RSA 9

1.4.1 Génération des clés 9

1.4.2 Chiffrement 9

1.4.3 Déchiffrement.. 10

1.5 Sécurité du RSA 10

1.6 Mise en œuvre du RSA 11

1.7 Conclusion.. 12

Chapitre 2 : Exponentiation modulaire de Montgomery

2.1 Introduction.. 13

2.2 L'exponentiation modulaire.. 13

2.3 L'exponentiation modulaire binaire.. 14

2.3.1 Exponentiation modulaire binaire MSB (Most significant bit).. 15

2.3.2 Exponentiation modulaire binaire LSB (Least Significant Bit).. 16

2.3.3 Comparaison des méthodes MSB et LSB.. 17

2.4 La multiplication modulaire.. 18

2.4.1. Définition.. 18

2.4.2 Description de l'algorithme Montgomery (A, B, N).. 18

2.4.3 Calcul de la multiplication Montgomery.. 19

2.4.3.1 Première méthode 20

2.4.3.2 Deuxième méthode.. 21

2.5 Variantes de la multiplication modulaire de Montgomery.. 23

2.5.1 La multiplication modulaire de Montgomery sans soustraction finale.. 24

2.5.2 Algorithme de Montgomery entrelacé en base 2 versions série...................... 25
2.5.3 Algorithme de Montgomery entrelacé en base 2^k version série........................ 28
2.5.4 Complexité calculatoire des algorithmes en bases 2 et 2^k.................................. 30
2.6 Exponentiation modulaire binaire LSB de Montgomery... 30
2.6.1 Fonctionnement de l'algorithme d'exponentiation binaire de montgomery......... 31
2.6.2 Complexité de l'algorithme de l'exponentiation modulaire.......................... 32
2.7 Conclusion... 33

Chapitre 3 : Systèmes embarqués

3.1 Introduction... 34
3.2 Système Embarqué.. 34
3.3 Système embarqué sur puce SoC.. 35
3.3.1 Architecture d'un système sur puce.. 36
3.4 Système embarqué sur les circuits FPGA SoPC... 37
3.4.1 Processeurs embarquées sur FPGAs.. 38
3.4.2 Système embarqué à base du processeur Microblaze................................... 38
3.4.2.1 Architecture du processeur Microblaze.. 40
3.4.2.2 On-Chip Peripheral Bus (OPB)... 41
3.4.2.3 Local Memory Bus (LMB)... 41
3.4.2.4 Fast Simplex Link (FSL)... 41
3.4.2.5 Xilinx Cache Link (XCL).. 42
3.4.2.6 La mémoire BRAM.. 42
3.4.2.7 Les Périphériques du processeur Microblaze... 42
3.5 Les critères de performance des SoPC... 43
3.5.1 Les performances temporelles... 43
3.5.2 La consommation... 44
3.5.3 La flexibilité.. 44
3.5.4 Les coûts.. 44
3.6 Flot de Conception d'un SoPC.. 45
3.7 Modèles d'implémentations des SoPC.. 46
3.7.1 Implémentation logicielle (SW).. 46
3.7.2 Implémentation matérielle (HW)... 47
3.7.3 Implémentation Co-conception (SW/HW)... 47

3.7.4 Conception du système et cycle de développement........ 48
3.7.4.1. Configuration de Microblaze et des périphériques........ 49
3.7.4.2. L'ajout d'un IP personnalisé........ 50
3.7.4.3Développement de la partie logicielle et chargement du système sur FPGA. 52
3.7.4.3.1 Interface de programmation de l'application........ 52
3.7.4.3.2 Compilation de l'API........ 52
3.7.5 Les systèmes d'exploitation compatibles avec Microblaze........ 53
3.7.5.1 uClinux........ 53
3.7.5.2 Asterix........ 54
3.7.5.3 Xilkernel........ 54
3.7.5.4 Standalone........ 54
3.8 Conclusion........ 54

Chapitre 4: Implémentation de la plateforme de chiffrement/déchiffrement

4.1 Introduction........ 55
4.2 Description de la Plateforme de chiffrement/déchiffrement........ 56
4.3 Implémentation de l'exponentiation modulaire sur FPGA........ 57
4.3.1 Description matérielle........ 58
4.3.2 Description logicielle........ 59
4.4 Approche logicielle........ 59
4.4.1 Implémentation de l'exponentiation modulaire........ 59
4.4.2 Implémentation de la MMM en base 2........ 60
4.4.3 Implémentation de la MMM en base 2^k........ 64
4.5 Approche matérielle........ 68
4.6 Application JAVA........ 72
4.6.1 Class BigIntegerPro........ 73
4.6.2 Class CommunicationRS232........ 75
4.6.3 Class Main........ 75
4.7 Conclusion........ 76

Chapitre 5 : Résultats d'implémentation

5.1 Introduction........ 77
5.2 Conception et Implémentation sur circuit FPGA........ 77

5.2.1 Description de l'architecture matérielle 77
5.2.2 Synthèse et implémentation sur circuit FPGA 78
5.2.3 Génération et chargement du BitStream sur le circuit FPGA 79
5.2.4 Description de la partie logicielle 79
5.2.5Chargement de la plateforme logicielle sur le circuit FPGA 81
5.3 Développement de l'application Java 81
5.3.1 Configuration du protocole de communication RS232 82
5.3.2 Génération des clés 82
5.3.3 Chiffrement et déchiffrement d'un message 83
5.3.4 Chiffrement et déchiffrement d'un fichier 84
5.4 Vérification des résultats 85
5.5 Performances temporelles et ressources occupées sur circuit FPGA 85
5.5.1 Performances temporelles 85
5.5.2 Les ressources occupées 87
5.6. Conclusion 89

Conclusion générale 91
Annexe A
Annexe B
Bibliographie

Liste des algorithmes :

Algorithme 2.1- Méthode d'exponentiation modulaire simple (M, e, N)........................ 14

Algorithme 2.2- Exponentiation modulaire binaire MSB.. 15

Algorithme 2.3- Exponentiation modulaire binaire LSB.. 16

Algorithme 2.4- Algorithme de Montgomery.. 19

Algorithme 2.5- Méthode 1 de calcul de la multiplication modulaire............................ 20

Algorithme 2.6- Méthode 2 de calcul de la multiplication modulaire............................ 21

Algorithme 2.7- Multiplication modulaire de Montgomery entrelacée sans soustraction Finale 24

Algorithme 2.8- algorithme de Montgomery en base 2.. 26

Algorithme 2.9- Algorithme de Montgomery en base 2^k... 29

Algorithme 2.10- Algorithme binaire LSB.. 31

Liste des tableaux :

Tableau 2.1- Déroulement de la méthode MSB pour e=55.. 16

Tableau 2.2- Déroulement de la méthode LSB pour e=55.. 17

Tableau 2.3- Etape "a" de la première méthode.. 21

Tableau 2.4- Etape "b" de la première méthode.. 21

Tableau 2.5- Etape "a" de la Méthode 2.. 22

Tableau 2.6- Etape "b" de la Méthode 2.. 23

Tableau 2.7- Etape "c" de la Méthode 2.. 23

Tableau 2.8- Etape "d" de la Méthode 2.. 23

Tableau 3.1- Fréquence du processeur Microblaze sur quelques circuits......................... 41

Tableau 4.1 : Fonctions en C des options de l'IPIF.. 69

Tableau 4.2- Les codes des signaux de contrôle.. 71

Tableau 4.3 : Classe BigIntegerPro et Classe CommunicationRS232............................... 73

Tableau.5.1- Approches et types d'implémentations.. 79

Tableau.5.2- Les fonctions des composants des deux types d'implémentation.................... 80

Tableau.5.3- Pilotes du Timer.. 81

Tableau.5.4- Performances temporelle du chiffrement avec une clé de taille 32 bits........... 86

Tableau.5.5- Performances temporelles de déchiffrement avec une clé de taille 1024 bits.... 87

Tableau 5.6- Ressources occupées sur le circuit FGPA... 88

Figure 1.1- Chiffrement symétrique.. 3
Figure 1.2- Chiffrement asymétrique.. 4
Figure 1.3- Signature électronique.. 5
Figure 1.4: Schéma bloc du principe de fonctionnement du RSA.................................. 7
Figure 2.1- Etapes d'exécution de l'algorithme 2.5.. 20
Figure 2.2- Etapes d'exécution de l'algorithme 2.6.. 22
Figure 2.3- La représentation des données.. 25
Figure 3. 1- Divergence entre la productivité de la conception et la complexité des Circuits..... 34
Figure 3.2- Plateforme d'implémentation.. 36
Figure 3.3- Architecture d'un système sur puce... 36
Figure 3.4- Carte de prototypage V2MB1000 et schéma synoptique d'un SoPC à base de Microblaze.. 39
Figure 3.5- Architecture de Microblaze... 40
Figure 3.6- Flot de conception d'un Système sur Puce. 45
Figure 3.7- Exécution séquentielle... 47
Figure 3.8. Exécution parallèle.. 47
Figure 3.9. Exemple d'implémentation en co-conception... 48
Figure.3.10- cycle de conception de Xilinx.. 49
Figure 3.11.Connexion d'un IP par l'utilisation d'une liaison FSL................................. 51
Figure 3.12- Ajout d'un IP personnalisé au bus système... 51
Figure 3.13- Compilation de la partie logicielle.. 52
Figure.4.1- Plateforme de chiffrement/déchiffrement.. 56
Figure 4.2- Architecture de l'approche logicielle.. 58
Figure 4.3- Architecture de l'approche matérielle... 58
Figure 4.4- Organigramme de la fonction expBinnary()... 60
Figure 4.5- Organigramme de la MMM en base 2.. 61
Figure 4.6- Organigramme de l'Implémentation de la MMM en base 2^k.......................... 65
Figure 4.7- Architecture générale pour le calcul de l'exponentiation modulaire par l'approche matérielle... 68
Figure 4.8- Organigramme du calcul de l'exponentiation modulaire............................... 71
Figure 4.9- Multiplication de deux grands nombres.. 73
Figure 4.10- Exemple de conversion d'un bigInteger en base 2^{16}................................ 74
Figure 4.11- Exemple de conversion d'un bigInteger en base 2^{32}................................ 74

Figure 4.12- Algorithme de la Classe Main.. 76
Figure 5.1. Menu principal de l'IHM... 82
Figure 5.2. configuration du port RS232.. 82
Figure 5.3. Génération des clés.. 83
Figure 5.4- Chiffrement et déchiffrement d'un message.. 84
Figure 5.5. Chiffrement et déchiffrement d'un fichier.. 84
Figure.5.6. Algorithme de vérification fonctionnelle de la plateforme.......................... 85
Figure.5.8. Implémentation de l'architecture Microblaze avec IP_MMM
sur le circuit XC2V1000fg465-4... 89

ملخص:

في هذه الأطروحة، نقوم بتصميم و تنفيذ نظام التشفير بالمفتاح العمومي RSA يستند على المعالج Microblaze، محمل علي دارة منطقية البرمجة (FPGA) ل Xilinx. مقاييس المفاتيح المستخدمة هي 1024 بت .

الأس و الضرب الترديدي هما العمليتان الرئيسيتان للتشفير و فك التشفير في نظام الRSA. تم استخدام خوارزميات الأس الثنائي SLB و مونتغومري لأنهما سريعة وبسيطة ا لتنفيذ. فعاليات خوارزمية مونتغومري (وقت التنفيذ، والمساحة المحتلة) تعتمد على أساس تمثيل البيانات.

قد تم تنفيذ نوعين من البرنامج: برنامج Software محض حيث يتم تنفيذ العمليات الحسابية من الأس والضرب الترديدي ل مونتغومري ب Microblaze و برنامج آخر يجمع بين Software عند حساب الأس ب MicroBlaze و hardware لنفيذ الضرب مونتغومري. نتائج تنفيذ النهج الثاني له أداء أفضل من حيث وقت التنفيذ.

Résumé

Dans ce travail, nous présentons la conception et l'implémentation du crypto système à clé publique le RSA embarqué sur circuit FPGA de Xilinx à base du processeur Microblaze. Les tailles des clés considérées sont de1024 bits.

L'exponentiation modulaire et la multiplication modulaire sont les opérations principales du chiffrement/déchiffrement RSA. Les algorithmes d'exponentiation binaire LSB et celui de Montgomery ont été utilisés car ils sont rapides et simples à implémenter. Les performances de l'algorithme de Montgomery (temps d'exécution et surface occupée) dépendent de la base de représentation des données.

Deux types d'implémentation ont été réalisés : une approche purement logicielle où les calculs de l'exponentiation modulaire et de la multiplication de Montgomery sont exécutées par le processeur Microblaze. Une autre approche combinant l'implémentation logicielle et matérielle où le calcul de l'exponentiation modulaire est effectué par Microblaze et la multiplication modulaire de Montgomery est implémentée sur matériel. Les résultats d'implémentation de la deuxième approche présente de meilleures performances en termes de temps d'exécution.

Abstract

In this work, we present the design and implementation of the RSA public key cryptosystem embedded on FPGA circuit of Xilinx and based on MicroBlaze processor. The considered key sizes are of 1024 bits.

The modular exponentiation and multiplication are the basic operations of RSA encryption/decryption. Binary exponentiation and Montgomery algorithms are used because they are fast and easy to implement. The performances of Montgomery algorithm (execution time and occupied area) depend on the radix representation of data.

Two types of implementation have been made: one software approach where computation of modular exponentiation and Montgomery multiplication are executed by the processor Microblaze. Another approach combines hardware and software implementation when the computing of the modular exponentiation is performed by MicroBlaze and the Montgomery modular multiplication is implemented on hardware. The implementation results of the second approach presents better performances in terms of execution time.

Introduction générale

Introduction générale

De nos jours, les efforts et les préoccupations pour la réalisation des systèmes électroniques se concentre sur leurs intégrations et leurs implémentations sur une puce de silicium. Par conséquent, les notions des systèmes embarqués se développent de plus en plus et ils sont définis en général comme étant des systèmes hétérogènes. En effet, ces derniers sont souvent constitués par un où plusieurs processeurs embarqués et un ensemble de périphériques qui fonctionnent au tour de ces processeurs.

Les systèmes embarqués sont dédiés généralement à une application bien précise qui peut être embarquée dans un circuit électronique à faible encombrement (faible poids). Leur technologie fait alors appel à une électronique et à des applications portables où l'on doit minimiser aussi bien l'encombrement que la consommation électrique. Le développement de ces systèmes ne consiste pas seulement en la conception de la partie matérielle, mais aussi la gestion software de ce matériel via le processeur embarqué, on parle ainsi de système SoC (System on Chip) qui consiste à cohabiter deux ressources logicielle et matérielle sur une même puce.

Dans ce genre de système, le terme qui revient souvent dans la littérature est le terme co-conception [1]. Cette approche à pour but de définir une partie logicielle qui est utilisée pour sa flexibilité et qui est recommandée pour le contrôle des applications. La partie matérielle est souvent préconisée à l'implémentation des parties sensibles au facteur temps réel. L'objectif de la co-conception est de réfléchir aux différentes actions que peut réaliser le matériel afin d'alléger le logiciel tout en tenant compte des contraintes de coût, des bénéfices en termes de performances, de surface occupée et de puissance consommée.

La technologie des systèmes embarqués sur des circuits de type ASICs est la plus performante sur tous les niveaux quant il s'agit d'applications à grand volume. Le seul obstacle devant les SoCs en technologie ASIC c'est le manque de souplesse, et l'impossibilité de faire des changements après la fabrication. Une erreur de conception ou un changement dans un modèle de communication standard nécessitera une nouvelle conception du système. La souplesse cherchée par les concepteurs des ASICs ne peut exister qu'en utilisant les circuits programmables.

Au cours de ces dernières années, les FPGA (Field Programmable Gate Array) sont devenus l'option favorite pour la mise en œuvre des systèmes embarqués. Grâce aux cellules SRAM, il est facile de reconfigurer un FPGA autant de fois pour implanter les fonctionnalités désirées. Les FPGA actuels disposent de ressources matérielles qui ne cessent d'augmenter en

quantité et en qualité, de plus ils intègrent une panoplie de ressources logiques et arithmétiques, de la mémoire et des processeurs (Microblaze, PowerPC). On parle ainsi de système sur circuit programmable (Système on Programmable Chip SoPC). La souplesse sans précédente des circuits FPGA est en fait une approche idéale pour les applications complexes de conception de systèmes embarqués, ainsi que pour les applications fortement calculatoire comme celles qu'on trouve dans le domaine de la cryptographie [2].

Ces dernière années, les réseaux de communication ne cessent d'évoluer, pour permette des échanges intensifs de données et des transactions électroniques. Cependant, les réseaux en question doivent êtres fiables et efficaces afin d'assurer un maximum de sécurité et de confidentialité aux utilisateurs. C'est le rôle de la cryptographie qui permet d'empêcher les intrus de pirater (copier, modifier ou supprimer) des informations au cours d'une transaction électronique.

En revanche, la cryptanalyse est l'étude des procédé cryptographique, dans le but de trouver des faiblesses, en particulier de décrypter des messages chiffrés sans connaitre la clé de déchiffrement. Pour que la cryptographie résiste à la cryptanalyse, des crypto-systèmes sont composés par diverses fonctions complexes qui assurent non seulement la sécurité des données mais aussi leurs intégrité, authenticité,…[3.4]. De ce fait, l'utilisation des systèmes SoPC pour la réalisation des crypto-systèmes est une solution qui permet l'implémentation de plusieurs fonctions cryptographiques sur une même puce et dont la gestion du système est assurée par un ou plusieurs processeurs [1].

La cryptographie a connu deux types : la cryptographie symétrique qui utilise une seule clé secrète pour le chiffrement et le déchiffrement et la cryptographie asymétrique qui utilise une paire de clés, une clé publique pour le chiffrement et une clé privée pour le déchiffrement.

Dans ce travail, on s'intéresse au cryptosystème à clé publique le RSA [5] qui utilise une paire de clés : une publique (E,N) pour le chiffrement et une privé (D,N) pour déchiffrement.

Les deux opérations de chiffrement et déchiffrement sont basées sur le calcul de l'exponentiation modulaire ($C=M^E \bmod N$) ou ($C=M^D \bmod N$) qui se réalise en une suite de multiplications modulaires ($A \times B \bmod N$).

Le nombre de ces multiplications modulaires dépend de la taille de la clé publique E qui doit être au moins égale à 1024 bits pour assurer une sécurité acceptable.

Pour accélérer le cryptosystème RSA, il faut d'un côté réduire le nombre de multiplications induites lors du calcul de l'exponentiation modulaire et de l'autre réduire le temps d'exécution de cette multiplication.

L'objectif de ce travail est d'implémenter un crypto-système RSA embarqué sur circuit FPGA à base du processeur Microblaze. Afin d'atteindre un meilleur compromis (temps d'exécution, surface occupée), deux approches d'implémentations ont été réalisée. Une approche purement logicielle où les calculs de l'exponentiation modulaire et de la multiplication de Montgomery sont exécutées par le processeur Microblaze. Une autre approche combinant l'implémentation logicielle et matérielle où le calcul de l'exponentiation modulaire est effectué par Microblaze et la multiplication modulaire de Montgomery est implémentée sur matériel.

Pour réaliser ce projet de fin d'étude qui nous a été proposé au sien de l'équipe AC2 de la division Architecture des Systèmes et Multimédia (ASM) au centre de développement des Technologies Avancées (CDTA), dont le thème est le suivant : « Contribution à la conception d'un crypto système embarqué sur FPGA », nous avons organisé notre mémoire en cinq chapitres répartis comme suit :

Le premier chapitre est consacré aux généralités sur la cryptographie nécessaires à la compréhension du fonctiennemet du cryptosystème RSA. Dans le deuxième chapitre, l'opération clé du cryptosystème RSA qui est l'exponentiation modulaire est exposée ainsi que la multiplication modulaire de Montgomery. Le troisième chapitre définit ce que c'est un système embarqué et plus précisément un système sur puce et un système programmable sur puce. Le quatrième chapitre est réservé à l'implémentation des variantes algorithmiques de l'exponentiation modulaire et de la multiplication modulaire de Montgomery. Le dernière chapitre est consacré à la vérification des résultats l'implémentation de ces variantes et à étudier les performances de notre système embarqué.

Chapitre 1 :

Généralité sur la cryptographie

1.1 Introduction

Depuis des millénaires, les dirigeants de tous les pays et de tous les royaumes cherchent des moyens efficaces pour protéger le secret de leurs communications. On trouve la trace du premier signe de cryptographie il y a 4000 ans en Égypte et depuis cette époque, les techniques pour protéger les secrets n'ont cessé de s'améliorer... et les idées pour briser les protections de proliférer.

Jusqu'à une époque très récente, la cryptographie n'était utilisée que dans des domaines très restreints, principalement militaires et grandes entreprises.

Avec l'avènement des réseaux et d'Internet, du commerce électronique, la cryptologie a connu un essor considérable où le transfert de l'information se fait à travers ces réseaux dans tous les domaines, ce qui a accru la nécessité de protéger ces informations pendant leur transfert. C'est le rôle de la cryptographie de protéger ces informations et d'assurer leur confidentialité.

Dans ce chapitre, un aperçu des techniques de la cryptographie moderne et de ces deux types, à savoir cryptographies symétrique et asymétrique est donné pour comprendre les fonctions offertes par la cryptographie.

1.2 Définition de la cryptographie

L'origine du mot cryptographie provient du grec kryptós (caché) et gráfein (écrire). On peut définir la cryptographie comme l'ensemble des techniques permettant de protéger une communication, par exemple l'assurance que l'information contenue dans un message ne soit révélée qu'au seul destinataire de ce message [7].

La manière la plus simple de transmettre un message de manière sécurisée entre deux entités est d'utiliser un canal sécurisé, par exemple en faisant appel à un porteur en qui vous avez une totale confiance et qui va aller porter votre message à votre correspondant. S'il se fait capturer par l'ennemi, ce porteur ne devra rien dire... Cependant, il est évident que les canaux sécurisés ont des contraintes trop importantes pour être couramment utilisés en pratique.

A cet effet, la cryptographie est une science permettant de correspondre de manière sécurisée en utilisant des canaux non sécurisés, elle permet de convertir des informations "en clair" en informations codées, c'est à dire non compréhensibles, puis, à partir de ces informations codées, on restitue les informations originales.

Pour cela, une fonction de chiffrement est appliquée au message à transmettre pour obtenir le texte chiffré qui pourra être transmis à l'autre entité. Celle-ci déchiffrera le texte chiffré afin de restituer le texte clair.

D'autres termes référant à la cryptologie sont nécessaires et leurs définitions sont les suivantes:

– **Le chiffrement :** appelé aussi « *cryptage* » qui est la procédure de rendre un message clair illisible ou chiffré.

– **Le déchiffrement :** appelé aussi « *décryptage* » qui est la procédure inverse du chiffrement et consiste à retrouver le message clair à partir du message chiffré en connaissant la clé de décryptage.

– **La clé de chiffrement** est une donnée (une suite de 0 et 1) utilisée pour le chiffrement afin de rendre le déchiffrement plus difficile. Il existe plusieurs types de clés: les clés privées qui doivent être gardées secrètes et les clés publiques qui sont diffusées.

La longueur de la clé est le nombre de bits significatifs qui définissent la valeur de la clé (le modulo). Cette longueur peut aller de quelques dizaines de bits à quelques milliers de bits.

- **Les données claires** sont des données (texte ou image) dans leur forme initiale, non chiffrées.
- **Les données chiffrées** sont des données rendues illisibles par un algorithme de chiffrement.

De nos jours, la cryptographie consiste à mettre au point des systèmes cryptographiques afin d'assurer un des piliers suivants [8,9]:

- **Confidentialité** : qui assure que les données ne pourront être dévoilées qu'aux personnes autorisées.
- **Intégrité**: qui permet de garantir la protection d'un fichier contre toutes modifications par un tiers.
- **Authentification** : qui permet de prouver l'identité d'une personne ou l'origine d'une donnée.
- **Non-Répudiation**: assure que l'émetteur ne peut pas nier qu'il a envoyé un message.

1.3 Types d'algorithmes de cryptographie

La cryptographie est divisée en deux sous-groupes : la cryptographie symétrique et la cryptographie asymétrique.

La première repose sur le fait que deux entités partagent un même secret qui leur permettra de communiquer de manière sécurisée.

La seconde permet à n'importe quel individu de chiffrer un message et que seul un unique individu (ou une unique entité) pourra déchiffrer [9].

1.3.1 Cryptographie symétrique

Aussi appelé chiffrement à clé privée ou chiffrement à clé secrète, consiste à utiliser la même clé pour le chiffrement et le déchiffrement.

Le chiffrement s'effectue en additionnant (fonction XOR) la clé au message coupé en blocs et en effectuant plusieurs permutations, ceci se fait plusieurs fois.

Le principal avantage de ce type de chiffrement est qu'il est très rapide.

Son inconvénient est qu'il faut disposer d'un canal sécurisé pour l'échange de la clé entre les deux personnes communicantes et pose le problème de gestion des clés qui pour que n personnes puissent communiquer, il faut gérer $n \times (n-1)/2$ clés.

La figure 1.1 montre le principe du chiffrement symétrique.

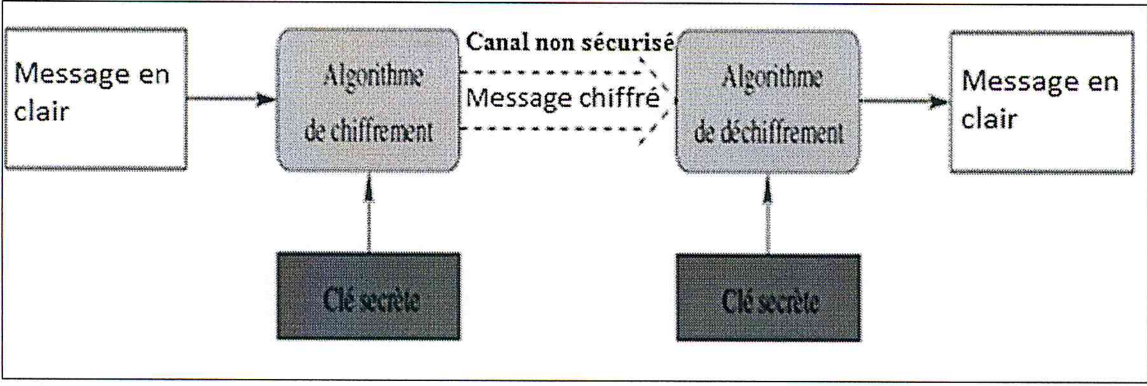


Figure 1.1- Chiffrement symétrique

Il existe deux types d'algorithmes de chiffrement symétrique. Le premier type, qui est aussi le plus utilisé en pratique, est le chiffrement par blocs (DES et AES).

Dans ce mode, les données sont traitées par bloc de bits, typiquement de 64 ou 128 bits. Le second type est le chiffrement à flot. Celui-ci permet, après une période d'initialisation, de chiffrer bit par bit [3,10,11,12].

1.3.2 Cryptographie asymétrique

Le concept de cryptographie asymétrique, aussi appelée cryptographie à clé publique, est apparu officiellement en 1976 dans le célèbre article de Whitfield Diffie et Martin Hellman, « New Directions in Cryptography ». Un tel système utilise deux clés distinctes (intrinsèquement liées): une clé publique pour crypter et une clé privée pour décrypter. La clé de chiffrement peut être largement diffusée et la clé de déchiffrement est bien sûr tenue secrète.

La cryptographie asymétrique est apparue pour résoudre les problèmes de la cryptographie symétrique. Elle s'intéresse à la sécurité de la communication. L'idée générale est que le destinataire qui choisit sa clé de déchiffrement (privée), utilise une fonction à trappe pour calculer sa clé de chiffrement (publique). Cette dernière est diffusée à toutes les personnes susceptibles de communiquer avec lui. Les fonctions à trappe utilisées dans la cryptographie asymétrique reposent sur des problèmes mathématiques réputés difficiles comme la factorisation des grands nombres [3,4].

La figure 1.2 montre le principe de chiffrement symétrique.

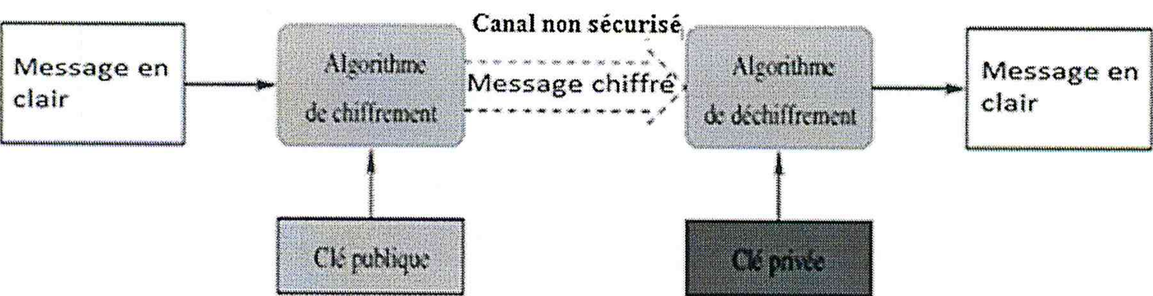


Figure 1.2- Chiffrement asymétrique

L'idée de la cryptographie à clé publique a aussi donné lieu à l'introduction du concept de signature électronique. Cette signature est l'analogue électronique d'une signature manuscrite qui s'appose sur un document afin d'engager la responsabilité du signataire. Toutefois, la signature électronique ne pourra pas être imitée ce qui implique en particulier que le signataire ne pourra jamais répudier ses engagements [3,13].

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.

À partir du message et de la signature, l'algorithme de vérification fournit soit une acceptation, soit un rejet de la signature vis-à-vis du message associé.

La figure 1.3 montre le principe de la signature électronique.

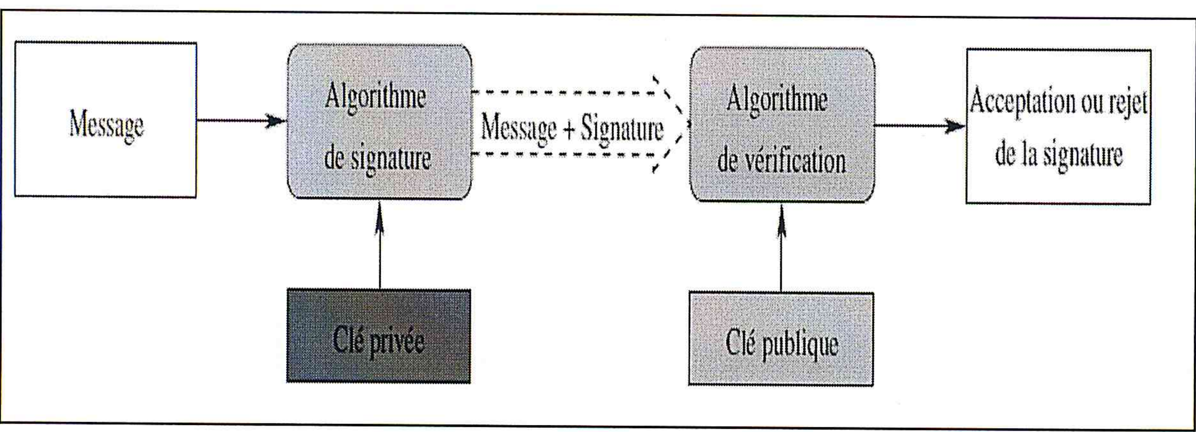


Figure 1.3- Signature électronique

Les cryptographies symétrique et asymétrique 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 grandes tailles en un temps raisonnable.

1.4. Le crypto système RSA

Le crypto-système RSA a été inventé en 1977 par Ron Rivest, Adi Shamir et Len Adleman. C'est en essayant de démontrer que l'idée de crypto système à clé publique introduit par Diffie et Hellman était une incohérence que les trois auteurs, devant leur échec, découvrirent ce système qui est devenu rapidement une référence [14].

Il sert de base à la sécurité dans beaucoup de systèmes d'information numérique tels que les applications de réseaux comme l'E-mail, l'e-commerce et les e-opérations bancaires qui se fondent sur des services comme les signatures numériques pour assurer la confidentialité, l'authentification et l'intégrité des données, et d'autres applications comme des cartes de crédit, les PDAs et téléphones portables.

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 [13].

1.4.1. Génération des clés

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

1. On choisit deux grands nombres premiers p et q pour avoir le modulo $N=p\times q$.
2. On calcule ensuite la fonction indicatrice d'Euler: $\Phi(N)=(p-1)\times(q-1)$, cette valeur va servir à déterminer les valeurs de la clé privée ***d*** et la clé publique ***e***.
3. On choisit ***e*** tel que $2 < e < \Phi(N)$ et ***e*** premiers avec $\Phi(N)$.
4. On calcule la clé privée ***d*** qui est l'inverse de *e* par l'algorithme d'Euclide étendu.
5. On rend public la paire (***e***, ***N***).
6. On cache la clé privée ***d***.
7. On cache ou on supprime p et q car ils doivent rester secrets.

On peut à présent chiffrer et déchiffrer un message avec l'algorithme RSA.

1.4.2. Chiffrement

L'algorithme de chiffrement utilise la clé publique pour chiffrer le message M par blocs de taille inférieure à N en effectuant l'opération suivante :

$$C = M^e \bmod N.$$

Ainsi, comme le modulo *N* et l'exposant *e* sont connus, alors tout le monde peut chiffrer un message.

1.4.3. Déchiffrement

La clé privée est utilisée pour retrouver le message d'origine M en réalisant l'opération suivante:

$$M=C^{d} \mod N.$$

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 1.4 :

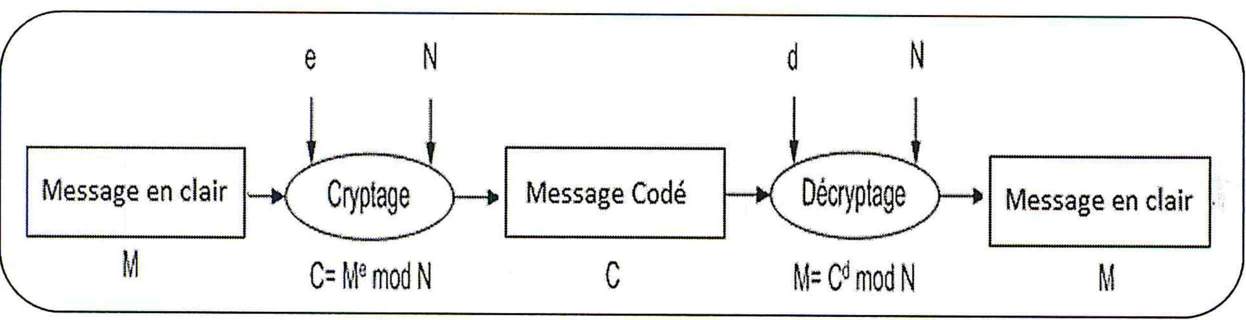


Figure 1.4: Schéma bloc du principe de fonctionnement du RSA.

1.5. Sécurité du RSA

La sécurité du RSA n'est pas due à une démonstration théorique que ce système est sûr, car celle-ci n'existe pas, mais elle provient de l'échec répété depuis plus de 30 ans de toutes les tentatives entreprises pour casser ce système. Une attaque du RSA passerait par la factorisation du nombre N. En effet celui qui sait factoriser $N=p\times q$ retrouve ensuite facilement **d** et peut donc lire les messages cryptés par le RSA, la robustesse du RSA apparaît donc liée à la difficulté de la factorisation qui augmente avec l'augmentation de la taille de N.

Actuellement les experts recommandent d'utiliser des nombres N de 1024 bits (309 chiffres décimaux) pour l'usage commerciale et 2048 bits (617chiffres décimaux) pour une garantie se prolongeant sur une longue période de temps. Les clés de 2048 bits seront toujours sûres contre la factorisation, mais avec le pouvoir de calculs grandissant des machines et le développement des méthodes de cryptanalyses personne ne peut predire le futur.

1.6. 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 N. C'est-à-dire calculer M^e mod N connue sous : *l'exponentiation modulaire* qui est une suite de multiplications modulaires et est utilisée simultanément dans le chiffrage et le déchiffrage [13].

En effet pour assurer un haut niveau de sécurité la longueur de N doit être d'au moins 1024 bits. Cette longueur ainsi que celle de l'exposant tendent à grandir dans le futur en même temps que la cryptanalyse et le pouvoir de calcul des machines progresseront.

La première règle de l'exponentiation modulaire est qu'on ne peut calculer $C=M^e$ mod N par la méthode directe en calculant M^e pour le diviser par N pour retrouver C qui est le reste de la division de M^e par N.

Cette méthode produit beaucoup de calculs intermédiaires qui donnent des résultats de calculs de plus en plus longs devront être stockées et l'espace mémoire requis pour le stockage de ces résultats ainsi que celui du nombre binaire M^e est énorme.

Quand les résultats temporaires sont réduits en modulo N à chaque étape de l'exponentiation, la taille des données manipulées reste limitée par celle de N, on ajoute ainsi de nouvelles opérations (pour chaque multiplication une division euclidienne), mais on gagne par ailleurs du fait que la taille des résultats est limitée par celle du modulo N.

De plus, le calcul de $C=M^e$ mod N nécessite (e-1) multiplications, ce qui est infaisable pour des clés de l'ordre de 1024 bits.

Une autre complexité est dans la réalisation de la multiplication elle-même celle-ci est liée à la taille du message M 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.

1.7. Conclusion

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

La cryptographie symétrique, utilisant une clé secrète pour le chiffrement et déchiffrement, est rapide et simple à implémenter mais nécessite l'échange préalable de clé via un canal de transmission sécurisé mais non disponible.

La cryptographie asymétrique, utilisant une paire de clé : publique pour le cryptage et secrète pour le décryptage, présente l'avantage d'échanger des messages de manière sûre sans échange préalable de secret qui est la clé. Plus la taille de la clé est grande, meilleure est la sécurité du système mais le temps de calcul est plus long: ce qui est gagné en sécurité est donc perdu en rapidité.

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.

L'exponentiation modulaire est une opération très gourmande en temps de calcul du moment que la taille de la clé est d'au moins de 1024 bits.

Pour accélérer les opérations de cryptage et de décryptage, on doit optimiser d'une part le calcul de l'exponentiation modulaire en réduisant le nombre de multiplications modulaires induites et d'autre part réduire le temps d'exécution de cette multiplication modulaire.

Dans le prochain chapitre, nous présentons la méthode d'exponentiation modulaire binaire « square and multiply » qui est rapide ainsi que ses variantes LSB et MSB et enfin la multiplication modulaire la plus rapide qui est basée sur l'algorithme de Montgomery, pour une éventuelle implémentation sur circuit FPGA de l'exponentiation modulaire basée Montgomery.

Chapitre 2 :

Exponentiation modulaire de Montgomery

2.1 Introduction

Le crypto système à clé publique RSA est basé sur le calcul de l'exponentiation modulaire qui est utilisée lors du chiffrage d'un message M en utilisant la formule $C = M^e Mod\ N$ et du déchiffrage pour restituer le message d'origine en utilisant la formule $M = C^d Mod\ N$, où *(e, N)* est la clé publique, *(d, N)* est la clé privée, *N* est le modulo, M est le message à chiffrer dont la taille en bits est comprise entre 0 et n-1 et *C* est le message chiffré.

Les tailles des clés actuelles du protocole RSA sont de l'ordre de 1024 et plus, qui rend le calcul de l'exponentiation onéreux.

En fait, l'exponentiation modulaire est réalisée en une série répétée de multiplications modulaires. Pour accélérer l'exécution du chiffrement/Déchiffrement du crypto système RSA, des optimisations doivent être faites respectivement sur les plans algorithmique (réduire le nombre d'opérations) et architecturale (augmenter la vitesse d'exécution de la multiplication modulaire).

Dans ce chapitre, nous présentons la méthode rapide d'exponentiation modulaire binaire et ses variantes LSB et MSB ainsi que la multiplication modulaire la plus utilisée basée sur l'algorithme de Montgomery, pour une éventuelle implémentation sur circuit FPGA de l'exponentiation modulaire basée sur Montgomery.

2.2 L'exponentiation modulaire

En mathématique et plus précisément en arithmétique modulaire, *l'exponentiation modulaire* est un type d'élévation à la puissance (exponentiation) modulo un entier. Elle est très utilisée en cryptographie, spécialement dans le crypto système à clé publique, le RSA.

Le calcul naïf de l'exponentielle modulaire est le suivant : on multiplie e fois le nombre M par lui-même et une fois l'entier M^e obtenu, on calcule son reste modulo N *via* l'algorithme de division euclidienne.

Une procédure simple pour calculer $C = M^e \bmod N$ basée sur la méthode de multiplication à la main est décrite par l'algorithme 2.1.

Algorithme 2.1- *Méthode d'exponentiation modulaire simple* (M, e, N)

1: $C = M$

2: ***Pour*** i ***de*** 1 ***jusqu'a*** $e - 1$

3: $C := (C \times M) \bmod N$

4: ***retourner*** C

Fin de l'algorithme.

Cette méthode exige (e-1) multiplications modulaires.

Elle calcule toutes les puissances de M: $M \rightarrow M^2 \rightarrow M^3 \rightarrow \ldots \rightarrow M^{e-1} \rightarrow M^e$.

La méthode de calcul d'exponentiation modulaire simple calcule plus de multiplications que nécessaire.

Pour calculer M^8 par exemple, 7 multiplications sont nécessaires, c-à-d :

$$M^8: M \rightarrow M^2 \rightarrow M^3 \rightarrow M^4 \rightarrow M^5 \rightarrow M^6 \rightarrow M^7 \rightarrow M^8.$$

Cette méthode de calcul est inefficace car il existe des algorithmes qui peuvent réduire ce nombre de multiplications de 7 à 3 en utilisant l'exponentiation binaire calculant M^8 comme suit : $M \rightarrow M^2 \rightarrow M^4 \rightarrow M^8$.

2.3 L'exponentiation modulaire binaire

C'est une méthode simple et pratique pour effectuer rapidement le calcul des puissances ($C = M^e \bmod N$) pour de grandes valeurs de l'exposant e en décomposant ce dernier en sa représentation binaire.

Sa complexité (comptée en nombre de multiplications à effectuer) est une fonction linéaire dépend du nombre de bits de l'exposant.

$$e = (e_{n-1}, e_{k-2}, \ldots\ldots\ldots\ldots e_1, e_0)_2 \quad avec \quad e = \sum_{i=0}^{n-1} e_i \times 2^i \quad et \ e_i \in \{0,1\}.$$

La méthode binaire est une suite de mise au carré et de multiplication, d'où son appellation "*square and multiply* ", parce que nous élevons au carré à chaque itération et nous multiplions seulement si le bit de l'exposant $e_i=1$.

Deux variantes de la méthode binaire existent qui dépendent du sens dans lequel les bits de l'exposant sont considérés : soit de la gauche vers la droite, à partir du bit le plus significatif, MSB (Most significant bit) ou L-R pour (Left-to-Right) ou dans le sens contraire,

à partir du bit le moins significatif LSB (Least Significant Bit) ou R-L pour (Right -to- Left) [13].

2.3.1 Exponentiation modulaire binaire MSB (Most significant bit)

Cette méthode est basée sur la lecture des bits de l'exposant *e* à partir du bit le plus significatif et il est représenté par l'algorithme 2.2.

Algorithme 2.2- Exponentiation modulaire binaire MSB

Entrée: M, e, N

Sortie: $C = M^e \; Mod \; N$

Début

$C = 1$

Pour i *de* $n - 1$ ***jusqu'*** à 0 *n nombre de bit de e*

\\ $e_{(n-1)}$ *le bit le plus significatif*

a. $C = C \times C \; Mod \; N$

b. ***Si*** $(e_i = 1)$ ***alors*** $C = C \times M \; Mod \; N$

Fin pour

Retourner C

Fin

Au début de l'algorithme, on initialise une variable $C=1$, à chaque itération, on élève la variable C au carré, ensuite selon le bit e_i, on cumule la multiplication $C \times M$ dans C.

Pour un exposant e de n-bits avec $e_{n-1}=1$, la méthode MSB nécessite:

- (n-1) multiplications modulaires à l'étape **'a'** (élévations au carré).
- Le nombre de multiplications modulaires à l'étape **'b'** dépend du nombre des bits de e différents de zéro $(H(e_i=1))$. Ce nombre vaut $H(e)-1$.
- D'où le nombre total de multiplications modulaires est :

$$n \leq nombre \; multiplications \leq 2(n-1)$$

Exemple : Calcul de M^{55}, $e = 55 = (110111)_2$, $n = 6$.

Le déroulement de l'algorithme 2.2 est représenté sur le tableau 2.1.

Tableau 2.1- Déroulement de la méthode MSB pour $e=55$			
I	e_i	*Etape a*	*Etape b*
5	1	$C = 1$	$C = 1 \times M = M$
4	1	$C = (M)^2 = M^2$	$C = (M)^2 \times M = M^3$
3	0	$C = (M^3)^2 = M^6$	$C = M^6$
2	1	$C = (M^6)^2 = M^{12}$	$C = (M)^{12} \times M = M^{13}$
1	1	$C = (M^{13})^2 = M^{26}$	$C = (M)^{26} \times M = M^{27}$
0	1	$C = (M^{27})^2 = M^{54}$	$C = (M)^{54} \times M = M^{55}$

Dans cet exemple, l'exposant e contient 6 bits ; donc 6-1=5 élévations au carré sont requises. Le nombre de bits de l'exposant différent de zéro est de 5 ; donc 5-1= 4 multiplications modulaires sont nécessaires.

Le nombre total de multiplications modulaires est de 9=5 élévations au carré et 4 multiplications.

2.3.2 Exponentiation modulaire binaire LSB (Least Significant Bit)

Cette méthode repose sur la lecture de l'exposant *e* à partir du bit le moins significatif jusqu'au bit le plus significatif, en ajoutant une variable supplémentaire S pour sauvegarder les puissances de M. Cette méthode est représentée par l'algorithme 2.3.

Algorithme 2.3- Exponentiation modulaire binaire LSB

Entrée: e, M, N
Sortie: $C = M^e \; Mod \; N$

Début
    $C = 1, S = M$
    Pour i ***de*** 0 ***jusqu'***à $n-2$ \\ *n nombre de bit de e*
        \\ e_0 *le bit le moins significatif*
        a. Si (e_i==1) **alors** C = C × S Mod N;
        b. S = S × S Mod N;
    Fin pour
    c. ***Si*** ($e_{n-1} == 1$) ***alors*** $C = C \times S \; Mod \; N$
Retourner C
Fin

Pour un exposant e de n bits, avec $e_{n-1}=1$, la méthode LSB nécessite:

- *(H(e)-1)* multiplications modulaires à l'étape '**a**' dépendant du nombre de bits de e différent de zéro $(e_i=1)$.
- *(n-1)* multiplications modulaires à l'étape '**b**'.
- Une multiplication modulaire à l'étape '**c**', si $e_{n-1}=1$.

D'où le nombre total de multiplications modulaires est: $n \leq nombre\ multiplications \leq 2(n-1)$.

Pour un exposant e de n bits, avec $e_{n-1}=1$, la méthode LSB nécessite:

- *(H(e)-1)* multiplications modulaires à l'étape '**a**' dépendant du nombre de bits de e différent de zéro $(e_i=1)$.
- *(n-1)* multiplications modulaires à l'étape '**b**'.
- Une multiplication modulaire à l'étape '**c**', si $e_{n-1}=1$.

D'où le nombre total de multiplications modulaires est : $n \leq nombre\ multiplications \leq 2(n-1)$.

<u>Exemple</u>: $e = 55 = (110111)_2$ donc $n= 6, C = 1, S = M$

Tableau 2.2- Déroulement de la méthode LSB pour e=55

i	e_i	*Etape a*	*Etape b*
0	1	$C = M$	$S = (M)^2 = M^2$
1	1	$C = M \times M^2 = M^3$	$S = (M^2)^2 = M^4$
2	1	$C = M^3 \times M^4 = M^7$	$S = (M^4)^2 = M^8$
3	0	$C = M^7$	$S = (M^8)^2 = M^{16}$
4	1	$C = M^7 \times M^{16} = M^{23}$	$S = (M^{16})^2 = M^{32}$
Etape c pour i=5	$e_5=1$	$C = M^{23} \times M^{32} = M^{55}$	/

Le nombre de multiplications modulaires de la méthode LSB est le même que celui de la MSB et vaut 9 = 5 élévations au carré et 4 multiplications [13].

2.3.3 Comparaison des méthodes MSB et LSB

Les deux méthodes d'exponentiations modulaires binaires MSB et LSB requièrent le même nombre de multiplications.

- Nombre d'élévations au carré = n-1
- Nombre de multiplications = au nombre de *'1'* dans l'exposant, celui-ci peut varier de un à *n-1*.

Ce qui signifie que nous avons :

- Au maximum : $(n-1) + (n-1)= 2(n-1)$ multiplications.
- Au minimum : $(n-1) + 1= n$ multiplications.
- Et en moyenne : $(n-1) +1/2(n-1)= 3/2(k-1) = 1.5\ (n-1)$ multiplications.

Dans la méthode **MSB,** les deux opérations de multiplications et d'élévations au carré sont exécutées séquentiellement. En revanche, dans la méthode **LSB** la multiplication et l'élévation au carré sont exécutées simultanément.

Pour une implémentation matérielle, la méthode LSB peut donner de meilleures performances en termes de temps d'exécution. Néanmoins pour une implémentation software, les deux algorithmes ont les mêmes performances, ceci est dû fait que dans le processeur les opérations sont exécutées séquentiellement. A cet effet, dans ce travail, nous avons utilisé la méthode LSB.

2.4. La multiplication modulaire

2.4.1. Définition

La multiplication modulaire est largement utilisée dans les applications cryptographiques à clé publique spécialement là où l'exponentiation modulaire est essentielle comme le cas du crypto système RSA.

La multiplication modulaire consiste à effectuer la multiplication de deux nombres A et B et de prendre le reste S obtenu de la division du produit (A×B) par un troisième nombre N, appelé modulo. Cette opération est donnée par l'expression : $S= (A \times B) \bmod N$.

Cette façon de calculer a des inconvénients : le résultat intermédiaire A×B aura une taille de mot double si les tailles de A et B sont égales. Alors, il est impossible d'implémenter en hardware des multiplieurs de très grandes tailles. En plus pour retrouver le reste, cette technique nécessite une division qui est une opération de calcul multi-précision [15] la plus compliquée et la plus coûteuse en temps de calcul.

2.4.2 Description de l'algorithme Montgomery (*A, B, N*)

Soit N un nombre impair de n bits qui est le modulo intervenant dans l'opération de multiplication modulaire.

L'algorithme de Montgomery peut être appliqué que si N est impair. En effet, dans le protocole RSA, N est le résultat de la multiplication de deux grands nombres premiers. Ainsi, N est forcement impair.

L'algorithme de Montgomery propose une réduction du modulo N, en une division par une puissance de la base $R=\beta^n$ (où β est la base de numération). R est appelé constante de Montgomery [6,16].

Comme N est impair, R est premier avec N, donc R est inversible modulo N. On note par R^{-1} l'inverse de R modulo N.

Le principe de cette multiplication est donné par les points suivants [17]:

Soit $C=A\times B$, *tel que* $0\leq A,\ B<N$.

Soit $N'=(-N)^{-1} \bmod R$ *(N') : est pré calculé une seule fois).*

Si on considère $q=(C\times N') \bmod R$, *alors :*

1. $S=(C+q\times N)/R$ *est un entier.*
2. $S=(C\times R^{-1}) \bmod N$.
3. $S<2\times N$.

L'algorithme 2.4 représente l'algorithme général de la multiplication de Montgomery.

Algorithme 2.4- Algorithme de Montgomery

Entrée: A,B,N *avec* $0\leq A,B<N$

Données : N',R^{-1} *et* R *avec* : $R\geq\beta^n$, $((-N)\times N')\bmod R=1$

$$R^{-1}\times R \bmod N=1$$

Variables : C,q

Sortie : S *tel que* $S=(A\times B\times R^{-1}) \bmod N$

Début

a. $C=A\times B$

b. $q=C\times N' \bmod R$

c. $S=(C+q\times N)/R$

d. ***Si***$(S\geq N)$ ***alors*** $S=S-N$

Retourner S;

Fin

2.4.3 Calcul de la multiplication Montgomery

A l'origine, la multiplication modulaire de Montgomery calcule non pas $S=A\times B \bmod N$, mais une réduction avec un facteur supplémentaire, c'est-à-dire $S=A\times B\times R^{-1} \text{ Mod } N$. Pour

parvenir à la réduction de la multiplication $(A \times B)$ en modulo N, il existe des méthodes pour éliminer le facteur R^{-1}.

La première méthode consiste à:

1. Calculer C tel que : $C = Montgomery\ (A, B, N)$
2. Calculer S tel que : $S = Montgomery\ (C, R^2, N)$

La deuxième méthode se base sur une conversion de A et B dans le domaine de Montgomery.

2.4.3.1 Première méthode

L'algorithme 2.5 représente la première méthode du calcul de la multiplication modulaire.

Algorithme 2.5- Méthode 1 de calcul de la multiplication modulaire

Entrée: $A, B, N\ avec\ 0 \le A, B < N$
Données : $R\ avec : R \ge \beta^n$
Variables: C
Sortie : $S\ tel\ que\ S = (A \times B)\ mod\ N$
Début

a. $C = Montgomery\ (A, B, N)$

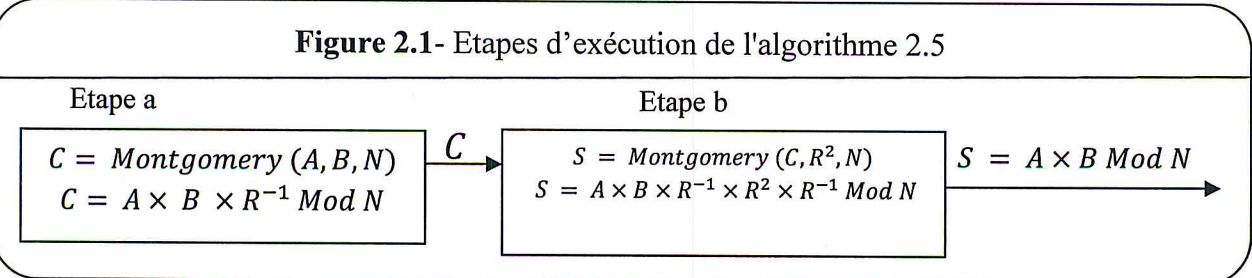
b. $S = Montgomery\ (C, R^2, N)$

Retourner S

Fin

Les étapes d'exécution de la première méthode sont résumées sur la figure 2.1.

Figure 2.1- Etapes d'exécution de l'algorithme 2.5



Exemple

Soit $A = 250$, $B=123$, $N=329$, nous calculons $A \times B\ Mod\ N$ en Base $\beta=10$.

Les prés calculs de l'algorithme 3 sont:

$R = \beta^3 = 1000.$

$R^2 = 1000^2\ Mod\ 329 = 169.$

$N' = -329^{-1}\ mod\ 1000 = 231$

Dans cet exemple, on applique l'algorithme 2.5 pour calculer la multiplication modulaire en utilisant Montgomery, les résultats obtenus sont représentés dans les deux tableaux ci-dessous.

Tableau 2.3- Etape "a" de la première méthode

Montgomery (A, B, N)	
$C = A \times B$ $q = C \times M' \bmod R$ $S = (C + q \times N)/R$	$C = 250 \times 123 = 30750$ $q = 30750 \times 231 \bmod 1000 = 250$ $q = 30750 + 250 \times 329 / 1000 = 113$
Montgomery (A,B,N)=113	

Tableau 2.4- Etape "b" de la première méthode

Montgomery $(113, R^2, N)$	
$C = 113 \times R^2$ $q = C \times M' \bmod R$ $S = (C + q \times N)/R$	$C = 113 \times 169 = 19097$ $q = 19097 \times 231 \bmod 1000 = 407$ $q = 19097 + 407 \times 329 / 1000 = 153$
$S = A \times B \bmod N = 153$	

2.4.3.2 Deuxième méthode

Cette méthode utilise une représentation particulière dite notation de Montgomery, celle-ci est noté par: *mon(x)*. Cette notation consiste à associer pour chaque entrée x inférieure à N, une valeur égale à $mon(x) = (x \times R^2 \times R^{-1}) = (x \times R)$. Ainsi le calcul de la multiplication modulaire de $(A \times B) \bmod N$ en utilisant la multiplication modulaire de Montgomery est effectué en trois étapes.

1. Convertir A et B dans le domaine de Montgomery
2. Calculer la multiplication modulaire dans le domaine de Montgomery
3. Sortir du domaine de Montgomery

L'algorithme 2.6 représente la deuxième méthode du calcul de la multiplication modulaire.

Algorithme 2.6- Méthode 2 de calcul de la multiplication modulaire

Entrée: A, B, N *avec* $0 \le A, B < N$
Données : R *avec* : $R \ge \beta^n$
Variables: C
Sortie : S *tel que* $S = (A \times B) \bmod N$
Début

a. $A = Mon(A)$ \\ *conversion de A dans le domaine*

b. $B = Mon(B)$ \\ *conversion de B dans le domaine*

c. $C = Montgomery\ (A, B, N)$ \\ *C est le résultat de la multiplication dans le domaine*

d. $S = Montgomery\ (S, 1, N)$ \\ *Sortie du domaine de Montgomery*

Retourner S;

Fin

Démonstration

Théorème : Soit $S= A\times B \; Mod \; N$, la représentation de S dans le domaine de Montgomery est :

$$Mon(S) = Mon(A) \times Mon(B) \times R^{-1} \, mod \, N \; .... (1)$$

$$Mon(S) = A \times B \times R \, mod \, N \; .... (2)$$

Preuve du théorème

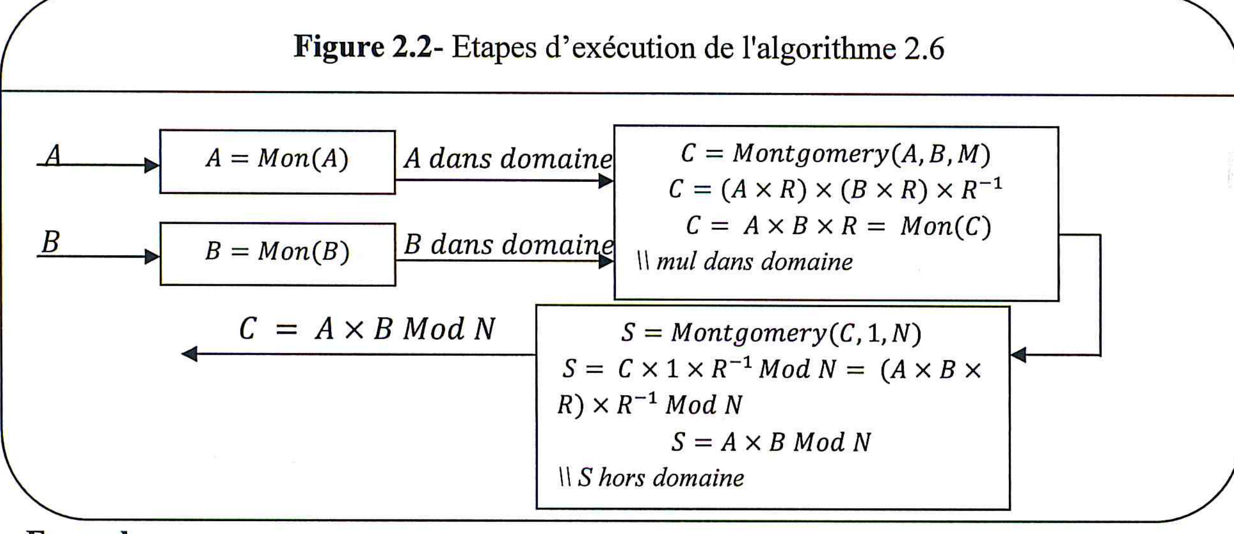
(1) $-\boldsymbol{Mon(A)} \times \boldsymbol{Mon(B)} \times \boldsymbol{R^{-1}} \, \boldsymbol{mod} = A \times R \; Mod \; N) \times (B \times R \; Mod \; N) \times R^{-1} \; Mod \; N$ $= A \times B \times (R \times R \times R^{-1}) \; Mod \; N = (A \times B) \times R \; Mod \; N = S \times R \; Mod \; N = \boldsymbol{Mon(S)}$

(2) – Selon la définition de **Mon** : $Mon(S) = S \times R \; Mod \; N$

$$\boldsymbol{Mon(S)} = S \times R \; Mod \; N = \boldsymbol{A \times B \times R \; Mod \; N}$$

Les étapes d'exécution de la deuxième méthode sont résumées sur la figure 2.2

Figure 2.2- Etapes d'exécution de l'algorithme 2.6



Exemple

Soit $A= 250$, $B=123$, $N=329$, nous calculons $A\times B \; Mod \; N$ en Base $\beta=10$.

Les prés calculs de l'algorithme 2.6 sont:

$R = \beta^3 =1000$. $\quad R^2 = 1000^2 \; Mod \; 329 = 169$. $\quad N' = -329^{-1} \; mod \; 1000 =231$

Dans cet exemple, on applique l'algorithme 2.6 pour calculer la multiplication modulaire en utilisant Montgomery, les résultats obtenus sont représentés dans les tableaux ci-dessous.

Tableau 2.5- Etape "a" de la Méthode 2

Mon(A)	
$C = A\times R^2$	$C= 250\times 169 = 42250$
$q = C\times N' \; mod \; R$	$q= 42250 \times 231 \; mod \; 1000 = 750$
$S = (C+q\times M)/R$	$S= 42250 + 750\times 329 \,/\, 1000 = 289$
$A=Mon(A)=289$	

Tableau 2.6- Etape "b" de la Méthode 2	
Mon (B)	
$C = B \times R^2$ $q = C \times M' \bmod R$ $S = (C+q \times M)/R$	$C=123 \times 169 = 20787$ $q= 20787 \times 231 \bmod 1000 = 797$ $S= 20787 + 797 \times 329 / 1000 = 283$
B=Mon (B)=283	

Tableau 2.7- Etape "c" de la Méthode 2	
Montgomery (A, B, M)	
$C = A \times B$ $q = C \times M' \bmod R$ $S = (C+q \times M)/R$	$C= 289 \times 283 = 81787$ $q= 81787 \times 231 \bmod 1000 = 797$ $S= 81787 + 797 \times 329 / 1000 = 344$
S=Montgomery(A,B,M)=344	

Tableau 2.8- Etape "d" de la Méthode 2	
Montgomery(S,1)	
C = S×1 q = C×M' mod R S = (C+q×M)/R	$C= 344 \times 1 = 344$ $q= 344 \times 231 \bmod 1000 = 464$ $S= 344 + 464 \times 329 / 1000 = 153$
S=A×B Mod M = 153	

2.5 Variantes de la multiplication modulaire de Montgomery

Dans le RSA, les clés de chiffrement ou de déchiffrement sont généralement de l'ordre de 1024 bits. Cependant, quand la taille allouée à l'exécution de l'algorithme est limitée, lors du calcul de la multiplication modulaire ; les résultats intermédiaires de l'exponentiation modulaire sont volumineux et présentent un grand inconvénient dans une implémentation matérielle. En fait, afin de rendre possible l'implémentation de la multiplication modulaire, des modifications ont été portées dans l'algorithme original, à partir desquelles plusieurs variantes ont été développée.[17].

2.5.1. La multiplication modulaire de Montgomery sans soustraction finale

L'algorithme de la multiplication modulaire de Montgomery sans soustraction finale est montré sur l'algorithme 2.7.

Algorithme 2.7- Multiplication modulaire de Montgomery entrelacée sans soustraction finale

Entrées: $A = \sum_{i=0}^{n} a_i * \beta^i$, $B = \sum_{i=0}^{n} b_i * \beta^i$, $N = \sum_{i=0}^{n-1} N_i * \beta^i$, avec $A \geq 0, B < 2 \times N$

Pré calculs : $N' = (-N)^{-1} \; mod \; \beta$, avec $\beta = 2^k$, et $\gcd(N, \beta) = 1, R = 2^{n \times k + k}$ avec $k \geq 2$.

Variable intermédiaire : $q_i, S_i = \sum_{j=0}^{n} S_{i,j} \times \beta^j$

Sortie: $S = \sum_{j=0}^{n} S_{n,j} \times \beta^j = (A \times B \times R^{-1} \; mod \; N)$

Debut

$S_0 = 0$

Pour i ***de*** 0 ***jusqu'***à n

$q_i = ((S_{i,0} + (a_i \times b_0)) \times N') \; mod \; \beta$

$S_{i+1} = (S_i + (a_i \times B)) + (q_i \times N) \,/\, \beta$

Fin pour

Fin

L'exécution de cet algorithme peut être résumée en les points suivants:

- Les opérandes A et B doivent être représentés sur $(n+1)$ chiffres, puisque ces derniers sont inferieurs à $2 \times N$.
- La constante R de Montgomery doit satisfaire la condition $R \geq 2^{n \times k+2}$
- La détermination de q_i est justifiée par le fait que $S_i+(a_i \times B)+(q_i \times N)$ est divisible par β. Celui-ci est calculé à chaque itération, en considérant uniquement la partie basse de l'expression : $q_i = (S_{i,0}+(a_i \times b_0)) \times N'$
- Apres avoir calculé q_i, S_{i+1} est obtenu par l'expression:

$$S_{i+1} = (S_i+(a_i \times B)) + (q_i \times N) / \beta$$

- Pour que les résultats intermédiaires S_{i+1} à chaque itération et à la sortie soient inférieurs strictement à $2 \times N$, il est nécessaire d'effectuer $(n+1)$ itérations.
- La complexité de l'algorithme 2.7 peut être calculée par:

$$Complexité_{alg2.7} = (n+1) \times [(2 \times mul\,(k, (n+1))) + 2 \times add(n + (2 \times k)) + mulBas(k)]$$

2.5.2 Algorithme de Montgomery entrelacé en base 2 versions série

Cet algorithme est basé sur le principe de l'algorithme entrelacé en base 2, dont les entrées *A, B* et *N*, sont représentées en base 2. Cet algorithme consiste à lire les bits de l'opérande *A* et les multiplies par *B*. L'algorithme de Montgomery en base *2* a fait l'objet de calcul de l'opération de $(S_i + a_i \times B) mod\ N$, où à chaque itération les calculs arithmétiques sont basés sur la lecture de chaque bit de *A*, qui sera multiplié par chaque mot de *B* et en ajoutant le résultat de l'itération précédente (S_i). L'avantage de cette variante est que le produit de $(a_i \times B)$ est effectué par l'opérateur logique *AND*. Pour une implémentation matérielle cette opération est moins couteuse en termes de temps d'exécution qu'une multiplication de deux mots. L'algorithme 2.8 montre cette la variante.

Déroulement de l'algorithme 2.8

En premier lieu, le traitement est axé sur la décomposition des opérandes *A*, *B* et le modulo *N* sur *(e+1)* mots et chaque mot contient *w* bits. Il est à noter qu'on rajoute 1 mot supplémentaire, pour répondre à la taille des opérandes exigés par l'algorithme de Montgomery sans soustraction finale et taille est de *(n+2)* bits au lieu de n bits. Ainsi la représentation des opérandes *A*, *B* et *N* devient :

$$A = \sum_{p=0}^{e} A[p] \times 2^{p\times w}, \text{ et } A[p] = \sum_{k=0}^{w} a_k^p \times 2^k.$$

$$B = \sum_{p=0}^{e} B[p] \times 2^{p\times w}, \text{ et } B[p] = \sum_{k=0}^{w} b_k^p \times 2^k.$$

$$N = \sum_{p=0}^{e} N[p] \times 2^{p\times w}, \text{ et } N[p] = \sum_{k=0}^{w} n_k^p \times 2^k,$$

La figure 2.3 montre la décomposition des données d'entrées où on prend *A* sur 1024 bits.

$0,a_n$	A[e-1]	....	A[2]	A[1]	A[0]
$0,b_n$	B[e-1]	....	B[2]	B[1]	B[0]
0	N[e-1]	....	N[2]	N [1]	N[0]
$S[e]_i$	$S[e-1]_i$	....	$S[2]_i$	$S[1]_i$	$S[0]_i$

Figure 2.3- La représentation des données.

Algorithme 2.8- algorithme de Montgomery en base 2

Début

Entrées : $A = \sum_{p=0}^{e} A[p] \times 2^{p\times w}, avec\ A[p] = \sum_{k=0}^{w-1} a_k^p \times 2^k$

$B = \sum_{p=0}^{e} B[p] \times 2^{p\times w},\ avec\ B[p] = \sum_{k=0}^{w-1} b_k^p \times 2^k$

$N = \sum_{p=0}^{e} N[p] \times 2^{p\times w}, avec\ \mathrm{N}[p] = \sum_{k=0}^{w-1} \mathrm{n}_k^p \times 2^k$

Variables intermédiaires :

$S = \sum_{p=0}^{e} S[p] \times 2^{p\times w}, avec\ S[p] = \sum_{k=0}^{w-1} s_k^p \times 2^k$

$Z0 = \sum_{p=0}^{e} Z0[p] \times 2^{p\times w},\ avec\ Z[p] = \sum_{k=0}^{w-1} z0_k^p \times 2^k$

$Z1 = \sum_{p=0}^{e} Z1[p] \times 2^{p\times w}, avec\ Z1[p] = \sum_{k=0}^{w-1} z1_k^p \times 2^k$

$Rz0, R1z1, R2z1$

Sortie : $S = \sum_{p=0}^{e} S[p] \times 2^{p\times w} = A \times B \times 2^{-(n+2)} mod\ M$

Début :

$S = \sum_{p=0}^{e} S[p] = 0.$

Pour p ***de*** 0 ***jusqu'à*** $e - 1$

Pour k ***de*** 0 ***jusqu'à*** $w - 1$

$Rz0 = 0,\ R1z1 = 0, R2z1 = 0$

$i = w \times p + k$

$q_{(i)} = (S[0] + a_k^p \times B[0]) \bmod 2$

Si $q_{(i)} == 0$ ***alors***

Pour j ***de*** 0 ***jusqu'à*** e

$(Rz0, Z0[j]) = S[j] + (a_k^p\ AND\ B[j]) + Rz0$

$S = Z0/2$ //(*décalage d'un bit à droite*)

Sinon

Pour j ***de*** 0 ***jusqu'à*** e

$(R2z1, R1z1, Z1[j]) = S[j] + (a_k^p\ AND\ B[j]) + N[j] + R1z1 + (R2z1 \times 2)$

$S = Z1/2$ //(*décalage d'un bit à droite*)

Fin pour

Fin pour

$p = e$, ***Pour*** k ***de*** 0 ***jusqu'à*** 1

$Rz0 = 0,\ R1z1 = 0, R2z1 = 0$

$i = w \times p + k$

$q_{(i)} = (S[0] + a_k^p \times B[0]) \bmod 2$

Si $q_{(i)} == 0$ ***alors***

Pour j ***de*** 0 ***jusqu'à*** e

$(Rz0, Z0[j]) = S[j] + (a_k^p\ AND\ B[j]) + Rz0$

$S = Z0/2$ //(*décalage d'un bit à droite*)

Sinon

Pour j ***de*** 0 ***jusqu'à*** e

$(R2z1, R1z1, Z1[j]) = S[j] + (a_k^p\ AND\ B[j]) + N[j] + R1z1 + (R2z1 \times 2)$

$S = Z1/2$ //(*décalage d'un bit à droite*)

Fin pour

Fin

Fin

Pour avoir une bonne implémentation de cet algorithme, des variables intermédiaires sont ajoutées pour stocker les résultats intermédiaires. Ces variables sont *Z0, Z1* et *S*. Elles ont la même représentation que *A*, *B* et *N*. La propagation des retenues lors de l'addition, nous pose un problème. De ce fait nous avons ajouté les trois variables *Rz0, R1z0, R1z1* chacune permet de stocker une retenue de 1 bit.

L'exécution de cet algorithme dépend de trois boucles. La boucle (p) qui a pour rôle de parcourir les mots de l'opérande *A*. La deuxième boucle (*k*) qui varie de 0 jusqu'à *w-1* pour parcourir chaque bits du mot *A[p]*. La dernière boucle (j) a pour l'objectif de parcourir les mots de l'opérande *B*. L'exécution de l'algorithme se fait en deux étapes : où la première correspond à la variation de p de 0 à *e-1* et la seconde est associée à *p=e*.

Dans la première étape, avant d'entamer les calculs itératifs, on commence par initialiser les mots de S par la valeur 0 tel que : S= $\sum_{p=0}^{e} S[p] = 0$.

Pour chaque bit de $A[p]$ représenté par a_k^p, on calcule la valeur de q_i donnée par :

$$q_i = (S_i[0] + a_k^p \times B[0]) mod\ 2.$$

La valeur de q_i est soit *1* ou *0*. On constate que l'exécution de la boucle *(j)* est dépendante de la valeur de q_i, tel que si q_i*=0,* on stocke dans $Z0[j]_i = S[j]_i + a_k^p \times B[j] + Rz0$. Cette formule doit être optimisée par l'utilisation de l'operateur logique *AND* comme une alternative de la multiplication ($a_k^p \times B[j]$) car on sait que a_k^p est un bit valant 1 ou 0.

Ainsi cette multiplication devient : (a_k^p & $B[j]$). Le *Rz0* est la valeur de la retenue issue de l'addition de deux mots de 32 bits de l'itération (j-1). Le résultat de cette addition est stocké dans $Z0[j]_i$. Cette opération s'exécute, en parcourant les opérandes B et S mot par mot. Le deuxième cas est lorsque q_i*=1*. On calcule les *Z1[j]* qui ont la mêmes formule que *Z0[j]* mais en ajoutant la valeur de *N[j]* qui représente le $j^{ème}$ mot du modulo *N*. Ceci est pour garantir que *Z1* est divisible par 2. On ajoute aussi au *Z1[j]* les deux retenues de l'itération précédente (j-1) : *R1z0* et *R1z1*, car $S[j]_i + a_k^p \times B[j] + N[j]$ donne une valeur de 34 bits. Les deux bits les plus significatifs représentent les retenues générées à l'itération *(j)*. À la fin de la boucle *(j)*, le résultat S_i*=(Z0|Z1)/2* est obtenu par un simple décalage à droite.

Une fois que la première étape est achevée, l'algorithme est complété par l'exécution de la seconde étape, où seulement les deux bits les moins significatifs du dernier mot de l'opérande A sont pris en considération dans le calcul. Ainsi, l'algorithme est finalisé au deuxième bit du $e^{ième}$ mot de l'opérande *A (p=e et k=1).*

2.5.3 Algorithme de Montgomery entrelacé en base 2^k version série

Cet algorithme est une généralisation de l'algorithme précédent à une base quelle conque 2^k. L'algorithme 2.9 représente cette variante.

Déroulement de l'algorithme 2.9

Dans cet algorithme, nous avons retenu le même format des données d'entrées que de la variante représentée en base 2. Les données d'entrées *A, B* et *N* sont représentées sur *n* bits décomposés en *e* mots de *k* bits et en ajoutant un mot pour respecter l'algorithme de Montgomery sans soustraction finale. En plus des données en question, l'algorithme de Montgomery nécessite une constante pré-calculée en l'occurrence *N`*. Celle-ci est calculée par $N` = -N^{-1} mod\ 2^k$. Cette valeur est ignorée dans l'algorithme en base 2 car sa valeur est égale à *1*.

Il existe des notions arithmétiques qu'il faut retenir pour implémenter cet algorithme en base 2^k. La multiplication de deux nombres de *k* bits donne un résultat de $(2 \times k)$ bits $(A_{k\,Bit} \times B_{k\,bit} = C_{(2*k)bit})$, et l'addition de deux nombres de *k* bits donne une valeur représentée sur *(k+1)* bits $(A_{k\ Bit} + B_{k\ bit} = C_{(k+1)bit})$. De ce fait, nous avons ajouté les variables C0= $\sum_{i=0}^{e} C0[i] * 2^{i*k}$ et C1= $\sum_{i=0}^{e} C1[i] * 2^{i*k}$ tel que i et k représentent respectivement, le nombre de mots et leurs tailles. Ces variables servent à stocker les retenues des multiplications.

En général, le résultat de la multiplication de deux mots de *k* bits est obtenu sur $(2 \times K)$ bits. Autrement dit, on obtient deux mots de *k* bits, un mot résultat et l'autre est la retenue. En effet, dans la multiplication à longue précision où les opérandes se trouvent décomposés sur un certain nombre de (*e*) mots, l'opération de multiplication est effectuée d'une manière itérative mot par mot, et la retenue qui correspond au poids fort doit être additionnée au mot résultat de *k* bits de l'itération prochaine.

Ainsi *P1* et *P2* sont utilisés comme des variables intermédiaires pour stocker les mots résultat entre deux multiplications. Le stockage des retenues des opérations d'addition est effectué par le biais de *R1, R2, R3, R4*. Le résultat de l'itération (*i*) de l'algorithme de Montgomery est stocké aussi dans la variable S.

L'exécution de l'algorithme commence par l'initialisation des variables intermédiaires à 0. La structure des données d'entrées exige deux boucles pour l'exécution de l'algorithme. La première boucle *(i)* permet de balayer les mots de l'opérande *A* et la deuxième boucle (*j*) correspond à la lecture des mots de l'opérande *B* et des résultats intermédiaires. Dans chaque itération *(i)*, on commence par le calcul de q_i donné par :

$$q_i = ((A[i] \times B[0] + S[0]_i) * N`) mod\ 2^k.$$

Puis les opérations arithmétiques sont effectuées dans la boucle (j) afin de calculer

$$(A[i] \times B[j]\ + (q_i \times N[j]) + S[j]).$$

Le résultat final S_{i+1} est obtenu par un simple décalage d'un mot de k bits de l'expression précédente. Le résultat de la multiplication de Montgomery est stocké dans la variable S.

Algorithme 2.9- Algorithme de Montgomery en base 2^k

Début

Entrées : $A = \sum_{i=0}^{e} A[i] \times 2^{i \times k}, avec\ \ A[i] = \sum_{l=0}^{k-1} a_l^i \times 2^l$

$B = \sum_{i=0}^{e} B[i] \times 2^{i \times k},\ \ avec\ \ B[i] = \sum_{l=0}^{k-1} b_l^i \times 2^l$

$N = \sum_{i=0}^{e} N[i] \times 2^{i \times k}, avec\ \mathrm{N}[i] = \sum_{l=0}^{k-1} \mathrm{n}_l^i \times 2^l$

Variables intermédiaires :

$S = \sum_{i=0}^{e} S[i] \times 2^{i \times k}, avec\ \ S[i] = \sum_{l=0}^{k-1} \mathrm{s}_l^i \times 2^l$

$C0 = \sum_{i=0}^{e} C0[i] \times 2^{i \times k},\ \ avec\ \ C0[i] = \sum_{l=0}^{k-1} c0_l^i \times 2^l$

$C1 = \sum_{i=0}^{e} C1[i] \times 2^{i \times k}, avec\ \ C1[i] = \sum_{l=0}^{k-1} c1_l^i \times 2^l$

$P1\ avec\ \ P1 = \sum_{l=0}^{k-1} \mathrm{p1}_l \times 2^l$

$P2\ avec\ \ P2 = \sum_{l=0}^{k-1} \mathrm{p2}_l \times 2^l$

$R1, R2, R3, R4$ *//chaque variable est de 1 bit*

Pré calculés:

$N' = -N[0]^{-1}\ mod\ 2^k$

Sortie : $S = \sum_{i=0}^{e} S[i] \times 2^{i \times k} = A \times B \times 2^{-(n+k)} mod\ M$

Début :

$S = \sum_{i=0}^{e} S[i] = 0$

$C0 = \sum_{i=0}^{e} C0[i] = 0$

$C1 = \sum_{i=0}^{e} C1[i] = 0$

Pour i ***de*** 0 ***jusqu'***à e

$R1 = R2 = R3 = R4 = 0$

$P1 = A[i] \times B[0]$

$P2 = S[0] + P1$

$q_i = P2 \times N'\ mod\ 2^k$

Pour j ***de*** 0 ***jusqu'***à e

$(C1[j], P1) = A[i] \times B[j]$

$(R2, R1, P1) = P1 + C1[j-1] + S[j] + R1 + (R2 \times 2)$

$(C2[j], P2) = q_i \times N[j]$

$S[j-1]) = P1 + P2 + C2[j-1] + R3 + (R4 \times 2)$

Fin pour

$S[j-1] = R1 + (R2 \times 2) + R3 + (R4 \times 2) + C1[j-1] + C2[j-1]$

Fin pour

Fin

Fin

2.5.4 Complexité calculatoire des algorithmes en bases 2 et 2^k

La complexité calculatoire des algorithmes présentés dans les sections précédentes est proportionnelle aux nombres de multiplications et d'additions effectués dans chaque itération que multiplié par le nombre d'itérations que nécessitent chacun des deux algorithmes.
Dans l'algorithme de Montgomery en base 2, le nombre d'itérations, effectués pour parcourir l'opérande A, est basé sur l'indice p et vaut $(e+1)$. Dans chaque itération p, le $p^{ième}$ mot de l'opérande A en cours d'exécution est parcouru w fois afin d'extraire les bits a_k^p . Pour chaque a_k^p obtenu les opérations arithmétiques et logiques se répètent $(e+1)$ fois, qui correspond au nombre de mots qui constituent les opérandes B, N et S. Les opérations en question sont : une comparaison, un *AND* logique et trois additions. Ainsi la complexité de cet algorithme est :

$$Complexité_{base2} = (e+1) \times \left[w \times \left((e+1) \times (Comparaison + And + 3Add\,)\right)\right] + 2 \times \left((e+1) \times (Comparaison + And + 3Add\,)\right).$$

En revanche, l'algorithme en base 2^k permet la réduction du nombre d'itérations. Ce dernier nécessite $(e+1)$ itérations pour parcourir les mots de l'opérande A, tel que e dépend de k (e= *nombre de bits(A) / k*). De ce fait, l'exécution de l'algorithme exige $(e+1)$ opérations. Ces opérations sont : deux multiplications et quatre additions. A noter qu'une multiplication est nécessaire pour calculer les q_i. Par conséquent, la complexité de l'algorithme en base 2^k est

$$Complexité_{base\ 2^k} = (e+1) \times [(e+1) \times (4Add + 2Mul)] + (e+1)Mul.$$

2.6 Exponentiation modulaire binaire LSB de Montgomery

L'étude comparative des deux algorithmes binaire LSB et MSB a conclu que la LSB permet la parallélisassions des deux opérations de multiplication modulaire, ce qui fait d'elle la méthode la plus rapide.

Le calcul de l'exponentiation modulaire nécessite des multiplications modulaires qui sont celles de Montgomery.

L'algorithme 2.10 montre l'utilisation de la multiplication modulaire de Montgomery dans la méthode binaire LSB.

Algorithme 2.10- Algorithme binaire LSB

Début

Entrées : M, E_n, N, R^2

Variables : C, S

Sorties : C

Début

$C = Montgomery(R2,1)$

$S = Montgomery(M, R2)$

$\boldsymbol{Pour}\ i\ \boldsymbol{de}\ 0\ \boldsymbol{jusqu'}\text{à}\ n-1$

$\boldsymbol{Si}\ (E_i == 1)\ \boldsymbol{alors}\ C = Montgomery(S, C)$

$S = Montgomery(S, S)$

$\boldsymbol{Fin\ pour}$

$C = Montgomery(C, 1)$

$\boldsymbol{Fin}$

2.6.1 Fonctionnement de l'algorithme d'exponentiation binaire de Montgomery

Selon l'organigramme montré ci-dessus, cet algorithme peut être divisé en trois étapes:

➤ **Conversion des données dans le domaine de Montgomery**

Cette étape consiste à calculer les deux variables d'initialisation *C* et *S* qui sont utilisées comme valeurs initiales dans l'algorithme de Montgomery. Celle-ci se présente comme deux multiplications les suivantes :

$$C = Mon(1) = Montgomery(1, r^2, N)$$

$$S = Mon(M) = Montgomery(M, r^2, N)$$

où : $r^2 = (2^{(k*n)+k})^2 mod\ N$, 2^k est la base de représentation des données, *e* est la taille du modulo *N* dans la base 2^k. Une fois cette étape est achevée, tout le processus itératif du calcul de l'exponentiation modulaire est effectué dans le domaine de Montgomery en utilisant les variables *C* et *S* [18].

➤ **Calcul itératif de la multiplication modulaire de Montgomery**

Cette étape consiste à faire des décalages sur l'exposant *E*, du bit le moins significatif au plus significatif à fin d'avoir la valeur du $i^{éme}$ bit. Selon la valeur obtenue, on calcul *C* par la multiplication $C = Montgomery(C, S, N)$ si le bit en question égale à '1'.

S est une variable qui varie à chaque itération. Quelle que soit la valeur du $i^{éme}$ bit, le calcul de S est indépendant de C et qui n'est autre qu'une élévation au carré:

$$S = Montgomery(S, S, N).$$

➢ **Représentation du résultat dans le domaine classique**

Cette étape consiste à sortir du domaine de Montgomery, en appliquant une multiplication de Montgomery sur la valeur de C obtenu à l'itération $i=n-1$. Le résultat est stocké dans C. Cette étape est effectué par : $C = Montgomery(C, 1, N)$.

Exemple

Soit $E = 3$ qui est représenté en base 2 par : $E=(11)_2$.

Etape 1

$$C = Mon(1) = Montgomery(1, r^2, N) = r\ Mod\ N$$

$$S = Mon(M) = Montgomery(M, r^2, N) = M \times r\ mod\ N$$

Etape 2

Pour $i=0$:

$$C = Montgomery(C, S, N) = r \times M \times r \times r^{-1}\ mod\ N = r \times M\ mod\ N.$$

$$S = Montgomery(S, S, N) = M \times r \times M \times r \times r^{-1}\ mod\ N = M^2 \times r\ mod\ N.$$

Pour $i=1$:

$$C = Montgomery(C, S, N) = r \times M \times M^2 \times r \times r^{-1}\ mod\ N = r \times M^3\ mod\ N.$$

$$S = Montgomery(S, S, N) = M^2 \times r \times M^2 \times r \times r^{-1}\ mod\ N = M^4 \times r\ mod\ N.$$

Etape 3

$$C = Montgomery(C, 1, N) = M^3 \times r \times 1 \times r^{-1}\ mod\ N = M^3\ mod\ N.$$

Ainsi nous avons obtenu le résultat correct dans le domaine classique.

2.6.2 Complexité de l'algorithme de l'exponentiation modulaire

Le coût de cet algorithme dépend du nombre d'utilisations de la multiplication de Montgomery. L'algorithme cité ci-dessus nécessite $(2n+3)$ multiplications de Montgomery, où n est le nombre de bits de l'exposant E.

Les $2n+3$ multiplications correspondent à :

- ➢ Deux multiplications modulaires lors de l'initialisation de C et S.
- ➢ $2 \times n$ multiplications modulaires pour i variant de 0 jusqu'à $n-1$. Le nombre en question correspond au pire cas, c'est-a-dire, si tous les bits de E sont égaux à 1. La dernière élévation au carré est supplémentaire car sa valeur n'entre pas dans les calculs.
- ➢ Une dernière multiplication modulaire pour sortir du domaine.

2.7 Conclusion

Dans ce chapitre, nous avons présenté l'exponentiation modulaire binaire « élévation au carré et multiplication et ses deux variantes MSB et LSB. Cette dernière a été utilisée pour sa rapidité car elle permet aux opérations d'élévation au carré et multiplication de s'exécuter en parallèle.

Afin d'accélérer l'exponentiation modulaire qui n'est autre qu'une suite de multiplications modulaires successives, la multiplication modulaire de Montgomery a été utilisé pour sa rapidité et sa simplicité d'implémentation.

Dans le but d'implémenter sur un circuit FPGA un crypto-système embarqué basé sur le protocole RSA, le chapitre suivant est consacré à la présentation des systèmes sur puce. En particulier les systèmes sur puce implémentés sur circuits FPGA SoPC (system on programmable chip).

Chapitre 3 :

Systèmes embarqués

3.1. Introduction

L'augmentation de la capacité d'intégration rend les systèmes électroniques de plus en plus complexes. Cette capacité d'intégration permet de concentrer des systèmes entiers sur une seule puce ce qui rend la conception et la vérification de ces systèmes fastidieuses et coûteuses. Par contre, les moyens et les outils de conception ne connaissent pas la même évolution comme le montre la Figure 3. **Erreur ! Signet non défini.**. La différence entre la capacité de conception et la capacité d'intégration ne cesse d'augmenter. Ces faits poussent les concepteurs à chercher des méthodes qui leurs permettent de concevoir et valider leurs systèmes dans des délais toujours plus court (respecter le Time To Market : TTM) [19].

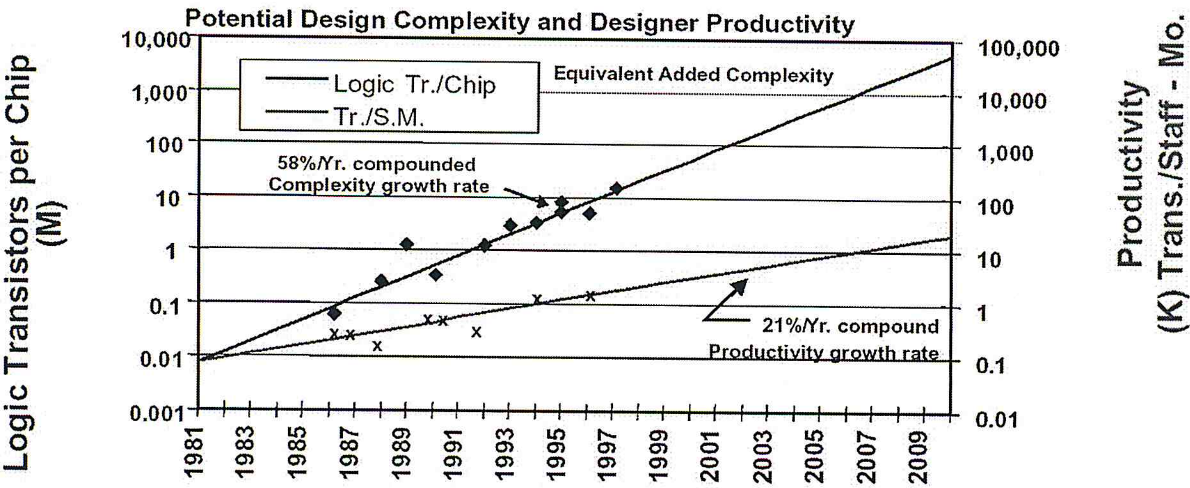


Figure 3. **Erreur ! Signet non défini.** : Divergence entre la productivité de la conception et la complexité des circuits

3.2. Système Embarqué

Un système embarqué est un système complexe qui intègre du logiciel et du matériel conçus ensemble afin de fournir des fonctionnalités données. Il contient généralement un ou plusieurs microprocesseurs destinés à exécuter un ensemble de programmes définis lors de la conception et stockés dans des mémoires.

Un système embarqué est autonome et ne possède pas des entrées/sorties standards tels qu'un clavier ou un écran d'ordinateur. Contrairement à un PC, l'interface IHM (Interface Homme machine) d'un système embarqué peut être aussi simple qu'une diode électroluminescente LED (Light Emitter Diode), ou des afficheurs à cristaux liquides LCD (Liquid Crystal Display) qui sont couramment utilisés.

La conception d'un système embarqué ne consiste pas seulement en la conception de la partie matérielle, mais aussi la gestion software de ce matériel via le processeur embarqué, on parle ainsi de système sur puce SoC (System on Chip). L'approche SoC consiste à cohabiter les deux ressources logicielle et matérielle sur une même puce : co-conception.

Les systèmes embarqués sont désormais utilisés dans des applications diverses tels que le transport (avionique, espace, automobile, ferroviaire), dans les appareils électriques et électroniques (appareils photo, télévision, téléphones portables…etc).

3.3. Système embarqué sur puce SoC

Le terme SoC (Système on Chip) ou système sur puce désigne une classe de circuits intégrés actuels. Un Système sur puce est un système embarqué sur une puce, pouvant intégrer sur un même substrat de silicium un ou plusieurs microprocesseurs, de la mémoire (statique, dynamique, flash, ROM, PROM, EPROM, EEPROM), des périphériques d'interface, ou tout autre composant nécessaire à la réalisation de la fonction attendue.

Les intérêts sont multiples (miniaturisation accrue, meilleures performances etc…) et ont donc largement contribué à pousser dans cette voie la majorité des fabricants de semi-conducteurs.

Pour la conception d'un SoC, il est primordial de sélectionner la plateforme matérielle d'implémentation. Cette sélection est basée sur la réponse à la question suivante : *Comment atteindre le fonctionnement correct d'un système avec un faible coût de réalisation et en respectant des contraintes supplémentaires ?*

En effet, pour les systèmes ne nécessitant pas de grandes capacités de traitement de données, les microcontrôleurs ou les microprocesseurs bas de gamme peuvent être un bon choix.

Si les besoins en calculs sont plus importants, les microprocesseurs plus puissants ou les DSPs peuvent être considérés. Ce type de solution est très flexible puisqu'il est basé principalement sur une écriture de programmes.

Pour obtenir des performances élevées, il est nécessaire de choisir des circuits spécifiques.

La première option, qui s'offre dans cette troisième classe, est d'utiliser des circuits programmable de type FPGAs. Ces derniers sont généralement recommandés pour le prototypage et la production en faible série. On parle alors de système SoPC (System on Programmable Chip).

Pour une large série de fabrication, la seconde option est de recourir aux circuits ASICs (Applicatrion-Specific Integrated Circuit) [20].

Les différents types de plateforme matérielle d'implémentation sont montrés sur la figure 3.1.

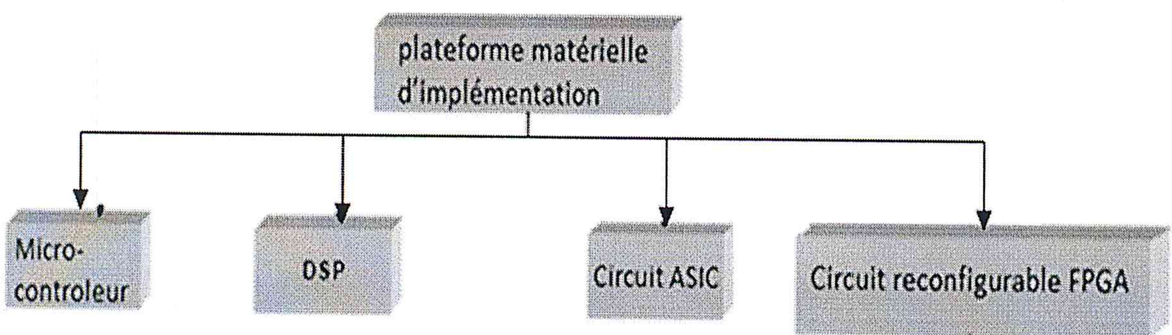


Figure 3.2. Plateforme d'implémentation

3.3.1. Architecture d'un système sur puce

Un système sur puce combine un ou plusieurs microprocesseurs pour le contrôle du système, un bus système, une mémoire interne, un ensemble de périphériques ou coprocesseurs dédiés à des tâches bien spécifiques et des interfaces qui assurent la communication avec l'extérieur du circuit [1].

L'architecture d'un SoC est montrée sur la figure 3.3.

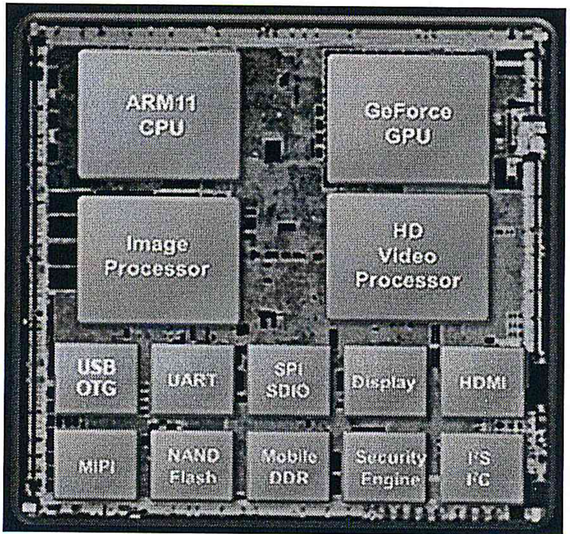


Figure 3.3. Architecture d'un système sur puce

Avec l'intégration d'une panoplie de composants sur une même puce, celle-ci est souvent considérée comme étant une architecture hétérogène. On peut référencer plusieurs types de composants qui constituent l'architecture d'un SOC suivant le degré de personnalisation possible.

En effet, les Soft-cores sont des IPs (Intellectuel Property) décrits dans un langage de description matériel synthétisable (VHDL, Verilog,…), très souples pour l'utilisation.

Les « Hard cores » sont des IPs optimisés en surface et en vitesse. Ces derniers sont implémentés directement sur silicium et font appel à des composants élémentaires d'une librairie générique. Ces composants sont placés et routés entre eux.

3.4. Système embarqué sur les circuits FPGA SoPC

L'implantation se fait sur une plate-forme reconfigurable. On utilise des composants configurables tels que les FPGA pour réaliser des systèmes sur des puces programmables (System on Programmable Chip) ou SoPC.

Le SoPC vient de SoC et signifie «un ensemble de blocs fonctionnels construit dans un composant programmable avec au moins un élément de traitement»

De nos jours, les composants FPGA haute densité, et les IP permettent l'intégration de systèmes complexes sur la même puce qui demande à être développées pour répondre aux besoins et aux nouvelles contraintes. Ces contraintes peuvent être intrinsèques, liées à l'architecture et à la composition du composant ainsi qu'extrinsèques liées à la technologie utilisée et à l'environnement de ce type de composant.

Il est évident que les FPGA sont aujourd'hui utilisés dans un large éventail d'applications. Ils ressemblent de plus en plus à des systèmes sur puce reconfigurables. D'ailleurs, intégrer un système embarqué dans un circuit FPGA est la nouvelle approche pour profiter des avantages offerts par le FPGA tels que la souplesse ou la simplicité d'intégration.

En effet, la plupart des systèmes sur puce intégrant désormais un microprocesseur. Ce n'est pas une coïncidence si l'une des dernières nouveautés en matière d'IP sur FPGA est le microprocesseur 32 bits embarqué.

Plusieurs raisons justifient l'intégration de fonctionnalités de traitement embarquées sur un FPGA, bien que certaines soient moins évidentes que d'autres.

En premier lieu, il n'y a aucune implémentation d'architecture fixe et donc aucune obligation que les fonctions soient réalisées par matériel plutôt que par logiciel. Il existe un vaste choix de solutions envisageables pour chaque application.

En deuxième lieu, une fois que la conception est adaptée de manière optimale sur un circuit FPGA, il n'y a aucune raison pour conserver cette conception éternellement sur le circuit. Pour de nombreuses applications, on peut utiliser une plate-forme de prototypage plus générale pour les premières étapes de définition, avec l'intention de la remplacer un jour par une plate-forme moins onéreuse, plus performante ou d'un facteur de forme différent.

3.4.1 Processeurs embarquées sur FPGAs

Lorsque l'on conçoit un système numérique complexe, on met en œuvre généralement un processeur embarqué [21].

Ce processeur embarqué est :

- Soit un bloc IP et on parle ainsi de processeur softcore.
- Soit il est implémenté sur silicium et on parle de processeur hardcore. Celui-ci est généralement plus performant que le processeur softcore.

Le choix d'un processeur pour le SoPC peut se faire sur différents critères :

- Processeur *hardcore* : pour ses performances au détriment de la flexibilité.
- Processeur *softcore* : pour sa flexibilité de mise à jour au détriment des performances moindres que le précédent.

- Généralement, on privilégie les processeurs softcore pour s'affranchir des problèmes d'obsolescence et pour pouvoir bénéficier facilement des évolutions apportées en refaisant une synthèse. La description d'un processeur softcore est souvent effectuée en utilisant un langage de description matérielle (VHDL, Verilog) et il est lié à un fondeur particulier de circuit FPGA (comme Altera ou Xilinx). De ce fait, son code source ne peut être implanté que dans les circuits FPGAs du fondeur approprié.

On trouvera principalement au niveau des processeurs softcore, le processeur NIOS et NIOS II d'Altera et le processeur Microblaze de Xilinx qui est utilisé dans ce travail.

A noter qu'un processeur hardcore Power PC d'IBM a été implémenté aussi par Xilinx sur silicium dans quelques familles de ses circuits FPGA. Les familles en question sont : Virtex II pro, Virtex-4 FX et Viretx-5 FX.

Tout-récemment, Xilinx intègra sur sa nouvelle génération de circuits FPGA, un processeur hardcore, en l'occurrence le processeur ARM.

Dans se qui suit, nous allons présenter un SoPC à base du processeur Microblaze.

3.4.2 Système embarqué à base du processeur Microblaze

La conception et la réalisation des systèmes embarqués sur puce nécessitent un cycle de développement assez long. Pour cette raison, il est recommandé de procéder en premier lieu au développement d'un prototype dans lequel les circuits FPGAs, pour leur reconfigurabilité sont souvent utilisés.

En effet, les fondeurs de ce type de circuits mettent généralement à la disposition des concepteurs, des cartes de prototypage à base de ces circuits. Ceci est dans le but de réaliser des implémentations en un cycle de temps relativement faible. La carte de prototypage utilisé dans ce travail est la carte V2MB1000 de Memec [22]. Cette dernière comporte un ensemble de composants qui permettent plus ou mois de développer et de vérifier le fonctionnement correct d'une application. Elle est équipée d'un circuit FPGA XC2V1000fg456-4 de la famille Virtex-II [23] qui permet d'implémenter toutes les ressources nécessaires au développement d'un SoPC. Le circuit en question est détaillé en annexe B.

A noter que ce circuit n'inclue pas dans sont architecture de processeur hardcore. A coté du circuit FPGA, une panoplie de composants est disponible sur la même carte, pour lui permettre de communiquer avec son environnement externe. On peut citer, entre autres, un port RS232, une mémoire DDR de taille 32 MB, deux afficheurs 7 segments, des GPIO (General Purpose Input Output),… etc. La carte V2 MB1000 est montrée sur la figure 3.4.

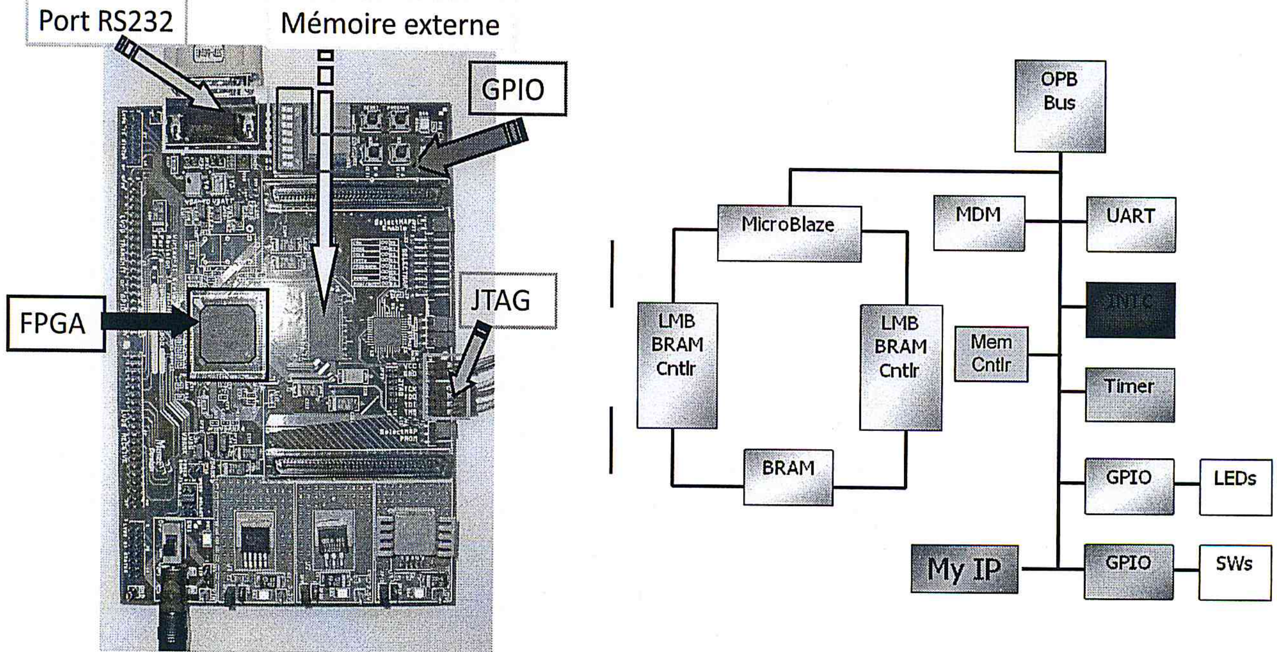


Figure 3.4. Carte de prototypage V2MB1000 et schéma synoptique d'un SoPC à base de Microblaze

L'implémentation du processeur Microblaze en tant que contrôleur principal du SoPC sur le circuit XC2V1000fg456-4, nécessite selon les besoins du concepteur, l'ajout sur le même circuit des périphériques qui jouent les rôles d'interfaces entre le processeur et les autres composants de la carte. Ses périphériques sont fournis par Xilinx sous forme d'IPs en langage VHDL. Ils seront connectés autour de Microblaze sur le bus système OPB, comme l'illustre

la figure 3.4. Parmi ces périphériques, on peut retrouver un UART (Universal asynchronous receiver/transmitter), un contrôleur de mémoire externe (mem Cntlr), un Timer, contrôleur d'interruption (INTC), un module pour le débogage (MDM),...etc. Il est à noter que d'autres périphériques sont nécessaires au fonctionnement du processeur et qu'il est indispensable de les inclure. Ces derniers sont une mémoire BRAM et ses contrôleurs, DCM (Digital Clock Manager).

3.4.2.1 Architecture du processeur Microblaze

Le processeur MicroBlaze de Xilinx est un processeur RISC de 32 bits [24]. Son code VHDL est fermé. Ses caractéristiques se résument comme suit :

- L'architecture du MicroBlaze est une architecture « Harvard » avec ses bus de d'instructions et bus de données séparés.
- Ses instructions sont de 32-bits.
- Il ne possède pas d'unité de gestion de mémoire (MMU : Memory Management Unit).
- Il possède 32 registres de 32-bits à usage général.
- Ses bus d'adresse sont également de 32-bits.
- Il possède un « pipeline » de 5 étages.

L'architecture du processeur microblaze est montrée sur la figure 3.5.

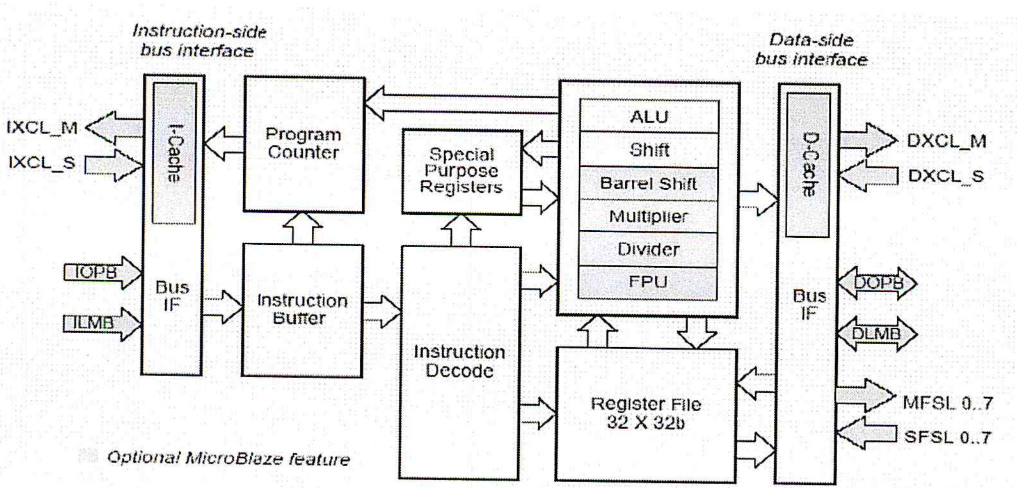


Figure 3.5. Architecture de Microblaze

Les parties en gris sont optionnelles et montrent à quel point ce processeur est configurable. Les mémoires Cache permettent au processeur de gagner en temps d'exécution en récupérant des données directement à l'intérieur sans aller les chercher dans la mémoire.

Le «Barrel Shift», le «Multiplier», le «Divider» et le «FPU: Floating Point Unit » permettent d'accélérer des traitements de données.

L'exécution d'une instruction à l'intérieur du pipeline se fait comme suit :

1. La lecture de l'instruction dans le buffer d'instruction.
2. Le décodage de l'instruction dans le décodeur d'instruction.
3. L'accès mémoire pour la lecture des opérandes.
4. Le calcul ou l'exécution de l'instruction dans l'ALU.
5. L'écriture du résultat dans la mémoire spéciale.

La fréquence du fonctionnement varie selon le circuit utilisé. Le tableau 3.1 illustre les performances de ce processeur sur quelques types de circuits.

FPGA	Fréquence	D-MIPS
Virtex-II	125Mhz	115
Virtex-II Pro	150Mhz	138
Virtex-4	180Mhz	166
Virtex-5	235Mhz	280

Tableau 3.1. Fréquence du processeur Microblaze sur quelques circuits

Généralement, l'interconnexion entre l'ensemble des composants qui constituent un SoPC est assurée par un système de bus bien déterminé. A cet effet, dans l'environnement ou il est implémenté, Microblaze possède quatre types de bus de communication.

3.4.2.2. On-Chip Peripheral Bus (OPB)

Le bus OPB, conçu par IBM pour ses microcontrôleurs PowerPC, permet de lier plusieurs maîtres à plusieurs esclaves. Il autorise un maximum de 16 maîtres et un nombre d'esclaves illimité selon les ressources disponibles. Xilinx conseille néanmoins un maximum de 16 esclaves. Comme ce bus est multi maîtres, il a donc une politique d'arbitrage paramétrable. Ce bus permet d'ajouter des périphériques au processeur MicroBlaze dont les besoins en communication seront faibles.

3.4.2.3. Local Memory Bus (LMB)

Le bus LMB est un bus synchrone utilisé principalement pour accéder aux blocs RAM inclus dans le circuit FPGA. Il utilise un minimum de signaux de contrôle et protocole simple pour s'assurer d'accéder à la mémoire rapidement.

3.4.2.4. Fast Simplex Link (FSL)

Le processeur MicroBlaze comporte 8 liens d'entrées/sorties FSL. Le bus FSL est un moyen rapide de communication entre le processeur et une autre entité. Chaque lien FSL est

unidirectionnel (simplex) et met en œuvre une FIFO (pour stocker les données) et des signaux de contrôle (FULL, EMPTY, WRITE, READ,...). Il met aussi à la disposition du développeur plusieurs fonctions intéressantes dont les plus utilisées sont : *"microblaze_bwrite_datafsl"* et *"microblaze_bread_datafsl"*. Ces deux fonctions permettent d'échanger des données entre différents microblazes, par exemple, en utilisant la FIFO déjà intégrée dans le bus FSL. Ces deux fonctions sont bloquantes; *bwrite* se bloque lorsque la FIFO du bus FSL est saturée et *bread* se bloque lorsque la FIFO est vide. Les communications sur les liens FSL se font très simplement grâce à des instructions prédéfinies. Elles peuvent atteindre les 300 Mo/s à 150 Mhz.

3.4.2.5. Xilinx Cache Link (XCL)

Le lien XCL est un lien FSL particulier, dédié à la connexion d'un contrôleur mémoire externe avec la mémoire cache interne. Ceci permet au contrôleur de cache, de ne pas être ralenti par la latence du bus OPB.

3.4.2.6. La mémoire BRAM

Les blocs RAM sont des composants configurables de taille limitée par la capacité du circuit FPGA. Celle-ci peut être de 8 Kbits, 16 Kbits, 32 Kbits, ou 64 Kbits. Le fonctionnement des BRAMs est similaire aux RAMs existant dans un ordinateur. Ils servent à stocker les codes des programmes à exécuter de façon organisée. Les données et les instructions, dans le processeur Microblaze, sont stockées séparément dans deux blocks mémoires. La BRAM sert aussi à stocker le noyau (Kernel) du système d'exploitation. Elle peut être configurée à partir des Blocs select RAM de 18 Kbits [5] qui se trouvent dans le circuit FPGA. Comme elle peut être implémentée sur CLB (Configurable Logic Block).

3.4.2.7. Les Périphériques du processeur Microblaze

De nombreux périphériques sont fournis avec MicroBlaze, afin de constituer un système complet et personnalisable. Il y a, entre autres :

- Contrôleur mémoires (SRAM, Flash)
- UART (Universal asynchronous receiver/transmitter)
- Timer/compteur
- Interface SPI
- Contrôleur d'interruptions
- GPIO (entrées-sorties génériques)

- Convertisseurs A/N et N/A
- DMA (Direct Memory Access)

De plus, des périphériques payants sont proposés en version d'évaluation.

- UART 16550
- Interface Ethernet
- Interface I²C
- Interface PCI

3.5. Les critères de performance des SoPC

Un des aspects fondamentaux d'un système SoPC est qu'il fonctionne correctement suivant les performances fixées lors des spécifications.

Les performances d'un SoPC peuvent être analysées suivant plusieurs critères, à savoir les performances temporelles, la consommation, la flexibilité et le coût [1].

3.5.1. Les performances temporelles

Ce premier critère concerne tous les « aspects temporels » d'un système. Ceux-ci peuvent être capturés de plusieurs manières:

- Le temps d'exécution qui est le premier élément que l'on observe pour évaluer la performance d'un système. On peut écrire le temps d'exécution comme une fonction de la latence et de la cadence. La latence L correspond au temps mis par le système entre l'acquisition d'une entrée et la production de la première sortie. Ensuite la cadence C caractérise le rythme auquel le système produit chacune de ses sorties (le nombre d'échantillons produits par seconde).
- Le débit consiste à observer le nombre de tâches traitées durant une période de temps (à la seconde par exemple). Le débit pour chaque système sera calculé pour un nombre de tâches donné sur le temps d'exécution total pour ces tâches, ce qui revient donc à comparer le temps d'exécution total de ces tâches.

Dans les systèmes temps réel, les traitements pris en charge sont contraints par des échéances de traitement. Le contrôle de processus industriels ou encore les systèmes de sécurité sont des applications des systèmes temps réels et font partie des nombreuses cibles potentielles des processeurs enfouis.

Pour respecter les contraintes de temps réel, il est nécessaire qu'une tâche soit traitée dans un temps inférieur ou égal à celui imposé par l'échéancier. Il est souvent nécessaire, dans ce

cas là, de diminuer la latence et/ou d'augmenter la cadence de traitement du système. Ces paramètres dépendent souvent de la manière dont l'architecture exploite le parallélisme.

3.5.2. La consommation

Le terme consommation regroupe les dissipations de puissance et d'énergie. Celle-ci est liée à la surface occupée par l'application sur le circuit et la puissance consommée au cours d'exécution.

L'analyse de la consommation de puissance permet d'anticiper sur les dissipations thermiques, pour concevoir les circuits de refroidissement (surtout pour les applications embarquées où le système de refroidissement requiert une surface, une masse et un coût non négligeables). Le coût des systèmes de refroidissement, leur taille ainsi que leur poids constituent autant de handicaps pour des applications portables.

3.5.3. La flexibilité

On entend par flexibilité la possibilité d'évolution ou d'adaptation d'un système. Il n'est pas toujours possible de prévoir toutes les applications qu'un système devra prendre en charge. Un critère indirect de performance est donc la capacité d'un système à pouvoir s'adapter à diverses applications.

En effet, cette flexibilité peut permettre de prolonger la durée de vie d'un système en le faisant évoluer vers de nouvelles applications, nouveaux standards ou nouvelles normes qui apparaissent sur le marché.

3.5.4. Les coûts

Les coûts sont aussi, indirectement, des facteurs de performance. Il est évident qu'un système aussi rapide qu'un autre mais moins onéreux sera considéré comme une solution plus attractive : nous pouvons alors considérer qu'il est indirectement plus performant.

Le coût est en fonction de nombreux éléments et il est donc particulièrement difficile de se faire une idée réaliste du coût réel du circuit final : la surface de silicium, le rendement de fabrication, le packaging, l'effort de conception, le coût de la programmation, le temps de mise sur le marché, la réutilisation et le test sont autant de facteurs influençant le coût final du produit.

3.6. Flot de Conception d'un SoPC

Le cahier des charges d'un système spécifie les fonctions des applications ciblées et les performances souhaitées. Pour faciliter la conception de ces systèmes, plusieurs logiciels permettent la description et la simulation des circuits à différents niveaux d'abstraction comme l'outil EDK de Xilinx.

De plus, la complexité accrue des systèmes a naturellement tendance à augmenter le temps de conception et donc l'automatisation de certaines phases permet de limiter le "time to marker".

Historiquement, les premiers outils remontent aux années 1980. Jusque là, les masques des circuits étaient dessinés « à la main ». Par un principe basé sur la réutilisation d'éléments matériels, au début des blocs de portes simples, puis des blocs complexes pré caractérisés jusqu'aux cœurs des composants actuels, la complexité des systèmes n'a jamais cessé de croître.

Actuellement, le point crucial du flot de conception de ces systèmes est le choix de l'architecture de traitement, car c'est elle qui va largement fixer le compromis de performances.

On peut représenter le flot de conception des systèmes sur puce par la figure 3.6.

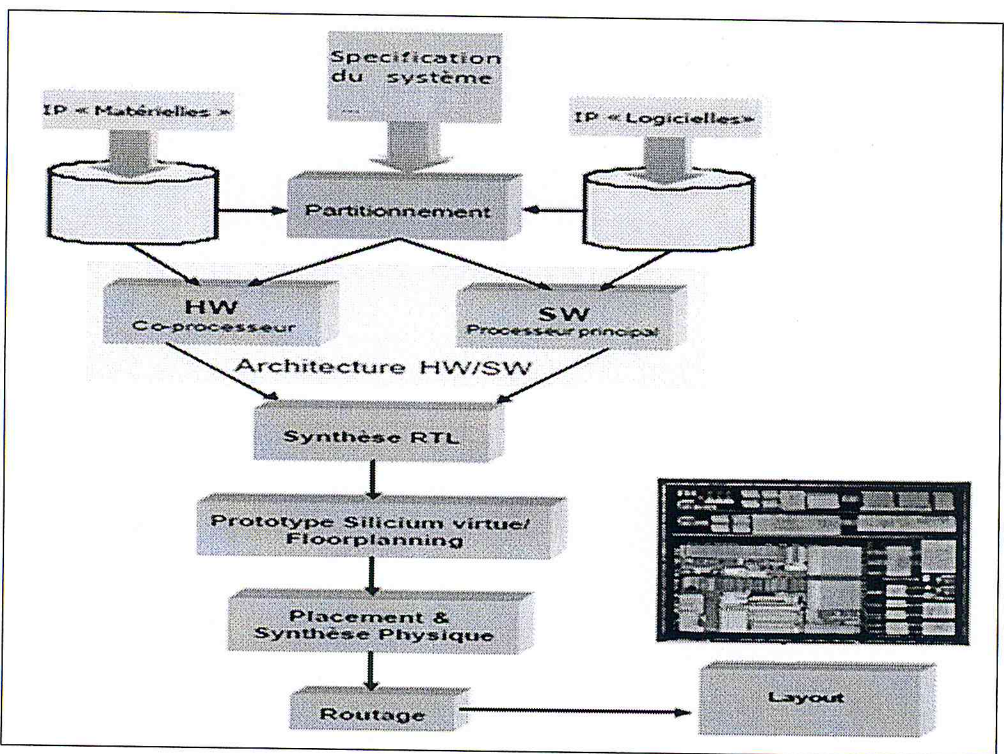


Figure 3.6. Flot de conception d'un Système sur Puce.

D'une manière générale, le flot de conception d'un système sur puce nécessite le développement de deux ressources à savoir, une ressource logicielle et une autre matérielle. La partie logicielle est utilisée pour rendre les applications à implémenter plus flexibles. La partie matérielle est développée pour accélérer les chemins sensibles à la contrainte temps réel des architectures associées aux applications.

Le partitionnement d'une application sur les deux ressources s'effectue généralement en tenant compte des performances à atteindre, c'est-à-dire la surface occupée, le temps d'exécution, la puissance consommée et le coût de développement.

En effet, la migration des tâches vers le matériel (HW) est toujours possible jusqu'à ce que les contraintes de performances soit atteintes, comme elles peuvent être basculées vers le logiciel (SW) [1].

La modélisation de la partie logicielle est réalisée par l'utilisation d'un langage de description machine en l'occurrence, le langage C, ou C++ ou encore l'assembleur et sera exécuté par un microprocesseur/microcontrôleur.

Pour la partie matérielle, celle-ci peut être décrite par un langage de description matérielle (HDL, VHDL,...).

Une fois l'étape de spécification de l'architecture du système est achevée, une synthèse logique est exécutée sur cette dernière. Cette étape n'est rien d'autre qu'une transformation de l'architecture en un ensemble d'équations booléennes. On parle ainsi du niveau Transfert Registre Level ou RTL.

Le résultat obtenu de la synthèse sous forme de net-list est optimisé et transformé en une autre net-list de blocs logiques qui sera placée/routée sur la plateforme utilisée.

3.7. Modèles d'implémentations des SoPC

Chaque famille d'architecture est basée sur un modèle de traitement. Un modèle (SW) est basé sur une exécution séquentielle d'un algorithme par un processeur. Un modèle (HW), pour les circuits dédiés, offre l'option d'exécuter des tâches non parallélisables par un processeur. Un troisième modèle est une combinaison entre les deux modèles (SW) et (HW) qui est le modèle dit Co-Conception.

3.7.1. Implémentation logicielle (SW)

Dans ce cas, le modèle de traitement est séquentiel. Les architectures basées sur ce modèle utilisent un ou plusieurs processeurs embarqués qui exécutent un ou plusieurs programmes définissant les opérations à réaliser et les données à récupérer de la mémoire.

En général, cette implémentation n'utilise qu'un faible nombre de ressources de calcul et des registres qui sont réutilisées dans le temps. Ces architectures sont généralement très flexibles, mais ne permettent pas d'atteindre des performances élevées.

La figure 3.7 illustre l'exécution des opérations de l'approche software.

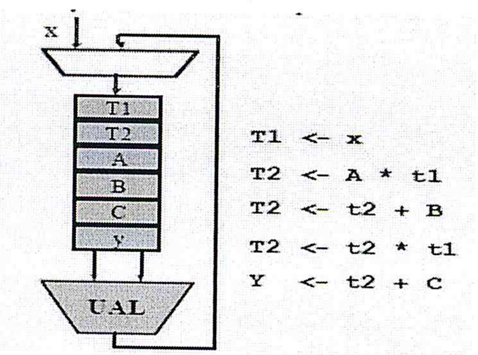


Figure 3.7- Exécution séquentielle

3.7.2. Implémentation matérielle (HW)

Les circuits FPGA sont basés sur l'approche hardware. Dans ce cas, afin d'exploiter le parallélisme de l'application, chaque opérateur traite des opérandes directement acheminés sur ses entrées.

Dans certaines plateformes d'implémentation matérielle, la diversité des IPs fournis dans une bibliothèque permet d'implémenter des fonctions complexes, avec des performances élevées. Néanmoins, la flexibilité reste faible.

La figure 3.8 montre l'exécution des opérations dans l'approche hardware.

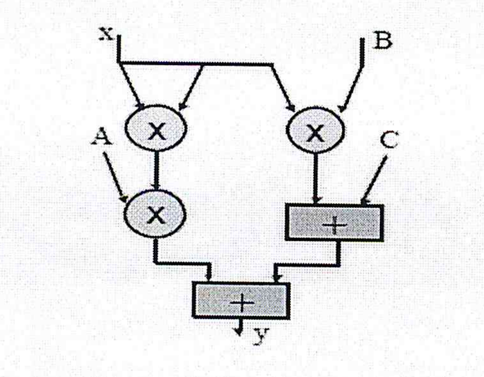


Figure 3.8. Exécution parallèle

3.7.3. Implémentation Co-conception (SW/HW)

Les architectures reconfigurables sont également basées sur cette approche. Afin d'apporter une souplesse supérieure aux circuits dédiés, l'utilisation de cette approche nous permet d'exploiter les avantages respectifs des matériels et logiciels.

Les caractéristiques de ce modèle de traitement confèrent une flexibilité certaine au matériel qui peut s'adapter à n'importe quelle application.
Pour mieux illustrer ce concept de développement, un exemple d'une architecture simple composée d'un microprocesseur et d'un coprocesseur est montré sur la figure3.9.

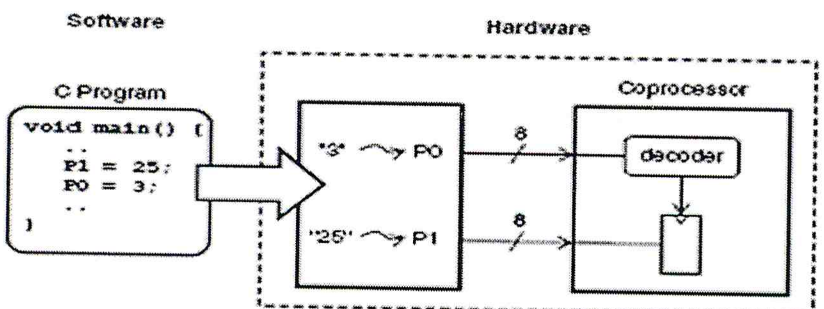


Figure 3.9. Exemple d'implémentation en co-conception

La communication entre les deux composants est établie grâce aux deux ports P0 et P1. Ces derniers permettent d'accéder au coprocesseur de l'extérieur en utilisant un programme décrit en langage machine.

L'architecture interne du coprocesseur est constituée d'un décodeur et d'un registre. Leurs tâches consistent respectivement à contrôler le registre interne et la réception de la donnée fournie de l'extérieur. Ainsi, quand celle-ci apparait sur le port P1, elle ne sera transmise au registre que si le décodeur reconnait le signal qui se présente sur P0.

3.7.4 Conception du système et cycle de développement

L'implémentation des applications sur des plateformes SoPC offre plusieurs avantages. On peut citer entre autres : l'ajout de son propre IP (IP personnalisé) au système embarqué. Cet IP sera implémenté sur du matériel et peut en effet être considéré comme un coprocesseur par rapport au processeur principal. D'une manière générale, le cycle de conception d'un SoPC sur FPGA à base du processeur Microblaze comporte trois étapes :

- Configuration de Microblaze et des périphériques.
- Ajout du périphérique personnalisé.
- Programmation de la partie logicielle et chargement du système sur le circuit FPGA.

A noter que la seconde étape est nécessaire uniquement si l'architecture de l'application ciblée, comporte un IP dédié à une fonction particulière de l'application.

Afin de créer des applications embarquées, Xilinx a développé un ensemble d'outils regroupés dans deux logiciels ISE (Integrated Software Environement) et EDK (Embedded Development Kit). Les outils d'ISE sont utilisés par EDK pour la synthèse et l'implémentation sur le circuit FPGA. L'interface graphique d'EDK est nommée XPS (Xilinx Plat forme Studio). Elle inclut aussi les outils GNU de programmation personnalisé par Xilinx pour Microblaze et le processeur hardcore Power PC d'IBM. Les outils en question sont : GCC pour la compilation des codes écrits en langage C/C++ et GDB pour le débogage [25].

EDK est constitué aussi d'un outil nommé SDK (Software Development Kit), dédié au développement des parties logicielles du système à implémenter. Le flot de conception des systèmes embarqués sur les FPGA de Xilinx est résumé sur la figure 3.10.

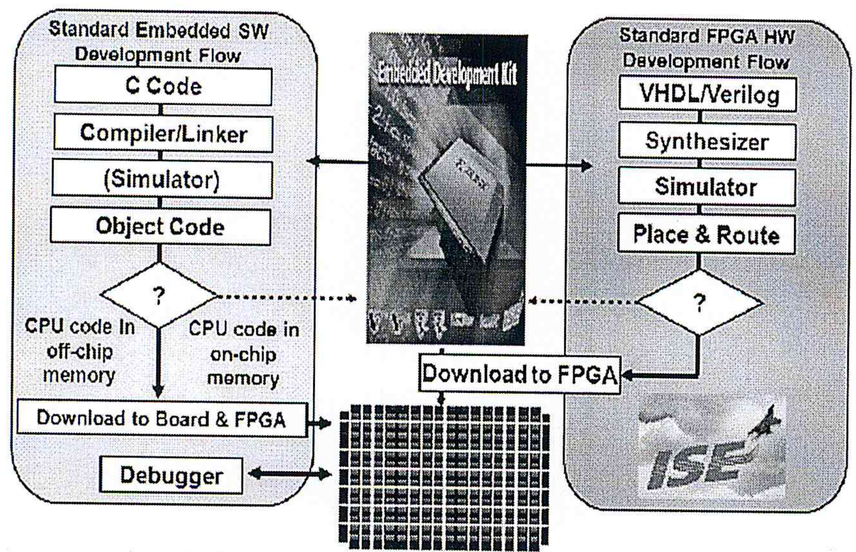


Figure.3.10- cycle de conception de Xilinx.

Dans ce qui suit, nous allons détailler les étapes citées ci-dessus pour la conception et l'implémentation d'un SoPC sur FPGA en utilisant le processeur Microblaze.

3.7.4.1. Configuration de Microblaze et des périphériques

Dans cette première étape, on procède en premier lieu dans XPS par :

- Choisir Microblaze en tant que processeur principal.
- Définition de la fréquence système. Sur la carte V2MB1000, il existe deux quartzs qui permettent de fournir deux fréquences, à savoir 100 Mhz et 24 Mhz.
- La polarisation du signal d'initialisation système (actif à l'état haut ou bas).
- Sélection des périphériques de base, tels que, l'UART, un module pour déboguer (en software, ou en hardware, ou pas), ...etc.

- Sélection de la taille de la mémoire interne BRAM (64 kbits, 32 kbits, 16 kbits ou 8 kbits).
- Sélection ou pas de l'unité arithmétique à virgule flottante et de la mémoire cache.

Une fois avoir achevé cette configuration, il sera question ensuite de sélectionner les composants de la carte à inclure dans l'application, tels que le port RS232, les leds, la mémoire externe …etc. A ce niveau de configuration, d'autres périphériques peuvent être ajoutés, en outre, un timer ou/et un contrôleur de mémoire. Bien que ces derniers, peuvent aussi être ajoutés lors de la conception.

A la fin de cette première étape, XPS génère trois fichiers principaux, en l'occurrence, system.MHS (Micropocessor Hardware Specification), system.MSS (Micropocessor Software Specification) et system.ucf. Le premier fichier définit l'architecture matérielle de la plate forme, les connexions et les adresses mémoire de chaque périphérique. Le second fichier comporte les noms des pilotes (drivers) associés à chaque périphérique et leur version. Le dernier fichier porte les informations concernant l'emplacement des signaux d'entrées/sorties sur les pins du circuit FPGA.

3.7.4.2. L'ajout d'un IP personnalisé

Dans cette seconde étape du cycle de conception, il est question de mettre en œuvre sur la plateforme SoPC son propre IP, une fois que celui-ci a été conçu et vérifié dans ISE. Généralement, le transfert des données entre les deux parties (processeur-IP) nécessite un système de communication qui assez complexe. Ce dernier est nommé *interface hardware/software* [livre] et qui a pour objectif, de réaliser la communication entre l'IP et le software exécuté par le processeur. En effet, l'implémentation d'un IP autour d'un processeur peut se faire en utilisant l'une des deux métodes suivantes :

- Une implémentation en coprocesseur (coprocesseur interface).
- Une implémentation via la mémoire et le bus système (memory mapped interface).

Xilinx offre sur Microblaze une interface FSL (Fast Link Simplex) qui sont des ports de communication point-à-point [1]. De ce fait, la liaison avec l'IP sera réalisée sans passer par le bus système, comme le montre la figure 3.11.

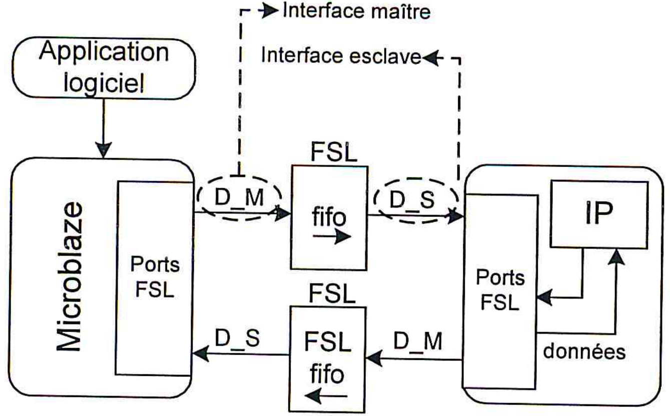


Figure 3.11.Connexion d'un IP par l'utilisation d'une liaison FSL

Dans la seconde approche d'implémentation, un espace mémoire est alloué à l'IP, pour permettre la communication entre ce dernier et le processeur. L'identification de l'IP par le processeur est effectuée par la déclaration de ses adressages dans la partie logicielle.

En général, ce type de communication est le plus répandu, bien qu'il soit considéré plus lent et plus complexe. Cette complexité relève principalement des intervenants mis en exécution. A savoir la mémoire et le bus système, comme ceci est montré sur la figure 3.12. En plus du bus système, cette configuration nécessite l'utilisation d'une interface entre ce dernier et l'IP. Cette interface nommée IPIF (IP Interface) a pour objectif de gérer le transfert des données entre l'IP, le bus, le décoder et le protocole de communication du bus [26,27].

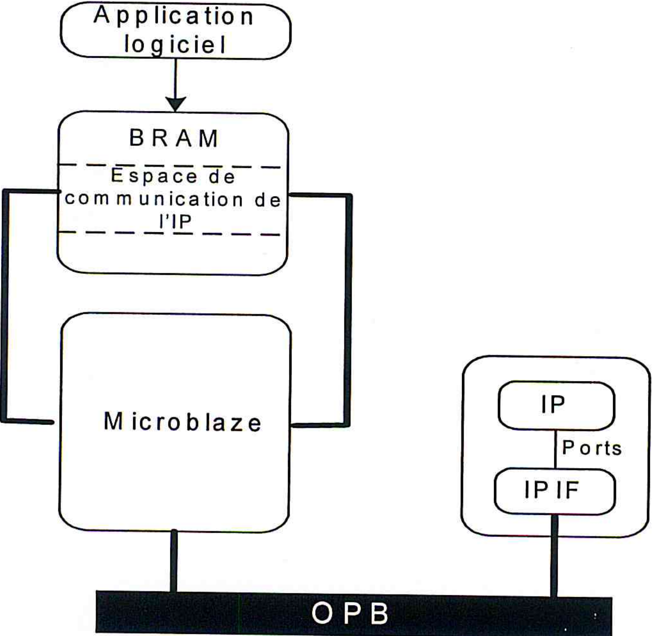


Figure 3.12- Ajout d'un IP personnalisé au bus système

3.7.4.3 Développement de la partie logicielle et chargement du système sur FPGA.

Cette étape est réalisée dans l'outil SDK. Elle consiste à décrire la partie logicielle de l'application. Les points essentiels à développer dans cette troisième étape sont :

- Réalisation d'une API (Application Programming Interface) qui est une interface de programmation de l'application.
- Compilation de l'API.
- Chargement du ficher de configuration (bitstream) sur le circuit FPGA et visualisation des résultats sur un HyperTerminal.
- Si nécessaire, vérification du fonctionnement du circuit.

3.7.4.3.1 Interface de programmation de l'application.

L'API joue un rôle considérable de la partie SW. Celle-ci constitue une passerelle entre l'OS et les différents périphériques du processeur. Elle est composée principalement par les pilotes des différents IPs intégrés dans le système.
Sa programmation peut être effectuée en langage C ou C++, où il sera question de spécifier les fonctions qui permettent d'accéder aux périphériques. A titre d'exemple, la communication entre l'UART et l'IP personnalisé est effectuée en déclarant des variables intermédiaires entre les fonctions de l'UART et celles de l'IP.

3.7.4.3.2 Compilation de l'API

Cette étape consiste à convertir le programme C/C++ de l'application en un langage très proche du langage du processeur. Cette conversion n'est rien d'autre qu'une compilation de l'API. Le flot de compilation est montré sur la figure 3.13.

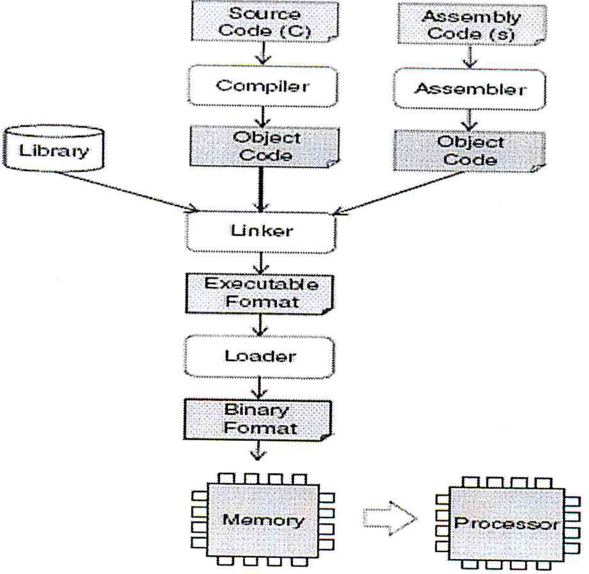


Figure 3.13- Compilation de la partie logicielle

L'outil utilisé dans cette étape est l'outil GCC de EDK. A travers ce processus, et avant que le code C/C++ ne soit implémenté sur le circuit FPGA, le fichier intermédiaire qui constitue le code machine de l'application est la représentation en binaire des instructions et des données utilisées. Ce code machine sera ensuite organisé pour sa mise en mémoire par le Linker. L'organisation en question est effectuée selon une disposition bien ordonnée pour les instructions et les données. Le linker permet ainsi de générer ces informations dans un exécutable de l'API (executable.elf). Finalement, le loader décode les informations portées par l'exécutable et assure son chargement dans la mémoire. A noter que cet outil prend en charge aussi la configuration du circuit FPGA, par le chargement de son bitstream à travers le port parallèle de l'ordinateur qui est lié au port JTAG de la carte de prototypage. Généralement, la configuration du circuit FPGA par son bitstream se déroule avant l'initialisation de la mémoire par l'exécutable de l'API [1].

3.7.5 Les systèmes d'exploitation compatibles avec Microblaze

Dans le but d'assurer la gestion des périphériques, plusieurs types de système d'exploitation OS (Operating System) sont compatibles avec le processeur Microblaze. Ces systèmes peuvent gérer les accès mémoire, la communication via le port RS232, etc.

Dans ce qui suit, quelques systèmes d'exploitation pouvant être embarqués avec le processeur Microblaze seront présentés tels que : Xilkernel, uClinux et Asterix.

Chaque système a ses propres caractéristiques, en termes de vitesse d'exécution, de la taille mémoire occupée et des fonctionnalités offertes [28].

3.7.5.1 uClinux

uClinux (prononcé "you-see-linux") est un type de Linux standard, destiné aux microprocesseurs qui n'ont pas une unité de gestion mémoire (MMU).

L'unité de gestion mémoire permet la traduction des adresses logiques en adresses physiques. Elle décide si la donnée est dans la RAM ou dans le disque dur et empêche le processeur à accéder directement à la mémoire.

Sans présence de cette unité, le programmeur est sensé faire la gestion de mémoire pour qu'il garde la cohérence entre les différents processus en utilisant vfork( ) qui est similaire au fok() de Linux.

uClinux est compatible avec plusieurs types de processeurs, Motorola Coldfire, Dragonball, Blackfin, ARM7TDMI et MicroBlaze.

uClinux contient deux package : le premier est le noyau (Kernel) du système et le second package contient toutes les bibliothèques et les libérés en C qui permettent la gestion des périphériques.

3.7.5.2 Asterix

Astérix est un petit noyau temps réel développé par Mälardalen Real-Time Research Center (MRTC), situé à l'Université de Mälardalen (Suède) où Il est enseigné et utilisé dans différents projets de recherche.

Astérix contient un environnement de débogage et fournit le soutien à une mesure à haute résolution de temps d'exécution des tâches et permet aussi l'exécution multitâches.

3.7.5.3 Xilkernel

Xilkernel est un noyau de petite taille, robuste et compatible avec les processeurs Microblaze et Power PC. L'OS en question est intégré dans EDK.

Xilkernel permet à l'utilisateur par exemple d'engager des gestionnaires d'interruption et peut être utilisé pour implémenter des applications à un niveau d'abstraction élevé, telle que les applications qui relèvent de la video, audio et réseaux.

3.7.5.4 Standalone

Cet OS est un système d'exploitation basic. Il est livré par défaut sur EDK et comporte les librairies d'entrées/sorties pour les pilotages des IPs, tel que l'UART, BRAM, Timer… etc. Dans ce travail, nous nous sommes limités à l'utilisation de cet OS.

3.8 Conclusion

Dans ce chapitre, nous avons introduit la notion de système embarqué et plus précisément système sur puce ou SoC puis SoPC et la relation qui lie les deux types de produits.

Avec les SoPC, de nouveaux défis de conception sont apparus. Ceci ne veut pas dire qu'il y a eu une rupture entre les deux types de systèmes. La méthodologie de conception propre au SoC reste applicable au SoPC de part sa technologie.

D'un point de vue matériel, la réalisation d'un système sur puce, aujourd'hui, est plus accessible grâce au FPGA de plus en plus performant alliés aux blocs IPs réutilisables

Le chapitre suivant sera consacré à l'objectif principal de ce travail, à savoir l'implémentation d'un crypto système embarqué sur circuit FPGA.

Chapitre 4 :

Implémentation de la plateforme de chiffrement/déchiffrement

4.1 Introduction

Notre travail consiste à embarquer sur la carte V2MB1000 de Memec un crypto-système à clé publique en l'occurrence le RSA pour une taille de clé de n bits.

L'objectif principal est de développer une plateforme de chiffrement/déchiffrement qui sera gérée par une application Java exécutée sur un PC.

Le crypto-système embarqué doit répondre à la contrainte temps réel. Pour ce faire, un des objectifs de ce travail est d'étudier l'influence de la base de représentation des données sur les performances d'exécution de l'exponentiation modulaire.

Les approches considérées sont:

1. Une implémentation purement logicielle, où les algorithmes d'exponentiation modulaire et de multiplication modulaire MMM sont exécutés par le processeur Microblaze.
2. Une implémentation basée sur l'utilisation d'un IP (Intellectuel Property) matériel pour l'exécution de la multiplication modulaire. Celui-ci sera utilisé pour le calcul de l'exponentiation modulaire. La gestion de ce calcul est faite par le processeur hôte ; le Microblaze.

Les performances de l'exécution de la multiplication modulaire de Montgomery dépendent de la base utilisée. A cet effet, dans l'approche logicielle, nous avons testé trois bases, à savoir la base 2, la base 2^{16} et la base 2^{32}, c.-à-d. le multiplieur est partitionné respectivement en bits (base 2), en mots de 16 bits (base 2^{16}) et en mots de 32 bits (base 2^{32}).

Il est à noter que toutes les implémentations de l'exponentiation modulaire sont basées sur la méthode binaire LSB.

Pour tester ces implémentations, nous avons dû connecter notre carte V2MB1000 au PC via le port RS232

Dans un premier temps, nous avons utilisé l'*HyperTerminal* de Windows pour visualiser sur écran les résultats du chiffrement/déchiffrement réalisé dans la carte.

Il est à noter que l'utilisation de l'HyperTerminal ne permet pas le dialogue avec la carte. Il est seulement utilisé pour la vérification du fonctionnement de l'architecture implémentée sur le circuit FPGA où les données d'entrées sont figées dans la BRAM lors du téléchargement du BitStream de l'architecture (configuration du FPGA).

Autrement dit, cet outil est limité juste à la transmission des données. A cet effet, nous avons développé une application Java avec une IHM pour rendre l'échange de l'information entre le PC et la carte plus flexible.

4.2 Description de la Plateforme de chiffrement/déchiffrement

Le schéma synoptique de la plateforme de chiffrement/déchiffrement réalisée dans ce travail est montré sur la figure 4.1.

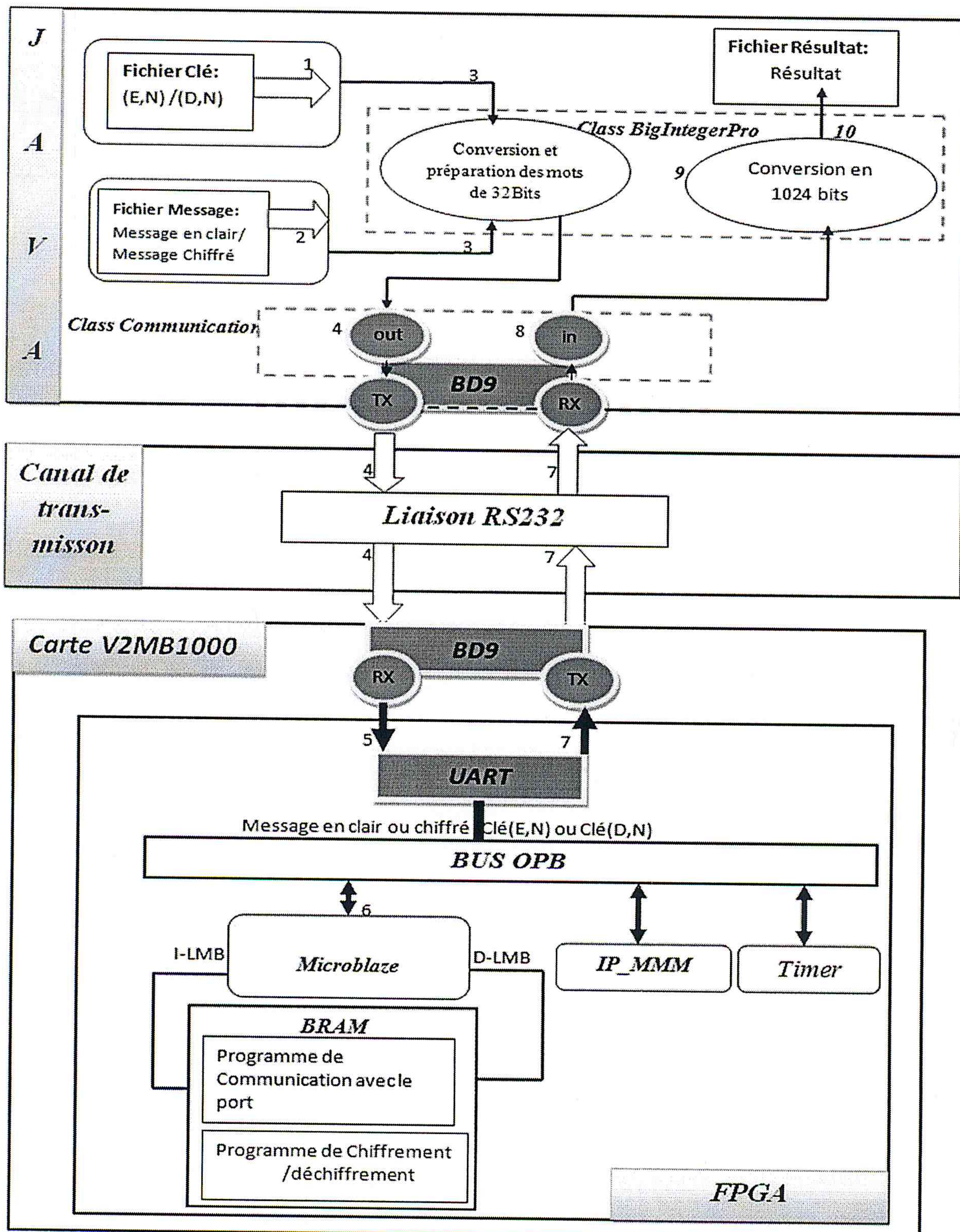


Figure.4.1- Plateforme de chiffrement/déchiffrement

La plateforme implémentée est composée de :

- La carte V2MB1000 qui est dédiée au calcul du chiffrement/déchiffrement qui est entamé dès que toutes les données sont entièrement transmises à partir du PC. De même, une fois que les opérations de chiffrement/déchiffrement sont achevées, les résultats sont transmis au PC.
- L'application Java dont le rôle est de transmettre et de recevoir les données vers ou de la carte V2MB1000.
- Le canal de transmission qui assure la liaison RS232 entre l'application Java et la carte à base FPGA.

Les opérations de chiffrement/déchiffrement par l'utilisation de cette plateforme sont définies comme suit:

1. Lire la clé à partir d'un fichier texte (E, N) où (D, N)
2. Lire le message en clair ou le cryptogramme à partir d'un fichier
3. Subdiviser la clé et le message en mots de 32 bits.
4. Transmission de l'ensemble de données via la liaison RS232 vers la carte V2MB1000.
5. Calcul de l'exponentiation modulaire dans le circuit FPGA de la carte V2MB1000, une fois que les résultats du chiffrement/déchiffrement sont obtenus, ceux-ci sont restitués par l'application Java via la liaison RS232.
6. Conversion du résultat dans l'application Java en décimal.
7. Sauvegarder le résultat dans un fichier.

4.3 Implémentation de l'exponentiation modulaire sur FPGA

L'implémentation de l'exponentiation modulaire est effectuée en deux étapes:

1. Réalisation par l'outil EDK des tâches suivantes : configuration (Microblaze, taille de BRAM,…), synthèse et génération du fichier de configuration du circuit FPGA.
2. Développement des pilotes et des programmes de calcul de l'exponentiation modulaire et de la multiplication modulaire dans l'environnement SDK.

Le chiffrement/déchiffrement par le protocole RSA est une opération qui fait intervenir un calcul modulaire complexe qui n'est pas supporté par les processeurs classiques. D'où l'implémentation de cette opération est réalisée d'une manière conjointe matérielle et logicielle dont les détails sont les suivants.

4.3.1 Description matérielle

Dans ce travail, nous avons utilisé trois architectures, les deux premières sont dédiées à la première approche d'implémentation (purement logicielle). Ces dernières sont basées sur les mêmes composants (Microblaze, BRAM, UART, Timer, bus OPB et LMB). La différence réside dans les options du processeur. Les options en question concernent l'utilisation du processeur avec ou sans un multiplieur 32 bits (mul32) et un Barrel shifter. L'objectif de ce choix est d'étudier l'influence de ces options sur les performances temporelles de l'exécution du chiffrement/déchiffrement par notre application embarqué dans le FPGA.

La troisième architecture est dédiée à la deuxième approche d'implémentation celle-ci consiste à adjoindre un IP matériel pour l'exécution de la multiplication modulaire de Montgomery. Cette architecture est constituée de : Microblaze, BRAM, UART, IP_MMM, Timer, bus OPB et LMB.

Les architectures correspondantes aux deux approches d'implémentations (logicielle matérielle) sont montrées respectivement sur les figures 4.2 et 4.3.

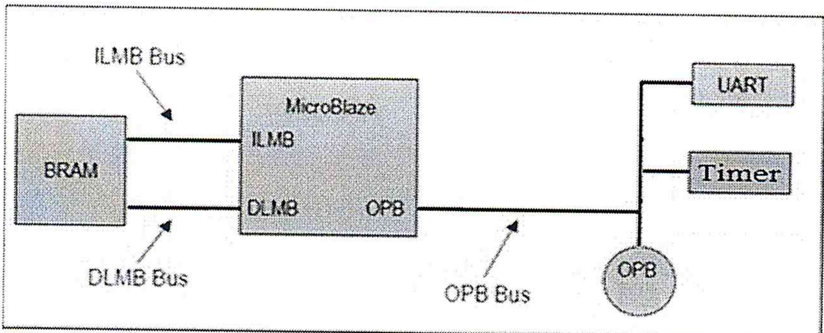


Figure 4.2- Architecture de l'approche logicielle

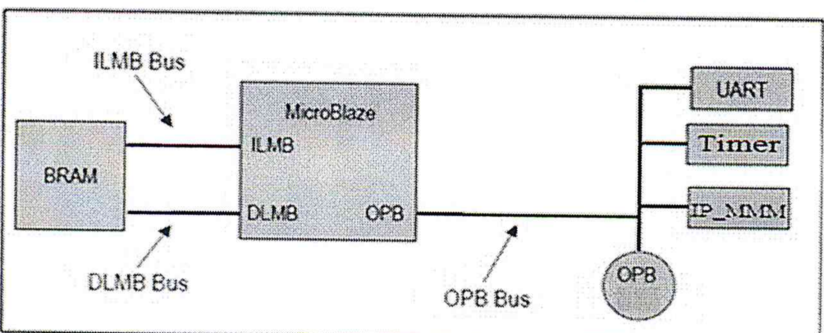


Figure 4.3- Architecture de l'approche matérielle

Ces architectures sont constituées des composants suivants:

- Processeur softcore Microblaze dont le rôle est l'exécution des programmes associés aux calculs de l'exponentiation et de la MMM.
- UART.

- Mémoire Bram
- Bus OPB et LMB.
- IP_MMM pour la multiplication modulaire.
- Timer qui compte le nombre de tops d'horloges, nécessaires à l'exécution. Celui-ci nous permettra de mesurer les performances de chacune des configurations implémentées.

  Le nombre en question est représenté par :

$$nbr_top_horloge_Total = nbr_top_horloge_Transmission + nbr_top_horloge_Execution + nbr_top_horloge_Réception$$

nbr_top_horloge_Transmission, *nbr_top_horloge_Execution* et *nbr_top_horloge_Reception* représentent respectivement le nombre de tops d'horloges nécessaires pour la transmission des données, le chiffrement/déchiffrement et la réception des résultats.

4.3.2 Description logicielle

La partie logicielle est constituée de programmes écrits en C et exécutés dans la carte V2MB1000 par le processeur Microblaze. Ces programmes consistent à faire le calcul de l'exponentiation modulaire requise par le chiffrement ou le déchiffrement.

Les fonctions associées au calcul de l'exponentiation modulaire qui est en fait un ensemble de multiplications de Montgomery sont basées sur des routines écrites en C.

4.4 Approche logicielle

4.4.1 Implémentation de l'exponentiation modulaire

La méthode d'exponentiation modulaire binaire LSB a été utilisée pour calculer $C = M^E mod\ N$. La fonction *expBinnary()*, qui utilise à son tour la fonction de multiplication modulaire de Montgomery. Cette fonction reçoit comme entrées les vecteurs qui correspondent au message M, à l'exposant E, aux constantes R^2, 1 et au modulo N. Cette fonction ne dépend pas de la base de représentation des données. L'organigramme d'exécution de l'algorithme.2.10, du chapitre deux, est montrée sur la figure 4.4 qui correspond a l'organigramme de la fonction *expBinnary()*.

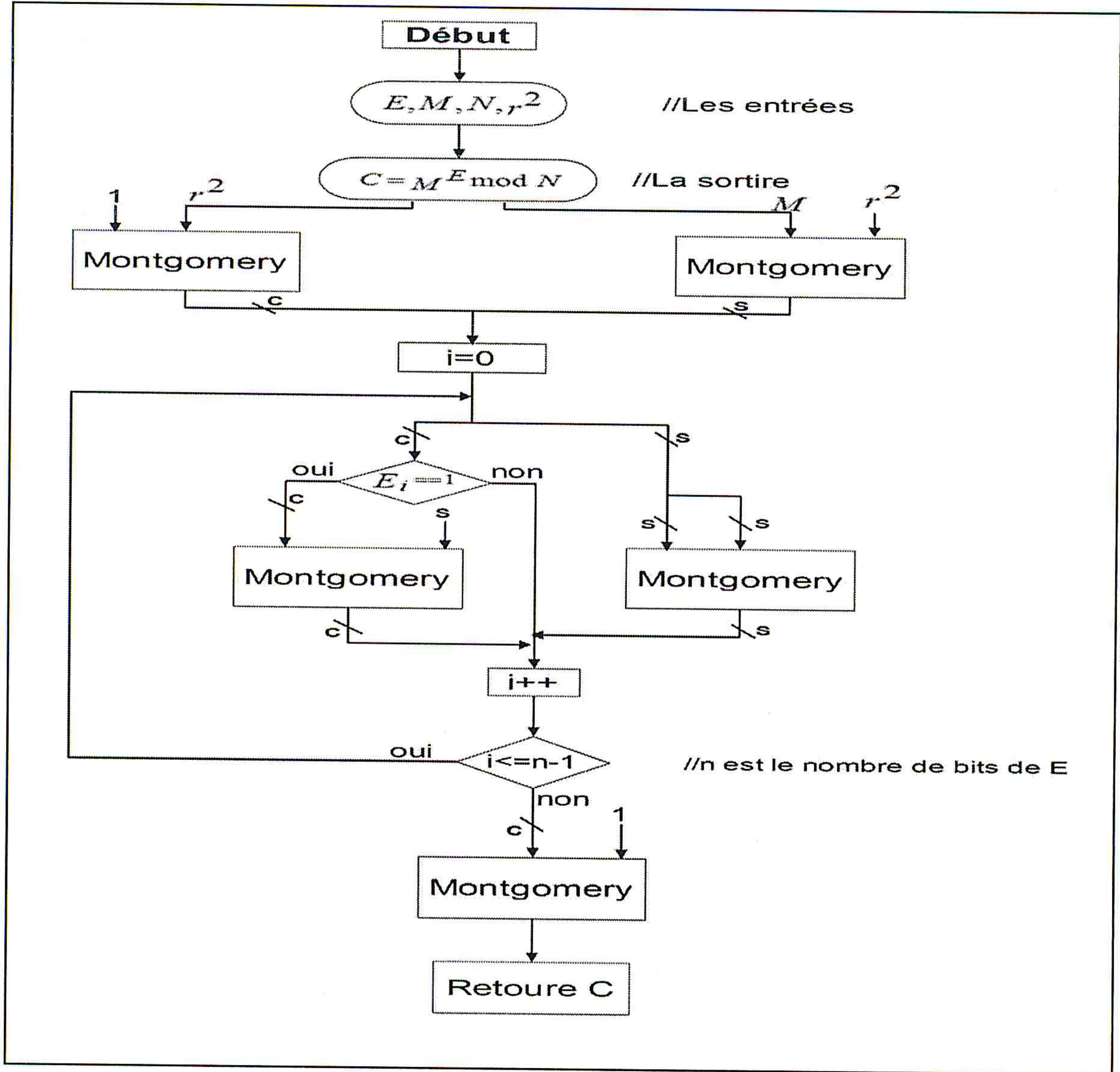


Figure 4.4- Organigramme de la fonction *expBinnary()*

4.4.2 Implémentation de la MMM en base 2

L'organigramme d'exécution de l'algorithme.2.8, du chapitre deux, est montrée, sur la figure 4.5 qui correspond a l'implémentation de la MMM en base 2. Les opérandes d'entrées sont subdivisées en (e+1) mots où chaque mot contient w bits. Puisque le processeur Microblaze est de 32 bits, on utilise w=32 bits. Les opérandes sont représentés par:

$A = \sum_{p=0}^{e} A[p] \times 2^{p*32}$, et $A[p] = \sum_{k=0}^{31} a_k^p \times 2^k$.

$B = \sum_{j=0}^{e} B[j] \times 2^{p*32}$, et $B[p] = \sum_{k=0}^{31} b_k^p \times 2^k$.

N= $\sum_{p=0}^{e} N[p] \times 2^{p*32}$, et $N[p] = \sum_{k=0}^{31} n_k^p \times 2^k$.

En effet, pour des entrées de 1024 bits, ces opérandes sont subdivisés sur 33 mots (e=32) de taille de 32 bits. Il est à noter que dans cet algorithme, on utilise (e+1) mots, pour répondre aux exigences de l'algorithme de Montgomery sans soustraction finale.

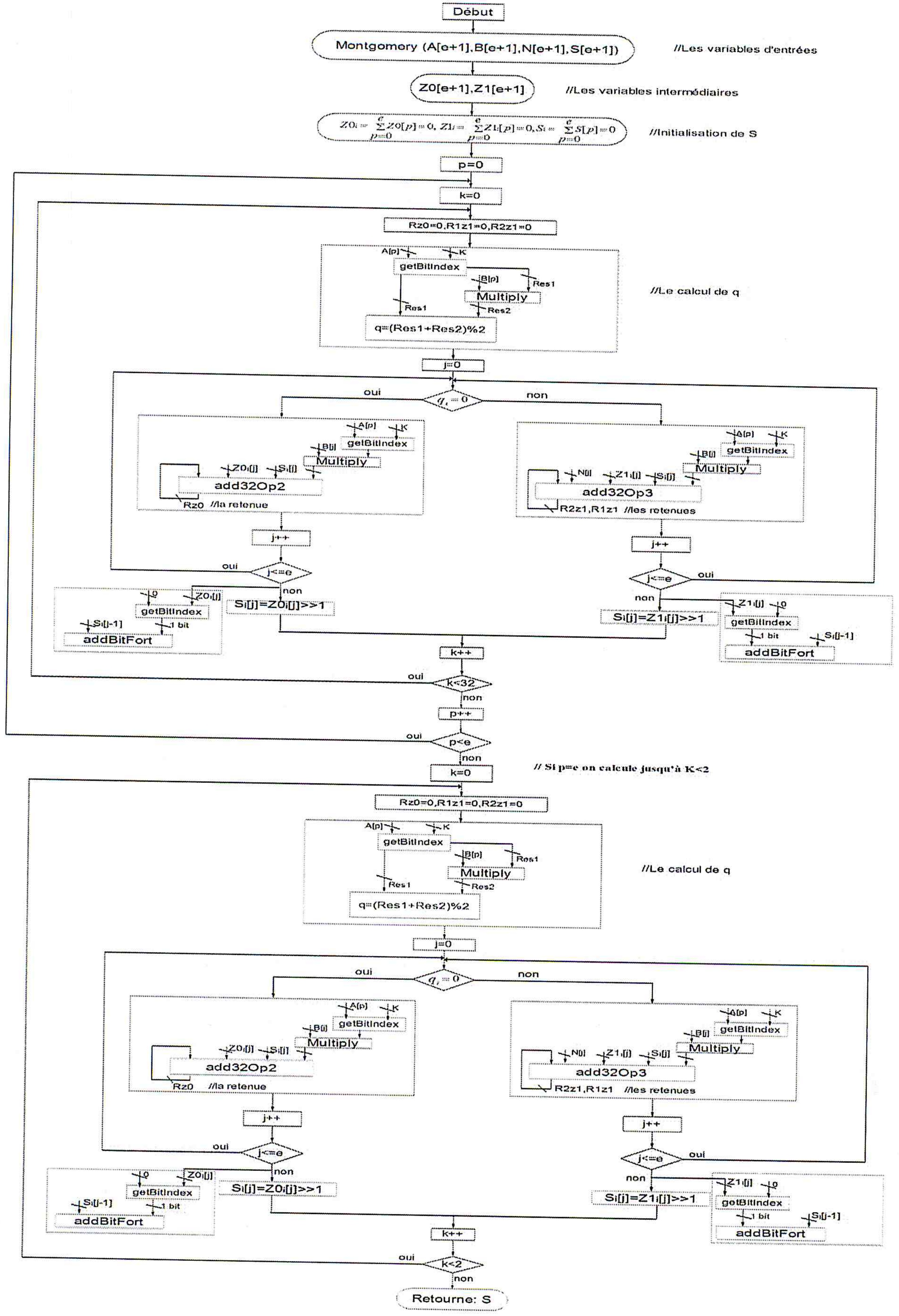


Figure 4.5- Organigramme de la MMM en base 2

Cet algorithme est décomposé en trois étapes:

- La première étape est une initialisation des variables intermédiaires S_i, $Z0_i$ et $Z1_i$.

$S_i = \sum_{p=0}^{e} S_i[p] \times 2^{p\times32} = 0$
$Z0_i = \sum_{p=0}^{e} Z0_i[p] \times 2^{p\times32} = 0$
$Z1_i = \sum_{p=0}^{e} Z1_i[p] \times 2^{p\times32} = 0$

- La deuxième étape est basée sur trois boucles :
  1. La boucle ($0 \leq p < e$) nous permet de parcourir les mots de l'opérande A.
  2. La boucle ($0 \leq k < 31$) est imbriquée dans la boucle (p) pour parcourir chaque bit du mot A[p] et permet de calculer $q_i = (S_{i-1}[0] + A[p][k] + B[0]) \; mod\; 2$.
  3. La boucle ($0 \leq j <$ e+1) est imbriquée dans la boucle (k) et permet de parcourir les mots de l'opérande B, pour calculer:
     - $Z0_i[j] = S_{i-1}[j] + A[p][k] \times B[i] + Rz0$, où Rz0 est le 33eme bit de $Z0_i$[j-1].
     - $Z1_i[j] = S_{i-1}[j] + A[p][k] \times B[j] + N[j] + R1z1 + (R2z1 \times 2), où\; R1z1$ et $R2z1$ sont respectivement le 33ème bit et le 34ème bit de $Z1_i$[j-1].
     - $S_i[j] = Z0_i[j]/2 \; ou \; S[j] = Z1_i[j]/2$ selon la valeur de q_i.

- La troisième étape est utilisée pour prendre en considération le dernier mot A[e] de l'opérande A. En effet, seulement les deux bits les moins significatifs de ce mot sont pris en considération dans les opérations arithmétiques de l'algorithme.

Les fonctions élémentaires de cet organigramme sont:

getBitIndex

Cette fonction consiste à récupérer la valeur du $k^{ème}$ bit (a_k^p) de chaque mot de l'opérande A[p]. Le résultat de la fonction est stocké dans la variable ***bit***. Le bit en question est utilisé dans l'opération $a_k^p \; and \; B$[j].

getBitIndex

```
unsigned long getBitIndex(long mot,
int i)
{
int bit=(mot>>i)&1;  // décalage
return bit;
}
```

add32Op2

La tâche associée à cette fonction est le calcul du terme $Z0_i$. Elle permet d'effectuer une addition de deux nombres de 32 bits *(a+b)* en leur ajourant un bit de retenue={0,1} au poids faible. Le résultat obtenu en l'occurrence *res1,* est représenté sur 33 bits. On sauvegarde les 32 bits les moins significatifs *(res1 (0)...res1 (31))* dans *res,* le 33ème bit est considéré comme étant une retenue. Celle-ci est additionnée au poids faible de la prochaine itération.

add32Op2

```
void add32Op2(unsigned long a, unsigned long b, unsigned long *res, unsigned long *retenue)
{
unsigned long long res1=(unsigned long long)a+b+(*retenue);
*res=(unsigned long) res1;
*retenue=res1>>32&1;
}
```

add32Op3

La tâche associée à cette fonction est le calcul du terme $Z1_i$. Elle représente une addition de trois opérandes a, b et c de 32bits avec deux retenues $retenue1$, $retenue2$. Cette fonction retourne un résultat $res1$ sur 34 bits. Les 32 bits les moins significatifs *(res1 (0) au res1 (31))* sont stockés dans res. Les deux bits retenues $res1(32)$ et $res1(33)$ sont stockés respectivement dans $retenue1$ et $retenue2$.

add32Op3

```
void add32op3(unsigned long a, unsigned long b, unsigned long m, unsigned long *res, unsigned long *retenu1, unsigned long *retenu2)
{
unsigned long long res1=(unsigned long long)a+b+m+(*retenu1)+(*retenu2*2);
*res=(unsigned long) res1;
*retenue1=res1>>32&1;
*retenue2=res1>>33&1;
}
```

addBitFort

Cette fonction permet de concaténer le bit le moins significatif du résultat obtenu à partir des deux fonctions add32op2 ou add32op3 à l'itération 'j' selon la valeur de q_i, avec les 31 bits du poids fort du résultat obtenu à l'itération 'j-1'. Cette fonction est utilisée pour

résoudre le problème de décalage des résultats intermédiaire $S_i[i]$ de l'algorithme de MMM en base 2.

addBitFort

```
void addBitFort(unsigned long *num, unsigned long a)
{
a=a<<31;
*num=*num | a;
}
```

4.4.3 Implémentation de la MMM en base 2^K

Une des caractéristiques de processeur Microblaze est que l'exécutée des opérations arithmétiques nécessite un certain nombre de tops d'horloges bien définie. Afin d'exploiter toutes les capacités du processeur en terme de nombre de cycles d'horloge pour l'exécution des opérations arithmétiques ou logiques, on essaye d'utiliser toute la taille des registres et des opérations arithmétiques et logiques (multiplieur et d'additionneur sur 32 bits) par l'augmentation de la base de la représentation des données. Pour avoir un bon compromis entre le temps d'exécution et la taille des données supportée par le Microblaze, on implémente l'algorithme de Montgomery en base 2^k. L'organigramme d'exécution de l'algorithme 2.9, du chapitre deux, est montrée, sur la figure 4.6 qui correspond à l'implémentation de la MMM en base 2^k.

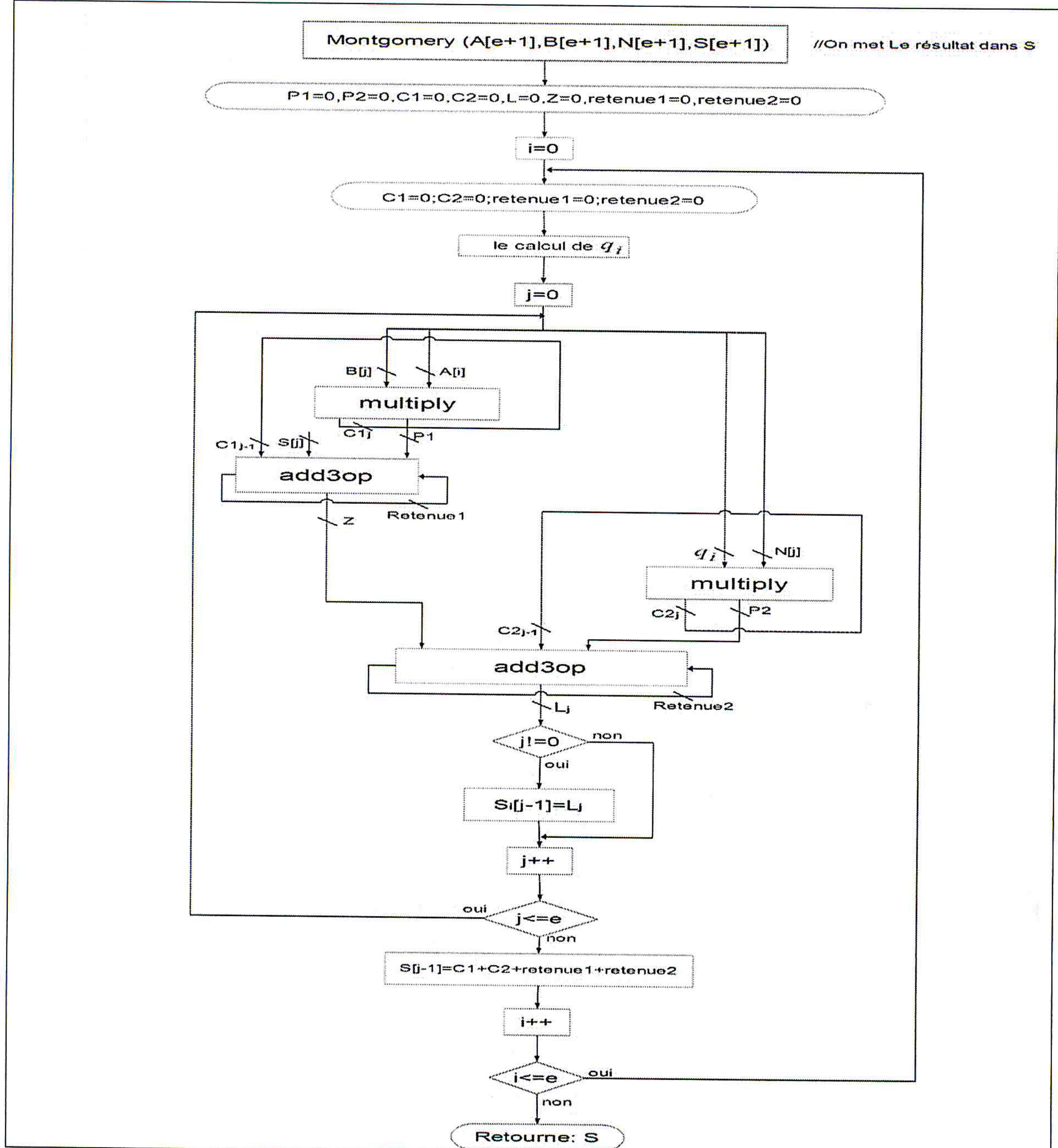


Figure 4.6- Organigramme de l'Implémentation de la MMM en base 2^k

Dans cet organigramme, les opérandes d'entrés A, B et N son subdivisées sur (e+1) mots, où chaque mot est codé sur k bits. Ces opérandes sont représentées par:

$A = \sum_{i=0}^{e} A[i] \times 2^{i*k}$,

$B = \sum_{j=0}^{e} B[j] \times 2^{j*k}$,

$N = \sum_{i=0}^{e} N[i] \times 2^{i*k}$,

Le codage en question dépend de la base utilisée. En effet, pour un modulo de 1024 bits, en base 2^{32}, les opérandes sont subdivisés sur 33 mots (e=32), ils sont codés sur 32 bits. Cet algorithme est décomposé en deux étapes:

- La première étape est une initialisation des variables intermédiaire S_i, L_i *et* Z_i :

  $S_i = \sum_{j=0}^{e} S_i[j] * 2^{j*k} = 0$

  $P1_j = 0,\ P2_j = 0,\ Z_j = 0,\ L_j = 0$, où $P1_j, P2_j,\ Z_j et\ L_j$ et sont codées sur k bits.

- La deuxième étape repose sur trois boucles :
  1. La boucle (0 ≤ i < e) nous permet de parcourir les mots de l'opérande A pour calculer $q_i\ =\ (A[i] \times B[0] + S_i[0]) \times N')\ mod\ 2k$, où N' est un nombre pré-calculé.
  2. La boucle (0 ≤ j < e) est imbriquée dans la boucle (i) et a pour rôle de parcourir les mots de l'opérande B. Les opérations mathématiques effectuées dans cette boucle sont:
     - $A[i] \times B[j] = P1_j + C1_j \times 2^k$ (3)
     - $P1_j + S_i[j] + C1_{j-1} + retenue1_{j-1} = Z_j + retenue1_j \times 2^k$ (4)

       où $P1_j$ *et* $C1_j$ sont des variables sur k bits *et* représente respectivement le mot le moins significatif et le plus significatif du résultat de la multiplication $A[i] \times B[j]$. $retenue1_{j-1}$ est une retenue de 2 bits associée à l'addition précédente de l'expression (4).
     - $q_i \times N[j] = P2_j +\ C2_j \times 2^k$ (5)
     - $L_j + retenue2_j \times 2^k = P2_j + Z_j + C2_{j-1} + retenue2_{j-1}$ (6)

       $P2_j$ représente le mot le moins significatif de la multiplication $q_i \times N[j]$, $C2_j$ est le mot le plus significatif de cette multiplication. $retenue2$ est représentée sur 2 bits. Celle-ci est générée par l'addition précédente de l'expression (6).

Finalement, une fois avoir obtenu $L_i = \sum_{j=0}^{k} L_j \times 2^k$, le calcul des résultats intermédiaires est effectué par un simple décalage de k bits vers le poids faible de L_i. Ainsi, le $j^{ème}$ mot de L_i correspond au $(j\text{-}1)^{ème}$ mot de S_i.

- $S_i[j-1] = L_j$ *(7)*

A la fin de ces boucles, le résultat *S* est représenté avec un ensemble de mots *S[j]* de k bits. Les fonctions élémentaires de cet organigramme sont:

multiply

Cette fonction permet de calculer les deux multiplications $A[i] \times B[j]$ et $q_i \times N[j]$. En effet. Dans chaque itération (i), A[i] et q_i sont considérés comme étant des constantes codées sur k bits. B[j] et N[j] sont variables.

Ainsi, la fonction étudiée assure l'exécution d'une multiplication de deux données de k bits, le résultat fourni par cette fonction est codé sur 2k bits. Les k bits les moins significatifs représentent le résultat de l'itération (j) ($P1_j$ ou $P2_j$), ils seront utilisés par la suite dans une

autre fonction d'addition pour calculer respectivement $Z1_j$ et L_j. Les k bits les plus significatifs en l'occurrence $C1_j$ et $C2_j$ représentent les retenues issues des multiplications citées ci-dessus. Ces derniers seront additionnés aux résultats des multiplications à l'itération (j+1).

multiply

```
void multiply(unsigned long A, unsigned long B, unsigned long *Retenue,
unsigned long *Resultat)
{
  unsigned long long r=(unsigned long long)A*B;
 // (unsigned long long) : pour obtenir un resultat de A*B sur 64 bits

 *Resultat=(unsigned long) r;
 // (unsigned long) : convertir  'r' sur 32 bits dans Resultat, qui représente
   les 32  bits les moins significatif de 'r'

 *Retenue=r>>32;  // décalage à droite de 32 bits pour obtenir les 32 bits
   les plus significatifs de 'r'
}
```

add3Op

Cette fonction nous permet de calculer Z_j et L_j données respectivement par les expressions (4) et (6), et qui nécessitent des opérations d'additions de trois nombres. En effet, à l'itération (j), l'addition de trois opérandes codées sur k bits donne comme résultat, un mot représenté sur k bits et deux retenues de poids 2^k. Les retenues sont bouclées sur les entrées de la fonction add3op, ils sont additionnés au résultat qui correspond à l'itération (j+1).

add3Op

```
void add3Op(unsigned long A, unsigned long B, unsigned long C, unsigned long *retenue,
unsigned long *resultat)
{
    unsigned long long r=(unsigned long long)A+B+C+(*retenue);
   *resultat=(unsigned long) r;
   *retenue=r>>32;
}
```

Remarque

Dans ce travail, nous nous somme intéressé au cas: k=16 bits et k=32 bits, pour un modulo N de taille 1024 bits.

4.5 Approche matérielle

Dans cette seconde approche, la multiplication modulaire de Montgomery est implémentée sous forme d'un IP sur matériel.

Le calcul de l'exponentiation modulaire, est réalisé par un programme où le calcul de la multiplication modulaire de cette fonction est réalisé dans l'IP matériel. Cette approche nous permet non seulement d'accélérer le calcul de la multiplication modulaire, mais aussi de rendre le calcul de l'exponentiation modulaire flexible. L'architecture globale de cette approche d'implémentation est montrée sur la figure 4.7.

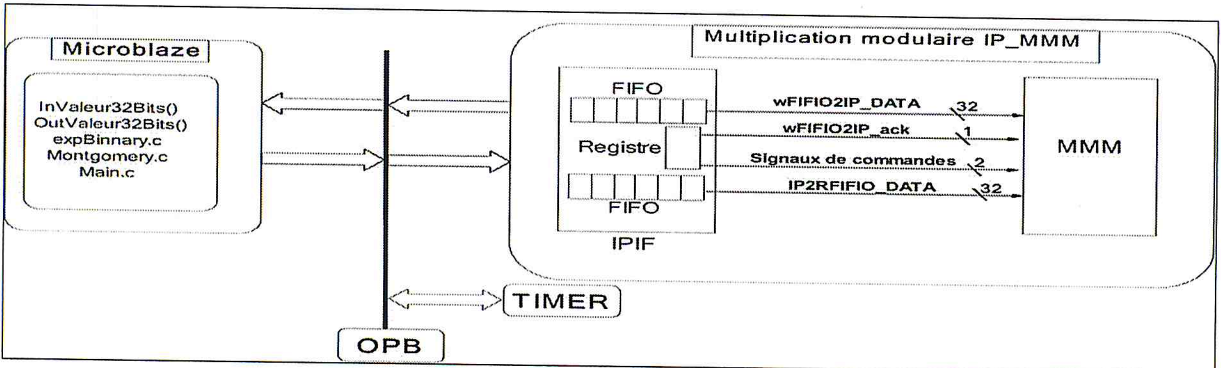


Figure 4.7 : Architecture générale pour le calcul de l'exponentiation modulaire par l'approche matérielle

Cette architecture est décomposée en deux parties :

1) La partie logicielle qui gère le calcul de l'exponentiation modulaire en utilisant l'IP comme un moyen d'accélérer les calculs de la multiplication modulaire qui sont une partie de cette fonction. Il est à noter que cette partie est prise en charge par le processeur Microblaze.

2) La partie matérielle qui est l'IP lui-même, qui est constituée de deux modules. Le premier module joue le rôle d'une interface entre le bus system OPB et l'IP. Cette interface est nommée IPIF, plus précisément, pour assurer la communication entre le processeur Microblaze et les périphériques qui l'entourent. Le second module assure l'exécution des opérations arithmétiques de la multiplication modulaire. A noter que dans ce travail, nous avons adapté le module développé dans [29] pour l'implémentation matériel de la multiplication modulaire de Montgomery.

L'interface IPIF possède plusieurs configurations. En effet elle peut être configurée comme des registres, FIFOs, générateur d'interruption …etc.

Dans cette implémentation, nous avons utilisé trois des ses options, à savoir, deux mémoires FIFOs et un registre. Les deux FIFOs sont utilisées respectivement pour la transmission et la réception des données entre le bus OPB et le module de la multiplication modulaire. Le

registre assure la commande de la MMM à partir du processeur Microblaze. Pour permettre cette échange de donnée, Xilinx à développer des pilotes (des fonctions en langage C) qui permettent de sélectionner les options de l'IPIF. Les fonctions en question sont résumées dans le tableau 4.1.

Fonction	Description
MY_MMM_32BIT_mWriteToFIFO(adresse ,Valeur);	Ecriture dans une FIFO
MY_MMM_32BIT_mReadFromFIFO(adresse);	Lecture à partir d'une FIFO
MY_MMM_32BIT_mWriteFromSlaveReg0(adresse,0);	Ecriture dans le registre d'indice 0
MY_MMM_32BIT_mReadFromSlaveReg0(adresse);	Lecture à partir d'un registre d'indice 0

Tableau 4.1- Fonctions en C des options de l'IPIF

Dans cette seconde approche d'implémentation, nous nous somme intéressés au développement des programmes qui sont exécutés sur le processeur. Les programmes en question sont des codes en langage C suivant une certaine hiérarchie. Ils assurent principalement :

- La transmission à partir de l'application Java du message en claire ou chiffré et de l'exposant vers le processeur Microblaze à travers le port RS232.
- Exécution de l'exponentiation modulaire par un échange itératif des données (opérandes et résultats) entre le processeur et l'IP.
- Transmission du résultat de l'exponentiation modulaire vers l'application Java à traves le port RS232.

Le programme principal (main) exécutant ces trois tâches est constitué de quatre fonctions. Ces dernières sont définies comme suit :

- *inValeur32bit() ;*
- *expBinnary() ;*
- *Montgomery() ;*
- *outValeur32bit() ;*

La fonction *inValeur32bit()* permet de reconstituer des mots de 32 bits, fournies, par UART au Microblaze, sur 8 bits. Cette fonction est basée sur deux boucles l'une est interne, l'autre est externe. Ces boucles permettent de :

- Constituer un mot de 32 bits.
- Charger dans la mémoire BRAM, une donnée de taille n bits, tel que $0<n\leq1024$ bits.

Cette fonction est basée sur le pilote *XUartLite_RecvByte(XPAR_RS232_BASEADDR)* qui est reconnu par le système d'exploitation de Microblaze dont la tâche est d'écrire dans les registres internes du processeur, une donnée reçue à partir du port RS232.

La fonction *expBinary()* assure l'exécution de l'algorithme d'exponentiation modulaire. Celle-ci reçoit en entrées, le message, l'exposant et la taille de l'exposant. Les deux premières entrées sont fournies par la fonction précédente sous forme d'un tableau constitué par *e* mots. Dans la description interne de *expBinary()*, deux valeurs constantes sont utilisées, à savoir, la valeur '1'et R^2, ces constantes sont nécessaires pour représenter les opérandes dans le domaine de Montgomery. Pour le calcul de l'exponentiation, cette fonction utilise, d'une manière itératif, une sous fonction nommé *montgomery32(),* pour effectuer les calculs intermédiaires des multiplications modulaire de Montgomery. Les résultats issus de ces multiplications et qui correspondent à *C=montgomery(C,S)* et *S=montgomery(S,S)* sont stockés respectivement dans deux tableaux nommés C[e+1] et S[e+1]. Finalement en utilisant cette approche d'implémentation, le nombre d'appels associes à l'utilisation de la sous fonction *montgomery32()* dans la fonction *expBinary()* est de $(2\times n+3)$.

Les résultats de l'exponentiation modulaire sont obtenus mot par mot à partir de la dernière multiplication modulaire. Celle-ci correspond à la représentation dans le domaine classique du résultat issu de l'itération $i=n-1$ de l'algorithme de l'exponentiation modulaire.

Pour transmettre le résultat vers le PC, nous avons développé un dernier programme dont la fonction est nommé *outValeur32()*. Celui-ci reçoit en entrée le résultat de la fonction *expBinary()* mot par mot sur 32 bits et fourni à l'UART après la décomposition de chaque mot, des blocs de 8 bits. Ce programme utilise la fonction *XUartLite_SendByte(XPAR_RS232_BASEADDR,valeur)*. A la fin, l'UART transfert le résultat bit par bit selon le protocole RS232 au PC. Le résultat en question est traité par l'application Java pour sa représentation en décimal.

La description de la fonction *montgomery32()* utilise essentiellement des pilotes associés à l'IPIF (voir tableau 4.1). Cette fonction reçoit en entrées sous forme de mots de 32 bits, deux opérandes et trois signaux de contrôle *int_control, int_operation et int_system*. Dans la fonction de *montgomery()*, ces signaux sont utilisées sous forme de code pour contrôler l'IP. Les codes attribués à ces signaux sont montrés dans le tableau 4.2.

Code	Variable	Description
00	int_control	début du transférer les données vers le module MMM
10	int_system	Remise à zéro de transfère des données et la MMM
01	int_opération	Début de calcul d'une multiplication modulaire de Montgomery

Tableau 4.2- Les codes des signaux de contrôle

Ces signaux sont transmis vers le module MMM par l'utilisation d'un registre de l'IPIF. Le pilote de ce registre est *MY_MMM_32BIT_mWriteFromSlaveReg0(adresse,valeur)*. Les opérandes sont transmises via une FIFO par l'utilisation du pilote *MY_MMM_32BIT_mWriteToFIFO(adresse, valeur)*. Les résultats de MMM sont fournis au processeur via une autre FIFO sous forme de mots de 32 bits par l'utilisation de pilote *MY_MMM_32BIT_mReadFromFIFO(adresse)*.

L'organigramme du fonctionnement de la fonction *montgomery32()* est montré sur la figure 4.8.

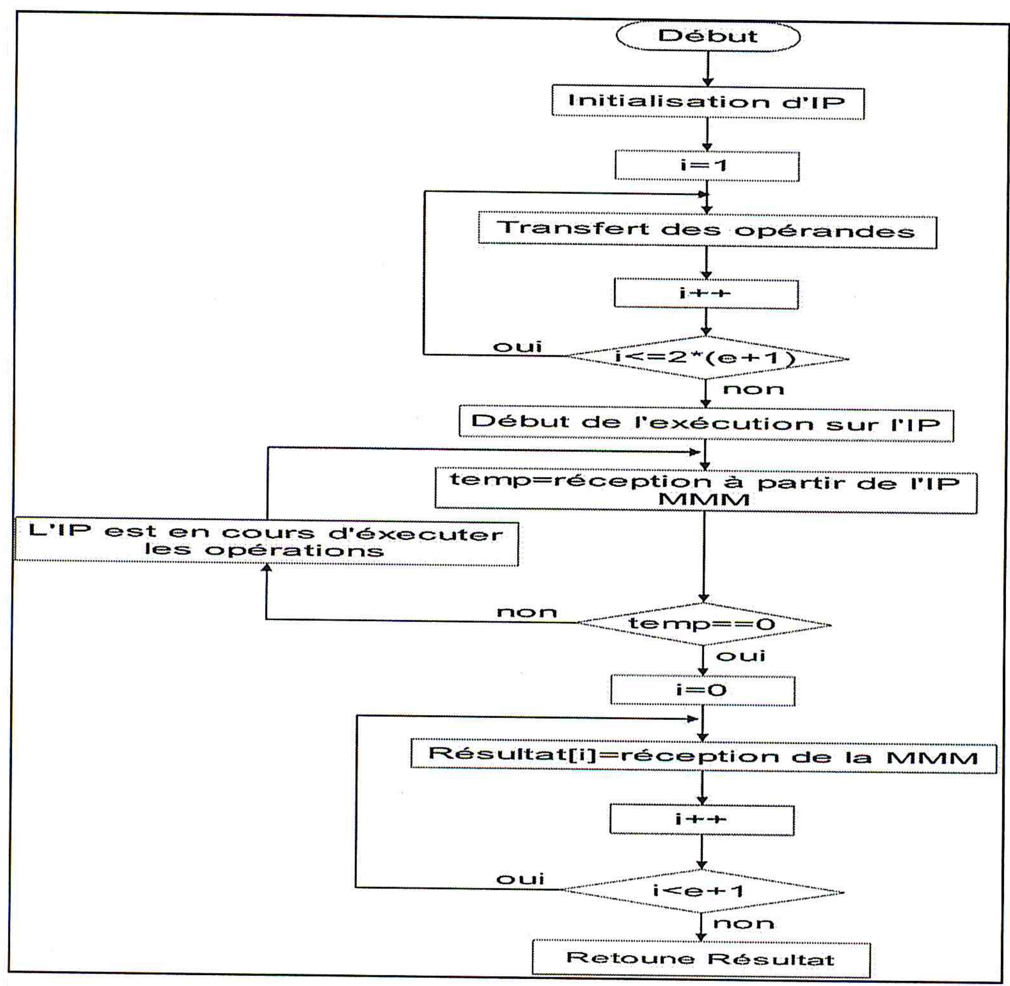


Figure 4.8- Organigramme du calcul de l'exponentiation modulaire

La fonction *montgomery32()* est la partie logiciel qui est exécutée sur le processeur Microblaze afin de contrôler la MMM implémenté en matériel. Le calcul d'une multiplication

modulaire est effectue par un certain protocole que nous avons définit. Ce protocole repose sur l'utilisation des deux FIFO et du registre de l'IPIF. Les deux FIFOs assurent le transfert point à point entre le processeur et l'IP. Le registre permet de contrôler le transfert des opérandes du processeur vers l'IP et de commander le début de l'opération. Le transfert dans le sens inverse, c'est-à-dire, de l'IP vers le processeur repose sur l'état de la liaison de communication au repos. En effet, quand l'IP est en état d'exécution d'une opération, la liaison IP-Microblaze est à l'état haut. Autrement dit, la valeur 0xFFFFFFFF est présente sur le bus données IP2RFIFO. Une fois que l'IP achève le calcul d'une opération, cette liaison de communication est remise à zéro (la valeur 0x00000000 est présente sur le bus IP2RFIFO). Ainsi, les résultats d'une opération sont transmis mot par mot après le premier zéro détecté sur le bus de données IP2RFIFO.

Le début d'une opération commence en premier lieu par le transfert sur le bus de données wFIFO2IP des deux opérandes A et B à travers la mémoire FIFO de l'IPIF. Ce transfert est commandé par l'utilisation du registre de l'IPIF qui reçoit le code "00". Afin de s'assurer du transfert complet des deux opérandes ($2\times$ (e+1)) mots, un compteur i est défini dans la fonction *montgomery32()*.

Une fois que ce transfert est achevé, *($i=2\times(e+1)$)*, l'IP passe à l'étape de l'exécution de l'opération, le début de l'exécution d'une opération est commandé par Microblaze par l'utilisation du code "01". Durant cette étape, la liaison IP_Microblaze est au repos, c'est-à-dire, qu'une valeur de 0xFFFFFFFF est présente sur le bus de données IP2WFIFO. Dès que celle-ci devient zéro, le transfert du résultat de calcul est entamé et les deux premiers zéro seront suivis par (e+1) mots. Pour assurer le transfert complet de ce résultat, un compteur 'i' est utilisé dans la fonction de Montgomery. Une fois que le résultat est entièrement transmis ($i=e+1$), l'IP est mis a zéro par l'utilisation du code "10". Ainsi, l'IP sera prêt pour le calcul d'une autre multiplication modulaire.

4.6 Application JAVA

La contribution de cette application dans le crypto système implémenté peut être considérée comme étant une interface entre la partie implémentée sur le circuit FPGA et l'utilisateur.

Les tâches attribuées à cette application se résument dans les points suivants:

- La conversion des données.
- La communication avec la carte V2MB1000.

Pour ce faire, cette application contient deux classes *Class BigInteger*, *Class CommunicationRS232*, et un programme principal en l'occurrence, *Class Main*. Ce dernier permet de gérer les deux premières classes. Les fonctions principales des deux classes sont représentées sur le tableau 4.3.

Class BigIntegerPro extends BigInteger	***Class CommunicationRS232***
public BigIntegerPro(int i, Random random);	public void outValeur16Bit(int valeur);
public String[] getTabBigInteger16Bit();	public void outValeur32Bit(int valeur);
public String[] getTabBigInteger32Bit();	public long inValeur16Bit();
public BigInteger getBigInteger16Bit (String[] val);	public long inValeur32Bit();
public BigInteger getBigInteger32Bit (String[] val);	
Public BigInteger getListBigIntegerFromString(String[] val)	
Public BigInteger getListBigIntegerToString(String[] val)	

Tableau 4.3- *Classe BigIntegerPro* et *Classe CommunicationRS232*

4.6.1 Class BigIntegerPro

La classe *BigIntegerPro* est une extension (extends) de la classe *BinInteger* de Java. Cette classe contient la définition d'un nombre BigInteger et toutes les opérations qui s'exécutent sur des entiers de grandes tailles. Les opérations en question peuvent être, des additions, des soustractions, des multiplications, des divisions, calcul de l'exponentiation…etc. Nous avons utilisé cette classe dans laquelle nous avons ajouté des fonctions afin de satisfaire nos besoins dans la programmation de la plateforme. Ces fonctions sont définies comme suit :

BigIntegerPro(String val) : permet de créer BigInteger à partir d'une chaine de caractères. Cette fonction retourne un nombre de type **BigInteger**.

BigIntegerPro(int i, Random random) : retourne un BigInteger de taille 'i', que le compilateur génère d'une manière aléatoire (Random). La multiplication de deux nombres A et B générés par cette fonction est montrée sur la figure 4.8.

```
BigIntegerPro A= null, B= null;
BigInteger C= null;
A = new BigIntegerPro(1024,new Random()); // Creation d'un BigInteger A
B = new BigIntegerPro(1024,new Random()); // Creation d'un BigInteger B
C = A. Multiply(B); // C=A*B
```

Figure 4.9 : Multiplication de deux grands nombres

getTabBigInteger16Bit() : Permet la conversion d'un *BigInteger* généré par l'une des deux fonctions précédentes en base 2^{16}. Le résultat de cette fonction est obtenu sous forme d'un tableau, dont les éléments (mots) sont représentés sur 16 bits.

Exemple : Soit A= 3765912389, sa représentation en utilisant cette fonction est montrée sur la figure 4.10

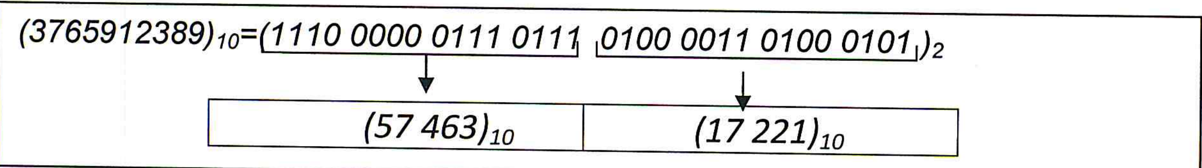


Figure 4.10 : Exemple de conversion d'un bigInteger en base 2^{16}

getTabBigInteger32Bit() : Le rôle de cette fonction est identique à la précédente, sauf que l'entrée (BigInteger) est convertie en base 2^{32}.

Exemple: Soit A= 321 799 863 630 512 452, sa représentation en utilisant cette fonction est montrée sur la figure 4.11

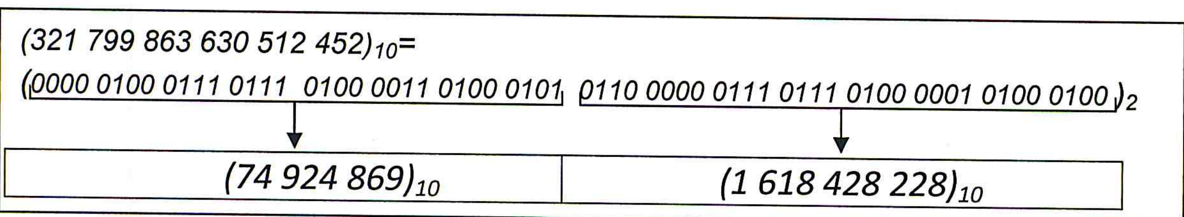


Figure 4.11 : Exemple de conversion d'un bigInteger en base 2^{32}

getBigInteger16Bit (String[ ] val): Assure la représentation d'un BigInteger en décimal, d'un nombre représenté en base 2^{16}. Autrement dit, elle peut être considérée comme étant la fonction inverse de la fonction ***getTabBigInteger16Bit().***

getBigInteger32Bit (String[ ] val) : Cette fonction permet la représentation d'un BigInteger en décimal, d'un entier représenté en base 2^{32}.

getListBigIntegerFromString(String[] val): Cette fonction permet de convertir une chaine de caractères de taille variante codé en ASCII en un tableau de BigInteger codé en binaire sur 1023 bits, le 1024 bits est considéré comme 0.

getListBigIntegerToString(String[] val): Cette fonction permet de reconstituer à partir d'un tableau de BigInteger codé en Binaire, une chaine de caractère Codée en ASCII. C'est la fonction inverse de *getListBigIntegerFromString(String[] val).*

4.6.2 Class CommunicationRS232

La class CommunicationRS232 a été crée principalement pour la transmission et la réception des données à travers le port RS232. Pour se faire, il est nécessaire d'établir une connexion avec ce port avant d'échanger les données en respectant le protocole RS232. Les principales fonctions de cette classe sont:

communtionRS232(String Port): Permet d'établir une connexion avec le port RS232.

outValeur16Bit(int valeur): Cette fonction assure:

- La décomposition sur des mots de 8 bits du résultat fourni à partir de la fonction ***getTabBigInteger16Bit()***.
- La transmission vers l'UART du port DB9 du PC, des données représentées sur 8 bits.

outValeur32Bit(int valeur): Cette fonction joue le même rôle que la précédente. Sauf qu'elle prend en charge les données fourni par la fonction *getTabBigInteger32Bit()*.

inValeur16Bit(): Consiste à reconstituer sur 16 bits, les résultats fournis par la carte V2MB1000 à travers le port RS232. Pour cela, dès qu'on reçoit deux mots de 8bits successive, on les stocke dans une valeur de 16 bits, le premier mot est le poids faible, le second est le poids fort.

inValeur32Bit(): Consiste à reconstituer sur 32 bits, les résultats fournis par la carte V2MB1000 à travers le port RS232.

4.6.3 Class Main

Cette classe est considérée comme étant le programme principal de l'application JAVA que nous avons développé. La classe en question est constituée par les différentes fonctions qui appartiennent aux classes *BigIntegerPro* et *CommunicationRS232*. Le code de cette classe est montré sur la figure 4.12.

```
Lire (Message);
Message.getListBigIntegerFromString();// decouper le message en paquet de 1024 bits
for(String list : Message)
{
List.getTabBigInteger(K)Bit();
lire(Clé);
Clé.getTabBigInteger(K)Bit();
communtionRS232("Com1");
for(String out : List) Communicationrs232.outValeur(K)Bit(out);
// envoyer tout les éléments de la table Message(mots de K bits)
for(String out : Clé)  Communicationrs232.outValeur(K)Bit(out);
// envoyer tout les éléments de la table Clé(mots de K bits)
for(i=1;i<33;i++) resultat[i]= Communicationrs232. inValeur(K)Bit();
// récupérer  les Résultats envoyé par le circuit FPGA
résultat. getBigInteger(K)Bit();
}
Enregistrer(résultat); // stocker les résultats
résultat.getListBigIntegerToString();// reconstituer le message_chiffré a partir des
resultas de 1024 bits
```

Figure 4.12- Algorithme de la Classe Main

4.7 Conclusion

Dans ce chapitre, nous avons présenté les applications logicielles des différentes implémentations de la multiplication modulaire de Montgomery et de l'exponentiation modulaire, à savoir la MMM en Base 2, en Base 2^{16} et en base 2^{32}. Ainsi, l'utilisation d'un IP_MMM pour le calcul de la multiplication modulaire.

La méthodologie, le cycle de développement et les résultats obtenus sont détaillés dans le chapitre suivant.

Chapitre 5 :

Résultats d'implémentation

5.1 Introduction

L'objectif principal de ce travail est de développer une plateforme de chiffrement/déchiffrement basée sur le protocole RSA. La réalisation de cette plateforme a été effectuée en trois étapes définies comme suit:

1. Conception de l'architecture globale de la plateforme et génération du fichier de configuration du circuit FPGA (BitStream).
2. Développement de l'application avec son IHM (Interface Homme Machine) en Java
3. Vérification en temps réel de la plateforme sur la carte V2MB1000.

Avant de parvenir à la vérification fonctionnelle sur la carte de prototypage V2MB1000, le cycle de conception de notre architecture a été effectué dans l'environnement EDK de Xilinx. Dans ce chapitre, nous allons présenter les différentes étapes de réalisation, ainsi que les résultats d'implémentation sur circuit FPGA.

5.2 Conception et Implémentation sur circuit FPGA

Nous avons développé trois architectures pour le calcul de l'exponentiation modulaire. Ces architectures sont : Microblaze sans options, Microblaze avec options et Microblaze avec IP_MMM.

Le cycle de conception pour l'implémentation de ces architectures est subdivisé en cinq étapes.

1. Description de l'architecture matérielle.
2. Synthèse et implémentation sur circuit FPGA.
3. Génération et chargement du BitStream sur le circuit FPGA.
4. Description de la partie logicielle.
5. Chargement de la partie logicielle sur le circuit FPGA.

5.2.1 Description de l'architecture matérielle

Dans cette étape, il est question de concevoir la partie matérielle de la plateforme de notre système. Pour ce faire, il est important de connaitre les composants élémentaires nécessaires pour son adéquation avec les besoins de l'application ciblée.

L'interface graphique d'EDK est nommée **XPS** (Xilinx platforme Studio). Cette interface comporte plusieurs assistants (Wizard) qui guident le concepteur pour créer des systèmes

embarqués à base du processeur Microblaze et des périphériques (IPs) qui l'entourent. L'interface XPS est composée de plusieurs outils tels que:

- **BSB (Base system builder)**

L'outil BSB nous permet de:

- Choisir Microblaze en tant que processeur principal.
- Définir de la fréquence système.

  Dans l'approche logicielle implémentée, la fréquence utilisée est de 100MHZ. Dans l'approche matérielle, nous avons utilisé une fréquence de 33MHZ qui a été imposée par le chemin critique de la multiplication modulaire implémentée.
- La polarisation du signal d'initialisation système (Reset actif à l'état haut ou bas).
- La sélection des périphériques de base, tels que : l'UART, un module pour débogage (en software, ou en hardware) et un Timer.
- Sélection de la taille de la mémoire interne BRAM où dans notre cas est de 32 kbits.

- **System Assembly View**

Cet outil permet de modifier une configuration ou les options des composants.

- **Create or import peripheral**

Cet outil nous permet de rajouter au bus système, un périphérique personnalisé. Nous avons utilisé cette option dans l'approche matérielle d'implémentation de la multiplication modulaire.

- **Launch Platform SDK**

Cet outil nous permet de créer la partie logicielle (programme en langage C) de notre application. En effet, il est considéré comme étant une passerelle entre XPS et SDK.

5.2.2 Synthèse et implémentation sur circuit FPGA

Cette étape n'est autre qu'une transformation de l'architecture globale en un ensemble d'équations booléennes. Cette étape est réalisée dans l'environnement ISE de Xilinx. Le résultat obtenu de la synthèse sous forme de net-list est optimisé et transformé en une autre net-list de blocs logiques qui sera placée et routée sur le circuit utilisé, en l'occurrence le circuit x2V1000 fg465-4. Cette famille de circuits est présentée dans l'annexe B.

Les résultats de l'implémentation sont obtenus dans un rapport qui porte le nombre de ressources internes utilisées et les taux d'occupation.

5.2.3 Génération et chargement du BitStream sur le circuit FPGA

Dans cette étape, il est question de:

1) Générer le fichier de configuration (BitStream) de la partie matérielle.
2) Charger le fichier de configuration sur le circuit FPGA.

Ces deux tâches sont réalisées dans l'interface graphique XPS d'EDK, en utilisant respectivement les deux outils: "generate BitStream" et "Download BitStream".

Le chargement du BitSream sur le circuit FPGA est effectué à travers le port parallèle du PC. La connexion entre ce dernier et la carte V2MB1000 est assurée par le câble JTAG.

5.2.4 Description de la partie logicielle

Cette étape consiste à créer:

- Les pilotes permettant à l'application java de communiquer avec le processeur.
- Les fonctions correspondantes aux différentes approches implémentées.

Les composants appartenant à cette partie sont décrits en langage C et sont résumés dans le tableau 5.1. Ces composants sont constitués de fonctions qui ont été décrites dans le chapitre précédent et sont détaillées dans le tableau 5.2.

	Types d'implémentation logicielle	Composant	Description
Approche logicielle d'implémentation	RSA_Binary_Base_2_1	expBinnary2.c Montgomery_Base2.c Communication32.c Main.c	Calcul de l'exponentiation modulaire Calcul de la MMM en Base 2 Communication avec le port RS232 Programme principal
	RSA_Binary_Base_2_16	expBinnary16.c Montgomery_Base16.c Communication16.c Main.c	Calcul de l'exponentiation modulaire Calcul de la MMM en Base 2^{16} Communication avec le port RS232 Programme principal
	RSA_Binnary_Base_2_32	expBinnary32.c Montgomery_Base32.c Communication32.c Main.c	Calcul de l'exponentiation modulaire Calcul de la MMM en Base 2^{32} Communication avec le port RS232 Programme principal
Approche matérielle d'implémentation	RSA_Binnary_IP_MMM	expBinnary.c Montgomery.c Communication32.c Main.c	Calcul d'exponentiation modulaire Calcul de la MMM avec IP_MMM Communication avec le port RS232 Programme principal

Tableau.5.1- Approches et types d'implémentations

Composant	Fonction
expBinary2.c	expBinary_Base2(M,E,N,Res);

Montgomery_Base2.c	getBitIndex(mot , i);
	add32Op2(a, b, res, retenue);
	add32op3(a, b, m, res, retenue1, retenue2);
	addBitFort(mot, a);
	Montgomery_base2(A,B,N,Res);

Communication32.c	InValeur32bits(); OutValeur32bits(valeur);
expBinary16.c	expBinary_Base16(M,E,N,Res);
Montgomery_Base16.c	multiply(A, B, Retenue, Resultat); add3Op(A, B, C, retenue, resultat);
Communication16.c	InValeur16bits(); OutValeur16bits(valeur);
Montgomery_Base32.c	multiply(A, B, Retenue, Resultat); add3Op(A, B, C, retenue, resultat);
expBinary32.c	expBinary_Base32(M,E,N,Res);
Communication32.c	InValeur32bits(); OutValeur32bits(valeur);
expBinary.c	expBinary() ;
Montgomery.c	Montgomery() ;
Communication32.c	InValeur32bits(); OutValeur32bits(valeur);

Tableau.5.2- Les fonctions des composants des deux types d'implémentation

Il est à noter que dans chacune des deux premières implémentations (Microblaze avec et sans option), trois types d'implémentations logicielle ont été effectuées. Ces dernières sont : RSA_Binary_Base2, RSA_Binary_Base16 et RSA_Binary_Base32.

Remarque

Le programme principal de chaque approche (Main.c) inclut aussi les pilotes associés au Timer. Ce dernier a été utilisé dans le but de quantifier le nombre de tops d'horloges nécessaires pour l'exécution de toute la plateforme (transmission et réception des données et exécution de l'exponentiation modulaire). Les pilotes en question sont montrés dans le tableau 5.3

Pilotes	Description
XTmrCtr deley;	Création d'une variable delay
XTmrCtr_Initialize(&deley,0x41c00000);	Déclaration du timer
XTmrCtr_SetResetValue(&deley,1,0);	Déclaration d'une valeur initiale du Timer
XTmrCtr_Start(&deley,1);	Lancement du comptage
XTmrCtr_Stop(&deley,1);	Arrêt du comptage
XTmrCtr_GetValue(&deley,1));	Récupération du nombre de tops d'horloge

Tableau.5.3- Pilotes du Timer

5.2.5 Chargement de la plateforme logicielle sur le circuit FPGA

L'outil SDK nous permet de développer la partie logicielle. Celle-ci est constituée par les composants que nous avons cités dans le paragraphe précédent. Cet outil est utilisé aussi pour choisir le type d'implémentation logicielle qui sera exécutée par le processeur Microblaze.

5.3 Développement de l'application Java

Cette application a été développée afin de piloter à partir du PC le crypto-système embarquée sur FPGA. Celle-ci permet de préparer et de transférer les données au circuit FPGA à travers le port RS232 (message, exposant, modulo).

A noter que dans la seconde approche d'implémentation (multiplication modulaire sur matérielle), le modulo est transmis dans le BitStream. Dans le but de vérifier le fonctionnement de l'application sur la carte V2MB1000, une IHM a été développée.

Pour répondre aux contraintes de l'application ciblée (implémentation d'un crypto-système RSA), cette IHM est constituée de quatre parties :

1. Une partie pour la configuration du port de communication RS232.
2. Une partie pour la génération des clés ((E, N), (D, N)).
3. Une partie pour le chiffrement et le déchiffrement d'un message.
4. Une dernière partie pour le chiffrement ou le déchiffrement d'un fichier (*.txt, *.jpg, *.bmp, ...etc).

Cette IHM est montrée sur la figure5.1

Figure 5.1. Menu principal de l'IHM

5.3.1 Configuration du protocole de communication RS232

Cette partie consiste à établir une connexion entre le PC et la carte V2MB1000. Le flux des données est transmis via la liaison RS232. Cette liaison doit être configurée avec les mêmes paramètres (débit de transmission, taille des données, bit de parité et bit stop) dans les deux parties (FPGA, PC). La configuration du port RS232 est montrée sur la figure 5.2

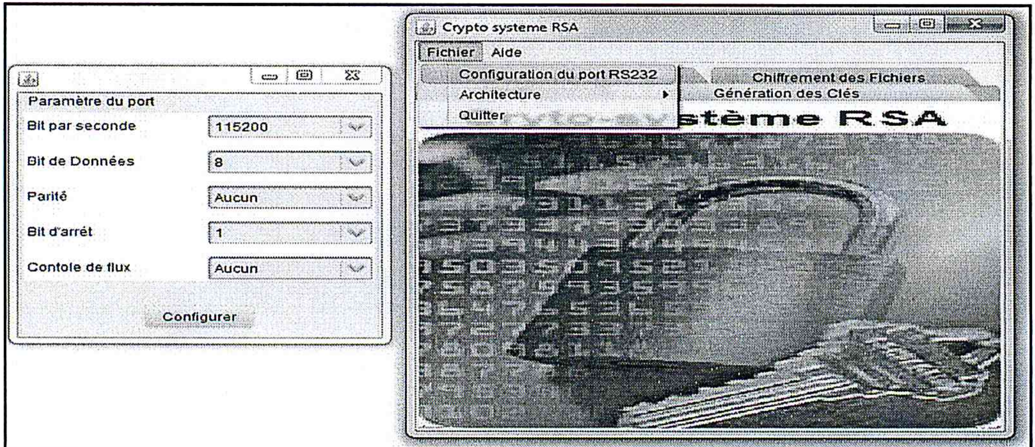


Figure 5.2. configuration du port RS232

5.3.2 Génération des clés

Cette partie assure la génération des clés (E, N), (D, N), en respectant les contraintes du protocole RSA. Celle-ci met aussi à la disposition de l'utilisateur une option pour sauvegarder ces clés dans des fichiers ".txt". Comme ceci est montré sur la figure 5.3.

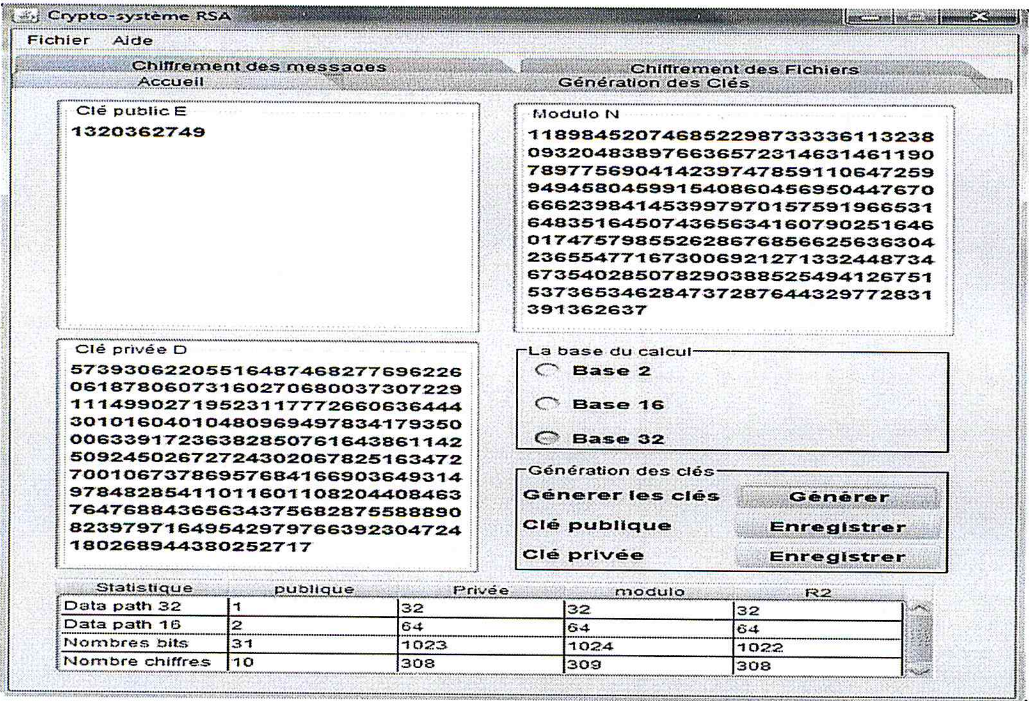


Figure 5.3. Génération des clés

5.3.3 Chiffrement et déchiffrement d'un message

Cette partie est considérée comme étant la fenêtre principale pour le chiffrement ou le déchiffrement d'un message. Elle est constituée de:

- Trois champs de texte. Le premier champ "*Message à chiffrer*" permet d'éditer un message à chiffrer de taille variable. Un deuxième champ "*Message chiffré*" permet de visualiser le résultat de chiffrement. Le troisième champ est associé à l'affichage du résultat de déchiffrement.
- Un tableau dont le rôle est l'affichage du nombre de mots de taille 1024 bits, à savoir 32 bits ou 16 bits qui représentent le nombre de digits du message.
- Quatre boutons en l'occurrence, "Importer la clé publique", " Importer la clé privée", "Chiffrer" et "Déchiffrer" qui permettent de mener les actions à effectuer.
- Deux champs pour afficher le chemin où se trouvent les clés sélectionnées.
- Une partie pour l'affichage des performances temporelles (temps de transmission, temps d'exécution de chiffrement/déchiffrement, temps de réception).

Les valeurs du couple de clés et du modulo sont montrés en annexe A.

Cette partie le l'IHM est montrée sur la figure 5.4.

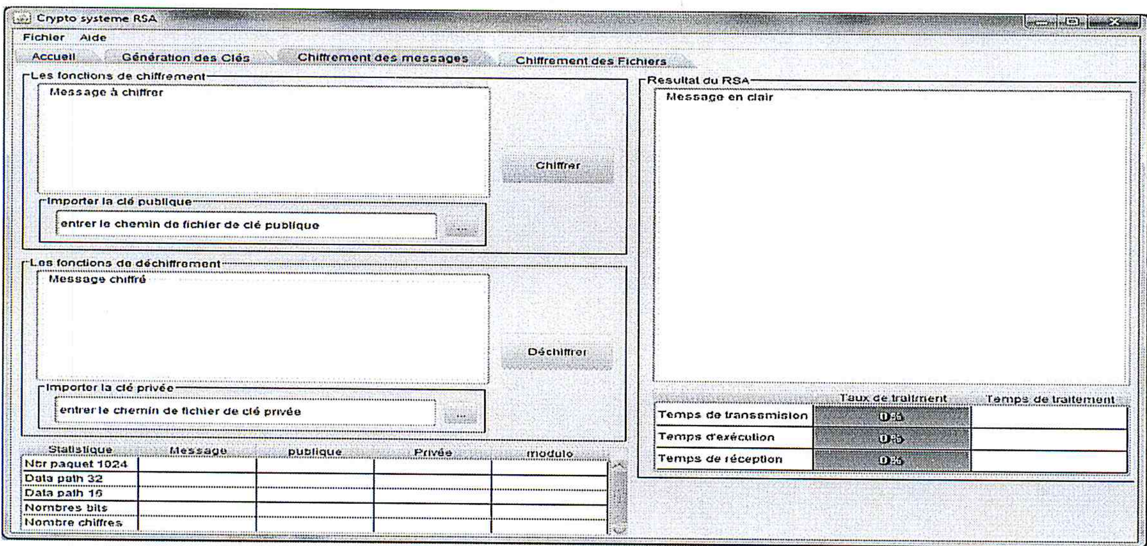


Figure 5.4- Chiffrement et déchiffrement d'un message

5.3.4 Chiffrement et déchiffrement d'un fichier

Cette partie assure le chiffrement et le déchiffrement d'un fichier d'une extension quelconque. Globalement, elle est identique à la partie précédente, à l'exception que les champs destinés au chiffrement et au déchiffrement des messages sont remplacés par des champs pour spécifier les chemins des fichiers à chiffrer ou à déchiffrer. Les champs en question sont repartis comme suit:

- Le champ du chiffrement est composé de quatre boutons dont trois sont destinés respectivement à sélectionner la clé de chiffrement, le fichier à chiffré et le chemin du stockage du résultat de chiffrement (fichier chiffré). Le quatrième bouton "Chiffrer" a pour rôle de transmettre le fichier à chiffrer vers la carte V2MB1000.
- Le champ du déchiffrement est identique au champ du chiffrement, à l'exception que dans ce cas, la clé à indiquer est la clé de déchiffrement.

Cette fenêtre est montrée sur la figure 5.5.

Figure 5.5. Chiffrement et déchiffrement d'un fichier

5.4 Vérification des résultats

Afin de vérifier l'exactitude des résultats de chiffrement ou de déchiffrement fournis par les différentes approches implémentées, une méthode de vérification a été développée. Celle-ci est basée sur la comparaison des résultats obtenus à partir de la carte V2MB1000 avec un modèle mathématique implémenté en langage Java. Ce modèle n'est autre qu'un calcul de l'exponentiation modulaire. Les résultats de cette opération seront comparés avec les résultats fournis par le circuit FPGA. Si cette comparaison donne un résultat différent de zéro, cela signifie que des erreurs existent dans la plateforme implémentée. La procédure de vérification est montrée sur la figure5.6. Le model mathématique utilisé pour la vérification est montré en annexe A.

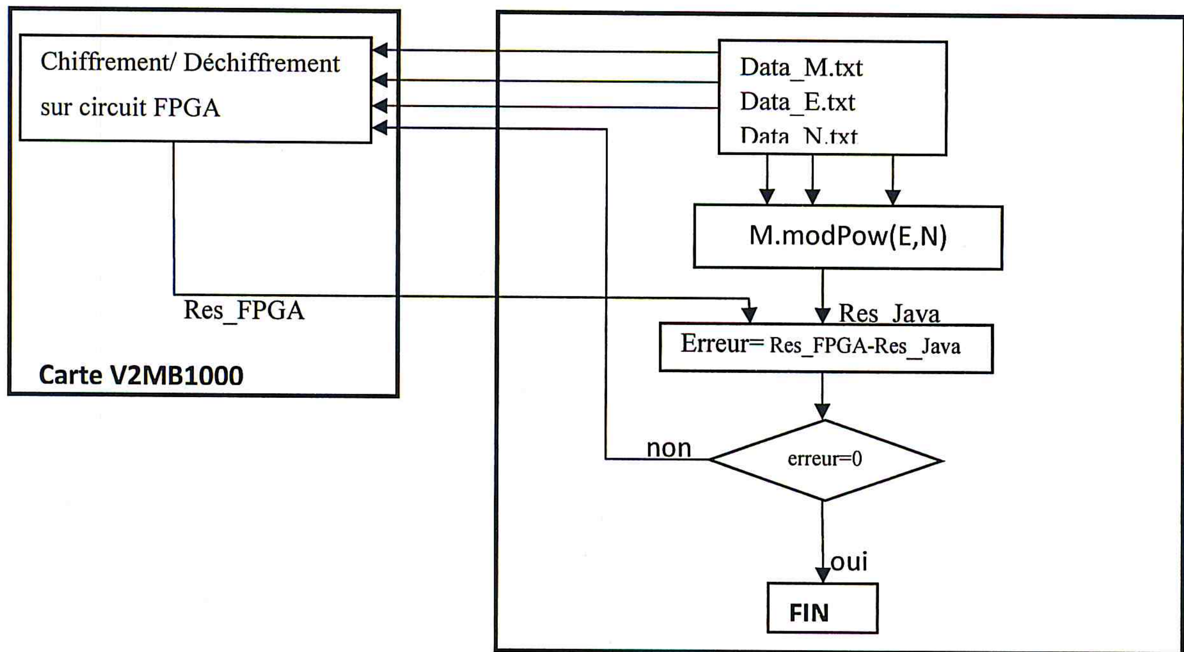


Figure.5.6. Algorithme de vérification fonctionnelle de la plateforme

5.5 Performances temporelles et ressources occupées sur circuit FPGA

5.5.1 Performances temporelles

L'analyse temporelle est une étape cruciale pour la détermination des performances de la plateforme implémentée. En effet, pour chaque approche d'implémentation, le temps d'exécution de la plateforme est déterminé par l'addition du temps de transmission des données, l'exécution de l'exponentiation modulaire et le temps de réception. Pour quantifier le nombre de tops d'horloges nécessaires pour chaque étape (transmission, exécution, réception),

les deux pilotes "XTmrCtr_Stop(&deley,1)", "XTmrCtr_Start(&deley,1)" sont utilisés respectivement au début et à la fin de chaque étape.
Les performances des différentes approches d'implémentations pour le chiffrement et le déchiffrement, d'un message de 1k bits et un modulo de 1024 bits, sont présentées dans les tableaux 5.4 et 5.5.

	Type d'implémentation	fréquence	Temps de transmission	Temps d'exécution	Temps de réception
Microblaze MIN	Exp_Base_2_1	100 Mhz	55,39 ms	18,89 s	9,67 ms
	Exp_Base_2_16	100 Mhz	56,33 ms	1,01 s	9,67 ms
	Exp_Base_2_32	100 Mhz	54,40 ms	732,23 ms	9,67 ms
Microblaze MAX	Exp_Base_2_1	100 Mhz	55,38 ms	10,16 s	9,67 ms
	Exp_Base_2_16	100 Mhz	55 ,54 ms	501,96 ms	9,67 ms
	Exp_Base_2_32	100 Mhz	54,69 ms	188,77 ms	9,67 ms
Microblaze avec IP_MMM	Exp_Base_2_32	33 Mhz	53,79 ms	15,03 ms	9,67 ms

Tableau.5.4. Performances temporelle du chiffrement avec une clé de taille 32 bits

Dans ces tableaux, les notions Microblaze Min, Microblaze Max et Microblaze avec IP_MMM représentent respectivement, Microblaze avec un minimum d'options (sans multiplieur et Barrel Shifter), Microblaze avec options (avec multiplieur et Barrel Shifter) et Microblaze avec IP_MMM.

De même, les différents types d'implémentations effectués sont notés par Exp_Base_2_k où 2_k représente la base de numération des données.
A partir de ces résultats, il est à noter que la base de numérotation des données et le support d'implémentation influent directement sur les performances temporelles du système.

En effet, l'utilisation du processeur Microblaze avec un minimum d'options présente un temps d'exécution le plus élevé par rapport à l'utilisation des options implémentées sur matériel, en l'occurrence la multiplication 32 bits et Barrel Shifter.

A titre d'exemple, en base 2^{32}, le temps d'exécution qui correspond à l'implémentation Microblaze Max, représente 26% du temps d'exécution de l'implémentation Microblaze Min.

	Type d'implémentation	**fréquence**	**Temps de transmission**	**Temps d'exécution**	**Temps de réception**
Microblaze MIN	**Exp_Base_2_1**	**100 Mhz**	**53,72 ms**	**6,55 mn**	**9,67 ms**
	Exp_Base_2_16	**100 Mhz**	**56,43 ms**	**36 s**	**9,67 ms**
	Exp_Base_2_32	**100 Mhz**	**56,03 ms**	**26 s**	**9,67 ms**
Microblaze MAX	**Exp_Base_2_1**	**100 Mhz**	**55,18 ms**	**3,48 mn**	**9,67 ms**
	Exp_Base_2_16	**100 Mhz**	**55 ,68 ms**	**17,59 s**	**9,67 ms**
	Exp_Base_2_32	**100 Mhz**	**55,06 ms**	**6,51 s**	**9,67 ms**
Microblaze avec IP_MMM	**Exp_Base_2_32**	**33 Mhz**	**54 ,80ms**	**526,75 ms**	**9,67 ms**

Tableau.5.5. Performances temporelles de déchiffrement avec une clé de taille 1024 bits

De même, la vérification de la base a montré que l'utilisation d'un bus de données de 32 bits donne un meilleur délai d'exécution. A titre d'exemple, dans l'implémentation Microblaze Max, le temps d'exécution associé à la base 2^{32} représente 37% du délai associé à la base 2^{16}. Ceci est du au fait de l'utilisation optimisée des capacités de Microblaze qui est un processeur de 32 bits, où son datapath est sur 32bits.

En revanche, les performances temporelles qui correspondent à l'implémentation de la multiplication modulaire sur matériel représente 8% du meilleur temps obtenu avec une implémentation purement logicielle, en l'occurrence Microblaze en base 2^{32}.

Il est à noter aussi que les délais associés à la transmission des données et la réception des résultats sont relativement identiques dans les différents cas étudiés. Le débit utilisé pour le fonctionnement de la plateforme développé est de 115200 bps.

5.5.2 Les ressources occupées

Le nombre de ressources occupées par la partie implémentée sur circuit FPGA et les taux d'occupation sont montrés sur le tableau 5.6.

	Slices	Block Select RAM	Mul18x18	I/O
Microbaze Min	1688 (32%)	16 (40%)	0 (0%)	5 (1%)
Microbaze Max	1836 (35%)	16 (40%)	3 (7%)	5 (1%)
Microblaze avec IP_MMM	2597 (50%)	22 (55%)	14 (35%)	5 (1%)

Tableau 5.6. Ressources occupées sur le circuit FGPA

Ces résultats sont quantifiés en termes de nombres de blocs mémoires, de blocs multiplieurs 18×18 et de slices occupés.

A partir de ces résultats, il est montré que le nombre de Multiplieurs est quasiment nul dans le cas d'un Microblaze Min. Ce nombre est respectivement 3 et 14 dans le cas de Microblaze Max et Microblze avec IP_MMM. On constate aussi que le nombre de mémoires a augmenté dans le cas de Microblaze avec IP. Cette augmentation qui est de 6 mémoires est utilisée dans l'IP_MMM pour le stockage des résultats intermédiaires. Le nombre de slices occupés est plus élevé dans le cas où la multiplication modulaire est implémentée sur matérielle (Microblaze avec IP_MMM). En effet, le nombre en question a augmenté de 30% par rapport a l'implémentation avec options (Microblaze Max). En revanche, l'utilisation d'un Microblaze avec un minimum d'options, résulte en un nombre de slices occupés relativement identique au nombre obtenu dans le cas de l'implémentation Microblaze Max. Ainsi, on constate que les options rajoutées (Multiplieur et Barrel Shifter) ne porte pas une grande influence sur le nombre de Slices occupés. L'implémentation de l'architecture matérielle sur le circuit XC2V1000fg465-4 est montrée sur la figure 5.8.

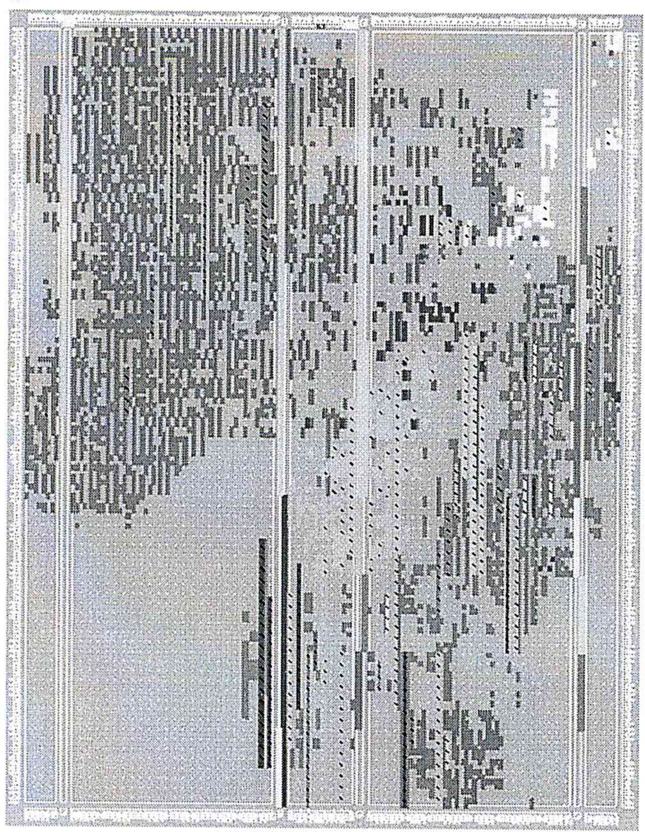


Figure.5.8. Implémentation de l'architecture Microblaze avec IP_MMM sur le circuit XC2V1000fg465-4

5.6. Conclusion

Dans ce chapitre, nous avons présenté la méthodologie de conception d'un SoPC dédié au chiffrement/déchiffrement. Dans ce travail nous avons testé plusieurs configurations matérielles (Microblaze avec et sans options et avec un IP pour l'accélération de l'exécution de la multiplication modulaire). Nous avons montré à travers les tableaux comparatifs que la configuration avec options est plus performante que celle sans options et ceci se justifie par le rôle que joue le multiplieur adjoint au Microblaze dans l'exécution de l'exponentiation. L'autre remarque est que les performances augmente avec l'augmentation de la base (toujours dans la configuration sans IP) est ceci est dû au fait de s'approcher de la taille du datapath du processeur.

La dernière configuration est la meilleure, dans la mesure où dans celle-ci, on adjoint un IP performant qui décharge le processeur du calcul de la multiplication modulaire. Il est à noter que ce type calcul n'est pas adapté aux processeurs classiques et c'est pour cela que l'écart en termes de performances avec les solutions logicielles est important malgré l'utilisation d'une horloge de 33 Mhz au lieu de 100 Mhz. En ce qui concerne la surface occupée la dernière

configuration occupe certes plus de surface, mais la surface additionnelle ne représente même pas la moitié, de la surface de la meilleure configuration logicielle, pour des performances temporelles multipliées par dix. A travers ces résultats, on constate que l'implémentation de notre système embarqué avec une multiplication modulaire sur matérielle engendre un bon compromis entre les ressources occupées et le temps d'exécution.

Conclusion générale

Conclusion générale

L'objectif de notre projet est la conception et l'implémentation d'un crypto système embarqué sur circuit FPGA de Xilinx.

Notre travail concerne deux principaux axes à savoir, la cryptographie moderne et la conception des systèmes embarqués sur circuit FPGA.

De ce fait, nous avons entamé notre projet par une étude des protocoles cryptographiques, en particulier le cryptosystème à clé publique RSA.

En second lieu, nous avons consacré une étude détaillée sur la conception des systèmes embarqués, et plus précisément, l'utilisation des circuits FPGAs de Xilinx comme plateforme d'implémentation.

En effet, le recours aux systèmes embarqués pour l'implémentation des applications complexes présente plusieurs avantages tels que la flexibilité du système, du moment qu'une partie de l'application est exécutée par un processeur embarqué, en l'occurrence le processeur Microblaze.

L'opération de base du cryptosystème RSA est l'exponentiation modulaire qui n'est autre qu'une suite de multiplications modulaires. Les performances d'exécution de ces deux opérations et les supports d'implémentation déterminent les performances du chiffrement/déchiffrement du RSA. Pour cela, nous avons étudié les algorithmes de calcul de l'exponentiation modulaire et de la multiplication modulaire de Montgomery. A travers cette étude, nous avons montré que la base de représentations des données β et les supports d'implémentation, sont les paramètres majeurs qui influent sur les performances d'exécution de cette opération.

Dans ce travail, deux approches ont été implémentées:

1. Une implémentation purement logicielle, où les algorithmes d'exponentiation modulaire et la MMM sont exécutés par le processeur Microblaze pour différentes valeurs de la base, à savoir β=2 et 2^k (k=16 ou 32) afin d'obtenir un meilleur temps d'exécution.
2. Une implémentation de la MMM sur matériel et de l'exponentiation modulaire en logiciel. Cette approche a été utilisée dans le but d'accélérer les calculs des résultats intermédiaires de l'exponentiation modulaire.

Enfin, après l'étude de la méthodologie de conception des systèmes embarqués, à savoir l'outil de développement EDK de Xilinx, une plateforme de chiffrement et de déchiffrement basée RSA a été concue et implémentée.

En effet, l'utilisation de la base de représentation des données comme paramètre d'influence, nous a conduits à concevoir plusieurs configurations de crypto systèmes. La variation de ce paramètre, nous a permis de déterminer l'impact des ressources internes du processeur Microblaze (taille des registres et les opérateurs arithmétiques) sur les performances d'une implémentation purement logicielle et nous avons vus qu'on s'approchant de la taille du datapath du processeur les performances sont meilleures sans matériel additionnel. Et nous avons vu aussi que l'apport de IP était incontestable dans l'augmentation des performances de la meilleure configuration logicielle multiplier par 10.

De plus, une interface de communication homme machine (IHM) a été réalisée dans le cadre de ce travail. Cette dernière rendait les opérations de chiffrement/déchiffrement plus conviviales.

Une dernière étape de ce travail est la vérification fonctionnelle de la plateforme pour les deux approches d'implémentation. Celle-ci a été rendu possible et facile grâce à l'interface IHM graphique qui permet un échange de données faciles entre la carte et le PC.

En termes de perspectives de ce travail, les performances d'exécution du protocole RSA peuvent être améliorées par le dédoublement de l'IP sur matériel (2multiplieurs en parallèles). D'autres pistes peuvent être étudiées qui consiste en l'utilisation de plusieurs processeurs microblaze (maitre esclaves). Plusieurs configurations peuvent découler et ne demandes qu'à être testées.

Annexe

Programme Java pour la génération des clés et des paramètres de la multiplication de Montgomery

```
public void generateKeys(int base){
BigInteger p= null,q= null,N= null,E=null,D=null,r2=null,mp=null,phi=null,
           one=new BigInteger("1");
boolean bool=false;

//-----Génération aléatoire de deux grandes nombres premiers p,q-------------//
while(!bool){
p= BigInteger.probablePrime(512,new Random());
if(p.bitLength()==512){
while(!bool){q=p.probablePrime(512,new Random());
if(q.bitLength()==512){
N=p.multiply(q);
if(N.bitLength()==1024)bool=true;
}}}}

//-------------------- Calcul de la constate de l'Euler phi------------------//
phi= q.subtract(one).multiply(p.subtract(one));//

//------génération de E tel que PGCD(E,phi)=1 et 2<E<phi--------------------//
bool=false;
while(!bool){
E=new BigInteger(32,new Random());
if(E.gcd(phi).compareTo(one)==0 && E.compareTo(new BigInteger("2"))>0 &&
E.compareTo(phi)<0)b=true;
}


//----------------Calcul de la clé de déchiffrement D-----------------------//
D=E.modInverse(phi);//calcul de = 1/E mod(phi) }}
```

Figure.1- Procédure de génération des clés

```
//--Calcul des paramètres de la multiplication de Montgomery de R^2 et N-//
if(base==1)
{r2=one;mp=one;}
else{
if(base16.isSelected()){r2= new BigInteger("2").pow(2080).mod(N);
mp=new BigInteger("-"+N.toString()).modInverse(new BigInteger("65536"));
      }
else {
      r2= new BigInteger("2").pow(2112).mod(N).
 mp=new BigInteger("-"+N.toString()).modInverse(new
BigInteger("4294967296"));
      }
}
```

Figure.2- Procédure de Calcul des paramètres de la multiplication de Montgomery en base 2^{32}

```
public BigInteger chiffrer(BigInteger Message, BigInteger E,
BigInteger N)
      {
      BigInteger C=M.modPow(E,N);
       return C;
      }

public BigInteger déchiffrer(BigInteger C BigInteger D
BigInteger N)
      {
      BigInteger Message= M.modPow(D,N);
       return Message;
      }
```

Figure 3 Figure.2- Procédure du modèle mathématique utilisé pour la vérification fonctionnelle

Valeurs des vecteurs de test

Modulo :

N=107053175198284149400010045256602711118696961445727580092013836328845372
549963204237151896190536244672544975832077571817984520185591876194110050864854522442218725362770476596932243235147111354397399826914958167492150721682812733749313677701818000211867229389458222461170841212862232709855945915557108510798379

Clé du chiffrement :

E = 51383

Clé de déchiffrement :

D=416041269590046548346496038325730210055710959482656529758368389866437001430476076798101748638424823359097487961146105060687948073503722188308118584021944629274906082527657743130621251489083180472019717604871720808213803426246957477636988950526865009483618958356576059251134960188812486506795403
68665811730215

Message en clair:

M=6474426092381699707349775388414002200185272305465478268602060475368781316995641364671095905268770123480130553921413133248423585077246494134930836

36138864317724709951274671584725702528379902941375359683167940301062354442
45620783180619478879378237364538602660000474067090768632900227270269392906
5301198850449

Paramètres de la multiplication de Montgomery en base 2

$R^2 \bmod N$ =2697135911225341240860794710804018171909027822392010330539667713
18508121531114544771873174195323601649677295691922890465515929186672458563
20604561054372967100709413442688377982896233359805751433071007952657513044
46273564258886153072078408485922001205859200578802792275410869957687539552
9731422542895076306388

N\`=1

Paramètres de la multiplication de Montgomery en base 2^{16}

R^2=33637015307423554405700404673379605625139381546388545237314073065031167
70447549201399882335659508212766596050309931815583885524266543284952341425
84582595064502530608944910138284369670509788793219422597797783151734354107
09790968143939650982465685541189527177596632461348208055560455082584510387
60469848074455

N\`=33661

Paramètres de la multiplication de Montgomery en base 2^{32}

R^2=10358103425380659758203608768996505038273686655108210383997428608247516
05972317528481738798653778056682366255617195198617079533029348822899147552
67658621730109223852608804669761020930292038956187336675545215410181550551
00230381863927973690493710485532261865869666499352507718830459521374284434
3636454714026253

N\`=1391035261

1. Description de FPGA (Field Programmable Gate Array)

Un FPGA ou (réseaux logiques programmables) consiste en une matrice carrée de cellules configurables CLB (Configurables Logique Blocs) permettant de réaliser des fonctions combinatoires et des fonctions séquentielles. Tout autour de ces blocs logiques configurables, nous trouvons des blocs d'entrées /sorties IOB (Input Output Blocs) dont le rôle est de gérer les entrées-sorties réalisant l'interface avec l'environnement externe du circuit. La figure 1 présente l'architecture générale d'un circuit FPGA.

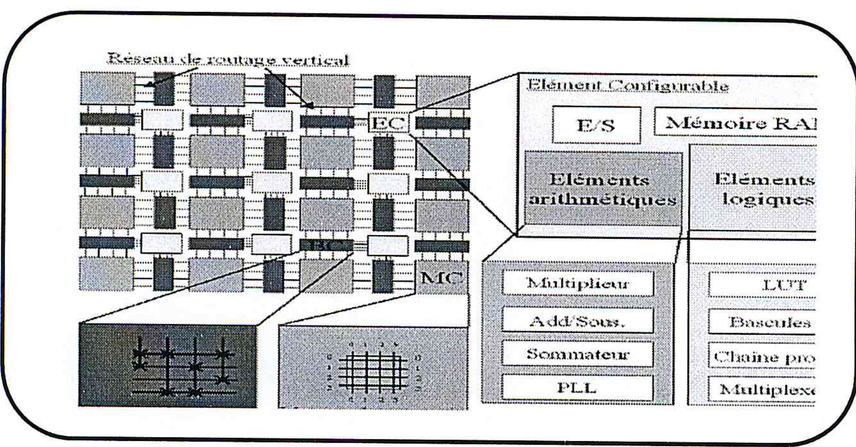


Figure. 1- Architecture générale d'un FPGA

2. Familles des circuits FPGA's

Grâce à l'évolution de technologie de fabrication des circuits intégrés, les FPGAs de Xilinx deviennent plus en plus performants avec des capacités d'intégration, sans cesse en augmentation. Longtemps réalisées autour de blocs logiques à base de LUT (Look Up Table), Les récentes familles des circuits FPGAs de Xilinx peuvent aujourd'hui comporter des mémoires de 18 Kbits, de blocs DSPs et des processeurs. Ces circuits sont caractérisés par une nomenclature spécifique qui définit les performances de chaque famille [23]. Cette nomenclature est montrée sur la figure 2.

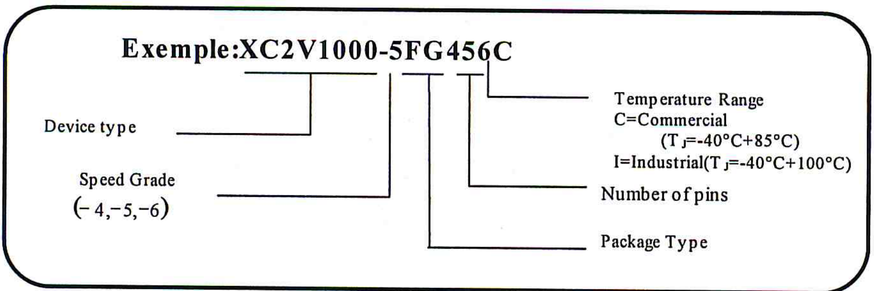


Figure.2- Nomenclature d'un FPGA de Xilinx

3. La famille Vertex-II

La famille Vertex-II a été conçu pour réaliser des conceptions à faible ou grande densité d'intégration et qui exigent des performances élevées (elle peut atteindre jusqu'à 10 million de protes logique). La fréquence de son utilisation peut être portée à 420 Mhz. Actuellement, cette famille est utilisée dans plusieurs application tels que : les réseaux, la télécommunication, la vidéo et les applications DSP (Digital signal Processing).

L'architecture d'un circuit de la famille Virtex-II est montrée sur la figure 3. Celle-ci se présente comme un réseau de bloc logique configurable. Ces derniers sont des blocs Input/Output (IOB) et des blocs logiques configurables (CLB) [23].

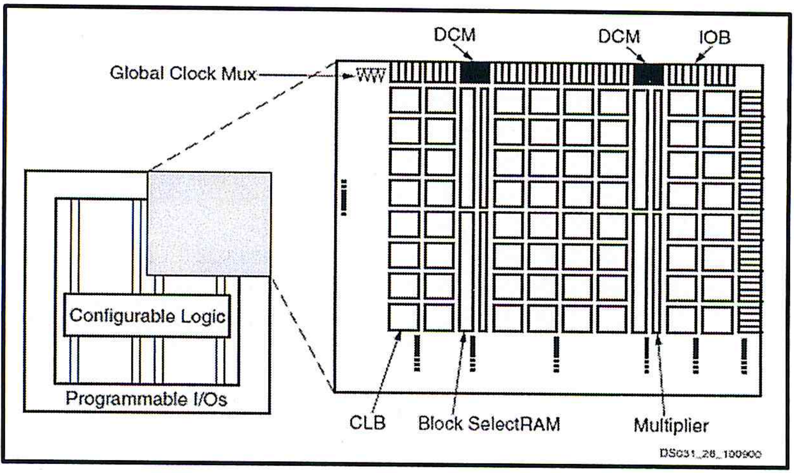


Figure. 3- Architecture interne d'un Vertex-II

3.1 Le bloc logique interne configurable

La logique interne configurable inclus quatre éléments majeurs organisés sous une forme régulière et un chemin dédié à la propagation des retenues. Les éléments en question sont :

- Bloc logique Configurable (CLB).
- Bloc select RAM 18 Kbits à double ports.
- Bloc multiplieur (18bit x18bit).
- DCM (Digitale Clock Manager).

Pour assurer le routage entre ces blocs d'une manière configurable, des ressources d'interconnexion configurables GRM (General Routing Matrix) sont fournies sur le circuit [23].

3.1.1 Bloc d'entrées/sorties (IOB)

Les blocs d'entrée/sortie fournissent une interface entre les broches externe et la logique interne. Les IOBs sont souvent connectée à une même matrice d'interconnexion et ils peuvent être configurés soit en : entrée, sortie, bidirectionnels. Ils incluent six bascules D, leur utilisation est souvent recommandée pour la synchronisation des signaux de communication. Les ressources internes d'un IOB sont montrées sur la figure 4.

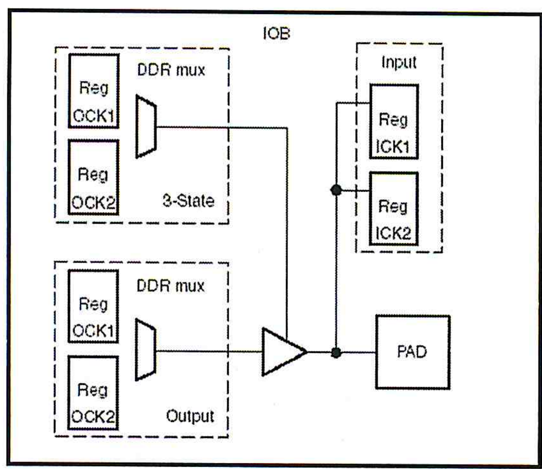


Figure. 4- Bloc IOB de VIRTEX II

3.1.2 Blocs logique configurables

Les CLBs sont implémentés sous forme d'une matrice ($N \times M$), ils incluent toute la logique nécessaire pour la conception des circuits combinatoire et séquentiels. Le CLB était l'élément déterminant des performances des premiers circuits FPGA. Mais ce n'est plus le cas pour le Vertex-II, où on ne parle plus de CLB mais de Slice. Chaque CLB est composé de quatre Slices. Leur connexion au routage globale est assurée par les matrices d'aiguillages GRM.

Chaque Slice est composé de :

- Deux générateurs de fonction F et G à quatre entrés et une sortie.
- Deux éléments de stockage (bascule D).
- Des portes logiques (XOR) dédié à l'implémentation des Full Adders.
- Une logique pour la propagation de la retenue.
- Des multiplexeurs.

Les générateurs de fonction peuvent être configurés comme: des Look Up Table (LUT) à quatre entrées, des RAMs à 16 bits ou des registres à décalage de 16 bits. Ces configurations et le schéma synoptique d'un Slice sont montrés sur la figure 5 [23].

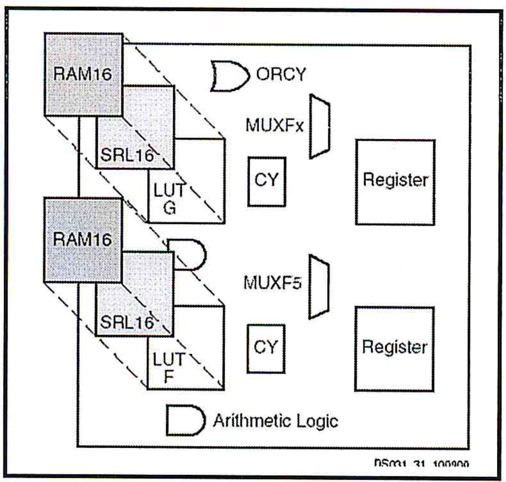


Figure.5- Configuration et schéma synoptique d'un Slice

3.1.3 Bloc Select RAM à 18 kbits

Le Virtex-II inclut dans son architecture un nombre important de blocs mémoire de 18 Kbits. Ceci augmente l'espace mémoire du circuit, si les CLBs sont configurés comme des mémoires. Chaque bloc SelectRAM est une RAM à double ports de taille 2 × 18 Kbits. Ces derniers peuvent avoir indépendamment les signaux d'horlige et de commandes. D'une manière générale, Les blocs SelecRAM sont implémentés suivant l'une des trois configurations suivant :

- Lire après écriture (Read After Write) ;
- Lire avant écriture (Read Before Write) ;
- Lire seulement (Read Only).

Ces blocs mémoires peuvent être utilisés comme des RAMs à simple ou double port, et ils supportent plusieurs configurations. Ces configurations sont déterminées par un compromit entre la taille de la donnée et le nombre d'adresses. Le compromis en question doit respecter la taille globale de la mémoire qui est de 18 Kbit [23].

3.1.4 Bloc Multiplieur 18 bits × 18 bits

La famille Virtex-II est constituée aussi par des blocs Multiplieurs18 bits × 18 bits.

La disposition des blocs multiplieurs se trouve implémentée juste à coté des blocs SelectRAM. Cette technique permet d'augmenter les performances d'une application si les opérandes proviennent d'une mémoire.

Ces blocs multiplieur permettent d'effectue une multiplication sur des opérandes signés ou non signés et avoir un résultat sur 36 bits, ils peuvent aussi avoir des entrées registrées [23].

3.1.5 Les DCMs

Le bloc DCM (Digital Clock Manager) fournit des solutions auto-étalonnées, pour la compensation de l'horloge, la multiplication de l'horloge et sa division, c'est-à-dire, possibilité de diminuer le temps de montée d'une horloge et d'éliminer les pertes dues aux routages. Le DCM utilise un routage particulier qui permet une bonne précision pour le contrôle d'horloge et la fréquence. Les DCMs sont situés au-dessus et au-dessous de chaque bloc Select Ram et multiplieur.

3.1.6 Routage globale et interconnexion

Les ressources de routage global d'un circuit sont optimisées pour d'éventuelles prévisions des performances d'une application donnée. L'accès au routage des différents blocs disponibles sur le circuit (IOB,CLB,Bloc RAM, Multiplieur et les DCMs) est réalisé d'une manière programmables grâce à des matrice d'aiguillage, comme ceci est montré sur la figure 6.

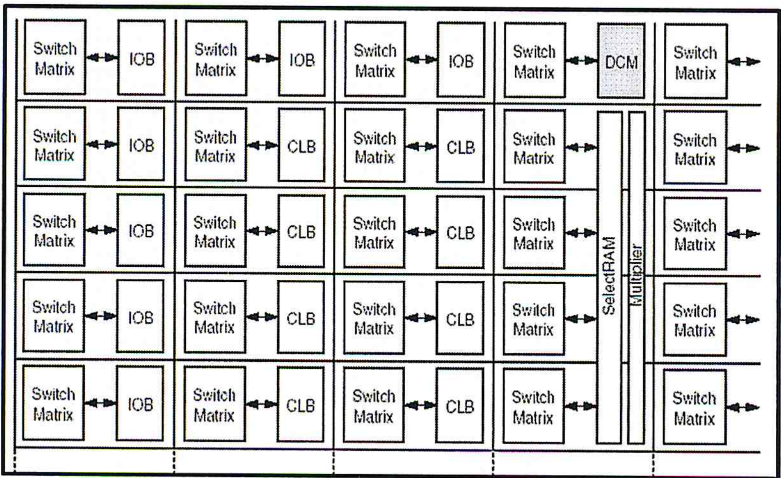


Figure.6- L'interconnexion des ressources interne du circuit FPGA.

3.2. Technique de programmation des FPGA

Les circuits FPGA ne possèdent pas de programme résident. A chaque mise sous tension, il est nécessaire de les configurer. Leur configuration permet d'établir des interconnexions entre les ressources internes utilisées. La configuration du circuit est mémorisée sur la couche réseau SRAM à partir d'une EPROM externe. En effet, un dispositif interne permet à chaque mise sous tension de charger la SRAM interne à partir de l'EPROM. Ainsi on conçoit aisément qu'un même circuit puisse être exploité successivement avec des EPROMs différents puisque sa programmation interne n'est jamais définitive. Le format des données du fichier de configuration est produit automatiquement par le logiciel de développement sous forme d'un ensemble de bits organisés en champs de données [23].

Bibliographie

Bibliographie

[1] P. R. Schaumont (2010) "*A Practical Introduction to Hardware/Software Codesign*", Springer (2010).

[2] R. Laue "*Efficient and Flexible Cryptographic Co-Processor Architecture for Server Application*", Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte (2008).

[3] B.Schneier, "*Applied cryptography Protocols, Algorithms, and Source in C* ", Wiley Second editon, (1995).

[4] "*Introduction à la Cryptographie*". PGP, Version 6.5.1, (1999).

[5] Ç.K. Koç RSA "*Hardware Implementation*", RSA Laboratories Version1.0, (1995).

[6] P.Montgomery, "*Modular Multiplication without Trial Division*", Mathematics of Computation, Vol. 44, pp.519-521, (1985).

[7] D.G.Mesquita, " *Architectures Reconfigurables et Cryptographie : une analyse de Robustesse et Contremesures Face aux Attaques par Canaux Cachés* ", thèse de Doctorat de l'université de Montpellier II (2004).

[8] Certicom Corporation. (1998),"*The Elliptic Curve Cryptosystem for Smart Cards*". Certicom Research, (1998).

[9] D. R. Kuhn, V. C. Hu,W. T.Polk, S.J. Chang, "*Introduction to Public Key Technology and the Federal PKI infrastructure*", NIST, (2001).

[10] A.J. Menezes, Paul C, V. Oorschot, S.A. Vanstone, "*Handbook of Applied Cryptography*", by CRC Press, ISBN: 0-8493-8523-7(1996).

[11] J. Daemen, V. Rijmen "*AES Proposal: Rijmen*",Document version 2, (1999).

[12] Z.E. Alaoui Ismaili, A.Moussa, "*Self-Partial and Dynamic Reconfiguration Implementation for AES using FPGA*",IJCSI International Journal of Computer Science Issues, Vol. 2, (2009).

[13] Ç.K. Koç RSA "*Hardware Implementation*", RSA Laboratories Version1.0, (1995).

[14] A.J. Menezes, Paul C, V. Oorschot, S.A. Vanstone, "*Handbook of Applied Cryptography*", by CRC Press, ISBN: 0-8493-8523-7, (1996)

[15] N. Anane, M. Anane, H. Bessalah, M. Issad, K. Messaoudi, "*Hardwired Algorithm for Variable and Long Precision Multiplication on FPGA*" The Mediterranean Journal of Computers and Networks, vol. 5, no. 4, pp. 132-137, (2009).

[16] P.Montgomery, "*Modular Multiplication Without Trial Division*", *Mathematics of Computation*, Vol. 44, pp.519-521, April (1985).

[17] Florent Bernard "*Etude des algorithmes arithmétiques et leur implémentation matérielle*", Thèse doctorat, Université de Saint-Denis-paris 8. (2007).

[18] R. Ghayoula, E. Hajlaoui, T. Korkobi, M.Traii, H.Trabelsi, "*FPGA Implementation of RSA Cryptosystem*" Transactions On Engineering, Computing and Technogy, Vol 14 (2006).

[19] R.Saleh, S.Wilton, S. Mirabbasi, A.Hu , M. Greenstreet, G. Lemieux, P.P.Pande,C.Grecu, A. Ivanov, "*System-on-Chip: Reuse and Integration*", Proceedings of the IEEE | Vol. 94,No. 6, (2006).

[20] J.P Deschamps, G.J.A Bioul, G.D. Sutter"*Synthesis Of Arithmetic Circuits Fpga, Asic, and Embedded Systems*", A John Wiley & Sons, (2006).

[21] C. Maxfield, "*FPGAs World Class Desogn*",Newnes.

[22] "*Virtex-II, V2MB1000 Development Board User Guide*", Memec Design, Version 3.0, December 2002.

[23] "*Virtex-II 1.5V Field-Programmable Gate Arrays*" DS031-1, v1.7, (2001)

[24] "*MicroBlaze Software Reference Guide*" (2002).

[25] "*Embedded System Tools Reference Manual, Embedded Development Kit*", EDK 9.1i UG111, v7.0, (2007).

[26] "*Tutorial: Designing Custom OPB Slave Peripherals for MicroBlaze*", (2002).

[27] "*OPB IPIF*", DS414,v3.01c, (2005).

[28] A.Ronnholm"*Evaluation of Real-Time Operating Systems for Xilinx MicroBlaze CPU* ", Performed for ABB Corporate Research, (2006).

[29] M.Issad *"Optimisation et Implémentation sur Circuit FPGA de la Multiplication Modulaire pour le RSA", thèse de Master, USTHB, (2011).*