N° : 01/2015-C/ELN

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie
Houari Boumediene

Faculté d'Electronique et d'Informatique

Présentée pour l'obtention du diplôme de DOCTORAT 3ème cycle LMD

Spécialité : Télécommunication et traitement de l'information

par : Issad Mohamed

Sujet

Crypto système embarqué sur FPGA

Soutenue publiquement, le 05/01/2015, devant le jury composé de:

M. S. Chitroub	Professeur à l'USTHB	Président
M. B. Boudraa	Professeur à l'USTHB	Directeur de thèse
M. A. R. Baba Ali	Maitre de conférences A à l'USTHB	Examinateur
M. R. Sadoun	Maitre de conférences A à l'ENP	Examinateur
M. A. Khouas	Maitre de conférences A à l'UMBB	Examinateur

A toute ma famille

Remerciements

Je tiens à remercier Dieu le tout puissant de m'avoir donné le courage et la patience afin de mener à bien ce projet.

Je remercie mon directeur de thèse, Monsieur ***BOUDRAA Bachir****, Professeur à l'Université des Sciences et de la Technologie Houari Boumediene, qui m'a témoigné sa confiance et pour son assistance constante.*

Mes remerciements et mes profondes gratitudes à Monsieur ***ANANE Mohamed****, Maitre de conférences à l'Ecole Nationale Supérieure d'Informatique, pour son assistance, ses encouragements et ses conseils.*

Je remercie Monsieur ***CHITROUB Salim****, Professeur à l'Université des Sciences et de la Technologie Houari Boumediene, pour avoir accepté de présider le jury de cette thèse.*

Mes remerciements aux membres du jury, Monsieur ***BABA ALI Ahmed Riad****, Maitre de conférences à l'Université des Sciences et de la Technologie Houari Boumediene, Monsieur* ***SADOUN Rabah****, Maitre de conférences à l'Ecole Nationale Polytechnique et Monsieur* ***KHOUAS Abdelhakim****, Maitre de conférences à l'Université M'Hamed Bougara, d'avoir accepté d'examiner ce travail.*

Résumé

Dans cette thèse, on s'intéresse à l'implémentation du crypto système à clé publique RSA (Rivest, Shamir and Adleman), basé sur la combinaison des deux ressources logicielle et matérielle. L'opération principale dans ce crypto système est l'exponentiation modulaire qui n'est rien d'autre que le calcul itératif de Multiplications Modulaire (MMs). Notre objectif par ce travail consiste en l'implémentation de la fonction d'exponentiation modulaire dans un environnement PSoC (Programmable System on Chip), où le processeur Microblaze de Xilinx est utilisé pour la flexibilité. Dans le but d'optimiser les performances d'exécution de cette fonction, nos travaux de recherches sont basés sur l'utilisation de l'algorithme R2L (Right to Left) qui repose sur le calcul parallèle de deux MMs. Cet algorithme est souvent utilisé pour améliorer le délai d'exécution de l'exponentiation modulaire. Par ailleurs, lorsque l'optimisation des ressources matérielles sur le support d'implémentation est une contrainte, l'algorithme en question peut être exécuté séquentiellement, en intégrant seulement un seul multiplieur modulaire comme un composant matériel, autour du processeur. De ce fait, le temps d'exécution de l'exponentiation modulaire devient dépendant de trois facteurs: les performances du multiplieur, la chaine de bit non-zéro de l'exposant et le débit de la liaison assurant la communication entre le processeur et le composant. Dans le but d'atteindre le meilleur compromis entre la vitesse d'exécution, les ressources matérielles et la flexibilité du crypto système, nous proposons trois approches d'implémentations. La première exploite un accélérateur matériel dédié à l'exécution d'une seule MM. La seconde approche est basée sur deux stratégies d'optimisation, où deux MMs sont implémentées en parallèle dans un composant matériel et des mémoires locales sont intégrées dans ce dernier, près des unités arithmétiques. La troisième approche est une implémentation purement logicielle. Les résultats d'implémentation sur le circuit FPGA XC5VLX50T-1, ont montré qu'en utilisant le partitionnement logiciel/matériel de la seconde approche, le chiffrement pour une taille de clé de 1024-bits est exécuté en 15,14 ms. Le nombre de slices occupés est de 2315 Slices.

Mots clés : RSA, exponentiation modulaire, multiplication modulaire, multiplication de Montgomery, FPGA, SoC, PSoC, Microblaze.

Tables des matières

Introduction générale

Chapitre 1. Généralités sur la Cryptographie et Protocole de Cryptographie à Clé Publique RSA

Liste des Abréviations

AES: Advanced Encryption Standard.

ASIC: Application-Specific Integrated Circuit.

BRAM: Bloc Memory.

CCP : Cryptographie à Clé Publique.

CMOS: Complementary Metal Oxide Semiconductor.

DCM: Digital Clock Manager.

DES: Data Encryption Standard

DHKE: Diffie–Hellman Key Exchange.

DLMB: Data Local Memory Bus.

DMA: Direct Memory Acces.

DSP: Digital Signal Processor.

ECC: Elliptic Curve Cryptosystems.

EDK: Embedded Development Kit.

FA: Full Adder.

FIFO: First In First Out.

FPGA: Field Programmable Gate Array.

FSL: Fast Simplex Link.

GCC: GNU Compiler Collection.

GDB: GNU Debugger

GPIO: General Purpose Input Output

IHM: Interface Homme Machine.

ILMB: Instruction Local Memory Bus.

IP: Intellectual Property.

IPIF: IP Interface.

IPIC: IP InterConnect.

ISE: Integrated Software Environment.

LCD: Liquid Crystal Display.

LED: Light Emitter Diode.

L2R: Left to Right.

MDM: MicroBlaze Debug *Module*.

MMM: Multiplication Modulaire de Montgomery.

MMMs : Multiplications Modulaire de Montgomery.

MMME : Multiplication Modulaire de Montgomery Entrelacée.

MMMESF : Multiplication Modulaire de Montgomery Entrelacée sans Soustraction Finale.

MMM_Core: Montgomery Modular Multiplier Core.

2MMMs_Core: 2 Montgomery Modular Multipliers Core.

OPB: On-Chip Peripheral Bus.

OS: Operating system.

PSoC: Programmable System On Chip

PGCD: Plus Grand Commun Diviseur.

PLB: Processor Local Bus.

PLD: Problème du Logarithme Discret

RISC: Reduced instruction Set Computer.

RSA: Rivest Shamir Adleman.

RTL: Register Transfert Level.

R2L: Right to Left.

SDK: Software Development Kit.

SoC: System On Chip.

SoPC: System on Programmable Chip

TRC: Théorème des Restes Chinois.

UART: Universal Asynchronous Receiver Transmitter.

UA: Unité Arithmétique.

UAs Unités Arithmétiques.

VHDL: Very High speed integrated circuit Hardware Description Language.

XST: Xilinx Synthesis Tool.

XPS: Xilinx Platform Studio.

Liste des figures

Liste des tableaux

Liste des algorithmes

Introduction générale

1. Contexte de la thèse

Le domaine de la cryptographie est un sujet qui s'est développé fortement ces dernières années, en raison d'un besoin croissant de sécurité dans les échanges commerciaux et plus généralement dans toutes les transactions des données numériques.

Les algorithmes cryptographiques pour être fiables, nécessitent d'importants calculs qui sont souvent proportionnels à la taille de la clé de chiffrement [1], [2], [3], [4]. Ces algorithmes demandent d'effectuer des opérations arithmétiques dans des corps finis [1], [5], [6] qui ne sont pas adaptées aux processeurs généralistes. De plus, avec le développement des réseaux de communications, les schémas cryptographiques sont composés de plus en plus par des fonctions complexes. La conception de tels systèmes sur une même puce ne consiste pas seulement en l'implémentation matérielle de la fonction désirée, mais aussi sa gestion software via un processeur embarqué. Ces systèmes peuvent être implémentés dans des composants spécifiques de type ASIC (Application Specific Integrated Circuit). Le terme utilisé est souvent SoC (System on Chip); ou encore, des implémentations sur des circuits programmable de types FPGA (Field Programmable Gate Array). On parle ainsi de systèmes PSoC (Programmable System on Chip) [5], [7], [8]. D'une manière générale, l'implémentation de ces systèmes consiste à cohabiter les deux ressources logicielle et matérielle sur une même puce [7], [9]. Le logiciel est utilisé pour la flexibilité. La partie matérielle est souvent préconisée à l'implémentation des applications sensibles au facteur temps réel.

Les mécanismes proposés par la cryptographie moderne, permettent de répondre aux besoins de la sécurité de l'information par le déploiement de crypto systèmes symétriques et asymétriques [1]. Dans la cryptographie symétrique, la clé de chiffrement est identique à la clé de déchiffrement. Son problème majeur est la nécessité de partager un grand nombre de clés afin de créer des canaux sécurisés entre chaque membre du réseau. En revanche, son avantage reste dans les performances et la rapidité du chiffrement et du déchiffrement.

La solution pour contourner la gestion des clés a été apportée par la cryptographie asymétrique, dite aussi Cryptographie à Clé Publique (CCP) [10]. Son principe de fonctionnement est basé sur l'utilisation de deux clés, définies par les couples *(E, N)* et *(D, N)*. Le couple *(E, N)* représente la clé publique. Celle-ci est utilisée pour le chiffrement des données. Le second couple est la clé privée, permettant de déchiffrer les données en clair. L'objectif de la cryptographie asymétrique est de rendre possible la publication de la clé de chiffrement, tout en gardant en secret la clé de déchiffrement. Les crypto systèmes à clé publique les plus connus de nos jours sont : le RSA (Rivest Shamir Adleman) [11] et l'ECC (Elliptic Curve Cryptosystems) [12]. Ces crypto systèmes sont généralement construits sur des problèmes mathématiques difficiles à résoudre. Leur sécurité repose sur l'utilisation de grande taille de clés, pouvant varier de quelques centaines de bits jusqu'à des milliers de bits.

Dans le protocole de cryptographie RSA, les opérations de chiffrement et de déchiffrement sont basées sur le calcul d'une exponentiation modulaire, défient par $Y=X^Z \bmod N$. X est la donnée à chiffrer ou à déchiffrer. Z est l'exposant qui peut être E ou D. N est le modulo [1], [11]. Les algorithmes développés dans la littérature pour calculer cette fonction ont montré que sa complexité est réduite au calcul itératif de la Multiplication Modulaire (MM), définie telle que : $(A \times B) \bmod N$ [1], [13], [14].

Le protocole ECC repose sur l'exécution d'une multiplication d'un scalaire *(K)* par un point *(P)* [15], [16]. A un bas niveau d'abstraction, le calcul de $K \times P$ est effectué par une combinaison des quatre opérations de base à savoir, l'addition modulaire, la soustraction modulaire, la multiplication modulaire et l'inverse modulaire [17].

Notre objectif dans cette thèse consiste en l'implémentation du crypto système à clé publique RSA, basée sur le processor softcore Microblaze de Xilinx [18]. Nos contributions concernent principalement l'optimisation des performances de ce crypto système, en termes de délai d'exécution, de ressources matérielles et de flexibilité.

2. Objectifs et contributions

Généralement, pour atteindre une implémentation matérielle ou logicielle du crypto système RSA, il est nécessaire de concevoir les deux parties qui constituent ses principaux niveaux d'abstractions, à savoir génération de clés et calcul d'une exponentiation modulaire. Pour ce faire, l'implémentation que nous proposons dans cette thèse est basée sur :

- la réalisation de la partie "génération des clés" en utilisant le langage Java sur PC,
- l'implémentation de l'exponentiation modulaire sur une carte de prototypage à base de circuit FPGA [19].

Au plus un bas niveau d'abstraction du protocole RSA, la MM est souvent considérée comme étant une opération critique. Une méthode efficace pour calculer cette opération a été proposée par Montgomery [20]. Cette dernière transforme la réduction en modulo *N*, en une série d'additions et de décalages à droite.

Dans ce présent travail, on s'intéresse plus particulièrement à l'implémentation sur circuit FPGA de la MM et de l'exponentiation modulaire. Les objectifs ciblés consistent en :

- L'optimisation de ces opérations en termes de temps d'exécution, de ressources matérielles occupées et de flexibilité.
- Leur implémentation on utilisant la combinaison des deux ressources logicielle et matérielle.

Notre première contribution est liée à l'optimisation de l'algorithme de la Multiplication Modulaire de Montgomery (MMM), afin d'adapter son exécution au ressources internes du processeur Microblaze.

Les performances d'exécution de la MMM dépendent de la taille du modulo et de la base β utilisée pour le codage des données. Lorsque la taille du modulo est élevée, l'implémentation matérielle de cette opération nécessite des additionneurs et des registres de grandes tailles. Ceci conduit à une complexité élevée en termes de temps d'exécution et de ressources matérielles. En effet, pour augmenter les performances de la MMM, plusieurs techniques d'optimisations sont proposées dans la littérature. L'implémentation réalisée dans [21] repose sur le codage de "*Booth"* [22] et sur la représentation des données en notation *"retenue conservée"* [23]. Cette notation est exploitée aussi dans [24] et [25] afin de concevoir un multiplieur modulaire performant. Tenka dans [26] a proposé une unité arithmétique composée d'une chaine de processeurs élémentaires, où toutes les opérations arithmétiques de l'algorithme de la MMM sont effectuées de manière pipeline. D'autres types d'architectures parallèles sont implémentés dans [27], [28] et [29], ces dernières reposent sur les réseaux systoliques afin d'augmenter le débit d'exécution de la MMM. Bien que ces travaux aient conduit à des architectures performantes, celles-ci sont souvent consommatrices de ressources matérielles et elles ne sont pas recommandées aux objectifs ciblés dans cette thèse.

Dans le but d'atteindre un meilleur compromis entre le temps d'exécution et les ressources matérielles, la méthode proposée dans le présent travail pour l'implémentation de la MMM, est basée sur la décomposition des opérandes en chiffres de taille *k-bits*. Autrement dit, toutes les opérandes qui sont mises en exécution sont représentées en base $\beta = 2^k$. Comme

le processeur Microblaze est un *32-bits*, les opérations arithmétiques de l'algorithme proposé sont effectuées en mode série sur une précision de taille *k-bits*, proche de la taille du bus de données du processeur. Pour compenser le faible degré de parallélisme, ces opérations sont implémentées dans une Unité Arithmétique *(UA)*, conçue avec une architecture pipeline.

Par ailleurs, l'étude de la complexité algorithmique de la MMM a montré que cette dernière est exécutée d'une manière itérative où, à chaque itération, un seul chiffre du multiplieur, choisi parmi les deux opérandes *A* ou *B* est mis en exécution. Autrement dit, si le multiplieur est codé sur *n chiffres*, on aura besoin de *n itérations* [30]. Comme le codage des nombres dépend essentiellement de la base β où $\beta=2^k$, par conséquent le paramètre *k* qui détermine la valeur de β peut être considéré comme un élément clé dans les performances d'exécution de la MMM. Son augmentation permet de réduire le nombre de chiffres associés aux codages des opérandes. Ce qui mène à la réduction du nombre d'itérations nécessaires à l'exécution de l'algorithme. Néanmoins, ce type d'optimisation ne conduit pas forcément à l'amélioration du délai d'exécution, car l'algorithme nécessite des multiplications de type *chiffre par opérandes*. Ces multiplications sont incontournables pour le déroulement d'une itération de calcul. De ce fait, dans le but d'accélérer l'exécution d'une itération, notre approche repose sur l'utilisation des blocks DSP48E [31], fournis par Xilinx dans son système de génération d'IPs *(Intellectuals Property)*. Ces blocks sont exploités dans l'architecture interne de l'*UA* afin de réaliser des multiplications *k*×*k-bits*. Pour déterminer l'impact du paramètre *k* sur les performances d'exécution de l'*UA*, trois valeurs ont été étudiées, à savoir, *k=16*, *k=32* et *k=64*.

Notre seconde contribution consiste en l'implémentation de la fonction d'exponentiation modulaire $Y=X^Z \bmod N$ dans un environnement PSoC. Les travaux réalisés dans le cadre de cette thèse pour calculer cette fonction, sont basés sur l'utilisation de l'algorithme R2L (Right to Left) [13] qui repose sur la présentation binaire de l'exposant *Z*. Dans cet algorithme, à chaque itération, une multiplication modulaire et une élévation au carré sont exécutées en parallèle. La multiplication modulaire n'est effectuée que si le $i^{ème}$ *bit* de *Z* est un *″1″* logique. En vertu du calcul parallèle de ces deux opérations, l'algorithme R2L est souvent recommandé pour améliorer le délai d'exécution de l'exponentiation modulaire. En revanche, lorsque l'optimisation des ressources matérielles sur le support d'implémentation est une contrainte, cet algorithme peut être exécuté séquentiellement, en intégrant dans un composant seulement un seul multiplieur modulaire, autour du processeur. De ce fait, le temps d'exécution de l'exponentiation modulaire devient dépendant de trois facteurs: les

performances de ce multiplieur, la chaine de bits "*non-zéro*" de l'exposant et le débit de la liaison assurant la communication entre le processeur et le composant.

Dans le but d'atteindre un meilleur compromis entre la vitesse d'exécution, les ressources matérielles occupées et la flexibilité du crypto système, nous avons réalisé trois approches d'implémentations. La première est basée sur l'implémentation autour du processeur Microblaze d'une seule MMM dans un composant matériel. Le contrôle de l'algorithme de l'exponentiation modulaire est assuré en entier par le processeur.

La seconde approche est considérée comme un défi à relever. En effet, le partitionnement logiciel/matériel proposé est basé sur l'implémentation parallèle de deux MMMs dans un composant matériel. Le contrôle de l'algorithme de l'exponentiation modulaire est exécuté de manière logicielle par le processeur embarqué. De plus, pour réduire la quantité des données transférées entre le processeur et le composant, les résultats intermédiaires de l'algorithme de l'exponentiation modulaire sont stockés dans des mémoires locales, à l'intérieur de notre accélérateur matériel.

La troisième approche est une implémentation purement logicielle où les algorithmes de la MMM et de l'exponentiation modulaire sont exécutés par le processeur Microblaze. Cette approche est utilisée principalement comme un modèle de comparaison, nécessitant le moins de ressources matérielles possibles.

Les approches proposées dans ce travail pour la réalisation de la MMM et de l'exponentiation modulaire ont été validées par des implémentations sur la carte de prototypage Genesys de Digilent [19]. Les opérations de chiffrement et de déchiffrement ont été appliquées à des données stockées sous forme de fichiers dans la mémoire interne d'un ordinateur, en utilisant des clés de taille *1024-bits*.

3. État de l'art

La conception et l'implémentation du crypto système RSA a fait l'objet de plusieurs travaux récents. Parmi ceux-ci, on distingue ceux qui proposent une solution purement matérielle ; ou encore, ceux qui se basent sur l'intégration d'un processeur embarqué pour la flexibilité.

a. Implémentations purement matérielles du crypto système RSA

Dans cette première catégorie [21], [24], [32], [33], [34], [35], les algorithmes de la MMM et de l'exponentiation modulaire sont réalisés sur matériel, sans qu'il y'est de processeur embarqué sur le support d'implémentation. Généralement, ce genre de travaux favorise l'optimisation des performances temporelles, au détriment de la flexibilité du crypto

système. Les optimisations développées peuvent atteindre le plus bas niveau d'abstraction de l'architecture matérielle de la MMM.

Dans [21], une implémentation de l'exponentiation modulaire sur circuit FPGA de Xilinx est présentée. L'architecture de la MMM proposée est basée sur le codage de Booth et sur la représentation des données en notation retenue conservée. Généralement, ce type d'optimisation est recommandé pour accélérer le calcul des résultats intermédiaires de cette opération. Pour réduire les ressources matérielles sur circuit FPGA, l'architecture de l'exponentiation modulaire implémentée, est basée sur l'algorithme L2R (Left to Right) [13]. Ce dernier repose sur l'utilisation d'un seul multiplieur modulaire, pour calculer séquentiellement deux MMMs à chaque itération.

[24] propose une optimisation de bas niveau, où l'implémentation de l'algorithme de la MMM et la représentation de ses résultats intermédiaires sont effectués respectivement en base 2 et en notation *"retenue conservée"*. Dans ce travail, les algorithmes L2R et R2L sont utilisés pour l'implémentation de l'exponentiation modulaire sur circuit FPGA.

La référence [32] présente l'implémentation en base 2^{17} de l'exponentiation modulaire sur FPGA. L'objectif principal de cette implémentation réside dans l'optimisation des ressources matérielles. Les auteurs proposent dans ce travail une approche d'implémentation purement séquentielle. Un seul block DSP48E est utilisé pour exécuter les opérations arithmétiques de l'algorithme de la MMM sur une précision de taille *17-bits*.

E.Öksüzoğlu et al dans [33] présentent une implémentation sur circuit FPGA des algorithmes de l'exponentiation modulaire et de la MMM en base 2^{16}. Ce travail repose sur l'architecture développée par Tenka dans [26] pour augmenter le débit de calcul de la MMM.

Wang et al dans [34] présentent une conception matérielle sur circuit FPGA d'un coprocesseur RSA. En vue d'implémenter ce dernier autour d'un processeur RISC *32-bits*, les auteurs ont développé un jeu d'instructions pour permettre le contrôle du coprocesseur RSA de manière logicielle.

Une implémentation matérielle d'un crypto système hybride combinant la cryptographie symétrique et la cryptographie asymétrique est présentée dans [35]. Dans ce travail, les algorithmes de cryptographie symétrique, en l'occurrence l'AES (Advanced Encryption Standard) et le DES (Data Encryption Standard) sont utilisés pour le chiffrement et le déchiffrement des données. Le crypto système RSA est implémenté pour chiffrer et déchiffrer la clé secrète utilisée dans l'AES et dans le DES.

b. Implémentations basées sur la combinaison logicielle/matérielle du crypto système RSA

La seconde catégorie d'implémentation du crypto système RSA, repose sur l'utilisation d'un processeur embarqué [36], [38], [39], [40], [41], [42], [44], [45] et [46]. Les travaux réalisés dans cette catégorie sont basés sur la conception conjointe des deux ressources, logicielle et matérielle. Dans la majorité des travaux étudiés, l'exponentiation modulaire est intégrée dans un système sur puce comme une boite noire. Le rôle attribué au processeur se limite uniquement à activer ou à désactiver le calcul l'exponentiation modulaire. Contrairement à ces travaux, où le contrôle de l'algorithme d'exponentiation modulaire est exécuté sur matériel, les conceptions que nous proposons dans cette thèse sont basées sur l'exécution de manière logicielle du contrôle en question. Ce dernier est assuré par le processeur Microblaze. Ceci permet, en effet, d'augmenter la flexibilité du crypto système RSA.

Hani et al dans [36] présentent l'implémentation du crypto système RSA dans un SoC basé sur le processeur Nios d'Altera [37]. Dans ce travail, les auteurs utilisent l'algorithme R2L pour l'implémentation de l'exponentiation modulaire comme un module autour du processeur. Dans le but de réduire les ressources matérielles, le système proposé est basé sur l'intégration d'une seule MMM.

[38] et [39] présentent des implémentations SoCs dédiées aux applications cryptographiques sur circuit intégré de technologie CMOS. Ces travaux sont basés sur l'intégration de l'exponentiation modulaire autour d'un processeur RISC *32-bits*.

Dans [40], une variété d'implémentations du crypto système RSA, basée sur le microcontrôleur 8051 est présentée. Dans ce travail, les algorithmes: L2R, R2L et le Théorème des Restes Chinois (TRC) [1] sont utilisés pour l'implémentation de l'exponentiation modulaire sur circuit FPGA. Le contrôle du crypto système est assuré par le microcontrôleur. L'objectif de ces implémentations est de constituer une étude comparative afin d'atteindre un meilleur compromis entre le délai d'exécution et les ressources matérielles occupées. Dans ces implémentations, le TRC est utilisé pour accélérer l'exécution de l'exponentiation modulaire. En effet, par l'utilisation de ce théorème, au lieu de calculer une exponentiation modulaire avec un exposant de taille *e-bits*, deux exponentiations modulaires sont exécutées en parallèle avec deux exposants de taille *e/2-bits*. Certes, cette méthode semble être efficace, puisque le délai d'exécution de l'exponentiation modulaire est optimisé

de *50%*. Cependant, son utilisation exige plus de ressources matérielles, où ces dernières sont multipliées par un facteur de 2.

La référence [41] présente quatre méthodes d'implémentation de l'exponentiation modulaire, basées sur le processeur Nios d'Altera [37]. La première est une implémentation purement logicielle où les algorithmes de la MMM et de l'exponentiation modulaire sont exécutés par le processeur embarqué. La seconde méthode repose sur l'implémentation d'une seule MMM sur matérielle, alors que le contrôle de l'exponentiation modulaire est assuré par le processeur Nios. La troisième méthode utilise l'algorithme TRC pour l'implémentation de l'exponentiation modulaire autour du processeur. La quatrième méthode est une implémentation purement matérielle. Pour accélérer l'exécution de la MMM, toutes ces implémentations sont basées sur l'architecture pipeline proposée par Tenka dans [26].

[42] présente l'implémentation de l'exponentiation modulaire dans une plateforme SoPC (System on Programmable Chip), basée sur le processeur ARM922T [43] pour la flexibilité et sur un circuit PLD (Programmable Logic Device) d'Altera pour les performances. Le partitionnement logiciel/matériel proposé dans cette implémentation repose sur l'exécution sur matériel de l'exponentiation modulaire. Le rôle attribué au processeur embarqué se limite uniquement à activer ou à désactiver le calcul de cette fonction. L'optimisation réalisée dans ce travail pour accélérer le calcul de la MMM, consiste en l'implémentation de l'algorithme de cette dernière en base 2, comme un réseau systolique.

4. Plan de la thèse

Les travaux réalisés dans le cadre de cette thèse sont répartis sur six chapitres, organisés comme suit :

Le chapitre 1 définit les différents types de protocoles utilisés dans la cryptographie moderne. Une étude détaillée est réservée au protocole de cryptographie à clé publique RSA, objet principal des travaux menés dans cette thèse.

Le chapitre 2 est consacré à la présentation des algorithmes de l'exponentiation modulaire et de la multiplication modulaire qui constituent le fondement de base de l'arithmétique utilisée par le protocole RSA. Dans ce chapitre, il sera aussi question de présenter les méthodes proposées dans ce travail pour l'optimisation de ces deux opérations, en vue de leurs implémentations dans un crypto système.

Le chapitre 3 est dédié à l'étude des systèmes sur puce et leur méthodologie de conception sur circuit FPGA.

Le chapitre 4 présente les architectures conçues pour l'implémentation sur circuit FPGA des parties matérielles du crypto système RSA. En particulier, nous allons détailler les architectures des composants réalisés dans les deux premières approches d'implémentations, où la mise en œuvre de l'exponentiation modulaire est basée sur la combinaison logicielle/matérielle.

Le chapitre 5 présente les parties logicielles des trois approches proposées pour l'implémentation de l'exponentiation modulaire. Dans ce chapitre, nous discuterons aussi d'une interface Homme/Machine, implémentée en langage Java pour la configuration du crypto système et la génération des clés.

Le chapitre 6 présente la méthodologie de simulation, élaborée dans ce travail pour vérifier le fonctionnement correct des composants réalisés. Ce chapitre est consacré aussi à la présentation des résultats d'implémentations.

Nous terminerons par une conclusion générale et quelques perspectives.

Chapitre 1

Généralités sur la Cryptographie et Protocole de Cryptographie à Clé Publique RSA

1.1. Introduction

La cryptographie est un outil servant à protéger la confidentialité des données sensibles lors de leur communication sur un canal non sécurisé. Afin de protéger des données en clair, on leur applique des transformations qui les rendent incompréhensibles. C'est ce qu'on appelle le chiffrement. Les données résultantes constituent un cryptogramme. Inversement, le déchiffrement est le procédé qui permet de restituer en clair les données à partir du cryptogramme. Dans la cryptographie moderne, les transformations en question sont issues à partir d'algorithmes cryptographiques. Ces deniers sont généralement construits à base de fonctions mathématiques et logiques. Les performances de ces algorithmes en termes de sécurité et de temps d'exécution dépendent d'un paramètre essentiel, en l'occurrence, la clé utilisée pour le chiffrent et le déchiffrent [2], [3], [47]. Le principe de base d'un système cryptographique est illustré sur la figure 1.1.

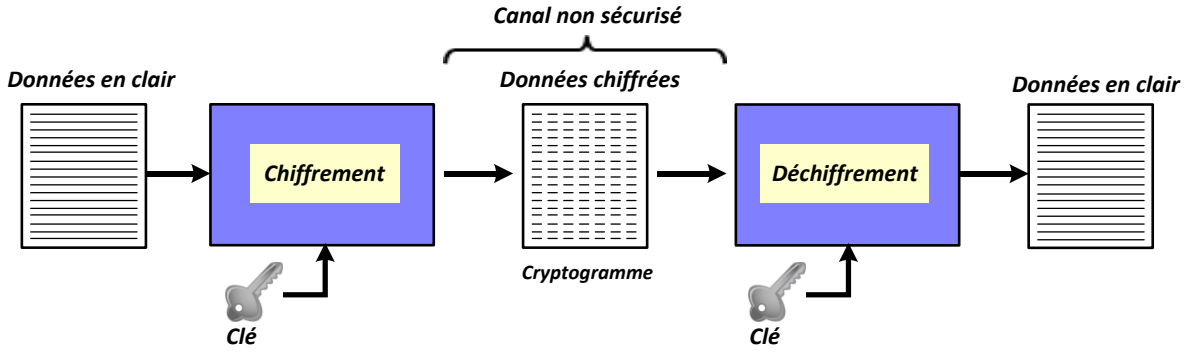


Figure 1.1 - Principe de base d'un système cryptographique.

Casser un cryptogramme consiste à retrouver les données en clair sans détenir la clé secrète, c'est l'objet de la cryptanalyse.

Dans la cryptanalyse, la méthode naïve pour reconstituer en clair un message chiffré, consiste à explorer toutes les variantes possibles de la clé, en vérifiant si l'information qui en résulte a un sens. D'où la robustesse du cryptogramme est proportionnelle à la longueur de la clé de chiffrement. Les clés sont en réalité des nombres extrêmement importants. Elles sont représentées, comme toute donnée informatique, par une suite de *0* et de *1*. Cette longueur peut aller de quelques dizaines de bits à quelques milliers de bits. Plus la clé est grande, plus sa durée de sécurisation est élevée.

D'une manière générale, la sécurité d'un système cryptographique, appelé encore crypto système, dépend non seulement de la taille de la clé, mais aussi de la solidité de l'algorithme qui constitue le crypto système. En effet, si on suppose que l'algorithme est parfaitement solide, c'est-à-dire qu'il n'existe pas de meilleurs moyens pour le casser, la solution d'essayer toutes les clés possibles reste l'alternative unique pour retrouver les données chiffrées. Ce type d'attaque est basé sur un échantillon des données chiffrées et les données correspondantes en clair. La complexité d'une telle attaque est proportionnelle à la taille de clé. En effet, si la clé est de taille *8 bits*, il y'a $2^8=256$ *clés* possibles. Il faudra donc *256 tentatives* pour trouver la bonne clé. De même, si la clé est codée sur *64 bits*, $2^{64} \approx 1{,}8\times10^{19}$ *clés* sont possibles. En faisant l'hypothèse qu'un super-ordinateur peut essayer un million de clés par second, il faudra *570776,2 ans* pour trouver la bonne clé [3].

Ce chapitre est consacré à la présentation des protocoles de cryptographie moderne. En particulier, on s'intéresse au crypto système RSA qui représente l'objectif principal de ce travail. Dans la première partie, des généralités incluant les objectifs et les différents types de cryptographies sont présentées. Ensuite, une étude détaillée est consacrée au protocole RSA et à ses différents niveaux d'abstraction. Dans ce chapitre, il sera aussi question d'aborder quelques méthodes de cryptanalyse dont l'objectif est de casser la sécurité du RSA sans détenir la clé privée.

1.2. Objectifs de la cryptographie

Les objectifs fondamentaux de la cryptographie sont la confidentialité, l'intégrité, l'authenticité et la non répudiation [1], [2], [3].

- **Confidentialité**

La confidentialité permet de protéger le contenu des informations sauvegardées ou transmises à travers un canal de transmission. Seules les personnes autorisées doivent pouvoir accéder aux informations ainsi protégées.

- **Intégrité**

Le problème de l'intégrité est le contrôle de toute modification, accidentelle ou intentionnelle des données sauvegardées ou transmises. Ce genre de problème est résolu à l'aide des fonctions de hachage qui, à une donnée de taille arbitraire, associe une donnée de taille fixe appelée empreinte. Une fonction de hachage est à sens unique, c'est à dire qu'il est impossible, étant donné une empreinte, de trouver les données d'origine.

- **Authenticité**

L'authentification qui est la signature numérique d'un document électronique, permet de lier le contenu du document à son signataire, en assurant à la fois l'intégrité du document (le document n'a pas été modifié) et l'authenticité du signataire. La vérification de la signature est publique: n'importe qui peut la vérifier en utilisant la clé publique du signataire.

- **Non répudiation**

C'est le corollaire direct de l'authentification. Celui qui a transmis des données, non modifiées et signées, ne pourra en aucun cas prétendre par la suite qu'il n'en est pas l'expéditeur.

La confidentialité est généralement fournie par le chiffrement tandis que les trois dernières sont fournies par une signature numérique.

1.3. Différents types de cryptographies

Les mécanismes utilisés par la cryptographie moderne sont basés essentiellement sur trois types de protocoles, à savoir, les protocoles symétriques, les protocoles asymétriques et les protocoles hybrides.

1.3.1. Protocoles de cryptographie symétriques

Dans les algorithmes de cryptographie symétriques, la clé de chiffrement est identique à la clé de déchiffrement, comme ceci est montré sur la figure 1.2.

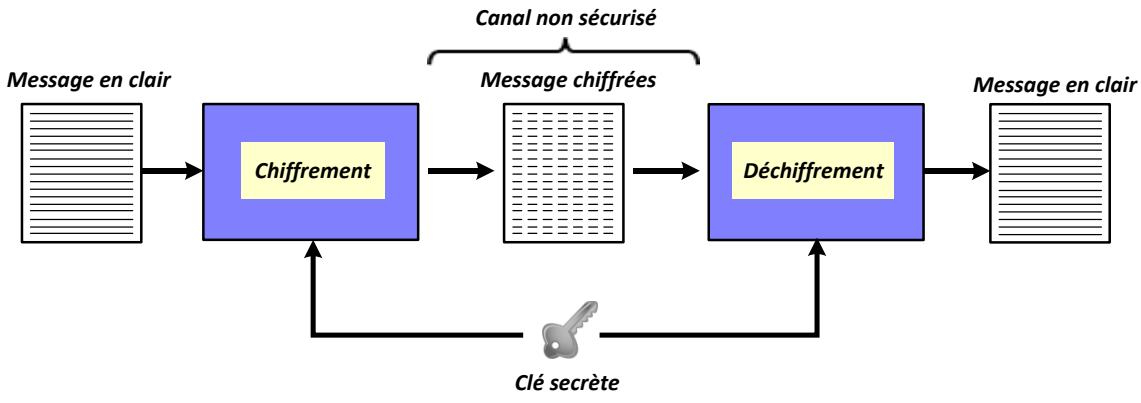


Figure 1.2. Schéma synoptique du chiffrement et de déchiffrement symétrique

Ainsi, c'est la même clé qui est utilisée à la fois pour chiffrer un message et de permettre aux destinataires de le déchiffrer [1], [4]. Cela ne va pas sans poser un problème majeur : l'échange préalable de la clé entre les utilisateurs. Ceci est particulièrement difficile à réaliser puisque, tant que la clé n'est pas transmise, il n'existe pas de moyen sûr d'échange d'information.

Le deuxième problème est le nombre de clés nécessaire pour sécuriser un réseau de communication. En effet, si l'on désire que chaque utilisateur du réseau puisse communiquer avec un autre utilisateur de manière sécurisée, une clé différente est alors utilisée pour chaque paire d'utilisateurs du réseau. Le nombre total de clés croît suivant un polynôme de degré 2. Si un groupe est constitué de *10 utilisateurs*, il sera question de mettre en jeu *45 clés* différentes. De même, pour *100 utilisateurs*, *4950 clés* sont nécessaires. Par conséquent, dans un environnement où il existe *n individus*, pour assurer une communication entre chacun des membres, il est nécessaire d'avoir $(n^2-n)/2$ *clés* [4], [10]. En revanche, l'intérêt majeur de ce type de protocole est que les algorithmes de chiffrement sont constitués d'opérations extrêmement simples. Le crypto système de chiffrement symétrique le plus utilisé de nos jours est l'AES (Advanced Encryption Standard) [48], [49]. Ce dernier supporte trois tailles de clés à savoir, *128 bits*, *192 bits* et *256 bits*. Son exécution repose principalement sur des décalages de bits, des combinaisons par l'utilisation d'opérateurs logiques *XOR* et des permutations de bits, etc.

1.3.2. Protocoles de cryptographie asymétriques

Les problèmes de distribution des clés sont résolus par la Cryptographie à Clé Publique (CCP). Ce concept a été introduit par Whitfield Diffie et Martin Hellman en 1976 [10].

La cryptographie à clé publique est un procédé asymétrique utilisant une paire de clés. Une clé publique pour le chiffrement des données et une clé privée pour le déchiffrement. La clé publique peut être ainsi distribuée, tout en conservant la clé privée. Tout utilisateur possédant une copie de la clé publique, peut crypter des informations que seul le destinataire possédant la clé privée est capable de les avoir en clair [1], [4]. Le schéma synoptique d'un chiffrement et déchiffrement asymétrique est montré sur la figure 1.3.

La cryptographie à clé publique présente un avantage majeur. En effet, elle permet d'échanger des messages de manière sécurisée sans aucun dispositif de sécurité. L'expéditeur et le destinataire n'ont plus besoin de partager des clés secrètes via une voie de transmission sécurisée.

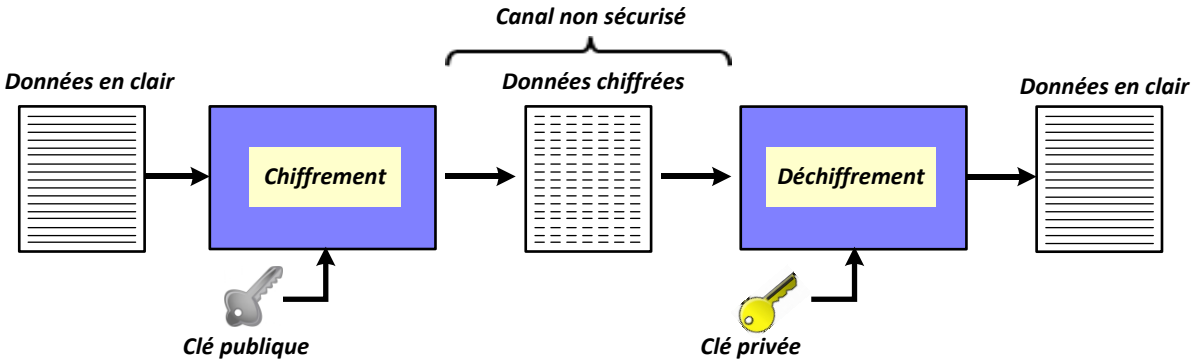


Figure 1.3. Schéma synoptique du chiffrement et de déchiffrement asymétrique

Les protocoles de cryptographie à clés publiques sont généralement construits sur des problèmes mathématiques difficiles à résoudre. En effet, toute leur sécurité est basée sur le problème de résoudre le logarithme discret [3], [4]. Ce qui a donné naissance au crypto système RSA [11]. Ce dernier est souvent adapté pour transporter les clés du chiffrement symétrique, ou pour le chiffrement des données de faible taille. Ceci est dû à sa complexité algorithmique qui croît avec la taille de la clé. De nos jours, les tailles des clés recommandées sont de l'ordre de *1024-bits* et plus.

1.3.3. Protocole d'échange de clés de Diffie-Hellman

Whitfield Diffie and Martin Hellman proposent le premier protocole de cryptographie à clé publique, nommé DHKE (Diffie–Hellman Key Exchange). A l'origine, ce protocole fut inventé pour résoudre le problème d'échange des clés utilisées dans la cryptographie symétrique. C'est-à-dire, si *A* et *B* désirant échanger des informations secrètes à travers un canal non sécurisé, ils peuvent utiliser ce protocole pour engendrer une clé secrète [3], [10].

Le concept de base du protocole DHKE repose sur le calcul de l'exponentiation modulaire qui est une fonction commutative dans un corps fini. Celle-ci est définie par l'équation (1.1).

$$(a^x)^y mod\ N = (a^y)^x mod\ N \qquad (1.1)$$

a, x et y sont strictement inférieurs à *N*

Le protocole DHKE entre deux utilisateurs *A* et *B* fonctionne comme suit :

1. *A* et *B* choisissent un grand nombre *N* et un entier *a*, tel que $a < N$.
2. *A* choisit un nombre *x* et calcule $a^x\ mod\ N$.
3. *A* transmet le résultat de $a^x\ mod\ N$ à *B*.
4. *B* choisit un nombre *y*, tel que $y < N$ et calcule $a^y\ mod\ N$.

5. *B* transmet le résultat $a^y \ mod \ N$ à *A*.
6. *A* calcule $(a^y)^x \ mod \ N$.
7. *B* fait le calcul analogue et obtient $(a^x)^y \ mod \ N$.
8. *A* et *B* gardent leurs exposants en secret.

De ce fait, il est clair que le résultat de l'exponentiation modulaire $a^{xy} \ mod \ N$ est une quantité commune entre les deux utilisateurs *A* et *B*. Ces derniers peuvent utiliser ce résultat pour constituer une clé secrète entre eux, puisque, chacun des deux utilisateurs choisit son propre exposant secrètement. Le concept du protocole DHKE est montré sur la figure 1.4.

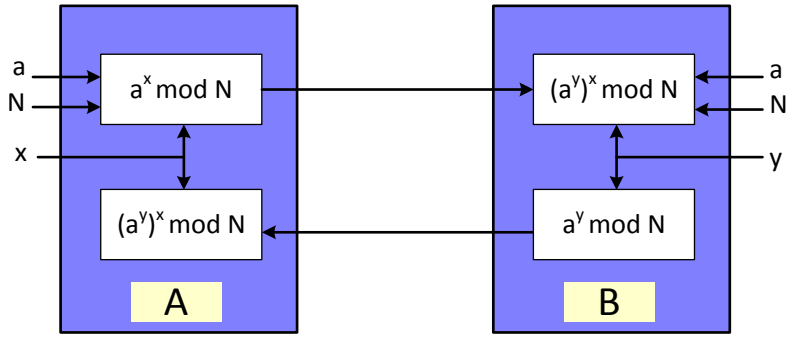


Figure 1.4. Principe du protocole de Diffie et Hellman

La sécurité de ce protocole dépend de la difficulté de calculer le logarithme discret qui est la fonction inverse de l'exponentiation modulaire. Par rapport à cette dernière, il n'existe aucun algorithme qui permet de résoudre le Problème du Logarithme Discret (PLD) en un temps rapide [3]. Le PLD est défini comme suit :

Considérons deux entier *a* et *b*, où : $0< a, \ b< N$. Le PLD constitue la difficulté de déterminer *x* tel que [1]:

$$b = a^x mod \ N \tag{1.2}$$

L'entier *x* est appelé le logarithme discret de *b* en base *a*. La seule méthode qui permet de calculer *x* consiste à essayer toutes ses différentes valeurs. Cette méthode constitue une attaque contre le protocole *DHKE*. Elle est nommée par : *" Brute Force Attack "*.

1.3.4. Protocole de cryptographie hybride

La cryptographie hybride est une combinaison entre la cryptographie symétrique et la cryptographie asymétrique [3], [47]. La combinaison en question permet non seulement une meilleure sécurité, mais aussi, de chiffrer et de déchiffrer rapidement les données.

Le fondement de la cryptographie hybride est basé sur les caractéristiques des deux premiers protocoles. En effet, les crypto systèmes à clé publique sont relativement lents

comparés aux crypto systèmes symétriques où ces derniers sont généralement *1000 fois* plus rapides [1]. En revanche, la cryptographie asymétrique permet une meilleure gestion des clés. Le chiffrement et le déchiffrement hybrides sont représentés respectivement sur les figures 1.5 et 1.6.

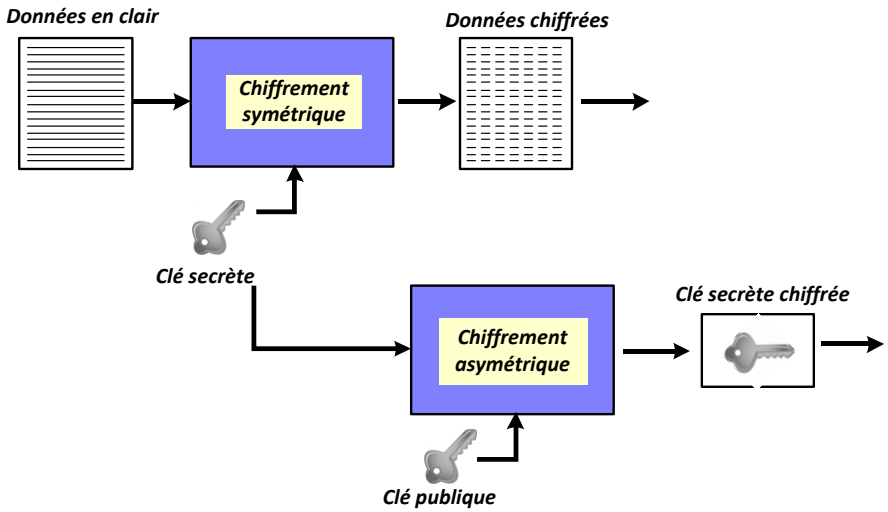


Figure 1.5. Chiffrement hybride

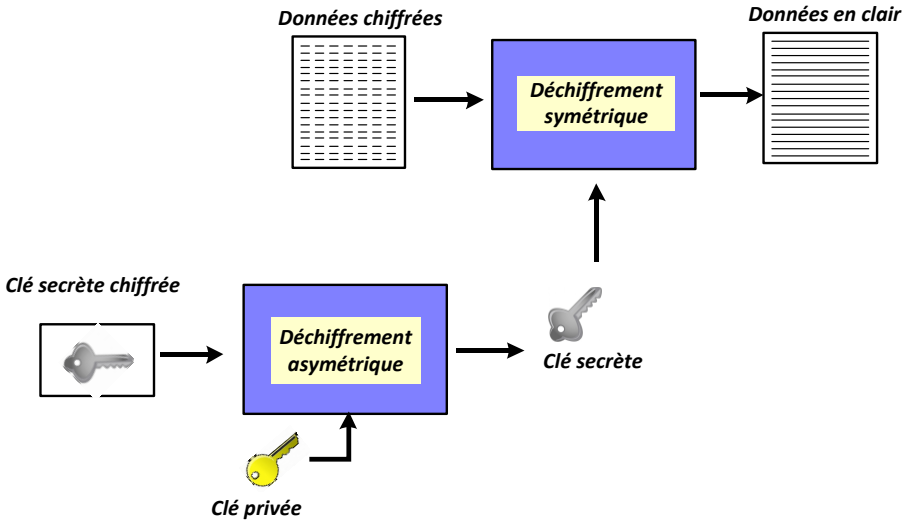


Figure 1.6. Déchiffrement hybride

Le fonctionnement d'un crypto système hybride est résumé comme suit :

Considérons, en premier lieu, le processus du chiffrement. Pour crypter des données en clair, ces dernières sont chiffrées par l'utilisation d'un algorithme symétrique. La clé secrète utilisée est ensuite cryptée avec la clé publique du destinataire (chiffrement asymétrique). Elle sera transmise avec les données chiffrées au destinataire. Le processus de déchiffrement est inverse. Le destinataire utilise sa clé privée pour récupérer la clé secrète qui permettra ensuite de décrypter les données en clair.

1.3.5. Signature numérique

L'un des services qu'offre la cryptographie à clé publique est la signature numérique. Cette dernière permet en effet de vérifier l'authenticité d'un message et son origine. La signature numérique fournit également une fonctionnalité de non répudiation, afin d'éviter que l'expéditeur ne prétende qu'il n'a pas envoyé les informations [3], [47]. Le schéma de base pour la création d'une signature numérique est montré sur la figure 1.7.

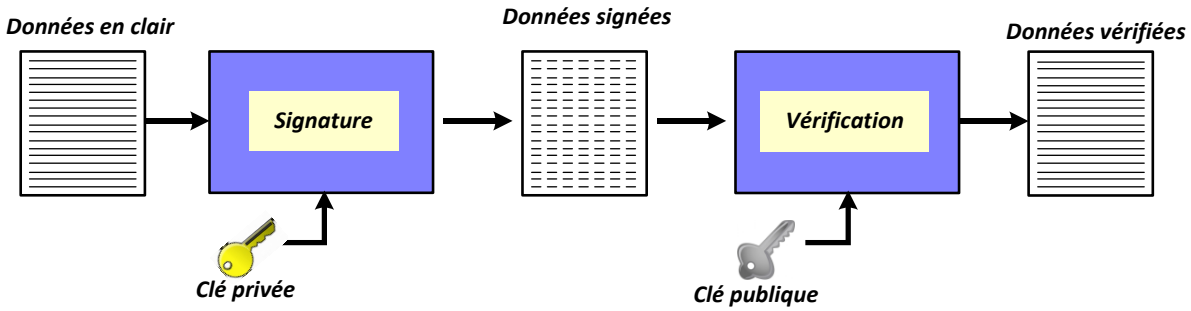


Figure 1.7. Schéma synoptique d'une signature numérique

Plutôt que de chiffrer des données en utilisant la clé publique, un expéditeur doit les chiffrer avec sa clé privé. Si les informations peuvent être déchiffrées avec la clé publique associée, ceci confirme l'origine de la transmission des données.

1.4. Protocole de cryptographie à clé publique RSA

Parmi les protocoles de cryptographie à clé publique, le RSA est le plus populaire. Nommé d'après ses trois inventeurs Ron Rivest, Adi Shamir et Leonard Adleman [11]. Ce protocole repose sur l'emploi de méthodes mathématiques issues de la théorie des nombres. Les clés publique et privée sont calculées en fonction de grands nombres premiers de *512-bits* ou plus. Pour obtenir les données en clair à partir de la clé publique et des données chiffrées, cela nécessite la factorisation du produit de deux grands nombre premiers. De ce fait, l'algorithme RSA est devenu un standard de facto dans la majorité des réseaux de communications.

Dans l'algorithme RSA, on substitue à l'arithmétique ordinaire des mécanismes de calcul en arithmétique modulaire. Les nombres manipulés sont strictement inférieurs à la valeur du modulo *N*. Autrement dit, les résultats des opérations arithmétiques sont des éléments d'un corps fini, noté par Z_N, où $Z_N = \{0,1,.....N-1\}$ [1], [5].

En utilisant ce type d'arithmétique, dès que le résultat *X*, issu des opérations arithmétiques, atteint ou dépasse la valeur d'un entier *N*, on le remplace par le reste *Y* obtenu de la division de *X* par *N*. Ceci est exprimé par : *X* est congru à *Y modulo N*. D'une manière générale, on dit que *X* est congru à *Y modulo N*, s'il existe un entier *k* dans Z_N tel que :

$$X - Y = k \times N \tag{1.3}$$

Dans la littérature, la relation reliant les trois entiers *X*, *Y* et *N* est souvent symbolisée par les équations (1.4) et (1.5).

$$Y = X mod\ N \tag{1.4}$$

$$X \equiv Y\ mod\ N \tag{1.5}$$

Le crypto système RSA fonctionne avec une paire unique de clés [1], [11]. Celle-ci est constituée par une clé publique *(e, N)* pour le chiffrement et une clé privée *(d, N)* pour le déchiffrement**.**

L'utilisation de ce protocole pour le chiffrement/déchiffrement, nécessite une étape initiale, où les deux clés publique et privée sont générées.

1.4.1. Génération des clés

La génération des deux clés est résumée dans les étapes suivantes :

1. Deux grands nombres premiers *p* et *q* sont choisis en premier lieu, pour calculer ensuite le modulo *N* tel que :

$$N = p \times q \tag{1.6}$$

2. Calcul de la fonction d'Euler [1]: $\Phi(N) = (p-1) \times (q-1)$. La valeur de $\Phi(N)$ va servir à déterminer les valeurs des deux entiers *e* et *d*.

3. *e* est choisi tel que :

$$\begin{cases} 2 < e < \Phi(N) \\ \text{et} \\ e \text{ est premier avec } \Phi(N) \end{cases}$$

4. Calcul de *d* qui est l'inverse de *e modulo* $\Phi(N)$. Sa valeur est obtenue en utilisant l'expression (1.7).

$$e \times d = 1\ mod\ \Phi(N) \tag{1.7}$$

e et *d* représentent respectivement les exposants publique et privé.

5. Le couple *(e, N)* est rendu publique. *d*, $\Phi(N)$, *p* et *q* sont gardés en secret.

1.4.2. Chiffrement des données suivant le protocole RSA

Le chiffrement d'un message *M* repose sur le calcul d'une exponentiation modulaire. Celle-ci est donnée par l'équation (1.8).

$$C = M^e mod\ N \tag{1.8}$$

Ainsi, comme le module *N* et l'exposant *e* sont connus, alors toute personne possédant le couple *(e, N)* peut chiffrer un message.

L'utilisation d'une arithmétique adaptée dans le protocole RSA, en l'occurrence l'arithmétique modulaire exige une condition sur la valeur numérique du message *M*. En effet, celle-ci doit être inférieure à *N*. Dans le cas contraire, c'est-à-dire si $M \geq N$, *M* doit être décomposé en blocs, puis chaque bloc sera chiffré séparément.

1.4.3. Déchiffrement des données suivant le protocole RSA

La clé privée *(d, N)* est utilisée pour déchiffrer le message *M* en clair, en utilisant l'exponentiation modulaire suivante:

$$M = C^d mod\ N \tag{1.9}$$

Seule, la personne possédant l'exposant privé *d* peut donc déchiffrer le message. Le principe de base du chiffrement et de déchiffrement RSA est représenté dans la figure 1.8.

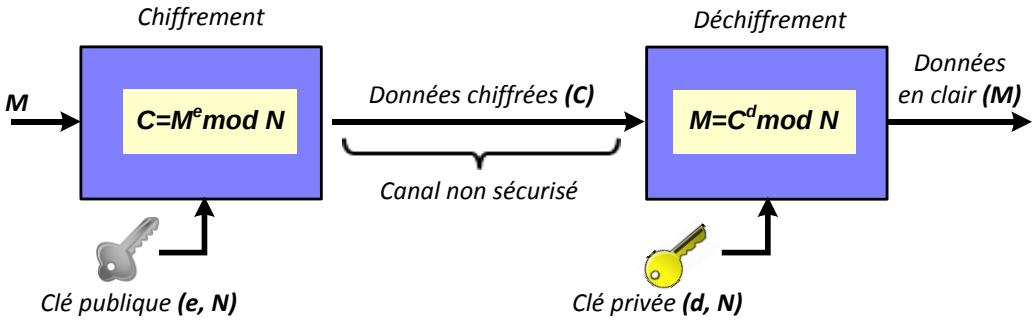


Figure 1.8. Principe de fonctionnement du crypto système RSA

1.5. Sécurité du protocole RSA

La sécurité du protocole RSA repose sur la difficulté de factoriser des nombres représentés sur une grande chaine de chiffres. En effet, la factorisation de *N* qui est le produit de deux grands nombres premier *p* et *q* (voir l'expression (1.6)), permet de calculer la fonction de l'Euler *Φ(N)*, où *Φ(N)=(p-1)×(q-1)*. Par conséquent, puisque *e* est publique, cela conduit au calcul de l'exposant privé *d* par l'utilisation de l'équation (1.7) [11].

Les différentes méthodes qui sont proposées dans la littérature [50] pour réaliser la factorisation d'un grand nombre entier, ont montré qu'il n'existe pas d'algorithme efficace, permettant d'effectuer cette opération en un temps raisonnable. De ce fait, les efforts réalisés

dans le domaine de la cryptanalyse tendent à trouver d'autre types d'attaque, en se basant sur les faiblesses du protocole, où encore, sur les supports d'implémentations.

D'une manière générale, les attaques les plus connues sont classées en trois types de famille et qui sont [1] :

- Attaques basées sur le protocole.
- Attaques mathématiques.
- Attaque par canal auxiliaire.

1.5.1. Attaques basées sur le protocole

Ces attaques exploitent les faiblesses du protocole RSA pour atteindre le message en clair. Les faiblesses en question sont liées soit à la taille de l'exposant privé *d*, ou à la valeur du modulo *N*.

En effet, si l'exposant *d* est de faible taille, le message *M* peut être obtenu par l'utilisation de la méthode *"Brute-Force-Attack"*. Cette méthode consiste à essayer toutes les combinaisons possibles de l'exposant *d* et de vérifier ensuite si le résultat a un sens. De ce fait, le choix de la taille de l'exposant *d* est une contrainte incontournable pour assurer un maximum de sécurité. De plus, si plusieurs utilisateurs possèdent la même valeur du modulo *N* dans leurs clés publiques, ils peuvent facilement calculer l'exposant privé *d* de chacun d'eux. Cette attaque nommée *"Attaque du modulo en commun"* [2], [51] est décrite comme suit :

Notons par (e_1, N) et (e_2, N) les clés publiques de deux utilisateurs *A* et *B*. Si le couple (d_1, N) est la clé privée de *A*, ce dernier par l'utilisation de l'équation (1.7) est capable de calculer la fonction d'Euler $\Phi(N)$. Ainsi, si on considère que le couple (d_2, N) est la clé privée de *B*, l'utilisateur *A* par l'intermédiaire de la clé publique (e_2, N) pourra facilement calculer la valeur de d_2. En vertu de ce type d'attaque, il est recommandé de ne pas transmettre un message chiffré pour plusieurs destinataires, en utilisant un même modulo.

1.5.2. Attaques Mathématiques

Les attaques mathématiques sont basées sur les méthodes qui permettent la factorisation des entiers. En d'autre termes, elles consistent à calculer les deux entiers *p* et *q* à partir de *N*, où $N = p \times q$.

Dans la littérature, plusieurs méthodes sont proposées pour effectuer cette opération [51]. L'ensemble de ces méthodes sont généralement basées sur le calcul de quelques fonctions dont la complexité est assez importante, telle que la division et la racine carré. Cette

complexité peut encore devenir intraitable lorsque l'entier *N* est représenté sur une grande chaine de chiffres.

En revanche, le développent progressif qu'a connue la technologie de fabrication des unités de calcul, a permis de relever des défis caractérisant la force du protocole RSA où quelques factorisations ont pu être réalisées. Le tableau 1 illustre les records atteints entre les années 1991 et 2005.

Tableau 1. Factorisations réalisées entre 1991 et 2005[1]

Taille du modulo en décimal	Taille du modulo en binaire	Date
100	330	Avril 1991
110	364	Avril 1992
120	397	Juin 1993
129	426	Avril 1994
140	463	Février 1999
155	512	Aout 1999
200	664	Mai 2005

De nos jours, la taille du modulo qui est utilisée dans diverses applications de cryptographie est de *1024-bits*. En effet d'après [1], le temps nécessaire pour réaliser la factorisation d'un modulo de cette taille est de l'ordre de 10 années.

1.5.2.1. Factorisation par calcul de division

Cette méthode est la plus simple approche pour factoriser l'entier *N*. Elle consiste à diviser *N* par tous les nombres premiers inferieurs à *N*, puis voir si le résultat obtenu est un entier. Dans le cas le plus défavorable, cette méthode nécessite $\sqrt{N}$ tests pour avoir un résultat satisfaisant [50].

1.5.2.2. Factorisation par calcul de différence de carrés.

Cette méthode repose sur la remarque suivant :

Tout nombre impair *N* peut être représenté par une différence de deux carrés telle que, $N=a^2-b^2$ [50], [52], [53]. De ce fait, *N* peut être réécrit sous la forme suivante :

$$N = (a - b) \times (a + b) \tag{1.10}$$

Cette méthode consiste à trouver un entier *a* tel que $a > \lceil\sqrt{N}\rceil$ et tester si le résultat de la soustraction a^2-N est un carré. Autrement dit, *b* correspond au résultat de la racine carrée $\sqrt{a^2 - N}$. Si cette condition est satisfaite, la factorisation de *N* correspond à $N=(a-b)\times(a+b)$.

1.5.3. Attaques par canal auxiliaire

Dans ce type d'attaque, les cryptanalystes tendent à étudier d'autres procédés permettant de contourner les formalismes mathématiques. Les méthodes développées dans ce sens, sont des attaques physiques, effectuées directement sur les supports d'implémentation (puce électronique). En d'autres termes, ces méthodes sont basées sur des informations obtenues par un canal auxiliaire qui cible la variation de la consommation électrique en fonction du temps. Par l'exploitation de ces informations, la clé privée peut facilement être restituée [1]. La figure 1.9 représente un échantillon d'un tracé de la consommation électrique en fonction du temps.

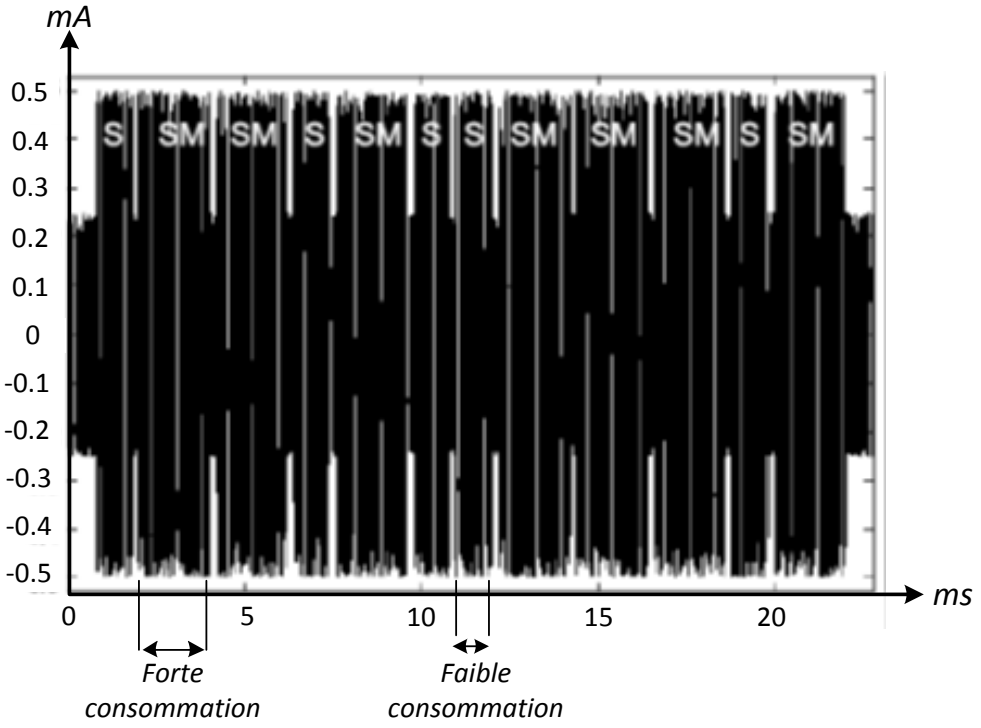


Figure 1.9. Consommation électrique en fonction du temps [1].

Cette figure montre que la consommation électrique varie d'une manière irrégulière en fonction du temps. En effet, il existe des intervalles temporels où la consommation est importante. Dans d'autres, celle-ci peut être considérée comme étant faible. L'origine de cette variation est due à l'algorithme utilisé pour l'exécution de l'exponentiation modulaire. Un des algorithmes qui est proposé dans la littérature pour calculer cette fonction, est basé sur la représentation binaire de l'exposant [13]. Dans cet algorithme, à chaque itération une multiplication modulaire et une élévation au carré sont exécutées. La multiplication modulaire n'est effectuée que si le $i^{ème}$ *bit* de l'exposant est un *″1″ logique*.

Sur le tracé de la consommation électrique en fonction du temps, les intervalles temporels où la consommation est forte se traduisent par une exécution d'une élévation au carré *(S)* et

d'une multiplication modulaire *(SM)*. Ce qui signifie que le $i^{ème}$ *bit* de l'exposant est un *"1" logique*. Par contre, dans les intervalles où la consommation se trouve moins intense, l'opération exécutée correspond à l'élévation au carré seule. Ce qui implique que le $i^{ème}$ *bit* de l'exposant est un *"0" logique*. Ainsi la déduction de l'échantillon de l'exposant qui est associé au tracé de la figure 1.9 est *011010011101*. Pour pallier à ce genre d'attaque, une des solutions qui peut être utilisée est de réaliser la multiplication modulaire même si le $i^{ème}$ *bit* de l'exposant est un *"0" logique*. Cependant, pour ne pas perturber l'exécution de l'algorithme avec des résultats intermédiaires erronés, le résultat obtenu de cette multiplication modulaire ne sera pas pris en considération.

1.6. Mise en œuvre du crypto système RSA et ses niveaux d'abstractions

De nos jours, le RSA est le crypto système le plus utilisé dans divers schémas cryptographiques tels que le chiffrement et le déchiffrement de données de faibles tailles, ou encore dans les signatures numériques. Généralement, son implémentation nécessite la réalisation de trois niveaux d'abstraction. Ces derniers sont représentés sur la figure 1.10.

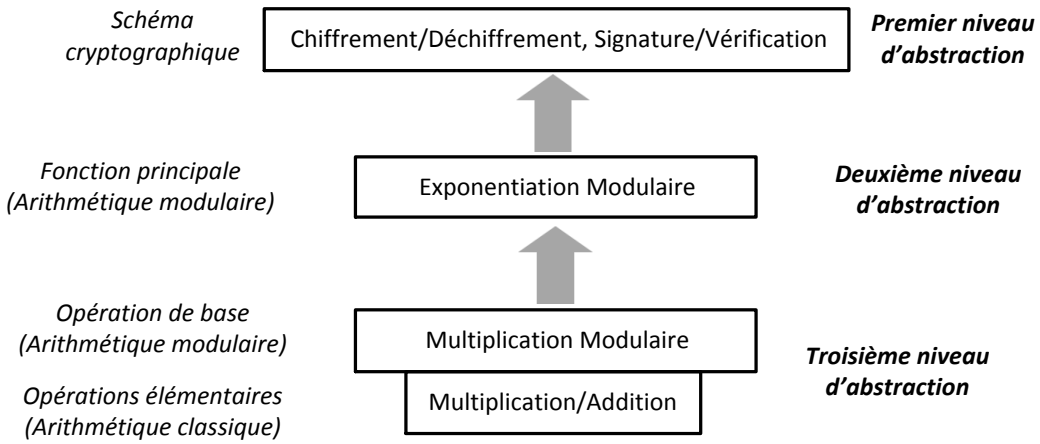


Figure 1.10. Niveaux d'abstractions du crypto système RSA

1. Le plus haut niveau d'abstraction consiste en:
    - La sélection du schéma cryptographique ciblé.
    - La génération et l'attribution des clés publique et privé.
2. Le second niveau d'abstraction correspond aux étapes de chiffrement et de déchiffrement, données par les équations (1.8) et (1.9). Ces dernières ne sont rien d'autres que le calcul de la fonction d'exponentiation modulaire dans le corps fini Z_N. Celle-ci est définie pour le reste de ce travail par l'équation (1.11).

$$Y = X^z mod\ N \tag{1.11}$$

Les entiers *N, X et Z* représentent respectivement le modulo, le message à chiffrer ou à déchiffrer et l'exposant public ou privé. *Y* est le résultat de chiffrement ou de déchiffrement.

3. Le troisième niveau d'abstraction est considéré comme étant la multiplication modulaire dans Z_N. Son implémentation est basée sur les opérations arithmétiques classiques, telles que, la multiplication et l'addition.

La réalisation du premier niveau d'abstraction est souvent attribuée à une autorité de certification [47]. Son implémentation peut être effectuée indépendamment du support d'implémentation des deux autres niveaux. De ce fait, l'amélioration des performances d'exécution du protocole RSA, en termes de temps d'exécution et du coût, dépend principalement de l'optimisation de l'exponentiation modulaire et de la multiplication modulaire.

En revanche, un facteur très important doit être considéré dans le premier niveau d'abstraction. Le facteur en question est la taille du modulo *N*. Ce dernier définit non seulement le niveau de sécurité du protocole RSA, mais aussi la taille des données à manipuler lors de l'exécution des opérations arithmétiques.

1.6.1. Calcul de la taille de la clé privée *(d, N)* en fonction de la taille du *modulo N*

Toute la sécurité et la complexité calculatoire du protocole RSA dépendent principalement de la taille de *N*. Celle-ci possède une influence directe sur la taille de la fonction de l'Euler *Φ(N)* qui, à son tour, influe sur la taille de l'exposant privé *d*, puisque *d* est calculé à partir de *Φ(N).*

A l'origine, *N* est obtenu par le produit des deux facteurs premiers *p* et *q*. De ce fait, pour un entier *N* de taille *m-bits*, où $m = \lceil log_2 N \rceil bits+1$, *p* et *q* sont représentés sur *m/2-bits*. Autrement dit, ils appartiennent à l'intervalle $[2^{m/2-1}, 2^{m/2}-1]$ [1]. La figure 1.11 résume le calcul de *N* et sa taille en fonction des deux facteurs *p* et *q*.

Les tailles des deux facteurs *p* et *q*, nous permettent par la suite de déterminer la taille de la fonction de l'Euler *Φ(N)* qui est calculée par la multiplication de *(p-1)* par *(q-1).* De ce fait, comme les deux opérandes *p-1* et *q-1* sont codés sur *m/2-bits*, *Φ(N)* sera représentée sur *m-bits*.

La taille de l'exposant privé *d* peut être déduite à partir de l'équation (1.7). Celle-ci nous permet d'exprimer *d* en fonction de *Φ(N)* comme suit :

$$d = 1/e\ mod\ \Phi\ (N) \tag{1.12}$$

L'équation (1.12) montre que la valeur de *d* est strictement inférieure à *Φ(N)*, puisque *d* est le reste de la division de *(1/e)* par *Φ(N)*. Sa taille peut être du même ordre que la taille de *Φ(N)* [54].

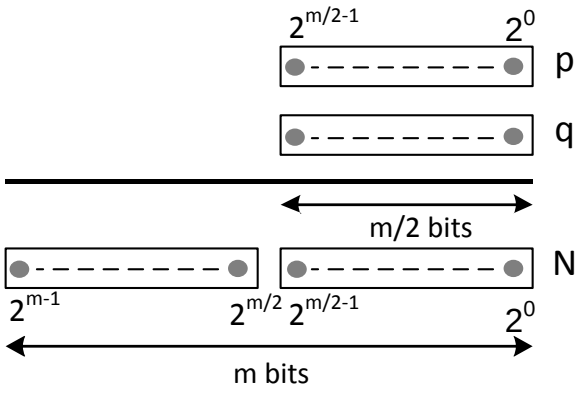


Figure 1.11. Calcul du modulo *N* et de sa taille

1.6.2. Influence de la taille du modulo sur la complexité des opérations arithmétiques

Les différents algorithmes développés dans la littérature pour le calcul de l'exponentiation modulaire ont montré que le nombre de multiplications modulaires à effectuer dépend de la taille de l'exposant [13]. Par conséquent, ce dernier peut avoir un impact considérable sur les performances d'exécution du crypto système RSA. En effet, pour un modulo *N* de taille *1024-bits*, l'utilisation de la méthode binaire [13] dont l'algorithme est basé sur le calcul à chaque itération d'une élévation au carré modulaire et d'une multiplication modulaire, nécessite une moyenne de *1536 multiplications modulaires*. La taille des opérandes dans chaque opération sont de l'ordre de la taille du modulo, c'est-à-dire, *1024-bits*. Ainsi, si on suppose qu'on dispose d'un processeur *32-bits* pour exécuter les opérations arithmétiques de base (multiplication et addition), chaque opérande doit être stocké dans *(1024/32=32) registres*. De ce fait, pour atteindre la précision des opérandes qui est de *1024-bits*, chaque multiplication nécessite $32^2=1024$ *multiplications* et *additions* de *32-bits*. De plus, chaque multiplication doit être réduite en *modulo N*. En effet, l'algorithme le plus utilisé pour calculer la multiplication modulaire, en l'occurrence l'algorithme de Montgomery [20], nécessite *1024* multiplications et additions de *32-bits* pour réaliser l'opération de réduction. De ce fait, le nombre total de multiplications et additions de *32-bits* pour effectuer une seule multiplication modulaire est de *2048*. En vertu de la méthode binaire qui exige *1536 multiplications modulaire*, le nombre global d'opérations de base nécessaires pour calculer l'exponentiation modulaire est de *2048×1536 = 3145728 opérations* [1].

1.7. Conclusion

Dans ce chapitre nous avons présenté les différents types de protocoles de cryptographie, à savoir, la cryptographie symétrique, la cryptographie asymétrique et un schéma de cryptographie hybride.

Nous nous sommes intéressés en particulier dans ce chapitre à l'un des algorithmes de cryptographie asymétrique le plus utilisé, en l'occurrence le RSA. Bien que ce dernier soit développé il y a plus de 30 ans, il constitue un des crypto systèmes les plus utilisés actuellement. Pour mieux comprendre les mécanismes et les faiblesses du protocole RSA, nous avons présenté également le type d'arithmétique utilisée et quelques méthodes qui permettent de déchiffrer les données en clair, sans détenir la clé privé. A l'issue de cette étude, nous avons montré que la taille du modulo possède un impact non seulement sur la complexité de ses opérations arithmétiques de base mais aussi sur son niveau de sécurité.

Les opérations arithmétiques nécessaires à la mise en œuvre du crypto système RSA sont basées essentiellement sur la fonction d'exponentiation modulaire qui est réalisée par une suite répétée de multiplications modulaires. Pour accélérer les opérations de chiffrement et de déchiffrement, on doit optimiser d'une part le calcul de l'exponentiation modulaire et d'autre, part réduire le temps d'exécution de la multiplication modulaire. De ce fait, notre objectif dans ce travail ne se limite pas uniquement à l'implémentation du crypto système RSA, mais de proposer une architecture matérielle qui présente le meilleur compromis entre le temps d'exécution et la surface occupée sur le circuit. Pour cela, une étude algorithmique des opérations de base, à savoir l'exponentiation modulaire et la multiplication modulaire est incontournable pour atteindre les objectifs du travail. Le chapitre suivant est consacré à l'étude de ces deux opérations.

Chapitre 2

Algorithmes de calcul de la multiplication et de l'exponentiation modulaires

2.1. Introduction

Les algorithmes développés dans la littérature pour le calcul de l'exponentiation modulaire, ont montré que toute sa complexité est réduite au calcul itératif de la multiplication modulaire [1], [13]. Ainsi, augmenter les performances du crypto système RSA revient donc à réduire d'une part le temps consenti à l'exécution de la multiplication modulaire et, d'autre part, à optimiser le nombre de multiplications modulaires requises par l'exponentiation modulaire.

Généralement, lorsqu'il s'agit d'implémenter un algorithme sur matériel, les exigences sont fortes. En particulier, il faut que l'implémentation utilise le moins de ressources matérielles possibles et que le temps d'exécution soit faible. Il est clair que minimiser tous ces paramètres n'est pas aisé. En effet, un gain en temps d'exécution se paie par un surcoût matériel. Réciproquement, un gain matériel conduit à un temps d'exécution plus long. De ce fait, avant de parvenir à la réalisation matérielle de l'algorithme ciblé, il est indispensable de procéder à l'évaluation de sa complexité et de proposer par la suite, des solutions qui permettent de répondre aux exigences de l'implémentation.

Ce chapitre est consacré à l'étude algorithmique de la fonction d'exponentiation modulaire et de la multiplication modulaire. Nous entamons notre chapitre par une étude de quelques méthodes de calcul de l'exponentiation modulaire et leurs complexités algorithmiques. Puis, nous présenterons un bref aperçu sur le calcul de la multiplication modulaire. Nous nous intéresserons en particulier à l'algorithme de la MMM, ses variantes et leurs complexités algorithmique.

Dans ce chapitre, il sera aussi question de présenter les méthodes proposées dans ce travail pour l'implémentation des algorithmes de la MMM et de l'exponentiation modulaire. Ces méthodes ont été développées dans le but d'adapter l'exécution de ces algorithmes aux ressources internes du processeur utilisé, à savoir, le processeur Microblaze de Xilinx.

2.2. Méthodes de calculs d'exponentiation modulaire

Rappelons que le chiffrement et le déchiffrement RSA nécessitent tous deux le calcul d'une exponentiation modulaire, définie par $Y=X^Z \bmod N$, où X, N et Z sont de très grands entiers.

La première règle de l'exponentiation modulaire est qu'on ne peut pas calculer $Y=X^Z \bmod N$ par le calcul direct de X^Z puis calculer ensuite le reste de la division de X^Z par N pour trouver le résultat Y. Ceci est dû à l'espace mémoire nécessaire pour stocker le résultat intermédiaire X^Z qui peut être très élevé [1], [13], [14]. D'où l'idée de réduire en *modulo N*, le résultat de chaque multiplication. Nous pouvons nous poser la question de savoir de combien de multiplications modulaires nous avons besoin pour calculer $Y=X^Z \bmod N$. La méthode naïve serait d'initialiser avec la valeur $Y = X \bmod N$ et de faire la multiplication modulaire Z *fois* jusqu'à l'obtention du résultat final $Y=X^Z \bmod N$. Cette méthode est pratiquement infaisable pour des exposants de l'ordre de *1024 bits*. Bien entendu il existe de meilleures méthodes qui diminuent énormément le nombre de multiplications modulaires. Dans ce qui suit nous allons présenter quelques-unes de ces méthodes.

2.2.1. La méthode binaire

Dans cette méthode, le calcul de l'exponentiation modulaire $Y=X^Z \bmod N$ est basé sur la représentation de l'exposant Z selon le schéma de Horner en *base 2* [55]. Celle-ci est donnée par l'expression (2.1).

$$Z = \sum_{i=0}^{e-1} z_{(i)} \times 2^i = (\dots((z_{(e-1)} \times 2 + z_{(e-2)}) \times 2 + z_{(e-3)}) \times 2 + \cdots + z_{(1)}) \times 2 + z_{(0)} \quad (2.1)$$

L'exécution de cette méthode repose principalement sur des décalages effectués sur l'exposant Z, où ce dernier est introduit dans les calculs *bit-par-bit*. De ce fait, suivant le sens des décalages, il en résulte deux algorithmes :

- Algorithme L2R (Left to Right).
- Algorithme R2L (Right to Left).

a) Algorithme L2R (Left to Right)

L'exécution de cet algorithme est basée sur la lecture de l'exposant *Z bit-par-bit*, poids fort en tête [13]. Il est défini comme suit :

Algorithme 1 –Algorithme L2R

Entrée*: X, N,* $Z = \sum_{i=0}^{e-1} z_{(i)} \times 2^i$

Variables intermédiaires : $S_{(i)}$, $C_{(i)}$

Sortie *: Y:=* X^Z *mod N*

Début

1. ***Si*** $z_{(e-1)}=1$ ***alors*** $C_{(e-1)}=X \mod N$
2. ***sinon*** $C_{(e-1)}=1$
3. ***Pour*** $i = e-2$ ***à*** 0 ***faire***
4. $S_{(i)} = (C_{(i+1)} \times C_{(i+1)}) \mod N$
5. ***Si*** $z_{(i)} = 1$ ***alors*** $C_{(i)} = (S_{(i)} \times X) \mod N$
6. ***Si non*** $C_{(i)}=S_{(i)}$
7. ***Fin pour***

Retourne $Y= C_{(0)}$

Cet algorithme reçoit en entrées le message *X*, le modulo *N* et l'exposant *Z*. Sa complexité en termes de nombre d'itérations est proportionnelle à la taille de l'exposant, où *(e-1)* itérations sont nécessaires pour avoir le résultat *Y* en sortie. A chaque itération *(i)* de l'algorithme, une élévation au carré modulaire et une multiplication modulaire sont calculées séquentiellement. Les résultats intermédiaires de ces deux opérations sont stockés respectivement dans deux variables $S_{(i)}$ et $C_{(i)}$.

- $S_{(i)}$ correspond au résultat obtenu de l'élévation au carré $(C_{(i+1)} \times C_{(i+1)}) \mod N$.
- $C_{(i)}$ est associé au résultat de la multiplication modulaire $(S_{(i)} \times X) \mod N$.

Il est à noter que la seconde opération est conditionnée par l'état du $i^{ème}$ *bit* de *Z*. Celle-ci n'est effectuée que si $z_{(i)}=1$. De ce fait, la condition de la ligne 5 peut avoir une influence considérable sur la complexité temporelle de l'algorithme 1. En effet, pour un exposant Z de taille *e-bits* avec $z_{(e-1)}=1$, l'algorithme 1 nécessite:

- *(e-1)* multiplications modulaires qui correspondent aux calculs de l'élévation au carré. Pour un exposant de taille *1024 bits*, on aura besoin de *1023* multiplications modulaires pour effectuer cette étape.

- Le nombre de multiplications modulaires de la ligne 5 dépend du nombre de bits *non zéro* dans la chaîne de bits de *Z*. Le nombre en question varie de *0* à *e-2*. Pour un exposant de taille *1024 bits*, cette étape nécessite un nombre de multiplications modulaires variant de *0* à *1023*.

D'où, le nombre total de multiplications modulaires pour calculer l'exponentiation modulaire $Y=X^Z \bmod N$, varie entre $2\times(e-1)$ pour le cas le plus défavorable et $(e-1)$ pour le meilleur cas. Dans le cas où la moitié de la chaine de bits de *Z* est *non zéro*, ce qui correspond à la valeur moyenne, le nombre total de multiplications modulaire *Nb_MM_alg1* est donné par l'équation (2.2) [13].

$$Nb\_MM\_alg1 = 3/2\times(e-1) \qquad (2.2)$$

b) Algorithme R2L (Right to Left)

Cet algorithme utilise des décalages à droite *bit-par-bit* sur l'exposant *Z* pour calculer l'exponentiation modulaire $Y=X^Z \bmod N$. L'algorithme R2L est défini comme suit [13]:

Algorithme 2 –Algorithme R2L

Entrée: $X, N, Z = \sum_{i=0}^{e-1} z_{(i)} \times 2^i$

Variables intermédiaires : $S_{(i)}$, $C_{(i)}$

Sortie : $Y:= X^Z \bmod N$

Début

1. $S_{(-1)}=X$
2. $C_{(-1)}=1$
3. ***Pour*** $i = 0$ ***à*** $e-1$ ***faire***
4. $S_{(i)} = (S_{(i-1)} \times S_{(i-1)}) \bmod N$
5. ***Si*** $Z_{(i)} = 1$ ***alors*** $C_{(i)} = (S_{(i-1)} \times C_{(i-1)}) \bmod N$
6. ***Si non*** $C_{(i)}=C_{(i-1)}$
7. ***Fin pour***

Retourne $Y= C_{(e-1)}$

Les entrées de cet algorithme sont constituées du message *X*, du modulo *N* et de l'exposant *Z*. A chaque itération *(i)* de l'algorithme, une élévation au carré modulaire et une multiplication modulaire sont calculées en parallèle. Les résultats intermédiaires obtenus à partir de ces deux opérations sont stockés dans deux variables $S_{(i)}$ et $C_{(i)}$.

- $S_{(i)}$ est obtenue par l'élévation au carré modulaire $(S_{(i-1)} \times S_{(i-1)}) \bmod N$.
- $C_{(i)}$ est obtenue par la multiplication modulaire $(S_{(i-1)} \times C_{(i-1)}) \bmod N$.

La condition de la ligne 5 qui est portée sur l'état du $i^{ème}$ *bit* de *Z* concerne principalement le calcul de la multiplication modulaire. Celle-ci n'est exécutée que si $z_{(i)}=1$. Comme les deux opérations sont exécutées à chaque itération indépendamment, cette condition n'affecte en aucun cas la complexité temporelle de l'algorithme. Par conséquent, le temps d'exécution de l'exponentiation modulaire est équivalent au temps global d'exécution des élévations au carré, dont le nombre est défini par la taille de *Z*. Ainsi, la complexité en termes de nombre de multiplications modulaires *Nb_MM_alg2* est évaluée par l'équation (2.3).

$$Nb\_MM\_alg2=e \tag{2.3}$$

2.2.2. La méthode M-ary

La méthode M-ary peut être considérée comme étant une optimisation de la méthode binaire. Son objectif consiste à réduire le nombre d'itérations nécessaires au calcul de l'exponentiation modulaire [13], [56]. Cette méthode est basée sur le codage de l'exposant *Z* en base $\beta=2^r$, défini par :

$Z = \sum_{i=0}^{e/r-1} Z[i] \times 2^{r \times i}$, avec $Z[i] \in \{0,1, \dots, 2^r - 1\}$.

Ce qui signifie que la chaine de bits de *Z* est décomposée en *e/r chiffres* de taille *r-bits*. Comme ceci est montré sur la figure 2.1.

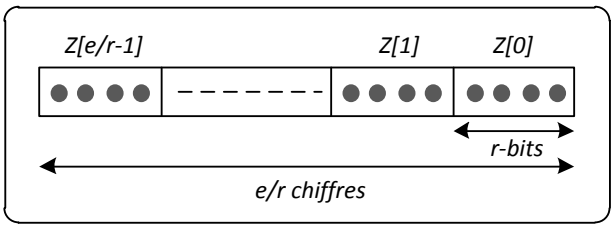


Figure 2.1. Représentation de l'exposant *Z* dans la méthode M-ary

Par rapport à la méthode binaire, où seulement un seul bit de Z est utilisé pour l'exécution de chaque itération ; dans la méthode M-ary, l'exposant est introduit *chiffre-par-chiffre*. Ainsi, à chaque itération, *r-bits* de *Z* sont utilisés à la fois dans le calcul de l'exponentiation modulaire. L'algorithme de la méthode M-ary est défini comme suit :

Algorithme 3– Algorithme de la méthode M-ary

Entrée*: X, N,* $Z = \sum_{i=0}^{e/r-1} Z[i] \times 2^{r \times i}$

Variables intermédiaires : $C_{(i)}$, $S_{(i)}$

Sortie *: Y:=* $X^Z \bmod N$

Début

1. *Calcul et stockage de* $X^{\omega} \bmod N$ *avec* $\omega \in \{1,2,3,\ldots, 2^r-1\}$
2. $C_{(e/r-1)}=X^{Z[e/r-1]} \bmod N.$
3. ***Pour*** *i = e/r-2* ***à*** *0* ***faire***
4. $S_{(i)} = (C_{(i+1)}^{2^r}) \bmod N$
5. ***Si*** $Z[i] \neq 0$ ***alors*** $C_{(i)} = (S_{(i)} \times X^{Z[i]}) \bmod N$
6. ***Si non*** $C_{(i)}=S_{(i)}$
7. ***Fin pour***

Retourne $Y= C_{(0)}$

Cet algorithme est basé sur la représentation de *Z* suivant le schéma de Horner [55] en base 2^r. Celle-ci est donnée par l'expression (2.4).

$$Z = \sum_{i=0}^{\frac{e}{r}-1} Z[i] \times 2^{r \times i} = (\ldots((Z[e/r-1] \times 2^r + Z[e/r-2]) \times 2^r + Z[e/r-3]) \times 2^r + \cdots + Z[1]) \times 2^r + Z[0] \quad (2.4)$$

où, $Z[i] = \sum_{j=0}^{j=r-1} z_{(j)} \times 2^j$

A chaque itération *(i)* de l'algorithme, un décalage à gauche est effectué sur l'exposant *Z*. Le chiffre le plus significatif, en l'occurrence *Z[e/r-1]* est le premier chiffre introduit dans les opérations arithmétiques. Pour mieux illustrer l'exécution de cet algorithme, considérons la première itération où *i=e/r-2*. Le résultat intermédiaire $C_{(e/r-2)}$ issu de cette itération est donné par l'équation (2.5).

$$C_{(e/r-2)} = X^{Z[e/r-1] \times 2^r + Z[e/r-2]} \bmod N$$

$$= X^{Z[e/r-1] \times 2^r} \bmod N \times X^{Z[e/r-2]} \bmod N \quad (2.5)$$

On pose $C_{(e/r-1)}=X^{Z[e/r-1]} \bmod N$. Ainsi, l'équation (2.5) devient,

$$C_{(e/r-2)} = (C_{(e/r-1)}^{2^r} \bmod N \times X^{Z[e/r-2]}) \bmod N$$

$$= S_{(e/r-2)} \times X^{Z[e/r-2]} \bmod N \quad (2.6)$$

avec, $S_{(e/r-2)} = C_{(e/r-1)}^{2^r} \bmod N$ (2.7)

Les résultats des équations (2.6) et (2.7) représentent respectivement les résultats intermédiaires $C_{(e/r-2)}$ et $S_{(e/r-2)}$ de l'algorithme 3 à l'itération *(e/r-2)*.

$C_{(e/r-2)}$ sera utilisé ensuite comme opérande d'entrée dans la prochaine itération pour calculer $C_{(e/r-3)}$. Celui-ci sera calculé de la même manière que le résultat précédant. Ainsi, l'exécution de chaque itération peut être généralisée sur l'ensemble des itérations, jusqu'à l'obtention du résultat de l'exponentiation modulaire $Y=C_{(0)}=X^Z \bmod N$.

Pour permettre l'exécution de l'algorithme 3, ce dernier nécessite quelques opérations supplémentaires qui précèdent le déroulement de ses itérations. Les opérations en question sont :

- Calcul des puissances $X^{\omega} \bmod N$, avec $\omega \in \{1,2,\ldots,2^r-1\}$. Ces puissances sont utilisées dans la multiplication modulaire de la ligne 5.
- Initialisation de la variable $C_{(i)}$ pour $i=(e/r)-1$. Cette opération est réalisée par le calcul de l'exponentiation modulaire $C_{(e/r-1)}=X^{Z[e/r-1]} \bmod N$.

Une fois ces calculs sont achevés, l'exécution de l'algorithme devient itérative. A chaque itération, les opérations suivantes sont réalisées :

- Une élévation à une puissance $2^r \bmod N$ du résultat intermédiaire $C_{(i+1)}$.
- Calcul du résultat intermédiaire $C_{(i)}$. Cette opération est conditionnée par la valeur du $i^{ème}$ chiffre de l'exposant *Z*, où celle-ci n'est effectuée que si $Z[i]\neq 0$.

La complexité de cette méthode, repose principalement sur l'évaluation du nombre de multiplications modulaires, calculées tout au long de l'algorithme 3. En effet, pour avoir le résultat final de l'exponentiation modulaire, cet algorithme nécessite deux étapes.

1. La première étape n'est rien d'autre qu'un pré-traitement, où les puissances $X^{\omega} \bmod N$ sont calculées, avec $\omega \in \{1,2,3,\ldots,2^r-1\}$. De ce fait, le nombre de multiplications modulaire, requis pour réaliser cette étape est de 2^r-2.
2. La deuxième étape concerne l'exécution itérative de l'algorithme où, à chaque itération, l'élévation $S_{(i)}=(C_{(i+1)}^{2^r}) \bmod N$ et la multiplication modulaire $C_{(i)} = (S_{(i)} \times X^{Z[i]}) \bmod N$ sont effectuées. Le calcul de ces deux opérations à chaque itération, nécessite $(r+1-1/2^r)$ multiplications modulaires. En vertu du processus itératif de cette étape qui est effectué sur *e/r-1 itérations*, le nombre de

multiplications modulaires requis pour achever son exécution est de $(e/r-1)\times(r+1-1/2^r)$.

De ce fait, le nombre global de multiplications modulaires *Nb_MM_alg3* nécessaire pour exécuter l'algorithme 3, peut être évalué par l'expression (2.8).

$$Nb\_MM\_alg3 = (2^r-2)+[(e/r-1)\times (r+1-1/2^r)] \quad (2.8)$$

2.2.3. Comparaison entre les algorithmes des méthodes binaires et M-ary

L'étude algorithmique des deux méthodes présentées dans ce chapitre, nous a permis d'évaluer dans un premier temps, la complexité d'exécution de la fonction d'exponentiation modulaire, en termes de nombre de multiplications modulaires requises. Néanmoins, lorsqu'il s'agit de mettre en œuvre ces algorithmes sur matériel, l'estimation du coût en termes d'unités de stockage est une contrainte incontournable et ne doit pas être négligeable.

En effet, le nombre de ces unités qui peuvent être des blocs mémoires ou des registres, est évalué en se basant sur le nombre de variables intermédiaires utilisées dans chaque algorithme. Pour un modulo N de taille *m-bits*, la méthode M-ary exige $(2^r-1)+2$ unités de stockage dont les tailles sont proportionnelles à m. Ces unités sont nécessaires pour sauvegarder tous les prés-calculs $X^{\omega} \bmod N$ de l'algorithme *3*. Les deux unités supplémentaires sont requises pour stocker les résultats intermédiaires $S_{(i)}$ et $C_{(i)}$.

La méthode binaire avec ses deux algorithmes nécessite moins de ressources. L'algorithme L2R utilise deux unités de stockage. La première correspond au stockage du message X. La seconde est dédiée aux variables intermédiaires $S_{(i)}$ et $C_{(i)}$. Comme ces dernières sont calculées séquentiellement, elles peuvent être stockées en alternance dans une même unité. En revanche, l'algorithme R2L, en vertu de l'exécution parallèle de ses deux multiplications modulaires, nécessite trois unités pour stocker le message X et les résultats intermédiaires $S_{(i)}$ et $C_{(i)}$.

La complexité de chaque algorithme et le nombre de ressources matérielles requis sont résumés dans le tableau 2. A travers ce dernier, on constate que l'algorithme R2L peut être considéré comme étant l'algorithme qui présente le meilleur compromis temps d'exécution/ressources occupées. De ce fait, les différentes approches proposées dans ce travail pour l'implémentation de l'exponentiation modulaire seront basées sur l'utilisation de cet algorithme.

Tableau 2. Complexité des algorithmes de calcul de l'exponentiation modulaire

Algorithme	**Pré calcul**	**Performances temporelles en termes de nombre de Multiplications modulaire**	**Multiplieur Modulaire**	**Unité de stockage**
Algorithme	Aucun	3/2×(e-1)	1	2
Algorithme	Aucun	e	2	3
Méthode M-ary	Puissance modulaire	$(2^r-2)+[(e/r-1)\times(r+1-1/2^r)]$	1	$(2^r-1)+2$

2.3. Multiplication modulaire

La multiplication modulaire est une opération critique dans le protocole de cryptographie RSA. Celle-ci est définie comme suit :

Considérons deux entiers A et B de Z_N. La multiplication modulaire consiste à effectuer le produit $(A\times B)$ et prendre le reste T obtenu de la division de ce produit par N. Cette opération est donnée par l'expression (2.9).

$$T = (A \times B) mod\ N \tag{2.9}$$

D'une manière générale, le calcul de $(A\times B)\ mod\ N$, nécessite deux phases : une phase de multiplication suivie d'une phase de réduction qui peut être effectuée par un calcul de division.

La division est une opération complexe, car son temps de calcul est beaucoup plus long que le temps de calcul de la multiplication [23], [57]. En cryptographie asymétrique, un usage intensif de la réduction modulaire est nécessaire. Il est alors très coûteux de réaliser cette réduction par division. Outre la complexité temporelle de la division, son inconvénient majeur est l'ordre dans lequel est réalisée cette opération. Celle-ci est réalisée à travers tous ses algorithmes, des poids forts vers les poids faibles. Alors que la multiplication est calculée dans le sens inverse. Ainsi, lorsqu'on veut enchaîner une opération de multiplication avec une phase de réduction modulaire par division, il est nécessaire d'attendre la fin de la multiplication pour avoir accès au poids forts et ainsi commencer la division. De plus, une fois la division entamée, il est nécessaire d'attendre la fin de celle-ci pour commencer une nouvelle multiplication, et ainsi de suite. Il y'a donc une perte de temps due à l'attente systématique du résultat complet. De ce fait, plusieurs méthodes dédiées au calcul de la multiplication modulaire sont proposées dans la littérature [6]. Ces dernières ont pour objectif principal de contourner le calcul de la division. Parmi celles-ci, la

méthode proposée par Montgomery se trouve la plus adaptée pour l'implémentation matérielle de la multiplication modulaire

2.3.1. Multiplication Modulaire de Montgomery (MMM)

En 1985, Montgomery propose un algorithme de multiplication modulaire qui transforme la réduction modulo *N* en une division par une puissance de la base [20]. En pratique, ceci semble être très intéressant, puisque la division par la base revient à faire un simple décalage à droite. En effet, pour réduire la multiplication *(A×B)* modulo *N*, où *A, B et N* sont des entiers codés sur *m-bits*, la méthode classique consiste à trouver le quotient *q* telle que les *m-bits* les plus significatifs des produits *A×B* et *q×N* soient égaux. La méthode de Montgomery repose sur le principe inverse, c'est-à-dire déterminer un quotient *q*, tel que les *m bits* les moins significatifs de *A×B* et de *q×N* seront identiques. Ainsi, nous obtenons une valeur équivalente à *A×B mod N* dont tous les bits de poids faible sont nuls, comme le montre la figure 2.2.

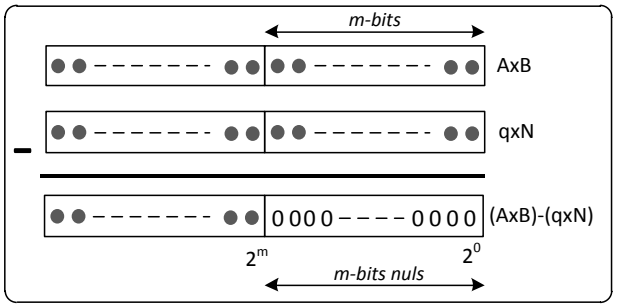


Figure 2.2. Principe de base de la multiplication de Montgomery

L'algorithme 4 présente la multiplication modulaire de Montgomery [6].

Algorithme 4 –Algorithme de la multiplication de Montgomery

Entrées *: A, B, N avec* $0 \leq A, B < N$

Prés-calculés *: N′ et R avec :* $R=\beta^n$, $(-N)\times N' = 1 \bmod R$ *et pgcd (N,R)=1*

Variables *: C, q*

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

1. ***Debut***
2. $C=A\times B$
3. $q = C\times N' \bmod R$
4. $T = (C + q\times N)/R$
5. ***Si*** $T \geq N$ ***alors*** $T = T - N$

Retourne *T*

La MMM de deux entiers *A* et *B* est définie par l'équation (2.10) [20].

$$T = Montgomery\ (A, B) = (A \times B \times R^{-1})\ mod\ N \qquad (2.10)$$

N est un entier impair représenté en base β sur *n chiffres*. *R* est choisi telle que $R=\beta^n$ avec, *pgcd(R, N)=1*.

- Dans le protocole RSA, la condition *pgcd (R, N) =1* est souvent satisfaite car *N*, est un nombre impair, issu de la multiplication de deux nombres premiers.
- La MMM nécessite un pré-calcul, en l'occurrence N' qui est obtenu par : $N' = (-1/N)\ mod\ R$.

Le facteur déterminant dans la méthode de Montgomery est sans doute la valeur du quotient *q*. En effet, Montgomery a montré que si $q= (A\times B\times N')\ mod\ R$, la quantité $(A\times B)-(q\times N)$ est non seulement un entier, mais aussi, elle est divisible par 2^m, où *m* est la taille de la représentation de *N* en *base 2*. Ceci peut être exprimé comme suit :

Si on pose : $T = \frac{A\times B}{R} - \frac{q\times N}{R}$, *alors T représente le reste de la division* $\left.\frac{A\times B}{R}\right/_N$ *où* $R=2^m$.

La complexité C_{algo4} de l'algorithme 4 en termes de nombre d'opérations est de trois multiplications, une addition et une soustraction. C_{algo4} est évaluée par l'équation (2.11).

$$C_{algo4} = 3\times Mul + Add + Sub \qquad (2.11)$$

Le résultat $T=(A\times B\times R^{-1})\ mod\ N$ de la MMM montre que celle-ci calcule non pas $(A\times B)\ mod\ N$, mais une réduction avec un facteur supplémentaire R^{-1}. Pour parvenir à la réduction de la multiplication $(A\times B)$ en modulo *N*, une opération supplémentaire doit être effectuée afin de supprimer ce facteur. Celle-ci est réalisée en utilisant une seconde multiplication de Montgomery, où les opérandes sont $(R^2\ mod\ N)$ et le résultat *T* de l'opération précédente. Le résultat final $(A\times B)\ mod\ N$ est obtenu par l'équation suivante :

$$[(R^2\ mod\ N)\ \times ((A\times B\ \times R^{-1})\ mod\ N\ )\ \times\ R^{-1}]\ mod\ N =\ (A\times B)\ mod\ N$$

2.3.2. Représentation de Montgomery et ses propriétés

Le faite d'avoir le facteur R^{-1}dans le résultat de la MMM pose un problème pour le calcul intensif de la multiplication modulaire. De ce fait, Montgomery utilise une représentation particulière dite notation de Montgomery. Celle-ci est notée par

mon(x). Elle consiste à associer à chaque valeur *x* inférieure à *N*, une valeur égale à *mon (x)=(x×R) mod N*, avec $R=\beta^n$. Cette notation donne quelques propriétés utilisées notamment dans le calcul de l'exponentiation modulaire [6]:

a. <u>*Stabilité*</u>. La MMM est stable par rapport à la multiplication.

$$Montgomery\ (mon(A), mon(B)) = (mon(A)\times mon(B)\times R^{-1}) \bmod N$$
$$= (A\times R\times B\times R\times R^{-1}) \bmod N = (A\times B\times R) \bmod N = mon\ (A\times B)$$

b. <u>*Conversion.*</u> Les conversions entre la représentation de Montgomery et la représentation classique s'effectuent en utilisant la multiplication de Montgomery elle-même :

1. Représentation classique vers la représentation de Montgomery

$$A \rightarrow mon(A) : Montgomery\ (A, R^2 \bmod N) = (A\times R^2\times R^{-1}) \bmod N$$
$$= (A\times R) \bmod N = mon(A)$$

2. Représentation de Montgomery vers la Représentation classique

$$mon(A) \rightarrow A : Montgomery\ (mon(A), 1) = (mon(A) \times 1) \bmod N$$
$$= (A\times R\times R^{-1}) \bmod N = A \bmod N$$

2.4. Exponentiation modulaire de Montgomery

La stabilité et la représentation de Montgomery semblent être intéressantes lorsque le calcul intensif de la multiplication modulaire est exigé. Ce qui est le cas pour le calcul de l'exponentiation modulaire $Y=X^Z \bmod N$. Les étapes de calcul de l'exponentiation modulaire, basées sur la MMM, sont résumées sur la figure 2.3.

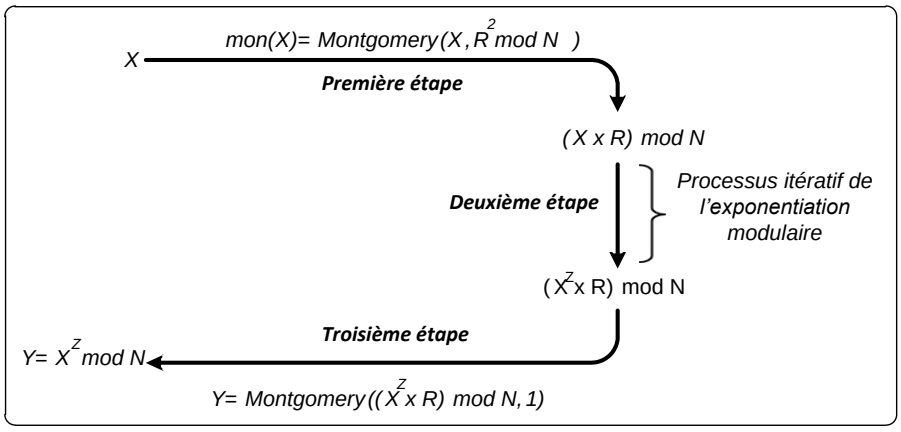


Figure 2.3. Etapes de calcul de l'exponentiation modulaire par l'utilisation de la multiplication de Montgomery

Une première conversion de la représentation classique de *X* vers la représentation de Montgomery (appelée aussi domaine de Montgomery) doit être

effectuée au début du processus de l'exponentiation modulaire. Cette étape est réalisée par une simple multiplication de Montgomery, en introduisant *X* et *(R^2 mod N)* comme opérandes d'entrées. On calcule ainsi *Montgomery (X, R^2mod N).* La valeur de R^2 *mod N* est une constante qui dépend de *N* et de la base *β.* Puis, les calculs sont exécutés dans le domaine de Montgomery tout au long du processus de calcul de l'exponentiation modulaire. Finalement, une conversion vers la représentation classique doit être réalisée. Celle-ci est effectuée par une dernière MMM, où les opérandes d'entrées sont le résultat obtenu de l'étape précédente et ″1″ [58], [59].

2.5. Variantes de la multiplication modulaire de Montgomery

Dans le RSA, les tailles des clés utilisées actuellement sont de l'ordre de *1024-bits* et plus. De ce fait, quand les ressources matérielles sont limitées sur les supports d'implémentations, l'exécution de la MMM et le calcul de ses résultats intermédiaires sont impossibles.

Pour ce faire, afin de faciliter l'implémentation de cette opération sur matériel, des modifications ont été apportées sur son algorithme original (algorithme 4). Celles-ci ont abouti à des variantes qui peuvent être implémentées suivant un mode parallèle ou sériel.

Dans ce qui suit, nous allons définir, en premier lieu, les deux multiplications de Montgomery entrelacées, avec et sans soustraction finale. Celles-ci sont considérées comme étant les principales variantes de l'algorithme original de Montgomery. Puis, nous présenterons la méthode proposée pour une éventuelle implémentation de la multiplication de Montgomery dans un crypto système embraqué.

2.5.1. Multiplication Modulaire de Montgomery Entrelacée (MMME)

L'entrelacement de la multiplication à la réduction modulaire est basé sur la représentation de l'opérande *A* en base $\beta=2^k$ [30], [60]. La méthode peut facilement être généralisée comme suit:

Soit *A* un entier positif représenté en base *β* par :

$A = \sum_{i=0}^{n-1} a_i \times \beta^i = a_0 + a_1 \times \beta^1 + a_2 \times \beta^2 + \ \ldots + a_{n-1}\beta^{n-1}$, $avec\ 0 \leq a_i \leq \beta - 1.$

En utilisant l'équation (2.10), le calcul de la MMM devient:

$$T = \left(\left(\sum_{i=0}^{n-1} a_i \times \beta^i \right) \times B \times \beta^{-n} \right) \ mod\ N$$

$$= \Big(\big((a_0 \times B) + (a_1 \times B) \times \beta + \cdots \ldots \ldots (a_i \times B) \times \beta^i + \cdots \ldots (a_{n-1} \times B) \times \beta^{n-1}\big) \times \beta^{-n}\Big) \bmod N$$

$$= ((\ldots((a_0 \times B \times \beta^{-1} \bmod N + a_1 \times B) \times \beta^{-1} + a_2 \times B) \times \beta^{-1} \bmod N + \cdots) \times \beta^{-1} \bmod N + a_{n-2} \times B) \times \beta^{-1} \bmod N + a_{n-1} \times B \times \beta^{-1} \bmod N$$

Ainsi, la MMM peut être calculée par l'utilisation de la formule de récurrence suivante :

$$\begin{cases} T_{(0)} = 0 \\ T_{(i+1)} = \big(T_{(i)} + (a_i \times B)\big) \times \beta^{-1} \bmod N, \; i = 0 \ldots\ldots n-1 \end{cases} \quad (2.12)$$

A chaque itération *(i)* un produit partiel $a_i \times B$ est calculé, puis additionné au résultat intermédiaire précédent $T_{(i)}$. Le tout est multiplié par β^1. Le résultat obtenu est réduit modulo *N*.

Le calcul de $T_{(i+1)}$ parait difficile en vertu du facteur β^1. Cette problématique peut être contournée en utilisant le théorème de Montgomery [20], [60]. En effet, l'application de ce dernier résulte en une propriété qui constitue le fondement de base de l'algorithme de la MMME. Cette propriété est définie comme suit :

si $q_{(i)} = ((T_{(i)} + (a_i \times B)) \times N') \bmod \beta$, la somme $T_{(i)} + (a_i \times B) + (q_{(i)} \times N)$ est divisible par β où $\beta = 2^k$. Par conséquent, le résultat intermédiaire $T_{(i+1)}$ de l'expression (2.12) est équivalent à :

$$T_{(i+1)} = (T_{(i)} + (a_i \times B) + (q_{(i)} \times N))/\beta \quad (2.13)$$

Avec $N' = -N_0^{-1} \bmod \beta$ et $pgcd(N, \beta) = 1$. N_0 est le chiffre le moins significatif de *N* en base β.

a) Algorithme de la MMME

La multiplication de Montgomery entrelacé se déroule selon l'algorithme 5. L'exécution de ce dernier pour le calcul d'une MMM est résumée dans les points suivants :

- La détermination de $q_{(i)}$ est justifiée par le fait que $T_{(i)} + (a_i \times B) + (q_{(i)} \times N)$ est divisible par 2^k. Celui-ci est calculé à chaque itération, en considérant uniquement le chiffre le moins significatif de l'expression : $((T_{(i),0} + (a_i \times b_0)) \times N')$.

- Après avoir calculé $q_{(i)}$, $T_{(i+1)}$ est obtenu par l'expression (2.13).

Les résultats intermédiaires $T_{(i+1)}$ issus de chaque itération, peuvent être supérieurs à N ($T_{(i+1)}<(2\times N)$) [61]. Ce qui signifie, qu'ils sont représentés sur $n+1$ *chiffres*. De ce fait, la soustraction finale de la ligne 7 est nécessaire pour s'assurer que le résultat de la MMM est strictement inférieur à N ($T_{(n)}<N$).

Algorithme 5 –Algorithme de la MMME

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

Pré-calculs : $N'=-N_0^{-1} \bmod \beta$, $R=\beta^n$ *avec* $\beta=2^k$, *et pgcd* $(N, \beta)=1$

Variables intermédiaires : $q_{(i)}$, $T_{(i)}=\sum_{j=0}^{n-1} T_{(i),j} \times \beta^j$

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

1. ***Debut***
2. $T_{(0)}=0$
3. ***pour i de*** **0** à $\boldsymbol{n-1}$ ***faire***
4. $q_{(i)} = ((T_{(i),0} + (a_i \times b_0)) \times N') \bmod \beta$
5. $T_{(i+1)} = (T_{(i)} + (a_i \times B) + (q_{(i)} \times N)) / \beta$
6. ***Fin pour***
7. ***Si*** $T_{(n)} < N$ ***alors*** $T=T_{(n)}$ ***Si non*** $T=T_{(n)}-N$
8. ***Fin***

Retourne $T= T_{(n)}$

b) Complexité de l'algorithme de la MMME

L'algorithme 5 est défini d'une manière générale pour une base $\beta=2^k$. Ce qui signifie que les chiffres a_i et $q_{(i)}$ sont représentés sur k *bits* dans le système $\{0,1,..., \beta-1\}$. Par conséquent, les calculs des termes $(a_i \times B)$ et $(q_{(i)} \times N)$ seront forcément réalisés en utilisant des multiplieurs $k \times m$ *bits*, où $m= n \times k$. Ainsi, la complexité C_{algo5} de l'algorithme 5 peut être évaluée par l'expression suivante :

$$C_{algo5}= n \times [(2 \times Mul(chiffre \times opérande) + Mul(chiffre \times chiffre)) + (2 \times Add)] + Sub$$

La multiplication $Mul(chiffre \times chiffre)$ de cette expression correspond à la seconde multiplication de la ligne 4 permettant de calculer $q_{(i)}$.

Comme nous l'avons indiqué précédemment, l'algorithme 5 nécessite une soustraction à la fin, vu que les résultats intermédiaires sont majorés par *(2×N)*. Walter montre dans [61] que si les entrées *A* et *B* sont inférieures à *2×N*, le résultat *T* est aussi inférieur à *2×N*. Ce qui signifie que la soustraction finale devient inutile. En revanche cette compatibilité entre les entrées/sorties n'est valable que si une itération supplémentaire est rajoutée.

2.5.2. Multiplication Modulaire de Montgomery Entrelacé sans Soustraction Finale (MMMESF)

L'élimination de l'instruction conditionnelle liée à la soustraction finale est intéressante dans la mesure où non seulement on obtient un gain matériel, mais aussi d'éviter des attaques par canal auxiliaire [27], [61], [62]. En effet, le fait de ne plus avoir de soustraction conditionnelle, nous permet d'exécuter l'algorithme de Montgomery en un temps constant quelle que soit les données en entrée. On peut dire ainsi que la sortie est stable par rapport aux entrées.

a) Algorithme de la MMMESF

L'algorithme de la MMMESF est défini comme suit :

Algorithme 6 –Algorithme de la MMMESF

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

Pré-calculs : $N' = -N_0^{-1} \bmod \beta$ *, avec* $\beta = 2^k$*, et pgcd* $(N, \beta) = 1$

$R = 2^{(n+1)\times k}$ *avec* $k \ge 2$.

Variables intermédiaires : $q_{(i)},\ T_{(i)} = \sum_{j=0}^{n} T_{(i),j} \times \beta^j$

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

1. ***Début***
2. $T_{(0)} = 0$
3. ***pour i de*** **0** à ***n faire***
4. $q_{(i)} = ((T_{(i),0} + (a_i \times b_0)) \times N') \bmod \beta$
5. $T_{(i+1)} = (T_{(i)} + (a_i \times B) + (q_{(i)} \times N)) / \beta$
6. ***Fin pour***
7. ***Fin***

Retourne $T = T_{(n+1)}$

Par rapport à l'algorithme 5, l'algorithme 6 se distingue par les points suivants :

1. Les opérandes *A* et *B* sont représentés sur *(n+1) chiffres*, puisque ces derniers sont inférieurs à $2 \times N$.
2. La constante *R* de Montgomery doit satisfaire la condition $R=2^{(n+1)\times k}$ avec $k \geq 2$.
3. Pour que le résultat *T* de l'algorithme soit inférieur à $2 \times N$, il est nécessaire d'effectuer *(n+1) itérations*.

b) Complexité de l'algorithme MMMESF

L'algorithme retenu dans ce travail pour l'implémentation de la MMM dans un sur circuit FPGA est l'algorithme 6. Notre choix est porté sur l'utilisation de cette variante pour les raisons suivantes :

- Le fait d'avoir enlevé la soustraction finale de l'algorithme de la MMME (algorithme 5). Ceci permet d'optimiser les ressources matérielles, par l'élimination d'un soustracteur dont la taille est de l'ordre de la taille du modulo.
- Exécution de l'algorithme de l'exponentiation modulaire en un temps constant.

Les performances de l'algorithme 6, dépendent de deux principaux paramètres, à savoir, la base de représentation des données β et la taille du modulo *N*.

Pour un modulo de taille *m-bits*, l'algorithme 6 aura besoins de *(m/k)+1 itérations* pour achever le calcul de la multiplication de Montgomery. Entrainant ainsi un temps de calcul $t_m=t_i \times (m/k+1)$, t_i étant le délai d'exécution d'une itération *(i)*. Le nombre d'itérations dépend essentiellement du paramètre *k* qui défini la base β utilisée pour la représentation des données. En effet, en base $\beta=2^k$ avec $k \geq 2$, les chiffres a_i, $q_{(i)}$ et *N'* sont représentés sur *k-bits* dans le système $\{0,1,..., 2^k-1\}$. De ce fait, l'augmentation de *k*, permet d'augmenter la taille de ces chiffres et de réduire ainsi le nombre d'itérations nécessaires à l'exécution de l'algorithme 6. En revanche, l'augmentation en question, bien qu'elle soit considérée comme étant une optimisation de l'algorithme, néanmoins elle engendre une complexité dans l'exécution de chaque itération *(i)*. La complexité en question peut être décrite comme suit :

- A chaque itération *(i)*, le calcul de $q_{(i)}$ nécessite une multiplication *k*×*k-bits*.
- L'obtention du résultat intermédiaire $T_{(i+1)}$ exige l'utilisation de deux multiplications de type *chiffre par opérande*.
- L'influence de la division par β de la ligne 5 est négligeable car, elle est considérée comme étant une division exacte. Autrement dit, la quantité $(T_{(i)}+(a_i \times B)+(q_i \times N))$ est divisible par β. Cette division peut être effectuée par un simple décalage à droite d'un chiffre.

La complexité en termes de nombre d'opérations élémentaires de l'algorithme 6 est évaluée par l'expression suivante :

$$C_{algo6} = (m/k+1) \times [\,(2 \times Mul\ (chiffre \times opérande)) + (2 \times Add) + Mul(chiffre \times chiffre)\,] \approx o(m/k) \qquad (2.14)$$

Lorsque la taille du module est élevée, l'implémentation matérielle de l'algorithme 6 nécessite de longs chemins de propagation de retenue et de longs registres. Ces derniers sont de l'ordre de la taille du modulo. Pour éliminer le problème de propagation de la retenue, des architectures basées sur les réseaux systoliques [27], [28] ou encore sur la représentation en retenue conservée (Carry Save) [25] sont souvent utilisées. Bien que ces implémentations aboutissent à de meilleures performances temporelles, leur coût en termes de ressources matérielles est très élevé. De plus ces méthodes ne s'adaptent pas au type d'implémentation ciblée dans ce travail, où il est question de réaliser sur circuit FPGA un crypto système, basé sur un processeur embarqué.

Pour ce faire, l'approche que nous avons proposée pour l'exécution de l'algorithme 6, sera une version modifiée de ce même algorithme, où toutes les opérations arithmétiques sont exécutées selon un mode sérial. Cette approche a été développée non seulement pour optimiser la surface occupée sur le circuit, mais aussi pour adapter l'exécution de la multiplication de Montgomery aux ressources internes du processeur utilisé.

2.6. Approche proposée pour l'implémentation de la MMM

Dans le but d'atteindre un meilleur compromis entre le temps d'exécution et les ressources nécessaires pour l'implémentation de la MMM, notre approche est basée sur la décomposition des opérande *A, B, N* et des résultats intermédiaires $T_{(i)}$ en *(n+1) chiffres* de taille *k-bits*, comme le montre la figure 2.4 [63], [64], [65], [66].

Autrement dit, toutes les données qui sont mises en exécution sont représentées en base $\beta = 2^k$.

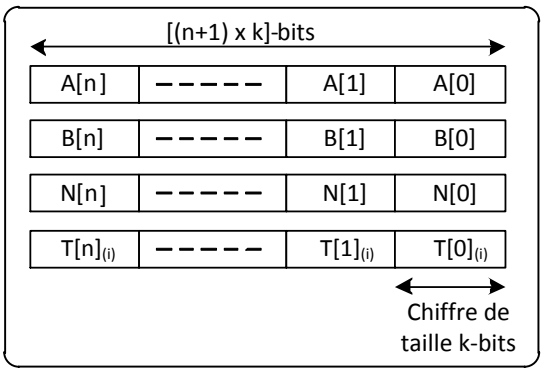


Figure 2.4. Décomposition des opérandes et des résultats intermédiaires en chiffres de taille *k-bits*

On apporte ensuite des modifications dans la partie arithmétique de l'algorithme 6, afin d'adapter son exécution aux ressources internes du processeur. Ainsi, pour éviter l'utilisation de registres internes de grandes tailles, toutes les données sont supposées stocker dans des mémoires embarrées. Les opérations arithmétiques de l'algorithme sont effectuées en mode série sur une précision de taille *k-bits*. Le résultat *T* de la multiplication de Montgomery est obtenu à la fin *chiffre-par-chiffre*.

a) Algorithme de la MMM proposé

L'algorithme proposé pour l'implémentation de la MMM, est défini par l'algorithme 7. Les entrées/sortie de ce dernier sont codées en base 2^k, telle que :

$A = \sum_{i=0}^{n} A[i] \times 2^{i \times k},\ B = \sum_{j=0}^{n} B[j] \times 2^{j \times k},\ N = \sum_{j=0}^{n} N[j] \times 2^{j \times k}$

$T = \sum_{j=0}^{n} T[j]_{(n+1)} \times 2^{j \times k},\ avec\ \ N[n] = 0$

Son exécution est basée sur le calcul itératif des résultats intermédiaires $T_{(i+1)}$, donnés par l'équation suivante :

$$T_{(i+1)} = \frac{T_{(i)} + (A[i] \times B) + (q_{(i)} \times N)}{2^k} \qquad (2.15)$$

Le déroulement de cet algorithme pour calculer $T_{(i+1)}$ repose sur l'utilisation de deux indices *(i)* et *(j)*. *(i)* désigne le $(i)^{éme}$ chiffre de l'opérande *A*. *(j)* permet de positionner les $(j)^{émes}$ chiffres de *B*, de *N* et de $T_{(i)}$, définis par *: B[j], N[j]* et $T[j]_{(i)}$. L'exécution de deux itérations successives (*j)* et (*j+1)* est montrée sur la figure 2.5.

Algorithme 7 – Algorithme proposé pour l'exécution de la MMM en base 2^k

entrées : $A = \sum_{i=0}^{n} A[i] \times 2^{i \times k}$, $B = \sum_{j=0}^{n} B[j] \times 2^{j \times k}$

$N = \sum_{j=0}^{n} N[j] \times 2^{j \times k}$ *avec* $N[n] = 0$

Pré calculés : $N' = -N[0]^{-1} \bmod \beta$, *avec* $\beta = 2^k$, *et pgcd* $(N, \beta) = 1$

$R = 2^{(n+1) \times k}$ *avec* $k \geq 2$.

Variables intermédiaires : $T_{(i+1)} = \sum_{j=0}^{n} T[j]_{(i+1)} \times 2^{j \times k}$,

$P1_{(i)} = \sum_{j=0}^{n} P1[j]_{(i)} \times 2^{j \times k}$, $C1 = \sum_{j=0}^{n} C1[j]_{(i)} \times 2^{j \times k}$, $Z_{(i)} = \sum_{j=0}^{n+1} Z[j]_{(i)} \times 2^{j \times k}$,

$P2_{(i)} = \sum_{j=0}^{n} P2[j]_{(i)} \times 2^{j \times k}$, $C2 = \sum_{j=0}^{n} C2[j]_{(i)} \times 2^{j \times k}$, $L_{(i)} = \sum_{j=0}^{n+1} L[j]_{(i)} \times 2^{j \times k}$,

$cy_1^j, cy_2^j, cy_3^j, cy_4^j$

Sortie: $T = \sum_{j=0}^{n} T[j]_{(n+1)} \times 2^{j \times k} = (A \times B \times R^{-1}) mod\ N$

Début

1. $T_{(0)} = \sum_{j=0}^{n} T[j]_{(0)} \times 2^{j \times k} = 0$
2. ***pour*** i ***de*** 0 ***à*** n ***faire***
3. $P1[0]_{(i)} = A[i] \times B[0]$
4. $Z[0]_{(i)} = P1[0]_{(i)} + T[0]_{(i)}$
5. $q_{(i)} = ((T[0]_{(i)} + (A[i] \times B[0])) \times N') \bmod 2^k$
6. $C1[-1]_{(i)} = C2[-1]_{(i)} = 0$
7. $cy_1^{-1} = cy_2^{-1} = cy_3^{-1} = cy_4^{-1} = 0$
8. ***pour*** j ***de*** 0 ***à*** n ***faire***
9. $(C1[j]_{(i)}, P1[j]_{(i)}) = A[i] \times B[j]$
10. $(cy_2^j, cy_1^j, Z[j]_{(i)}) = P1[j]_{(i)} + C1[j-1]_{(i)} + T[j]_{(i)} + cy_1^{j-1} + cy_2^{j-1}$
11. $(C2[j]_{(i)}, P2[j]_{(i)}) = q_{(i)} \times N[j]$
12. $(cy_4^j, cy_3^j, L[j]_{(i)}) = Z[j]_{(i)} + P2[j]_{(i)} + C2[j-1]_{(i)} + cy_3^{j-1} + cy_4^{j-1}$
13. $T[j-1]_{(i+1)} = L[j]_{(i)}$
14. ***fin pour***
15. $Z[n+1]_{(i)} = C1[n]_{(i)} + cy_1^n + cy_2^n$
16. $L[n+1]_{(i)} = Z[n+1]_{(i)} + cy_3^n + cy_4^n$
17. $T[n]_{(i+1)} = L[n+1]_{(i)}$
18. ***fin pour***

Retourner $\boldsymbol{T = T_{(n+1)}}$

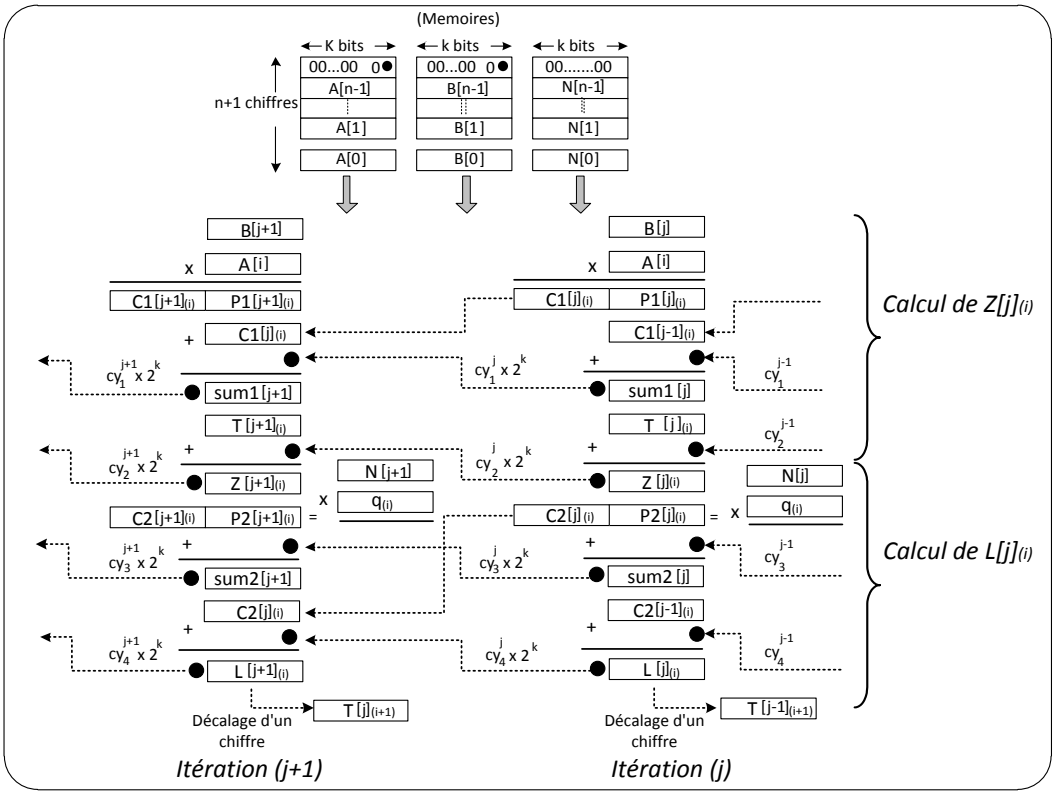


Figure 2.5. Exécution des deux itérations *(j)* et *(j+1)* de l'algorithme 7

A la fin de l'itération *(i-1)*, où *j=n*, l'itération *(i)* commence à partir de la lecture des chiffres *A[i]*, *B[0]*, *N[0]* et $T[0]_{(i)}$. La lecture de *B[j]*, *N[j]* et $T[j]_{(i)}$ se répète tout au long de cette itération, en faisant incrémenter *j* de *0* à *n*.

Pour calculer $T_{(i+1)}$, nous avons introduit deux variables intermédiaires $Z_{(i)}$ et $L_{(i)}$, données respectivement par les expressions (2.16) et (2.17). Les valeurs de leurs $j^{ème}$ chiffres $Z[j]_{(i)}$ et $L[j]_{(i)}$ sont obtenues séquentiellement où, nous procédons d'abord par le calcul $Z[j]_{(i)}$ qui sera suivi par $L[j]_{(i)}$.

$$Z_{(i)} = \sum_{j=0}^{n+1} Z[j]_{(i)} \times 2^{j \times k} = T_{(i)} + (A[i] \times B) \tag{2.16}$$

$$L_{(i)} = \sum_{j=0}^{n+1} L[j]_{(i)} \times 2^{j \times k} = T_{(i)} + (A[i] \times B) + (q_{(i)} \times N)$$

$$= Z_{(i)} + (q_{(i)} \times N) \tag{2.17}$$

A chaque itération *(i)*, où un seul chiffre de l'opérande *A* est sélectionné, on procède en premier lieu à la ligne 5 au calcul de $q_{(i)}$. Ce dernier est déterminé dès la parution du premier chiffre de $Z_{(i)}$ et reste constant durant l'exécution de l'itération *(i)*. $q_{(i)}$ est calculé par l'expression suivante:

$$q_{(i)} = ((T[0]_{(i)} + (A[i] \times B[0])) \times N') \bmod 2^k$$

Puis, on exécute les opérations arithmétiques séquentiellement *chiffre-par-chiffre* sur une précision de *k-bits*, jusqu'à l'accumulation complète du résultat intermédiaire $T_{(i+1)}$. Comme les opérandes sont décomposés sur *(n+1) chiffres*, à chaque itération *(i)* les $(j)^{ème}$ chiffres de $Z_{(i)}$ sont obtenus *chiffre-par-chiffre*, en utilisant la multiplication de la ligne 9 et les additions de la ligne 10 de l'algorithme 7. $L_{(i)}$ est calculé en fonction de $q_{(i)}$ (voir équation (2.17)). Ses $(j)^{ème}$ chiffres sont obtenus par la multiplication de la ligne 11 et les additions de la ligne 12.

Dans cet algorithme, à l'itération *(j)* :

- $C1[j]_{(i)}$ and $C2[j]_{(i)}$ représentent respectivement les chiffres les plus significatifs des produits $A[i] \times B[j]$ et $q_{(i)} \times N[j]$.
- (cy_1^j, cy_2^j) et (cy_3^j, cy_4^j) représentent respectivement les retenues générées par les additions des lignes 10 et 12 à l'itération *(j)*.

Ces retenues, de même que pour $C1[j]_{(i)}$ et $C2[j]_{(i)}$, sont initialisées au début de chaque itération *(i)* et seront introduites dans les additions de l'itération *(j+1)*, comme le montre la figure 2.5.

Les lignes 15 et 16 sont rajoutées dans l'algorithme afin de tenir compte, d'une part, des retenues cy_1^n, cy_2^n, cy_3^n, cy_4^n et, d'autre part, de la valeur de $C1[n]_{(i)}$, obtenues à l'itération *(j=n)*. De ce fait, le résultat intermédiaire $L_{(i)}$ ne sera validé qu'après *(n+2) itérations*. Le terme $C2[n]_{(i)}$ n'est pas pris en considération dans cette itération, car le chiffre le plus significatif du modulo *N* est nul *(N[n]=0)*.

Le calcul du résultat intermédiaire $T_{(i+1)}$ donné par $T_{(i+1)} = L_{(i)}/2^k$ est effectué par un simple décalage de *k-bits* vers les poids faibles de $L_{(i)}$. Ainsi, le $(j)^{éme}$ chiffre de $T_{(i+1)}$ en l'occurrence $T[j]_{(i+1)}$ n'est autre que le $(j+1)^{éme}$ chiffre de $L_{(i)}$, comme ceci est montré sur la figure 2.6.

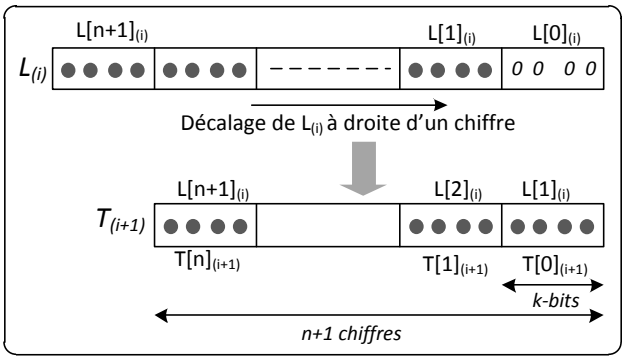


Figure 2.6 Calcul des résultats intermédiaires $T_{(i+1)}$ par décalage à droite de $L_{(i)}$

b) Complexité de l'algorithme 7

La complexité d'exécution de l'algorithme 7 dépend essentiellement de la base $\beta=2^k$, utilisée pour la représentation des données. Celle-ci possède une influence considérable sur la quantité des opérations arithmétique élémentaires (multiplications et additions) qui doivent être effectuées à chaque itération *(j)* de la boucle interne.

En effet, à chaque itération *(i)* de l'algorithme 7, *(n+1)* lectures mémoires et *(n+1)* opérations arithmétiques sont nécessaires pour calculer un seul chiffre de la variable $L_{(i)}$. De ce fait, pour un *modulo N* de taille *m-bits où* $m=n\times k$, le nombre d'opérations élémentaires *Nb_Op* nécessaire pour compléter une itération *(i)*, est défini par l'expression suivante:

$$Nb\_Op = (\frac{m}{k}+1)\times[(2\times Mul(chiffre\times chiffre))+(4\times Add)]+4\times Add \qquad (2.18)$$

Le résultat final de l'opération qui correspond à $T=T_{(n+1)}$, ne sera obtenu qu'après lecture des *(n+1) chiffres* de l'opérande *A*. Ainsi, on peut considérer que la complexité globale C_{alg7} de cet algorithme est équivalente à :

$$C_{algo7} = (\frac{m}{k}+1)\times[Nb\_Op + Mul(chiffre\times chiffre)]\approx o((m/k)^2) \qquad (2.19)$$

Il est à noter que les multiplications *Mul(chiffre×chiffre)* de l'équation (2.19) correspondent aux multiplications de la ligne 5, utilisées pour calculer $q_{(i)}$.

A partir de ces deux dernières expressions, on constate que le paramètre *k* est un élément clé dans l'évaluation des performances temporelles de l'algorithme 7. En effet, pour un modulo de taille *m bits*, quand *k* augmente, le nombre de chiffres *(m/k+1)* associé au codage des opérandes en base $\beta=2^k$ diminue. Par conséquent, le nombre d'itérations *(i)* et le nombre d'opérations élémentaires seront réduits. Cependant, pour une implémentation sur matériel de cet algorithme, en utilisant un bus de données de taille *k-bits*, la complexité des opérateurs arithmétiques de base (multiplieurs et additionneurs) augmente. Ce qui fait augmenter le chemin critique de l'algorithme 7. Ceci se répercute généralement sur la période d'horloge de son l'architecture matérielle qui est proportionnelle à la taille du chemin critique.

A travers cette étude, on constate que la conséquence du paramètre *k* sur les performances d'exécution de l'algorithme 7 est d'une grande importance. Plus encore, l'influence de ce paramètre peut s'avérer comme étant un facteur déterminant dans la complexité d'exécution de l'exponentiation modulaire, puisque cette dernière est

basée sur le calcul de la multiplication modulaire. Ainsi, pour mieux illustrer l'impact du paramètre *k* sur les performances de l'algorithme 2, utilisé pour le calcul de la fonction d'exponentiation modulaire, nous allons présenter dans ce qui suit une version modifiée de l'algorithme 2. Celle-ci est issue principalement de l'approche proposée pour l'implémentation de la MMM.

2.7. Application de l'Algorithme de la MMM proposé au calcul de l'exponentiation modulaire

Pour atteindre l'implémentation logicielle ou matérielle de l'exponentiation modulaire $Y=X^Z mod N$, en utilisant la multiplication de Montgomery et l'algorithme R2L (algorithme 2), ce dernier doit être modifié pour les deux raisons suivantes :

- La multiplication de Montgomery exige la conversion des données d'entrée dans le domaine de Montgomery, puis représentation du résultat dans le domaine classique des nombres.
- L'algorithme 7 repose sur le codage de toutes les données qui sont mises en exécution en base $\beta=2^k$.

a) Algorithme R2L modifié

La version de l'algorithme R2L issue de l'adaptation des deux algorithmes 2 et 7 est donnée par l'algorithme 8.

Les entrées de cet algorithme, représentées par l'exposant *Z*, le message *X*, le modulo *N*, la constantes *N'* et la valeur de $R^2 \ mod \ N$, sont codées en base $\beta=2^k$. La fonction *Montgomery(.)* est associée au calcul de la MMM, en utilisant l'algorithme 7. Celle-ci, en plus des opérandes d'entrée, dépend aussi de la constante *N'* et de la base β. Cette fonction fournit à chaque itération de l'algorithme, deux résultats intermédiaires notés par $S_{(i)}$ et $C_{(i)}$. Ces derniers correspondent respectivement aux résultats intermédiaires de l'élévation au carré et de la multiplication modulaire dans le domaine de Montgomery.

L'exécution de cet algorithme repose sur l'utilisation de trois indices. Ces derniers sont définis comme suit :

- L'indice *(j)* permet de positionner les $j^{ème}$ *chiffres* de toutes des données mises en exécution
- L'indice *(w)* est associé au codage des $j^{ème}$ *chiffre*s de l'exposant *Z* en base 2.

- L'indice *(i)* désigne les itérations de l'algorithme de la méthode binaire R2L (algorithme 2). Son utilisation permet de positionner le $i^{ème}$ bit de l'exposant *Z*.

Algorithme 8–Algorithme proposé pour l'exécution de l'exponentiation modulaire

Entrée: $Z = \sum_{j=0}^{e/k-1} Z[j] \times 2^{j \times k}$, *avec* $Z[j] = \sum_{w=0}^{k-1} z_w^j \times 2^j$

$X = \sum_{j=0}^{n} X[j] \times 2^{j \times k}$, $N = \sum_{j=0}^{n} N[j] \times 2^{j \times k}$, $N[n] = X[n] = 0$

Pré-calculés: $N' = -N[0]^{-1} \bmod \beta$, *avec* $\beta = 2^k$ *et pgcd (N, β)=1 avec k≥2.*

$R^2 \bmod N$ *avec* $R = \beta^{n+1} = 2^{kx(n+1)}$

Variables: $S_{(i)} = \sum_{j=0}^{n} S_{(i)}[j] \times 2^{j \times k}$, $C_{(i)} = \sum_{j=0}^{n} C_{(i)}[j] \times 2^{j \times k}$

Sortie : $Y := \sum_{j=0}^{n-1} Y[j] \times 2^{j \times k} = X^Z \bmod N$

Début

1. $S_{(-1)} = Montgomery(X, R^2 \bmod N)$
2. $C_{(-1)} = Montgomery(1, R^2 \bmod N)$
3. ***pour*** j ***de*** 0 ***à*** $e/k - 1$ ***faire***
4. ***pour*** w ***de*** 0 ***à*** $\boldsymbol{k} - 1$ ***faire***
5. $i = j \times k + w$
6. $S_{(i)} = Montgomery(S_{(i-1)}, S_{(i-1)})$
7. ***si*** $z_w^j = 1$ ***alors*** $C_{(i)} = Montgomery(C_{(i-1)}, S_{(i-1)})$
8. ***sinon*** $C_{(i)} = C_{(i-1)}$
9. ***fin pour***
10. ***fin pour***
11. $Y = Montgomery(C_{(e-1)}, 1)$

Retourner *Y*

Le calcul de l'exponentiation modulaire $Y = X^Z \bmod N$ en utilisant l'algorithme 8 est effectué en trois étapes, à savoir, une étape d'initialisation, une étape d'exécution itérative et une étape de conversion du résultat dans le domaine classique des nombres.

Première étape **:** Elle consiste à initialiser les variables $S_{(i)}$ et $C_{(i)}$, respectivement par les valeurs issues de la conversion dans le domaine de Montgomery du message *X* et de *"1"*. Les résultats de cette étape sont définis par :

$$S_{(-1)} = Montgomery(X, R^2 mod N) = (X \times R) \bmod N$$

$$C_{(-1)} = Montgomery(1, R^2 mod N) = R \bmod N$$

Deuxième étape **:** Elle correspond au calcul itératif des deux multiplications modulaires dans le domaine de Montgomery. Cette étape est basée sur deux boucles imbriquées, définies par les deux indices *(j)* et *(w)*.

- La boucle *j (0 ≤ j ≤ (e /k)-1)* est externe. Elle permet de parcourir les chiffres $Z[j]$ de l'exposant *Z*.
- La boucle *w (0 ≤ w ≤ k-1)* est imbriquée dans la boucle *j*. Celle-ci permet de parcourir les bits z_w^jdu *(j)*^{ème} chiffre de l'exposant *Z*.

Après l'initialisation des variables $S_{(i)}$ et $C_{(i)}$ par les résultats $S_{(-1)}$ et $C_{(-1)}$ de l'étape précédente, le processus itératif procède par la lecture du premier chiffre de *Z*. Ce dernier sera introduit par la suite *chiffre-par-chiffre*, suivant l'indice *(j)*.
En effet, à chaque itération *(w)* de la boucle interne, un décalage à droite d'un bit est effectué sur le *(j)*^{ème} chiffre de *Z* et deux multiplications modulaires de Montgomery sont réalisées. Celles-ci sont définies par :

$$S_{(i)} = Montgomery\left(S_{(i-1)}, S_{(i-1)}\right) = \left(S_{(i-1)}, \times S_{(i-1)} \times R^{-1}\right) mod\ N$$

$$C_{(i)} = Montgomery\left(C_{(i-1)}, S_{(i-1)}\right) = \left(C_{(i-1)}, \times S_{(i-1)} \times R^{-1}\right) mod\ N$$

La première opération correspond à l'élévation au carré de la variable $S_{(i)}$. Celle-ci est exécutée indépendamment du *i*^{ème} *bit* z_w^jde Z. Contrairement au calcul de la variable $C_{(i)}$ qui est renouvelée par le résultat de la seconde opération seulement dans le cas où z_w^jest égal à *"1"*.

Troisième étape : On convertit vers la représentation classique des nombres, le résultat $C_{(e-1)}=(X^Z \times R)\ mod\ N$, issu de l'itération $i=(j\times k)+w=e-1$ où $j=(e/k)-1$ *et* $w=k-1$. Cette étape permet d'éliminer le facteur supplémentaire *R*, où $C_{(e-1)}$ et *'1'* sont utilisés comme opérandes. L'opération ainsi réalisée est définie telle que :

$$Y = Montgomery\left(C_{(e-1)}, 1\right) = (((X^Z \times R) mod\ N) \times 1 \times R^{-1})\ mod\ N$$
$$= X^Z mod\ N$$

A la fin de l'algorithme, le résultat $Y=X^Z mod\ N$ de l'exponentiation modulaire est obtenu *chiffre-par-chiffre*.

b) Complexité de l'algorithme 8

La complexité C_{algo8} de l'algorithme 8 en termes de nombre d'opérations élémentaires dépend de deux facteurs. Le premier est le nombre d'itérations nécessaires pour achever le calcul de l'exponentiation modulaire. Ce facteur est déterminé par la taille *(e)* de l'exposant *Z*. Le second est associé à la complexité C_{algo7} de l'algorithme 7. Celle-ci est définie par l'équation (2.19).

La taille de l'exposant possède un impact sur le nombre de multiplications de Montgomery à effectuer tout au long de l'algorithme 8. En effet, comme l'élévation au carré et la multiplication modulaire sont calculées en parallèle à chaque itération, le nombre requis de multiplications modulaires est de *e+2*. Les deux opérations supplémentaires correspondent aux étapes d'initialisation et de conversion du résultat dans le domaine classique des nombres.

En tenant compte de la complexité C_{algo7}, C_{algo8} est calculée comme suit :

$$C_{algo8} = (e+2) \times C_{algo7}$$
$$= (e+2) \times \left(\frac{m}{k}+1\right) \times \left[Nb_{Op} + Mul(chiffre \times chiffre)\right] \approx (e+2) \times o((m/k)^2) \quad (2.20)$$

Nb_Op est le nombre d'opérations nécessaires pour calculer une itération *(i)* de l'algorithme 7. *Nb_Op* est donné par l'équation (2.18).

L'expression (2.20) montre que la complexité de l'algorithme 8, dépend non seulement de la taille *(e)* de l'exposant *Z*, mais aussi du paramètre *k* qui détermine la base β utilisée pour la représentation des données. De ce fait, un des objectifs de ce travail est d'étudier l'impact de la base β non seulement sur la complexité d'implémentation de la multiplication modulaire en utilisant l'algorithme 7, mais aussi sur les performances d'exécution de la fonction d'exponentiation modulaire.

2.8. Conclusion

Dans ce chapitre, nous avons présenté quelques méthodes de calculs qui simplifient l'exécution de l'exponentiation modulaire. L'étude algorithmique de ces méthodes a révélé que l'utilisation de l'algorithme binaire "Right to Left" pour l'implémentation de cette fonction, permet non seulement d'accélérer son exécution, mais aussi d'aboutir à un meilleur compromis entre son temps de calcul et les ressources matérielles requises. Ceci est dû principalement à l'exécution parallèle de deux opérations de base, à savoir l'élévation au carrée modulaire et la multiplication modulaire.

Dans ce chapitre, il a été aussi question de présenter une méthode efficace pour le calcul de la multiplication modulaire, en l'occurrence la multiplication de Montgomery. En effet, en vertu du rôle attribué à cette opération dans l'exécution de l'exponentiation modulaire, une étude concernant les deux variantes les plus utilisées pour le calcul de la MMM et leurs complexités algorithmiques a été présentée. A l'issue de cette étude, nous avons constaté qu'en plus de la taille du modulo, la base

de représentation des données est un facteur clé qui peut avoir une influence considérable dans les performances d'exécution de la multiplication en question. De ce fait, dans le but d'atteindre un compromis optimum entre les ressources matérielles requises pour l'implémentation de cette opération et le temps d'exécution, l'approche que nous avons proposé est basé essentiellement sur :

- L'utilisation d'une base élevée qui permet de réduire le nombre d'itérations.
- La décomposition des opérandes et des résultats intermédiaires qui pourra notamment optimiser la taille des registres internes et des opérateurs arithmétiques.

Dans le chapitre 4, nous présenterons les différentes approches proposées pour l'implémentation de ce crypto système, en utilisant un circuit FPGA de Xilinx. Auparavant, nous allons donner des notions utiles à notre implémentation, concernant les systèmes embarqués sur FPGA ainsi que leur méthodologie de conception.

Chapitre 3

Systèmes embarqués sur circuit FPGA et leur méthodologie de conception

3.1. Introduction

Depuis son apparition, le transistor a permis une évolution spectaculaire dans l'industrie de fabrication des systèmes électroniques. 1965, Gordon Moore a exprimé une loi qui prédite que le nombre de transistors sur une puce de silicium double tous les dix-huit mois [9], [67]. En conséquence, les systèmes électroniques sont devenus de plus en plus puissants et moins coûteux. Parallèlement à cela, bien que la capacité d'intégration des transistors augmente, néanmoins le potentiel de son exploitation par les concepteurs dans leur productivité reste inférieur. Ceci est dû principalement aux outils et aux techniques classiques de conception qui ne connaissent pas la même évolution.

L'évolution des performances que dicte la loi de Moore, résulte de la mise en exploitation progressive d'un phénomène que recèle le silicium. Lorsque la distance entre transistors est réduite, la vitesse de traitement s'accroit. En revanche, cette croissance peut être ralentie ou même arrêtée sous les effets de bruits parasites. De ce fait, l'amélioration des performances des systèmes ne peuvent être atteinte que par la parallélisation de leur exécution. De plus, si dans le passé, les concepteurs se sont souciés à l'optimisation de la fréquence du traitement à un bas niveau d'abstraction, les efforts et les préoccupations actuelles se concentrent sur l'intégration et l'implémentation des systèmes électroniques complexes sur une même puce de silicium. Ainsi, des notions de systèmes embarqués se développent de plus en plus. Ces derniers sont constitués généralement par un ou plusieurs processeurs et un ensemble de composants qui fonctionnent autour de ces processeurs [7], [68].

Ce chapitre est consacré à la présentation des systèmes embarqués et leur méthodologie de conception sur circuit FPGA de Xilinx. Il est reparti sur trois parties. La première présente des généralités sur ces systèmes et l'approche de conception logicielle/matérielle. La seconde

partie est consacrée à la présentation de leur l'architecture sur circuit FPGA de Xilinx. Dans la troisième partie, nous présentons la méthodologie de conception de ces systèmes en utilisant le processeur Microblaze de Xilinx.

3.2. Système embarqué sur puce

L'architecture d'un système embarqué dans une puce combine un ou plusieurs processeurs pour le contrôle du système, un bus système, un générateur d'horloge, une mémoire interne, un ensemble de périphériques ou coprocesseurs dédiés à des tâches bien spécifiques, des interfaces d'entrées/sorties, etc [67].

Un système embarqué est autonome et ne possède pas d'entrées/sorties standards tels qu'un clavier ou un écran d'ordinateur. Contrairement à un PC, l'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). Un exemple d'architecture simplifiée d'un système embarqué est montré sur la figure 3.1.

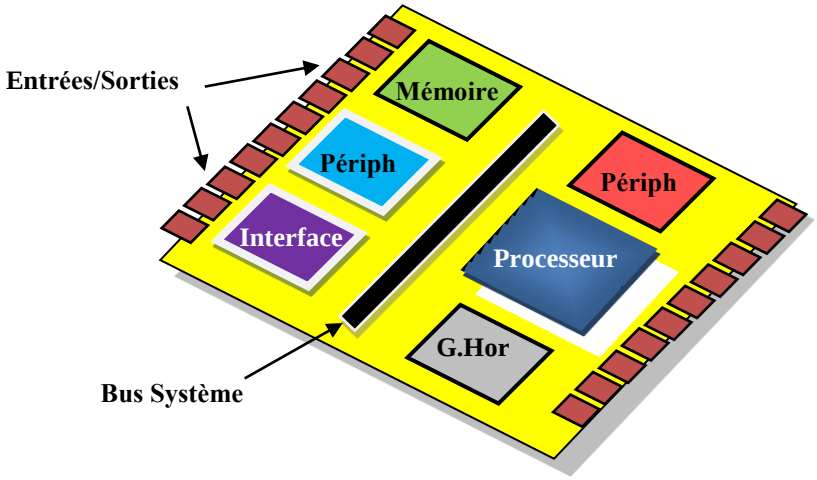


Figure 3.1. Architecture d'un système embarqué sur puce.

3.2.1. Plateformes d'implémentation de systèmes embarqués

Pour atteindre une conception d'un système sur puce, il est primordial de procéder par la sélection d'une plateforme matérielle d'implémentation. La sélection en question 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* [5] *?*

En effet, pour les systèmes ne nécessitant pas de grandes capacités de traitement de données, les microcontrôleurs tels que le 8051 d'Intel peuvent être un bon choix. Si les besoins en calculs sont plus importants, les microprocesseurs plus puissants ou les DSPs (Digital Signal Processor) peuvent être considérés. Ce type de solution est très flexible puisqu'il est basé principalement sur une écriture de programmes.

Dans le cas où l'optimisation ciblée est la conception d'un système avec 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 ASICs (Application Specific Integrated Circuit). Ces derniers sont recommandés pour une large série de fabrication. La seconde option est de recourir aux circuits programmables de type FPGAs (Field Progammable Gate Array). Ces circuits sont généralement conseillés pour le prototypage et la production en faible série. Dans ces deux types de plateformes, on parle souvent de SoC (System on Chip) et de système sur circuits programmables PSoC (Programmable System on Chip). Les différents types de plateformes d'implémentations sont montrés sur la figure 3.2 [5], [68].

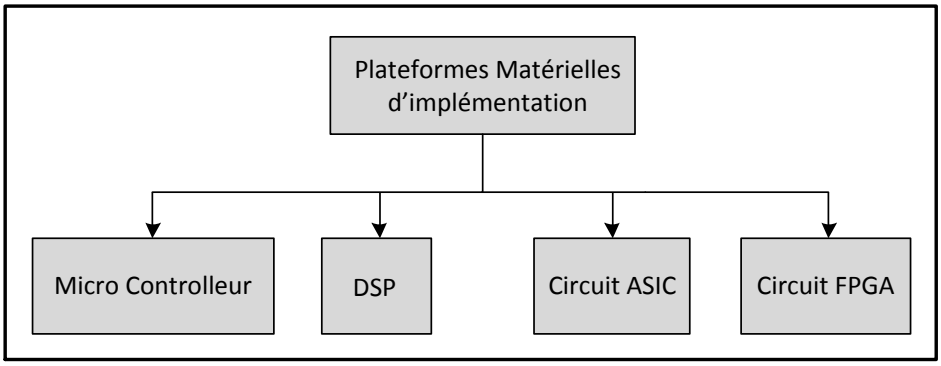


Figure 3.2. Plateformes d'implémentations de systèmes embarqués

3.2.2. Approche de conception logicielle/matérielle

Les développeurs d'applications en logiciel et les concepteurs de systèmes sur matériel ont toujours mené dans le passé leurs activités d'une manière indépendante. En effet, le programmeur du logiciel, pense n'avoir aucun souci sur l'exécution de son produit au bas niveau du matériel utilisé. De même, les préoccupations majeures des concepteurs de systèmes sur matériel, ont toujours orienté leurs efforts vers des optimisations, parfois à un bas niveau d'abstraction. Ceci est dans le but de réaliser des produits performants, en termes de temps d'exécution et de surface occupée sur le circuit. Néanmoins, les implémentations réalisées dans cette seconde catégorie ne favorisent pas la flexibilité du système, puisqu'elles sont dédiées généralement à des applications bien spécifiques.

L'approche de conception logiciel/matériel recommandée notamment pour la réalisation des systèmes embarqués de type SoC où PSoC, a permis de réexaminer les frontières entre le logiciel et le matériel. En effet, cette approche est une méthode de développement basée sur la combinaison conjointe des deux ressources logicielle et matérielle. La partie logicielle est exploitée pour sa flexibilité. La partie matérielle est utilisée pour augmenter les performances

temporelles. Le partitionnement d'une application sur les deux ressources s'effectue généralement en tenant compte des performances à atteindre, c'est-à-dire surface occupée, temps d'exécution, puissance consommée et coût de développement. La migration des tâches vers le matériel est toujours possible jusqu'à ce que les contraintes de performances soit atteintes.

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, en assembleur. Cette partie est exécutée par le processeur embarqué. Pour la partie matérielle, celle-ci est décrite par l'utilisation d'un langage de description matérielle, tel que le VHDL ou le Verilog [7].

Pour mieux illustrer le concept d'implémentation logicielle/matérielle, un exemple d'une architecture simple composée d'un processeur et d'un coprocesseur est montré sur la figure 3.3.

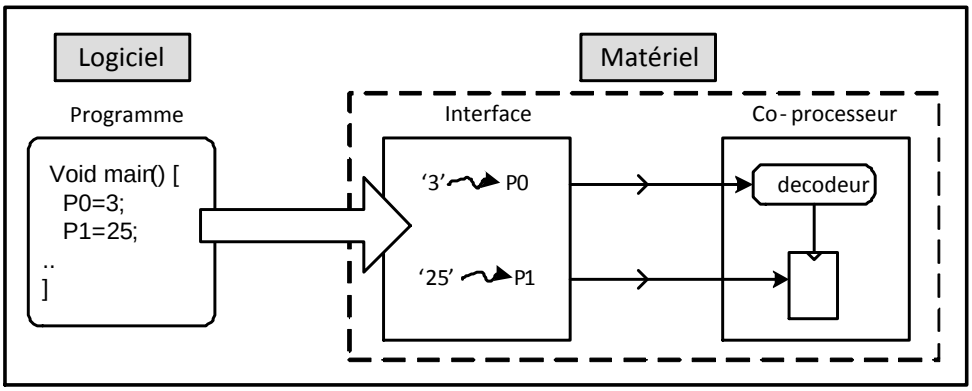


Figure 3.3. Modèle d'implémentation logicielle/matérielle

Le logiciel est exécuté sur un processeur embarqué. La partie matérielle est représentée par une interface et un co-processeur. La communication entre ce dernier et le processeur est établie grâce aux deux ports *P0* et *P1* de l'interface. Ces derniers permettent d'accéder au coprocesseur de l'extérieur en utilisant un programme 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 à recevoir la donnée fournie par le processeur. Ainsi, quand la donnée *"25"* apparait sur le port *P1*, elle ne sera transmise au registre que si le décodeur reconnait la donnée *"3"* qui se présente sur *P0*.

Les plateformes SoC et PSoC associées au concept logiciel/matériel sont avantageuses pour plusieurs raisons [7]:

- Les performances temporelles sont plus élevées par rapport à une implémentation logicielle.
- Possibilité de modifier une application par rapport à ses versions ultérieures.
- La flexibilité du système lui donne une solution réutilisable qui fonctionne pour de multiples applications. C'est-à-dire, si on aura besoin de changer une application, il suffit uniquement de modifier la partie logicielle. En conséquence, les coûts de conception par application diminuent.
- Lors de la conception d'une application donnée, les deux ressources logicielle et matérielle sont généralement complémentaires. En d'autre terme, quand le contrôle de l'application est assez complexe, il est recommandé d'investir des efforts dans la partie logicielle. Quand il est question d'améliorer les performances temporelles, il est préférable de rajouter du matériel.

Une courbe montrant les performances du système en fonction du coût de conception est montrée sur la **figure 3.4.**

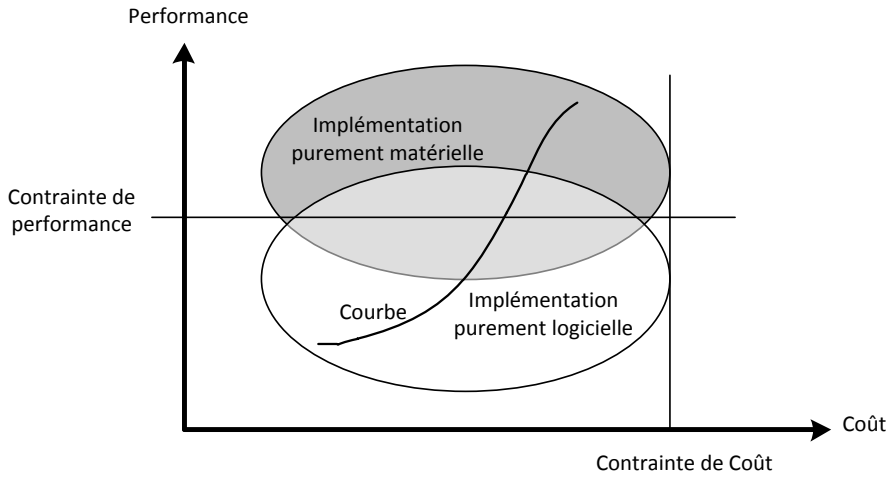


Figure 3.4. Performances des approches d'implémentation en fonction du coût [69].

Cette courbe montre que plus le coût augmente, l'implémentation est orientée vers du matériel pur. Néanmoins, le coût en question possède toujours une limite.

D'une manière générale, avant d'entamer l'implémentation et la réalisation du système, il est nécessaire d'effectuer un partitionnement logiciel/matériel équitable des différentes fonctions qui le constituent.

3.3. Système embarqué sur circuit FPGA

Avec le développement sans cesse de la technologie de fabrication des circuits FPGAs, plusieurs alternatives sont rendues possibles pour l'intégration des systèmes embarqués sur

puce, ou plus encore, des réseaux de communications qui combinent plusieurs processeurs à la fois. Les FPGAs actuels intègrent, en plus d'une panoplie de ressources logiques et arithmétiques, de la mémoire et des processeurs sous forme d'IPs [8], [68].

Il est évident que les FPGA sont aujourd'hui utilisés dans un large éventail d'applications. D'ailleurs, intégrer un système embarqué dans un circuit FPGA est la nouvelle approche pour profiter des avantages offerts par ce de type de circuits tel que la rapidité d'intégration. De plus, les circuits FPGAs sont caractérisés par leur reconfigurabilité. Autrement-dit, une fois la conception d'une application adaptée de manière optimale, il n'y a aucune raison pour conserver cette conception éternellement sur le circuit. Ainsi, pour de nombreuses applications, un FPGA peut être utilisé comme une plateforme de prototypage pour les premières implémentations, avec l'intention de la remplacer un jour par une plateforme plus performante.

3.3.1. Processeurs embarqués sur FPGA

Lorsque l'on conçoit un système numérique complexe, on met en œuvre généralement un processeur embarqué. Ce processeur est :

- Soit un bloc IP : on parle de processeur softcore.
- Soit, il est implémenté sur silicium: on parle de processeur hardcore. Généralement ce type de processeur est plus performant que le type précédent.

Le choix d'un processeur pour le PSoC 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.

Généralement, on privilégie les processeurs softcore pour s'affranchir des problèmes d'obsolescence et pour pouvoir bénéficier facilement des évolutions apportées. La description d'un processeur softcore est effectuée avec 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 implémenté que dans les circuits FPGAs du fondeur approprié.

On trouvera principalement au niveau des processeurs softcores propriétaires, le processeur NIOS II d'Altera et le processeur Microblaze de Xilinx [8], [68].

A noter qu'un processeur hardcore Power PC d'IBM a été implémenté aussi par Xilinx sur quelques familles de ses circuits FPGA. Les familles en question sont : Virtex-2 Pro, Virtex-4 FX et Virtex-5 FX. Tout récemment, Xilinx a intégré sur sa nouvelle génération de circuits FPGA, un processeur hardcore, en l'occurrence le processeur ARM.

3.3.2. 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 sont souvent utilisés. En effet, les fondeurs de ce type de circuits mettent 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 mise à l'étude dans ce travail est la carte Genesys de Digilent [19]. Cette dernière comporte un ensemble de composants qui permettent de développer et de vérifier le fonctionnement correct d'une application PSoC. Elle est équipée du circuit FPGA XC5VLX50T de la famille Virtex-5 [70].

A côté 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 RAM de taille *256-MBytes*, deux ports USBs, des interrupteurs, des LEds, un port Ethernet, un afficheur LCD, un générateur d'horloge programmable qui peut atteindre une fréquence de *400Mhz*, etc. La carte de prototypage Genesys est montrée sur la figure 3.5.

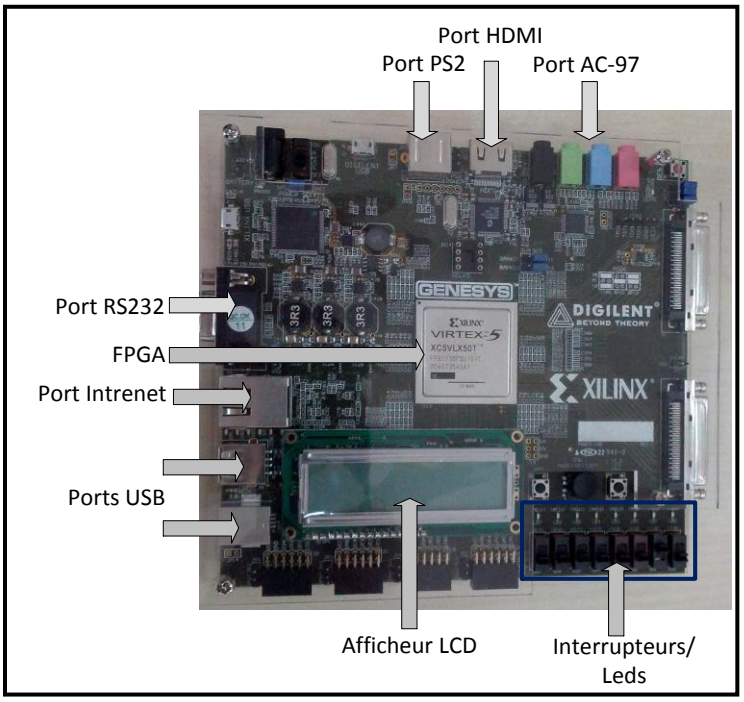


Figure 3.5. Carte de prototypage Genesys

L'implémentation du processeur Microblaze en tant que contrôleur principal du PSoC, nécessite, selon les besoins du concepteur, l'ajout de périphériques sur le même circuit. Ces derniers jouent le rôle d'interface entre le processeur et les autres composants de la carte. Ils

sont connectés autour de Microblaze sur l'un des deux bus OPB (On Chip Peripheral Bus) ou PLB (Processor Local Bus), comme le montre la figure 3.6.

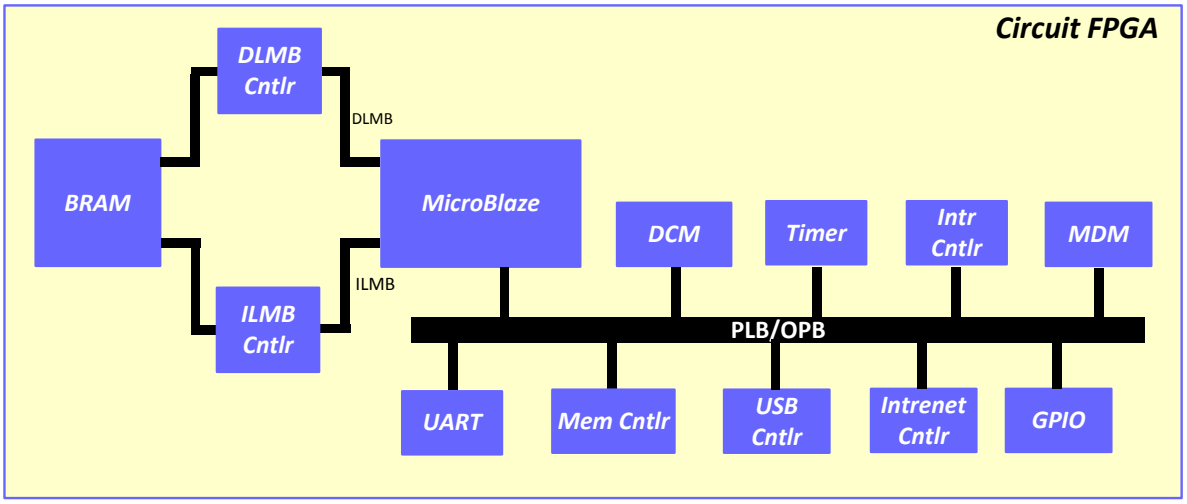


Figure 3.6. Architecture d'un PSoC à base du processeur Microblaze

Parmi ces périphériques, on peut retrouver un UART (Universal Asynchronous Receiver/Transmitter) pour la communication avec le port RS232 de la carte, un contrôleur de mémoire externe (Mem Cntlr), des contrôleurs de ports USB et Ethernet (USB et Ethernet Cntlr) et des GPIOs (General Purpose Input Output). Ces derniers permettent de communiquer avec les LEDs, les interrupteurs et l'afficheur LCD.

D'autres périphériques de base sont indispensables au fonctionnement du système. Les périphériques en question sont constitués de :

- Une mémoire BRAM et ses contrôleurs (DLMB Cntlr et ILMB Cntlr).
- Une DCM (Digital Clock Manager) dont le rôle est la gestion du signal horloge dans le circuit FPGA.
- Un Timer pour calculer les performances temporelles du système.
- Un contrôleur d'interruption (Intr Cntlr).
- Un MDM (MicroBlaze Debug *Module)* pour vérifier le fonctionnement correct des applications.

3.4. Méthodologie de conception d'un système embarqué à base du processeur Microblaze

L'intérêt majeur de réaliser des applications embarquées sur une plateforme PSoC est de pouvoir implémenter autour du processeur une IP matérielle personnalisée. Cette dernière peut être une unité de calcul performante ou encore, un co-processeur dédié à une application

spécifique. Cette IP sera implémentée sur matériel et peut en effet être soumise à des modifications, afin de la ré-implémenter avec des versions plus performantes.

L'architecture de base d'un PSoC comportant une IP personnalisée est montrée sur la figure 3.7. Cette architecture est constituée du processeur Microblaze, de la mémoire BRAM, d'une DCM, d'un UART et d'un composant nommé *"My_Component"*. Ce dernier est considéré comme une enveloppe qui englobe l'IP personnalisée. Le rôle attribué à l'UART est le transfert bidirectionnel des données à travers le port RS232.

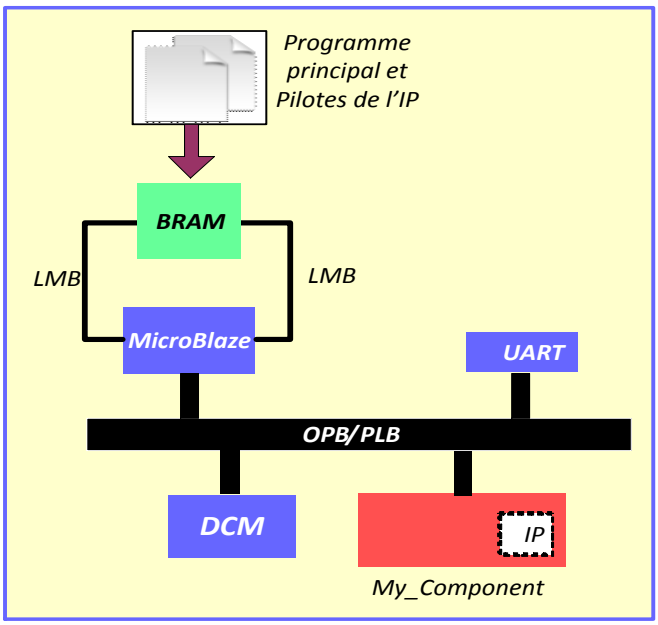


Figure 3.7. Implémentation d'une IP matérielle dans un PSoC

Dans cette architecture, la DCM et l'UART sont considérés comme étant des composants. Leurs rôles sont respectivement la gestion du signal d'horloge et la communication avec le Port RS232 de la carte de prototypage Genesys.

D'une manière générale, le cycle de conception de ce système comporte trois étapes, définies comme suit [7] :

1. Conception de la partie matérielle. Celle-ci consiste en la configuration du processeur Microblaze, de ses périphériques de base et l'implémentation de l'IP.
2. Développent de la partie logicielle de l'application. Celle-ci nous permet de gérer et de synchroniser en logiciel la communication entre les différents composants qui constituent la partie matérielle du système embarqué. Dans cette partie, il est aussi question de développer les pilotes logiciels de l'IP. Ces derniers sont utilisés par le processeur pour contrôler son fonctionnement.
3. Chargement du système sur circuit FPGA.

Pour parvenir à la réalisation d'un PSoC, Xilinx a développé un ensemble d'outils regroupés dans deux logiciels ISE (Integrated Software Environment) [71] et EDK (Embedded Development Kit) [72]. Les outils d'ISE sont utilisés par EDK pour la synthèse et l'implémentation de la partie matérielle du système sur le circuit FPGA. L'interface graphique d'EDK est nommée XPS (Xilinx Platform Studio). Elle inclut les outils de programmation personnalisés par Xilinx pour Microblaze et le processeur hardcore Power PC d'IBM. Les outils en questions sont : GCC (GNU Compiler Collection) pour la compilation des pilotes écrits en langage C/C++ et GDB (GNU Debugger) pour débugger les applications [72]. Dans ce qui suit, nous allons détailler les étapes citées ci-dessus pour la conception et à l'implémentation d'un PSoC sur FPGA en utilisant le processeur Microblaze.

3.4.1. Configuration de Microblaze et de ses périphériques de base

Dans cette première étape, on procède en premier lieu dans l'interface graphique XPS par :

- Choisir Microblaze en tant que processeur principal.
- Définir la fréquence système.
- La polarisation du signal d'initialisation système (actif à l'état haut ou bas).
- Sélection des périphériques de base, tels que l'UART et un module pour débuguer (en software ou en hardware).
- Sélection de la taille de la mémoire BRAM (*8-kbits*, 16-kbits, *32-kbits* ou *64-kbits*).
- Configuration des paramètres du protocole de communication RS232.
- Sélection ou pas de l'unité arithmétique à virgule flottante et de la mémoire cache.

Une fois cette configuration achevée, il sera question ensuite de sélectionner les composants de la carte à inclure dans l'application, tels que : 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 être ajoutés lors de la conception.

A la fin de cette première étape, EDK génère trois fichiers principaux, en l'occurrence, *system.mhs* (Microprocessor Hardware Specification), *system.mss* (Microprocessor Software Specification) et *system.ucf*. Le premier fichier décrit comme boite noire, les différents composants qui constituent le PSoC, les interconnexions et le mapping des adresses. Le second fichier comporte les noms et les versions de l'OS (Operating System) du processeur. Il est constitué aussi par les pilotes logiciels de chaque périphérique et leurs versions. Le dernier

fichier porte les informations concernant l'emplacement des signaux d'entrées/sorties sur les pins du circuit FPGA.

3.4.2. Implémentation d'une IP matérielle autour du processeur

Dans cette étape, il est question d'implémenter dans la partie matérielle du PSoC une IP personnalisée, une fois que la conception de cette dernière soit achevée dans ISE. Généralement, le transfert des données entre les deux parties (processeur et IP) nécessite un système de communication assez complexe. En Général, l'implémentation d'une IP autour d'un processeur peut se faire en utilisant l'une des trois approches suivantes [7] :

1. Une implémentation avec des instructions personnalisées.
2. Une implémentation en coprocesseur.
3. Une implémentation via le bus système et la mémoire BRAM.

Dans le premier type d'implémentation, l'IP est intégré directement dans le processeur. Cette approche ne peut pas être appliquée dans le cas du processeur Microblaze, car son code VHDL est fermé. De plus, l'intégration de l'IP à l'intérieur du processeur entraine des modifications qui peuvent avoir une influence sur son fonctionnement et sur ses performances. Ces modifications affectent principalement le pipeline du processeur. Autrement dit, le chemin critique qui constitue le plus long étage du pipeline, peut dans certain cas devenir plus élevé. Ce qui se traduit par la diminution de la fréquence du système. Pour ce faire, Xilinx offre une option sur le processeur Microbalze qui permet de connecter l'IP par une implémentation en coprocesseur, sans passer par le bus principal. Cette solution peut être réalisée en utilisant les interfaces FSL (Fast Simplex Link) qui sont des ports de communication point-à-point [73]. Bien que cette solution soit souvent recommandée pour un haut débit de transfert de données ; en revanche, le fait d'avoir une seule ligne de communication entre les deux parties, réduit de la flexibilité dans contrôle de l'IP à partir du processeur. Autrement dit, la ligne de communication est utilisée à la fois pour :

- Le transfert des données entre le processeur et l'IP.
- Le contrôle de l'IP à partir du processeur.
- La génération des interruptions à partir de l'IP.

Le troisième type d'implémentation est le plus répandu, bien qu'il soit considéré comme étant plus lent et plus complexe. Cette complexité relève principalement des intervenants mis en exécution, à savoir le bus d'interconnexion et la mémoire BRAM. Ce système de communication est basé sur l'utilisation d'une interface configurable qui offre plusieurs

options aux concepteurs. Ces options ne sont rien d'autres qu'un ensemble de composants, permettant une meilleure flexibilité pour le transfert des données et pour le contrôle de l'IP à partir du processeur.

Dans ce travail, nous nous sommes intéressés à l'utilisation de ce type de système de communication.

3.4.3. Implémentation de l'IP via le bus système et la mémoire BRAM

Dans cette implémentation, la communication entre l'IP et le processeur Microblaze est assurée par le bus système. Pour établir la communication avec l'IP à partir de la partie logicielle de l'application, un espace mémoire lui est alloué dans la mémoire locale du processeur (BRAM). L'identification de l'IP par le processeur est effectuée en déclarant sa basse adresse mémoire dans la description de ses pilotes, stockés dans la mémoire BRAM. Ce genre d'implémentation est nommé *memory-mapped interface* [7]. L'architecture de cette approche d'implémentation est montrée sur la figure 3.8.

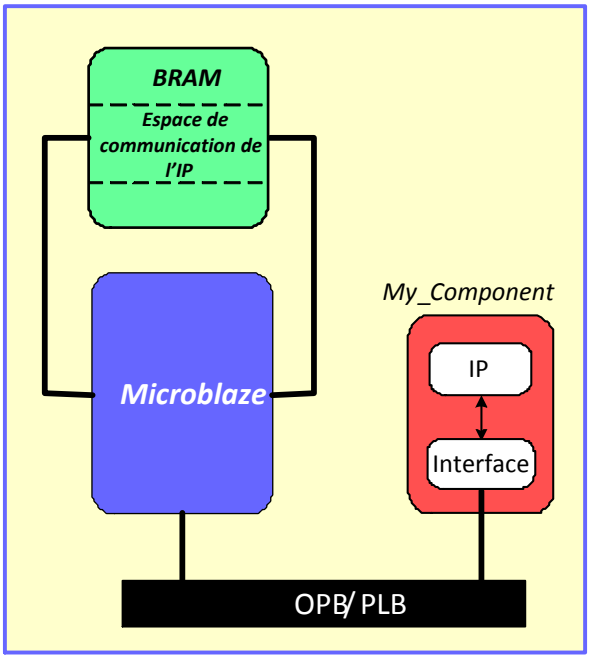


Figure 3.8. Architecture d'une implémentation via le bus système et la mémoire BRAM

D'une manière générale, l'implémentation d'une IP autour du processeur nécessite une conception conjointe de deux parties, à savoir, une partie matérielle et une partie logicielle, comme ceci est montré sur la figure 3.9.

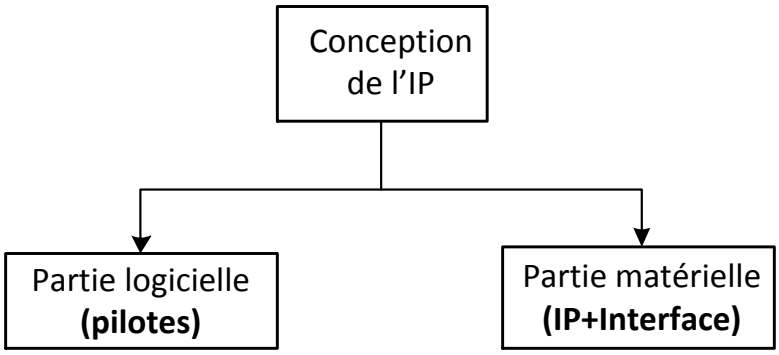


Figure 3.9. Conception conjointe logicielle/matérielle de l'IP

La partie matérielle est constituée de l'IP et d'une interface, dont le rôle est d'assurer la communication entre l'IP et le bus système. La partie logicielle constitue les pilotes de l'IP. Ces derniers permettent de contrôler son exécution à partir du programme principal de l'application.

Le processus de réalisation de ces deux parties est une tâche assez complexe et peut prendre un temps considérable dans le cycle de conception du système. Pour cela, Xilinx fournit dans son système de génération de ses IP's, une interface optimisée avec ses pilotes. Cette interface nommée *IPIF (IP Interface)*, a pour objectif de décoder le protocole du bus et de gérer le transfert des données entre le processeur et l'IP [74], [75]. Celle-ci est configurée puis générée dans EDK. A noter que les pilotes de cette interface sont liés aux deux systèmes d'exploitation, en l'occurrence ″Standalone″ et ″Xilkernel″ [76], [77], [78], [79], développés par Xilinx aux processeurs Microblaze et Power PC.

3.4.4. IP InterFace (IPIF)

L'utilisation de l'IPIF nous permet non seulement de réduire le temps de conception du système, mais aussi de ne pas nous soucier du protocole de communication du bus. Cette interface est un softcore paramétrable par rapport aux options qu'elle offre. Ces dernières ne sont rien d'autre qu'un ensemble de composants internes qui nous permettent d'avoir l'accès à l'IP à partir de la partie logicielle de l'application. En d'autres termes, la communication ne se fait plus avec l'IP, mais avec les composants internes de l'interface IPIF.

Les composants de cette interface sont définis comme suit [74]:

- 32 registres de tailles paramétrables (*8-bits*, *16-bits* ou *32-bits*). Ces registres peuvent être configurés en entrée ou en sortie.

- Deux mémoires FIFOs, nommées Write FIFO et Read FIFO. Celles-ci permettent de transférer des données de *32-bits* respectivement du processeur vers l'IP et dans le sens inverse. La profondeur de ces FIFOs est configurable de *1* à *512*.
- DMA (Direct Memory Acces). L'utilisation de cette option permet à l'IP d'avoir un accès direct à la mémoire BRAM.
- Générateur de signal d'interruption.
- Un aiguilleur du bus de données.

Le schéma bloc de l'*IPIF* est montré sur la figure 3.10.

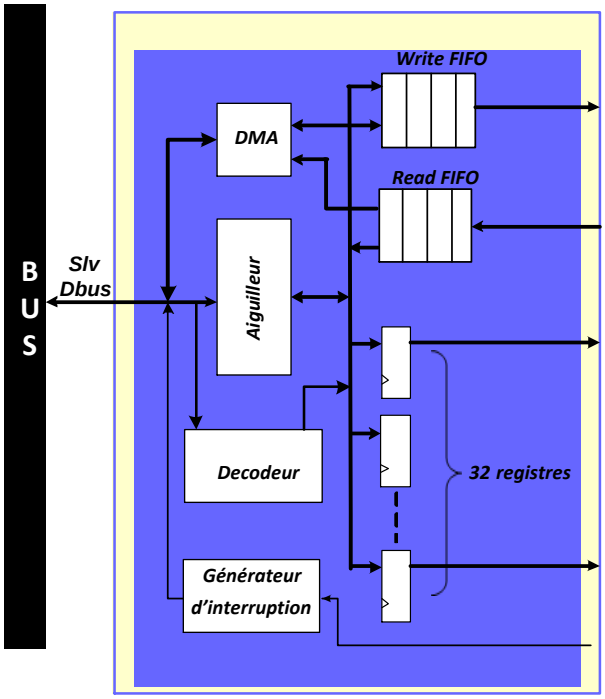


Figure 3.10. Architecture interne de l'interface *IPIF*

Une fois avoir choisi les composants nécessaires au fonctionnement de l'IP, leur sélection est effectuée à partir de la partie logicielle, à travers les adresses qui leurs sont attribuées. Les adresses en question sont déclarées dans les pilotes de l'interface IPIF. En plus des composants que nous avons cités ci-dessus, l'interface *IPIF* comporte un décodeur connecté directement au bus système. Sa tâche est de décoder l'adresse générée par Microblaze pour la sélection de l'un des composants choisis.

L'accès aux composants internes de l'interface *IPIF* est effectué en utilisant des fonctions écrites en langage C et qu'on retrouve également dans ses pilotes. Quelques-unes de ces fonctions sont résumées dans le tableau 3.

Tableau 3. Fonctions des pilotes de l'interface IPIF écrites en langage C

Fonction en langage C	Description
MY_Component_mWriteSlaveReg1(baseaddr,0,data)	Ecriture de la donnée ″data″ dans le registre d'indice 1
MY_ComponentReadSlaveReg1(baseaddr)	Lecture de la donnée du registre d'indice 1
MY_Component_mWriteToFIFO(baseaddr,0,data)	Ecriture de la donnée ″data″ dans la mémoire Write FIFO
MY_Component_mReadFromFIFO(baseaddr,0,data)	Lecture de la donnée de la mémoire Read FIFO

3.4.5. Communication IPIF-IP

Le transfert des données entre l'interface *IPIF* et l'IP est organisé sur un ensemble de signaux constituant un système de communication. Ce dernier est nommé *IPIC (IP InterConnect)*. Ces signaux sont détaillés dans [74].

Généralement, pour respecter le protocole de communication du bus, un circuit d'adaptation intermédiaire entre l'interface *IPIF* et l'IP doit être conçu et rajouté. Ce circuit est nécessaire dans le cas où l'IP est réalisée avec une réception parallèle des données et que l'interface *IPIF* ne possède qu'un seul bus pour la transmission de ces données. De même, si la taille du bus de données de sortie de l'IP est supérieure à la taille du bus de donnée d'entrée de l'*IPIF*, l'adaptation de ces bus est assurée par le même circuit. L'architecture matérielle de toute la chaine de communication est montrée sur la figure 3.11.

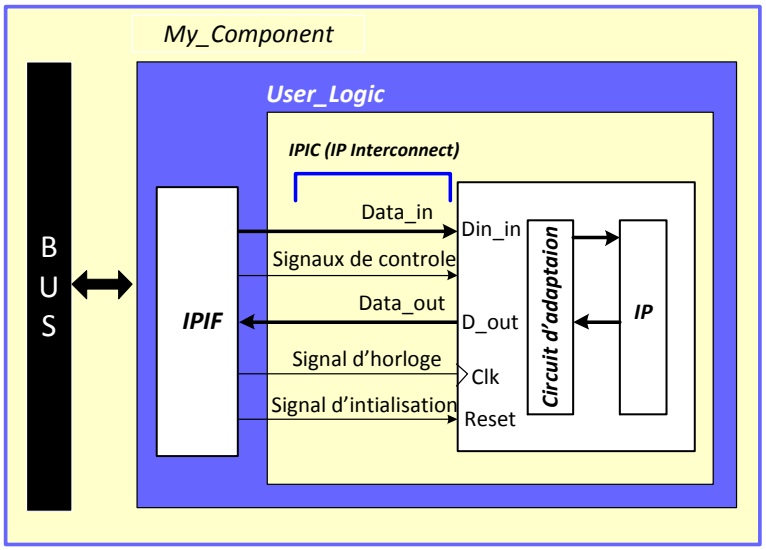


Figure 3.11. Architecture de la chaine de communication IP-Bus

La connexion de l'IP et la configuration de l'interface *IPIF* sont réalisées en utilisant dans EDK l'outil *Create and Import Peripheral Wizard.* Cet outil assure la génération de deux fichiers VHDL et des pilotes qui comportent les fonctions du tableau 3. Les deux fichiers VHDL seront utilisés pour compléter la conception de la partie matérielle du composant *My_Component*.

Le premier fichier VHDL est une enveloppe dont la nomenclature est laissée au choix du concepteur. Dans ce cas, ce fichier est nommé *My_component.vhd.* Le second fichier nommé par défaut *user_logic.vhd,* est constitué par une description VHDL d'un composant qui englobe l'IP et le circuit d'adaptation. Les entrées/sorties de ce composant seront connectées à l'interface *IPIF* à travers le système de communication *IPIC*. Dans *My_component.vhd,* on retrouve ainsi la déclaration du composant *user_logic* et le modèle VHDL de l'interface *IPIF*. La représentation hiérarchique des fichiers VHDL qui correspondent aux composants de la chaine de communication est montrée sur la figure 3.12.

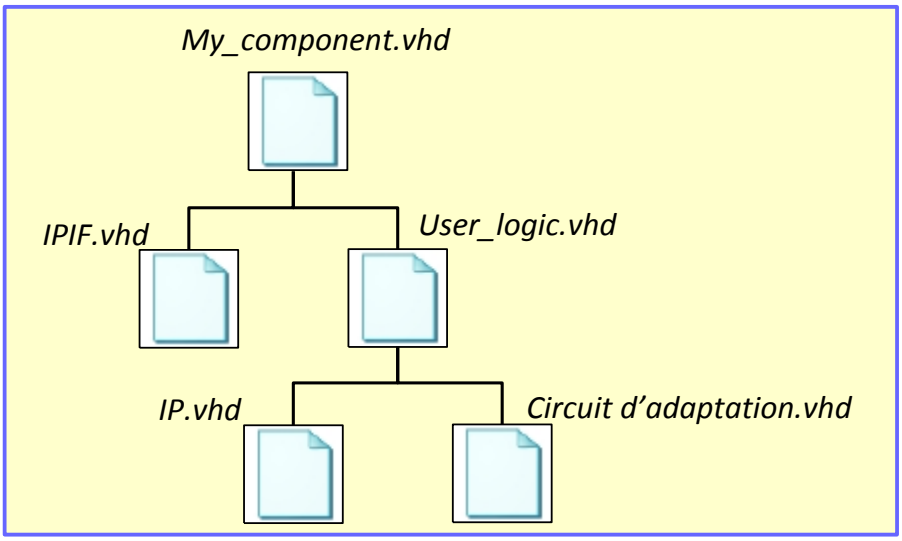
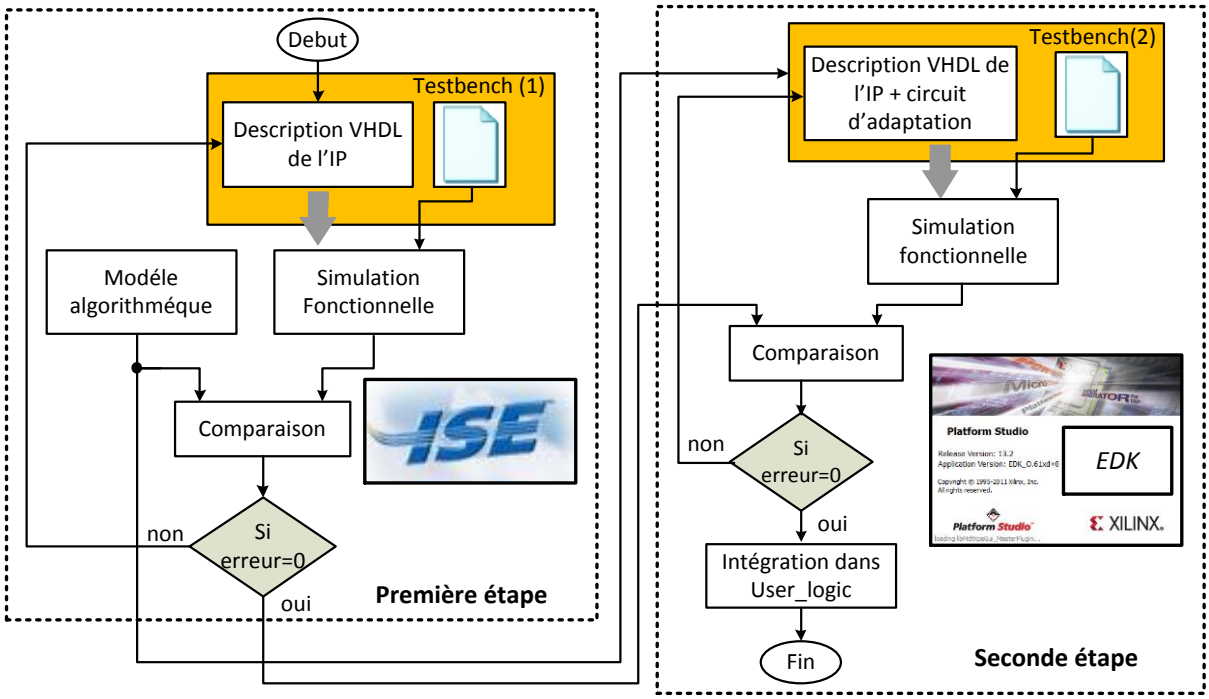


Figure 3.12. Structure hiérarchique des fichiers VHDL

Dans l'architecture du module *My_component,* la partie la plus importante et qui nécessite une durée considérable dans le cycle de conception du système embarqué, est celle qui correspond à la réalisation des composants : IP et circuit d'adaptation. En effet, avant d'intégrer la description VHDL de ces deux composants dans le fichier *user_logic.vhd,* ces derniers doivent être conçus d'une manière rigoureuse. Le flot de conception et de simulation de ces deux composants est montré sur la figure 3.13.

La conception et la vérification fonctionnelle du module *"IP-Circuit d'adaptation"*, sont effectuées en deux étapes. La première est réalisée dans ISE et elle consiste en la conception

et la vérification fonctionnelle de l'IP seule. Dans la seconde étape, le module constitué par l'IP et le circuit d'adaptation sera réalisé puis intégré dans le composant *user_logic*



(a) Simulation dans ISE (b) Simulation dans EDK

Figure 3.13. Organigramme des simulations fonctionnelles.

a. Première étape

Avant d'entamer la description VHDL de l'IP, on constitue en premier lieu son modèle algorithmique. Ce dernier peut être réalisé en utilisant à titre d'exemple Matlab ou Maple [80]. Ensuite, la description VHDL de l'IP est effectuée dans ISE. Dans cette étape, un fichier *testbench* est nécessaire pour exécuter la simulation fonctionnelle. Ce fichier n'est rien d'autre qu'une description VHDL des données d'entrée, de l'horloge, du signal d'initialisation, etc. Une fois la réalisation de ces fichiers est accomplie, une simulation fonctionnelle est effectuée en utilisant le logiciel Modelsim [81]. Finalement, une vérification du résultat est nécessaire pour confirmer le fonctionnement correct de l'IP. Celle-ci prend en considération, les résultats issus du modèle algorithmique et de la simulation fonctionnelle de l'IP.

b. Deuxième étape

Dans cette étape, il question de construire la description VHDL du module *"IP-Circuit d'adaptation"*. Le processus de cette étape est effectué dans EDK et consiste en la réalisation et l'intégration au fur et à mesure de ce module dans le composant *user_logic*. L'implémentation de ce module est basée sur des simulations fonctionnelles qu'on effectue sur l'architecture globale du composant *My_component*. Le testbench utilisé dans ces

simulations est constitué d'une procédure qui fait appel à la partie logicielle de l'IP. L'outil utilisé dans ces simulations est le logiciel modelsim. Le processus de cette étape prend fin, une fois la réalisation du module *"IP-Circuit d'adaptation"* achevée et le résultat de la simulation fonctionnelle se trouve identique au résultat du modèle algorithmique.

3.4.6. Réalisation de la partie logicielle de l'application et chargement du système sur circuit FPGA

Les points essentiels à développer dans cette étape du cycle de conception du système embarqué sont [7] :

- Réalisation du programme principal de l'application.
- Compilation du programme principal.
- Chargement du ficher de configuration (bitstream) sur la carte de prototypage à base du circuit FPGA et visualisation des résultats sur l'HyperTerminal de Windows.

Le programme principal de la partie logicielle du système embarqué, constitue une passerelle entre l'OS du processeur et les deux périphériques qui l'entourent, à savoir, l'UART et le composant *My_component*. Il est constitué principalement par les pilotes de ces deux périphériques, comme ceci est montré sur la figure 3.14.

La description du programme principal peut être effectuée en langage C ou en C++ dans l'outil SDK de Xilinx (Software Development Kit) [72]. Les pilotes qui correspondent à la réception et à la transmission des données à travers l'UART, suivant le protocole de communication RS232, sont respectivement :

- *XUartLite_RecvByte(XPAR_RS232_BASEADDR).*
- *XUartLite_SendByte(XPAR_RS232_BASEADDR, data).*

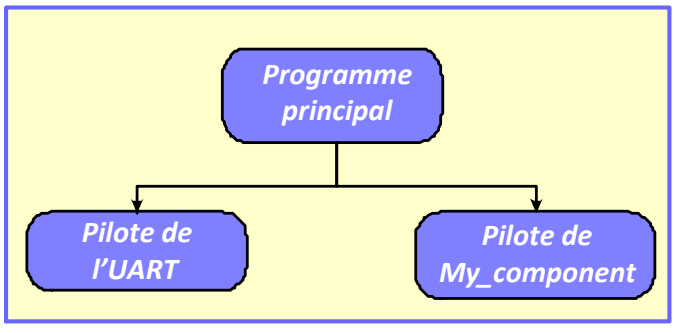


Figure 3.14. Structure du programme principal du système embarqué

La compilation du programme principal consiste à convertir son code source en langage machine du processeur. Le flot de compilation est montré sur la figure 3.15.

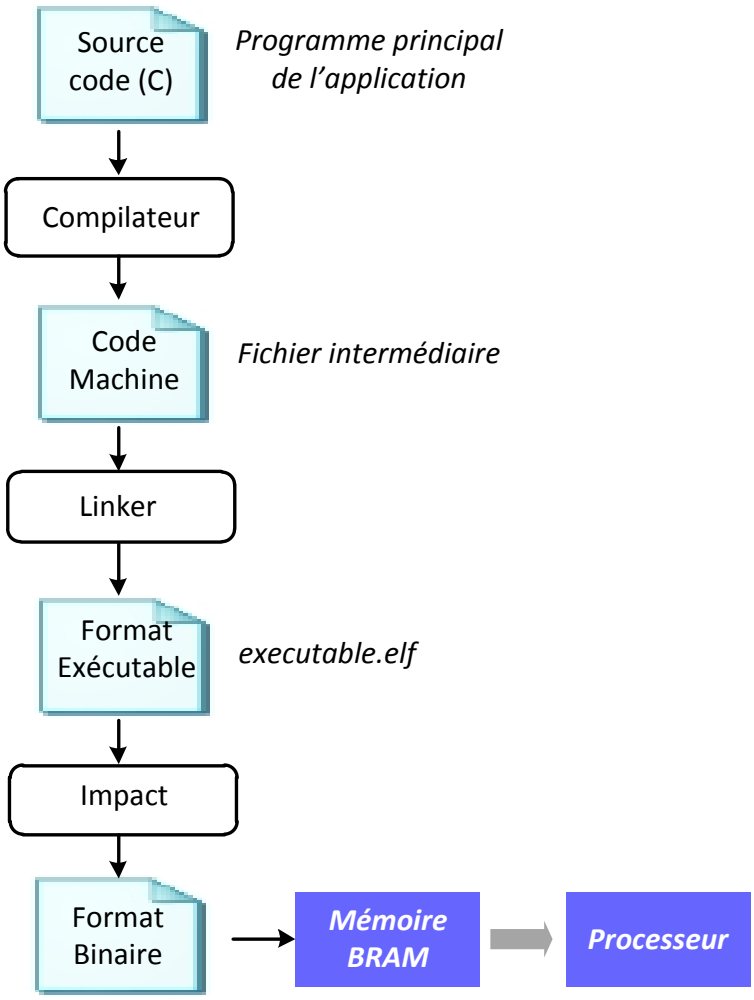


Figure 3.15. Compilation de la partie logicielle de l'application

L'outil utilisé dans cette étape est le compilateur GCC d'EDK. A travers ce processus et avant de charger le programme principal de l'application dans la mémoire BRAM, il est converti en un fichier intermédiaire qui constitue son code machine. Ce dernier n'est rien d'autre que la représentation en binaire des instructions et des données utilisées. Ce code 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 *(executable.elf)*. Finalement, l'outil ″Impact″ décode les informations portées par l'exécutable et assure son chargement dans la mémoire BRAM. A noter que cet outil prend en charge aussi la configuration du circuit FPGA, par le chargement de son bitstream. Généralement, la configuration du circuit FPGA se déroule avant l'initialisation de la mémoire par l'exécutable de l'application [7].

3.5. Conclusion

Ce chapitre a été consacré à la présentation de la tendance utilisée actuellement pour la réalisation des systèmes embarqués sur puce.

En effet, pour aboutir à une réalisation complète de ce genre de système, il nécessaire de maitriser un certain nombre de paramètres. Ces derniers relèvent principalement de :

- La technologie de la plateforme matérielle du circuit utilisé.
- L'aspect architectural de ces systèmes.
- Le cycle et les outils qui permettent leurs développements.
- La conception conjointe des deux ressources logicielle et matérielle.

Pour ce faire, nous avons introduit dans ce chapitre, des généralités sur les systèmes embarqués, leur architecture de base, ainsi que les plateformes qui peuvent être utilisées pour leur implémentation matérielle. Notre étude a été en particulier axée sur la méthodologie de conception sur des plateformes utilisant des circuits programmables de type FPGA. Le choix de s'orienté vers des implémentations et des prototypages sur ce genre de circuits, présente plus qu'un avantage. On peut citer entre autres, la panoplie d'IP Cores développés parallèlement par les fabricants de ces circuits, leur flexibilité et leur reconfigurabilité. Dans ce chapitre, nous nous sommes aussi intéressés à la relation qui peut exister entre les composants de base qui constituent un système embarqué et sa mise en œuvre sur une carte de prototypage. La complexité majeure qui relève de cette relation, réside dans la liaison à établir entre le système embarqué et les composants qui sont disponibles sur la carte utilisée.

En vue d'atteindre l'objectif principal de ce projet qui consiste en l'implémentation d'un crypto système embarqué sur circuit FPGA, l'étude que nous avons menée dans ce chapitre est jugée primordiale et incontournable. Celle-ci a été effectuée afin de se familiariser avec ce genre de systèmes et mieux comprendre les défis à relever pour leur réalisation.

Chapitre 4

Implémentation sur circuit FPGA de la partie matérielle du crypto système RSA

4.1. Introduction

L'objectif principal de ce travail consiste en la réalisation d'une plateforme de chiffrement et de déchiffrement RSA, basée sur un environnement PSoC. Pour ce faire, les deux algorithmes 7 et 8 présentés dans le chapitre 2, sont utilisés respectivement pour l'implémentation de la MMM et de l'exponentiation modulaire sur circuit FPGA. Ces deux algorithmes sont proposés dans ce travail, afin d'optimiser et d'adapter l'exécution de ces deux opérations aux ressources internes du processeur utilisé, à savoir le processeur Microblaze de Xilinx [18].

En effet, bien que l'algorithme 8 soit recommandé pour l'optimisation du temps d'exécution de l'exponentiation modulaire, en vertu de l'exécution parallèle des deux MMMs; néanmoins, lorsque l'optimisation des ressources matérielles sur le support d'implémentation est une contrainte, cet algorithme peut être implémenté en se basant sur une exécution séquentielle des deux MMMs. Dans ce cas, seulement un seul multiplieur modulaire est utilisé pour calculer en série ces deux opérations. Si cette implémentation est retenue, la condition de la ligne 7 de l'algorithme 8 peut avoir une influence considérable sur les performances d'exécution de l'exponentiation modulaire. Cette condition montre que le calcul de $C_{(i)}$ n'est effectué que si le $i^{ème}$ bit de l'exposant Z est un *"1"* logique ($z_w^j = 1$). De ce fait, la complexité en termes de nombre de MMMs qui correspond au cas le plus défavorable est de $(2 \times e)+3$; c'est-à-dire lorsque tous les bits de Z sont non zéros. Alors que pour une implémentation parallèle, le nombre en question se trouve réduit à $(e+2)$. Ainsi, dans le but d'atteindre un meilleur compromis entre le temps d'exécution, les ressources matérielles requises et la flexibilité du crypto système, nous proposons dans ce travail trois approches d'implémentations.

La première approche repose sur l'implémentation matérielle d'une seule MMM. Celle-ci est intégrée autour du processeur dans un composant matériel. Le contrôle de l'algorithme d'exponentiation modulaire est exécuté en logiciel par le processeur Microblaze. Dans cette implémentation, les deux MMMs de chaque itération *(w)* de l'algorithme 8, sont calculées séquentiellement. Les opérandes de chaque opération et le résultat obtenu, sont stockés dans la mémoire locale du processeur.

La seconde approche d'implémentation consiste en l'exploitation de l'exécution parallèle des deux MMMs de l'algorithme 8. En effet, dans le but d'accélérer le temps d'exécution de l'exponentiation modulaire, les deux opérations sont implémentées en parallèle dans un composant matériel. De plus, à l'exception de l'exposant *Z* stocké dans la mémoire BRAM du processeur, toutes les données d'entrée et les résultats intermédiaires de l'algorithme 8 sont stockés dans l'architecture interne du composant réalisé.

La troisième approche est purement logicielle. Dans ce cas, la fonction d'exponentiation modulaire et la MMM sont exécutées par le processeur Microblaze de Xilinx. Cette approche sera présentée dans le chapitre 5.

Avant de parvenir à la mise en œuvre de ces trois approches d'implémentation, nous nous sommes intéressés en premier lieu à la conception d'une unité arithmétique, dédiée à l'exécution de la MMM sur matériel. Cette unité est réalisée en tenant compte des optimisations proposées dans l'algorithme 7.

Ce chapitre est organisé en deux parties. La première est une description globale de la plateforme proposée pour l'implémentation du crypto système RSA. La seconde partie est consacrée à la présentation des architectures matérielles implémentées sur circuit FPGA. Elle est entamée par la description de l'unité arithmétique proposée pour l'implémentation de l'algorithme 7. Ensuite, nous présentons l'architecture de la partie matérielle du crypto système. Nous terminerons par la description des parties matérielles des composants réalisés, pour la mise en œuvre de l'exponentiation modulaire.

4.2. Plateforme proposée pour la réalisation du crypto système RSA

La plateforme de chiffrement et de déchiffrement RSA implémentée dans ce travail est montrée sur la figure 4.1. Celle-ci est constituée de :

Une carte de prototypage à base de circuit FPGA*:* Le rôle attribué à ce dernier est l'exécution de l'exponentiation modulaire.

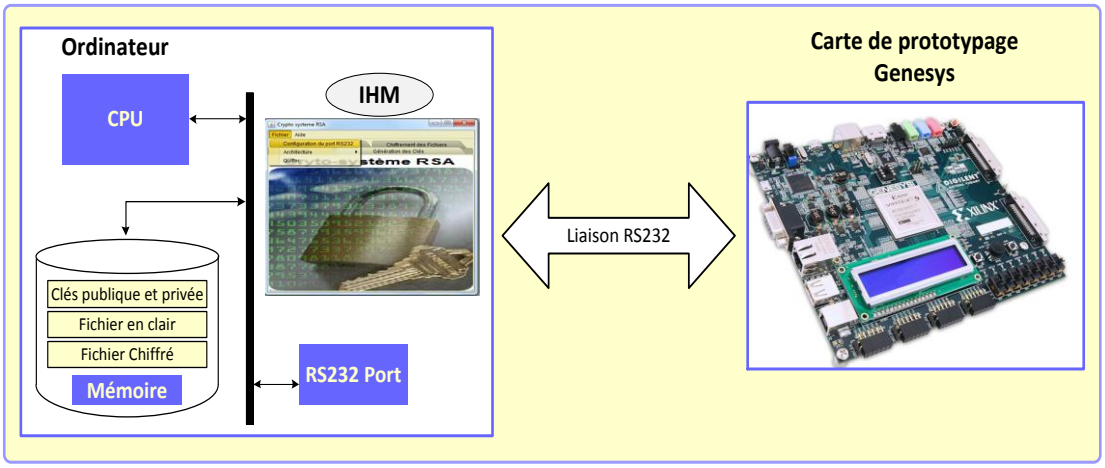


Figure 4.1. Plateforme de chiffrement/déchiffrement RSA

Un canal de communication: Ce dernier est basé sur le protocole RS232.

Une Interface Homme/Machine (IHM) : Cette interface est exécutée sur un ordinateur. Son implémentation est réalisée en langage Java.

Pour permettre une utilisation flexible de la partie du crypto système, implémentée sur circuit FPGA, l'*IHM* fournit plusieurs options, telles que:

- Configuration des paramètres du protocole RS232.
- Génération des deux clés publique et privée suivant le protocole du crypto système RSA.
- Configuration et sélection de l'approche d'implémentation utilisée pour le calcul de l'exponentiation modulaire.
- Sélection de la base β de représentation des données. Dans ce travail, nous avons utilisé les deux bases $\beta=2^{16}$ et $\beta=2^{32}$.
- Décomposition des données en chiffres de taille *8-bits*. Les données en question sont constituées de l'exposant *Z*, du modulo *N*, du message *X*, de la constante *N'* de Montgomery, de la valeur de $R^2 \bmod N$ et des tailles de l'exposant et du message.
- Transmission des données vers le circuit FPGA.
- Réception du résultat de l'exponentiation modulaire et son affichage sur écran.

- Calcul des délais d'exécutions qui correspondent à la transmission des données, à l'exécution de l'algorithme de l'exponentiation modulaire et à la réception du résultat.

La plateforme développée dans ce travail permet non seulement de chiffrer et de déchiffrer des données avec une taille arbitraire, mais aussi, de traiter des données stockées sous forme de fichiers dans la mémoire interne de l'ordinateur. Son fonctionnement est résumé comme suit :

Une fois la configuration du protocole RS232 est effectuée, on procède dans l'*IHM* par la génération des deux clés publique et privée, définies respectivement par les couples *(e, N)* et *(d, N)*. L'implémentation de cette étape est réalisée en langage Java, en se basant sur les équations (1.6) et (1.7) du chapitre 1. Ces clés sont alors stockées sous forme de fichiers *(.txt)* dans le disque dur de l'ordinateur. Ensuite, les données constituées par l'exposant *Z* qui peut être *"e"* ou *"d"*, le modulo *N*, le message *X*, la constante *N'*, la valeur de $R^2 \bmod N$ et les tailles de l'exposant et du message, seront codées en base $\beta=2^k$, puis décomposées sur des chiffres de taille *8-bits* et transmises vers la carte de prototypage Genesys.

Après réception des données, ces dernières seront restituées en premier lieu en base $\beta=2^k$. Cette opération est réalisée par le processeur MicroBlaze. Ensuite l'algorithme de l'exponentiation modulaire sera exécuté en utilisant l'une des trois approches proposées pour son implémentation. Une fois l'opération du chiffrement ou du déchiffrement achevée dans le circuit FPGA, le résultat *Y* est transféré vers l'ordinateur sur des chiffres de taille *8-bits*. Il sera stocké à la fin dans la mémoire interne de l'ordinateur.

La complexité de l'exécution du crypto système RSA sur la plateforme proposée dépend essentiellement de la partie implémentée sur le circuit FPGA et qui correspond au calcul de l'exponentiation modulaire. Cette partie peut être considérée comme étant le point clé de la plateforme.

4.3. Solutions proposées pour l'implémentation de la MMM et de l'exponentiation modulaire sur circuit FPGA

Dans le but d'implémenter la MMM et l'exponentiations modulaire sur circuit FPGA, les opérations arithmétiques de l'algorithme 7 sont implémentées sur matériel dans une Unité Arithmétique *(UA)*. Cette dernière est utilisée comme une boite noire

dans les implémentations basées sur la conception conjointe des ressources logicielle et matérielle (approches d'implémentations 1 et 2).

Dans ce qui suit, nous décrivons d'abord l'architecture interne de l'*UA*. Ensuite, nous présenterons la conception de la partie matérielle du crypto système et celles des composants, réalisés dans les deux premières approches d'implémentations.

4.3.1. Architecture interne de l'unité arithmétique

La conception d'une architecture matérielle pour l'implémentation de l'algorithme 7, relève principalement des opérations arithmétiques requises pour son exécution. Dans le chapitre 2, nous avons montré que les algorithmes des deux variantes utilisées pour calculer la MMM sont basés sur le calcul itératif des résultats intermédiaires $T_{(i+1)}$, donné par l'équation (2.15). Pour adapter l'exécution de la MMM au bus de données du processeur, l'algorithme 7 proposé dans ce travail repose sur :

- Le codage des opérandes *A, B, N* et des résultats intermédiaires $T_{(i)}$ en base $\beta=2^k$, tels que :
  $A = \sum_{i=0}^{n} A[i] \times 2^{i\times k}$, $B = \sum_{j=0}^{n} B[j] \times 2^{j\times k}$, $N = \sum_{j=0}^{n} N[j] \times 2^{j\times k}$
  $T = \sum_{j=0}^{n} T[j]_{(n+1)} \times 2^{j\times k}$.
- L'utilisation de deux variables intermédiaires $Z_{(i)}$ et $L_{(i)}$. Ces dernières ont été définies dans le chapitre 2 respectivement par les équations (2.16) et (2.17).
- L'exécution séquentielle des opérations arithmétiques sur une précision de taille *k-bits.* Les *j*^{ème} *chiffres* de $Z_{(i)}$ et $L_{(i)}$, sont obtenus *chiffre-par-chiffre*, en utilisant les équations de récurrence suivantes :

$$
\begin{cases}
C1[-1]_{(i)} = cy_1^{-1} = cy_2^{-1} = 0 \\
C2[-1]_{(i)} = cy_3^{-1} = cy_4^{-1} = 0 \\
\textit{Pour j varie de 0 à n} \\
A[i]\times B[j] = P1[j]_{(i)} + \left(C1[j]_{(i)}\times 2^k\right) & (4.1) \\
(cy_2^j 2^k, cy_1^j 2^k, Z[j]_{(i)}) = P1[j]_{(i)} + C1[j-1]_{(i)} + T[j]_{(i)} + cy_1^{j-1} + cy_2^{j-1} & (4.2) \\
q_{(i)}\times N[j] = P2[j]_{(i)} + (C2[j]_{(i)}\times 2^k) & (4.3) \\
(cy_4^j 2^k, cy_3^j 2^k, L[j]_{(i)}) = Z[j]_{(i)} + P2[j]_{(i)} + C2[j-1]_{(i)} + cy_3^{j-1} + cy_4^{j-1} & (4.4)
\end{cases}
$$

avec, $q_{(i)} = ((T[0]_{(i)} + (A[i]\times B[0]))\times N')\ mod\ 2^k$ (4.5)

En revanche, les performances de l'algorithme 7 en termes de temps d'exécution peuvent être limitées, puisque chacune de ses itérations *(i)* est exécutée en mode série. Ainsi, pour compenser le faible degré du parallélise, l'architecture proposée pour l'implémentation matérielle de l'*UA* est basée sur le traitement des opérations arithmétiques en pipeline.

a. Architecture pipeline de l'*UA*

D'une manière générale, pour exécuter une itération *(i)* d'un processus de calcul dans une architecture pipeline, celle-ci doit être décomposée en plusieurs opérations élémentaires distinctes. Une opération de base peut être une génération d'adresse, une lecture de mémoire ou encore, une opération arithmétique. La figure 4.2 est un exemple d'exécution de deux itérations successives *(i)* et *(i+1)*.

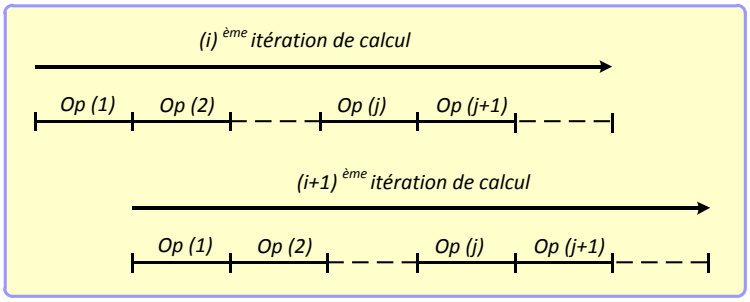


Figure 4.2. Exécution en pipeline de deux itérations de calcul

L'utilisation de cette méthode pour l'optimisation des performances temporelles des architectures sérielles, présente plusieurs avantages. On peut citer entre autres :

- L'exécution de l'itération *(i+1)* peut commencer sans attendre que l'itération *(i)* soit terminée.
- Les opérations d'indices *(j+1)* et *(j)* correspondant respectivement aux itérations *(i)* et *(i+1)* peuvent être exécutées en parallèle.
- Plus la granularité des opérations de base est faible, plus la fréquence d'exécution du pipeline est élevée.

En se basant sur ce type d'optimisation, l'architecture pipeline de l'*UA* implémentée dans ce travail, est illustrée sur la figure 4.3. La conception de cette unité repose sur l'utilisation des ressources internes disponibles sur le circuit FPGA utilisé, à savoir :

- Les blocs DSP48E [31], pour réaliser les multiplications des équations (4.1), (4.3) et (4.5).
- Les chemins de propagation des retenues [70], pour l'implémentation des additions des équations (4.2) et (4.4).

Ces ressources sont présentées en annexe A.

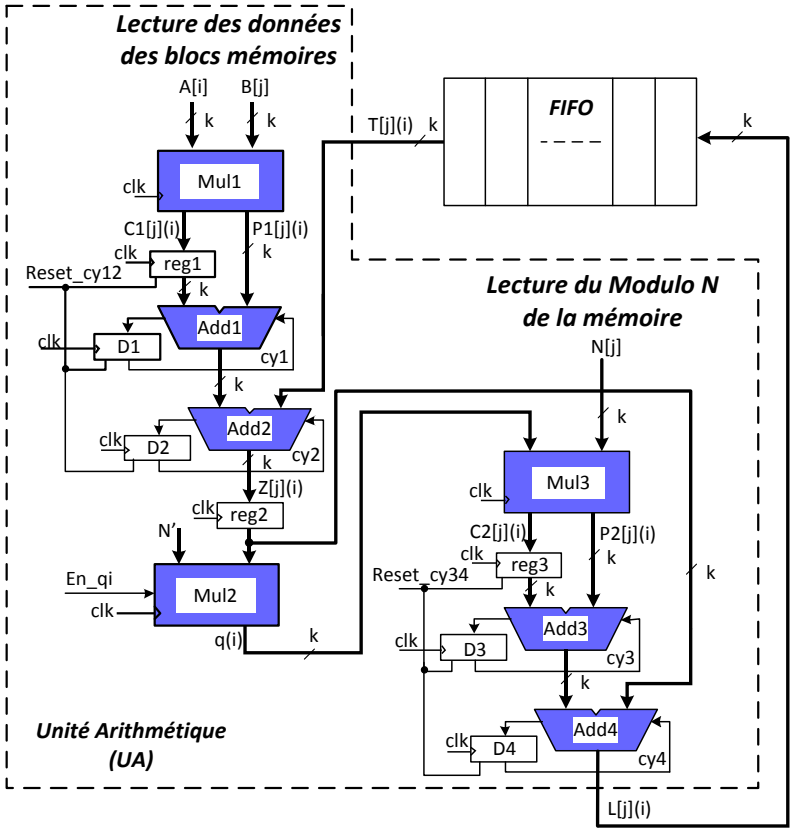


Figure 4.3. Architecture interne de l'unité arithmétique

La description VHDL de l'*UA* est décrite d'une manière générique qui dépend des paramètres *k* et *n*. Elle est composée de:

- Trois multiplieurs (*Mul1*, *Mul2* et *Mul3*).
- Quatre additionneurs à propagation de retenue (*Add1*, *Add2*, *Add3* et *Add4*).
- Trois registres et quatre bascules *D*.

L'exécution de l'algorithme 7, en utilisant l'*UA* proposée, suppose que les opérandes *A*, *B* et *N* sont stockés dans des blocs mémoires. Les résultats intermédiaires $T_{(i)}$ sont obtenus à partir d'une mémoire *FIFO*. En plus des opérandes d'entrées, l'*UA* reçoit trois signaux de contrôle, en l'occurrence, *En_qi*, *Reset_cy12* et *Reset_cy34*. Le premier permet de maintenir $q_{(i)}$ constant. Les deux autres assurent l'initialisation des retenues au début de chaque itération *(i)* de l'algorithme 7.

A chaque itération *(i)* où le $i^{ème}$ chiffre de l'opérande *A* est sélectionné, le processus débute par l'exécution des équations (4.1) et (4.2), pour calculer le $j^{ème}$ chiffre $Z[j]_{(i)}$ de $Z_{(i)}$. Ces équations sont exécutées en utilisant le multiplieur *Mul1* et les additionneurs *Add1* et *Add2*. Ensuite, le $j^{ème}$ chiffre $L[j]_{(i)}$ de $L_{(i)}$ calculé par les équations (4.3) et (4.4), est obtenu par l'utilisation du multiplieur *Mul3* et les additionneurs *Add3* et *Add4.*

La valeur de $q_{(i)}$ donnée par l'équation (4.5) est obtenue par le multiplieur *Mul2* au début de chaque itération *(i),* dès que le chiffre le moins significatif $Z[0]_{(i)}$ de $Z_{(i)}$ soit calculé. Ensuite, $q_{(i)}$ sera maintenu constant durant toute la durée d'exécution de l'itération *(i).*

Les retenues cy_1^j, cy_2^j, cy_3^j, cy_4^j et les chiffres les plus significatifs $C1[j]_{(i)}$, $C2[j]_{(i)}$, issus des équations (4.1) et (4.3), sont :

- initialisés à zéro au début de chaque itération *(i),*
- retardés d'un top d'horloge en utilisant les bascules *D1*, *D2*, *D3*, *D4* et les registres *reg1*, *reg3*, pour qu'ils soient additionnés avec les chiffres d'indice *(j+1).*

L'exécution de l'*UA* est identique pour chaque itération *(i)* de l'algorithme 7 où les $(j)^{éme}$ chiffres de $L_{(i)}$ sont calculés dans un pipeline avec une profondeur de *six étages.*

Dans [66], nous avons réalisé un crypto système RSA. L'architecture pipeline de l'*UA* implémentée, a été conçue avec une profondeur de *quatre étages*. Bien que les résultats obtenus aient montré que l'implémentation proposée présente un bon compromis entre le temps d'exécution et les ressources matérielles occupées, néanmoins, le rapport fourni par l'analyseur temporel a révélé une fréquence relativement faible qui est de *63 Mhz*. De ce fait, l'amélioration que nous proposons dans ce présent travail, consiste à rajouter deux étages au pipeline de l'*UA*, afin d'augmenter la fréquence du système [63], [64].

Pour calculer le $j^{ème}$ chiffre $L[j]_{(i)}$ de $L_{(i)}$ en considérant $A[i]$ et $q_{(i)}$ constants, les étapes sont définies comme suit :

1. Génération de l'adresse du $(j)^{éme}$ chiffre de l'opérande *B* (*B[j]*).
2. Lecture de *B[j]* et calcul du produit $A[i] \times B[j]$ par le multiplieur *Mul1*.
3. Calcul du $j^{ème}$ chiffre $Z[j]_{(i)}$ de $Z_{(i)}$ en utilisant les additionneurs *Add1* et *Add2*.

4. Génération de l'adresse du $(j)^{éme}$ chiffre $N[j]$ du modulo N.
5. Lecture de $N[j]$ et calcul du produit $q_{(i)} \times N[j]$ en utilisant le multiplieur *Mul3*
6. Calcul du $j^{ème}$ chiffre $L[j]_{(i)}$ de $L_{(i)}$ par l'utilisation des additionneurs *Add3* et *Add4*.

Le processeur de calcul de deux chiffres successifs $L[j]_{(i)}$ et $L[j+1]_{(i)}$ est montré sur la figure 4.4.

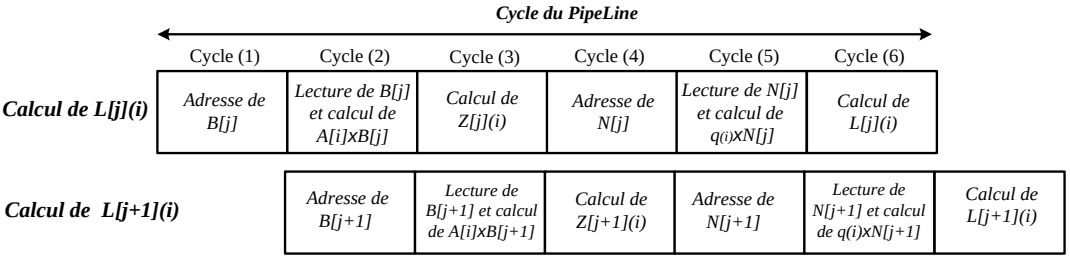


Figure 4.4. Principe de calcul des opérations arithmétiques en pipeline

b. Evaluation de la complexité d'exécution de l'*UA*

La complexité globale de l'*UA* en termes du nombre de tops d'horloge dépend principalement :

- du codage de l'opérande A en base $\beta=2^k$ qui définit le nombre d'itération (i) nécessaires pour l'exécution de l'algorithme 7,
- du nombre de tops d'horloges requis pour calculer un seul chiffre de la variable $L_{(i)}$.

Dans le chapitre 2, paragraphe 2.6 (b), nous avons montré que l'exécution d'une seule itération (i) de l'algorithme 7, nécessite $(n+1)$ *cycles* pour calculer le $j^{ème}$ chiffre de $L_{(i)}$. Si on considère le retard du pipeline de l'*UA* qui est de *six tops d'horloge,* la variable $L_{(i)}$ sera entièrement obtenue après $(n+7)$ *tops d'horloge*.

De plus, puisque le résultat intermédiaire $T_{(i+1)}$ est obtenu de la sortie de la *FIFO chiffre-par-chiffre* à l'itération $(i+1)$; de ce fait, le résultat final de la MMM, en l'occurrence $T_{(n+1)}$ qui est calculé à l'itération $i=n$, ne sera valable en sortie qu'après une itération supplémentaire, c'est-à-dire pour $i= n+1$. Ce qui signifie que le nombre d'itérations (i) nécessaire pour exécuter une seule multiplication de Montgomery est de $(n+2)$ *itérations*. Par conséquent, le nombre de tops d'horloge globale *(NbTH1)*, nécessaire pour avoir le résultat final $T_{(n+1)}$, peut être évalué par l'expression (4.6)

$$NbTH1 = (n+2) \times (n+7) \tag{4.6}$$

4.3.2. Architecture de la partie matérielle du crypto système RSA réalisé

Les trois approches proposées dans ce travail pour l'implémentation de l'exponentiation modulaire sur circuit FPGA, sont réalisées en utilisant une conception conjointe des deux parties, matérielle et logicielle. La première partie représente l'architecture de sa partie matérielle, basée sur le processeur Microblaze de Xilinx. La seconde représente ses pilotes logiciels. La conception de ces pilotes sera présentée dans le chapitre 5.

L'architecture de la partie matérielle du crypto système réalisé, est montrée sur la figure 4.5.

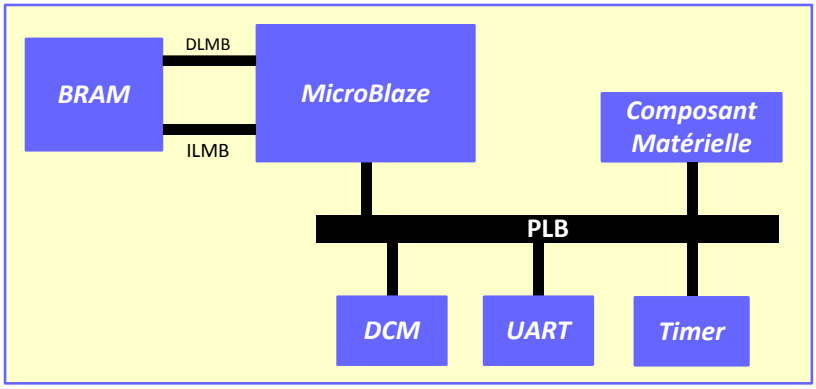


Figure 4.5. Architecture de la partie matérielle du crypto système RSA

La conception de cette architecture a été réalisée en utilisant l'environnement XPS 13.2 de Xilinx [72]. Le système embarqué réalisé est composé de:

- Un processeur Microblaze.
- Une DCM (Digital Clock Manager).
- Une mémoire local BRAM (BlockRAM).
- Un bus de données DLMB (Data Local Memory Bus)
- Un bus d'instructions ILMB (Instruction Local Memory Bus).
- Un bus système PLB (Processor Local Bus).
- Un UART (Universal Asynchronous Receiver Transmitter).
- Un Timer.
- Un composant matériel.

Tous les périphériques implémentés autour du processeur communiquent avec ce dernier à travers le bus PLB qui est de *32-bits*. Le composant matériel permet de

calculer les multiplications de Montgomery, nécessaires à l'exécution de l'algorithme d'exponentiation modulaire.

Dans cette architecture, l'*UART* est intégré afin de communiquer avec le port RS232 de la carte. La taille de la mémoire *BRAM* et le débit de l'*UART* ont été configurés respectivement à *16-kbits* et *115200 bps*. Le rôle attribué au *Timer* est de quantifier le nombre de tops d'horloge nécessaires pour :

- La réception des données transmises par l'*IHM*.
- L'exécution de la multiplication modulaire de Montgomery et de l'exponentiation modulaire.
- Le transfert du résultat.

Afin de réaliser le calcul de l'exponentiation modulaire $Y= X^Z \ mod \ N$, le processeur Microblaze reçoit à travers l'*UART*, l'exposant *Z*, le message *X*, leurs tailles, le modulo *N*, la constante *N'* et la valeur de $R^2 \ mod \ N$. Avant d'entamer l'exécution des algorithmes 7 et 8, le processeur procède par la restitution de toutes les données d'entrée suivant leur codage en base 2^k, ensuite les deux algorithmes seront exécutés. Une fois avoir achevé toutes les opérations arithmétiques, Microblaze commence par la décomposition du résultat *Y* sur des chiffres de taille *8-bits*, avant sa transmission vers l'ordinateur à travers l'*UART*.

L'élément le plus important dans l'architecture du système embarqué, est le composant matériel, conçue pour accélérer l'exécution de l'exponentiation modulaire. Son architecture interne est différente pour les deux approches proposées, où les implémentations sont basées sur la combinaison des ressources logicielle et matérielle.

4.3.3. Première approche d'implémentation : *Réalisation de l'exponentiation modulaire avec une seule MMM dans un composant matériel*

Dans le but d'accélérer l'exécution de l'exponentiation modulaire, le partitionnement logiciel/matériel proposé dans cette première approche est basé sur l'implémentation de manière logicielle du contrôle de l'algorithme 8 [65], [82]. Ce contrôle est assuré en entier par le processeur Microbalze. Le calcul des MMMs de chaque itération *(w)* de l'algorithme 8, sont exécutées par l'unité arithmétique présentée dans le paragraphe 4.3.1. Celle-ci est intégrée comme une boite noire autour du processeur, dans un composant nommé *MMM_Core*.

Ainsi, lors de l'exécution de l'exponentiation modulaire, pour calculer une seule MMM, *MMM_Core* reçoit deux opérandes *A* et *B* à partir de la mémoire locale du processeur (mémoire BRAM). Suivant le processus d'exécution de l'algorithme 8, les deux opérandes peuvent être le message *X*, *"1"*, la valeur de R^2 *mod N* où les deux résultats intermédiaires $S_{(i)}$ et $C_{(i)}$. Le résultat *T* obtenu par chaque MMM est transféré à la mémoire BRAM, pour qu'il soit utilisé comme opérande d'entrée lors de l'exécution de la prochaine opération.

Le rôle attribué au processeur Microblaze dans cette approche est résumé dans les points suivants :

1. Décalage à droite du $j^{ème}$ chiffre de l'exposant Z.
2. Tester le $w^{ème}$ *bit* z_w^j de *Z[j]* selon la ligne 7 de l'algorithme 8.
3. Activer et désactiver le fonctionnement du composant *MMM_Core*.
4. Transmission des données vers *MMM_Core*.
5. Transfert du résultat *T* issu de chaque MMM vers la mémoire BRAM.

L'intégration de *MMM_Core* dans la partie matérielle du crypto système embarqué nécessite non seulement la conception de sa partie matérielle, mais aussi, la réalisation de ses pilotes qui constituent sa partie logicielle. Dans ce qui suit, on se limitera à la présentation de son architecture matérielle. La partie logicielle sera présentée dans le chapitre 5.

4.3.3.1. Partie matérielle du composant *MMM_Core*

L'architecture de la partie matérielle du composant *MMM_Core*, basée sur un seul multiplieur de Montgomery est montrée sur la figure 4.6. Celle-ci est constituée de l'interface *IPIF* et d'un module, nommé *MulMong*.

L'interface *IPIF* a pour rôle de décoder le protocole de communication du bus système *PLB* [74]. *MulMong* assure l'exécution séquentielle des deux MMMs de chaque itération *(w)* de l'algorithme 8. Il est constitué de l'*UA*, d'une mémoire FIFO, d'un circuit de contrôle *MMM_Ctr* et de trois mémoires *RAMs* pour stocker deux opérandes et le modulo *N*. La taille de chacune de ces mémoires est de *(n+1)×k-bits*. Les composants internes configurés dans l'interface l'*IPIF* sont : deux mémoires FIFOs : *W_FIFO* et *R_FIFO* et un registre d'instruction *Inst_Registre*. *W_FIFO* est utilisée pour transférer les données vers l'architecture interne du module *MulMong*. *R_FIFO* assure la transmission du résultat *T* de la MMM vers la mémoire BRAM. Le

processeur assure le contrôle du composant en utilisant des instructions, transmises à travers le registre *Inst_Registre*. Les détails de ce registre sont montrés sur la figure 4.7.

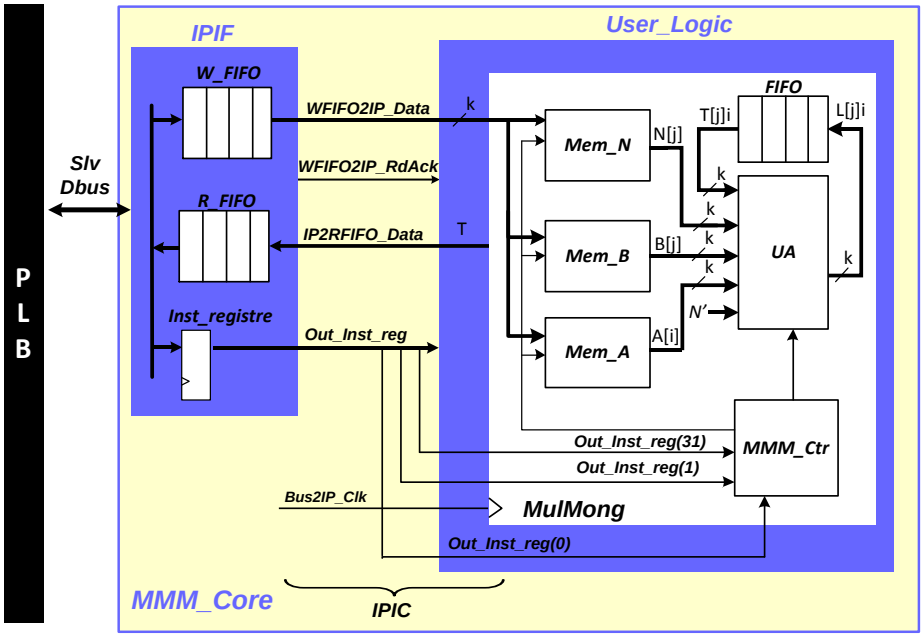


Figure 4.6. Architecture interne du composant *MMM_Core*

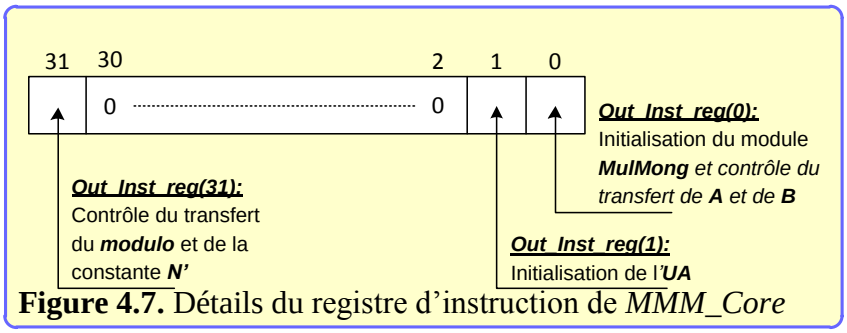


Figure 4.7. Détails du registre d'instruction de *MMM_Core*

Les codes binaires des instructions utilisées pour contrôler la partie matérielle de *MMM_Core* sont présentés dans le tableau 4.1.

Tableau 4.1. Format des instructions du composant *MMM_Core*

Instruction	***Code Binaire***	***Description***
ResetIPMMM	0000…..0001	Initialisation du module *MulMong*.
RunWriteModNc	1000…..0001	Transfert du modulo et de la constante *N'*.
RunWriteAB	0000…..0000	Transfert des deux opérandes *A* et *B*.
RunMMM	0000…..0010	Execution de la MMM

Le transfert de données et d'instructions entre le processeur Microblaze et le module *MulMong* est effectué à travers le système d'interconnexion *IPIC* [74], [75]. Ce dernier est constitué par un ensemble de signaux, générés par l'interface *IPIF* après le décodage du protocole du bus système *PLB*.

Les signaux de l'*IPIC* utilisés dans cette approche d'implémentation sont définis comme suit :

- ***Bus2IP_Clk*** : Signal d'horloge, transmis par le bus système *PLB*.
- ***WFIFO2IP_Data*** : Bus de données de sortie de la mémoire *W_FIFO*. Ce bus est utilisé pour transmettre les données vers le module *MulMong*.
- ***WFIFO2IP_RdAck*** : Ce signal permet d'indiquer par une impulsion active à l'état haut que la donnée transmise par le processeur, sera valide après le prochain top d'horloge.
- ***IP2RFIFO_Data*** : Bus de données d'entrée de la mémoire *R_FIFO*. Il assure le transfert du résultat *T* de la MMM vers le processeur.
- ***Out_Inst_reg(0)*** : Ce signal représente le bit le moins significatif du registre d'instruction. Il permet d'une part d'initialiser le module *MulMong* et d'autre part, d'activer le chargement des opérandes *A* et *B*. Les états de ce signal pour effectuer ces deux opérations sont respectivement *Out_Inst_reg(0)=1* et *Out_Inst_reg(0)=0*.
- ***Out_Inst_reg(1)*** : Ce signal correspond au second bit du registre d'instruction. Il est utilisé pour initialiser à zéro l'*UA*, une fois avoir achevé l'exécution des opérations arithmétiques de l'algorithme 7. Les états de ce signal pour réaliser l'initialisation en question et pour valider le fonctionnement de *l'UA*, sont respectivement *Out_Inst_reg(1)=0* et *Out_Inst_reg(1)=1*.
- ***Out_Inst_reg(31)*** : Ce signal représente le bit le plus significatif du registre d'instruction. Il permet de valider par un *"1"* logique le chargement du modulo *N* et de la constante *N'*, indépendamment des deux opérandes.

4.3.3.2. Architecture matérielle du module *MulMong*

L'architecture interne du module *MulMong* est montrée sur la figure 4.8. Celle-ci est composée de :

- trois mémoires *RAMs*, nommées : *Mem_N*, *Mem_A* et *Mem_B*,
- une unité arithmétique,

- une mémoire *FIFO* pour stocker les résultats intermédiaires $T_{(i)}$ de l'algorithme 7,
- deux registres *N'_Reg* et *T_Reg* assurant respectivement le stockage de la constante *N'* et la synchronisation du transfert du résultat *T* vers le processeur Micoblaze.
- trois multiplexeurs *Mux_1*, *Mux_2* et *Mux_3*.
- un circuit de contrôle *MMM_Ctr*.

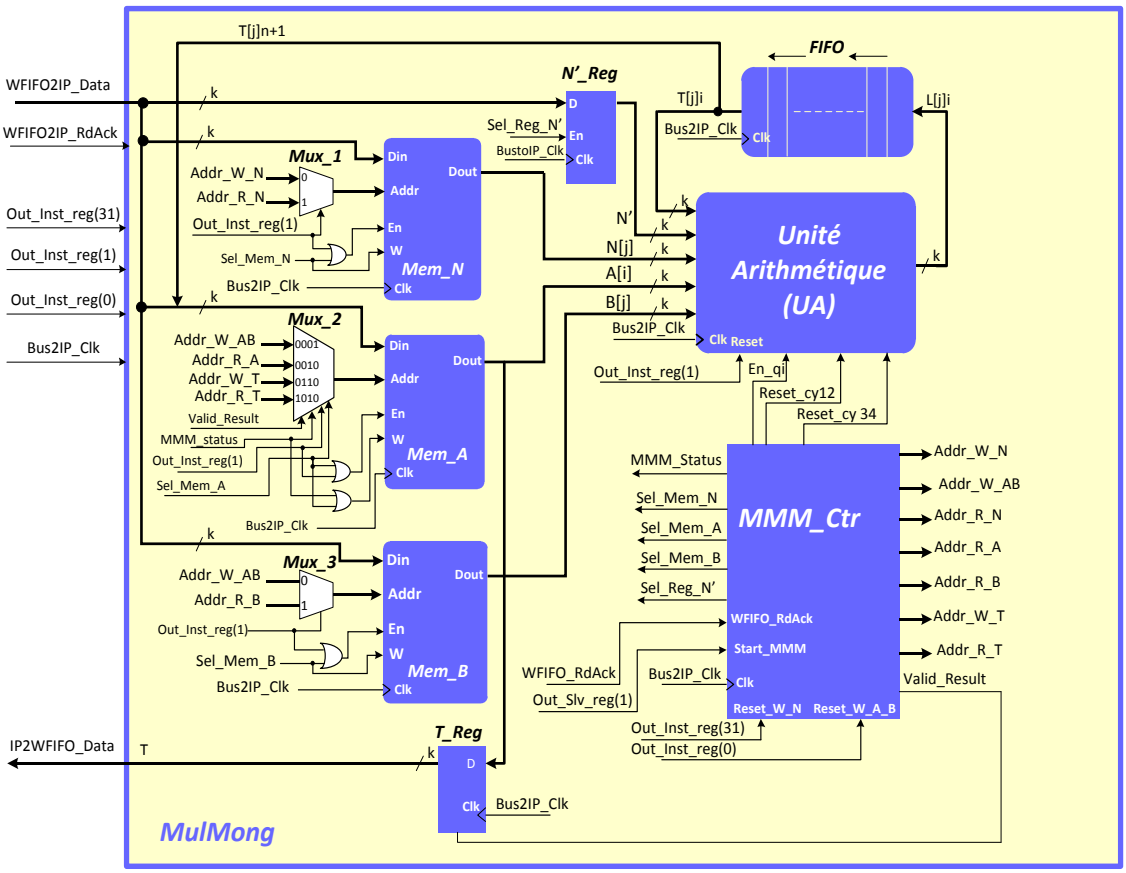


Figure 4.8. Architecture interne du module *MulMong*

Le rôle attribué au circuit de contrôle est de générer :

- les adresses mémoires qui correspondent aux phases de lecture et d'écriture, ainsi que leurs signaux de commandes,
- les signaux de sélection des bus d'adresses,
- les signaux de contrôle de l'*UA* et des deux registres *N'_Reg* et *T_Reg*.

Le calcul de l'exponentiation modulaire est entamé par le chargement du modulo *N* et de la constante *N'*. Ces dernières restent stockés durant tout le processus d'exécution de l'algorithme 8, respectivement dans la mémoire *Mem_N* et dans le

registre *N'_Reg*. Les données stockées dans les mémoires *Mem_A* et *Mem _B* sont renouvelées à chaque itération *(w)* de algorithme 8 par deux opérandes *A* et *B*.

La sélection des adresses générées par le circuit de contrôle sont sélectionnées aux entrées des bus d'adresses des trois mémoires, en utilisant les multiplexeurs *Mux_1*, *Mux_2* et *Mux_3*. Les tables de vérité de ces multiplexeurs et les configurations des mémoires utilisées, sont détaillées dans l'annexe B.

4.3.3.3. Etapes d'exécution d'une MMM par le module *MulMong*

L'exécution d'une MMM utilisant le module *MulMong* est effectuée en trois étapes. La première consiste en la réception des données et leurs stockages en mémoire. La seconde correspond à l'exécution des opérations arithmétiques de l'algorithme 7. La troisième est associée au transfert du résultat *T* de la MMM.

1. Réception des données.

Le contrôle du chargement des données d'entrée dans les unités de stockage qui leur sont associées, est basé sur l'impulsion fournie par le processeur sur le signal *WFIFO2IP_RdAck*. Ce dernier est utilisé par le circuit de contrôle *MMM_Ctr* pour :

- Générer les adresses mémoires lors de leur configuration en mode d'écriture.
- Contrôler la sélection des mémoires *Mem_A*, *Mem_B*, *Mem_N* et du registre *N'_Reg*, en utilisant respectivement les signaux de sélection *Sel_Mem_A*, *Sel_Mem_B*, *Sel_Mem_N* et *Sel_Reg_N'*.

La sélection de chaque unité de stockage est effectuée séquentiellement, selon l'ordre des données transmises par le processeur Microblaze. Le contrôle de cette opération repose sur les quatre signaux de sélection, cités précédemment. Chacun de ces derniers est forcé à *"1"*, lors de la réception de la donnée qui lui associée.

L'adresse initiale utilisée pour le stockage des chiffres les moins significatifs de chaque opérande est l'adresse *0x01*. Celle-ci s'incrémente au front montant du signal *WFIFO2IP_RdAck.*

Avant de commencer le chargement des deux opérandes *A* et *B*, *MulMong* procède par la réception du Modulo *N* et de la constante *N'*. Le début de cette opération est entamé dès que le signal *Out_Inst_reg(31)* est mis à *"1"*. Ce dernier est contrôlé par Microblaze par l'intermédiaire de l'instruction *RunWriteModNc*. Ainsi, le circuit de contrôle *MMM_Ctr* débute par l'initialisation des adresses transmises à

travers le bus d'adresse *Addr_W_N* à la valeur initiale *0x01*. Puis, celles-ci s'incrémenteront au fur et mesure que les $j^{ème}$ *chiffres* de *N* seront réceptionnés.

A la fin du chargement de *N* et de *N'*, Microblaze transmet l'instruction *RunWriteAB*. Celle-ci force le signal *Out_Inst_reg(0)* à *″0″*. Ce qui valide la réception des deux opérandes *A* et *B*. Les adresses mémoires qui correspondent au stockage de ces opérandes sont transmises par *MMM_Ctr* à travers le bus *Addr_W_AB*.

Une fois les $j^{ème}$ chiffres des deux opérandes *A* et *B* complémentent stockés, Microblaze transmet l'instruction *RunMMM* qui force le signal *Out_Inst_reg(1)* à *″1″*. Ainsi, le module *MulMong* entame l'exécution des opérations arithmétiques de l'algorithme 7.

2. Exécution des opérations arithmétiques de l'algorithme 7.

A chaque cycle de calcul de la MMM, les mémoires *Mem_A*, *Mem_B* et *Mem_N* sont configurées en lectures. Leur adressage est assuré respectivement par les bus *Addr_R_A*, *Addr_R_B* et *Addr_R_N*. Les opérations arithmétiques de l'algorithme 7 sont exécutées dans l'*UA*. Celle-ci reçoit en entrées *chiffre-par-chiffre*, le modulo *N*, les deux opérandes *A* et *B*, les résultats intermédiaires $T_{(i)}$ et la constante *N'*. Le contrôle de cette unité est assuré par les trois signaux *En_qi*, *Reset_cy12* et *Reset_cy34*, générés par le circuit de contrôle. Le premier permet de maintenir $q_{(i)}$ constant durant l'exécution d'une itération *(i)* de l'algorithme 7. Les deux autres assurent respectivement initialisation de (cy^j_1, cy^j_2, *C1[j]*) et de (cy^j_3, cy^j_4, *C2[j]*). Les $j^{ème}$ chiffres $L[j]_{(i)}$ de la variable intermédiaire $L_{(i)}$ données par l'équation (4.4), sont stockés *chiffre-par-chiffre* à chaque top d'horloge dans la mémoire *FIFO* du module *MulMong* (voir figure 4.8). Celle-ci fournit en sortie, les $j^{ème}$ chiffres des résultats intermédiaires $T_{(i+1)}$ de l'algorithme 7.

La synchronisation des trois signaux de contrôle *En_qi*, *Reset_cy12* et *Reset_cy34* est effectuée suivant l'ordre d'exécution des opérations arithmétiques. *En_qi* étant initialisé à *"0"*, bascule à *"1"* dès que le chiffre le moins significatif $Z[0]_{(i)}$ de la variable $Z_{(i)}$ soit calculé. Pour maintenir la valeur de $q_{(i)}$ constante durant toute la durée d'exécution de l'itération *(i)*, le signal *En_qi* est remis à *"0"* au prochain top d'horloge. Puis il sera maintenu dans cet état, jusqu'à ce que la variable $L_{(i)}$ soit entièrement calculée.

Les états des signaux utilisés pour l'initialisation des retenues, en l'occurrence *Reset_cy12* et *Reset_cy34*, dépendent de l'ordre de calcul de ces retenues. En effet,

ces signaux initialisés à *"0"* au départ, basculent à *"1"* dès que les premières retenues (cy^0_1, cy^0_2, *C1[0]*) et (cy^0_3, cy^0_4, *C2[0]*) issues de l'itération *(j=0)* seront obtenues. Comme le calcul $L[j]_{(i)}$ est retardé de *3 tops* d'horloge par rapport au calcul de $Z[j]_{(i)}$, le basculement à *"1"* des deux signaux se trouve décalé avec le même délai.

Le résultat *T* de la MMM obtenu à l'itération *i=n* est stocké à l'itération *i=n+1* dans la mémoire *Mem_A*, avant qu'il ne soit transmis au processeur Microblaze.

3. Transfert du résultat T vers le processeur Microblaze.

Le transfert du résultat *T* de la MMM vers le processeur est effectué à travers le registre *T_Reg*. En effet, puisque le débit de la liaison assurant la communication entre le processeur Microblaze et le composant *MMM_Core* est défini par l'impulsion du signal *WFIFO2IP_RdAck*, nous avons utilisé cette impulsion pour faire incrémenter l'adressage de la Mémoire *Mem_A* durant le transfert de *T*. La sélection du bus d'adresse *Addr_R_T*, généré par le circuit de contrôle *MMM_Ctr*, est assurée par le multiplexeur *Mux_2*. Pour distinguer le déroulement de cette étape des deux étapes précédentes, *MMM_Ctr* fournit le signal *Valid_Result* qui est à *"1"* seulement lors du transfert du résultat *T*. L'utilisation de ce signal permet au module *MulMong* d'indiquer son état au processeur Microblaze. En effet, lors du transfert des données vers *MMM_Core* et durant l'exécution de la MMM, *Valid_Result* est forcé à *"0"*. Ce dernier maintien le registre *T_Reg* dans un état initial, défini par *0xFFFFFFFF*.
Lors du transfert du résultat *T*, *Valid_Result* bascule à l'état *"1"*. Ensuite, le module *MulMong* procède par initialisation du registre *Reg_T* par une donnée dont tous les bits sont nuls (*0x00000000*), afin d'indiquer au processeur Microblaze le début du transfert de *T*. Lorsque Microblaze reçoit cette requête, il ne prend en considération que les *(n+1) chiffres* qui succéderont au premier zéro.

Une fois le transfert de *T* achevé, *MMM_Core* est remis à son état initial par l'intermédiaire du signal *Out_Inst_reg(0)*. Ce dernier est forcé à *"1"* dès que le processeur Microblaze transmet l'instruction *ResetIPMMM*.

Les résultats de simulation qui correspondent au transfert de *T* au processeur Microblaze, sont présentés dans le chapitre 6, sur la figure 6.4.

4.3.4. Deuxième approche d'implémentation : *Réalisation de l'exponentiation modulaire avec deux MMMs dans un composant matériel*

L'objectif principal de cette deuxième approche consiste en l'optimisation du temps d'exécution de l'exponentiation modulaire, par rapport à la première approche.

Le partitionnement logiciel/matériel proposé est basé sur l'implémentation parallèle des deux MMMs de l'algorithme 8. Ces opérations sont exécutées par deux unités arithmétiques *(UAs),* intégrées autour du processeur dans un composant matériel nommée *2MMMs_Core* [83], [84]. De plus, pour réduire la quantité des données transférées entre ce composant et le processeur, toutes les données mises en exécution dans l'algorithme 8 sont stockées dans des mémoires locales près des deux *UAs* ; à l'exception de l'exposant *Z,* stocké dans la mémoire BRAM du processeur.

Dans le but de contrôler de manière logicielle l'exécution de l'algorithme 8, les tâches attribuées au processeur Microblaze sont résumées comme suit :

1. Activer et désactiver le fonctionnement du composant *2MMMs_Core*.
2. Initialisation des deux *UAs* à la fin d'exécution de chaque itération *(w)* de l'algorithme 8.
3. Décalage à droite du $j^{ème}$ chiffre de l'exposant *Z* et transfert du $w^{ème}$ bit z_w^j de *Z[j]* vers l'architecture interne du composant *2MMMs_Core*. Le test de la ligne 7 de l'algorithme 8 porté sur z_w^j est effectué dans le composant *2MMMs_Core*.

L'exécution de l'exponentiation modulaire $Y=X^Z \ mod \ N$ en se basant sur ce partitionnement, nécessite la réalisation des deux parties à savoir, l'architecture matérielle du composant *2MMMcore* et sa partie logicielle. Dans ce qui suit, nous allons présenter la partie matérielle de ce composant. La partie logicielle sera présentée dans le chapitre 5.

4.3.4.1. Partie matérielle du composant *2MMMs_Core*

L'architecture globale du composant *2MMMs_Core* est montrée sur la figure 4.9. Celle-ci est constituée de l'interface *IPIF*, d'un circuit d'initialisation et d'un module, nommé *2MulMong.* L'interface *IPIF* permet de décoder le protocole de communication du bus système *PLB*. Le circuit d'initialisation est utilisé pour remettre à l'état initial les deux *UAs* à la fin d'exécution de chaque itération *(w)* de l'algorithme 8. Le module *2MulMong* assure l'exécution parallèle des deux MMMs de chaque itération *(w)* de l'algorithme 8.

Pour permettre la communication entre le processeur Microblaze et le module *2MulMong*, les composants internes configurés dans l'interface *IPIF* sont définis comme suit:

- Deux mémoires FIFOs *W_FIFO* et *R_FIFO:* La première est utilisée pour transférer les données d'entrée vers l'architecture interne du module *2MulMong*. La seconde assure la transmission du résultat *Y* vers le processeur.
- Deux registres d'états *St_Registre1* et *St_Registre2* : *St_Registre1* permet au processeur de vérifier si les données d'entrée sont entièrement transmises vers le module *2MulMong*. Les états de ce registre durant le chargement des données et lors de l'exécution de la fonction l'exponentiation modulaire sont respectivement *0xFFFFFFFF* et *0x00000000*. *St_Registre2* est utilisé par le module *2MulMong* pour indiquer son état durant le calcul parallèle des deux MMMs. En effet, lors du chargement des données et pendant l'exécution des deux opérations, l'état de ce registre est *0x00000000*. Une fois les opérations arithmétique de l'algorithme 7 (algorithme de la MMM) accomplies, *2MulMong* transmet le code *0x0000FFFF* à travers ce registre au processeur microblaze ; pour indiquer la fin d'exécution d'une itération *(w)* de l'algorithme algorithme 8.
- Un registre d'instruction *Inst_Registre* : Ce registre permet au processeur de contrôler de manière logicielle le fonctionnement du module *2MulMong*, durant le processus de calcul de l'exponentiation modulaire.

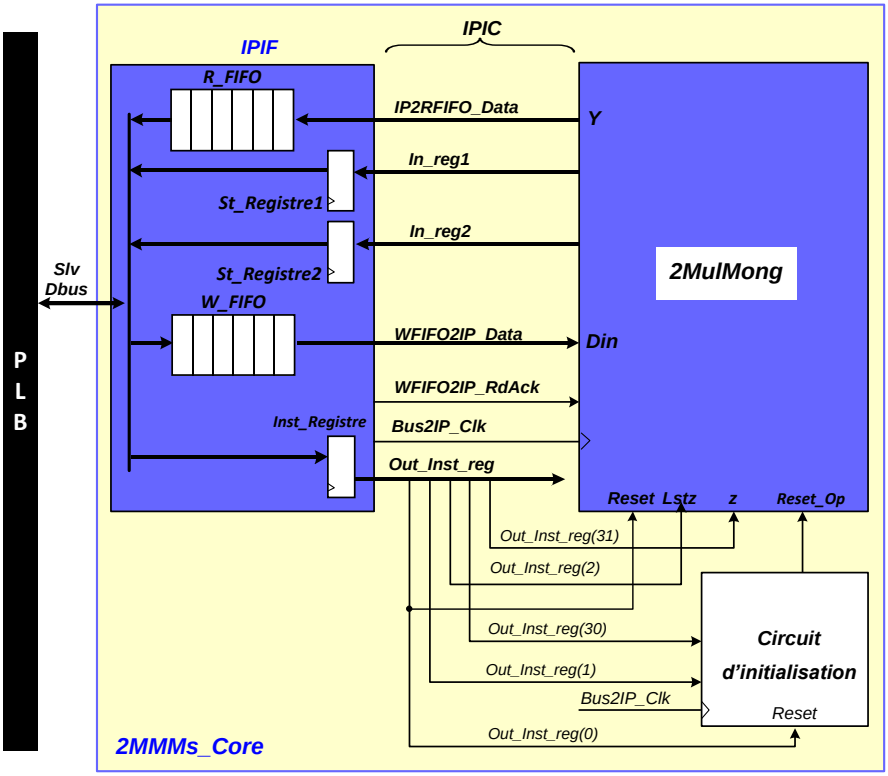


Figure 4.9. Architecture interne du composant *2MMMs_Core*

Pour contrôler les étapes d'exécution de l'algorithme 8, le processeur MicroBlaze utilise un certain nombre d'instructions. Ces dernières sont définies dans la partie logicielle du composant *2MMMs_Core* et seront transmises à travers le registre d'instruction. Les détails de ce registre sont montrés sur la figure 4.10.

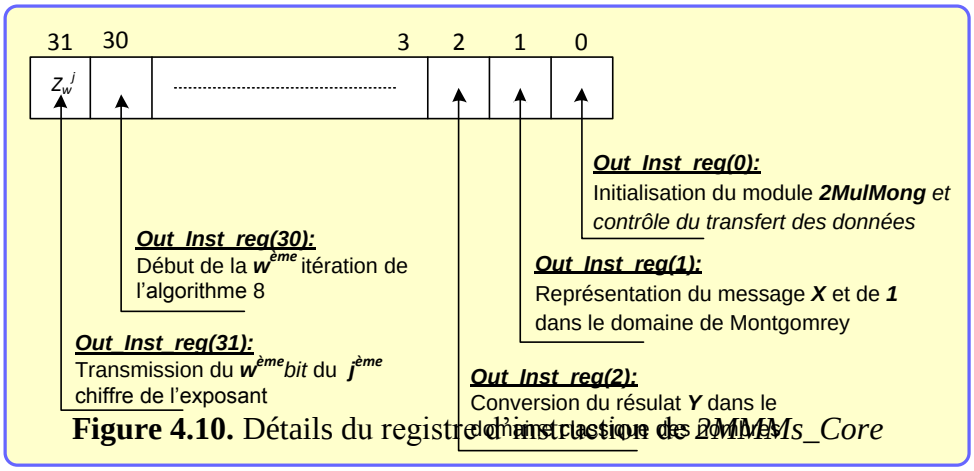


Figure 4.10. Détails du registre d'instruction de *2MMMs_Core*

Les codes binaires des instructions utilisées pour contrôler la partie matérielle du composant *2MMMs_Core* sont présentés dans le tableau 4.2.

Tableau 4.2. Format des instructions du composant *2MMMs_Core*

Instruction	***Code Binaire***	***Description***
ResetIP2MMMs	0000…..0001	Initialisation du module 2*MulMong*
RunWriteData	0000…..0000	Transfert des données.
RunMonIn	0000…..0010	Représentation du message *X* et de *1* dans le domaine de Montgomery.
RunMMMs	z100…..0010	Exécution des deux MMMs de chaque itération de l'algorithme 8.
RunMonOut	0000…..0110	Conversion du résultat *Y* dans le domaine classique des nombres.

Les signaux d'interconnexion entre les trois blocs intégrés dans *2MMMs_Core* sont définis comme suit:

- ***Bus2IP_Clk*** : Signal d'horloge, transmis par le bus système *PLB*.
- ***WFIFO2IP_Data*** : Bus de données de sortie de la mémoire *W_FIFO*. Ce bus est utilisé pour transmettre les données vers le module *2MulMong*.
- ***WFIFO2IP_RdAck*** : Ce signal permet d'indiquer par une impulsion active à l'état haut que la donnée transmise par le processeur, sera valide après le prochain top d'horloge.

- ***IP2RFIFO_Data*** : Bus de données d'entrée de la mémoire *R_FIFO*. Il assure le transfert du résultat *Y* de l'exponentiation modulaire au processeur Microblaze.
- ***In_reg1, In_reg2***: Représentent respectivement les bus d'entrées des deux registres d'états *St_Registre1* et *St_Registre2*.
- ***Out_Inst_reg(0)*** : Signal représentant le bit le moins significatif du registre d'instruction. Il permet d'une part d'initialiser le module *2MulMong* une fois l'exécution de l'exponentiation modulaire achevée ; d'autre part, d'activer le chargement des données. Les états de ce signal pour effectuer ces deux opérations sont respectivement *Out_Inst_reg(0)=1* et *Out_Inst_reg(0)=0*.
- ***Out_Inst_reg(1)*** : Ce signal correspond au bit du poids 2^1 du registre d'instruction, actif à l'état haut. Son rôle est la mise en exécution du calcul parallèle des deux MMMs.
- ***Out_Inst_reg(2) :*** Signal actif à l'état haut, associé au bit du poids 2^2 du registre d'instruction. Ce signal permet d'indiquer, d'une part, au module *2MulMong* la dernière itération de l'algorithme 8 ; d'autre part, le début de la conversion du résultat dans la représentation classique des nombres.
- ***Out_Inst_reg(30)*** : Ce signal correspondant au bit du poids 2^{30} du registre d'instruction. Il permet d'initialiser avec une impulsion active à l'état haut, le module *2MulMong* à la fin d'exécution de chaque itération de l'algorithme 8.
- ***Out_Inst_reg(31)*** : Signal représentant le bit du poids fort du registre d'instruction. Sa tâche est la transmission du $w^{ème}$ bit z_w^j de l'exposant *Z*.
- ***Reset_Op :*** Ce signal est généré par le circuit d'initialisation. Il permet d'initialiser les deux unités arithmétiques du module *2MulMong,* à la fin d'exécution des opérations arithmétiques des deux MMMs. Le déroulement de ces derniers en fonction de l'état de ce signal est montré sur le tableau 4.3.

Tableau 4.3. Exécution des deux *UAs*
en fonction du signal d'initialisation *Reset_Op*

Reset_Op	***Description***
0	Initialisation des deux unités arithmétiques
1	Calcul des opérations arithmétiques des deux MMMs

Le module *2MulMong* est considéré comme étant le composant de base permettant l'accélération des opérations arithmétiques de l'algorithme de l'exponentiation modulaire. Les tâches qui lui sont attribuées représentent la partie la plus importante de l'architecture matérielle du composant *2MMMs_Core*.

4.3.4.2. Architecture matérielle du module *2MulMong*

L'architecture interne du module *2MulMong* est montrée sur la figure 4.11. Celle-ci est constituée de:

a) Deux unités arithmétiques *UAS* et *UAC*, utilisées respectivement pour calculer en parallèle les deux résultats intermédiaire $S_{(i)}$ et $C_{(i)}$ de l'algorithme 8.
b) Deux mémoires *FIFO_S* et *FIFO_C* pour le stockage des résultats intermédiaires des deux MMMs.
c) Deux registres *N'_Reg* et *Y_Reg* de taille *k-bits*. Ces registres assurent respectivement le stockage de la constante *N'* et la synchronisation du transfert du résultat *Y* vers le processeur.
d) Six blocs mémoires, nommés *Mem_R, Mem_N, Mem_A1, Mem_B1, Mem_A2* et *Mem_B2*. Les deux premiers assurent respectivement le stockage de la valeur de $R^2 \bmod N$ et du modulo *N*. Les autres sont utilisées comme des unités de stockage temporaire. Initialement, le message *X* est chargé dans la mémoire *Mem_A1*. Ensuite, durant l'exécution de l'exponentiation modulaire, cette mémoire ainsi que *Mem_B1 et Mem_A2,* assurent le stockage du résultat $S_{(i)}$ de l'élévation au carré. Le second résultat intermédiaire, en l'occurrence $C_{(i)}$ est stocké dans *Mem_B2*. Le résultat *Y* est transféré en sortie à partir de cette mémoire. Les configurations de ces mémoires selon les phases d'exécutions de l'exponentiation modulaire, sont détaillées dans l'annexe C.
e) Trois multiplexeurs *Mux_1*, *Mux_2* et *Mux_3*. Ces derniers sont utilisés aux entrées des deux unités arithmétiques. Ils permettent de sélectionner à partir des mémoires, les opérandes d'entrée des MMMs, selon le processus d'exécution de l'algorithme 8.

    - *Mux_1* assure la sélection entre $R^2 \bmod N$ et $S_{(i)}$ pour calculer $(X \times R^2 \bmod N \times R^{-1}) \bmod N$ et $(S_{(i)} \times S_{(i)} \times R^{-1}) \bmod N$.
    - *Mux_2* et *Mux_3* sélectionnent respectivement entre $(1$ *ou* $S_{(i)})$ et $(R^2 \bmod N$ *ou* $C_{(i)})$, pour calculer: $(1 \times R^2 \bmod N \times R^{-1}) \bmod N$, $(S_{(i)} \times C_{(i)} \times R^{-1}) \bmod N$ et $(1 \times C_{(i)} \times R^{-1}) \bmod N$.

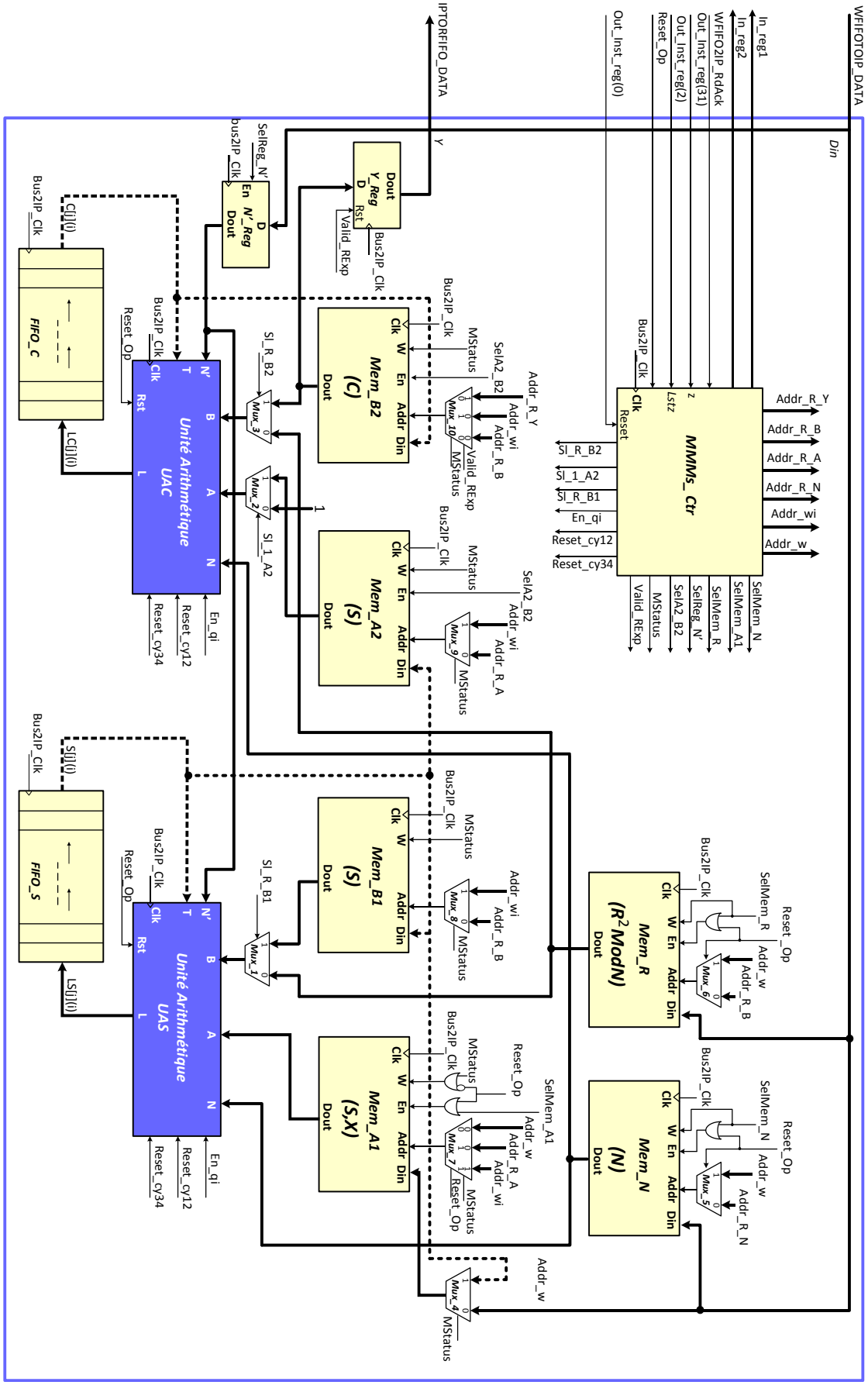


Figure 4.11. Architecture matérielle du module *2MulMong*

f) Un multiplexeur *Mux_4* assurant l'initialisation de la mémoire *Mem_A1* avec le message *X*. Ce multiplexeur est utilisé à l'entrée du bus de données de la mémoire *Mem_A1*.

g) Six multiplexeurs *Mux_5, Mux_6, Mux_7, Mux_8, Mux_9* et *Mux_10.* Ces derniers assurent la sélection des bus d'adresses qui correspondent aux phases de lecture et d'écriture aux entrées *Addr* des mémoires.

h) Un circuit de contrôle *MMMs_Ctr.* Son rôle est de générer :

- les adresses mémoires qui correspondent aux phases d'écriture et de lecture,
- les signaux de contrôle des mémoires et du registre *N'_Reg,*
- les signaux de sélection des opérandes,
- les signaux de contrôle des deux unités arithmétiques.

Les signaux d'entrée/sortie du circuit *MMMs_Ctr* sont présentés dans l'annexe C.

4.3.4.3. Etapes d'exécution de l'exponentiation modulaire par le module *2MulMong*

Le calcul de l'exponentiation modulaire $Y=X^Z \ mod \ N$ en utilisant le module *2MulMong* pour accélérer les étapes d'exécution de l'algorithme 8, est effectué en trois phases. La première est la réception des données fournies par le processeur Microblaze. La seconde correspond à l'exécution de l'algorithme 8. La troisième est associée au transfert du résultat *Y* vers le processeur.

1. Réception des données.

Initialement, le module *2MulMong* est maintenu dans son état de repos par le signal *Out_Inst_reg(0)* du registre d'instruction. Le début de chargement des données dans les unités de stockage qui leur sont associées est entamé dès que le processeur Microblaze transmet l'instruction *RunWriteData* qui force le signal *Out_Inst_reg(0)* à *″0″*.

Le stockage des données d'entrée constituées par : le modulo *N*, le message *X* la valeur de $R^2 \ mod \ N$ et la constante *N'* est basé sur :

- La sélection séquentielle de chaque unité de stockage, suivant l'ordre des données transmises par Microblaze.

- L'aiguillage du bus d'adresse *Addr_W,* assurant l'adressage des mémoires *Mem_N, Mem_A1* et *Mem_R.*

A noter que la mémoire *Mem_A1* est utilisée pour la réception du message *X*. La valeur numérique de ce dernier sera par la suite écrasée par la valeur du résultat intermédiaire $S_{(i)}$, calculée à l'itération *(i)* par l'*UAS*. La sélection entre *X* et $S_{(i)}$ à l'entrée du bus de données de la mémoire *Mem_A1* est assurée par le multiplexeur *Mux_4,* commandé par le signal *MStatus.* Ce signal est généré par le circuit de contrôle *MMMs_Ctr*. La table de vérité du multiplexeur *Mux_4* est montrée sur le tableau 4.4.

Tableau 4.4. Table de vérité du Multiplexeur *Mux_4*

MStatus	***Dout***
0	*X*
1	$S_{(i)}$

La sélection de chacune des trois mémoires *Mem_N, Mem_A1, Mem_R* et du registre *N'_Reg*, est réalisée respectivement par l'utilisation des signaux de sélection *SelMem_N, SelMem_A1, SelMem_R* et *SelReg_N',* fournis par le circuit de contrôle *MMMs_Ctr*. Ces signaux sont actifs à l'état haut seulement lors de la réception des données qui leur sont associées.

2. Exécution de l'algorithme 8

Le calcul de l'exponentiation modulaire par le module *2MulMong,* est effectué selon le processus d'exécution de l'algorithme 8 en trois étapes ; à savoir :

a. représentation du message *X* et de *"1"* dans le domaine de Montgomery,
b. exécution itérative de l'élévation au carré et de la multiplication modulaire dans le domaine de Montgomery,
c. conversion du résultat *Y* de l'exponentiation modulaire vers la représentation classique des nombres.

Dans le but de distinguer l'exécution de chacune de ces trois étapes, la sélection des opérandes aux entrées des deux unités arithmétiques *UAS* et *UAC* est assurée par les multiplexeur *Mux_1*, *Mux_2* et *Mux_3*. Ces derniers sont contrôlés respectivement par les trois signaux *Sl_R_B1*, *Sl_1_A2* et *Sl_R_B2,* générés par le circuit de contrôle

MMMs_Ctr. La sélection des opérandes en fonction de ces signaux et les étapes qui constituent l'algorithme 8, sont montrées sur le tableau 4.5.

Tableau 4.5. Sélection des opérandes en fonction des étapes de calcul de l'algorithme 8

Sel_R_B1	***Sel_1_A1***	***Sel_R_B2***	***Opérandes Sélectionnés***			***Etapes de l'algorithme 8***
			Mux_1	***Mux_2***	***Mux_3***	
0	0	0	$R^2 mod\ N$	1	$R^2 mod\ N$	**Etape a.** Calcul de : $S_{(-1)} = (X \times R^2 mod\ N \times R^{-1})\ mod\ N$ $C_{(-1)} = (1 \times R^2 mod\ N \times R^{-1})\ mod\ N$
1	1	1	$S_{(i-1)}$	$S_{(i-1)}$	$C_{(i-1)}$	**Etape b.** Calcul de : $S_{(i)} = (S_{(i-1)} \times S_{(i-1)} \times R^{-1}) mod\ N$ $C_{(i)} = (S_{(i-1)} \times C_{(i-1)} \times R^{-1})\ mod\ N$
1	0	1	-	1	$C_{(e-1)} = X^Z \times R$	**Etape c.** Calcul de : $Y = (\boldsymbol{X^Z \times R \times 1 \times R^{-1}})\ mod\ N$ $= X^Z\ mod\ N$

L'exécution parallèle des deux MMMs dans les deux unités arithmétiques *UAS* et *UAC* est basée sur le signal d'initialisation *Reset_Op*, généré par le circuit d'initialisation (voir figure 4.9). Ce signal est à l'état bas durant le chargement des données transmises par le processeur Microblaze. Il devient à l'état haut, dès le début d'exécution des opérations arithmétiques des deux MMMs. Son utilisation dans l'architecture interne du module *2MulMong* se résume dans :

- l'initialisation des deux unités arithmétiques à la fin d'exécution de chaque itération *(w)* de l'algorithme 8,
- le contrôle de la génération des adresses mémoires lors de la lecture des opérandes.

L'exécution des trois étapes de l'algorithme 8 par le module *2MulMong*, est résumée comme suit :

a. Représentation du message X et de 1 dans le domaine de Montgomery :

Cette étape est entamée dès que les données transmises par le processeur seraient entièrement stockées dans les unités de stockage qui leur sont associées. Le début de

cette étape est contrôlé par le processeur Microblaze par la transmission de l'instruction *RunMonIn.* Celle-ci, par l'intermédiaire du signal *Out_Inst_reg(1),* force le signal *Reset_Op* à l'état haut. Ainsi, durant l'exécution parallèle des deux MMMs, les deux unités arithmétiques *UAS* et *UAC* reçoivent *chiffre-par-chiffre* :

- le message *X* et *″1″* sur leurs entrées *A,*
- la valeur de R^2 *mod N* sur leurs entrées *B.*

La sélection de *″1″* comme opérande à l'entrée *A* de l'unité *UAC* est effectuée par le multiplieur *Mux2*. La sélection de R^2 *mod N* à l'entrée *B* de *UAS* et de *UAC* est assurée respectivement par les multiplieurs *Mux_1 et Mux_3*.

Les résultats issus de cette étape, à savoir : $S_{(-1)} = X \times R \ mod \ N$ est stocké dans *Mem_A1*, *Mem_B1* et *Mem_A2 ;* $C_{(-1)} = 1 \times R \ mod \ N$ est stocké dans *Mem_B2.*

b. Calcul itératif de l'élévation au carré et de la multiplication modulaire dans le domaine de Montgomery :

Dans cette étape, il est question de calculer à chaque itération *(w)* de l'algorithme 8, une l'élévation au carré et une multiplication modulaire dans le domaine de Montgomery. Ces deux opérations sont définies telles que :

$$S_{(i)} = \left(S_{(i-1)} \times S_{(i-1)} \times R^{-1}\right) mod \ N$$
$$C_{(i)} = \left(S_{(i-1)} \times C_{(i-1)} \times R^{-1}\right) mod \ N$$

Le début d'exécution de chaque itération de l'algorithme 8 est contrôlé par le processeur Microblaze, en utilisant l'instruction *RunMMMs*. Celle-ci est utilisée, d'une part, pour transmettre le $i^{ème}$ bit de l'exposant Z ; d'autre part, pour contrôler le signal d'initialisation *Reset_Op*.

Pour calculer la première opération (l'élévation au carré), le bus de données de sortie de la mémoire *Mem_A1* est inséré directement à l'entrée de l'opérande *A* de l'unité *UAS.* Le multiplexeur *Mux_1* sélectionne vers l'entrée de l'opérande *B* de l'*UAS,* le bus de données de sortie de la mémoire *Mem_B1*.

Le résultat $C_{(i)}$ de la deuxième opération (la multiplication modulaire) est obtenu par l'utilisation des multiplexeurs *Mux_2* et *Mux_3.* Ces derniers sélectionnent respectivement aux entrées *A* et *B* de l'unité *UAC* les bus de données de sortie des deux mémoires *Mem_A2* et *Mem_B2.*

Le processus de calcul dans le domaine de Montgomery est effectué de la même manière sur l'ensemble des itérations, jusqu'à épuisement de tous les bits de l'exposant. Le résultat obtenu à la fin de cette étape, on l'occurrence $C_{(e-1)} = (X^Z \times R)\ mod\ N$ est stocké dans la mémoire *Mem_B2*.

c. Représentation du résultat Y de l'exponentiation modulaire dans le domaine de Montgomery

La sortie du domaine de Montgomery se traduit par l'élimination du facteur *R* du résultat $C_{(e-1)}$. Cette opération correspond à la dernière MMM qui constitue la troisième étape de l'algorithme 8. Les opérations arithmétiques de cette étape sont effectuées dans l'unité arithmétique *UAC*. Celle-ci reçoit à travers les deux multiplexeurs *Mux_2* et *Mux_3* respectivement *"1"* et le résultat $C_{(e-1)}$ de l'étape précédente, pour calculer : $Y = (X^Z \times R \times 1\ \times R^{-1})\ mod\ N = X^Z\ mod\ N$.

Le début d'exécution de cette étape est contrôlé par le processeur Microblaze. En effet, pour indiquer au module *2MulMong* la fin d'exécution des MMMs dans le domaine de Montgomery, Microblaze transmet l'instruction *RunMonOut*. Cette dernière par l'intermédiaire du signal *Out_Inst_Reg(2)* du registre d'instruction, contrôle le basculement à l'état bas du signal de sélection *Sl_1_A2* du multiplexeur *Mux_2*. Ce qui permet la sélection de *"1"* à l'entrée *A* de l'*UAC*.

3. Transfert du résultat Y de l'exponentiation modulaire vers le processeur Microblaze.

Le transfert du résultat *Y* vers le processeur est effectué à travers le registre *Y_Reg*. En effet, comme le débit de la liaison assurant la communication entre le processeur Microblaze et le composant *2MMMs_Core* est défini par l'impulsion du signal *WFIFO2IP_RdAck*, celle-ci est utilisée pour générer l'adressage de la mémoire *Mem_B2* lors de la lecture du résultat *Y*. La sélection du bus d'adresse correspondant, en l'occurrence *Addr_R_Y* est assurée par le multiplexeur *Mux_10*. Pour distinguer la phase du transfert du résultat *Y* des deux phases précédentes, le circuit de contrôle *MMMs_Ctr* génère le signal *Valid_RExp*. Ce dernier est à *"1"* seulement durant l'exécution de cette troisième phase.

L'utilisation du signal *Valid_RExp* permet au module *2MulMong* d'indiquer son état au processeur Microblaze. En effet, lors du transfert des données vers *2MMMs_Core* et durant l'exécution de l'algorithme de l'exponentiation modulaire,

Valid_RExp est forcé *à "0"*. Ce dernier maintient le registre *Y_Reg* dans un état initial, défini par *0xFFFFFFFF*. Lors du transfert du résultat *Y*, *Valid_RExp* bascule à l'état *"1"*. Ensuite, le module *2MulMong* procède par initialisation du registre *Y_Reg* par une donnée dont tous les bits sont nuls (*0x00000000*), afin d'indiquer au processeur Microblaze le début du transfert de *Y*. Lorsque Microblaze reçoit cette donnée, il ne prend en considération que les *n chiffres* qui succéderont au premier zéro. Une fois le transfert de *Y* achevé, *2MMMs_Core* est remis à son état initial par l'intermédiaire du signal *Out_Inst_reg(0)*. Ce dernier est forcé à *"1"* dès que le processeur Microbalze transmet l'instruction *ResetIP2MMMs.* Les résultats des simulations fonctionnelles associés au transfert du résultat *Y* sont présentés sur la figure 6.6 du chapitre 6.

4.4. Conclusion

Dans ce chapitre, nous avons présenté les approches proposées pour l'implémentation dans un système embarqué des deux opérations de base du crypto système RSA, en l'occurrence la multiplication modulaire et l'exponentiation modulaire. Nous avons entamé par une description globale de la plateforme de chiffrement et de déchiffrement réalisée. Ensuite, nous avons présenté les architectures matérielles implémentées sur circuit FPGA. Ces architectures correspondent principalement :

- au système embarqué,
- à l'unité arithmétique, conçue pour adapter l'exécution de la MMM aux ressources internes du processeur Microblaze,
- aux deux composants *MMM_Core* et *2MMMs_Core*, considérés comme étant des accélérateurs matériels pour calculer la fonction d'exponentiation modulaire.

Les implémentations présentées dans ce chapitre, ont été conçues à l'origine pour atteindre la réalisation d'un crypto système RSA qui présente le meilleur compromis entre le temps d'exécution et les ressources matérielles occupées. De ce fait, pour réunir ces deux contraintes dans une même implémentation et rajouter de la flexibilité au contrôle du crypto système, il est nécessaire de compléter sa conception par sa partie logicielle.

Le prochain chapitre sera dédié à la présentation des parties logicielles réalisées, pour le contrôle des deux composants *MMM_Core* et *2MMMs_Core* et pour la configuration du crypto système RSA.

Chapitre 5

Implémentations logicielles du crypto système RSA

5.1 Introduction

La réalisation de la plateforme de chiffrement et de déchiffrement, basée sur les deux composants *MMM_Core* et *2MMMs_Core*, nécessite non seulement l'implémentation de leurs architectures matérielles, mais aussi la conception de leurs parties logicielles.

Ainsi, pour compléter les approches d'implémentations présentées dans le chapitre 4, les parties logicielles développées, ont été réalisées sur deux niveaux d'abstractions. Le premier est implémenté sur circuit FPGA. Il est constitué par des fonctions décrites en langage C. Ces dernières sont exécutées par le processeur Microblaze pour contrôler l'exécution de l'exponentiation modulaire, en utilisant les approches étudiées dans le chapitre précédent. Le second niveau d'abstraction est une Interface Homme/Machine *(IHM)*, développée en langage Jave. Celle-ci est exécutée sur un ordinateur. Son rôle réside dans :

- La configuration de la plateforme de chiffrement/déchiffrement RSA.
- Le transfert des données entre l'ordinateur et la partie embarquée sur circuit FPGA.

De plus, dans le but de constituer un autre modèle de comparaison pour la mise en œuvre de l'exponentiation modulaire, une troisième approche d'implémentation est présentée dans ce chapitre. Celle-ci est basée sur l'exploitation de la partie matérielle du système embarqué, implémenté sans aucun composant matériel personnalisé. Cette troisième approche est une implémentation purement logicielle, où les algorithmes de la MMM et de l'exponentiation modulaire son exécutés par le processeur Microbalze. Ce chapitre est consacré à la présentation des différentes parties logicielles développées. Il est organisé en cinq parties. La première est une description globale

des pilotes logiciels exécutés par le processeur Microblaze. La seconde et la troisième décrivent respectivement les parties logicielles des deux composants *MMM_Core* et *2MMMs_Core*. La quatrième partie présente la troisième approche d'implémentation. La cinquième partie est consacrée à la description de l'*IHM*.

5.2 Partie logicielle du système embarqué sur FPGA

La partie logicielle du crypto système présenté dans le chapitre 4 a été réalisée en utilisant l'outil SDK de Xilinx. Elle est constituée par un certain nombre de fonctions. Ces dernières sont stockées dans la mémoire BRAM du processeur, puis exécutées par ce dernier.

Le rôle attribué au processeur Microblaze pour le déroulement de l'algorithme de l'exponentiation modulaire peut se résumer dans :

- La réception et la restitution des données sur des mots de taille *k bits*.
- Le décalage *bit-par-bit* de l'exposant *Z.*
- Le contrôle de la partie matérielle des deux composants *MMM_Core* et *2MMMs_Core*.
- L'exécution des algorithmes de multiplication de Montgomery et de l'exponentiation modulaire, dans le cas de l'implémentation purement logicielle (troisième approche d'implémentation).
- Le contrôle du *Timer* pour évaluer la complexité temporelle de chacune des trois approches d'implémentations proposées.
- Transmission du résultat vers l'ordinateur.

Les codes développés pour l'implémentation de ces fonctionnalités sont repartis selon une hiérarchie dont le nombre de nivaux dépend de l'approche utilisée pour l'implémentation du système embarqué. L'ensemble des fonctions implémentées sont montrées sur la figure 5.1.

La description en langage C du programme principal *(main(.))* est identique pour les trois approches proposées. Il est constitué des fonctions de communication avec l'*UART*, de la fonction permettant le contrôle du *Timer* et de la fonction de calcul de l'exponentiation modulaire. Celle-ci peut être une des trois fonctions : *ExpBinary1(.)*, *ExpBinary2(.)* ou *ExpBinary3(.)*. La sélection de ces dernières pour l'implémentation du système embarqué, dépend de la l'approche utilisée pour l'exécution de l'exponentiation modulaire.

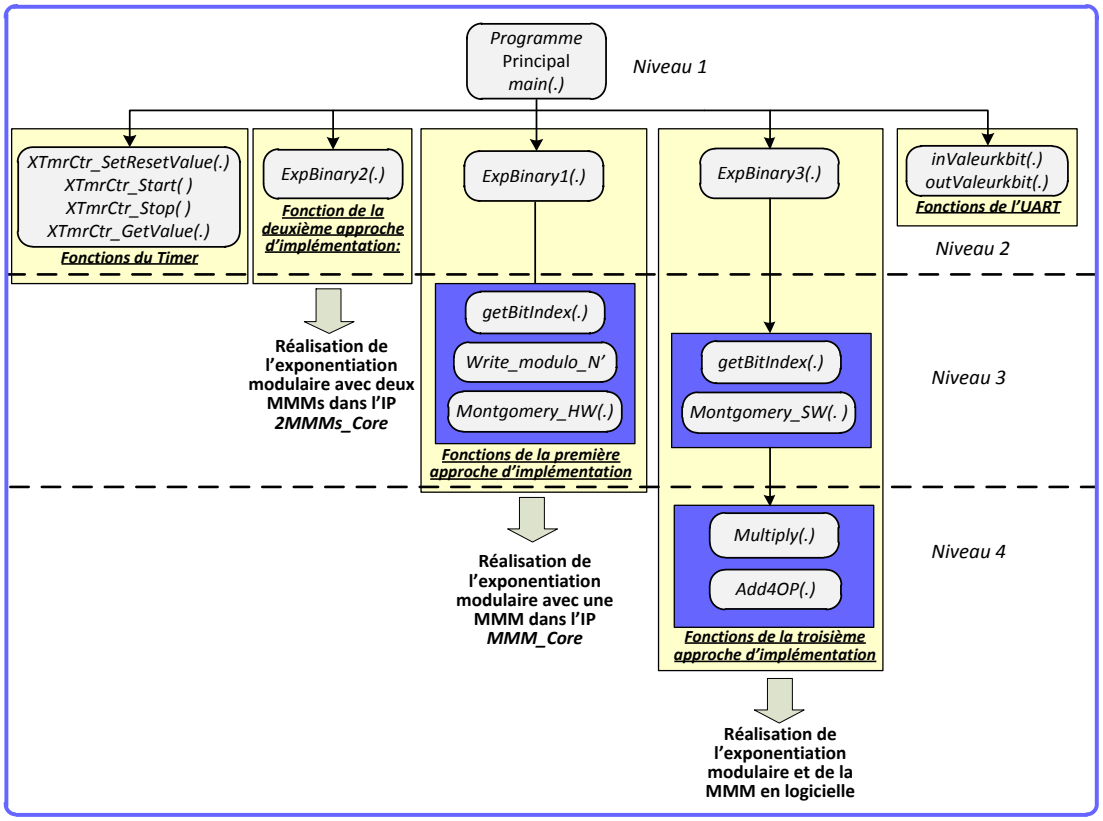


Figure 5.1. Organisation des fonctions des parties logicielles du système embarqué selon l'approche d'implémentation

L'organigramme du programme *main (.)* est montré sur la figure 5.2. Les tâches exécutées pour calculer une exponentiation modulaire sont réparties sur quatre étapes :

1. Initialisation des variables

Dans cette étape, toutes les variables intermédiaires sont initialisées à zéro pour entamer la réception des données d'entrée.

2. Réception des données d'entrées

Dans cette étape, l'ensemble des données transmises par l'*IHM* sont restituées et stockées dans la mémoire locale du processeur sur des chiffres de taille *k-bits*. Le déroulement de cette étape est basé sur la fonction *inValeurkbit(.)*. Son code C est présenté sur la figure 5.3.

En effet, comme la liaison de communication entre l'*IHM* et la carte de prototypage Genesys utilisée est de type RS232, à chaque trame de donnée transmise par le l'ordinateur, seulement *8-bits* sont actifs. De ce fait, le rôle de la fonction

inValeurkbit(.), est de restituer une donnée codée sur *k-bits* à partir des données d'entrée codées sur *8-bits.* Son implémentation repose sur :

- Une fonction assurant la transmission vers le processeur Microblaze, des *8-bits* reçus à travers l'*UART*. La fonction en question est définie par : *XUartLite_RecvByte(.).* Celle-ci est fournie par Xilinx dans EDK [85].
- Des décalages à gauche de *8-bits*, suivant un indice *(j)*.

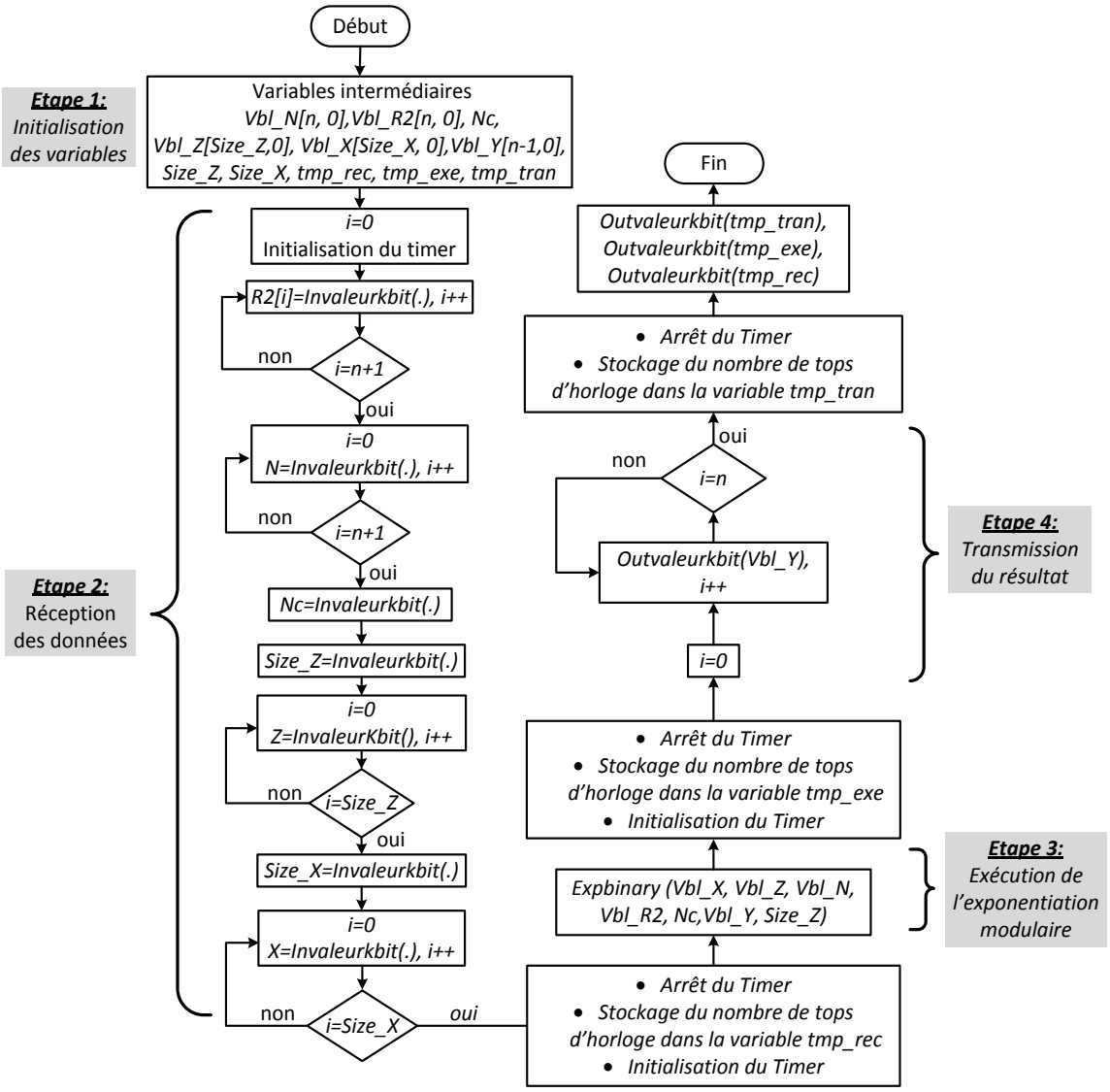


Figure 5.2. Organigramme du programme principal *"main(.)"*

Pour permettre l'organisation des données mises en exécution entre l'*IHM* et le processeur MicroBlaze, nous avons introduit des variables intermédiaires dans le code C du programme principal. Les variables en question, leurs tailles et leurs désignations sont résumées dans le tableau 5.1.

```
inline unsigned long inValeurkBit()
{
  unsigned long a=0,tmp;int j;
     for (j=0; j<= (k/8-1); j++)
         {
             tmp=XUartLite_RecvByte(baseaddr)<<(8*i);
            a|=tmp;
         }
            return a;
 }
```

Figure 5.3. Code C de la fonction *"inValeurkbit(.)"*

Tableau 5.1. Variables intermédiaires du programme principal

Variable	***Taille***	***Désignation***
Vbl_N	(n+1)× k bits	Modulo
Vbl_R2	(n+1)× k bits	Constante R^2 *mod N*
Nc	k bits	Constante *N'* de Montgomery
Vbl_Z	k bits	Exposant
Vbl_X	(n+1)× k bits	Message
Vbl_Y	n× k bits	Résultat de l'exponentiation modulaire
Size_Z	k bits	Taille de l'exposant en termes du nombre de chiffre
Size_X	k bits	Nombre de chiffre de taille *k bits* dans le message *X*
tmp_rec	32 bits	Nombre de tops d'horloge correspondant à la réception des données
tmp_exe	32 bits	Nombre de tops d'horloge correspondant à l'exécution de l'exponentiation modulaire
tmp_tran	32 bits	Nombre de tops d'horloge correspondant à la transmission du résultat

Une fois l'étape de la réception des données achevée, la valeur indiquée par le *Timer* est stockée dans la variable *tmp_rec*. Celle-ci sera utilisée dans l'application Java pour calculer le délai écoulé durant cette étape. Ensuite, le *Timer* sera initialisé pour entamer le comptage des tops d'horloge de l'étape suivante. Les fonctions associées au *Timer* sont définies comme suit :

- ***XTmrCtr_SetResetValue(.):*** Assure son initialisation.
- ***XTmrCtr_Start (.), XTmrCtr_Stop(.):*** Permettent respectivement d'activer et de désactiver son fonctionnement.

- ***XTmrCtr_GetValue(.):*** Permet de sauvegarder dans une variable intermédiaire le nombre de tops d'horloge calculé.

Ces fonctions sont fournies par l'outil EDK [85].

3. Exécution de l'exponentiation modulaire

Dans cette étape, on effectue le calcul de l'exponentiation modulaire $Y=X^Z \bmod N$, en utilisant l'une des trois fonctions *ExpBinary1(.)*, *ExpBinary2(.)* et *ExpBinary3(.)*. Une fois l'exécution achevée, le résultat sera stocké *chiffre-par-chiffre* dans la variable *Vbl_Y [n-1, 0]*. De même, le nombre de tops d'horloge calculé par le *Timer*, sera sauvegardé dans la variable *tmp_exe*.

4. Transmission du résultat Y

Cette étape correspond à la transmission du résultat *Y* vers l'extérieur du circuit FPGA. La fonction utilisée, en l'occurrence *outValeurkbit(.)*, reçoit en entrée le résultat *Y* sur des chiffres de taille *k-bits*. Cette fonction assure leur décomposition et leur transmission vers l'*UART* sur des chiffres de taille *8-bits*. Le code C de la fonction *outValeurkbit(.)* est donné sur la figure 5.4. Son implémentation est basée sur :

- Des décalages à droite suivant l'indice *(j)*.
- La fonction *XUartLite_SendByte(baseaddr, data),* fournie par Xilinx [85]. Celle-ci permet de transmettre à chaque itération *(j), 8-bits par 8 bits* le résultat de l'exponentiation modulaire vers l'extérieur du circuit FPGA.

```
inline void outValeurkbit (unsigned long valeur)
  {
     int j;
       for(j=0;j<=(k/8-1);j++)
      XUartLite_SendByte(0x baseaddr,valeur>>(8*j));
  }
```

Figure 5.4. Code C de la fonction *"outValeurkbit(.)"*

Une fois la transmission du résultat de l'exponentiation modulaire achevée, le nombre de tops d'horloge est stocké dans la variable *tmp_tran.* Le nombre en question ainsi que *tmp_rec* et *tmp_exe* qui correspondent aux deux étapes précédentes seront transmis, en utilisant la fonction *outValeurkbit(.)* vers l'application Java.

Les délais d'exécution t_{rec}, t_{exe} et t_{tran} qui correspondent aux étapes : de réception des données, d'exécution de l'exponentiation modulaire et de transmission du résultat, sont calculés et affichés dans l'*IHM*. A noter que ces délais sont calculés par la multiplication du nombre de tops d'horloge associé à chaque étape par la période d'horloge du système embarqué.

5.3 Partie logicielle du composant *MMM_Core* de la première approche d'implémentation

La partie logicielle du composant *MMM_Core* proposée dans la première approche d'implémentation, est constituée par des fonctions écrites en langage C. Ces dernières sont réparties sur deux niveaux d'abstraction (voir figure 5.1). Le bas niveau d'abstraction est constitué par :

- la fonction de décalage de l'exposant *"getBitIndex(.)"*,
- la fonction de chargement du modulo et de la constante *N' "Write_Modulo_N'(.)"*,
- la fonction de contrôle de *MMM_Core "Montgomery_HW(.)"*,

Le plus haut niveau d'abstraction comporte la fonction *ExpBinary1(.).* Celle-ci permet de contrôler de manière logicielle l'exécution l'algorithme 8.

5.3.1 Fonction de décalage de l'exposant *"getBitIndex(.)"*

Le rôle attribué à cette fonction est d'effectuer, à chaque itération *(w)* de l'algorithme 8, un décalage à droite sur le $j^{ème}$ chiffre de l'exposant *Z* ; afin d'obtenir son $w^{ème}$ bit z_w^j.

5.3.2 Fonction de chargement du modulo et de la constante *N' "Write_Modulo_N'(.)"*

Cette fonction permet de transmettre le modulo *N* et la constante *N'* vers l'architecture interne du module *MulMong* de *MMM_Core* (voir figure 4.6). A noter que la transmission de *N* et de *N'* est effectuée une seule fois, lors du début d'exécution de l'algorithme 8. L'organigramme de la fonction *Write_Modulo_N'(.)* est illustré sur la figure 5.5.

La fonction *Write_Modulo_N'(.)* reçoit en entrée le modulo *N*, la constante *N'* et l'instruction *"RunWriteModNc"*, définie dans le tableau 4.1 du chapitre 4. Son implémentation repose sur :

1. Les pilotes permettant l'écriture dans le registre d'instruction *Inst_Registre* et dans la mémoire *W_FIFO* (voir figure 4.6). Les fonctions permettant la communication avec ces deux composants sont respectivement :
    - *MMM_Core_mWriteSlaveReg0(baseaddr,0, RunWriteModNc)*
    - *MMM_Core _mWriteToFIFO(baseaddr,0,N[j]).*
2. L'utilisation d'un indice *j* pour transmettre *chiffre-par-chiffre* le modulo *N*.

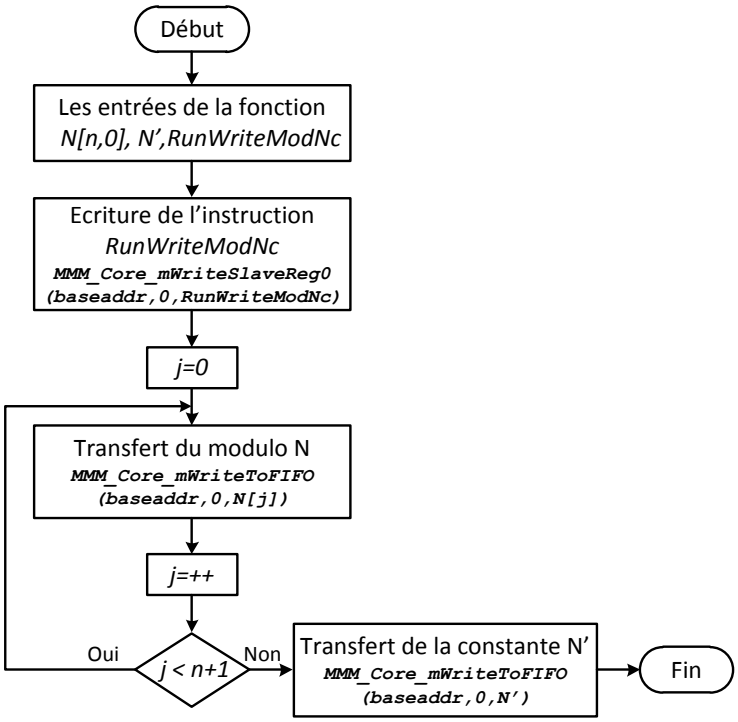


Figure 5.5. Organigramme de la fonction *"Write_Modulo_N'(.)"*

Avant d'entamer le transfert du modulo *N* et de la constante *N'* vers le module *MulMong*, Microblaze procède en premier lieu par la transmission de l'instruction *"RunWriteModNc"*. Une fois celle-ci est décodée par le composant *MMM_Core*, *N* et *N'* seront transmis séquentiellement. Le transfert des $j^{ème}$ chiffres du modulo *N* et de la constante *N'* est effectué suivant l'incrémentation de l'indice *j* de *0* à *n+1*. Les valeurs de ce dernier pour distinguer entre *N* et *N*' sont respectivement $0 \leq j < n+1$ et $j=n+1$.
Une fois les deux données sont complément chargées dans l'architecture interne du module *MulMong*, le processeur Microblaze utilise la fonction *Montgoemry_HW(.)* pour contrôler de façon logicielle l'exécution de *MMM_Core*.

5.3.3 Fonction de contrôle de *MMM_Core "Montgomery_HW(.)"*

Cette fonction est utilisée par le processeur pour contrôler :

1. Le transfert des deux opérandes *A* et *B* vers l'architecture interne du module *MulMong*.
2. L'exécution des opérations arithmétiques de la MMM par l'*UA* du module *MulMong*.
3. La transmission du résultat *T* obtenu, vers la mémoire locale du processeur.

Son organigramme est illustré sur la figure 5.6.

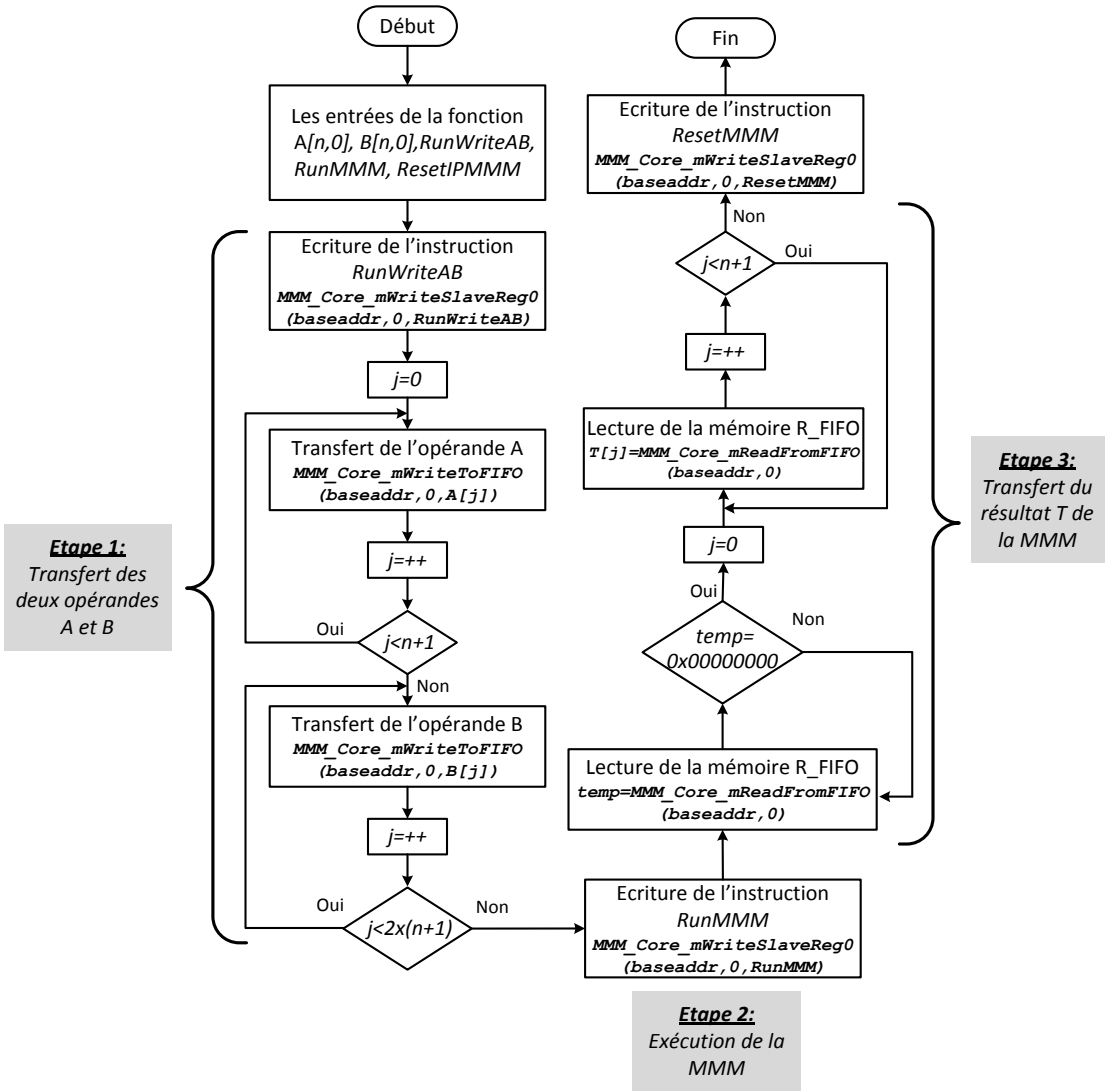


Figure 5.6. Organigramme de la fonction *"Montgomery_HW(.)"*

1. Transfert des deux opérandes A et B.

Pour activer le transfert de deux opérandes, Microblaze procède par la transmission de l'instruction *"RunWriteAB"*, définie dans le tableau 4.1. Une fois celle-ci est décodée par *MMM_Core*, Microblaze transmet ensuite séquentiellement les deux opérandes *A* et *B* qui seront stockés dans l'architecture du module *MulMong*.

Pour différentier entre ces opérandes, le transfert de leurs $j^{ème}$ chiffres est effectué en utilisant un indice *j*, variant de *0* à $2\times(n+1)$. Les valeurs de *j* associées à chaque opérande sont respectivement $0\leq j<n+1$ et $n+1\leq j<2\times(n+1)$.

Les fonctions utilisées pour transmettre l'instruction *"RunWriteAB"* et les $j^{ème}$ chiffres de *A* et de *B* à chaque itération *(j)* sont:

- *MMM_Core_mWriteSlaveReg0(baseaddr,0,RunWriteAB),*
- *MMM_Core _mWriteToFIFO(baseaddr,0,A[j]).*
- *MMM_Core _mWriteToFIFO(baseaddr,0,B[j]).*

2. Exécution des opérations arithmétiques de la MMM

Cette étape correspond à l'exécution sur matériel des opérations arithmétiques de l'algorithme 7, utilisé pour calculer la MMM. En fait, une fois les opérandes *A* et *B* se trouvant complétement stockés dans l'architecture interne du module *MulMong*, Microblaze transmet à l'itération $j=2\times(n+1)$ l'instruction *"RunMMM"*. La fonction utilisée est : *MMM_Core_mWriteSlaveReg0(baseaddr,0,RunMMM)*. Puis, l'exécution de la MMM sera effectuée sur matériel, en utilisant le module *MulMong*. Celle-ci est détaillée dans paragraphe 4.3.3.3 du chapitre 4.

3. Transfert du résultat T de la MMM vers le processeur Microblaze.

Le transfert de *T* est basé sur l'état du bus *IP2RFIFO_Data*, assurant l'interconnexion entre la sortie du module *MulMong* et le bus de données d'entrée de la mémoire *R_FIFO* (voir figure 4.6). En effet, durant l'étape de transmission des opérandes et lors de l'exécution de la MMM, la liaison en question est forcée à l'état haut. En d'autres termes, le module *MulMong* transmet à travers le bus *IP2RFIFO_Data* au processeur Microbalze la donnée *0xFFFFFFFF*. Une fois le résultat *T* complémentent calculé, *MulMong* initialise la mémoire *R_FIFO* par une donnée dont tous les bits sont nuls (*0x00000000*). Celle-ci sera suivie par les *n+1 chiffres* de *T*. La réception du $j^{ème}$ chiffre par le processeur Microblaze est effectuée en utilisant la fonction *MMM_CORE_mReadFromFIFO(baseaddr,0).* Celle-ci est associée à la lecture des données transmises à travers la mémoire *R_FIFO*.

5.3.4 Fonction d'exécution de l'exponentiation modulaire *"ExpBinary1(.)"*

La fonction *ExpBinary1(.)* permet de contrôler de manière logicielle l'exécution de l'exponentiation modulaire $Y=X^Z \bmod N$. Dans le but d'accélérer le calcul des deux MMMs de chaque itération *(w)* de l'algorithme 8, ces deux opérations sont exécutées

séquentiellement dans l'architecture interne du composant *MMM_Core*. Le processus d'exécution est basé sur les trois fonctions précédentes.

Pour calculer une exponentiation modulaire, Microblaze transmet en premier lieu le modulo *N* et la constante *N'*, en utilisant la fonction *Write_Modulo_N'(.)*. Ensuite, pour exécuter une MMM, le composant *MMM_Core* reçoit deux opérandes *A* et *B*. En effet, selon le déroulement de l'algorithme 8, *A* et *B* peuvent être le message *X*, *"1"*, la valeur de $R^2 \bmod N$ ou les deux résultats intermédiaires $S_{(i)}$ et $C_{(i)}$. Le résultat *T* issu de chaque MMM est transféré à la mémoire BRAM, pour qu'il soit introduit comme opérande d'entrée lors de l'exécution de la prochaine opération. Le décalage des $j^{ème}$ chiffres de l'exposant Z et le contrôle de *MMM_Core* durant le processus itératif de l'algorithme 8, sont effectués par le processeur Microblaze en utilisant respectivement les fonctions *GetBitIndex(.)* et la fonction *Montgomery_HW(.)*.

5.4 Partie logicielle du composant 2*MMMs_Core* de la seconde approche d'implémentation

Le contrôle logicielle du composant *2MMMs_Core* conçue pour optimiser le délai d'exécution de l'exponentiation modulaire $Y=X^Z \bmod N$, est basé sur la fonction *ExpBinary2(.)*. Celle-ci est utilisée par le processeur Microblaze pour contrôler l'exécution parallèle des deux unités arithmétiques *UAS* et *UAC* du module *2MulMong* (voir figure 4.11). L'organigramme de la fonction *ExpBinary2(.)* est montré sur la figure 5.7.

La synchronisation de manière logicielle des modules implémentés dans le composant *2MMMs_Core*, repose sur les instructions définies dans le tableau 4.2 du chapitre 4. L'exécution de l'exponentiation modulaire utilisant *ExpBinary2(.)*, est effectuée en trois phases :

1. la première consiste en la transmission des données d'entrées vers l'architecture interne du composant *2MMMs_Core*,
2. la seconde correspond à l'exécution de l'algorithme 8,
3. la troisième est associée au transfert du résultat vers le processeur Microblaze.

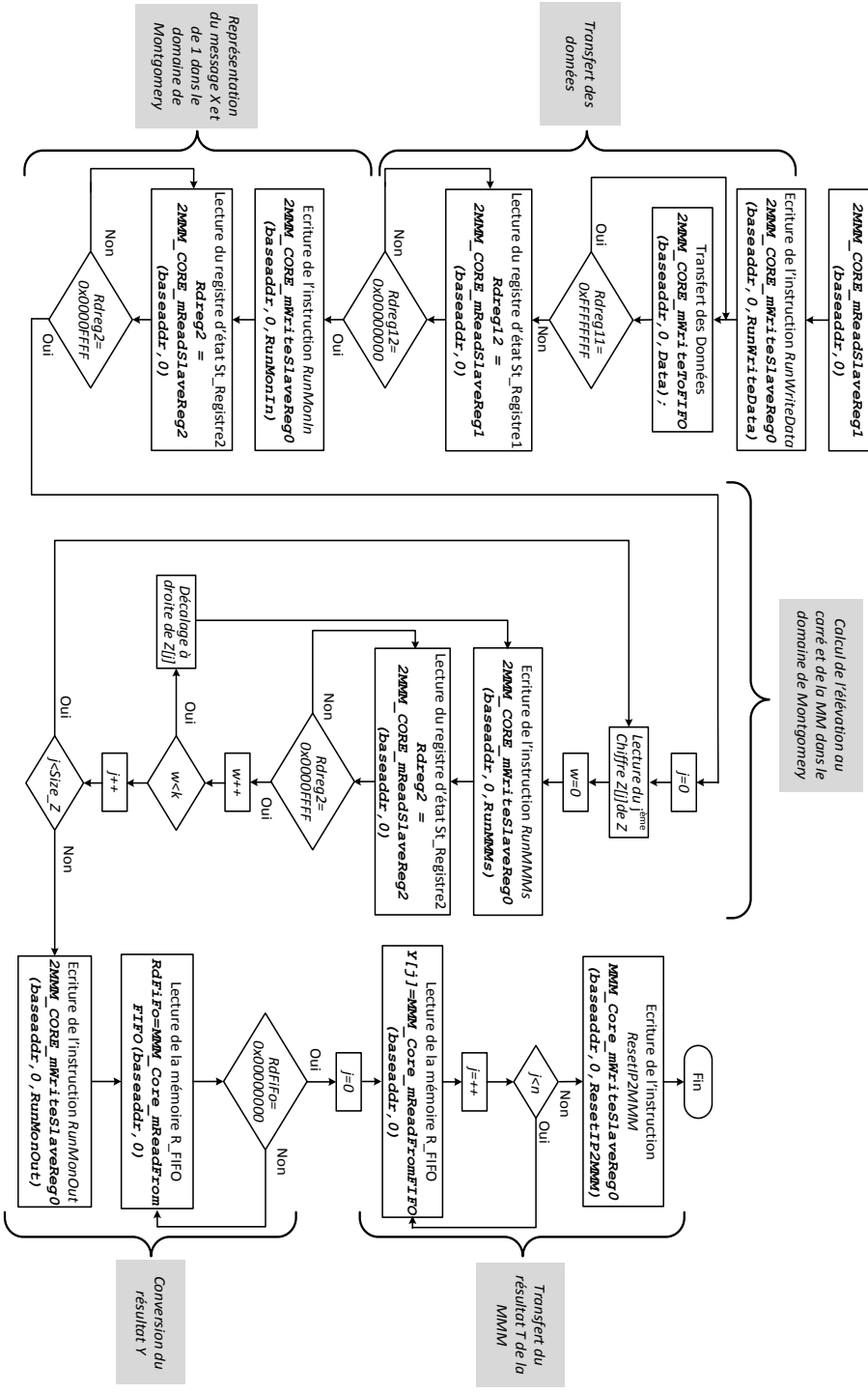


Figure 5.7. Organigramme de la fonction *"ExpBinary2(.)"*

1. Transmission des données d'entrées

Lors de cette première phase, la fonction *ExpBinary2(.)* est utilisée par le processeur Microblaze pour transmettre les données d'entrée vers l'architecture interne du composant *2MMMs_Core*. Les données en question sont représentées en *base* 2^k sur des chiffres de taille *k-bits*. Elles sont constituées par :

- le $j^{ème}$ chiffre de l'exposant *Z.*
- le message *X* où $X = \sum_{j=0}^{n} X[j] \times 2^{j \times k}$,
- le modulo *N* où $N = \sum_{j=0}^{n} N[j] \times 2^{j \times k}$,
- la valeur de $R^2 mod\ N$, définie telle que : $R^2 mod\ N = R2 = \sum_{j=0}^{n} R2[j] \times 2^{j \times k}$
- la constante *N'* et la taille de l'exposant *Z (Size_Z).*

Initialement, *2MMMs_Core* est maintenu dans son état de repos par le signal *Out_Inst_reg(0)* qui est à l'état haut (voir figure 4.9). Le début de la transmission des données vers le composant *2MMMs_Core* dépend d'une part de l'état du registre *St_Registre1*, contrôlé par le module *2MulMong* ; d'autre part, du basculement à l'état bas du signal *Out_Inst_reg(0).*

En effet, avant la transmission des données, le processeur MicroBlaze procède en premier lieu par la lecture du *St_Registre1*. Si l'état de ce dernier est *0xFFFFFFFF*, le processeur transmet l'instruction *"RunWriteData"* pour entamer le transfert des données.

Les fonctions associées aux composants de l'interface *IPIF* utilisés lors de cette étape, en l'occurrence le registre d'état *St_Registre1* et le registre d'instruction, sont respectivement : *2MMMs_Core_mReadSlaveReg1(baseaddr,0)* et *2MMMs_Core _mWriteSlaveReg0(baseaddr,0,RunWriteData).*

Dans le but de contrôler l'ordre du transfert de toutes les données, un indice *j* est utilisé dans le code C de la fonction *ExpBinary2(.).* Cet indice s'incrémente à chaque fois qu'une donnée de taille *k-bits* est transmise par le processeur. Les valeurs de *j* et les fonctions utilisées suivant la donnée en entrée, sont résumées dans le tableau 5.2.

2. Exécution de l'algorithme d'exponentiation modulaire

Dans cette seconde phase, il est question de mettre en exécution les trois étapes qui constituent le déroulement de l'algorithme 8, à savoir :

1. représentation du message *"X"* et de *"1"* dans le domaine de Montgomery,

2. exécution itérative de l'élévation au carré et de la multiplication modulaire dans le domaine de Montgomery,
3. conversion du résultat *Y* de l'exponentiation modulaire vers la représentation classique des nombres.

Tableau 5.2. Organisation de la transmission des données vers *2MMMs_Core*

Donnée	***j***	***fonction***
Le modulo N	*0≤j<n+1*	*2MMMs_CORE_mWriteToFIFO (baseaddr,0,N[j]);*
Le message X	*n+1≤j<[2x(n+1)]+1*	*2MMMs_CORE_mWriteToFIFO (baseaddr,0,X[j-(n+1)]);*
R^2mod N	*2x(n+1)+1≤[3x(n+1)]+1*	*2MMMs_CORE_mWriteToFIFO (baseaddr,0,R2[j-(2*(n+1))]);*
La constante N'	*[3 x (n+1)]+1*	*2MMMs_CORE_mWriteToFIFO (baseaddr,0,Nc)*

En effet, une fois les données entièrement transmises vers le module *2MulMong*, ce dernier transforme l'état du registre *St_Registre1* à *0x00000000*, afin d'indiquer au processeur Microblaze sa disponibilité pour entamer l'exécution de l'exponentiation modulaire. Le processeur de son côté, transmet l'instruction *"RunMonIn"* pour valider l'exécution de la première étape de l'algorithme 8. Cette instruction est transmise en utilisant la fonction *2MMMs_CORE_mWriteSlaveReg0(baseaddr,0,RunMonIn).*

A la fin d'exécution de cette première étape les résultats $S_{(-1)}$ et $C_{(-1)}$ obtenus sont stockés dans les mémoires *Mem_A1*, *Mem_B1*, *Mem_A2* et *Mem_B2* du module *2MulMong.* Ces résultats sont :

$$S_{(-1)} = (X \times R^2 mod\ N\ \times R^{-1})\ mod\ N = X \times R\ mod\ N$$

$$C_{(-1)} = (1 \times R^2 mod\ N\ \times R^{-1})\ mod\ N = R\ mod\ N$$

Une fois le stockage de ces résultats accompli, le processus de calcul de l'exponentiation modulaire entame l'exécution de la seconde étape. Celle-ci correspond au calcul itératif de l'élévation au carré et de la multiplication modulaire dans le domaine de Montgomery. Les résultats intermédiaires $S_{(i)}$ et $C_{(i)}$ calculés à chaque itération *(w)* de l'algorithme 8 sont donnés par les expressions suivantes :

$$S_{(i)} = \left(S_{(i-1)} \times S_{(i-1)} \times R^{-1}\right) mod\ N$$
$$C_{(i)} = \left(S_{(i-1)} \times C_{(i-1)} \times R^{-1}\right) mod\ N$$

Dans le but de contrôler le calcul itératif de ces deux opérations, l'exécution de cette étape repose sur l'utilisation de deux boucles imbriquées. Ces dernières sont définies dans le code C de la fonction *ExpBinary2(.)* par deux indices *(j)* et *(w)* (voir figure 5.7).

- La boucle *w (0 ≤ j < size_Z)* est externe. Celle-ci nous permet de parcourir les $j^{ème}$ chiffres de l'exposant *Z*.
- La boucle *w (0 ≤ w < k)* est imbriquée dans la boucle *j*. Celle-ci nous permet de parcourir les bits z_w^j du $(j)^{ème}$ chiffre de *Z*.

Le processus itératif qui correspond aux calculs des deux MMMs procède par la lecture du premier chiffre $Z[0]$ de *Z*. Ce dernier sera introduit par la suite *chiffre-par-chiffre*, selon l'indice *j*. L'incrémentation *(j+1)* qui correspond au $(j+1)^{éme}$ chiffre de *Z* est conditionnée par la valeur de l'indice *(w)*. En effet, une fois avoir parcouru tous les bits du $j^{ème}$ chiffre de *Z*, ce qui correspond à *w=k-1*, l'indice *(w)* est remis à *0 (w=0)* ; l'indice *(j)* est incrémenté *(j=j+1)*.

Pour contrôler l'exécution des deux MMMs de chaque itération *(w)* de l'algorithme 8, le processeur Microblaze effectue parallèlement au calcul de ces opérations une lecture du registre d'état *St_Registre2*. En effet, durant le chargement des données et lors de l'exécution des opérations arithmétiques, l'état de ce registre est *0x00000000*. Une fois le calcul parallèle des deux MMMs achevé et que les résultats $S_{(i)}$ et $C_{(i)}$ obtenus sont entièrement stockés dans les mémoires du module *2MulMong*, ce dernier transforme l'état du registre *St_Registre2* à *0x0000FFFF*. Microbalze exécute un décalage sur le $j^{ème}$ chiffre de *Z*, puis transmet l'instruction *"RunMMMs"*, afin d'exécuter les deux MMMs de la prochaine itération. Cette instruction est transmise en utilisant la fonction *2MMMs_CORE_mWriteSlaveReg0(baseaddr,0,RunMMMs)*.

Le processus du déroulement de l'exponentiation modulaire dans le domaine de Montgomery est identique pour l'ensemble des itérations de l'algorithme 8. Il sera complètement exécuté, à la suite de l'introduction de tous les bits de l'exposant Z. A la fin du processus itératif, le processeur transmet l'instruction *"RunMonOut"*, afin d'indiquer au Module *2MulMong* le début de la conversion du résultat $Y=(X^Z \times R\ mod$

N) vers la représentation classique des nombres. La fonction utilisée est: *2MMMs_CORE_mWriteSlaveReg0(baseaddr,0, RunMonOut).*

La dernière MMM calculée par le module *2MulMong* est définie par l'expression suivante :

$$Y = (X^Z \times R \times 1 \times R^{-1}) \; mod \; N = X^Z \; mod \; N$$

3. Transmission du résultat Y de l'exponentiation modulaire vers le processeur Microblaze

Le transfert de *Y* vers le processeur est basé sur l'état du bus *IP2RFIFO_Data*, assurant l'interconnexion entre la sortie du module *2MulMong* et le bus de données d'entrée de la mémoire *R_FIFO* (voir figure 4.9). En effet, durant la phase de transmission des données et lors de l'exécution de l'exponentiation modulaire, le module *2MulMong* force cette liaison à l'état haut. Autrement dit, la donnée transmise au processeur est *0xFFFFFFFF*. Une fois le résultat *Y* complémentent calculé, *2MulMong* initialise la mémoire *R_FIFO* par une donnée dont tous les bits sont nuls (*0x00000000*). Celle-ci sera suivie par les *n chiffres* qui constituent le codage de *Y* en base 2^k. La réception du $j^{ème}$ chiffre par le processeur Microblaze est effectuée en utilisant la fonction *Y[j]= MMM_CORE_mReadFromFIFO(baseaddr,0).*

Une fois le transfert de *Y* achevé, Microblaze transmet l'instruction *"ResetIP2MMMs"* pour remettre le composant *2MMMs_Core* à son état initial.

5.5 Troisième approche d'implémentation : *Exécution de l'exponentiation modulaire et de la MMM de manière logicielle*

L'objectif ciblé par l'implémentation de cette troisième approche est de constituer un modèle de comparaison et d'évaluation, nécessitant le moins de ressources matérielles possible [86]. L'architecture du système embarqué dans cette approche est implémentée sans aucun composant matériel. Les algorithmes d'exponentiation modulaire et de la MMM sont exécutés par le processeur Microblaze, en utilisant respectivement les fonctions *ExpBinary3(.)* et *Montgomery_SW(.)*.

5.5.1 Fonction d'exécution de l'exponentiation modulaire *"ExpBinary3(.)"*

Cette fonction correspond à l'implémentation logicielle de l'algorithme 8, utilisé pour calculer l'exponentiation modulaire $Y=X^Z \; mod \; N$. Elle est constituée par deux fonctions en l'occurrence *getBitIndex(.)* et *Montgomery_SW(.)* (voir figure 5.1). La première permet d'effectuer des décalages à droite sur l'exposant *Z*. La seconde

assure l'exécution itérative de la MMM de manière logicielle. L'organigramme de la fonction *ExpBinnary3(.)* est présenté sur la figure 5.8.

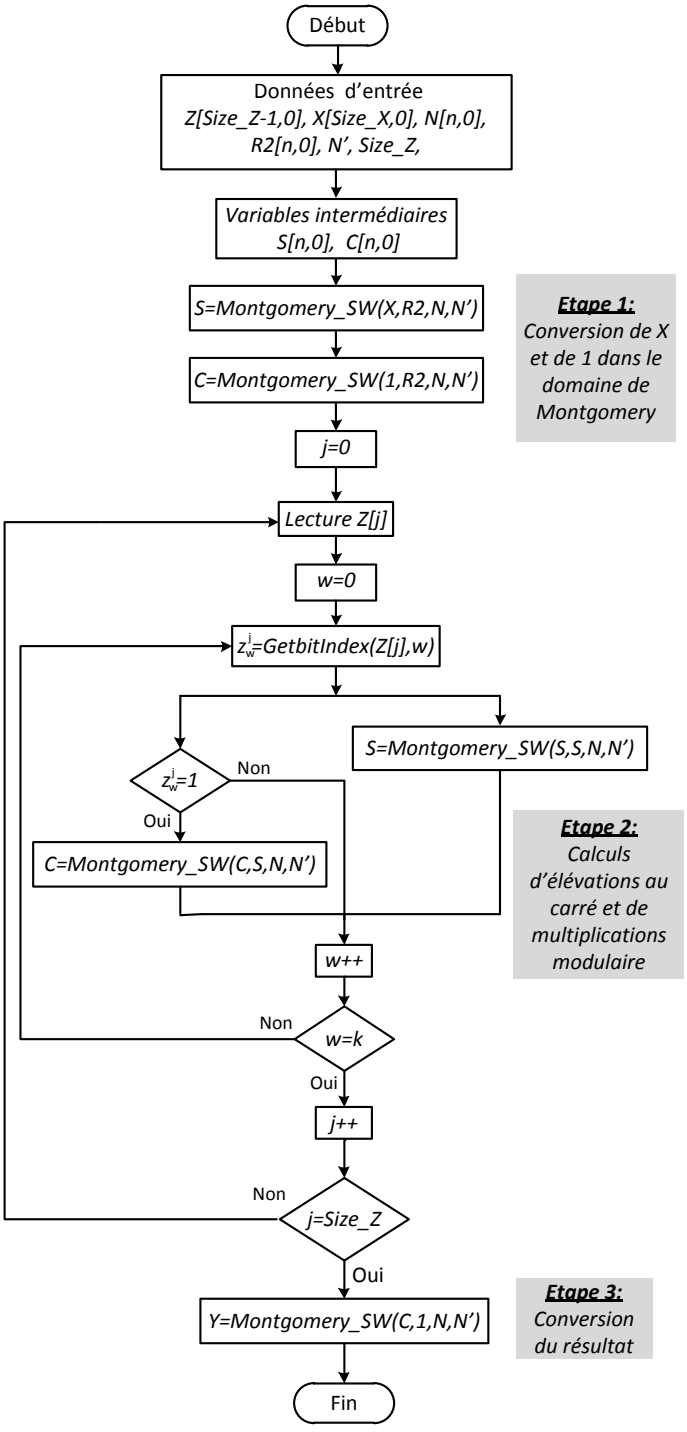


Figure 5.8. Organigramme de la fonction *"ExpBinary3(.)"*

Pour entamer le calcul de la fonction d'exponentiation modulaire, *ExpBinary3(.)* reçoit en entrée à partir de la mémoire *BRAM* du processeur l'ensemble des données transmises par l'Interface Homme/Machine (voir figure 4.1 du chapitre 4). A l'exception de la constante *N'* et de la taille *Size_Z* de l'exposant *Z*, représentées sur un

chiffre de taille *k-bits*, les autres données sont introduites selon leur représentation en base 2^k. Les résultats intermédiaires de l'algorithme 8 qui correspondent aux résultats de la multiplication modulaire $C_{(i)}$ et à l'élévation au carré $S_{(i)}$, sont stockés séparément dans deux variables. La fonction *ExpBinary3(.)* fournit en sortie le résultat *Y* de l'exponentiation modulaire selon son codage en base 2^k.

Les deux MMMs requises pour l'exécution de chaque itération *(w)* de l'algorithme 8, sont calculées séquentiellement par la fonction *Montgomery_SW(.)*. Celle-ci reçoit en entrée deux opérandes *(A, B)*, le modulo *N* et la constante *N'*.

Le résultat obtenu sera stocké dans la mémoire *BRAM*, afin qu'il soit utilisé comme opérande d'entrée dans la prochaine MMM. Rappelons que selon l'exécution de l'algorithme 8, les deux opérandes *A* et *B* peuvent être le message *X*, *"1"*, la valeur de $R^2 \bmod N$ ou les deux résultats intermédiaires $S_{(i)}$ et $C_{(i)}$.

5.5.2 Fonction d'exécution de la MMM de manière logicielle *"Montgomery_SW(.)"*

La fonction *Montgomery_SW(.)* correspond à la description en langage C de l'algorithme 7. Elle est utilisée par le processeur Microblaze pour calculer de manière logicielle une MMM. En effet, en analysant les opérations arithmétiques de l'algorithme 7, données par les expressions (4.1), (4.2), (4.3) et (4.4), on constate que ces dernières sont basées sur le calcul de multiplications, suivies par des additions. Pour ce faire, le calcul de ces opérations dans la fonction *Montgomery_SW(.)*, est assuré respectivement par deux fonctions nommées : *Multiply(.)* et *Add4OP(.)*.

a. Fonction de multiplication *"Multply(.)"*

Multiply(.) permet de calculer les multiplications des lignes 9 et 11 de l'algorithme 7, données respectivement par les expressions (4.1) et (4.3). Sa description en langage C est montrée sur la figure 5.9.

```
void multiply(unsigned long a, unsigned long b ,unsigned long *p,
              unsigned long *c)
   {
     unsigned long long pp=(unsigned long long) a*b;
        *p=(unsigned long) pp;
        *c=pp>>k;
   }
```

Figure 5.9. Code C de la fonction *"multiply(.)"*

b. Fonction d'addition *"Add4OP(.)"*

L'implémentation en langage C de la fonction *AddOP(.)* est montrée sur la figure 5.10. Son rôle consiste en l'exécution des additions des lignes 10 et 12 de l'algorithme 7, données respectivement par les équations (4.2) et (4.4).

```
void add4Op (unsigned long x, unsigned long  y, unsigned long z,
             unsigned long *resultat, unsigned long *cy)
   {
     unsigned long long r=(unsigned long long) x+y+z+(*cy);
     *resultat=(unsigned long) r;
     *cy=r>>k;
   }
```

Figure 5.10. Code C de la fonction *"Add4OP(.)"*

c. Exécution de la fonction *"Montgomery_SW(.)"*

La fonction *Montgomery_SW(.)* reçoit en entrée :

- deux opérandes *A* et *B*, où $A = \sum_{i=0}^{i=n} A[i] \times 2^k$ et $B = \sum_{j=0}^{j=n} B[j] \times 2^k$,
- le modulo *N*, avec $N = \sum_{j=0}^{j=n} N[j] \times 2^k$,
- la constate *N'* codée sur *k-bits*.

Elle fournit en sortie le résultat T de la MMM, où $T = \sum_{j=0}^{j=n} T[j] \times 2^k$. L'organigramme de cette fonction est montré sur la figure 5.11. Son exécution pour calculer une MMM est basée sur un certain nombre de variables intermédiaires. Ces dernières permettent de stocker les résultats obtenus des différentes opérations arithmétiques de l'algorithme 7. Les variables en question et leurs descriptions sont définies dans le tableau 5.3.

L'exécution de l'algorithme 7 via la fonction *Montgomery_SW(.),* repose sur l'utilisation de deux boucles imbriquées, définies par les indices *(i)* et *(j)*. Le premier est associé à la boucle externe. Il permet de parcourir les $i^{ème}$ chiffres de l'opérande *A*. Le second est utilisé pour positionner les $j^{ème}$ chiffres de *B*, de *N* et de $T_{(i)}$ dans les opérations arithmétiques de la boucle interne. A noter que ces deux indices varient de *0* à *n*.

Au début de chaque itération *(i)*, on procède par :

- L'initialisation des variables *Retenue1* et *Retenue2*, associées respectivement aux retenues des additions, définies par les expressions (4.2) et (4.4) du chapitre 4.

- L'initialisation des variables intermédiaires *C1* et *C2*. Ces dernières correspondent respectivement aux chiffres les plus significatifs des résultats issus des multiplications $A[i] \times B[j]$ et $q_{(i)} \times N[j]$, définies respectivement par les équations (4.1) et (4.3).
- La sélection du $i^{ème}$ chiffre de l'opérande *A*.
- L'initialisation de l'indice *(j).*

Une fois l'étape d'initialisation accomplie, l'exécution des opérations arithmétiques est entamée par le calcul de $q_{(i)}$:

$$q_{(i)} = Z0 \times N' mod\ 2^k, \text{ où } Z0 = P0 + T[0]_{(i)} \text{ et } P0 = A[i] \times B[0]$$

Ensuite, en incrémentant l'indice *(j)* de *0* à *n,* les fonctions *Multiply(.)* et *Add4OP(.)* sont exécutées séquentiellement dans la boucle interne de l'organigramme, dans le but de calculer les $j^{ème}$ chiffres des deux variables $Z_{(i)}$ et $L_{(i)}$.

Le premier test de l'organigramme est utilisé afin de vérifier la valeur du chiffre le moins significatif $L[0]_{(i)}$ de $L_{(i)}$. En effet, sachant que le résultat intermédiaire $T_{(i+1)}$ est obtenu par la division $T_{(i+1)}=L_{(i)} /2^k$. Ce qui correspond à un simple décalage à droite de la variable $L_{(i)}$ de *k-bits* (voir figure (2.6) du chapitre 2). De plus, on sait que le chiffre le moins significatif $L[0]_{(i)}$ est souvent nul puisque l'opération $L_{(i)} /2^k$ est une division exacte. Ainsi, ce premier test permet simplement d'éliminer $L[0]_{(i)}$ et de considérer uniquement les *n chiffres* suivants.

Le second test porte sur la valeur de l'indice *(j)* et permet de prendre en considération la valeur de *C1* et des retenues obtenues à l'itération *(j=n).* Ces dernières sont utilisées pour compléter le calcul de $T_{(i+1)}$, où son dernier chiffre $T[n]_{(i+1)}$ est obtenu par la somme des retenues et de *C1* , issus de l'itération *j=(n)* (voir les lignes 15, 16 et 17 de l'algorithme 7).

Une fois avoir calculé le dernier chiffre du résultat intermédiaire $T_{(i+1)}$, l'indice *(j)* est initialisé à zéro *(j=0)* ; l'indice *(i)* est incrémenté *(i=i+1).* L'exécution de l'algorithme 7 par le processeur Microblaze se répète de la même manière, sur toutes les itérations. Il sera complétement exécuté, lorsque le chiffre le plus significatif *A[n]* de l'opérande *A* sera introduit dans les calculs. Ce qui correspond à la dernière itération *(i=n).* Le résultat alors obtenu, en l'occurrence $T_{(n+1)}$ représente le résultat *T* de la MMM. La vérification du nombre d'itérations *(i)* écoulées est assurée par l'utilisation du troisième test de l'organigramme.

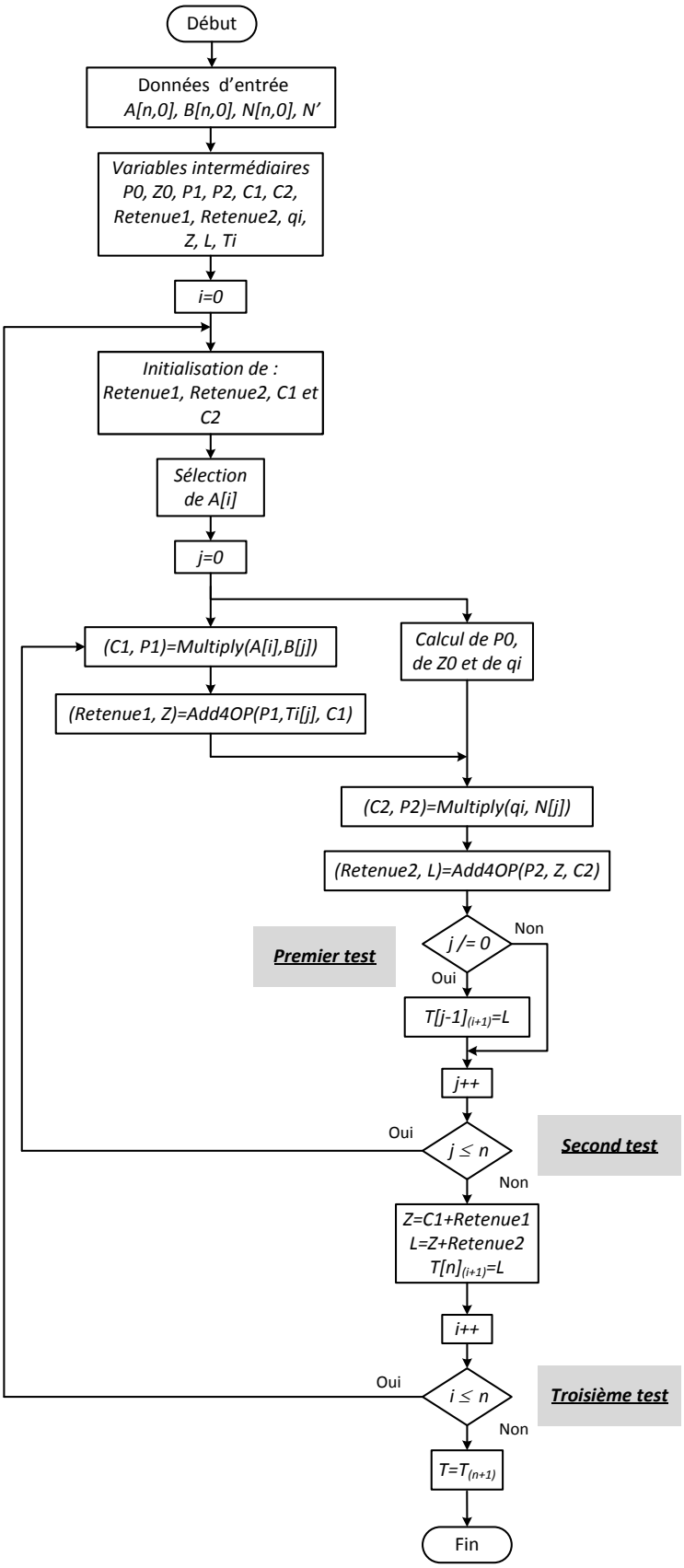


Figure 5.11. Organigramme de la fonction *"Montgomery_SW(.)"*

Tableau 5.3. Variables intermédiaires de la fonction *Montgomery_SW(.)*

Variable	Déscription
P0	Résultat de la multiplication de la ligne 3 de l'algorithme 7.
Z0	Résultat de l'addition de la ligne 4 de l'algorithme 7.
P1, C1	Chiffres le mois et le plus significatif de la multiplication (4.1).
P2, C2	Chiffres le mois et le plus significatif de la multiplication (4.3).
Retenue1	Résultat de la somme (cy_1+cy_2) de la ligne 10. cy_1 et cy_2 représentent les retenues des additions de l'équation (4.2).
Retenue2	Résultat de la somme (cy_3+cy_4) de la ligne 12. cy_3 et cy_4 représentent les retenues des additions de l'équation (4.4).
qi	Résultat de la ligne 5 de l'algorithme 7.
Z	$j^{ème}$ *chiffre* de la variable intermédiaire $Z_{(i)}$.
L	$j^{ème}$ *chiffre* de la variable intermédiaire $L_{(i)}$
Ti	Résultat intermédiaire $T_{(i)}$ de l'algorithme 7.

5.6 Mise au point de l'Interface Homme/Machine (*IHM)*

Cette interface a été développée à l'origine non seulement, pour vérifier le fonctionnement correct de la partie embarquée sur circuit FPGA ; mais aussi, pour permettre une configuration flexible de la plateforme de chiffrement et de déchiffrement proposée. Sa conception a été réalisée en langage *Java,* en utilisant l'outil *Eclipse*. Le menu principal de l'*IHM* est montré sur la figure 5.12. Cette interface permet :

1. de configurer le protocole de communication RS232,
2. de sélectionner le type d'architecture utilisé pour l'implémentation de l'exponentiation modulaire,
3. de chiffrer et de déchiffrer un message,
4. de chiffrer et de déchiffrer des données, stockées dans le disque dur de l'ordinateur sous forme de fichiers,
5. de générer la clé publique et la clé privée ; définies respectivement dans le chapitre 1 par *(e, N)* et *(d, N)*,
6. de calculer la constante N' de la MMM et la valeur de $R^2 mod\ N$,
7. de choisir la base β de représentation des données.

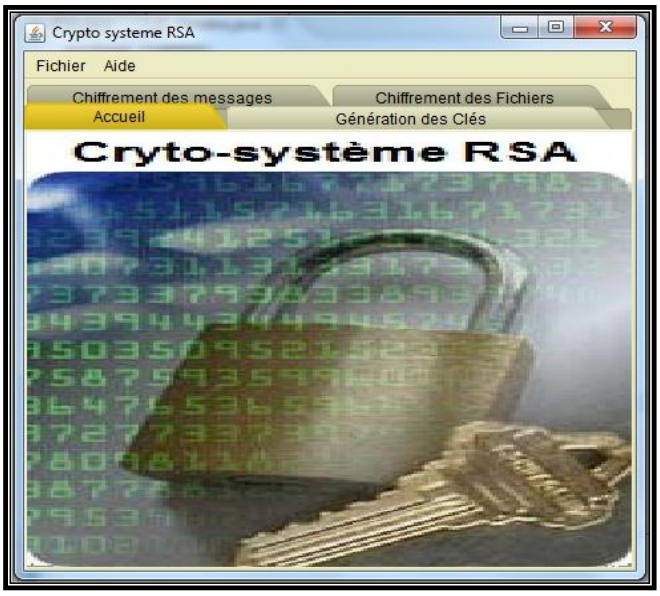


Figure 5.12. Menu principal de l'IHM

5.6.1 Configuration du protocole de communication RS232

Cette partie permet de configurer les paramètres du protocole RS232, utilisé comme liaison de communication entre l'ordinateur et la carte de prototypage Genesys. Les paramètres en question concernent : le débit de transmission des données, leurs tailles, le bit de parité et le bit stop. A noter que ces derniers doivent être définis avec les mêmes paramètres, utilisés lors de la configuration de l'*UART* du système embarqué. La configuration du protocole RS232 est montrée sur la figure 5.13.

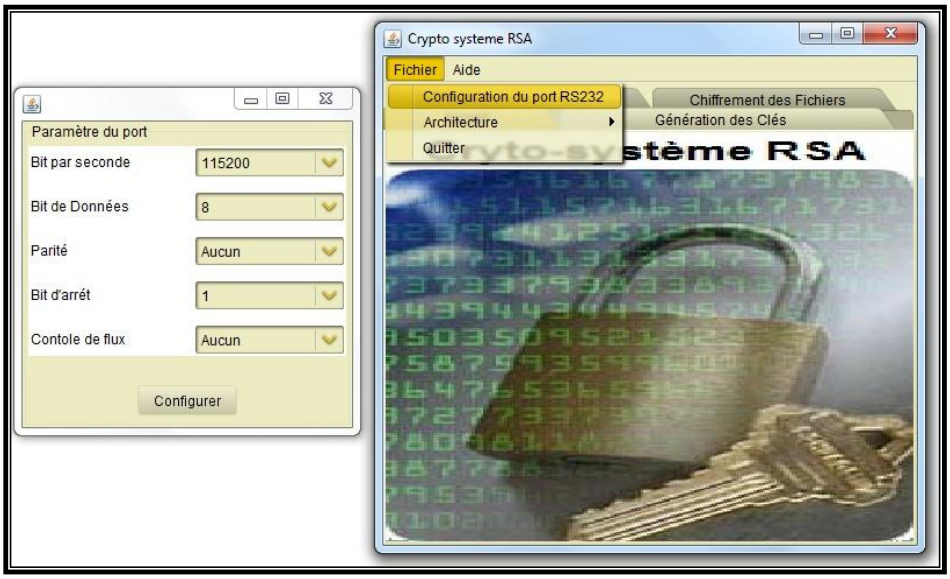


Figure 5.13. Configuration du protocole RS232

5.6.2 Sélection du type d'architecture du système embarqué

Cette partie permet de configurer l'approche utilisée pour l'implémentation de l'exponentiation modulaire sur circuit FPGA. La sélection est portée sur une implémentation purement logicielle ou une implémentation basée sur la combinaison logicielle/matérielle. La configuration du type d'architecture du système embarqué est montrée sur la figure 5.14.

Figure 5.14. Configuration du type d'architecture.

5.6.3 Chiffrement et déchiffrement d'un message

Cette partie est utilisée pour le chiffrement ou le déchiffrement d'un message de taille variable. Le résultat obtenu est affiché sur écran. La fenêtre qui correspond à cette partie de l'*IHM* est montrée sur la figure 5.15. Elle est constituée de:

- Trois champs. Le premier, noté par *"Message à chiffrer"*, est utilisé pour écrire un message en clair. Le second désigné par *"Message à déchiffrer"* permet de visualiser le résultat du chiffrement. Le troisième est associé à l'affichage du message déchiffré.
- Un tableau permettant d'indiquer le nombre de blocs obtenu de la décomposition du message *X* en une séquence de sous messages, dont les valeurs numérique sont inférieures à la valeur du modulo *N*. Ce tableau nous permet aussi d'afficher le nombre de chiffres qui représentent le codage du message *X* en base 2^k avec $k = 16$ ou 32
- Deux champs pour afficher l'emplacement des clés dans le disque dur de l'ordinateur.

- Quatre boutons notés par : *"Importer la clé publique"*, *" Importer la clé privée"*, *"Chiffrer"* et *"Déchiffrer"*. Ces derniers nous permettent d'activer les actions à effectuer.
- Une partie pour l'affichage des performances temporelles *(temps de transmission, temps d'exécution de chiffrement/déchiffrement, temps de réception).*

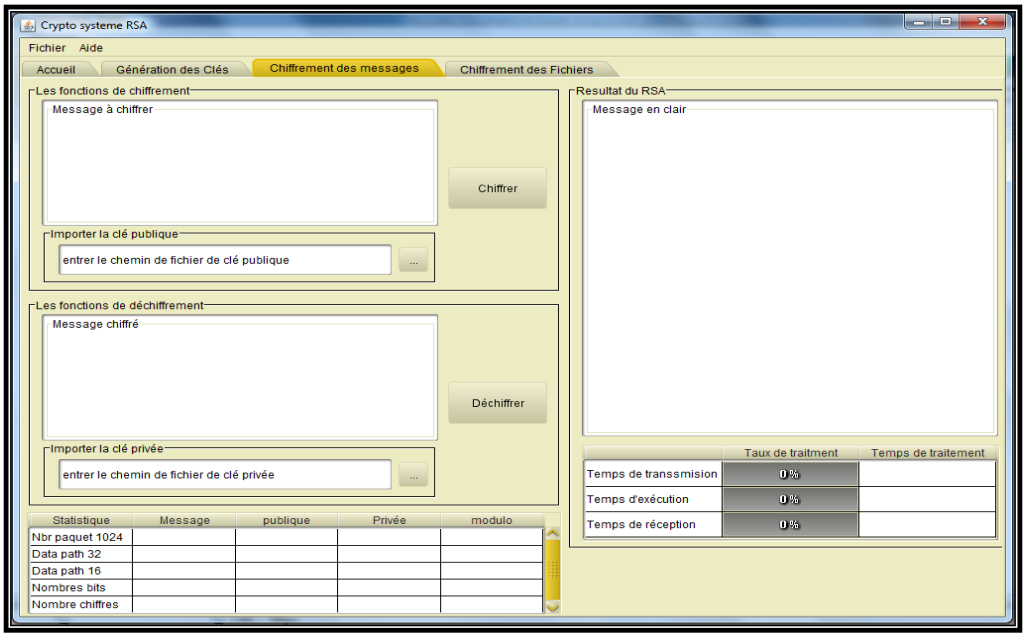


Figure 5.15. Chiffrement et déchiffrement d'un message

5.6.4 Chiffrement et déchiffrement d'un fichier

La fenêtre associée à cette partie de l'*IHM*, est montrée sur la figure 5.16. Elle permet d'activer le chiffrement et le déchiffrement d'un fichier. 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 l'emplacement dans le disque dur, 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 chiffrer et le chemin du stockage du résultat de chiffrement (fichier chiffré). Le quatrième bouton *"Chiffrer"* permet de transmettre le fichier en clair au circuit FPGA pour le chiffrement.
- 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.

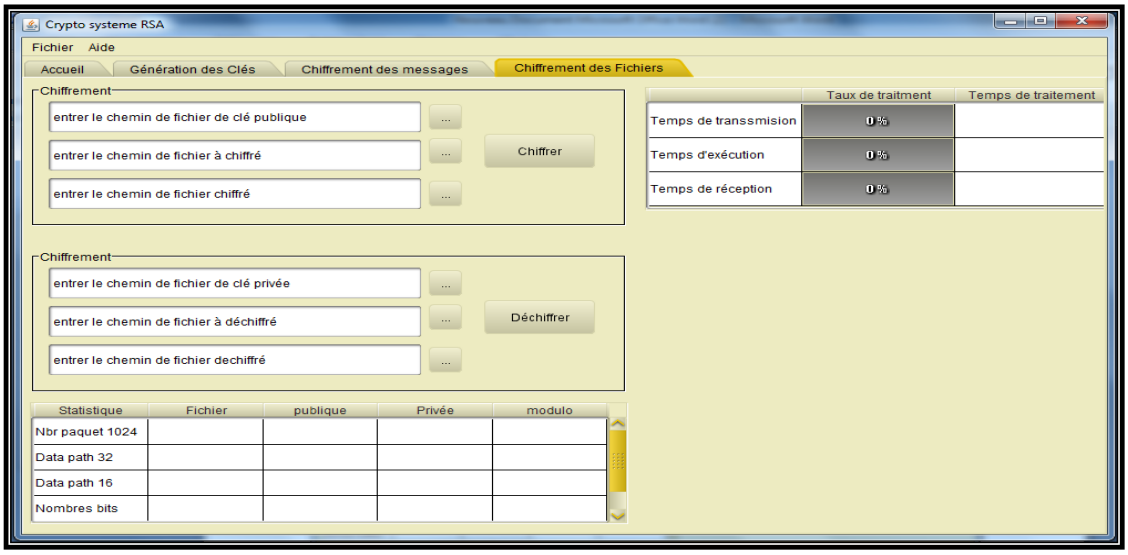


Figure 5.16. Chiffrement et déchiffrement d'un fichier.

5.6.5 Génération des clés et calcul des constantes de la MMM

Cette partie permet :

1. de générer la clé publique *(e, N)* et la clé privée *(d, N)*.
2. de calculer les deux constantes *N'* et $R^2 mod\ N$,
3. de choisir la base β, utilisée pour le codage des données.

Les valeurs décimales des clés obtenues sont stockées séparément dans deux fichiers textes. Ces derniers contiennent aussi les valeurs des deux constantes *N'* et $R^2\ mod\ N$. La partie de l'*IHM* permettant de réaliser ces trois opérations est montrée sur la figure 5.17.

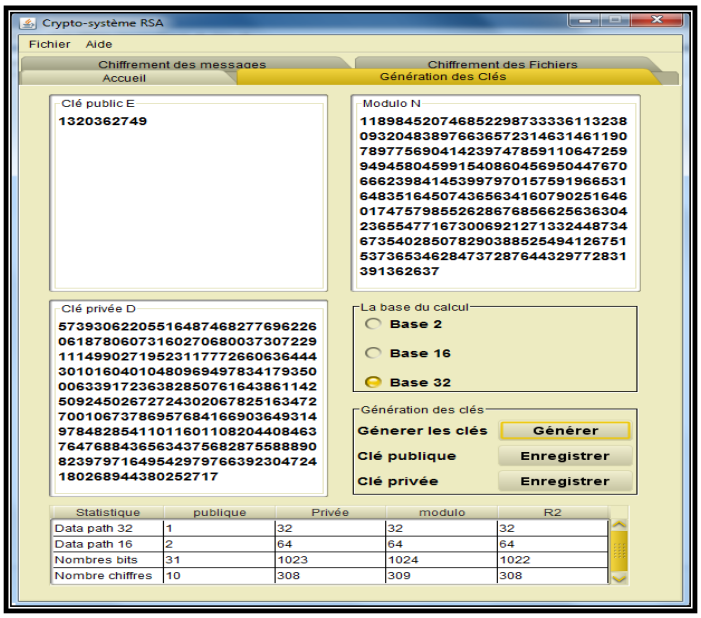


Figure 5.17. Génération des clés et calculs des constantes da la MMM

Le programme Java de cette partie est basé sur la classe *"BigIntegerPro"*, disponible dans la bibliothèque d'*Eclipse*. Cette classe contient la définition d'un nombre de type *BigInteger* et toutes les opérations arithmétiques 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. Le programme Java de cette partie de l'*IHM* est montré sur la figure 5.18.

```
1.public void generateKeys()
2.{this.GenerteKey.addActionListener(new ActionListener()
3.  {@Override
4.   public void actionPerformed(ActionEvent arg0)
5.   {isGenerateKey=false;
6.    while(!isGenerateKey)
7.       {try
8.         {boolean b=false;
9.           while(!b)
10.{p=new BigIntegerPro(BigInteger.probablePrime(512,new
11. Random()).toString());
12. if(p.bitLength()==512)
13.  {q=new BigIntegerPro(p.nextProbablePrime().toString());
14.   if(q.bitLength()==512)
15.   {while(!b)
16.    {N=new BigIntegerPro(p.multiply(q).toString());
17.     if(N.bitLength()==1024)b=true;
18.    }
19.   }
20  }
21.phi=new BigIntegerPro(q.subtract(newBigIntegerPro("1")).multiply
22.                     (p.subtract new BigIntegerPro("1"))).toString());
23.                      b=false;
24.                      while(!b)
25.{
26. E=new BigIntegerPro(32,new Random());
27. if(E.gcd(phi).compareTo(new BigIntegerPro("1"))==0 && E.compareTo
28. (new BigIntegerPro("2"))>0 && E.compareTo(phi)<0)
29. b=true;
30.}
31. D=new BigIntegerPro(E.modInverse(phi).toString());
32.If
33. (base16.isSelected())
34. {R2= new BigIntegerPro(new
35.  BigInteger("2").pow(2080).mod(N).toString());
36.  Nc=new BigIntegerPro(new BigInteger("-"+N.toString()).modInverse
37. (new BigInteger("65536")).toString());base=16;}
38.  else
39.  {R2= new BigIntegerPro(new
40.   BigInteger("2").pow(2112).mod(N).toString());
41.   Nc=new BigIntegerPro(new BigInteger("-"+N.toString()).modInverse(new
42.   BigInteger("4294967296")).toString());base=32;}
```

Figure 5.18. Programme Java de génération des clés et de calcul des constantes N' et $R^2 mod\ N$

Les clés publique et privée sont calculées selon le protocole du crypto système RSA, défini dans le chapitre 1. La taille de ces clés est de *1024-bits*.

a) Génération des clés

Les étapes de génération des deux clés: *(e, N)* et de *(d, N)*, sont définies comme suit :

1. Génération aléatoire de deux nombres premiers, p et q en utilisant les fonctions *probablePrime* et *nextProbablePrime*.
2. Calcul du modulo *N* par le produit $p \times q$ de la ligne 16 du programme.
3. Calcul de la fonction d'Euler $\Phi(N)$ en utilisant la syntaxe :

```
phi=newBigIntegerPro(q.subtract(newBigIntegerPro("1")).
multiply (p.subtract new BigIntegerPro("1"))).toString())
```

4. Calcul des exposants *e* et *d*.

   e est calculé sur *32-bits* par les lignes 26, 27 et 28, tel que :

$$\begin{cases} 2 < e < \Phi(N) \\ \text{et} \\ e \text{ est premier avec } \Phi(N) \end{cases}$$

   L'exposant *d* est calculé par la ligne 31, en utilisant la syntaxe :

```
D=new BigIntegerPro(E.modInverse(phi).toString())
```

b) Calcul de *N'* et de $R^2 \bmod N$

Ces deux constantes sont calculées suivant les conditions d'exécution de la MMM. Leurs valeurs numériques sont calculées en fonction du facteur *R* et du modulo *N*, telles que :

- $N' = -N_0^{-1} \bmod \beta$, N_0 est le chiffre le moins significatif du modulo *N* en base $\beta = 2^k$.
- $R = 2^{k \times (n+1)}$ avec $k \geq 2$.

N' et *R* sont définies respectivement dans les paragraphes 2.5.1 et 2.5.2 du chapitre 2.

Comme les valeurs de *N'* et de *R* sont calculées en fonction de la base β ; dans ce travail, nous avons utilisé les deux bases $\beta = 2^{16}$ et $\beta = 2^{32}$ afin d'évaluer les performances d'exécution de notre système embarqué.

1. Calcul de N' et de $R^2 \bmod N$ en base 2^{16}

N' est calculée par les lignes 36 et 37. La valeur de $R^2 \bmod N$ est obtenue par les lignes 34 et 35.

2. Calcul de N' et de $R^2 mod\ N$ en base 2^{32}

N' est obtenue en utilisant les lignes 41 et 42. La valeur de $R^2 mod\ N$ est calculée par les lignes 39 et 40.

5.7 Conclusion

Dans ce chapitre, nous avons abordé les parties logicielles des trois approches d'implémentations, proposées pour la mise en œuvre de l'exponentiation modulaire dans un environnement PSoC. Pour compléter la réalisation des trois niveaux d'abstractions du crypto système RSA, une interface homme machine a été également présentée. Cette interface qui est exécutée sur un ordinateur indépendamment du support d'implémentation de l'exponentiation modulaire, comporte un certain nombre d'options qui permettent la configuration complète du crypto système.

Le chapitre suivant sera consacré à la présentation des résultats d'implémentations et à des discussions. Ces dernières concernent l'étude des performances d'exécution de la MMM et de l'exponentiation modulaire, en fonction de la base de représentation des données et des approches d'implémentations proposées.

Chapitre 6

Simulations et résultats d'implémentations

6.1. Introduction

Les trois approches proposées dans ce travail pour l'implémentation de la fonction d'exponentiation modulaire dans un environnement PSoC, ont été conçues afin de réaliser un crypto système RSA, présentant le meilleur compromis entre :

- le temps d'exécution,
- les ressources matérielles occupées par la partie implémentée sur circuit FPGA.
- la flexibilité du système.

Pour atteindre cet objectif, la réalisation du crypto système qui correspond à chacune des trois approches, a été élaborée en trois étapes. Ces dernières sont définies comme suit :

1. Conception et implémentation des deux parties logicielles et matérielles de chaque approche.
2. Génération du fichier de configuration du circuit FPGA.
3. Vérification du fonctionnement du crypto système, en utilisant la carte de prototypage Genesys et l'*IHM* présentée dans le chapitre 5.

La conception de la partie matérielle de chacune des trois approches, a été réalisée en utilisant l'outil ISE 13.2 de Xilinx, suivant la méthodologie présentée dans le chapitre 3. Dans le but de vérifier le fonctionnement correcte de l'*UA* et des deux approches basées sur la combinaison logicielle/matérielle, des simulations fonctionnelles et des comparaisons avec des modèles mathématiques ont été effectuées, en utilisant respectivement les outils : Modelsim SE 6.4 1.0 0C [81] et Maple 9.5 [80].

Ce chapitre est consacré à la présentation des résultats d'implémentations des méthodes proposées pour la réalisation de la MMM et de la fonction d'exponentiation modulaire. Il est organisé en trois parties. La première décrit la méthodologie élaborée pour vérifier le fonctionnement des deux composants *MMM_Core* et *2MMMs_Core*. Cette vérification est basée principalement sur des simulations fonctionnelles. La seconde partie est consacrée à la présentation des performances d'exécution de l'*UA* et des trois approches proposées pour l'implémentation de l'exponentiation modulaire. La troisième partie est réservée à des discussions et comparaisons, par rapport aux performances atteintes par quelques-unes des implémentations citées dans l'état de l'art.

6.2. Méthode de vérification des deux composants *MMM_Core* et *2MMMs_Core*

Dans le cycle de conception des deux composants *MMM_Core* et *2MMMs_Core*, la simulation fonctionnelle de leurs architectures dans l'environnement *PSoC*, représente la phase la plus importante et la plus longue. En effet, cette étape nous permet non seulement de vérifier le fonctionnement correct des deux composants mais aussi, de concevoir leurs parties logicielles.

Dans ce travail, la méthode suivie pour concevoir et vérifier le fonctionnement des deux composants, est montré sur la figure 6.1. Celle-ci est constituée de cinq étapes. Ces dernières sont définies comme suit :

1. Spécification de la partie matérielle du *PSoC*.
2. Génération des vecteurs de test et réalisations des modèles mathématiques, pour la vérification des résultats.
3. Implémentation des parties logicielles des deux composants.
4. Simulations fonctionnelles.
5. Vérifications des résultats.

6.2.1. Spécification de la partie matérielle du *PSoC*

Cette étape est réalisée dans l'interface graphique XPS d'EDK. Elle consiste en :

- La configuration du processeur Microblaze et ses périphériques de base.
- Description de l'architecture matérielle des composants *MMM_Core* et *2MMMs_Core*.

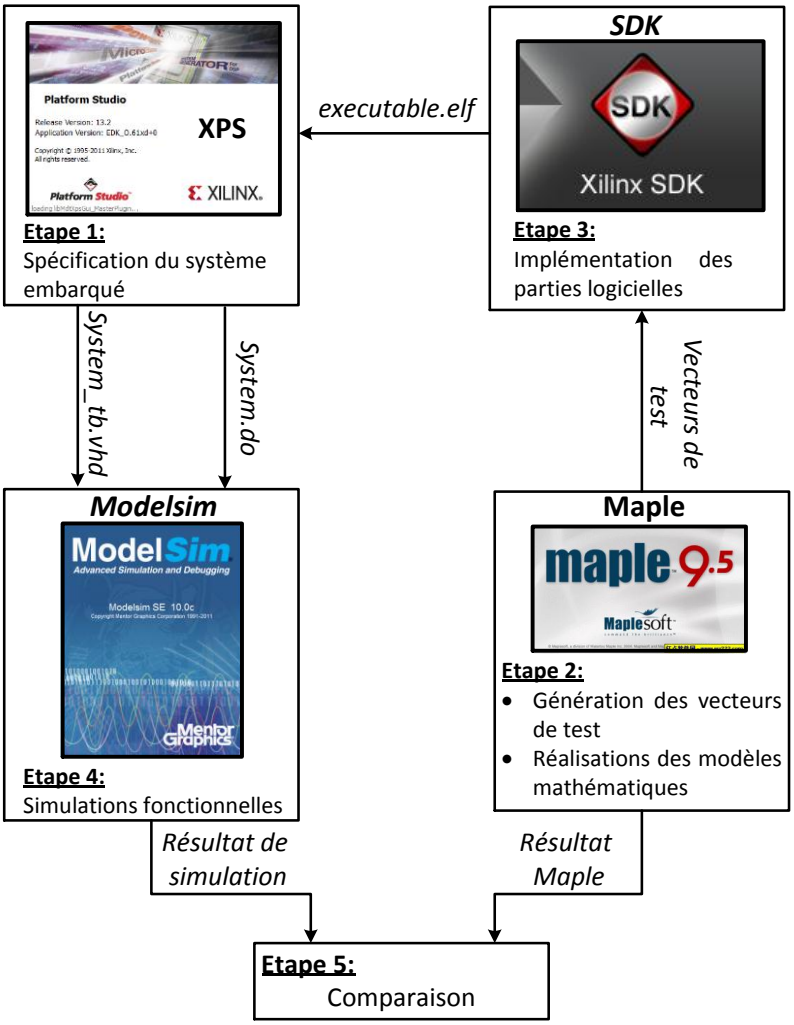


Figure 6.1. Méthode de vérification des deux composants *MMM_Core* et *2MMMs_Core*

Avant d'entamer la conception des deux composants, nous avons procédé en premier lieu dans l'outil ISE par la réalisation de l'*UA* (voir figure 4.3). Dans le but de concevoir une unité indépendante de la taille des opérandes et de la base β où $\beta=2^k$, l'*UA* a été réalisée en se basant sur une description VHDL paramétrable, par rapport à la taille des opérandes et par rapport au paramètre *k*. Les multiplieurs *Mul1 Mul2* et *Mul3* utilisés dans cette unité pour calculer les multiplications de l'algorithme de la MMM (algorithme 7), ont été générés en utilisant l'outil CoreGenerator d'ISE. Ces multiplieurs sont basés sur les blocs *DSP48E* de Xilinx [31]. Les additionneurs *Add1*, *Add2*, *Add3* et *Add4* utilisés pour calculer les additions de l'algorithme 7, sont des additionneurs à propagation de retenue. Leurs implémentations reposent sur une description RTL (Register Transfert Level). Autrement dit, nous avons en un premier temps réalisé la cellule élémentaire qui n'est autre qu'un additionneur complet (Full

Adder). Puis, dans un second temps, les additionneurs de l'*UA* ont été générés suivant la taille du paramètre *k*.

Une fois avoir achevé la conception et la vérification du fonctionnement de l'*UA*, les architectures internes de *MMM_Core* et *2MMMs_Core* ont été complétées par :

- La configuration des interfaces IPIF qui leurs sont associées.
- La conception des deux modules *MulMong* et *2MulMong*.

Ces modules ont été implémentés en langage VHDL. Les blocs mémoires utilisés, ont été générés en utilisant l'outil CoreGenerator.

A la fin de l'étape de spécification de la partie matérielle du *PSoC*, deux fichiers nommés ***system.do*** et ***system_tb.vhd*** sont générés en utilisant l'outil *Generate Simulation Hdl Files* de l'interface graphique XPS. Ces fichiers seront utilisés par la suite par l'outil Modelsim pour effectuer la simulation fonctionnelle. Les fichiers en question sont définis comme suit :

system.do : est constitué par des scripts. Ces deniers permettent d'indiquer à l'outil Modelsim, l'emplacement dans le disque dur des fichiers VHDL, correspondant aux composants intégrés dans le circuit FPGA.

system_tb.vhd : ce fichier est le *test bench* utilisé pour la simulation fonctionnelle des deux composants *MMM_Core* et *2MMMs_Core*. Il comporte les descriptions VHDL de l'horloge et du signal d'initialisation système.

6.2.2. Génération des vecteurs de test et réalisation des modèles mathématiques

Dans le but de vérifier le fonctionnement correct des composants *MMM_Core* et *2MMMs_Core*, nous avons utilisé l'outil Maple pour :

- La génération et la représentation en base 2^k des vecteurs de test.
- Les réalisations des modèles mathématiques pour la vérification des résultats issus des deux composants.

Les vecteurs de test générés seront considérés comme étant les données d'entrées des deux composants, lors de leurs simulations fonctionnelles. A noter que chacun de ces composants possède ses propres vecteurs de test et son propre modèle mathématique. Ceci est justifié par le fait qu'elles sont implémentées selon deux concepts différents.

A l'origine, les vecteurs de test issus de l'étape de génération sont codés en base 10. Cependant, comme les approches proposées dans ce travail pour l'implémentation de la MMM et de l'exponentiation modulaire, exigent des données représentées en base 2^k ; de ce fait, un programme de conversion a été implémenté sous Maple. Ce programme permet de générer les $j^{ème}$ chiffres qui correspondent aux codages des données en base 2^k, à partir de leur représentation décimale. Le code source du programme en question est montré sur la figure 6.2.

```
k:=32;    ## Valeur du paramètre k qui définit la base utilisée
n:=32;    ## Nombre de chiffres qui correspond au codage d'une donnée
          ## de taille 1024 bits en base 2^32

Data[-1]:=Data:## initialisation par la valeur décimale du vecteur du test
for j from 0 to n do
    Data[j]:= trunc(evalf(Data[j-1]/(2^k))): ## calcul du quotient de
                                             ## la j^ème division par 2^k
    Dout[j]:= Data[j-1] mod (2^k):           ## calcul du j^ème chiffre
    Dout_hex[j]:=convert(Dout [j],hex) : ## conversion du j^ème chiffre
                                         ## en hexadécimale
end do:
```

Figure 6.2. Programme Maple pour le codage des vecteurs de test en base 2^k

Pour obtenir le $j^{ème}$ chiffre d'un vecteur de test, l'implémentation du programme de conversion est basée sur le calcul du reste qui correspond à la division par 2^k, du quotient obtenu à l'itération *(j-1)*

6.2.3. Implémentation des parties logicielles des deux composants

Dans cette étape, il est question de réaliser les pilotes logiciels de bas niveaux des deux composants. Ces pilotes sont décrits en langage C dans l'outil SDK.

Pour accomplir la simulation fonctionnelle de chaque composant, un fichier nommé ***TestApp_Peripheral.c*** a été développé. Ce dernier est constitué principalement par trois parties. La première est consacrée à la définition des instructions utilisées par le processeur pour contrôler le fonctionnement des deux composants. Ces instructions sont définies dans les tableaux 4.1 et 4.2 du chapitre 4. La seconde partie est réservée à l'insertion des vecteurs de test (données d'entrées), générés par l'outil Maple. La troisième partie est une procédure, utilisée pour :

- Le transfert des données et des d'instructions vers l'architecture interne des deux modules *MulMong* et *2MulMong*, implémentés respectivement dans *MMM_Core* et *2MMMs_Core*.

- Réception et transfert des résultats vers le processeur Microbalze.

Toutes les données qui sont mises en exécution lors des simulations fonctionnelles sont représentées en base $\beta=2^k$ sur *(n+1) chiffres*. Chaque chiffre est codé en hexadécimal.

Une fois la description des pilotes achevée, ces derniers sont compilés dans l'outil SDK. Le résultat obtenu de la compilation, en l'occurrence ***executable.elf*** correspond à l'exécutable de la partie logicielle du *PSoC*. Pour tenir compte des vecteurs de test et des instructions lors de la simulation fonctionnelle du système, l'exécutable issu de la compilation est stocké dans la mémoire BRAM du processeur.

6.2.4. Simulations fonctionnelles des deux composants *MMM_Core* et *2MMMs_Core*

Pour parvenir aux simulations fonctionnelles de *MMM_Core* et *2MMMs_Core*, l'outil de simulation Modelsim, fait appel aux deux fichiers ***system.do*** et ***system_tb.vhd***. Les résultats de simulation figurent sous forme de chronogrammes dans un éditeur de courbes logiques.

6.2.5. Vérification des résultats par des modèles mathématiques

Cette étape consiste en la vérification des résultats obtenus de la simulation fonctionnelle, en utilisant des modèles mathématiques. Ces derniers correspondent :

- au calcul de la MMM, définie par l'expression 2.10 du chapitre 2,
- au calcul de l'exponentiation modulaire $Y=X^Z \bmod N$.

Une fois avoir vérifié l'exactitude des résultats, les deux composants ont été implémenté sur circuit FPGA. Puis, testés sur la carte de prototypage Genesys de Digilent. Pour vérifier les résultats *T* et *Y* calculés respectivement par *MMM_Core* et *2MMMs_Cores,* nous avons introduit après les étapes de simulations la syntaxe « *xil_printf("0x%08x \n\r", Data[j]);* » dans leurs pilotes de bas niveaux. *Data[j]* représente le $j^{ème}$ chiffre des résultats *T* ou *Y*. La syntaxe en question permet de visualiser sur l'*HyperTerminal* de Windows l'un des deux résultats.

6.2.6. Simulation fonctionnelle de *MMM_Core*

La vérification du fonctionnement de *MMM_Core,* proposée pour le calcul itératif de la MMM dans la première approche d'implémentation, est basée sur :

- la génération des vecteurs de test pour une sécurité de taille *1024-bits*.

- la réalisation d'un modèle mathématique pour le calcul de la MMM dans le domaine de Montgomery.
- des simulations en bases 2^{16} et 2^{32}
- la vérification du résultat T de la MMM.

Dans la première approche d'implémentation, les vecteurs de test générés correspondent aux données d'entrées de *MMM_Core*. Ces vecteurs sont constitués de deux opérandes A et B, du modulo N et de la constante N'. A noter que toutes les données sont introduites suivant leur codage en base $\beta = 2^k$.

Les valeurs décimales des vecteurs de test utilisées pour la simulation fonctionnée de *MMM_Core*, sont définies telles que :

A=*64744260923816997073497753884140022001852723054654782686020604753687813169956413646710959052687701234801305539214131332484235850772464941349308363613886431772470995127467158472570252837990294137535968316794030106235444245620783180619478879378237364538602660000474067090768632900227270269392906530119885044 9.*

B=*10358103425380659758203608768996505038273686655108210383997428608247516059723175284817387986537780566823662556171951986170795330293488228991475526765862173010922385260880466976102093029203895618733667554521541018155055100230381863927973690493710485532261865869666499352507718830459521374284434363645471402625 3.*

N=*10705317519828414940001004525660271111869696144572758009201383632884537254996320423715189619053624467254497583207757181798452018559187619411005086485452244221872536277047659693224323514711135439739982691495816749215072168281273374931367770181800021186722938945822246117084121286223270985594591555710851079837 9.*

Les valeurs de N' en bases 2^{16} et 2^{32} sont respectivement *33661* et *1391035261*.
La valeur décimale du résultat T obtenu de la simulation est :

T=*23785195022492532699068334475452195055301772564206145187884850954594308543167892477051442160549794073196004150997829947626708938430491729689928461692278561723266813330427778596358526963123366679505632 2*

52110037789256906633152884398648796712127178649773879247528542557696323477202327399595728149654801576745112.

La vérification du comportement de *MMM_Core* pour calculer une MMM, a été effectuée en se basant sur des simulations effectuées selon les trois étapes:

1. chargement des données dans les unités de stockage du module *MulMong*.
2. exécution des opérations arithmétiques de l'algorithme de la MMM (algorithme 7)
3. transfert du résultat *T* vers le processeur Microblaze.

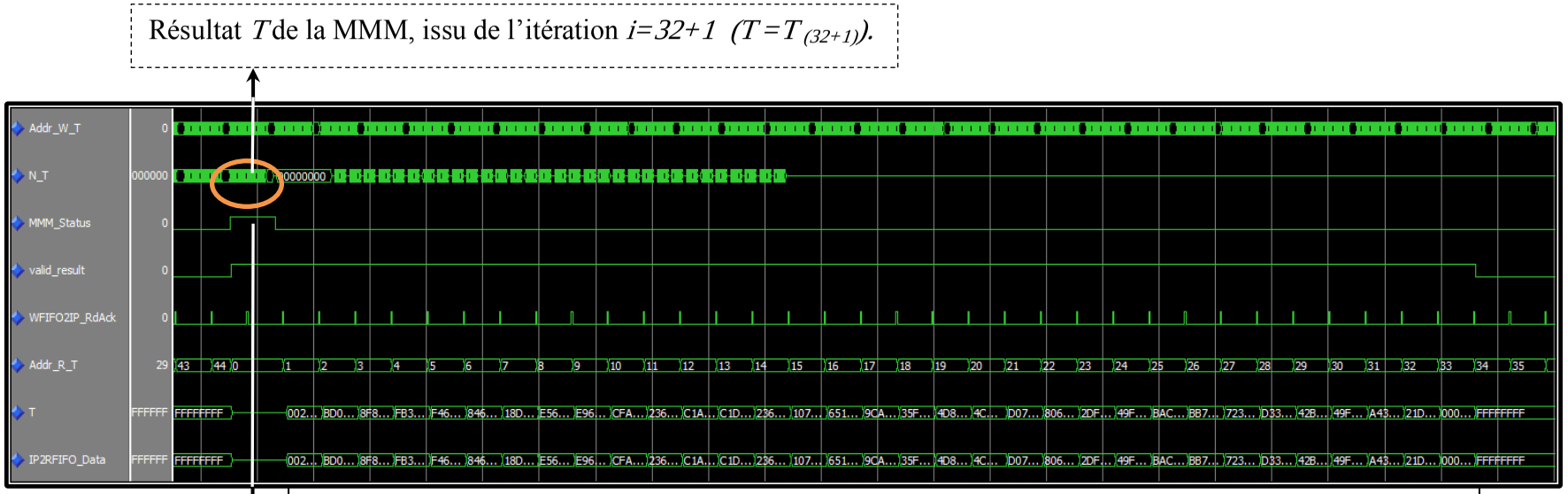
Ces trois étapes sont détaillées dans le paragraphe 4.3.3.3 du chapitre 4.

Afin de vérifier le résultat *T* fournie par *MMM_Core*, un calcul similaire a été effectué sur Maple par l'utilisation d'un modèle mathématique. Ce dernier n'est rien d'autre que le calcul de la MMM. La procédure de ce modèle est montrée sur la figure 6.3. Les résultats de simulations fonctionnelles associés au transfert du résultat *T* en base 2^{32}, sont représentés sur la figure 6.4.

```
restart; ## Initialisation
k:=32;   ## Valeur du paramètre k qui définit la base utilisée
n:=32;   ## Nombre de chiffres qui correspond au codage d'une donnée
         ## de taille 1024 bits en base 2^32

Montgomery:=proc(A,B,N)          ## Définition de la procédure de la MMM
   local R,T,R_1;
   R:=2^((n+1)*k);               ## Calcul du facteur R de la MMM
   R_1:=(R^(-1)) mod N;          ## Calcul du facteur R^-1
   T:=(A*B*R_1) mod N;           ## Exécution de la MMM
end proc:
```

Figure 6.3. Programme Maple du modèle mathématique de la MMM



Signal	***Description***
Addr_W_T	Bus d'adresse associé au stockage du résultat *T*
N_T	Bus de données de sortie de la mémoire *FIFO* du module *MulMong*
MMM_Status	Signal permettant d'indiquer par un *"1"* logique la fin d'exécution de la MMM
Valid_result	Signal permettant de valider le transfert du résultat *T* vers le processeur Microblaze
WFIFO_RdAck	Signal utilisé pour synchroniser le transfert de *T* vers le processeur Microblaze
Addr_R_T	Bus d'adresse de lecture du résultat *T*
T	Bus de données de sortie du module *MulMong*
IP2RFIFO_Data	Bus de données de la mémoire *FIFO (R_FIFO)* de l'interface *IPIF*

Figure 6.4. Chronogrammes issus de la simulation du transfert du résultat *T* vers le processeur Microblaze

6.2.7. Simulation fonctionnelle de *2MMMs_Core*

Les conditions d'implémentation du composant *2MMMs_Core* sont différentes de celles correspondant à l'implémentation de *MMM_Core*. En effet, le partitionnement logiciel/matériel proposé dans la seconde approche pour l''implémentation de l'exponentiation modulaire $Y=X^Z \ mod \ N$ est basé sur:

- le contrôle logiciel de l'algorithme 8 utilisé pour l'exécution de la fonction d'exponentiation modulaire,
- l'utilisation du composant *2MMMs_Core* comme accélérateur matériel,
- l'implémentation de deux unités arithmétiques *UAS* et *UAC*, pour calculer en parallèle les deux MMMs de chaque itération *(w)* de l'algorithme 8,
- le stockage des données d'entrée et des résultats intermédiaires $S_{(i)}$ et $C_{(i)}$ de l'algorithme 8 dans des mémoires locales. Ces dernières sont implémentées près des deux unités arithmétiques,

En tenant compte de ce partitionnement, le résultat *Y* calculé en utilisant le composant *2MMMs_Core*, ne sera obtenu en sortie qu'une fois avoir achevé l'exécution des trois étapes qui constituent le déroulement de l'algorithme 8. Autrement dit, jusqu'à l'épuisement de toute la chaine de bits de l'exposant *Z*, et après avoir représenté le résultat *Y* dans le domaine classique des nombres.

Pour ce faire, la simulation du fonctionnement du composant *2MMMs_Core* a été réalisée en se basant sur le calcul d'une exponentiation modulaire. Les vecteurs de test utilisés sont constitués :

- du modulo *N*,
- de la valeur numérique du message *X*,
- d'un exposant Z,
- de la valeur de $R^2 mod \ N$,
- et de la constante *N'*.

A noter que $R^2 mod \ N$ et *N'* sont calculées en fonction du modulo *N* et de la base β, utilisée pour le codage des données.

Les valeurs décimales des vecteurs de test, générées par l'outil Maple sont définies telles que :

N=
1070531751982841494000100452566027111186969614457275800920138363288453725499632042371518961905362446725449758320775718179845201855918761941100508648545224422187253627704765969322432351471113543973998269149581674921507216828127337493136777018180002118672293894582224611708412128622327098559459155571085107983 79

X=
6474426092381699707349775388414002200185272305465478268602060475368781316995641364671095905268770123480130553921413133248423585077246494134930836361388643177247099512746715847257025283799029413753596831679403010623544424562078318061947887937823736453860266000047406709076863290022727026939290653011988504 49.

Z=*1643233063.*

Valeurs de $R^2 mod\ N$ et de *N'* en base 2^{16} :

R^2 mod N=
3363701530742355440570040467337960562513938154638854523731407306503116770447549201399882335659508212766596050309931815583885524266543284952341425845825950645025306089449101382843696705097887932194225977977831517343541070979096814393965098246568554118952717759663246134820805556045508258451038760469848074455.

N'=*33661.*

Valeurs de $R^2 mod\ N$ et de *N'* en base 2^{32} :

R^2 mod N=
1035810342538065975820360876899650503827368665510821038399742860824751605972317528481738798653778056682366255617195198617079533029348822899147552676586217301092238526088046697610209302920389561873366755452154101815505510023038186392797369049371048553226186586966649935250771883045952137428443436364547140262 53.

N'= *1391035261.*

La valeur décimale du résultat *Y* obtenue par la simulation est :

Y=
4225146391627099592649534562739944516849544195387030551750586036029732431074004214045702420082490744885969689667650896205330089675818916564400686281388648152137268195416268579257464038019566884660341971821706638766960572834574823235240370644414777955167119277361690702399256834218561633625067554971000571895 0.

La vérification du fonctionnement du composants 2*MMMs_Core* pour calculer une exponentiation modulaire, a été effectuée en se basant sur la simulation des différentes phases qui constituent son exécution, à savoir:

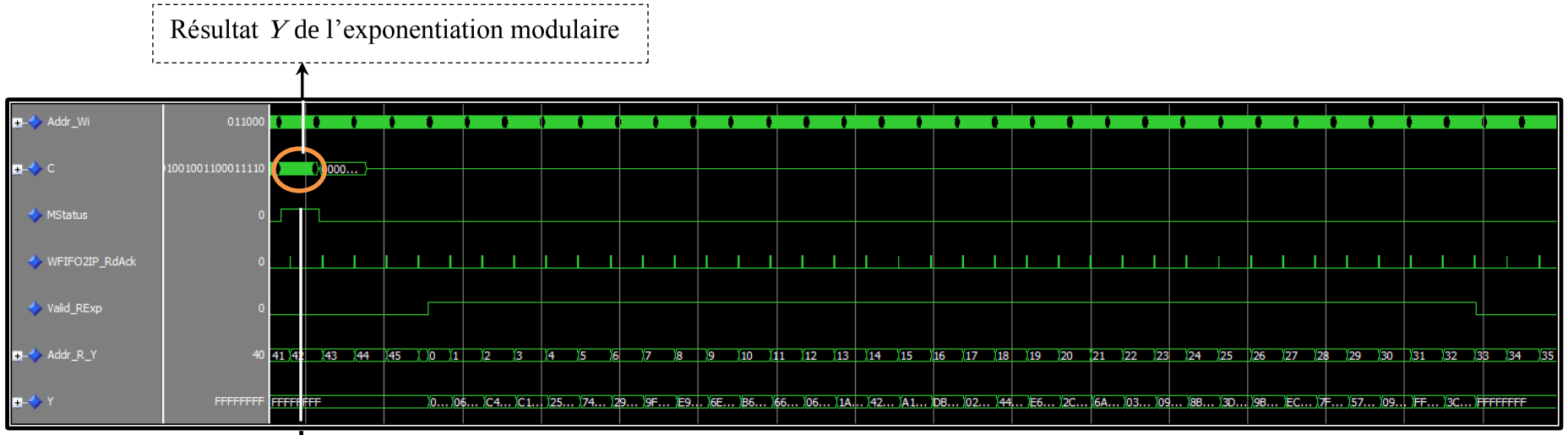
- Chargement des données d'entrées dans les unités de stockages qui leurs sont associées dans le module 2*MulMong*.
- Exécution des trois étapes de l'algorithme 8.
- Transition entre deux itérations successives *(w)* et *(w+1)* de l'algorithme 8.
- Transfert du résultat *Y* vers le processeur Microblaze.

Le résultat *Y* a été vérifié sur Maple, où le calcul d'une exponentiation modulaire a été effectué sur les mêmes données. Le modèle mathématique implémenté pour vérifier l'exactitude de *Y*, est montré sur la figure 6.5.

```
restart; ## Initialisation
Exp_Mod:=proc(X,Z,N)## Définition de la procédure de l'exponentiation modulaire
local Y;
 Y:=X &^Z mod N;     ## Exécution de la fonction de l'exponentiation modulaire
end proc:
```

Figure 6.5. Programme Maple pour le calcul de la fonction d'exponentiation modulaire

Les chronogrammes de simulation de l'étape du transfert du résultat *Y* en base 2^{32}, sont présentés sur la figure 6.6.



Signal	***Description***
Addr_W	Bus d'adresse associé au stockage du résultat *Y*
C	Bus de données de sortie de la mémoire *FIFO_C* du module *2MulMong*
MStatus	Signal permettant de valider le stockage des résultats intermédiaire de l'algorithme 8
WFIFO_RdAck	Signal utilisé pour synchroniser le transfert de *Y* vers le processeur Microblaze
Valid_result	Signal permettant de valider le transfert du résultat *Y* vers le processeur Microblaze
Addr_R_Y	Bus d'adresse de lecture du résultat *Y*
Y	Bus de données de sortie du module *2MulMong*
IP2RFIFO_Data	Bus de données de la mémoire *FIFO (R_FIFO)* de l'interface *IPIF*

Figure 6.6- Chronogrammes de simulation du transfert du résultat *Y* vers processeur Microblaze

6.3. Performances des approches proposées pour l'implémentation de la MMM et de la fonction d'exponentiation modulaire

La complexité d'exécution de la fonction d'exponentiation modulaire, dépend de six facteurs, à savoir :

1. la base $\beta=2^k$ utilisée pour le codage des données,
2. la taille de l'exposant *Z*,
3. le nombre de *"1"* logique dans la chaine de bits de l'exposant *Z*,
4. l'approche utilisée pour son implémentation (séquentielle ou parallèle).
5. les performances d'exécution de l'unité arithmétique, proposée pour l'implémentation de l'algorithme de la MMM sur matériel,
6. la performance de la liaison de communication assurant le transfert de données et d'instructions entre le processeur Microblaze et les deux composants *MMM_Core* et *2MMMs_Core*.

Dans le but d'étudier l'influence de ces facteurs sur les performances d'exécution du crypto système RSA, en particulier sur la partie implémentée sur circuit FPGA, l'évaluation de la complexité d'exécution des implémentations réalisées sera portée sur :

1. Les performances d'exécutions de l'*UA*.
2. Les performances temporelles d'une MMM.
3. Les performances de chiffrement et de déchiffrement pour une taille de clé de *1024-bits*.

6.3.1. Performances d'exécutions de l'*UA* proposée pour le calcul de la MMM

L'unité arithmétique présentée dans le chapitre 4, a été implémentée sur un circuit FPGA de la famille Virtex-5, en l'occurrence le circuit *XC5VLX50T-1ff1136*. Dans le chapitre 2, nous avons montré que la complexité de l'algorithme 7, utilisé pour concevoir l'architecture interne de l'*UA* dépend principalement de la base β, où $\beta=2^k$. De ce fait, dans le but d'étudier l'impact de ce paramètre sur les ressources matérielles requises et le temps d'exécution de l'*UA*, trois configurations sont considérées, à savoir : $\beta=2^{16}$, $\beta=2^{32}$ et $\beta=2^{64}$ pour un modulo *N* de taille *m =1024 bits*.

a) Ressources matérielles requises :

Les ressources matérielles requises pour l'implémentation de l'*UA* en fonction des valeurs de β, sont montrées sur le tableau 6.1. Ces ressources sont quantifiées en termes de nombre de *slices* et de blocs *DSP48E*

Tableau 6.1. Ressources matérielles occupées par l*'UA*

m (bits)	β	***(k, n)***	***Slices***	***DSP48E***
1024	2^{16}	(16,64)	39	3
	2^{32}	(32,32)	123	11
	2^{64}	(64,16)	257	42

- *n* représente le nombre de chiffres associés au codage de *N* en base 2^k, calculé tel que *: n=m/k*

A partir de ces résultats, on constate que le nombre de slices requis par les trois configurations est très faible. La comparaison avec le nombre total de slices disponibles sur le circuit FPGA utilisé qui est de *7200 slices* [70], montre que notre l'*UA* est optimisée en terme de nombre de slices. De ce fait, on peut considérer que l'objectif ciblé par la méthode proposée dans le chapitre 2 pour l'implémentation de la MMM, est largement atteint.

En revanche, l'analyse de l'influence de la base β, montre que cette dernière présente un impact considérable sur les ressources matérielles. En effet, comme la taille des additionneurs et des multiplieurs utilisés dans l'architecture interne de l'*UA* est définie par le paramètre *k*, par conséquent l'augmentation de ce dernier induit à une consommation élevée les ressources matérielles. En particulier, dans le nombre de blocs *DSP48E*.

Le résultat qui correspond à l'utilisation de la base 2^{64} montre que cette configuration présente un inconvénient majeur dans le nombre de blocs *DSP48E* requis. En effet, la comparaison par rapport au nombre total de blocs disponibles sur le circuit FPGA *XC5VLX50T*, a révélé que cette ressource est utilisée avec un taux de *87,5 %*. Ce qui signifie que l'implémentation de *l'UA* en base 2^{64} nécessite un nombre important de blocs *DSP48E*. De plus, comparé à l'utilisation de la base 2^{32}, le nombre de blocs *DSP48E* associé à cette configuration ne représente que 26 % du nombre obtenu avec l'utilisation de la base 2^{64}.

b) Performances temporelles de l'*UA*.

Les performances temporelles de l'*UA* en fonction de la base β sont illustrées sur le tableau 6.2.

Tableau 6.2. Performances temporelles de l'*UA* en fonction de β

m (bits)	β	***(k,n)***	t_{clk} ***(ns)***	t_i ***(ns)***	t_M ***(µs)***
1024	2^{16}	(16,64)	6.489	460.719	30.407
	2^{32}	(32,32)	9.624	375.336	12.421
	2^{64}	(64,16)	27.3	627.9	11.302

Les délais mentionnés dans ce tableau sont définie comme suit :

- t_{clk} : représente la période de l'horloge.
- t_i : correspond au temps d'exécution d'une itération *(i)* de l'algorithme de la MMM (algorithme 7).
- t_M : est le temps d'exécution d'une MMM.

La période d'horloge t_{clk} est déterminée par le délai du chemin critique de l'*UA*. En effet, en analysant l'architecture pipeline de cette unité (voir figure 4.3), t_{clk} est calculée à partir de l'étage, où le délai de calcul est le plus élevé. Le délai en question correspond au temps de calcul de l'un des deux produits $A[i]\times B[j]$ et $q_{(i)}\times N[j]$. Chacun de ces derniers nécessite un multiplieur dont la taille du bus de donnée est de *k-bit*. L'analyse temporelle pour déterminer la valeur de t_{clk} a été effectuée en utilisant l'outil *Timing Analyser* d'ISE.

t_i et t_M sont calculés par la multiplication de t_{clk} par le nombre de tops d'horloge nécessaire respectivement :

- à l'exécution d'une itération *(i)*, pour achever le calcul des $j^{ème}$ chiffres de la variable $L_{(i)}$,
- à l'exécution d'une MMM, pour avoir le résultat *T*.

En effet, dans le paragraphe 4.3.1.b du chapitre 4, l'évaluation de la complexité d'exécution de l'*UA* a montré que :

- Le calcul de la variable $L_{(i)}$ nécessite $(n+7)$ tops d'horloge.
- Le calcul du résultat *T* de la MMM est effectué sur $(n+2)\times(n+7)$ tops d'horloge.

De ce fait, t_i et t_M peuvent être obtenus en utilisant les deux formules suivantes :

$$t_i = (n+7) \times t_{clk} \quad (6.1)$$

$$t_M = (n+2) \times (n+7) \times t_{clk} \quad (6.2)$$

Les résultats du tableau 6.2 montrent qu'en augmentant le paramètre *k*, la période d'horloge t_{clk} augmente. Ceci est dû au chemin critique de l'*UA* qui devient plus long. En d'autre termes, les complexités des deux multiplieurs *Mul1* et *Mul3* utilisés respectivement pour calculer les produits $A[i] \times B[j]$ et $q_{(i)} \times N[j]$, deviennent plus élevées.

En revanche, les résultats liés au temps d'exécution t_M montrent que, bien que la complexité des opérations de base (les additions et les multiplications utilisées) augmente avec l'augmentation de *k*, les temps d'exécution issus des trois configurations diminuent. Ceci est justifié par l'optimisation du nombre d'itérations nécessaires à l'exécution de l'algorithme 7. Le nombre en question est inversement proportionnelle à *k*. Il est calculé par *(m/k)+1*. *m* représente le nombre de bits, associés au codage du modulo *N* en *base 2*. En effet, comme on peut le constater à partir du tableau 6.2, le temps d'exécution t_M obtenu avec l'utilisation de la base 2^{64}, présente le meilleur profil en termes de vitesse d'exécution. Comparées aux résultats qui correspondent à $\beta=2^{16}$ et $\beta=2^{32}$, les améliorations sont respectivement de l'ordre de *62,83%* et de *11,43 %*.

Bien qu'il soit montré que l'utilisation de la base 2^{64} permet d'optimiser le temps d'exécution de l'*UA*, le nombre de ressources matérielles requises pour son implémentation est élevé. En particulier, en termes du nombre de blocs *DSP48E*, la configuration en question nécessite *42 blocs*. De plus, l'implémentation de l'exponentiation modulaire en utilisant la seconde approche, où le composant *2MMMs_Core*, réalisé avec *2 UAs* en parallèle, est impossible. En effet, l'évaluation du nombre de blocs *DSP48E* requis pour l'implémentation de ce composant, montre que cette dernière nécessite *84 blocs*. Ce qui dépasse le nombre maximal disponible sur le circuit FPGA utilisé qui est de *48 DSP48E* [87]. De ce fait, les trois approches proposées dans ce travail pour l'implémentation de la fonction d'exponentiation modulaire ont été réalisées, en utilisant seulement les deux bases 2^{16} et 2^{32}.

6.3.2. Performances temporelles d'une MMM

Dans cette étude on s'intéresse aux performances temporelles qui correspondent à l'exécution d'une MMM dans les trois approches proposées, pour l'implémentation de

l'exponentiation modulaire. Les résultats sont relevés à partir des tests réels. Ces derniers sont effectués sur la carte de prototypage Genesys.

Dans les implémentations proposées, l'utilisation de l'approche de conception logicielle/matérielle permet d'améliorer le temps d'exécution par rapport à une implémentation purement logicielle. Cependant, une problématique liée à la liaison assurant la communication entre les composants réalisés et le processeur doit être considérée. En effet, la première et la deuxième approche d'implémentation sont basées sur une communication du type *"Memory-Mapped-Interface"* [7]. Celle-ci est caractérisée par un faible débit de transfert de données et d'instructions. De ce fait, les performances d'exécution de l'exponentiation modulaire deviennent dépendantes non seulement de la complexité algorithmique de l'algorithme 8, mais aussi, des performances de cette liaison. Ce facteur peut avoir un impact considérable dans la première implémentation, où les résultats intermédiaires de l'algorithme 8 sont stockés dans la mémoire locale du processeur. En d'autres termes, la MMM est calculée d'une manière intensive, impliquant une grande quantité de données transférées entre le processeur et le composant *MMM_Core*. Ainsi, l'objectif de la deuxième approche d'implémentation consiste en non seulement l'exploitation du parallélisme de l'algorithme 8, mais aussi, la réduction des données transférées entre le composant *2MMMs_Core* et le processeur Microblaze.

Dans les deux approches basées sur la conception logicielle/matérielle, les rapports fournis par l'analyseur temporelle ont révélé que les fréquences maximales f_{max} qui correspondent aux implémentations réalisées sur FPGA en bases 2^{16} et 2^{32}, sont respectivement *127,47 Mhz* et *100,261 Mhz*. Ces fréquences sont calculées à partir du chemin critique de l'*UA*. De ce fait, les tests physiques effectués sur la carte de prototypage ont été réalisés avec une fréquence *f=100Mhz*.

Les performances temporelles d'une MMM en fonction de l'approche d'implémentation et de β sont illustrées sur le tableau 6.3. Ces performances sont évaluées en termes du nombre de tops d'horloge *(NbTH_MMM)* et de temps d'exécution *(T_{MMM})*. *NbTH_MMM* et T_{MMM} sont calculés en incluant les délais associés aux chargements des données dans chacune des trois approches. T_{MMM} est calculé en multipliant *NbTH_MMM* par la période d'horloge t_{clk}. Cette dernière est calculée telle que : $t_{clk} = 1/f$, avec $f = 100Mhz$.

Tableau 6.3. Performances temporelles d'une MMM en fonction de β et de l'approche d'implémentation

β	*Approche*	*f (Mhz)*	*NBTH_MMM*	t_{MMM} *(ms)*
2^{16}	**Approche (1)**, une MMM dans *MMM_Core*	100	19387	0.193
	Approche (2), deux MMMs dans *2MMMs_Core*	100	4686	0.046
	Approche (3), Implémentation purement logicielle	100	421889	4.218
2^{32}	**Approche (1)**, une MMM dans *MMM_Core*	100	7191	0.071
	Approche (2), deux MMMs dans *2MMMs_Core*	100	1326	0.013
	Approche (3), Implémentation purement logicielle	100	169484	1.694

Dans la première approche d'implémentation, où une seule MMM est implémentée dans *MMM_Core*, *NbTH_MMM* est calculé en activant le *Timer*, après la transmission de l'instruction de chargement de deux opérandes *"RunWriteAB"* par le processeur Microblaze. Le *Timer* est désactivé dès la transmission du chiffre le plus significatif du résultat *T* par *MMM_Core.* Ainsi, la complexité d'exécution d'une MMM en termes de nombre de tops d'horloge dans la première approche d'implémentation, peut être évaluée en utilisant l'expression suivante :

$$NbTH\_MMM = NbTH1 + NbTH2 + NbTH3 \qquad (6.3)$$

- *NbTH1* correspond à la complexité de *l'UA.* Celle-ci est définie par l'équation (4.6) du chapitre 4. La complexité en question dépend principalement du nombre de chiffres associé au codage des données en base 2^k qui est de *(n+1) chiffres.* Dans les deux bases étudiées, à savoir $\beta=2^{16}$ et $\beta=2^{32}$ et pour une sécurité de taille *1024-bits,* les données sont codées respectivement sur *65 chiffres* et *33 chiffres.*
- *NbTH2* et *NbTH3* représentent respectivement, le nombre de tops d'horloge nécessaires au chargement de deux opérandes à partir de la mémoire BRAM et au transfert du résultat *T* vers le processeur Microblaze.

Dans la deuxième approche, où deux *MMMs* sont implémentées dans l'architecture interne du composant *2MMMs_Core*, comme toutes les données sont

stockées dans des mémoires locales, près des deux unités arithmétiques *UAS* et *UAC*, *NbTH_MMM* est réduit à la complexité *NbTH1* de l'*UA.*

Dans la troisième approche d'implémentation (implémentation logicielle de la MMM et de l'exponentiation modulaire), *NbTH_MMM* est déterminé en activant et en désactivant le *Timer* respectivement, à l'entrée et à la sortie de la fonction *Montgomery_SW*. Celle-ci est détaillée dans le paragraphe 5.5.2 du chapitre 5.

A partir des résultats illustrés dans le tableau 6.3, on constate que les implémentations réalisées dans la deuxième approche, présentent les meilleurs temps d'exécution. A titre d'exemple, en utilisant la base 2^{32}, comparé au résultat de l'implémentation purement logicielle, le gain est de l'ordre de *99,23%*. De plus, le fait d'avoir stocké toutes les données dans des mémoires locales, à l'intérieur de l'architecture interne du module *2MulMong*, ceci nous a permis de réduire le temps d'exécution de *81,69%,* par rapport au résultat de la première approche.

Ces résultats montrent aussi que, bien que l'implémentation réalisée dans la première approche soit plus efficace par rapport à la solution purement logicielle, le partitionnement logiciel/matériel proposé, conduit à une complexité élevée. En effet, d'après l'équation (4.6), en utilisant une sécurité de *1024-bits* et $\beta=2^{32}$ *(n= 32)*, l'*UA* nécessite seulement *1326 tops d'horloge* pour exécuter une MMM avec la seconde approche. Alors que les résultats ont montré que cette opération est calculée sur *7191 tops d'horloge*, lorsqu'on utilise la première approche. La différence est due aux deux complexités *NbTH2* et *NbTH3* qui représentent *81,5%* de *NbTH_MMM*. De ce fait, dans la première approche d'implémentation, la communication entre le processeur et *MMM_Core*, peut être considérée comme un goulot d'étranglement, limitant les performances d'exécution de l'exponentiation modulaire.

6.3.3. Performances d'exécutions de l'exponentiation modulaire

Le fonctionnement correct de la plateforme de chiffrement et de déchiffrement RSA, présenté dans le chapitre 4, a été validé par des tests effectués sur la carte de prototypage Genesys. Ces tests ont été vérifiés pour les trois approches proposées pour l'implémentation de la fonction d'exponentiation modulaire $Y=X^Z \bmod N$. Dans le but d'évaluer les performances de ces approches, les tests effectués ont été basés sur l'utilisation :

- d'une clé de taille *1024-bits*,
- de deux bases β pour le codage des données, à savoir $\beta = 2^{16}$ et $\beta = 2^{32}$.
- de trois exposants notés par *Z1*, *Z2* et *Z3*.
- d'un message *X* dont la valeur numérique est inférieure à la valeur du Modulo *N*.

Les deux premiers exposants *Z1* et *Z2* sont choisis avec une taille de *16-bits*. Leurs valeurs sont respectivement : $Z1=32769=(1000000000000001)_2$ et $Z2=65535=(1111111111111111)_2$. Ces deniers sont utilisés principalement pour déterminer l'influence de la chaine de bits de l'exposant sur le temps d'exécution de l'exponentiation modulaire. Le troisième exposant *Z3* est de taille *1024-bits*, dont *515-bits* sont non zéro. Cet échantillon est proche de la valeur moyenne où le nombre de bits non zéro est de *512-bits*. En effet, l'utilisation de cet exposant permet d'évaluer les délais d'exécution moyens des approches (1) et (3), puisque ces dernières sont basées sur l'exécution séquentielle des deux MMMs de l'algorithme 8.

a) Performances temporelles de l'exponentiation modulaire

Les performances temporelles de l'exponentiation modulaire en fonction de β, des trois approches d'implémentation et des trois exposants utilisés, sont illustrées sur le tableau 6.4. Les paramètres du protocole RS232 pour l'évaluation de ces performances, ont été configurés à *115200-bps, None, 8-bits, 1*.

Dans ce tableau, t_{Rec} et t_{Tran} représentent respectivement le temps de réception des données d'entrées et le temps de transmission du résultat *Y* à travers l'*UART*. *NbTH_EM* est le nombre de tops d'horloge nécessaire à l'exécution de l'exponentiation modulaire dans les trois approches d'implémentation. t_{ExpMod}, représente son délai d'exécution. t_{Rec}, t_{Tran} et t_{ExpMod} sont obtenus par la multiplication du nombre de tops d'horloge indiqué par le *Timer*, par la période d'horloge t_{clk}.

L'utilisation de *Z1* et de *Z2* montre que le nombre de bits non zéro de l'exposant, présente une influence considérable sur les temps d'exécutions t_{ExpMod} des approches d'implémentation (1) et (3). En effet, comme on peut le constater, dans les deux bases utilisées, t_{ExpMod} qui correspond au chiffrement avec *Z1*, est inférieur au délai obtenu avec l'utilisation de *Z2*. Ceci est dû principalement au nombre de bits non zéro de cet échantillon qui représente le cas le plus défavorable, lorsque un exposant de taille *16-bits* est utilisé. Alors que, le temps d'exécution t_{ExpMod} reste constant dans la seconde approche d'implémentation. La raison est que les deux MMMs de chaque itération *(w)*

de l'algorithme 8 sont exécutées en parallèle, indépendamment de la valeur du $i^{ème}$ bit de l'exposant.

Tableau 6.4. Performances temporelles d'exécution de l'exponentiation en fonction de β, de l'approche d'implémentation et des trois exposants utilisés

β	*Approche*	*Exposant Z*	*f (Mhz)*	t_{Rec} *(ms)*	t_{Tran} *(ms)*	*NbTH_EM*	t_{ExpMod} *(ms)*
2^{16}	**Approche (1)**, une MMM dans *MMM_Core*	Z1	100	74.27	12.18	6.82 × 10^5	6.82
		Z2	100	78.06	12.17	9.38 × 10^5	9.38
		Z3	100	76.42	12.18	282.42 × 10^5	282.42
	Approche (2), deux MMMs dans 2*MMMs_Core*	Z1	100	76.78	12.18	1.76 × 10^5	1.76
		Z2	100	78.59	12.17	1.76 × 10^5	1.76
		Z3	100	78.45	12.17	48.05 × 10^5	48.05
	Approche (3), Implémentation purement logicielle	Z1	100	74.97	12.18	442.62 × 10^5	442.62
		Z2	100	78.29	12.18	610.09 × 10^5	610.09
		Z3	100	76.66	12.17	18445.9 × 10^5	18445.9
2^{32}	**Approche (1)**, une MMM dans *MMM_Core*	Z1	100	62.52	9.67	1.50 × 10^5	1.50
		Z2	100	62.30	9.67	2.49 × 10^5	2.49
		Z3	100	61.36	9.67	109.52 × 10^5	109.52
	Approche (2), deux MMMs dans 2*MMMs_Core*	Z1	100	51.69	8.06	0.36 × 10^5	0.36
		Z2	100	50.52	8.06	0.36 × 10^5	0.36
		Z3	100	51.04	8.06	15.14 × 10^5	15.14
	Approche (3), Implémentation purement logicielle	Z1	100	60.22	9.67	35.59 × 10^5	35.59
		Z2	100	60.45	9.67	59.32 × 10^5	59.32
		Z3	100	63.03	9.67	2613.86 × 10^5	2613.86

Ces résultats montrent aussi que pour les deux bases utilisées, le crypto système de la seconde approche présente le meilleur temps d'exécution. En effet, comme le partitionnement logiciel/matériel proposé est basé sur l'implémentation parallèle des deux MMMs et le stockage de toutes les données dans l'architecture interne du composant *2MMMs_Cores*, ceci permet d'optimiser le temps d'exécution de t_{ExpMod}. A titre d'exemple en base 2^{32}, le délai obtenu (t_{ExpMod} = *15.14 ms*) représente des améliorations de l'ordre de *86%* et de *99%*, comparé respectivement aux résultats obtenus avec les approches (1) et (3). A noter que ces taux sont calculés en utilisant l'exposant *Z3*.

b) Ressources matérielles requises :

Les ressources matérielles requises pour l'implémentation des trois approches proposé pour la réalisation de l'exponentiation modulaire en fonction de β, sont montrées sur le tableau 6.5. Ces résultats sont mentionnés en termes : de nombre de *slices*, de blocs *RAM (36-kbits et 18-kbits)* et de blocs *DSP48E*.

Tableau 6.5. Ressources matérielles requises pour l'implémentation des trois approches proposées

β	*Approche*	*Slices*	*Bloc RAM 36-kbits*	*Bloc RAM 18-kbits*	*DSP48E Core*
2^{16}	**Approche (1)**, une MMM dans *MMM_Core*	1081	4	6	6
	Approche (2), deux MMMs dans *2MMMs_Core*	1311	4	11	9
	Approche (3), Implémentation purement logicielle	1038	4	-	3
2^{32}	**Approche (1)**, une MMM dans *MMM_Core*	1645	4	6	14
	Approche (2), deux MMMs dans *2MMMs_Core*	2315	4	11	25
	Approche (3), Implémentation purement logicielle	1038	4	-	3

La troisième approche d'implémentation peut être considérée comme une architecture de base. En effet, quelle que soit la base utilisée, le processeur Microblaze et ses périphériques, composés par la *BRAM*, le *Timer* et l'*UART* occupent *1038 slices*, *4 blocs RAM 36-kbits* et *3 DSP48E*.

En base 2^{16}, la comparaison entre la première approche et la troisième approche a montré que l'implémentation de *MMM_Core*, nécessite *43 slices*, *6 blocs RAM18-kbits* et *3 DSP48E*, comme ressources supplémentaires. Ces dernières sont occupées principalement par le composant en question qui intègre une seule MMM dans son architecture interne. En base 2^{32}, la comparaison entre les approches (1) et (3) montre qu'à l'exception du nombre de blocs mémoires qui reste constant, les deux autres ressources ont augmentée de *607 slices* et *11 DSP48E*. Dans le cas de la deuxième approche d'implémentation, où deux MMMs sont implémentées dans *2MMMs_Core*, l'augmentation de β de 2^{16} à de 2^{32} a révélé que *1004 slices* et *16 DSP48Es* sont nécessaires à son intégration sur circuit FPGA.

A partir de cette analyse, on constate que l'augmentation de β induit à une augmentation systématique dans la consommation des ressources matérielles. En particulier, dans le nombre de slices et de blocs *DSP48E*. Cette augmentation est due à la complexité en termes de ressources matérielles de l'*UA*, qui est intégrée dans les

deux composants *MMM_Core* et *2MMMs_Core*. En effet, comme il est indiqué dans le tableau 6.1, l'utilisation d'une base élevée, induit à une augmentation dans le nombre de ressources matérielles requises pour l'implémentation de cette unité.

Ces résultats montrent aussi que dans les deux bases utilisées, la seconde approche d'implémentation, nécessite plus de ressources matérielles, par rapport aux deux autres approches. En effet, comparé aux résultats de la troisième approche, en base 2^{32} la différence est de *1277 slices, 11 blocs RAM 18-kbits* et *22 DSP48Es*. Ainsi, on constate que le nombre des ressources matérielles requises pour l'implémentation de *2MMMs_Core* de la seconde approche, double par rapport à celui de l'implémentation de *MMM_Core* de la première approche. Ce résultat est attendu, car le composant *2MMMs_Core* est implémenté avec deux *UAs* parallèles.

L'analyse des performances temporelles a montré que le partitionnement logiciel/matériel, proposé dans la seconde approche d'implémentation, combiné à l'utilisation de la base 2^{32}, permettent d'améliorer d'une manière significative le temps d'exécution t_{ExpMod}. En ce qui concerne les ressources matérielles occupées, cette configuration occupe certes plus de ressources ; néanmoins, en tenant compte du gain important obtenu en termes de vitesse d'exécution, cette implémentation peut être considérée comme étant la solution qui présente le meilleur compromis entre : le temps d'exécution, les ressources matérielles occupées et la flexibilité du système. De plus, bien que le nombre de blocs mémoires utilisés *(11 blocs RAM 18-kbits)* soit élevé, l'application au chiffrement avec une clé de *1024-bits*, montre que seulement *8-kbits* sont occupés par toutes les données stockées. Ce qui représente un taux de *4%* de la capacité totale, fournie par les *11 blocs* occupés. Cette capacité est de : *11×18-kbits=198-kbits*. L'espace restant peut être exploité dans le futur pour réaliser un crypto système RSA avec des tailles de clés supérieures à *1024-bits*. Des clés de sécurité de cet ordre peuvent facilement être atteintes, puisque l'*UA* proposée dans ce travail est conçue indépendamment de la taille de la clé utilisée.

6.4. Comparaisons avec des travaux de l'état de l'art

Le tableau 6.6 illustre la comparaison de notre meilleure configuration (approche 2) avec les performances de quelques implémentations récentes, pour une taille de clé de *1024-bits*.

Tableau 6.6. Comparaison des performances avec des travaux utilisant une taille de clé de *1024-bits.*

Référence	***Conception***	***Processeur utilisé***	***f (Mhz)***	***Temps (ms)***	***Ressources matérielles***	***Famille du Circuit FPGA***
Notre travail	Logicielle/ Matérielle	Microblaze de Xilinx	100	15.14	2315 Slices 11 RAMs 25 DSP48E	Virtex-5
Perin et al [29]	Matérielle	-	130	3.23	6642 Slices 130 DSP48E	Virtex-5
Song et al [32]	Matérielle	-	447	36.37	180 Slices 1 RAM 1 DSP48E	Virtex-5
Wang et al [34]	Matérielle	-	200	6.79	5730 Slices	Virtex-5
Hani et al [36]	Logicielle/ Matérielle	Nios d'Altera	66	31.93	12881 LEs	Ep1s40
Uhsadel et al [40]	Logicielle/ Matérielle	Microcontrôleur 8051 d'Intel	111	17.12	27467 Slices	Virtex-4
Simka et al [41]	Logicielle/ Matérielle	Nios d'Altera	100	39	4112 LEs 27 RAMs	Ep20k200

Une approche d'implémentation basée sur les réseaux systoliques est présentée dans [29]. Les résultats d'implémentation montrent que l'utilisation de ces réseaux conduit à un meilleur délai d'exécution. Néanmoins, ce type d'optimisation nécessite beaucoup de ressources matérielles. En particulier le nombre de blocs *DSP48E* est jugé très élevé *(130 blocs).*

Song et al dans [32] présentent une implémentation purement matérielle de l'exponentiation modulaire. Les optimisations développées dans ce travail favorisent la réduction des ressources matérielles occupées, au profit du délai d'exécution. Comparer aux performances temporelles de cette implémentation, l'exécution de l'exponentiation modulaire par notre seconde approche, est approximativement 2 fois plus rapide.

Wang et al dans [34] présentent une conception matérielle d'un co-processeur RSA. La comparaison des performances temporelles montre que l'exécution de l'exponentiation modulaire, est approximativement 2 fois plus rapide que le résultat obtenu avec notre seconde approche. Cette différence est attendue, car les résultats

d'implémentation de ce travail concernent l'exécution de l'exponentiation modulaire sur matériel seulement. Cependant, en termes de nombre de Slices occupés, notre implémentation nécessite un nombre inférieur. La différence est de *3415 slices*.

Hani et al dans [36] présentent un crypto système RSA, basé sur le processeur Nios d'Altera. Dans ce travail, une IP Core dédiée au calcul de l'exponentiation modulaire est implémentée avec une seule MMM. Le contrôle de l'exposant est réalisé à l'intérieur de l'IP ; contrairement à nos implémentations, où le contrôle en question est effectué de manière logicielle par le processeur Microblaze. l'IP Core présentée dans cet article nécessite un nombre important en termes de ressources matérielles (*12881 Logic Elements*).

Usadel et al dans [40] présentent des implémentations de l'exponentiation modulaire, basées sur la combinaison des deux ressources : logicielle et matérielle. Dans ce travail, le microcontrôleur 8051 est utilisé pour la flexibilité et le circuit FPGA, permet d'augmenter les performances temporelles. En comparant aux résultats obtenus avec notre seconde approche, on constate que celle-ci conduit à de meilleures performances. En effet, le temps d'exécution et le nombre de slices obtenus représentent respectivement 88,43% et 8,4% des résultats mentionnés dans cet article.

L'implémentation de Simka et al dans [41] utilise une seule MMM sur matériel. Le contrôle de l'algorithme de l'exponentiation modulaire est exécuté de manière logicielle par le processeur Nios d'Altera. En termes de performances temporelles, le partitionnement logiciel/matériel proposé dans cet article, résulte en un temps d'exécution supérieur au délai obtenu avec notre seconde approche. De même, la comparaison en termes de ressources matérielles montre que notre méthode d'implémentation en base 2^{32} nécessite moins de ressources.

6.5. Conclusion

Dans ce chapitre, nous avons présenté les résultats d'implémentations des méthodes proposées pour la réalisation du crypto système RSA. Les résultats en question concernent principalement les parties implémentées sur circuit FPGA, en l'occurrence :

- L'unité arithmétique, réalisée pour l'exécution de la MMM.
- Les trois approches proposées pour l'implémentation logicielle/matérielle de l'exponentiation modulaire, basé sur le processeur Microblaze de Xilinx.

En vertue de la complexité algorithmique de la MMM et de l'exponentiation modulaire, nous avons présenté dans ce chapitre, la méthodologie élaborée pour vérifier le fonctionnement des implémentations proposées pour l'exécution de ces opérations sur circuit FPGA. Cette méthodologie est basée sur des simulations fonctionnelles et des comparaisons. Ces dernières sont effectuées en utilisant des modèles mathématiques.

Dans ce chapitre, nous avons également présenté les performances d'exécution des approches proposées pour l'implémentation de ces opérations. Les paramètres considérés pour étudier ces performances sont :

- L'utilisation d'une clé de taille *1024-bits*.
- L'utilisation des trois bases 2^{16}, 2^{32} et 2^{64} pour le codage des données.

Les résultats d'implémentation ont montré que l'implémentation parallèle de deux MMMs dans un composant matériel, combinée à l'utilisation de la base 2^{32}, permet non seulement d'optimiser le délai d'exécution de l'exponentiation modulaire, mais aussi d'atteindre un meilleur compromis entre le temps d'exécution, les ressources matérielles occupées et la flexibilité du crypto système.

Conclusion générale

L'objectif des travaux réalisés dans le cadre de cette thèse, est d'étudier la conception conjointe logicielle et matérielle du crypto système à clé publique RSA. Nous nous sommes intéressés en particulier à l'optimisation des opérations arithmétiques de ce crypto système, en vue de leurs implémentations dans un environnement PSoC, où le processeur Microblaze de Xilinx est utilisé pour la flexibilité.

L'opération de base du protocole de cryptographie RSA est l'exponentiation modulaire qui n'est rien d'autre qu'une suite de multiplications modulaires. De ce fait, pour optimiser les opérations du chiffrement et de déchirement, il faut d'une part réduire le nombre de multiplications induites lors du calcul de l'exponentiation modulaire et d'autre part, optimiser la complexité d'exécution de cette multiplication. Pour se faire, dans un premier temps, nos travaux étaient orientés vers l'étude et l'optimisation de la MMM. La méthode proposée pour sa réalisation, a été développée principalement pour :

- Optimiser les ressources matérielles requises pour son implémentation.
- Adapter son exécution au processeur Microblaze.

Par la suite, nous avons proposé trois approches pour l'implémentation de la fonction d'exponentiation modulaire, en se basant sur la combinaison des deux ressources logicielle et matérielle. La première approche repose sur l'utilisation du processeur Microblaze pour la flexibilité et l'implémentation d'une seule MMM sur matériel. La seconde est basée sur l'implémentation parallèle de deux MMMs dans un composant matériel. Le contrôle de l'algorithme de l'exponentiation modulaire est assuré par le processeur Microblaze. La troisième approche est une implémentation purement logicielle où les algorithmes de la MMM et de l'exponentiation modulaire sont exécutés par le processeur Microblaze.

A travers les résultats d'implémentions obtenus, nous avons montré que lorsque les opérandes et les résultats de la MMM sont stockés dans la mémoire BRAM du processeur Microblaze, les performances du RSA peuvent être limitées, en vertu de la quantité importantes des données échangées entre le processeur et le composant matériel. De ce fait, le partitionnement logiciel/matériel proposé dans la seconde approche d'implémentation est basé non seulement sur l'exécution parallèle de deux MMMs, mais aussi, sur le stockage de toutes les données dans des mémoires locales, près des unités arithmétiques. Cette stratégie

d'optimisation, nous a permis d'une part, de compenser le faible degré du parallélisme introduit dans l'exécution des opérations arithmétiques de l'*UA* et d'autre part, de réduire le nombre des ressources matérielles occupées sur le circuit FPGA.

Nos résultats d'implémentation ont montré aussi que la seconde solution proposée pour l'implémentation de l'exponentiation modulaire, combinée à l'utilisation de la base 2^{32}, permet non seulement d'optimiser son délai d'exécution, mais aussi d'atteindre un meilleur compromis entre :

- la flexibilité du crypto système
- le temps d'exécution,
- les ressources matérielles occupées,

La comparaison par rapport aux travaux de l'état de l'art qui propose des implémentations purement matérielle de l'exponentiation, a montré que notre meilleure configuration (deuxième approche d'implémentation) est moins performante en termes de délai d'exécution. Ce résultat est attendu car l'objectif ciblé par notre réalisation est de favoriser la flexibilité du crypto système, en utilisant un processeur embarqué, pour un délai d'exécution optimum.

En perspectives à ce travail, le crypto système RSA de la seconde approche sera exploité pour la réalisation d'un crypto système hybride, utilisant les deux types de cryptographies, symétrique et asymétrique. Dans ce crypto système, le RSA servira à transporter la clé de chiffrement symétrique. Nous comptons aussi étudier, l'influence de la taille de la clé sur les performances d'exécution des méthodes proposées pour la réalisation de la MMM et de l'exponentiation modulaire. Des tailles de clés supérieures à *1024-bits* seront testées. D'autres techniques de chiffrement asymétrique, telle que celles utilisant les courbes elliptiques, seront mises au point et évaluer sur circuit FPGA.

Annexe A

Architectures d'un circuit FPGA de la famille Virtex-5 et du processeur Microblaze.

Cette annexe est réservée à la description :

1. Des circuits FPGAs de la famille Virtex-5. On s'intéressera en particulier à leur aspect architectural et aux ressources matérielles disponibles sur ces circuits, exploitées dans ce travail pour la réalisation du crypto système RSA.
2. Du processeur Microbalze, utilisé comme étant le contrôleur principal de la partie embarqué sur circuit FPGA.

A.1. Famille Virtex-5 des circuits FPGA de Xilinx

La famille Virtex-5 a été conçue pour réaliser des conceptions à faible ou grande densité d'intégration et qui exigent des performances élevées. En plus, des ressources arithmétiques et logiques, la famille Virtex-5 supporte des processeurs embarqués softcore (Microblaze) et hardcore (Power Pc). Cette famille contient quatre sous familles nommées Virtex-5 LX (utilisée dans notre travail), Virtex-5 SX, Virtex-5 TX et Virtex-5 FX. Ces dernières se distinguent par le type des ressources matérielles qu'elles englobent. A titre d'exemple, la famille Virtex-5 FX inclut des processeurs hardcore, contrairement aux autres familles [87]. L'architecture d'un circuit FPGA, appartenant à la famille utilisée, est montrée sur la figure 1.A.

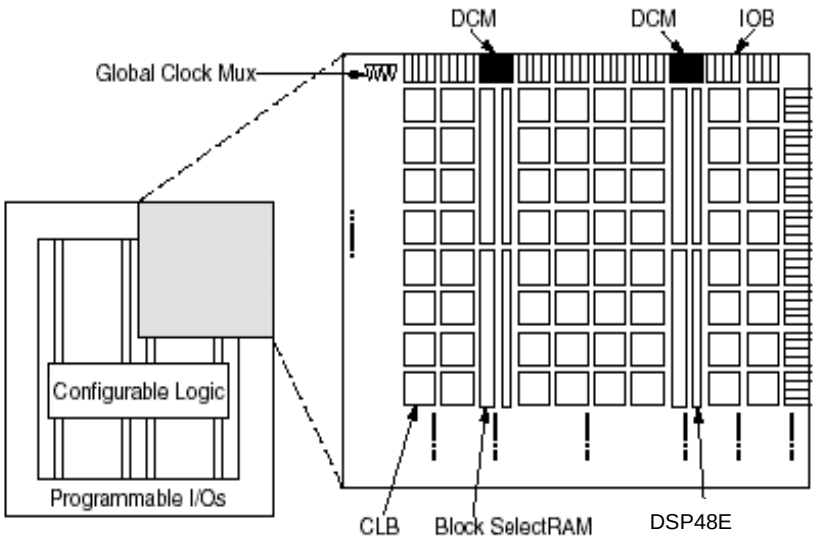


Figure. 1.A. Architecture interne d'un circuit FPGA de la famille Virtex-5

L’architecture d’un circuit FPGA de la famille Virtex-5, se présente comme un réseau régulier de blocs logiques, composé par [70]:

- Des blocs Input/Output (IOB).
- Des blocs Logique Configurables (CLB).
- Des blocs Select RAM à doubles ports.
- Des blocs DSP48E.
- Des DCMs (Digital Clock Manager).

En plus à ces composants, un circuit Virtex-5 est constitué aussi d’un chemin dédié à la propagation de la retenue (Carry Chain). Ce dernier permet d’implémenter des additionneurs performants. La communication entre l’ensemble des ressources disponibles sur le circuit, est assurée par des matrices d'interconnections programmables, nommées GRM (General Routing Matrix).

A.1.1. Blocs Input/Output (IOB)

Les IOBs sont des blocs qui permettent de communiquer avec la logique interne du circuit. Ils peuvent être configurables, en entrée, en sortie, ou bidirectionnelle.

A.1.2. Bloc logique configurable (CLB)

Les CLBs sont implémentés sous forme d’une matrice sur un circuit FPGA. Ils incluent toute la logique nécessaire pour la conception des circuits combinatoires et séquentiels. Chaque CLB de la famille Virtex-5 est composé de deux slices. Ces derniers constituent les éléments de base du circuit FPGA. La figure 2.A illustre l’implémentation des slices dans un CLB.

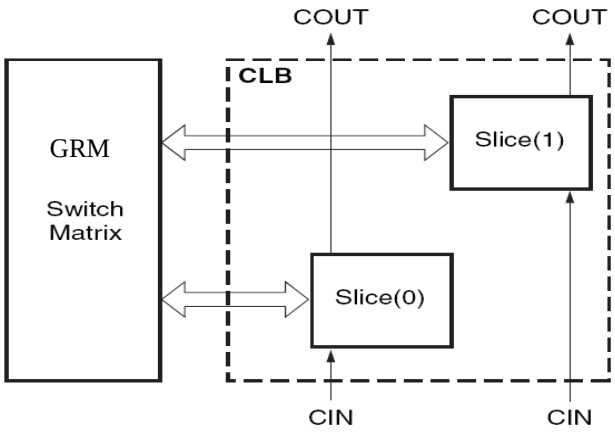


Figure. 2. A. Implémentation des slices dans un CLB Virtex-5 [70]

a. Description d’un slice

L’architecture interne d’un slice est montrée sur la figure 3.A. Il est composé de :

- Quatre LUTs (Look Up Table) à six entrées et deux sorties, dédiées à l'implémentation de circuits logiques combinatoires.
- Quatre bascules D.
- Quatre portes logiques *xor* et des multiplexeurs. Ces ressources permettent de réaliser des additionneurs à propagation de retenues avec des performances élevées.

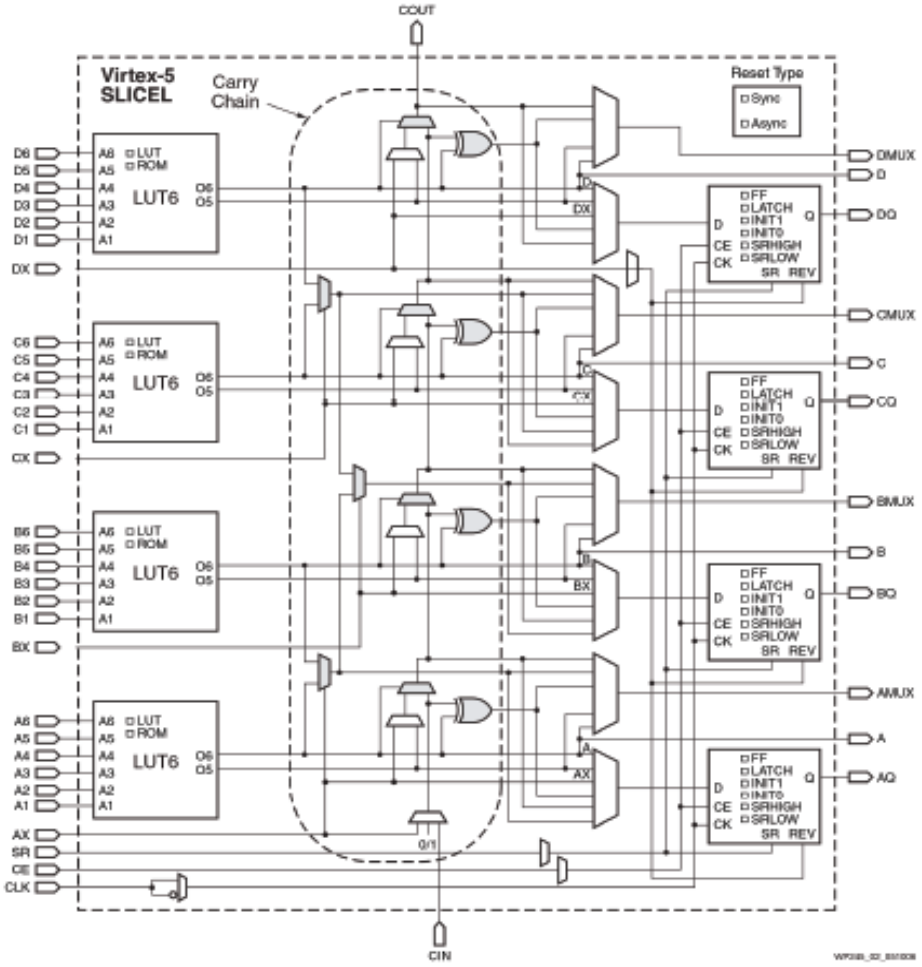


Figure.3.A. Architecture interne d'un slice [70]

b. Chemin de propagation de la retenue

L'addition est une opération cruciale dans diverses applications qui relèvent du domaine de traitement du signal numérique. Cette opération est connue par son chemin critique défini par la propagation de la retenue. Les circuits FPGA's dans leur majorité, comportent des ressources optimisées et spécifiques au calcul de cette retenue. D'une manière générale, l'opération en question est implémentée sur un circuit FPGA sous forme d'un additionneur à propagation de retenue, où cette dernière suit des chemins verticaux ; de la partie inférieure du circuit à sa partie

supérieure. Comme ceci est montré sur la figure 4.A. Si la taille de l'additionneur dépasse le nombre de slices d'une colonne, la retenue revient vers le bas du circuit et la propagation continue sur une des colonnes adjacentes.

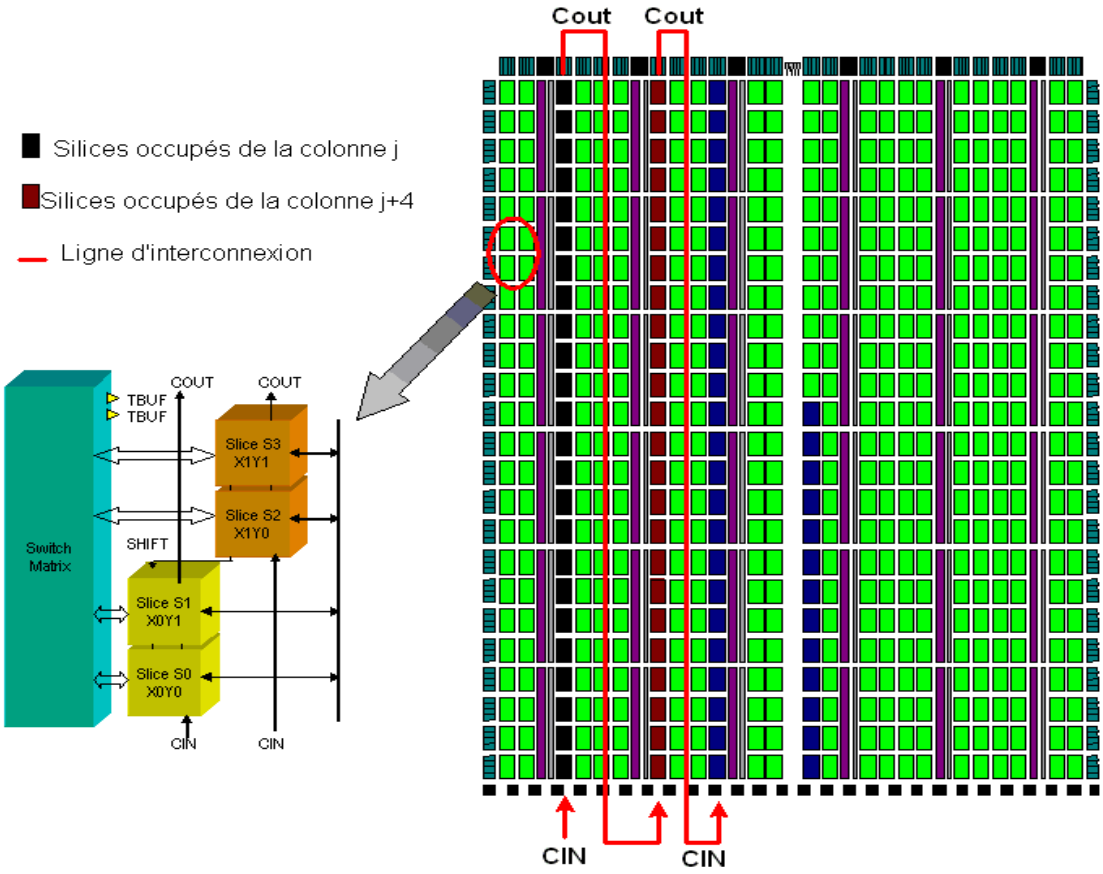


Figure.4.A. Chemin de propagation de retenue (Carry Chain)

c. Cellule élémentaire d'un additionneur sur un circuit FPGA de Xilinx

Un additionneur à propagation de retenue qui permet de sommer deux opérandes codés sur n bits, est constitué de n cellules élémentaires, nommées Additionneur Complet (Full Adder). Ce dernier permet de réduire en une somme *s(i)* et une retenue *c(i+1)*, trois bits du même poids *(i)*. Ces trois bits représentent respectivement: deux bits *a(i)* et *b(i)* de deux opérandes *A* et *B*. Le troisième bit est la retenue *c(i)* générée par le Full Adder du poids *(i-1)*. Les équations combinatoires représentants la somme *s(i)* et la retenue *c(i+1)* sont données par les expressions suivantes [23], [88] :

$$s(i) = a(i) \text{ xor } b(i) \text{ xor } c(i).$$

$$c(i+1) = [a(i) \text{ and } b(i)] \text{ or } [c(i) \text{ and } (a(i) \text{ xor } b(i))].$$

La table de vérité de la somme *s(i)* et la retenue *c(i+1)* est montrée sur le tableau A.1.

Tableau A.1. Table de vérité de la somme *(s)* et la retenue *(c)*

c(i)	*b(i)*	*a(i)*	*a(i) xor b(i)*	*s(i)*	*c(i+1)*
0	0	0	0	0	0
0	0	1	1	1	0
0	1	0	1	1	0
0	1	1	0	0	1
1	0	0	0	1	0
1	0	1	1	0	1
1	1	0	1	0	1
1	1	1	0	1	1

L'implémentation d'un Full Adder dans un slice repose sur la technique dite ″Propagation et Génération [88]″. Cette dernière est basée sur le calcul de: *a(i) xor b(i)*. Elle est décrite comme suit :

A partir de la table de vérité de *s(i)* et *c(i+1)*, on constate que :

- Si *a(i) xor b(i)=0, (a(i) = b(i))* ; la retenue sortante *c(i+1)* est identique à *a(i)* et *b(i)*. Ceci résulte en une génération de retenue par le full adder du poids *(i)*.
- Si *a(i) xor b(i)=1, (a(i) ≠ b(i))* ; la retenue sortante *c(i+1)* est identique à la retenue entrante *c(i)*. Cela signifie que le full adder du poids *(i)* propage la retenue *c(i)*.

Chaque full adder implémenté dans un slice, lui est associé un multiplexeur *muxcy* pour calculer la retenue *c* et une porte *xorcy* pour le calcul de la somme *s*. Par conséquent, si on pose *P(i) = a(i) xor b(i)*, ce signal peut être utilisé comme un signal de sélection dans le *muxcy* pour calculer la retenue *c(i+1)*. *P(i)* est obtenu à partir d'une LUT qui reçoit en entrées, les deux bits *a(i)* et *b(i)*. Le *muxcy* reçoit sur ses entrées la retenue *c(i)* fournie par le full adder du poids *(i-1)* et un des deux bits *a(i)* ou *b(i)*. Il permet donc de sélectionner suivant l'état de *P(i)*, entre la retenue entrante *c(i)* ou *a(i)*. La somme *s(i)* est calculée par l'utilisation de la porte *xorcy* qui reçoit sur ses entrées le signal *P(i)* et la retenue *c(i)*. Vu la structure symétrique d'un slice, la longueur de la propagation de la retenue à l'intérieur d'un slice est de deux bits. Autrement dit, un slice est capable d'implémenter deux Full Adders. Le schéma

détaillé de l'implémentation dans un slice de deux Fulls Adders successifs est montré sur la figure 5.A.

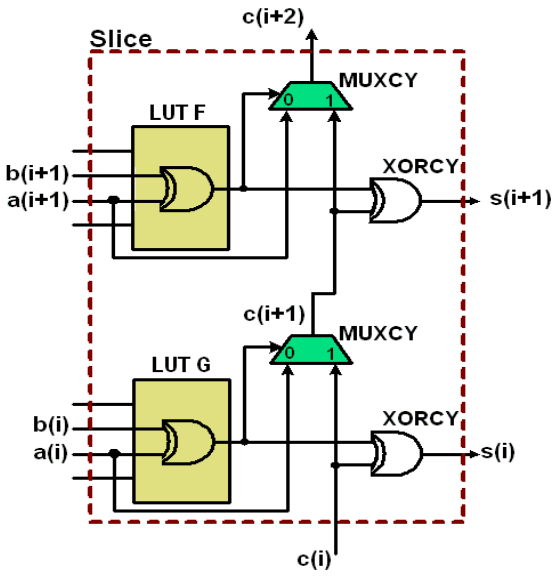


Figure.5.A. Implémentation de deux fulls adders successifs

A.1.3. Bloc Select RAM

Les circuits FPGAs de la famille Virtex-5 contiennent des blocs mémoires à double ports. Chaque bloc peut emmagasiner jusqu'à *36-kbits* de données. Ces mémoires sont implémentées dans des colonnes où leur nombre dépend de la taille du circuit utilisé [70], [87]. Les blocs Select RAM peuvent être configurés comme des mémoires à *36 kbits*, *2 × 18-kbits*, *1×18-kbits*, où comme une mémoire FIFO.

A.1.4. Bloc DSP48E

Chaque DSP48E est constitué :

- D'un multiplieur à deux entrées *A* et *B*, codées respectivement sur *25-bits* et *18-bits*.
- D'un additionneur/soustracteur à trois entrées.

La figure 6.A décrit un schéma simplifié d'un bloc DSP48E [31]. Il peut être configuré comme :

- un multiplieur,
- un multiplieur accumulateur,
- un multiplieur suivi d'un additionneur.

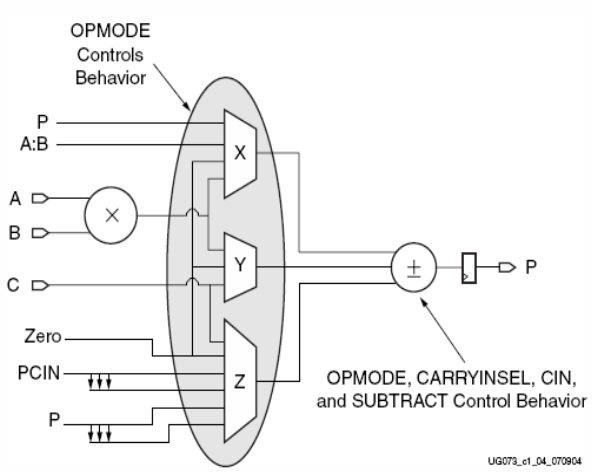


Figure 6.A. Schéma simplifié du bloc DSP48E

A.2. Architecture du processeur Microblaze

Le processeur MicroBlaze de Xilinx est un processeur *32-bits* à jeu d'instructions réduit RISC (Reduced Instruction Set Computer). Son architecture est présentée sur la figure 7.A [18].

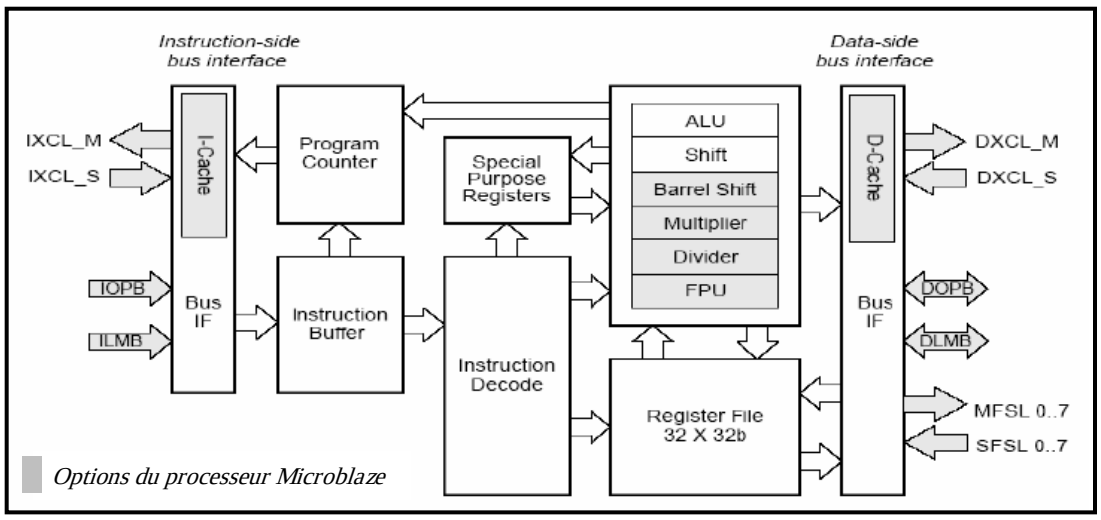


Figure 7.A. Architecture interne du processeur Microblaze

Les caractéristiques de Microblaze peuvent se résumer dans les points suivants :

- Sa conception est basée sur une architecture Harvard avec des bus d'instruction et de données séparés.
- Il possède 32 registres à usage général.
- Les instructions sont exécutées dans un pipeline de 5 étages.

Les parties grises sont optionnelles et montrent à quel point ce processeur est configurable. Ses options, en l'occurrence le Barrel Shift, le Multiplier, le Divider,

l'unité FPU (Floating Point Unit) et les mémoires caches, sont des composants implémentés sur matériel. Leurs utilisations permettent d'accélérer le traitement des données. Les fréquences maximales du processeur Microblaze varient selon la famille du circuit FPGA utilisée. Les fréquences en question, sont illustrées dans le tableau A.2

Tableau A.2. Fréquences du processeur Microblaze

Famille du circuit	Fréquence maximale
Virtex-2	125Mhz
Virtex-2 Pro	150Mhz
Virtex-4	180Mhz
Virtex-5	235Mhz

Microblaze possède quatre systèmes de communications qui lui permettent de communiquer avec les composants qui l'entourent dans un PSoC. Ces systèmes sont définis comme suit:

- On-Chip Peripheral Bus (OPB).
- Local Memory Bus (LMB).
- Fast Simplex Link (FSL).
- Xilinx Cache Link (XCL).

A.2.1. On-Chip Peripheral Bus (OPB)

Le bus OPB, conçu par IBM pour ses processeurs PowerPC, peut être utilisé comme un bus système, assurant la communication entre Microblaze et les IPs qui l'entourent. Il autorise un maximum de 16 Microblaze, configurées en maîtres, et un nombre d'esclaves limités, selon les ressources matérielles disponibles.

Dans la nouvelle génération des circuits FPGA de Xilinx, le bus OPB est replacé par le bus PLB (Processor Local Bus).

A.2.2. Local Memory Bus (LMB)

Le bus LMB est utilisé par le processeur pour communiquer avec les mémoires d'instructions et de données. Ce bus est connecté au processeur Microblaze via deux ports de 32 bits. Ces deniers sont nommés respectivement ILMB (Instruction Local Memory Bus) et DLMB (Data Local Memory Bus).

A.2.3. Fast Simplex Link (FSL)

MicroBlaze comporte 8 ports d'entrées/sorties FSL. La connexion FSL est un moyen rapide qui permet la communication direct entre le processeur et un

périphérique, sans passer par la mémoire BRAM du processeur. Chaque lien FSL est unidirectionnel.

A.2.4. Xilinx Cache Link (XCL)

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

Annexe B

Signaux de contrôle du circuit MMM_Ctr et configuration des mémoires internes du module MulMong réalisé dans la première approche d'implémentation.

Cette annexe décrit les signaux d'entrées/sorties du circuit de contrôle *MMM_Ctr* ainsi que les configurations des mémoires implémentées dans le module *MulMong*.

B.1. Signaux d'entrée/sortie du circuit de contrôle *MMM_Ctr*

Le circuit *MMM_Ctr* assure le contrôle des composants implémentés dans l'architecture interne du module *MulMon*g. Il reçoit en entrée le signal d'horloge *Bus2IP_Clk*, les trois signaux de sortie du registre d'instruction et le signal *WFIFO2IP_Data* (voir figure 4.8). Les signaux fournis par ce circuit sont présentés sur le tableau B.1.

Tableau B.1. Signaux de sortie du circuit de contrôle *MMM_Ctr*

Signal de sortie	**Description**
Addr_W_AB	Bus d'adresse utilisé pour le stockage des opérandes *A* et *B* respectivement dans les mémoires *Mem_A et Mem_B.*
Addr_W_N	Bus d'adresse correspondant au chargement du modulo *N* dans la mémoire *Mem_N*.
Addr_R_A, Addr_R_B, Addr_R_N	Bus d'adresses utilisés respectivement pour de la lecture de *A*, de *B* et de *N*, lors de l'exécution de la MMM.
Addr_W_T, Addr_R_T	Bus d'adresses utilisés respectivement pour le stockage du résultat *T* et pour son transfert au processeur Microblaze.
Sel_Mem_A, Sel_Mem_B, Sel_Mem_N, Sel_Reg_N'	Ces signaux sont actifs à l'état haut. Ils assurent respectivement la sélection des mémoires *Mem_A, Mem_B, Mem_N* et du registre *N'_Reg* lors de la réception des données qui leur sont associés.
En_qi, Reset_cy12, Reset_cy34	Signaux de contrôle de l'*UA*.
MMM_Status	Ce signal permet d'indiquer par un *"1"* logique la fin d'exécution de la MMM. Il est à l'état haut à l'itération $i = n+1$ de l'algorithme 7.
Valid_Result	Ce signal est actif à l'état haut. Il permet de valider le transfert du résultat *T* de la MMM vers le processeur Microblaze

B.2. Configurations des mémoires *Mem_N*, *Mem_A* et *Mem_B* du module *MulMong*

a. Configuration de la mémoire *Mem_N*

La mémoire *Mem_N* est configurée en écriture et en lecture respectivement lors du chargement du modulo *N* et pendant l'exécution de la MMM. La sélection de son adressage en phases d'écriture et de lecture est assurée par le multiplexeur *Mux1* (Voir figure 4.8). Ce dernier est commandé par le signal d'entrée *Out_Inst_reg(1)* du registre d'instruction *Inst_registre*. Ce signal est à l'état bas durant la phase du chargement des données. Il bascule à l'état haut dès le début d'exécution de MMM. La table de vérité du multiplexeur *Mux_1* et la configuration de la mémoire *Mem_N* sont présentées dans le tableau B.2.

Tableau B.2. Table de vérité du multiplexeur *Mux_1* et configuration de *Mem_N*

Table de vérité du Multiplexeur Mux_1			***Configuration de la mémoire Mem_N***
Out_Inst_reg(1)	***Sel_Mem_N***	***Sortie du Mux_1***	
0	0	*Addr_W_N*	**La mémoire est désactivée**
0	1	*Addr_W_N*	**Mode d'écriture** Stockage de N
1	0	*Addr_R_N*	**Mode de lecture** Lecture de *N*

b. Configuration de la mémoire *Mem_A*

La mémoire *Mem_A* est utilisée pour le stockage respectif de l'opérande *A* et du résultat *T* de la MMM. La sélection des bus d'adresses générés par le circuit de contrôle, en l'occurrence *Addr_W_AB*, *Addr_R_A*, *Addr_W_T* et *Addr_R_T*, est assurée par le Multiplexeur *Mux_2* (Voir figure 4.8). Ces adresses correspondent aux différentes étapes de calcul de la MMM, à savoir, chargement de *A*, exécution de la MMM, stockage de *T* et son transfert vers le processeur Microblaze. La table de vérité du multiplexeur *Mux_2* et la configuration de la mémoire *Mem_A* sont résumées dans le tableau B.3.

c. Configuration de la mémoire *Mem_B.*

La mémoire *Mem_B* assure le stockage de l'opérande *B.* Elle est configurée en écriture et en lecture respectivement lors de la réception de l'opérande *B* et pendant l'exécution de la MMM. La sélection des bus d'adresse qui correspondent à l'exécution de ces deux étapes est assurée par le multiplexeur *Mux_3*

(voir figure 4.8). Ce dernier est commandé par le signal d'entrée *Out_Inst_reg(1)*. La table de vérité du multiplexeur *Mux_3* et la configuration de la mémoire *Mem_B* sont illustrées dans le tableau B.4.

Tableau B.3. Table de vérité du multiplexeur *Mux_2* et configuration de *Mem_A*

Table de vérité du multiplexeur Mux_2					***Configuration de la mémoire Mem_A***
Valid_Result	***MMM_Status***	***Out_ Slv_reg (1)***	***Sel_Mem A***	***Sortie du Mux_2***	
0	0	0	1	*Addr_W_AB*	**Mode d'écriture :** Stockage de l'opérande *A*
0	0	1	0	*Addr_R_A*	**Mode de lecture :** lecture de l'opérande *A*
0	1	1	0	*Addr_W_T*	**Mode d'écriture :** Stockage de T
1	0	1	0	*Addr_R_T*	**Mode de lecture :** lecture de *T*

Tableau B.4. Table de vérité du multiplexeur *Mux_3* et configuration de *Mem_B*

Table de vérité du multiplexeur Mux_3			***Configuration de la mémoire Mem_B***
Out_Inst_reg(1)	***Sel_Mem_B***	***Sortie du Mux_3***	
0	0	*Addr_W_AB*	**La mémoire est désactivée**
0	1	*Addr_W_AB*	**Mode d'écriture** Stockage de *B*
1	0	*Addr_R_B*	**Mode de lecture** Lecture de *B*

Annexe C

Signaux de contrôle du circuit MMMs_Ctr et configuration des mémoires internes du module 2MulMong réalisé dans la seconde approche d'implémentation

Cette annexe est consacrée à la description des signaux d'entrées/sorties du circuit de contrôle *MMMs_Ctr* et à la présentation des configurations des mémoires implémentées dans le module *2MulMong*.

C.1. Signaux d'entrée/sortie du circuit de contrôle *MMMs_Ctr*

Le circuit *MMMs_Ctr* assure la gestion et la synchronisation des composants implémentés dans l'architecture interne du module *2MulMong* (voir figure 4.11). Il reçoit en entrées :

- Le signal d'initialisation *Out_Inst_reg(0),* utilisé pour activer/désactiver le fonctionnement du module *2MulMong*.
- Le signal d'horloge *Bus2IP_Clk.*
- Le signal *WFIFO2IP_RdAck* utilisé par le processeur pour indiquer la transmission d'une donnée de taille *k-bits* au prochain top d'horloge.
- Le signal *Reset_Op* généré par le circuit d'initialisation du composant *2MMMs_Core*. Ce signal est utilisé pour entamer l'exécution des opérations arithmétiques et la génération des adresses mémoires qui correspondent à la phase de lecture.
- Le signal *Out_Inst_reg(31)* correspondant au $w^{ème}$ bit du $j^{ème}$ chiffre *Z[j]* de l'exposant *Z.*
- Le signal *Out_Inst_reg(2),* utilisé par le processeur pour indiquer au module *2MulMong* la dernière itération de l'algorithme 8 ainsi que le début de la conversion du résultat vers la représentation classique des nombres.

Les signaux générés par le circuit de contrôle sont divisés en cinq catégories. Ces dernières sont définies selon les fonctionnalités implémentées dans le module *2MulMong*.

a. Bus d'adresses des mémoires

Cette catégorie est constituée par six bus d'adresse, utilisés pour l'adressage des mémoires durant le chargement et la lecture des données. Les bus d'adresses générés par *MMMs_Ctr* sont décrits dans le tableau C.1.

Tableau C.1. Bus d'adresses générés par le circuit de contrôle *MMMs_Ctr*

Bus d'adresse	Description
Addr_W	représente le bus d'adresse de la phase du stockage des données d'entrées dans leurs mémoires respectives.
Addr_R_A	utilisé pour l'adressage des mémoires *Mem_A1* et *Mem_A2* durant l'exécution des deux MMMs.
Addr_R_B	corresponds à l'adressage des mémoires *Mem_R*, *Mem_B1* et *Mem_B2* durant l'exécution des deux MMMs.
Addr_R_N	associé à l'adressage de la mémoire *Mem_N*, lors de l'exécution des opérations arithmétique.
Addr_Wi	utilisé pour l'adressage des mémoires *Mem_A1*, *Mem_B1*, *Mem_A2* et *Mem_B2* durant le stockage des résultats intermédiaires $S_{(i)}$ et $C_{(i)}$ de l'algorithme 8.
Addr_R_Y	bus adresse de la mémoire *Mem_B2*, utilisé lors du transfert du résultat *Y* de l'exponentiation modulaire vers processeur Microblaze.

b. Signaux de contrôle des deux unités arithmétiques *UAS* et *UAC*

Ces signaux assurent le contrôle des unités arithmétiques implémentées dans le module *2MulMong.* Leur description est détaillée dans le tableau C.2.

Tableau C.2. Signaux de contrôles des deux unités arithmétiques

Signaux	Description
En_qi, Reset_cy12, Reset_cy34	Le premier signal permet de maintenir les $q_{(i)}$ calculés dans les deux unités arithmétiques constants, durant l'exécution d'une itération *(i)* de l'algorithme 7. Les deux autres sont utilisés pour initialiser les retenus au début de chaque itération *(i).*

c. Signaux de contrôle des mémoires et des registres *N'_Reg et Y_Reg*

Cette catégorie est dédiée aux contrôles des mémoires et des deux registres *N'_Reg* et *Y_Reg.* Ces signaux sont présentés dans le tableau C.3.

Tableau C.3. Signaux de contrôles des mémoires et des registres *N'_Reg* et *Y_Reg*.

Signal de sortie	Description
SelMem_R, SelMem_N, SelMem_A1	Ces signaux sont actifs à l'état haut. Leur tâche consiste en la sélection des mémoires *Mem_R, Mem_N* et *Mem_A1* durant le chargement respectif de : $R^2 \bmod N$, du modulo *N* et du message *X*.
SelReg_N'	Ce signal est actif à l'état haut. Il est utilisé pour activer l'horloge du registre *N'_Reg,* lors de la réception de la constante *N'*.
SelA2_B2	Ce signal est actif à l'état bas. Il permet de désactiver l'horloge des mémoires *Mem_A2* et *Mem_B2*, si le $w^{ème}$ bit z_w^j du $j^{ème}$ chiffre *Z[j]* de *Z*, est un *"0"* logique. Ce signal est généré par le circuit *MMMs_Ctr* en fonction du signal d'entrée *Out_Inst_reg(31)*, utilisé par Microblaze pour transmettre z_w^j au module *2MulMong*.
MStatus	Signal actif à l'état haut. Il permet de valider le stockage des résultats intermédiaire $S_{(i)}$ et $C_{(i)}$.
Valid_RExp	Ce signal est à l'état haut durant la lecture du résultat *Y*. Il permet d'une part de sélectionner le bus d'adresse *Addr_R_Y* à l'entrée *Addr* de la mémoire *Mem_B2*, lors du transfert du résultat *Y* vers le processeur ; d'autre part, d'activer le registre de sortie *Y_Reg*

d. Signaux de sélection des opérandes

Ces signaux assurent la sélection des opérandes d'entrée aux deux unités arithmétiques *UAS* et *UAC*, en utilisant les multiplexeurs *Mux_1*, *Mux_2* et *Mux_3* (voir figure 4.11). A noter que la sélection en question est effectuée en fonction des étapes d'exécution de l'algorithme 8. Ces signaux sont définis dans le tableau C.4.

Tableau C.4. Signaux de sélection des opérandes.

Signal	Description
Sl_R_B1	Ce signal assure la commande du multiplexeur *Mux_1* pour sélectionner la constante $R^2 \bmod N$ ou le résultat $S_{(i)}$ de l'élévation au carré, à l'entrée *B* de l'unité arithmétique *UAS.*
Sl_1_A2	Ce signal assure la commande du multiplexeur *Mux_2* pour sélectionner à l'entrée *A* de l'unité arithmétique *UAC, " 1 "* ou $S_{(i)}$.
Sl_R_B2	Ce signal assure la commande du multiplexeur *Mux_3* pour la sélection à l'entrée *B* de l'unité arithmétique *UAC*, $R^2 \bmod M$ ou $C_{(i)}$.

e. Signaux de contrôle des registres d'état *St_Registre1* et *St_Registre2* de l'interface *IPIF*

Ces signaux sont constitués par deux bus de taille *32-bits*. Ils sont utilisés par le module *2MulMong* pour indiquer au processeur Microbalze à travers l'interface *IPIF*, l'état du chargement des données et l'état d'exécution des opérations arithmétiques. Ces bus sont définis dans le tableau C. 5.

Tableau C.5. Signaux de contrôles des registres d'état de l'interface *IPIF*

Bus	**Description**
In_reg1	Bus d'entrée du registre d'état *St_Registre1*. Il est utilisé par le module *2MulMong* pour indiquer son état lors de la réception des données. Les états de ce bus durant le chargement des données et lors de l'exécution de la fonction l'exponentiation modulaire, sont respectivement *0xFFFFFFFF* et *0x00000000*.
In_reg2	Bus d'entrée du registre d'état *St_Registre2*. Ce bus est utilisé par le module *2MulMong* pour indiquer l'état des deux unités arithmétiques lors de l'exécution des deux MMMs. L'état de ce bus est *0x00000000,* durant le chargement des données et pendant l'exécution de l'algorithme 7. Une fois la $n^{ème}$ itération de cet algorithme atteinte, le circuit de contrôle *MMMs_Ctr* transforme l'état de ce bus à *0x0000FFFF* pour indiquer au processeur, la fin d'une itération *(w)* de l'algorithme d'exponentiation modulaire (algorithme 8).

C.2. Configurations des mémoires implémentées dans le module 2*MulMong*

Dans la deuxième approche d'implémentation, le module *2MulMong* est utilisé pour calculer en parallèle les deux MMMs de chaque itération *(w)* de l'algorithme 8. L'exécution de l'exponentiation modulaire $Y=X^Z \mod N$ dans cette approche, nécessite quatre types d'accès aux mémoires *Mem_R*, *Mem_N*, *Mem_A1*, *Mem_B1*, *Mem_A2* et *Mem_B2*, implémentées dans le module *2MulMong* (voir figure 4.11). Ces accès sont résumés dans les points suivants :

- Un accès lors du chargement des données d'entrée.
- Un accès durant l'exécution des opérations arithmétiques des deux MMMs.
- Un accès lors du stockage des résultats intermédiaires $S_{(i)}$ et $C_{(i)}$ de l'algorithme 8.
- Un accès pendant le transfert du résultat *Y* vers le processeur Microblaze.

a. Configuration de la mémoire *Mem_R*

La mémoire *Mem_R* assure le stockage de la valeur de R^2 *mod N*. Elle est configurée en écriture et en lecture respectivement durant le chargement de la donnée en question et lors de la conversion du message *X* et de *1* dans le domaine de Montgomery (première étape de l'algorithme 8). La sélection des adresses mémoire qui correspondent aux deux configurations, est assurée par le multiplexeur *Mux_6*. Ce dernier est commandé par le signal d'entrée *Reset_Op* (voir figure 4.11). La sélection de la mémoire *Mem_R2* en phase d'écriture est assurée par le signal *SelMem_R*, généré par le circuit *MMMs_Ctr*. La table de vérité du multiplexeur *Mux_6* et la configuration de cette mémoire sont présentées dans le tableau C.6.

Tableau C.6. Table de vérité du multiplexeur *Mux_6* et configuration de *Mem_R*

Table de vérité du multiplexeur Mux_6			***Configuration de la mémoire Mem_R***
Reset_Op	***SelMem_R***	***Sortie du Mux_6***	
0	0	*Addr_W*	**La mémoire est désactivée**
0	1	*Addr_W*	**Mode d'écriture** Stockage de R^2 *mod N*
1	0	*Addr_R_B*	**Mode de lecture** Lecture de R^2*mod N*

b. Configuration de la mémoire *Mem_N*

La mémoire *Mem_N* est utilisée pour le stockage du modulo *N*. La sélection de cette mémoire lors du chargement de *N* est réalisée par *SelMem_N*, fourni *MMMs_Ctr*. La sélection des bus d'adresse *Addr_W* et *Addr_R_N* assurant l'adressage de cette mémoire respectivement en mode d'écriture et en mode de lecture, est assurée par le multiplexeur *Mux_5*. Ce dernier est commandé par le signal *Reset_Op*. La table de vérité du multiplexeur *Mux_5* et la configuration de la mémoire *Mem_N* sont présentées dans le tableau C.7.

Tableau C.7 Table de vérité du Multiplexeur *Mux_5* et configuration de *Mem_N*

Table de vérité du multiplexeur Mux_5			***Configuration de la mémoire Mem_N***
Reset_Op	***SelMem_N***	***Sortie du Mux_5***	
0	0	*Addr_W*	**La mémoire est désactivée**
0	1	*Addr_W*	**Mode d'écriture** Stockage du modulo *N*
1	0	*Addr_R_N*	**Mode de lecture** Lecture de modulo *N*

c. Configuration de la mémoire *Mem_A1*

La mémoire *Mem_A1* est consacrée au stockage du message *X* et des résultats intermédiaires $S_{(i)}$ de l'algorithme 8. La sélection entre ces derniers à l'entrée de cette mémoire est assurée par le multiplexeur *Mux_4*. Son adressage pour le stockage de *X*, de $S_{(i)}$ et leur lecture, est réalisé en utilisant respectivement les bus d'adresse *Addr_W*, *Addr_Wi* et *Addr_R_A*, générés par *MMMs_Ctr*. Ces adresses sont sélectionnées par l'intermédiaire du multiplexeur *Mux_7*, commandé par les deux signaux *Reset_Op* et *MStatus* (voir figure 4.11). La sélection de cette mémoire lors du stockage et de lecture des données, est effectuée en activant son horloge uniquement pendant l'exécution de ces deux opérations. Le signal permettant de réaliser cette sélection, est généré par la combinaison des signaux *SelMem_A1 et Reset_Op* dans une porte *OR*. Sa configuration en mode d'écriture ou de lecture est effectuée en combinant dans une porte *OR* l'inverse du signal *Reset_Op (Not Reset_Op)* et le signal *MStatus*. La table de vérité du multiplexeur *Mux_7* et la configuration de cette mémoire sont présentées dans le tableau C.8

Tableau C.8. Table de vérité du multiplexeur *Mux_7* et configuration de *Mem_A1*

Table de vérité du multiplexeur Mux_7				***Configuration de la mémoire Mem_A1***
Reset_Op	***MStatus***	***SelMem_A1***	***Sortie du Mux_7***	
0	0	0	*Addr_W*	***Mem_A1* est désactivée**
0	0	1	*Addr_W*	**Mode d'écriture** Stockage du message *X*
1	0	0	*Addr_R_A*	**Mode de lecture** Lecture du message *X* ou de $S_{(i)}$
1	1	0	*Addr_Wi*	**Mode d'écriture** Stockage de $S_{(i)}$

d. Configuration de la mémoire *Mem_B1*

La mémoire *Mem_B1* est dédiée au stockage du résultat intermédiaire $S_{(i)}$ de l'algorithme 8, calculé par l'unité arithmétique *UAS*. Elle reçoit sur son bus de donnée d'entrée, le bus de sortie de la mémoire *FIFO_S*. L'adressage de *Mem_B1* en phases de lecture et d'écriture est réalisé respectivement par les bus d'adresse *Addr_B_R* et *Addr_Wi*. Leur sélection est assurée par le multiplexeur *Mux_8*. Ce dernier est commandé par le signal *MStatus* (voir figure 4.11). La table de vérité *Mux_8* et la configuration de la mémoire *Mem_B1* sont montrées sur le tableau C.9.

Tableau C.9. Table de vérité du multiplexeur *Mux_8* et configuration de *Mem_B1*

Table de vérité du multiplexeur Mux_8		***Configuration de la mémoire Mem_B1***
MStatus	***Sortie du Mux_8***	
0	*Addr_R_B*	**Mode de lecture** Lecture du résultat intermédiaire $S_{(i)}$
1	*Addr_Wi*	**Mode d'écriture** Stockage du résultat intermédiaire $S_{(i)}$

e. Configuration de la mémoire *Mem_A2*

La mémoire *Mem_A2* est consacrée au stockage du résultat de l'élévation au carré $S_{(i)}$ de l'algorithme 8. Elle reçoit en entrée le bus de sortie de la mémoire *FIFO_S*. Son adressage en phases de lecture et d'écriture est assuré par les deux bus d'adresse *Addr_A_R* et *Addr_Wi.* Ces derniers sont sélectionnés par le multiplexeur *Mux_9* en utilisant le signal *MStatus* comme signal de commande (voir figure 4.11). La table de vérité résumant la configuration de la mémoire *Mem_A2* en fonction des deux signaux *MStatus* et *SelA2_B2* est illustrée dans le tableau C.10.

Tableau C.10. Table de vérité du multiplexeur *Mux_9* et configuration de *Mem_A2*

Table de vérité du multiplexeur Mux_9			***Configuration de la mémoire Mem_A2***
MStatus	***SelA2_B2***	***Sortie du Mux_9***	
0	0	Cas indéfini	Cas indéfini
0	1	*Addr_A_R*	**Mode de lecture** Lecture du résultat intermédiaire $S_{(i)}$
1	0	*Addr_Wi*	***Mem_A2* est désactivée**
1	1	*Addr_Wi*	**Mode d'écriture** Stockage du résultat intermédiaire $S_{(i)}$

SelA2_B2 est utilisé pour désactiver le stockage de $S_{(i)}$ dans *Mem_A2,* si le $w^{ème}$ bit z_w^j du $j^{ème}$ chiffre *Z[j]* de l'exposant *Z* est un *"0"* logique. Ce signal est défini dans le tableau C.3.

f. Configuration de la mémoire *Mem_B2*

La mémoire *Mem_B2* est utilisée pour :

- Le stockage du résultat intermédiaire $C_{(i)}$ de l'algorithme 8.
- Le stockage et le transfert vers le processeur Microblaze du résultat *Y* de l'exponentiation modulaire.

Elle reçoit sur son bus de donnée d'entrée le bus de sortie de la mémoire *FIFO_C*. La sélection de son adressage est assurée par le multiplexeur *Mux_10*. Cet adressage correspond:

- Au stockage de $C_{(i)}$ et sa lecture durant le calcul de $C_{(i)}$. Les bus d'adresse utilisés pour réaliser ces deux opérations sont respectivement *Addr_Wi* et *Addr_R_B*.
- A la lecture du résultat final $Y=X^Z \ mod \ N$ lors de son transfert vers le processeur. Le bus d'adresse associé à cette étape est *Addr_R_Y*.

Le contrôle du multiplexeur *Mux_10* et de la mémoire *Mem_B2* pour synchroniser les opérations d'écriture et de lecture, est assuré par les trois signaux *MStatus*, *SelA2_B2* et *Valid_RExp* (voir figure 4.11). La table de vérité du multiplexeur *Mux_10* et la configuration de la mémoire *Mem_B2* sont présentées dans le tableau C.11.

Tableau C.11. Table de vérité du multiplexeur *Mux_10* et configuration de *Mem_B2*

Table de vérité du multiplexeur Mux_10				***Configuration de la mémoire Mem_A2***
Valid_RExp	***MStatus***	***SelA2_B2***	***Sortie du Mux_10***	
0	0	1	*Addr_R_B*	**Mode de lecture** Lecture de $C_{(i)}$
0	1	0	Cas indéfini	***Mem_B2* est désactivée**
0	1	1	*Addr_Wi*	**Mode d'écriture** Stockage de $C_{(i)}$
1	0	1	*Addr_R_Y*	**Mode de lecture** Lecture de *Y*

Bibliographie

[1] C. Paar and J. Pelzl, "*Understanding Cryptography*", Springer-Verlag, 2010.

[2] B.Schneier, "*Applied Cryptography Protocols, Algorithms, and Source in C*", Wiley, Second edition, 1995.

[3] B.Schneier, "*Cryptographie Appliquée, Algorithme, Protocoles et code Sources en C*", Traduit par : L.Viennot, Vuibert informatique, 1996.

[4] A.J. Menezes, Paul C. V. Oorschot and S.A. Vanstone, "*Handbook of Applied Cryptography*", CRC Press, 1996.

[5] J.P Deschamps, G.J.A Bioul and G.D. Sutter, "*Synthesis of Arithmetic Circuits FPGA, ASIC, and Embedded Systems*", Wiley Inter Science, 2006.

[6] T. Plantard, "*Arithmétique Modulaire pour la Cryptographie*", Thèse PhD, Université Montpellier II, Laboratoire d'Informatique, de Robotique et de Microélectronique, 2005.

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

[8] C.Maxfield*, "FPGAs World Class Desogn",* Elsivier, 2009.

[9] F.Vahid and T. Givargis, "*Embedded System Design: A Unified Hardware/Software Approach*", Department of Computer Science and Engineering, University of California, 1999.

[10] W. Diffie and M.E. Hellman, "*New Directions in Cryptography*", IEEE Transactions on Information Theory, vol.IT22, No.6, pp.644–654, 1976.

[11] R.L. Rivest, A. Shamir, and L. Adleman, "*A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*", Communication. ACM, vol 21, No.2, pp.120-126, 1978.

[12] N. Koblitz, "*Introduction to Elliptic Curves and Modular Forms*", Graduate Texts in Mathematics, Vol. 97, Springer, 1984.

[13] Ç.K. Koç, "*High-Speed RSA Implementation*", RSA Laboratories, RSA Data security, Version 2.0, 1994.

[14] Ç.K. Koç "*RSA Hardware Implementation*", RSA Laboratories, RSA Data security, Version 1.0, 1995.

[15] E. Oswald "*Introduction to Elliptic Curve Cryptography*", Institute for Applied Information Processing and Communication A-8010, Graz, Austria, 2005.

[16] A.A.A. Gutub and M.K. Ibrahim, "*High Radix Parallel Architecture for Gf(P) Elliptic Curve Processor*", In Proc . IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 2, pp. 625-628, Hong Kong, 2003.

[17] X. Guo and P. Schaumont, "*Optimized System-on-Chip Integration of a Programmable ECC Coprocessor*", Journal ACM Transactions on Reconfigurable Technology and Systems, vol. 4, No 1, 2010.

[18] *"MicroBlaze Processor Reference Guide"*, UG081 (v13.2). *http://china.xilinx.com/support/documentation/sw_manuals/xilinx13_2/mb_ref_guide.pdf*

[19] *"Genesys Board, Reference Manual"*, Revision, 2012. *http://www.digilentinc.com/Data/Products/GENESYS/Genesys_rm.pdf*

[20] P.Montgomery, *"Modular Multiplication Without Trial Division"*, Mathematics of Computation, vol.44, pp.519-521, 1985.

[21] S. S. Ghoreishi, M.A. Pourmina, H. Bozorgi and M. Dousti, *"High Speed RSA Implementation Based on Modified Booth's Technique and Montgomery's Multiplication for FPGA Platform*", In Proc. CENICS '09, pp.86-93, 2009.

[22] G.W.Bewick, "*Fast Multiplication: Algorithms and Implementation*", PhD these, Stanford University, 1994.

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

[24] R. Verma, M. Dutta and R.Vig, "*Modified Montgomery for RSA Cryptosystem*", International Journal of Computer, Information Science and Engineering, vol.7, No.12, pp.64–68, 2013.

[25] C. McIvor, M. McLoone, J. McCanny, A. Daly and W. Marnane, "*Fast Montgomery Modular Multiplication and RSA Cryptographic Processor Architectures*", In Proc. 37th Annual Asilomar Conference on Signals, Systems and Computers, pp.379–384, 2003.

[26] A. F. Tenca and C. K. Koc, "*A Scalable Architecture for Montgomery Multiplication*", In Proc. Workshop on Cryptographic Hardware and Embedded Systems, pp.94–108, 1999.

[27] S.B.Ors, L.Batina, B.Preneel and J. Vandewalle, "*Hardware Implementation of a Montgomery Modular Multiplier in a Systolic Array*", In Proc. 17th International Parallel & Distributed Processing Symposium, IEEE Computer Society, 2003.

[28] N.Nedjah and L.M.Mourelle, "*A Review of Modular Multiplication Methods and Respective Hardware Implementation*", Informatica, vol.30, pp.111-129, 2006.

[29] G.Perin, D.G.Mesquita, and J.B.Martins, "*Montgomery Modular Multiplication on Reconfigurable Hardware: Systolic versus Multiplexed Implementation*", International Journal of Reconfigurable Computing, pp.1-10, 2011.

[30] F. Bernard, *"Etude des Algorithmes Arithmétiques et leur Implémentation Matérielle"*, Thèse PhD, Université de Saint-Denis-paris 8, 2007.

[31] *"Virtex-5 FPGA Xtreme DSP Design Considerations User Guide"*, UG193 (v3.5), 2012.
http://www.xilinx.com/support/documentation/user_guides/ug193.pdf

[32] B. Song, K. Kawakami, K. Nakano and Y. Ito, "*An RSA Encryption Hardware Algorithm Using a Single DSP Block and Single Block RAM on the FPGA*", In Proc. First International Conference on Networking and Computing, pp.140-147, 2010.

[33] E.Oksuzoglu and E.Savaş, *"Parametric Secure and Compact Implementation of RSA on FPGA"*, In Proc. Reconfigurable Computing and FPGAs, pp.391-396, 2008.

[34] Z. Wang, Z. Jia, L. Ju and R. Chen, "*ASIP-Based Design and Implementation of RSA for Embedded Systems*", In Proc. 14th IEEE International Conference on High Performance Computing and Communications pp.1375-1382, 2012.

[35] A.A.A. Gutub and F.A.A. Khan, "*Hybrid Crypto Hardware Utilizing Symmetric-Key and Public-Key Cryptosystems*", In Proc. International Conference on Advanced Computer Science Applications and Technologies, pp.116-121, Kuala Lumpur, 2012.

[36] M. K.Hani, H. Y. Wen and A. Paniandi, "*Design and Implementation Of a Private and Public Key Crypto Processor for Next Generation IT Security Applications*", Malaysian Journal of Computer Science, vol.19, No.1, pp.29 – 45, 2006.

[37] "*Nios Soft Core Embedded Processor*", Data Shett, 2000, version 1. *http://extras.springer.com/2001/978-3-662-04615-9/DS/DS_EXC~2.pdf*

[38] R. Lu, J.Han, X.Zeng, Q.Li, L.Mai and J.Zhao, "*A Low-Cost Cryptographic Processor for Security Embedded System"*, In Proc. 13th Asia and South Pacific Design Automation Conference, pp.113-114, Korea, 2008.

[39] M. Wu, X. Zeng, J. Han, Y. Wu and Y. Fan, "*A High-Performance Platform-Based SoC for Information Security*", In Proc. Asia and South Pacific Design Automation Conference, pp.122-123, Japan,2006.

[40] L. Uhsadel, M. Ullrich, I.Verbauwhede and B. Preneel, "*Interface Design for Mapping a Variety of RSA Exponentiation Algorithms on a HW/SW Co-design Platform*", In Proc. 23rd International Conference on Application-Specific Systems, Architectures and Processors, pp.109 – 116, 2012.

[41] M.Šimka, V.Fischer and M. Drutarovský, "*Hardware-Software Co-design in Embedded Asymmetric Cryptography Application – a Case Study"*, In Proc. Field-Programmable Logic and Applications (FPL), pp.1075–1078, 2003.

[42] P.Hämäläinen, N. Liu, M. Hännikäinen, and T. D. Hämäläinen, "*Acceleration of Modular Exponentiation on System-on-a-Programmable-Chip*", In Proc. International Symposium on System-on-Chip, pp.14-17, 2005.

[43] "*ARM922T Technical Reference Manual"*, ARM, 2001. *http://infocenter.arm.com/help/topic/com.arm.doc.ddi0184b/DDI0184.pdf*

[44] A. Salman, M. Rogawski and J.P. Kaps, "*Efficient Hardware Accelerator for IPSec based on Partial Reconfiguration on Xilinx FPGAs*", In Proc, International Conference on Reconfigurable Computing and FPGAs, pp.242-248,2011.

[45] R. Paul, S.S.C.Pal and S.Sau, "*Novel Architecture of Modular Exponent on Reconfigurable System*", In Proc. Students Conference on Engineering and Systems, pp.1-6, 2012.

[46] N.Nedjah and L.M.Mourelle, *"Software/Hardware Co-Design of Efficient and Secure Cryptographic Hardware*", Journal of Universal Computer Science, vol. 11, No.1, pp.66-82, 2005.

[47] *"Introduction à la Cryptographie, Logiciel de Chiffrement et de Déchiffrement PGP (Pretty Good et Pretty)"*, Version 6.5.1, 1999.

[48] J. Daemen, V. Rijmen, "*AES Proposal: The Rijndael Block Cipher*", version 2, 1999.

[49] "*Announcing the Advanced Encryption Standard (AES)* ", FIPS PUBS 197, NIST, 2001.

[50] S. Dietrich, "*Cryptography and Integer Factorization*", Ecole d'Automne, Cerist, Alger, 2007.

[51] A. de Vries. "*The Ray Attack, an Inefficient Trial to Break RSA Cryptosystems*", University of Applied Sciences, Germany, 2003.

[52] E. Landquist, "*The Quadratic Sieve Factoring Algorithm*", MATH 488: Cryptographic Algorithms, 2001.

[53] C. Seibert, "*Integer Factorization using the Quadratic Sieve*", University of Minnesota, 2011.

[54] A. Mousa, "*Sensitivity of Changing the RSA Parameters on the Complexity and Performance of the Algorithm*", Journal of Applied Sciences vol.5, No.1, pp.60-63, 2005.

[55] S. Boldo and M. Daumas, "*A Simple Test Qualifying the Accuracy of Horner's Rule for Polynomials*", Rapport de Recherche N° 4004, INRIA, 2004.

[56] N. Nedjah, L.M. Mourelle, M. Santana and S. Raposo, "*Massively Parallel Modular Exponentiation Method and its Implementation in Software and Hardware for High Performance Cryptographic Systems",* IET Computers & Digital Techniques, vol.6, Issue.5, pp.290–301, 2012.

[57] M.J. Flynn and S.F. Oberman "*Advanced Computer Arithmetic Design*" A Wiley Inter Science Publication John Wiley & Sons, INC, 2001.

[58] H.Kim and S.Lee, *"Design and Implementation of a Private and Public Key Crypto Processor and Its Application to a Security System",* IEEE Transactions on Consumer Electronics, vol.50, No.1, 2004.

[59] N.Mantens, *"Secure and Efficient Coprocessor Design for Cryptographic Applications on FPGAs"*, PhD thesis, University of Leuven, Belgium, 2007.

[60] H. Orup *"Exponentiation, Modular Multiplication and VLSI Implementation of High-Speed RSA Cryptography",* Ph.D Thesis, Department of Computer Science, University of Aarhus, Denmark, August 1995.

[61] C.D. Walter, "*Precise Bounds for Montgomery Modular Multiplication and Some Potentially Insecure RSA Moduli*", Lecture Notes in Computer Science, Topics in Cryptology - CT-RSA, pp.30–39, 2002.

[62] L.Batina, S.B.Ors, B.Preneel and J. Vandewalle,*"Hardware Architectures for Public Key Cryptography*", Elsevier, Intergration. The VLSI journal, vol. 34, pp.1–64, 2003.

[63] M.Issad, B.Boudraa and M.Anane *"Hardware Implementation of Montgomery Multiplication for Embedded Cryptosystems*", the 9th International Conference on Design & Technology of Integrated Systems in Nanoscale Era, Santorini, Greece, 2014.

[64] M.Issad, B.Boudraa and M.Anane, *"Efficient Hardware Implementation of Montgomery Multiplication for Embedded Cryptosystems",* International Congress on Telecommunication and Application'14 University of A.Mira, Bejaia, Algeria, 2014.

[65] M.Issad, B. Boudraa, M. Anane, S. Seddiki, *"FPGA Implementation of Modular Exponentiation Using Single Modular Multiplier*", International Conference on Circuits, Systems, Signal Processing, Communications and Computers, Venice, Italy, 2014.

[66] M.Issad, B.Boudraa, M.Anane and N.Anane, "*Software/Hardware Co-Design of Modular Exponentiation for Efficient RSA Cryptosystem*", Journal of Circuits, Systems, and Computers, vol.23, No.3, pp.1-30, Mars 2014.

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

[68] B. Daya "*Rapid Prototyping of Embedded Systems Using Field Programmable Gate Arrays*", Bachelor of Science in Electrical Engineering, Spring, 2009.

[69] A.Fraboulet, F.Jumel, L.Morel and T.Risset, *"Conception et Programmation de Systèmes Embarqués",* Lab CITI, INSA de Lyon.

[70] *"Virtex-5 User Guide",* UG190 (v1.2), 2006.
http://www.ece.iastate.edu/~zambreno/classes/cpre583/2006/documents/xilinx/ug190.pdf

[71] *"ISE Design Suite Software Manual*", UG631 (v 13.1), 2011.
http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/irn.pdf

[72] *"Embedded System Tools Reference Manual EDK",* UG111 (v13.2), 2011.
http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/est_rm.pdf

[73] H.P. Rosinger, *"Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link (FSL) Channel",* XAPP529, 2004.

[74] "*PLB IPIF*", DS448 (v2.02a), 2005.
http://www.xilinx.com/support/documentation/ip_documentation/plb_ipif.pdf

[75] *"Tutorial: Designing Custom OPB Slave Peripherals for MicroBlaze*", 2002.
http://www.cs.columbia.edu/~sedwards/classes/2004/4840/opb_tutorial.pdf

[76] "*EDK OS and Libraries Reference Guide*", Embedded Development Kit EDK 6.2i, UG114 (v3.0), 2004.
http://www.xilinx.com/ise/embedded/edk_libs_ref_guide.pdf

[77] "*Standalone Board Support Package*", EDK 9.1i, 2007.
http://www.xilinx.com/ise/embedded/edk91i_docs/standalone_v1_00_a.pdf

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

[79] *"Using EDK to Run Xilkernel on a MicroBlaze Processor",* Example Design, UG758, 2010.
http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_4/ug758.pdf

[80] *"Maple 9.5 Getting Started Guide",* Maplesoft, a division of Waterloo Maple Inc. 2004.

[81] "*Modelsim SE User's Reference Manual",* Software version, 2010.

[82] M.Issad, B.Boudraa and M.Anane, *"Flexible Implementation of RSA Cryptosystem Using PSoC Platform",* 1ère Doctoriales sur les Télécommunications et le Traitement de l'Information, USTHB, 2013.

[83] M.Issad, B.Boudraa and M.Anane, *"Enhanced Modular Exponentiation on PSoC Platform for RSA Cryptosystem*", the 23rd IEEE International Symposium on Industrial Electronics, Istanbul, Turkey, 2014.

[84] M.Issad, B.Boudraa and M.Anane, *"PSoC Platform for RSA Public Key Cryptosystem"*, International Congress on Telecommunication and Application'14 University of A.Mira, Bejaia, Algeria, 2014.

[85] *"Xilinx Device Drivers Documentation",* Generated on 24 Jun 2004 for Xilinx Device Drivers.
http://www.xilinx.com/ise/embedded/edk6_2docs/xilinx_drivers.pdf

[86] M.Issad, M.Anane, B.Boudraa et N.Anane *"Implémentation du crypto système RSA dans un environnement SOPC"*, 6ème séminaire sur les systèmes de détection Architecture et technologies, Alger, 2014.

[87] *"Virtex-5 Family Overview",* DS100 (v5.0), 2009.
http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf

[88] J.M.Muler, "*Arithmétique des Ordinateurs* ", Masson, 1989.

Résumé: Dans cette thèse, on s'intéresse à l'implémentation logicielle / matérielle du crypto système à clé publique RSA (Rivest, Shamir and Adleman). L'opération principale dans ce crypto système est l'Exponentiation Modulaire (EM) qui n'est rien d'autre que le calcul itératif de Multiplications Modulaire (MMs). Notre objectif dans ce travail consiste à implémenter la fonction EM sur circuit FPGA. Le processeur Microblaze de Xilinx est utilisé pour la flexibilité. Nos travaux de recherche sont basés sur l'utilisation de l'algorithme R2L (Right to Left) qui repose sur le calcul parallèle de deux MMs. Cet algorithme est souvent recommandé pour améliorer le délai d'exécution de l'EM. Par ailleurs, lorsque l'optimisation des ressources matérielles sur le support d'implémentation est une contrainte, l'algorithme en question peut être exécuté séquentiellement, en utilisant seulement un seul multiplieur modulaire comme un composant matériel, autour du processeur. De ce fait, le temps d'exécution de l'EM devient dépendant: des performances du multiplieur, de la chaine de bit non-zéro de l'exposant et du débit de la liaison assurant la communication entre le processeur et le composant. Dans le but d'atteindre le meilleur compromis entre la vitesse d'exécution, les ressources matérielles et la flexibilité du système, nous proposons trois approches d'implémentations. La première exploite un accélérateur matériel dédié à l'exécution d'une seule MM. La seconde approche est basée sur deux stratégies d'optimisation, où deux MMs sont implémentées en parallèle dans un composant matériel et des mémoires locales sont intégrées dans ce dernier, près des unités arithmétiques. La troisième approche est une implémentation purement logicielle. Les résultats d'implémentation sur le circuit FPGA XC5VLX50T-1, ont montré qu'en utilisant le partitionnement logiciel/matériel de la seconde approche, le chiffrement pour une taille de clé de 1024-bits est exécuté en 15,14 ms. Le nombre de slices occupés sur le circuit FPGA est de 2315 slices.

Mots clés —RSA, exponentiation modulaire, multiplication modulaire, multiplication de Montgomery, FPGA, PSoC, SoC, Microblaze.

Abstract: In this thesis, we present an implementation of RSA (Rivest, Shamir and Adleman) public key cryptosystem based on Hardware /Software co-design. The main operation of RSA is the Modular Exponentiation (ME) which is performed by repeated Modular Multiplications (MMs). The researches presented in this thesis are based on the R2L (Right to Left) algorithm for the implementation of the ME as a Programmable System on Chip. The processor MicroBlaze of Xilinx is used for flexibility. The R2L method is often suggested to improve the timing performance, since it is based on parallel computations of MMs. However, if the optimization of hardware resources is a constraint, this method can be executed sequentially using a single modular multiplier as a custom component. Consequently, the execution time of the ME becomes dependent of: the capability of the component to perform the MMs, the non-zero bit string of the exponent and the communication link between the processor and the custom component. In order to achieve the best tradeoff between area, speed and flexibility, we propose three implementations. The first one takes benefit of a hardware accelerator dedicated to the MM execution. The second one is based on a dual strategy. Two parallel modular multipliers are implemented within a custom component and local memories are used close to the arithmetic units to minimize the communication link influence. The last one is a software solution. The implementation on the FPGA circuit XC5VLX50T-1, shows that the application to RSA 1024-bits, the ME runs in 15,14ms, while using only 2315 slices.

***Keywords*— RSA, modular exponentiation, modular multiplication, Montgomery multiplication, FPGA, PSoC, SoC, Microblaze.**

ملخص: الهدف من هذه الأطروحة، يتمثل في زرع نظام التشفير RSA (Rivest, Chamir andAdleman) العملية الرئيسية في هذا النظام هي عملية الأس الترديدي التي تستند على عملية الضرب الترديدي. هدفنا زرع عملية الأس في شريحة مكونة من نظام مبرمج FPGA و ذلك باستعمال معالج المعلومات Microblaze لـ Xilinx .من أجل تحسين خصائص تشغيل هذه العملية، أعمال البحث التي أنجزت في هذه الأطروحة تستند على خوارزمية R2L (Right-to- Left) التي تتركز أساسا على حساب عمليتين للضرب الترديدي على التوازي. بالرغم من أن هذه الطريقة قد طورت لتحسين زمن حساب الأس الترديدي، لكن عندما تكون مساحة الزرع مقيدة، هاتين العمليتين يمكن انجازهما عل التسلسل. و ذلك باستعمال كتلة IP (IntellectualProperty) بجوار Microblaze .دورIP هو حساب عملية الضرب الترديدي. باستعمال هذه الطريقة، خصائص زرع الأس الترديدي تصبح معتمدة على: خصائص الـ IP، عدد الأصفار في الأس و مميزات وسيلة الاتصال التي تربط الـ IP بـ Microblaze. هدفنا من هذه الأطروحة هو زرع ثلاث طرق لحساب عملية الأس الترديدي و ذلك من أجل التوصل إلى أحسن الخصائص التي تتعلق بين زمن حساب هذه العملية، مساحة التشغيل و مرونة النظام. الطريقة الأولى تستند على استعمال IP تتكون من عملية واحدة التي تمثل الضرب الترديدي. الطريقة الثانية تعتمد على زرع عمليتين للضرب الترديدي على التوازي داخل الـ IP. الطريقة الثالثة تعتمد على زرع برنامج يقوم بحساب الأس الترديدي. نتائج التشفير بمفتاح ذو مقاس 1024-bits، بينت بأن زمن حساب الأس الترديدي هو15,14 ms .المساحة المشغولة بداخل دارة FPGA قدرت بـ 2315 slices.

كلمات مفتاح— نظام التشفيرRSA، الأس الترديدي، الضرب الترديدي، الضرب مونقمري.