République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Ecole Nationale Polytechnique
Departement de l'Electronique
Laboratoire de Traitement de Signal

Mémoire de Magister en Electronique

Option : Traitement de Signal et Communication

Présenté par :
Mme HENTABLI Wahiba
Ingénieur d'état en Electronique de l'USTHB

Intitulé

Etude et mise en œuvre d'un IP-Core RSA

Soutenu publiquement le **30/09/2012** devant le jury composé de :

President :	Aksas Rabia	Professeur	ENP
Rapporteur :	Sadoun Rabah	Professeur	ENP
Examinateurs :	Haddadi Mourad	Professeur	ENP
	Mehenni Mohamed	Professeur	ENP
	Belouchrani Adel	Professeur	ENP

ENP 2012
Laboratoire Ecole Nationale Polytechnique (ENP)
10, Avenue des Freres Oudek, Hassen Badi, BP.182, 16200 El-Harrach, Alger, Algerie
www.enp.edu.dz

ملخص

عملنا يتضمن اختزال خوارزمية نظام التشفير RSA على شريحة FPGA و تهايتها للحل Soc .لاختزال خوارزمية RSA اخترنا طريقة MSB ثنائي الأسي التي تعتمد على نظام Montgomery الذي يساهم في اسراع وحدات الأسية. هذا النظام يعتمد على انجاز طريقة التوازي التي تحقق عمليتان ضرب في ان واحد مما يسمح بخفض وقت وظيفه الشفرة و فك الشفرة.

من اجل مقارنة نتائج التصميم و التحليل المكتسبة باستعمال بيئة التطوير ISE Xilinx و بيئة التطوير Microblaze EDK , تكتمل المقارنة ايضا عند تصميم نظام RSA بستعمال مكتبة GMP تحتوي على الارقم ذات الحجم الكبير

الكلمات المفتاحية : علم التشفير , RSA , Montgomery , Soc , FPGA , الخلية الملكية الفكرية, Microblaze ,EDK ,GMP , FSL. (نظام التشفير RSA محمول من الموقع Open-Core)

Résumé

Notre mémoire a pour objet l'étude d'un IP Core RSA_512 bits (Circuit Open-core), son extension et son adaptation à une solution SoC. Cet IP-core a été étudié, testé, et implémenté sous l'environnement ISE/EDK. Son étude a montré que sa conception est basée sur la méthode d'exponentiation binaire MSB. Elle est basée d'une part, sur une structure parallèle permettant l'éxécution de deux multiplications modulaires en même temps ce qui permet de réduire les temps de Cryptage/Décryptage des données. D'autre part, elle repose sur l'intégration de l'algorithme de Montgomery qui permet d'accélérer l'exponentiation modulaire.

Les résultats d'implémentation hardware de l'IP-Core sous ISE Xilinx ont été comparés, à ceux obtenus de l'implémentation software de l'algorithme RSA sous Linux C++ en s'appuyant sur la bibliothèque GMP qui supporte les grands nombres, afin d'évaluer les performances de chaque approche. Ils ont été comparés par la suite à ceux obtenus par une implémentation SOC sous EDK. L'IP-Core a été interconnecté via un bus de transmission rapide FSL.

Mots clefs : Cryptographie – RSA – Montgomery –GMP – EDK/Microblaze– SoC – FPGA – IP-Core - FSL

Abstract

Our project consists on the study of RSA_512 bits IP-Core (Open-core Circuit) , its adaptation and extension on the SoC. This later was studied, tested and implemented under ISE/EDK environnement. The conception of this one is based on MSB Binary Exponentiation method. First, it is based on parallel structure which allow the fast block multipliers working in a pipelined fashion to take advantage of massive parallelism, and especially ensure minimum time consumption for crypting/decrypting data, in general purpose processors. And second, uses the Montgomery algorithm as the best solution to accelerate the modular exponentiation.

The hardware implementation results of the IP-Core under ISE Xilinx was discussed and compared to the ones obtained from the software implementation of the RSA algorithm under Linux C++ environment using the GMP Library which handle the big integer number, in purpose to evaluate the performance of each approach. And after that, to the ones obtained from the SOC hardware implementation under EDK Platform. The RSA IP-Core was connected to the Microblaze Processor using FSL bus to fast transferring data.

Key words : Cryptography – RSA – Montgomery – GMP – EDK/Microblaze – SoC – FPGA – IP-Core - FSL.

Remerciements

Je tiens avant tout à remercier DIEU le tout puissant, tout clément, de m'avoir donné la force a réaliser mon travail.

Je tiens à remercier vivement Monsieur SADOUN Rabah, Chargé de cours à l'Ecole Nationale Polytechnique d'Alger, pour son aide, son suivi, ses conseils et directives et pour son soutien tout au long de mon parcours d'études.

Mes sincères remerciements vont à Monsieur AKSSAS, Professeur à l'Ecole Nationale Polytechnique d'Alger, d'avoir accepté de présider le jury.

Je tiens également à remercier tous les membres du jury d'avoir bien voulu accepter d'examiner mon travail.

Mes remercions vont à tous les enseignants de l'Ecole Nationale Polytechnique d'Alger, spécialement ceux du département de l'Electronique, pour leur apport en savoir.

Enfin, mes remerciements vont à toute personne ayant contribuée, de prés ou de loin, à réaliser ce travail.

Dédicaces

Je dédie ce modeste travail à ma mère, ma mère, ma mère, ma mère,……., ma mère, à mon père, à ma très chère grand-mère Mani, et mes chers frères Fawzi et Hamza, qui m'ont soutenu sans relâche, dans toutes les circonstances, tout au long de mon parcours d'études.

Je dédie aussi, ce travail à la mémoire de ma très chère tante Djamila qui a tant souhaité voir ce grand jour de réussite.

Je dédie aussi, à mon cher oncle Djorlal Mohamed Lamine Professeur à l'Ecole Nationale Polytechnique d'Alger, et toute sa famille, de m'avoir soutenu sans relâche, dans toutes les circonstances, tout au long de mon parcours d'études.

Je dédie aussi, à toute ma grande famille et mes amies pour leur soutien, encouragements, et réconfort tout au long de mon parcours.

Enfin, je dédie à toute personne ayant contribuée, de prés ou de loin, à réaliser ce travail.

Sommaire

Chapitre III : Implémentation de RSA en langage C

Chapitre IV : Etude de L'IP-Core RSA_512

Liste des figures

Liste des tableaux & Algorithmes

Glossaire

A

ASIC : Application Specific Integrated Circuit
ANSI : Agence de Normalisation et de standardisation internationale

B

BSB : Base System builder
BRAM: block of random access memory

C

Cell : Cellule
CPLD: Complex Programmable Logic Device
CLB : Configurable Logic Bloc

D

DSS : Digital Signature Standard
DLMB : Data Local Memory Bus
DOPB : Data On-Chip Peripheral Bus

E

EDK : Embedded Development Kit
EPLD : Erasable Programmable Logic Device
elf file : Executable and Linkable Format file

F

FPGA : Field Programmable Gate Array
FSL : Fast Link Simplex
FIFO : First In, First Out

G

GUI : Graphical User Interface

H

HW : Hard Ware

I

IOB : Input/Output Bloc
ISE : Integrated Software Environment
IP-Core : Intellectual Property Core
ISO : International Standards Organisation
ILMB : Instruction Local Memory Bus
IOPB : Instruction On-chip Peripheral Bus

J

JTAG : Joint Test Action Group

L

LSB : Least significant bit

M

μP : Micro Processeur
μC : Micro Contrôleur
mod : Modulo
Mux : Multiplexer
mss file : Microprocessor Software Specification file
mhs file : Microprocessor Hardware Specification file
MMM : Multiplication Modulaire de Montgomery
MMME : Multiplication Modulaire de Montgomery Entrelacé
MMMSF : Multiplication Modulaire de Montgomery avec soustraction finale

N

ns : nanosecondes

O

OPB : On-chip Peripheral Bus

P

PE : Processor élément
PLD : Programmable Logic Device
PC : Personnel Computer

R

RSA : Rivest, Shamir and Adleman
RS232 : Recommended Standard 232
RISC : Reduced Instruction-Set Computer

S

SRAM : Synchrone Random Access Memory
SoC : System on Chip
SW : Soft Ware
SDK : Software Development Kit

U

us: microsecondes

V

VHDL : Very high speed integrated circuit Hardware Description Language
VGA : Video Graphics Array

X

XCL : Xilinx CacheLink
XPS : Xilinx Platform Studio

Introduction générale

Le développement des technologies de l'information et de la communication a accru la nécessité de protéger des milliers d'informations circulant sous forme numérique à travers tout les réseaux, plus spécialement le réseau mondial Internet. Ces informations se trouvent susceptibles d'être lues, copiées, modifiées ou même supprimées sur ce réseau non contrôlé. Pour assurer la confidentialité de ces données, on fait appel à la cryptologie.

La cryptologie est une science mathématique dont l'object est l'étude des méthodes permettant d'assurer les services d'intégrités, d'authenticités, et de confidentialités dans les systèmes d'informations et de communications. Elles regroupent deux sciences duales et étroitement liées : la cryptographie qui est la science de la construction d'algorithmes cryptographiques et la cryptanalyse qui est la science de l'étude des attaques sur ces algorithmes.

En cryptographie, protéger un message consiste à appliquer sur ce dernier une transformation pour qu'il ne soit compréhensible juste pour des personnes autorisées. c'est ce qu'on appelle le chiffrement qui donne un texte chiffré (cryptogramme) à partir d'un texte clair. Inversement, le déchiffrement de ce message est l'action qui permet de reconstruire le texte clair à partir du texte chiffré. Pour cela, on utilise un certain nombre de système basé sur des algorithmes cryptographiques qui dépendent d'un paramètre appelé clé.

La clé est une valeur utilisée pour chiffrer ou déchiffrer un message. Selon le Choix de ces clés on distingue deux types de cryptographie : cryptographie symétrique qui utilise la même clé pour chiffrer et déchiffrer. La cryptographie asymétrique qui par contre, utilise deux clés ; une pour chiffrer et l'autre pour déchiffrer.

C'est aux systèmes de chiffrement asymétriques (à clé publique) que nous nous sommes intéressés car le problème d'échange préalable de clés ne se pose plus. Les crypto systèmes à clés publiques sont généralement construits sur des problèmes mathématiques difficiles à résoudre. Le premier est celui de la factorisation d'un entier à deux nombres premiers, ce qui donne naissance au crypto- système RSA.

Actuellement le protocole cryptographique à clé publique RSA est le plus utilisé. Il offre un niveau de sécurité acceptable des données transmises. Sa sécurité vient de la difficulté de la factorisation des grands nombres.

L'opération clé du crypto-système RSA est l'exponentiation modulaire, utilisée dans le cryptage et décryptage. Elle s'exécute en une série répétée de multiplications modulaires.

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

Une implémentation software du protocole RSA s'avère très lente. Un moyen d'accélérer les opérations mathématiques impliquées dans le protocole RSA est l'utilisation d'une solution matérielle : telle que les circuits FPGA (Field Programmable Gate Array) qui offrent la possibilité de changer la taille des clés par reconfiguration de ce dernier, selon le niveau de sécurité et les performances désirées.

Le but de ce projet est de réaliser une implémentation hardware d'un IP-Core RSA_512 sur un circuit FPGA afin d'augmenter les performances de calcul de l'exponontiation modulaire. Cette opération est très consommatrice de temps pour une implémentation software, mais prend moins de temps pour une implémentation Hardware.

Le défi est de concevoir un IP-Core pour des opérandes dont la taille est plus élevée que 512 bits, adaptée aux ressources du circuit FPGA de la famille Virtex-5 pour trou le meilleur compromis entre la vitesse de calcul et la surface occupée, et surtout que le temps de chiffrage/déchiffrage doit rester réduit et constant au fur et à mesure que la taille de l'opérande augumente.

Pour cela, nous avons organisé notre mémoire en six chapitres répartis comme suit :

Le premier chapitre a été consacré aux généralités sur la cryptographie et ses différents types ainsi que le protocole RSA et son principe de fonctionnement y compris sa mise en œuvre.

Le deuxième Chapitre porte sur l'étude comparative des différents algorithmes de multiplication et exponentiation modulaire pour dégager celui le plus adapté à notre implémentation matérielle. Notre choix a été porté sur l'algorithme de Montgomery, le plus rapide pour la multiplication et la méthode binaire rapide connue sous le nom "Square and multiply'' pour l'exponentiation.

Le troisième chapitre a été consacré pour l'étude de l'algorithme RSA implémenté en langage C, ainsi que l'étude de la bibliothèque GMP et son impact sur la rapidité de l'algorithme qui est mise en œuvre, pour obtenir des résultats de compilations.

Le quatrième chapitre, a été consacré à l'étude de l'architecture globale de notre IP-Core RSA_512 afin de présenter les différents blocs de multiplications modulaires de calcul et de contrôle y compris le rôle des fifos et mémoires constituant la totalité de l'architecture.

Le cinquième chapitre a porté sur l'extension de l'architecture globale de RSA_512 vers RSA_1024. Les résultats de simulation et implémentation des deux architectures sur le circuit FPGA de la famille Virtex-5, ont été enregistrés à partir de l'outil Modelsim sous l'environnement ISE de Xilinx, afin de tirer les performances temporelles et le taux d'occupation du circuit. Ces derniers ont été comparés avec ceux conclus dans le chapitre 3.

Le sixième chapitre a porté sur l'implémentation de L'IP-Core dans le système embarqué EDK, en étudiant l'interfaçage de l'IP-Core RSA_512 avec Microblaze à travers les bus FSL et son intégration dans une solution Soc.

Nous terminons notre travail par l'évaluation des performances des résultats obtenus dans le chapitre 3 et chapitre 5. L'étude comparative est déduite dans une conclusion générale.

Chapitre I

Introduction a la Cryptographie RSA

Introduction

Le développement des technologies de l'information et de la communication a accru la nécessité de protéger des milliers d'informations circulant sous forme numérique à travers tout les réseaux, plus spécialement le réseau mondial Internet. Ces informations se trouvent susceptibles d'être lues, copiées, modifiées ou même supprimées sur ce réseau non contrôlé. Pour assurer la confidentialité de ces données, on fait appel a la cryptologie.

La cryptologie est une science mathématique dont l'object est l'étude des méthodes permettant d'assurer les services d'intégrités, d'authenticités, et de confidentialités dans les systèmes d'informations et de communications. Elles regroupent deux sciences duales et étroitement liées : la cryptographie qui est la science de la construction d'algorithmes cryptographiques et la cryptanalyse qui est la science de l'étude des attaques sur ces algorithmes.

En cryptographie, protéger un message consiste à appliquer sur ce dernier une transformation pour qu'il ne soit compréhensible juste pour des personnes autorisées. c'est ce qu'on appelle le chiffrement qui donne un texte chiffré (cryptogramme) à partir d'un texte clair. Inversement, le déchiffrement de ce message est l'action qui permet de reconstruire le texte clair à partir du texte chiffré. Pour cela, on utilise un certain nombre de système basé sur des algorithmes cryptographiques qui dépendent d'un paramètre appelé clé.

La clé est une valeur utilisée pour chiffrer ou déchiffrer un message. Selon le Choix de ces clés on distingue deux types de cryptographie : cryptographie symétrique qui utilise la même clé pour chiffrer et déchiffrer. La cryptographie asymétrique qui par contre, utilise deux clés ; une pour chiffrer et l'autre pour déchiffrer.

C'est aux systèmes de chiffrement asymétriques (à clé publique) que nous nous sommes intéressés car le problème d'échange préalable de clés ne se pose plus. Les crypto systèmes à clés publiques sont généralement construits sur des problèmes mathématiques difficiles à résoudre. Le premier est celui de la factorisation d'un entier à deux nombres premiers, ce qui donne naissance au crypto- système RSA.

Actuellement le protocole cryptographique à clé publique RSA est le plus utilisé. Il offre un niveau de sécurité acceptable des données transmises. Sa sécurité vient de la difficulté de la factorisation des grands nombres.

L'opération clé du crypto-système RSA est l'exponentiation modulaire, utilisée dans le cryptage et décryptage. Elle s'exécute en une série répétée de multiplications modulaires.

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

Une implémentation software du protocole RSA s'avère très lente. Un moyen d'accélérer les opérations mathématiques impliquées dans le protocole RSA est l'utilisation d'une solution matérielle : telle que les circuits FPGA (Field Programmable Gate Array) qui offrent la possibilité de changer la taille des clés par reconfiguration de ce dernier, selon le niveau de sécurité et les performances désirées.

Le but de ce projet est de réaliser une implémentation hardware d'un IP-Core RSA_512 sur un circuit FPGA afin d'augmenter les performances de calcul de l'exponontiation modulaire. Cette opération est très consommatrice de temps pour une implémentation software, mais prend moins de temps pour une implémentation Hardware.

Le défi est de concevoir un IP-Core pour des opérandes dont la taille est plus élevée que 512 bits, adaptée aux ressources du circuit FPGA de la famille Virtex-5 pour trou le meilleur compromis entre la vitesse de calcul et la surface occupée, et surtout que le temps de chiffrage/déchiffrage doit rester réduit et constant au fur et à mesure que la taille de l'opérande augumente.

Pour cela, nous avons organisé notre mémoire en six chapitres répartis comme suit :

Le premier chapitre a été consacré aux généralités sur la cryptographie et ses différents types ainsi que le protocole RSA et son principe de fonctionnement y compris sa mise en œuvre.

Le deuxième Chapitre porte sur l'étude comparative des différents algorithmes de multiplication et exponentiation modulaire pour dégager celui le plus adapté à notre implémentation matérielle. Notre choix a été porté sur l'algorithme de Montgomery, le plus rapide pour la multiplication et la méthode binaire rapide connue sous le nom "Square and multiply'' pour l'exponentiation.

Le troisième chapitre a été consacré pour l'étude de l'algorithme RSA implémenté en langage C, ainsi que l'étude de la bibliothèque GMP et son impact sur la rapidité de l'algorithme qui est mise en œuvre, pour obtenir des résultats de compilations.

Le quatrième chapitre, a été consacré à l'étude de l'architecture globale de notre IP-Core RSA_512 afin de présenter les différents blocs de multiplications modulaires de calcul et de contrôle y compris le rôle des fifos et mémoires constituant la totalité de l'architecture.

Le cinquième chapitre a porté sur l'extension de l'architecture globale de RSA_512 vers RSA_1024. Les résultats de simulation et implémentation des deux architectures sur le circuit FPGA de la famille Virtex-5, ont été enregistrés à partir de l'outil Modelsim sous l'environnement ISE de Xilinx, afin de tirer les performances temporelles et le taux d'occupation du circuit. Ces derniers ont été comparés avec ceux conclus dans le chapitre 3.

Le sixième chapitre a porté sur l'implémentation de L'IP-Core dans le système embarqué EDK, en étudiant l'interfaçage de l'IP-Core RSA_512 avec Microblaze à travers les bus FSL et son intégration dans une solution Soc.

Nous terminons notre travail par l'évaluation des performances des résultats obtenus dans le chapitre 3 et chapitre 5. L'étude comparative est déduite dans une conclusion générale.

I.1. Introduction et origine de la Cryptographie [1]

La cryptographie [1] a vu le jour pour la première fois il y a environ 4000 ans, au bord du Nil, où était utilisée par les scribes égyptiens pour raconter l'histoire de leurs maîtres, cependant elle était loin d'être utilisée pour des fins militaires.

On notera un évènement très marquant où on a conçut pour la première fois, à Sparte (Grèce) un procédé de chiffrement militaire, des historiens grecs mentionnent l'utilisation de ce procédé par les Spartiates vers 475 avant J.C. Ultérieurement, les Grecs mettaient au point plusieurs précédés stéganographiques, ils sont aussi à l'origine du premier procédé de substitution, mis par l'écrivain grec Polybe, mais ce dernier n'a vu son utilisation que dans les opérations militaires menées par J.César.

Au moyen âge, la cryptologie évolue très faiblement car elle représentait peu d'importance. Cependant, elle fut utilisée par les arabes. En 1467, le savant italien Leon Batista Alberti inventa la substitution poly-alphabétique et mis en place le cadran chiffrant d'Alberti. Il ira plus loin encore pour mettre en place un répertoire de 336 groupes de mots représentés par toutes les combinaisons allant de 11 à 4444.

Ces découvertes ne seront utilisées que 400 ans plus tard. Depuis ce temps là, de nombreuses découvertes apportées par des savants européens ont enrichi la cryptographie, notamment la découverte de la notion de clef littérale par Giovanni Batista Belaso, qu'il appela "mot de passe", ou la découverte du "carré de Vigenère" qui pouvait dissimuler un message dans l'image d'un champ d'étoiles.

A l'aube du 20ème siècle, le savoir en cryptographie et cryptanalyse[2] est important. C'est dans le domaine militaire que l'on verra le plus cette science des écritures secrètes. Durant la Seconde Guerre Mondiale, la cryptographie connût un développement considérable notamment avec *l'utilisation de la machine ENIGMA*. Cette dernière possédait un fonctionnement particulièrement simple car l'objet était équipé d'un clavier pour la saisie du message, de différentes roues pour le codage, et enfin d'un tableau lumineux pour afficher les résultats.

I.2. Définition de la Cryptographie [1]

L'homme a toujours ressenti le besoin de dissimuler des informations, bien avant même l'apparition des premiers ordinateurs et de machines à calculer. Depuis sa création, le réseau Internet a tellement évolué qu'il est devenu un outil essentiel de communication. Cependant, cette communication met de plus en plus en jeu des problèmes stratégique liés à l'activité des entreprises sur le Web. Les transactions faites à travers le réseau peuvent être interceptées, d'autant plus que les lois ont du mal à se

mettre en place sur Internet, il faut donc garantir *la sécurité de ces informations*, c'est la cryptographie[1] qui s'en charge.

La cryptographie est une des disciplines de la cryptologie qui s'attache à protéger des messages assurant *confidentialité, authenticité et intégrité* en s'aidant souvent sur des secrets ou clés. *C'est la science qui permet de rendre une information inaccessible*, sauf pour ceux à qui elle est destinée.

I.3. Définition d'une clé [2] [3]

La cryptographie est historiquement l'art de cacher une information pour la rendre inintelligible à toute personne ne connaissant pas un certain secret. [3] Ce secret est la clé utilisée lors de l'opération du chiffrement et déchiffrement.

Le fait de coder un message de telle façon à le rendre incomprehensible s'appelle chiffrement. La méthode inverse est appelée déchiffrement, elle consiste à retrouver le message original,.

Le résultat de cette modification (le message chiffré) est appelé cryptogramme (en anglais ciphertext) par opposition au message initial, appelé message en clair (en anglais plaintext). Voir la figure I.1

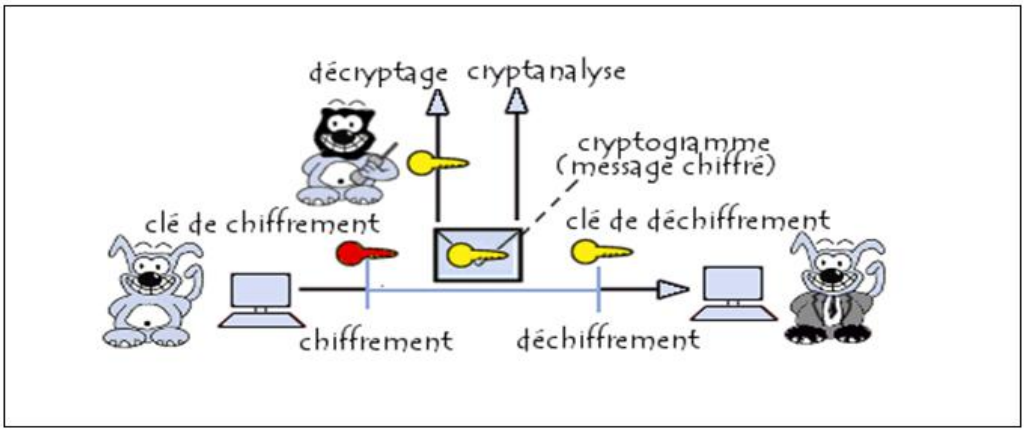


Figure I.1 : Système Cryptographique [3]

La clé peut se présenter sous plusieurs formes: mots, phrases ou procédure. La protection apportée par un algorithme de chiffrement est liée à la longueur de cette clé, qui peut s'exprimer en bits. C'est donc une borne supérieure sur la sécurité du système.

Une clé de chiffrement peut être *symétrique* (cryptographie symétrique) ou *asymétrique* (cryptographie asymétrique)[3]. Dans le premier cas, la même clé sert à chiffrer et déchiffrer le

message. Dans le second, on utilise deux clés différentes ; la clé de chiffrement est publique alors que celle servant au déchiffrement est gardée secrète (clé privée, ne peut pas se déduire de la clé publique).

I.4. Différents types de cryptographie [3]

Selon la nature de la clé utilisée dans le système cryptographique. Il existe deux types de Cryptographie différents :

- *La cryptographie à clef secrète*, dite également *symétrique* ou bien classique.
- *La cryptographie à clef publique*, dite également *asymétrique* ou moderne.

I.4.1. Cryptographie Symétrique (classique) [4]

Toutes les deux visent à assurer *la confidentialité* de l'information, mais la cryptographie à *clef secrète* ; exige au préalable la mise en commun de la clef (symétrique) entre l'expéditeur et le destinataire [4], nécessaire au chiffrement ainsi qu'au déchiffrement des messages d'ou la ***Symétrie***.

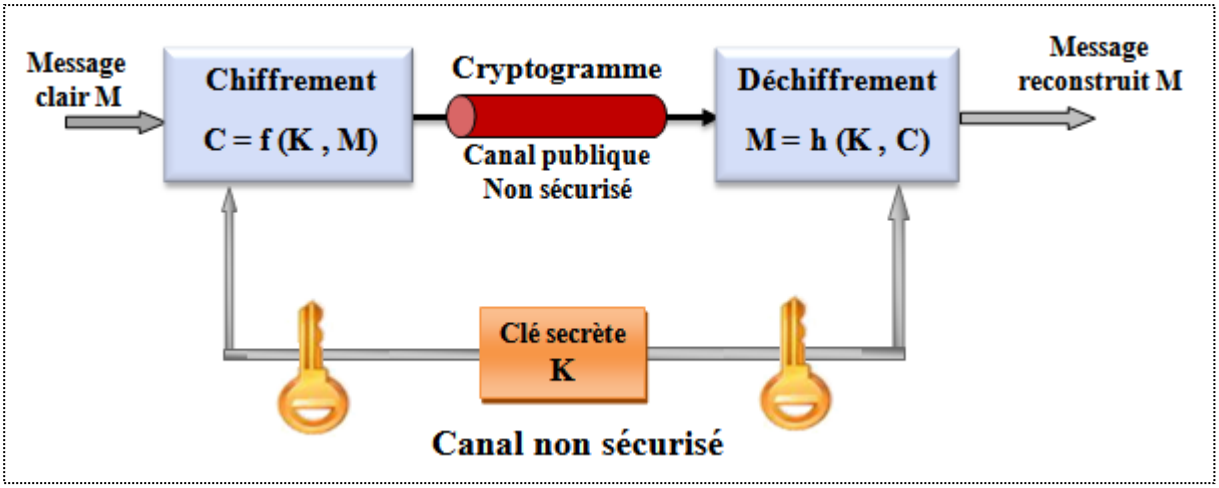


Figure I.2 : Cryptographie Symétrique [3]

Les algorithmes de chiffrement symétriques sont souvent basés sur des techniques de substitutions et de transpositions. Ce qui offre un moyen rapide et efficace pour chiffrer un message. Dans ce type de cryptographie, il existe plusieurs méthodes de chiffrement tels que : DES [3] (Data Encryption Standard), l'AES (Advanced Encryption Standard) [3], Blowfish, IDEA **.** Ces systèmes permettent un chiffrement rapide et facile mais malheureusement présentent des inconvénients lors de leurs mises en pratique :

- La clé qui doit rester totalement confidentielle, elle est transmise avec le message de façon sure (mise a la disposition de tout le monde) . La clé de chiffrement = La clé de déchiffrement.
- Si la clé secrète est piratée par un opposant, alors ce dernier pourra déchiffrer le message encodé avec celle-ci.
- Il est difficile de trouver un canal sécurisé pour la distribution des clés secrètes.
- Puisque une clé différente est utilisée pour chaque paire différente d'utilisateurs du réseau, le nombre total des clés augmente en fonction du nombre total d'utilisateurs.
- La clé est de petite taille (de 128 bits), fassilement cassable : , il existe $\mathbf{2^{128}}$ valeur de clé.

I.4.2. Cryptographie Asymétrique (à clé publique) [3]

Un chiffrement asymétrique [3] utilise une clé de chiffrement différente de celle de déchiffrement. La clé de chiffrement est souvent connue de tous le monde, (appelée clé publique). Elle permet de construire un message chiffré. Cependant seuls les participants connaissant la clé de déchiffrement (appelée clef privée), peuvent déchiffrer les messages.

La clé publique est publiée dans des annuaires et ainsi connue de tous le monde, alors que la clé privée n'est connue que de la personne à qui la paire appartient. Pour envoyer un texte chiffré à une personne, il faut utiliser la clé publique de cette personne et chiffrer le texte. Une fois reçu, le destinataire utilise sa clé privée correspondante pour retrouver le message clair. Comme le montre la figure I.3

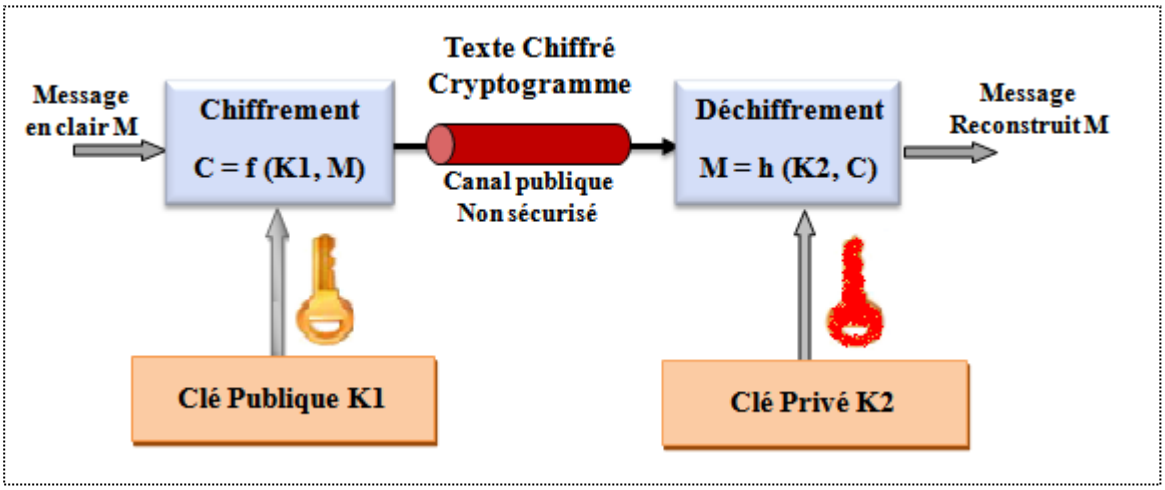


Figure I.3 : Cryptographie Asymétrique [3]

Dans ce type de cryptographie, il existe plusieurs méthodes de chiffrement tels que : RSA, DSA-DH, El Gamal, les courbes élliptiques. Ils ont l'avantage de la signature numérique, ainsi que la confidentialité de la clé secréte, qui est de 1024 bits (il existe $\mathbf{2^{1024}}$ valeur de clé.). Mais présente, par contre les inconvenients suivants :

- Le temps de calcul qui est nettement plus long, et requiert beaucoup d'opérations. Pour cela, il n'est pas recommandé pour de grande quantité d'informations.
- la lenteur de l'algorithme, Par exemple (RSA est 1000 fois plus lent que DES)

Les chiffrements asymétriques [3] sont souvent basés sur l'existence de fonctions mathématiques dites à sens unique (ou sens unique avec trappe).

Si pour exemple, Monsieur X souhaite recevoir des messages chiffrés de n'importe le qui, il génère une fonction à sens unique et à brèche secrète (Clé Publique) à l'aide d'un algorithme de chiffrement asymétrique (RSA par exemple). Il le diffuse, mais garde secret l'information permettant d'inverser cette fonction (Clé Privée). On parle donc de clé publique pour celle qu'on diffuse (sans se soucier de la sécurité) et clé privée pour l'information secrète (propriété de Monsieur X).

Une fonction à sens unique [3] (fonction a brèche secrète) est une fonction mathématique facilement calculable mais dont la réciproque est impossible à calculer. Par exemple, le produit de deux grands nombres premiers est une opération mathématique simple et facilement calculable, mais trouver à partir de ce produit les deux nombres premiers est un problème difficile a résoudre. Ce problème est appelé problème de *factorisation*. La factorisation en produit de deux nombres premiers est une fonction à sens unique avec trappe ; si nous connaissons l'un des deux nombres premiers, alors il sera plus facile, de retrouver le second nombre premier par une simple division. C'est pour cette raison que RSA vient alléger le probléme de cette factorisation.

I.5. Le chiffrement RSA [4]

Cet algorithme a été décrit en 1977 par Ron Rivest, Adi Shamir et Len Adleman, d'où le sigle RSA. C'est un algorithme de cryptographie asymétrique, très utilisé dans le commerce électronique, et plus généralement pour échanger des données confidentielles sur Internet. A présent, le brevet de cet algorithme appartient à la société américaine RSA Data Dynamics et aux Public Key Parteners qui détiennent les droits en général sur les algorithmes à clef publique. L'algorithme RSA se base essentiellement sur **la difficulté de factoriser un très grand nombre en deux nombres premiers**.

I.5.1. Principe de fonctionnement de RSA [8]

Avec l'algorithme Asymétrique RSA, on a 2 clés (Clé public, Clé privé), la première clé est possédé par tout le monde, il ya aucun risque que l'expéditeur la transmet à n'importe le qui. La deuxième clé (la clé privé) elle est possédé seulement par le destinataire, car elle sert à décrypter le message déjà crypté par la clé publique.

Ronald Rivest, Adi Shamir et Leonard Adleman [8], ont publié l'idée d'utiliser l'anneau Z/nZ et le petit théorème de Fermat pour obtenir des fonctions trappes, ou fonctions à sens unique à brèche secrète. RSA repose sur le calcul de l'exponentiation modulaire, son fonctionnement s'appuit sur les trois importantes étapes cités ci-dessous :

I.5.1.1. Génération des clés [5]

Partons du fait qu'on va choisir deux grands nombres premiers (de l'ordre de 200 chiffre) de manière totalement aléatoire et appelons les **p, q,** pour se faire il existe des algorithmes de génération aléatoire de nombres premiers.

- ***Calcul de la clé publique***

1. Soit **p, q :** deux nombre premiers tel que **:** $\mathbf{p > 10^{100}}$ **,** $\mathbf{q > 10^{100}}$

2. Soit **n** le produit de ces deux nombres p et q tel que : **n = p * q**
3. Calcule de la fonction indicatrice d'Euler de n, tel que : **¢(n) = (p-l) * (q-1)** .
4. Calcule d'un nombre entier **e** tel que le **:** **PGCD (¢(n) ,e) = 1** et **p, q < e < ¢(n)**

On déduit la clé publique = **(n, e)**

- ***Calcul de la clé privé [4]***

5. Soit un nombre entier **d** , tel que : **p, q < d < ¢(n)**
   calculant d en utilisant l'algorithme d'Euclide étendu de tel sorte que :
6. **e * d** soit divisible par **¢(n) ,** ce qui implique que **: e * d= mod ¢(n) ,** ou bien : **d = 1/e mod ¢(n)**

On déduit la clé privée = **(n, d)**

I.5.1.2. Le cryptage avec l'algorithme RSA [5] [6]

Soit à crypter et envoyer un message *m* dont la taille est inférieure à n. Dans le cas où la taille du message m est supérieure à celle de n, il sera question de découper m en bloc Mi dont la taille est strictement inférieure a n.

Supposant qu'un expéditeur veux envoyer un message à un destinataire , il suffit qu'il se procure de la clé publique du destinataire afin de pouvoir calculer le message chiffré C (voir figure I.4) . L'expéditeur crée le texte chiffré **C** à partir du texte clair **m** avec la Clé publique **(n, e)** publié par le destinataire selon l'équation d'exponontiation modulaire suivante:

$$\mathbf{C = } m^e \mathbf{\bmod n}$$

A fin de mieux sécuriser le texte clair, l'éxpéditeur de sa part crée la signature **S** du message haché m' (empreinte du texte clair) en utilisant sa propre clé privé **(n, d)** selon l'équation d'exponontiation modulaire suivante **:**

$$\mathbf{S = } m'^{d} \mathbf{\bmod n}$$

(La clé privé sera communiqué au destinataire pour des besoin ultérieures) voir figure I.4

I.5.1.3. Le décryptage avec l'algorithme RSA [5]

Maintenant que le destinataire a recu le texte chiffré **C** envoyé par l'expéditeur , il suffit juste de le décrypter en utilisant la clé privé de l'éxpediteur **(n, d),** (lui a été déjà envoyé), afin de pouvoir réstituer le message original (text clair **m**), selon l'équation d'exponontiation modulaire suivante **:** voir figure I.4

$$m = C^d \mathbf{\bmod n}$$

Pour authentifier le message clair envoyé par l'éxpéditeur et s'assurer qu'il na pas été intercepté par un intrus, le destinataire procéde à calculer son propre text condensé m'' d'une part , et à décrypter le condensé envoyé par l'expéditeur m' en utilisant sa propre clé publique **(n, e)** d'autre part. Ces deux taches seront faites en paralléle par le destinataire afin de vérifier l'intégrité et l'hauthenticité du message envoyé par l'éxpéditeur .

Le décryptage du cendensé se fait selon l'équation d'exponontiation modulaire suivante **:** voir figure I.4

$$m' = S^e \bmod n$$

La figure I.4 explique le principe de fonctionnment de l'algorithme RSA avec les détails des opérations de cryptage et décryptage des messages, en tenant compte des aspets qui assurent leur securité, inetgrité et authenticité.

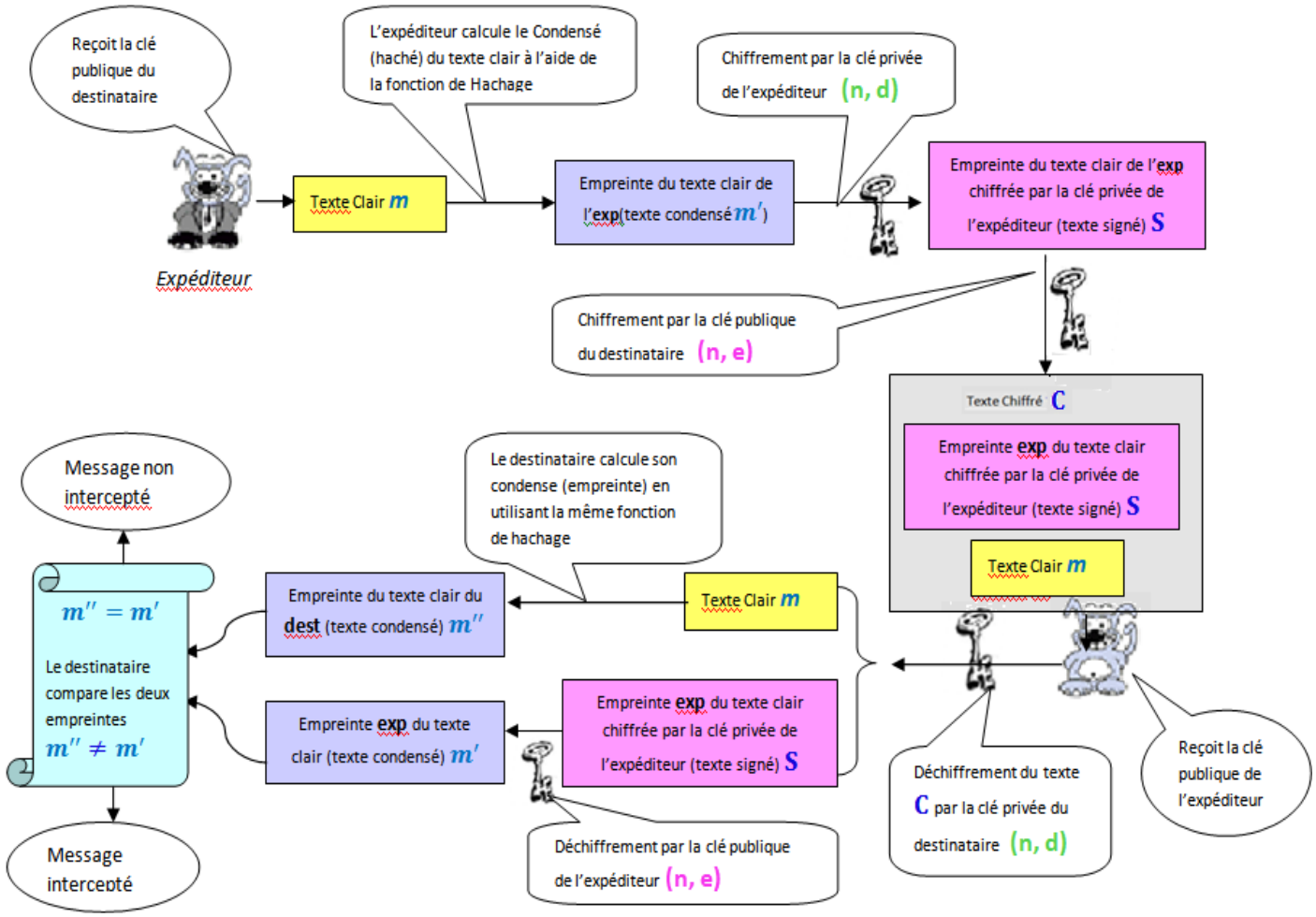


Figure I.4 : Principe de fonctionnement de RSA

Exemple :

Soit Bob un émetteur, crypte un message clair et l'envoit à une destinatrice Alice. Bob choisit deux nombres premiers : 503 et 563 car ils ne sont divisibles que par 1 et eux-mêmes.

1. **p = 503** et **q = 563**
2. Une fois **p** et **q** généré, Bob calcule : **n = 503*563** = **283189**
3. Bob calcule l'indicateur d'Euler : **¢(n) = (503-1)*(563-1)** = **282124**
4. Bob choisit e de tel sorte que : **p , q < e < ¢(n) et PGCD(e, ¢(n)) = l**

   on sait que e se trouve entre : 563 < e < 282124, on commence a tester les équations suivantes :

   PGCD(282124, 564) = 4
   PGCD(282124, 565) = 1
   PGCD(282124, 566) = 2
   PGCD(282124, 567) = 1

5. Donc: **e = 565** et **la Clé publique de Bob sera: (565, 283189)**.
   Bob peut maintenant donner cette clé à Alice pour crypter son message, ou bien l'a publié à tout le monde.

6. Bob calcule d , en utilisant l'algorithme d'Euclide étendu : **d = 1/e mod ¢(n)**

   565 * 564 mod 282124 = 566
   565 * 565 mod 282124 = 37101
   .....
   565 * 140313 mod 282124 = 1
   565 * 140314 mod 282124 = 566

7. Donc: **d= 140313** et **la Clé privé de Alice : (140313, 283189)**.
   Alice peut maintenant décrypter le message crypté par Bob, en utilisant la clé privé calculé

I.5.2. Securité de RSA [15]

Bien que la clé publique et la clé privée soient liées dans un crypto-système à clé publique, il est très difficile de déduire la clé privée en connaissant seulement la clé publique.

Toutefois, il est toujours possible de déduire la clé privée si l'on dispose de suffisamment de temps et de puissance de calcul. II est donc très important de choisir une clé de taille convenable; pour assurer la sécurité du système .

Le paramètre essentiel pour la sécurité d'un crypto-système est la longueur de la clé. Une clé de longueur de 128 bits est considérée comme sûre pour les systèmes symétriques, néanmoins pour les asymétriques cette longueur est considérée comme extrêmement faible.

Il existe plusieurs façons d'essayer de casser une clé. La manière la plus simple de trouver une clé de taille n=512 bits, dans le chiffrement RSA, et d'essayer de factoriser le nombre n, vue que l'attaque exhaustive est infaisable. En 1999, le plus grand entier qui a été factorisé avec l'algorithme le plus performant, est constitué de 155 chiffres (soit une clé de 512 bits), Pour un haut niveau de sécurité, il faut donc choisir des clés plus grandes. Si l'on admet que la puissance des ordinateurs double tous les 18 mois (loi de Moore), une clé de 2048 bits devrait tenir jusqu'h 2079.

I.5.3. Mise en œuvre du protocole RSA [6]

Le principe de crypto-système **RSA** s'avère assez simple, mais sa mise en œuvre pose parfois des problèmes dans la génération de deux nombres premiers p et q et l'élévation d'un grand nombre à une grande puissance modulo n. La mise en œuvre de **RSA** s'appuis sur I' exponentiation modulaire, qui est une opération lourde à effectuer vu la taille de la clé de déchiffrement. Le principe de chiffrement du protocole RSA est illustré sur la figure I.5

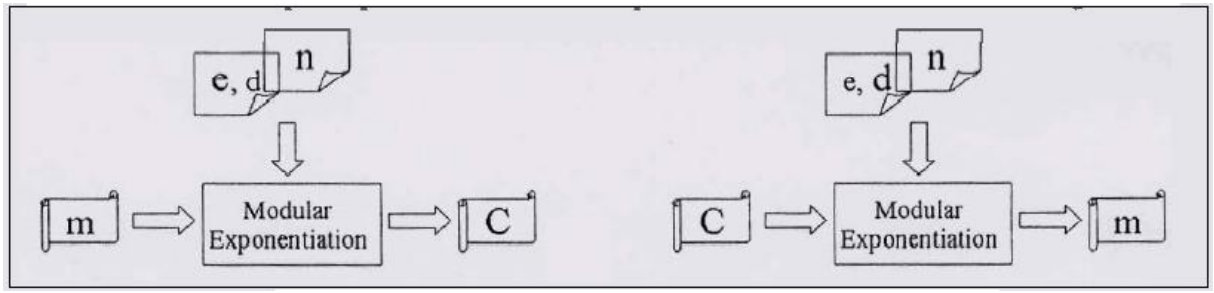


Figure I.5 : Chiffrement / déchiffrement de RSA [8]

I.5.4. Besoin en arithmétique et complexité de RSA [5] [8]

Les fonctions de chiffrement et de déchiffrement dans le protocole **RSA** nécessite tous les deux une exponentiation modulo n. Celle-ci est même utilisée dans la plupart des crypto-systèmes à clé publique.

Le problème de I'exponentiation modulaire est de calculer $C=m^e \bmod n$ ou **m, n** et **e** sont de très grands entiers. En effet pour assurer un haut niveau de sécurité, la longueur de **n** doit être au moins 1024 bits. La longueur de n et de l'exposant e tendent à grandir dans le futur en même temps que la cryptanalyse.

La première règle de I'exponentiation modulaire est qu'on ne peut pas calculer $C = m^e \bmod n$ par le calcul directe de m^e puis calculer ensuite le reste de la division de **C** par **n**. Le résultat temporaire doit être réduit à modulo **n** à chaque étape de I'exponentiation modulaire et ceci car l'espace requit au stockage du nombre binaire m^e est énorme.

Par exemple si **m** et **e** ont seulement 256 chacun, nous avons besoin de

$[log_2(m^e)]+1 = [e * log_2(m)]+1 = [2^{255} * 255]+1 = 10^{79}$ pour stocker m^e .Ce nombre est très important et nous disposons pas de mémoire suffisante pour pouvoir le stocker.

Nous pouvons poser la question sur combien de multiplications modulaires nous avons besoin pour calculer $\mathbf{m^e \bmod n}$. Une méthode naïve pour calculer $\mathbf{C=m^e \bmod n}$ serait de commencer par le calcule de : $\mathbf{C=m \bmod n}$ et de faire I'opération $\mathbf{C=C*m \bmod n}$ jusqu'à obtenir $\mathbf{C=m^e \bmod n}$. Cette méthode requit e-1 multiplications modulaires. Il existe plusieurs méthodes qui ont pour objectif de diminuer le nombre de multiplication modulaire.

Actuellement, la méthode la plus utilisée et qui s'adapte bien avec une implémentation matérielle est la méthode binaire.

I.6. Conclusion :

Le chiffrement RSA présenté dans ce chapitre est fondé sur l'hypothèse qu'il est très difficile de factoriser un nombre très grand en deux nombres premiers, ceci le met en tète des chiffrements à clé publique.

Cependant, vue sa lenteur due au calcul de l'exponentiation modulaire qui n'est autre qu'une suite de multiplication modulaire, le met au sommets des algorithmes des chiffrements à clé publique. Ceci dit, la multiplication modulaire a un intérêt majeur sur le calcul de I' exponentiation modulaire et que toute amélioration dans son calcul est traduite dans l'exécution du protocole RSA, d'où l'intérêt d'accorder un soin particulier à la réalisation matérielle de cette opération, ce qui consiste à choisir un algorithme efficace en terme de temps de calcul et surface occupé. Ceci sera l'objet du chapitre prochain.

Chapitre II

Exponentiation / Multiplication modulaire

II. 1. Introduction

Le protocole RSA est basé sur l'opération de l'exponentiation modulaire utilisée dans les opérations de cryptage et décryptage des données, et définies par les équations mathématiques ci-dessous :

$$S = x^e \bmod n \qquad x = s^d \bmod n$$

Où ***x*** est le message à chiffrer dont la taille est compris entre 0 et n-1. ***n*** est le modulo dont la taille doit être d'au moins 512 bits pour assurer la sécurité de l'information. ***e*** et ***d*** sont respectivement ***la clé publique*** pour le chiffrement et la ***clé privée*** pour le déchiffrement.

L'exponentiation modulaire [10] est réalisée en une série répétée de multiplications modulaires dont le nombre est égale aux tailles de ***e*** et ***d*** qui peut aller jusqu'à 1024 bits. C'est pourquoi pour accélérer les opérations de cryptage et de décryptage, on doit optimiser le calcul de l'exponentiation modulaire en réduisant le nombre de multiplication modulaire et le temps d'exécution de cette derniere.

Dans ce chapitre, nous allons étudier les différentes méthodes de calcul de la multiplication modulaire et celles de l'exponentiation modulaire proposées dans la littérature pour choisir les méthodes les plus adaptées à l'implémentation hardware du protocole RSA.

Dans ce qui suit, nous allons commencer par étudier les différentes méthodes de calcul de l'exponentiation modulaire et choisir la méthode binaire comme solution optimal et adéquate a notre architecture.

II.2. Exponentiation modulaire [13]

L'exponentiation modulaire est l'opération clé du protocole RSA. Elle consiste à calculer les puissances successives d'un nombre donné, et d'en calculer ensuite le reste de sa division par un autre nombre (le modulo). Elle s'exécute en une succession de multiplications modulaires. Pour assurer un haut niveau de sécurité de RSA, la taille du modulo n et de l'exposant d doit être au moins égale à 512 bits et c'est cette valeur qui détermine le nombre de multiplications.

L'exponentiation modulaire sur de grands entiers est une opération très gourmande en temps de calcul. Plusieurs méthodes permettent d'accélérer cette opération. Elles visent toutes à réduire le nombre de multiplications modulaires et paralléliser leur exécution si c'est possible.

La multiplication modulaire est aussi une opération très consommatrice de temps. Plusieurs techniques ont été proposées dans les dernières années pour obtenir une implémentation efficace de la multiplication modulaire.

Les différentes méthodes de calcul d'exponentiation modulaire ont été proposés dans les dernières années tel que la méthode de chaîne d'addition, d'arbre de puissance et la méthode du facteur qui ont été destinées pour des algorithmes de petits exposants contrairement aux méthodes binaire, M-aire et celle des fenêtres glissantes qui sont plus efficaces dans le cas des grands exposants.

La méthode des fenêtres glissantes est une amélioration de la méthode M-aire qui est elle-même une optimisation de la méthode binaire. La méthode des fenêtres glissantes est la plus rapide mais elle est difficile à mettre en œuvre en une architecture hardware. Alors que la méthode binaire est la méthode classique pour calculer l'exponentiation modulaire ainsi elle permet d'éviter les pré-calculs.

La méthode binaire [13] est celle qui convient le mieux à notre champ d'étude en particulier la méthode binaire MSB car elle nous permet de calculer les grands exposants, et elle nous offre la possibilité de calculer en parallèle la multiplication et l'élévation au carré donc il y a une optimisation du temps d'exécution de 50%, de plus elle est simple à réaliser.

Dans ce qui suit nous allons présenter la méthode d'exponentiation Binaire qui répond le plus au besoin de notre architecture.

II .2.1. L'Exponentiation Binaire $S = x^e \ mod \ n$[10]

La méthode d'exponentiation binaire est aussi connue sous le nom de « Square and multiply method ». L'idée basique de la méthode binaire est de calculer x^y en utilisant l'expression binaire de l'exposant y.

$\mathbf{e= (e_{k-1}\, e_{k-2}\ldots\ldots\ldots\ldots\ldots e_1\, e_0)}$ **et** $\mathbf{e} = \sum_{i=0}^{k-1} \mathbf{e_i} \times \mathbf{2}^{i}$ $\mathbf{e_i} \in \{\mathbf{0}, \mathbf{1}\}$

Où k est le nombre de bit de l'exposant y

La méthode binaire ***est importante pour accélérer le calcul de l'exponentiation modulaire*** car cette dernière est divisée en une série d'élévation au carré et de multiplication. Il y a généralement deux algorithmes utilisés dans cette méthode qui peuvent convertir le calcul de l'exponentiation $\boldsymbol{S = x^e \ mod \ n}$, en une série de multiplications modulaires dans le crypto système RSA .

II.2.1.1 L'algorithme binaire LSB (Least Significant Bit) [12]

Dans cette methode, les bits de y sont lus du bit de poids faible au bit de poids fort et elle utilise une variable, en voici l'algorithme de la méthode LSB d'une manière générale:

Algorithme II.1 : Algorithme de la méthode Binaire LSB

Entrée: *e, X, m;*
Sortie: $S = X^e \bmod m$;
$S = 1; X = X;$
Début
Pour $i = 0$ ***à*** $k - 1$ ***faire***
Début
a. Si $(e_i = 1)$, ***alors*** $S = S \times X \bmod m$;
b. $X = X \times X \bmod m$;
Fin;
c. Si $e_{k-1} = 1$, ***alors*** $S = S \times X \bmod m$;
Fin.

II.2.1.2 L'algorithme binaire MSB (Most Significant Bit) [12]

Dans cette méthode, les bits de y sont lus du bit de poids fort au bit de poids faible et l'élévation au carré est exécutée pour chaque bit par contre la multiplication modulaire est exécutée uniquement pour les bits égaux à 1. Et voici l'algorithme de la méthode MSB :[12]

Algorithme II.2 : Algorithme de la méthode Binaire MSB

Entrée:e, *X, m;*
Sortie: $S = X^e \bmod m$;
$S = 1; X = X;$
Début
Pour $k - 1$ ***à*** $i = 0$ ***faire :***
Début
a. Si $(e_i = 1)$, ***alors*** $S = S \times X \bmod m$;
b. $X = X \times X \bmod m$;
Fin;
c. Si $e_{k-1} = 1$, ***alors*** $S = S \times X \bmod m$;
Fin.

Exemple :

e= 250_{10} =11111010_2 donc k=8 qui est le nombre de bit de l'exposant. La méthode binaire procède comme suit :

Tableau II.3- Déroulement de l'Algorithme MSB pour *e* =250 [12]

i	e_i	Etape a	Etape b
7	1	1	X
6	1	$(X)^2 = X^2$	$X^2 \times X = X^3$
5	1	$(X^3)^2 = X^6$	$X^6 \times X = X^7$
4	1	$(X^7)^2 = X^{14}$	$X^{14} \times X = X^{15}$
3	1	$(X^{15})^2 = X^{30}$	$X^{30} \times X = M^{31}$
2	0	$(X^{31})^2 = X^{62}$	X^{62}
1	1	$(X^{62})^2 = X^{124}$	$X^{124} \times X = X^{125}$
0	0	$(X^{125})^2 = X^{250}$	X^{250}

II.2.2. Comparaison entre la méthode LSB et MSB [12]

Les deux méthode binaires MSB et LSB [12] requièrent le même nombre de multiplication, soit 2× (k/2)+1×(k/2) = 1.5k multiplications. En espace mémoire, la méthode MSB utilise deux registres X et S contrairement à la méthode LSB qui utilise en plus des registres X et S, un autre registre tampon. (nécessite plus d'espace).

Dans la méthode MSB la multiplication décrite en (étape b) et l'élévation au carré décrit en (étape a) sont indépendants l'une de l'autre, alors que ces deux étapes peuvent s'exécuter en parallèle, ce qui impliquera que le le temps d'exécution sera optimisé à 50% comparé à la méthode LSB. D'où notre intérêt se porte sur la méthode Binaire MSB appelé : Left to Right.

II.2.3. Exponentiation binaire MSB basé sur la multiplication de Montgomery [13]

Dans ce module principal le choix a été porté sur la méthode d'exponentiation binaire MSB (***Square and Multiply) [15]*** qui permet aux deux Blocs ci-dessous de faire les multiplications modulaires de Montgomery.

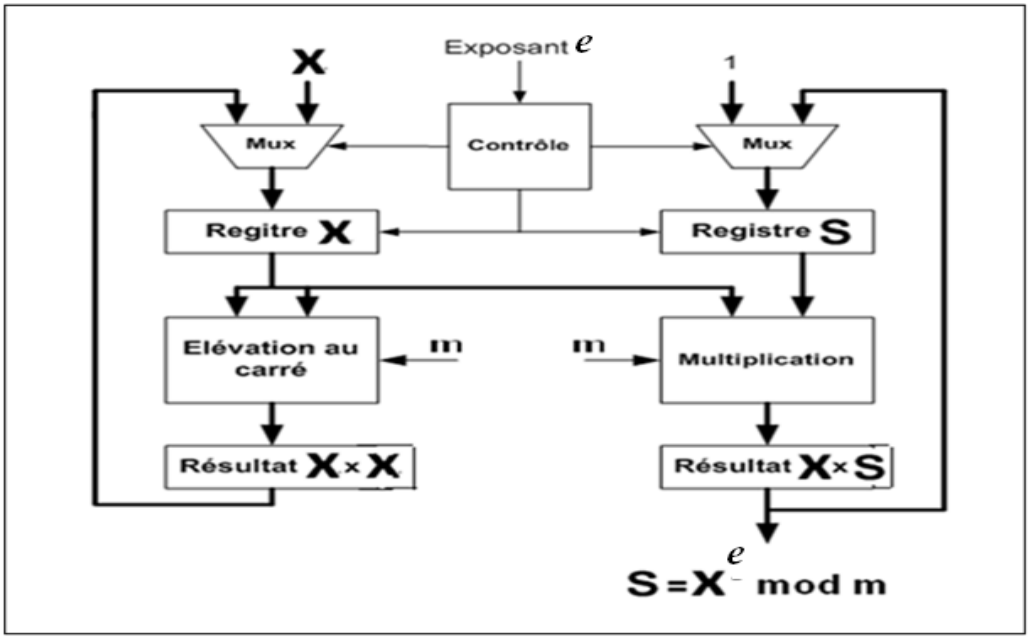


Figure II.1 : Schéma bloc de l'exponentiation modulaire binaire basé sur la multiplication de Montgomery [12]

Les registres X et S sont respectivement initialisées à 1 et X. L'exposant Y, représenté en binaire est utilisé pour contrôler les deux multiplications modulaires de Montgomery à savoir l'élévation au carré et la multiplication qui sont exécutées en parallèle indépendamment. Ces deux opérations se répètent pour tous les bits de l'exposant pour donner le résultat final ($\mathbf{S = X^e \bmod m}$) de l'exponentiation modulaire binaire. L'algorithme II.4 représente le déroulement de l'algorithme de la méthode binaire MSB basée sur la multiplication de Montgomery.[14]

L'algorithme II.4. La méthode binaire MSB basée sur la multiplication de Montgomery [14]

Entrée: *e, X, m, r^2;*
Sortie: *$S = X^e$ mod m;*
S=Montgomery (1, r^2, m); X = Montgomery (X, r^2, m);

Début
Pour *i = k – 1* ***à*** *0* ***faire***
Début
a. Si *(e_i = 1)* ***alors*** *S =Montgomery (S, X ,m) ;*
b. *X = Montgomery(X, X, m);*
Fin;
S = Montgomery (S, 1, m).
Fin.

II.2.4. Exponentiation binaire MSB basée sur Montgomery Ladder [12][13]

L'exponentiation de Montgomery Ladder a le même principe que l'exponentiation binaire de Montgomery, sauf que la différence réside dans le rôle des blocs des multiplieurs.

Dans l'exponentiation binaire classique, les deux blocs de multiplication de Montgomery garde la meme fonction durant le déroulement de l'algorithme ; l'un dédié à la multiplication et l'autre pour l'élévation au carré, les deux Blocs éffectuent la même fonction jusqu'à l'obtention du résultat final. Contrairement à l'exponentiation de Montgomery Ladder, les deux blocs de multiplication se permute de fonctions dés-que l'exposant e_i change de valeur :

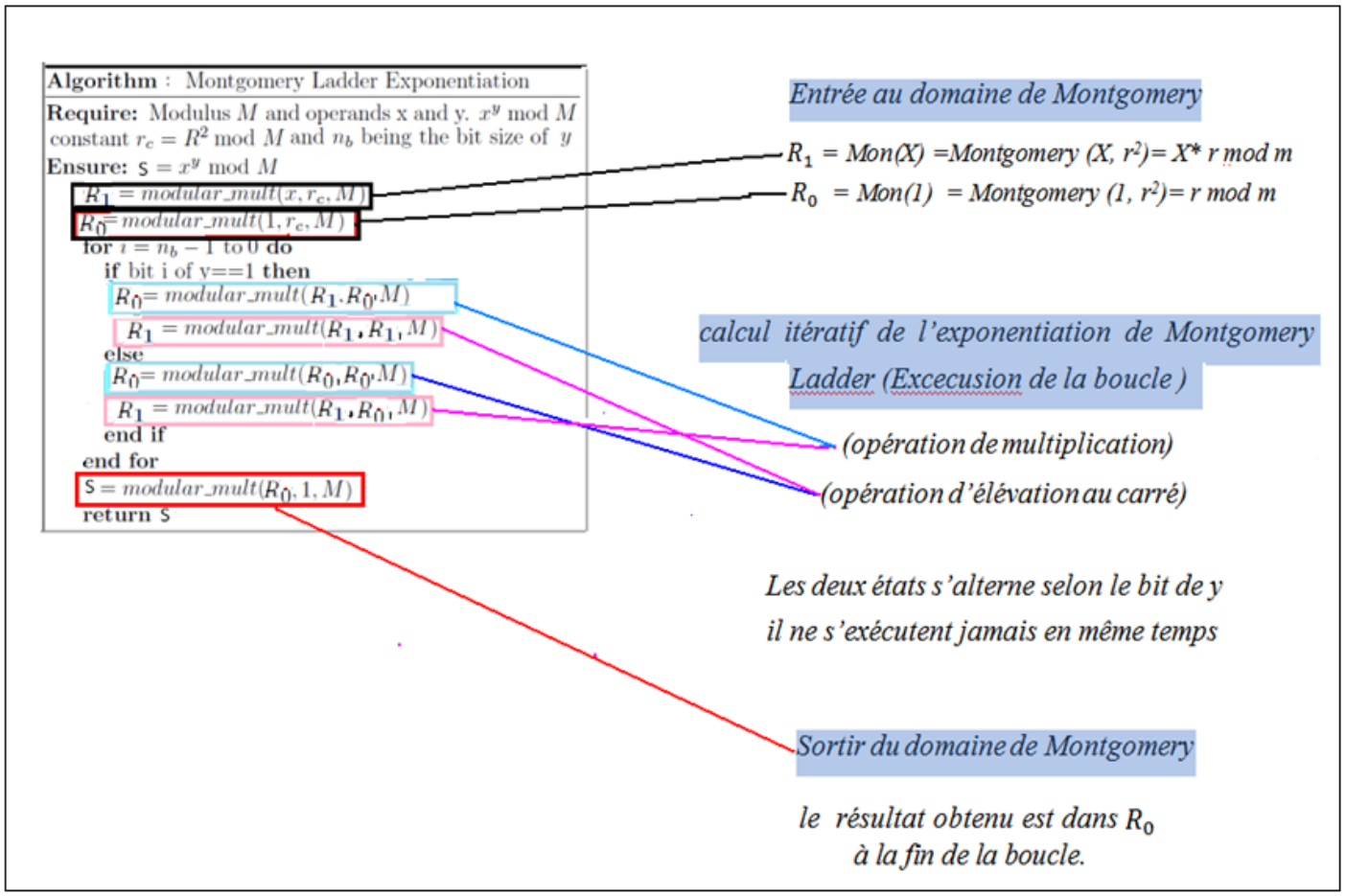
Quand $e_i = 1$

Le premier bloc éffectue la multiplication et le deuxième l'élévation au carré

Quand $e_i = 0$

Le premier bloc éffectue l'élévation au carré et le deuxième la multiplication, l'explication détaillé de l'algorithme est décrite ci-dessous :

Algorithme. II.5. Montgomery Ladder Exponentiation [12]



C'est un algorithme qui permet d'accélérer l'exponentiation modulaire[14] grâce à l'avantage du parallélisme (qui est une propriété intrinsèque de Montgomery Ladder), car c'est un Bi-processeur (comme le montre la figure II.2) il portent les critère ci-dessous :

- A chaque itération, les deux multiplications sont indépendantes, et s'exécutent en parallèle
- A chaque itération, les deux multiplications partagent des opérandes communes
- La valeur de ***R1/ R0*** est invariante le long de tout l'algorithme (= ***x***).

Il est classé parmi les meilleurs algorithme d'exponentiation rapide, vue sa grande protection contre les attaques :

- *Les timing –attaques.*
- *SPA (Simple Power analysis)*
- *Doubling attacks*

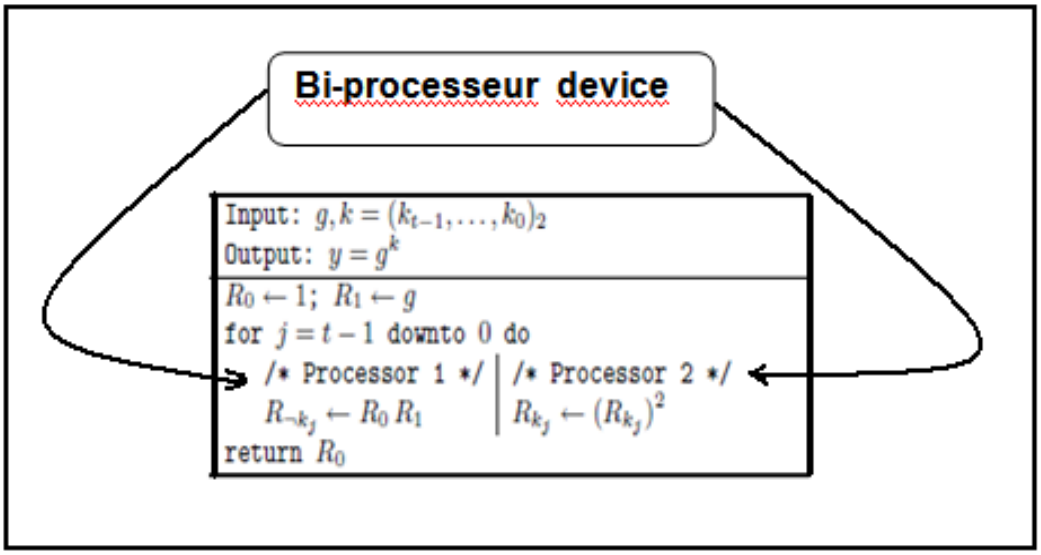


Figure II.2. Le Parallélisme de Montgomery Ladder [14]

II.2.4.1. Déroulement de l'algorithme de Montgomery Ladder [14] [15]

1.Entrée dans le domaine de Montgomery :

Initialisation des variables R_0 et R_1 respectivement à **1** et **x** dans le domaine de Montgomery en réalisant les multiplications de Montgomery appliquées aux constantes **1** et **x** avec $r^2=2^{2(k*n)} \bmod N$ (r : est la constante de Montgomery, dans notre cas $r = 2^{(k*n)}$, donc $r= 2^{(16*32)}$)

{ k= 16 bits(taille du paquet) , n =32 (nombre de mots)}.

$$R_0 = Mon(1) = Montgomery\ (1, r^2) = r \bmod m$$
$$R_1 = Mon(X) = Montgomery\ (X, r^2) = X^* r \bmod m$$

***2.Exécution de la boucle* :**

calcul itératif de l'exponentiation de Montgomery Ladder (Exécution de la boucle)

pour y_i=1

R_0 = *Montgomery* (R_0 , R_1 , *M*) (opération de multiplication)

R_1 = *Montgomery* (R_1 , R_1 , *M*) (opération d'élévation au carré)

Pour y_i=0

R_0 = *Montgomery* (R_0 , R_0 , *M*) (opération d'élévation au carré)

R_1 = *Montgomery* (R_0, R_1 , *M*) (opération de multiplication)

3.Sortie du domaine de Montgomery :

Sortir du domaine de Montgomery avec le résultat obtenu qui est dans S à la fin de la boucle**.**

S= *Montgomery* (R_0, *1, M*) indépendamment de y_i

L'entrée au domaine de Montgomery Ladder [20] et la sortie du domaine revient à faire deux multiplications supplémentaires ce qui signifie que l'exponentiation modulaire pour un exposant de k bits nécessitera (k+2) multiplications de Montgomery.

Ceci se traduit par :

- Une multiplication pour initialiser les variables R_0 et R_1 dans le domaine de Montgomery.
- k multiplications dans la boucle (multiplication et élévation au carré qui s'éffectuent en parallèle)
- Une dernière multiplication pour sortir du domaine de Montgomery.

L'élévation au carré de la $(k+2)^{éme}$ itération est faite d'une manière cohèrente, puisque la multiplication qui permet l'entrée et la sortie du domaine de Montgomery sont effectués par le bloc multiplieur. Ceci n'engendre ni diminution des performances du circuit ni erreur de calcul.

II.3. Multiplication Modulaire [19]

La multiplication modulaire étant la clé de voûte des algorithmes de cryptographie à clé publique. Toute amélioration de sa complexité a des répercussions positives sur la complexité des protocoles cryptographiques et spécialement le RSA, qui est l'objet de notre étude. C'est pour cela qu'elle a été fortement étudiée, et sa complexité a été largement améliorée. Nous allons dans ce qui suit abordé les algorithmes de multiplications les plus utilisés.

La multiplication modulo n de deux nombres entiers a et b est le reste de la division du produit C = a×b par le modulo n. l'opération de récupération du reste est dite réduction modulaire, elle est équivalente a une division euclidienne de l'entier C par une constante n, dans laquelle on désire récupérer que le reste r :

$$C= a\times b = r + q\times N \text{ avec } 0 \leq r < N$$

Le calcul du quotient q n'est pas obligatoire pour la réduction d'une multiplication modulaire, néanmoins certains algorithmes utilisent le calcul du quotient pour effectuer leur réduction. Ceci permet à de nombreux algorithmes de pré-calculer avantageusement différentes valeurs pour accélérer le calcul de la réduction.

Deux approches sont possibles sur la façon de calculer une multiplication modulaire. Une première catégorie intègre la réduction à la multiplication elle même. Cette technique a pour inconvénient d'imposer un algorithme pour éffectuer la multiplication. C'est pour cela que la majorité des algorithmes de multiplications modulaires[19] appartiennent à la seconde catégorie où la multiplication modulaire y est décomposée en une multiplication suivie d'une réduction modulaire.

Les algorithmes de multiplication modulaire que nous allons présenter sont dits généralisés car ils fonctionnent pour tous les modulos, leur complexité ne tient pas compte du fait que le modulo appartienne ou pas à une classe de nombres spécifiques.

II.3.1. Les différentes algorithmes de multiplication modulaire [13]

Plusieurs algorithmes de multiplication modulaires existent, tel que ; Algorithme de Taylor, Blakley, Takagi, Barett, Quisquater, et Montgomery. Ces derniers sont tous des algorithmes généralisés fonctionnant pour n'importe quel modulo. Cependant on relève une différence majeure entre la manière dont est calculée une multiplication modulaire, les deux algorithmes de Montgomery et Barrett commencent par faire la multiplication c =a* b puis réduisent le résultat de cette multiplication, ils sont

dits « ***algorithmes à réduction*** ». Par contre, ceux de Blakley et Takagi réduisent leurs calculs au fur et à mesure de la multiplication, ils appartiennent à la classe des algorithmes à «***réduction intégrée*** » .

L'algorithme de Montgomery entrelacé et sans soustraction finale appartient à une troisième catégorie celle des **« *algorithmes entrelacés* »**

Les algorithmes intégrés Blakley et Takagi ne sont pas trop complexes, et n'exigent que des additions mais sont conseillés pour des petits modulos, car la taille des additionneurs ainsi que le nombre d'itérations à faire les rendent peut performants pour de grands modulos. Les deux algorithmes à réduction Montgomery et Barrett ont une complexité plus intéressante en comparaison avec celle des algorithmes intégrés, leurs seul inconvénient est qu'ils exigent un pré-calcul ***: l'inversion et la division respectivement.***

Vu que la division est nécessaire pour le calcul de l'algorithme de Barrett, il n'est conseillé que dans le cas d'une simple réduction (modulo pas trop grand). Bien que l'algorithme de Montgomery exige une représentation particulière, il est recommandé dans le calcul intensif tel que l'exponentiation modulaire, car c'est le plus rapide. De plus le temps de conversion des nombres dans la représentation de Montgomery est négligeable par rapport au calcul nécessaire pour l'exponentiation modulaire ce qui rend son seul inconvénient négligeable devant ces avantages. Le tableau II.6 résume les différentes caractéristiques des différents algorithmes de multiplication modulaire.

Algorithme. II.6. Caractéristiques des algorithmes de multiplications modulaires [14]

Algorithme	Type	Pré-calcul	Représentation	Complexité
Blakely	Intégré	Aucun	Simple	3n add(n)
Takagi	Intégré	Aucun	Borrow save	3(n+1)
Barrett	Réduction	Division	Simple	3 mult(n) + 2add(n)
Montgomery	Réduction	Inversion	mon(x)=x×r mod N	2 mul(n)+ mulBas(n) +2add(n)
Montgomery entrelacé avec soustraction finale	Entrelacé	Aucun	mon(x)=x×r mod N	(n)add(n) + (n+1)add(n+1).
Montgomery entrelacé sans soustraction finale	Entrelacé	Aucun	mon(x)=x×r mod N	(n+2)add(n+1) + (n+2)add(n+2).

Puisque la comparaison des algorithmes à réduction affirme que celui de Montgomery est le plus adapté aux calculs intensifs de la multiplication modulaire, notre étude sera consacré a cet algorithme par ce qui s'en suit.

II.3.2. L'algorithme de Montgomery [17]

Les méthodes de multiplication de Montgomery constitue le cœur de l'opération d'exponentiation modulaire {qui est une séries de multiplication modulaire de Montgomery (méthode binaire).Le problème majeur de la multiplication modulaire est la réduction par le modulo qui est une division très coûteuse dépendant de la taille du modulo M.

En 1985, Peter Montgomery [21] a proposé un algorithme qui calcule le produit modulaire sans passer par cette division. Le principe consiste à remplacer cette opération par une division par la base $R=\beta^n$. (ou n est le nombres de chiffres qui compose le modulo N). Il est clair que cette division n'est qu'un simple décalage. R est appellé ***Constante de Montgomery***. Cet algorithme propose de calculer :

{ $S = A* B * R^{-1}$ mod N }, qui est potentiellement plus rapide qu'un calcul de multiplication modulaire , au lieu de calculer { *$S = A* B$ mod N }* qui nécessite plusieurs divisions par N (c'est couteux en terme de temps).

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

Soit N et R deux entiers tel que :

- *N est représenté sur n chiffres*
- *R est choisi tel que $R \geq \beta^n$, avec R> N et PGCD(R,N)=1*

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

Soit $N'=(-N)^{-1}$ mod R (N'est pré-calculé une seule fois)

Si on considere : m= ($C * N'$) mod R , alors :

1. *S=(C+m*N)/R est un entier*
2. *S=($C*R^{-1}$) mod N*
3. *S<2*N*

Remarque1 :

- En cryptographie asymétrique , la condition PGCD(R,N) =1 est souvent satisfaite, car N, est un nombre premier, ou il est issu de la multiplication de deux nombres premiers
- La multiplication modulaire de Montgomery nécessite un pré-calculé, en l'occurrence N' qui est déterminé par : $N' * -N = 1 \bmod R$. L'algorithme de Montgomery est présenté comme suit :

Algorithme II.7. L'algorithme de Peter Montgomery

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

Données : N' *et* R *avec* $R \geq \beta^n$, $N' * -N = 1 \bmod R$, $R' * R = 1 \bmod N$

Variables : *C, m*

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

Début

$C \leftarrow A * B$

$m \leftarrow (C * N') \bmod R$

$S \leftarrow (C + m * N)/R$

Si $S \geq N$ *alors* $S \leftarrow S - N$

Retourne S

La complexité de l'algorithme de Montgomery [17] est de trois multiplications, une addition et une soustraction. Etant donné que la seconde multiplication $(C * N' \bmod R)$ est calculé en modulo R(division par la base). Ceci revient a prendre en considération uniquement la moitié de la multiplication $(C * N')$ (multiplication partie basse), ainsi la complexité de l'algorithme II.7 est donné par :

$$C_{algo1} = 2\,mul(n) + mulBas(n) + add(n) + sous(n)$$

mulBas (n) : multiplication partie basse

Le résultat S = (C* R^{-1}) mod N de la multiplication modulaire de Montgomery montre que celle-ci calcule non pas $A * B \bmod N$, mais une réduction avec un facteur supplémentaire R^{-1}. Pour parvenir à la réduction de la multiplication (*A*B)* en modulo *N*, une opération supplémentaire doit être éffectuer afin de supprimer ce facteur. Celle-ci est réalisé en multipliant $(A * B * R^{-1} \bmod N)$ *par* $(R^2 \bmod N)$ qui est une quantité pré-calculé.

L'opération en question n'est autre qu'une seconde multiplication de Montgomery, pour donner :

$$S = [((A * B * R^{-1}) \bmod N) * (R^2 \bmod N) * R^{-1}] \bmod N = A * B \bmod N$$

II.3.2.1. Propriété de Montgomery [19][23]

Le fait d'avoir un facteur supplémentaire dans le résultat de la multiplication modulaire de Montgomery, en l'occurrence le facteur R^{-1} , cela pose un probléme pour le calcul intensif de la multiplication modulaire. De ce fait, Montgomery utilise une représentation particulière dite ***notation de Montgomery,*** notée par *mon (x).* Cette notation consiste à associer à chaque valeur x inférieure a N, une valeur égale a *mon(x)= (x*R) mod N*, avec R = β^n. Cette notation donne quelques propriétées utilisés notamment dans le calcul de l'exponentiation modulaire :

a. ***Stabilité***: La multiplication modulaire de Montgomery est stable par rapport à la multiplication

Montgomery (*mon(A), mon(B))=(mon(A) * mon(B) * R^{-1})mod N*

$= A * R * B * R * R^{-1} \mod N$

$= A * B * R \mod N = mon(A*B)$

b. ***Conversion :*** Les conversions entre la représentation de Montgomery et la représentation classique s'éffectuent en utilisant la multiplication de Montgomery elle-même

1. Représentation classique vers la représentation de Montgomery

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

$= (A*R) \mod N = mon(A)$

2. Représentation de Montgomery vers la Représentation classique

$mon(A) \rightarrow A: Montgomery (mon(A), 1) = (mon(A) * 1) \mod N$

$= (A * R * R^{-1}) \mod N = A \ mon \ N$

La stabilité et la représentation de Montgomery semblent être intéressantes lorsque le calcul intensif de la multiplication modulaire est exigé. C'est notamment le cas de l'exponentiation modulaire ***S= A^e mod N***. En effet, une première conversion de la représentation classique de l'opérande A vers la représentation de Montgomery (appellé aussi domaine de Montgomery) doit être effectué au début du processus de l'exponentiation modulaire. Cette étape est éffectué par une simple multiplication de Montgomery, en introduisant *A et($R^2 mod N$)* comme données d'entrés. On calcul ainsi Montgomery *(A, $R^2 mod N$). R^2* est une constante, elle est pré-calculé une seule fois pour chaque modulo N . Puis les calculs peuvent se faire dans le domaine de Montgomery tout au long du processus de l'exponentiation et en utilisant l'algorithme II.7. Finalement, une conversion vers la représentation classique doit être réalisée. Celle-ci est éffectué par une multiplication de Montgomery en prenant le résultat obtenu de

l'étape précédente et ''1'' comme opérandes d'entrées. Les étapes de calcul et de conversion sont résumées sur la figure II.3

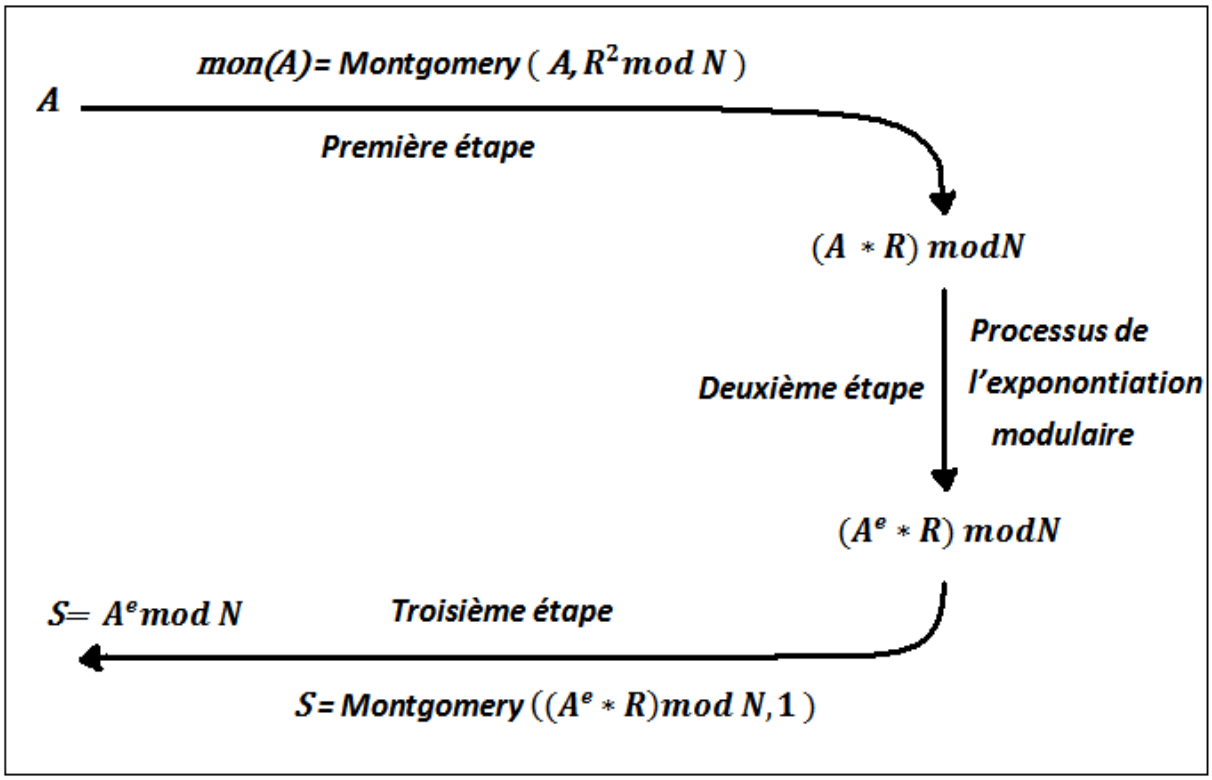


Figure II.3. Etapes de calcul de S= A^e mod N en utilisant la multiplication de Montgomery[20]

II.3.2.2. Variante de la multiplication de l'algorithme Montgomery (MMM) [19],[20]

Dans le RSA les clefs de déchiffrement sont généralement de l'ordre de 1024 bits. Cependant, quand la surface d'implémentation est limité, l'exécution de la multiplication modulaire de Montgomery (MMM) se trouve impossible.

En effet, afin de faciliter l'implémentation de la MMM, des modifications ont été apporté sur son algorithme original. Celles-ci ont abouties à des variables qui peuvent être implémenter suivant un mode parallèle ou sériel. Généralement l'utilisation du premier mode résulte des architectures encombrantes, pour ce faire, nous nous somme orienté vers une conception d'une approche qui nous permettra d'optimiser la surface du circuit et qui sera dédié au grandes tailles de clé.

Dans ce qui suit, nous allons définir en premier lieu, la multiplication de Montgomery entrelacé puis nous représentons la taille des résultats intermédiaires.

Cette étape est très importante pour une implémentation matérielle, car c'est à travers celle-ci, qu'on pourra déterminer la taille des bus d'interconnexions et de l'espace mémoire si le stockage des résultats intermédiaires est nécessaire. Cette méthode a été développé dans le but d'implémenter une architecture qui offre un bon compromis entre la surface occupé et le temps d'exécution.

II.3.2.2.1.La multiplication modulaire de Montgomery entrelacée (MMME) [17],[19]

L'algorithme de Montgomery sans soustraction finale, représenté par l'algorithme entrelacé qui facilite le calcul sur de grands nombres est le plus répondu. La suppression de la soustraction est très avantageuse pour la réduction de la surface occupée par l'architecture matérielle

L'idée d'entrelacer la multiplication à la réduction modulaire est basée essentiellement sur la représentation du multiplieur A. La méthode peut être généralisé en base $\beta = 2^k$ (k=16) : L'algorithme est détaillé ci-dessous .

Algorithme II.8. L'algorithme Montgomery (MMME) [19]

Entrées *:* $A= \sum_{i=0}^{n} a_i * \beta^i$, $B= \sum_{j=0}^{n} b_j * \beta^j$, $N= \sum_{j=0}^{n-1} n_j * \beta^j$ *, avec 0<A,B<2*N*

Pré-calculés *:* $m' = -m_0^{-1} \bmod \beta$*, avec* $\beta=2^k$ *(k=16 bits), gcd(N, β)=1*

Dans notre cas $m'=n\_c$ ***:*** *constante de multiplication (calculé en partie basse)*

m_0 *= est le mot le moins significatif du modulo N*

$r=2^{(n*k)+2}$ *avec k>2 (n= 32 mots)*

Variables intermédiaires *:* $m_i, S_i = \sum_{j=0}^{n} s_{i,j} \beta^j$

Sortie: $S = \sum_{j=0}^{n} s_{n,j} * \beta^j = S_{n+2} = A^* B * R^{-1} \bmod N$

Début

$S_0=0$

Pour *i* ***de*** *0* ***à*** *n* ***faire :***

$m_i = ([S_{i,0} + (a_i * b_0)] * m') \bmod \beta$ ……(1)

$S_{i+1} = [S_i + (a_i * B) + (m_i * N)] / \beta$ ……(2)

Fin pour.

Retourne S

Cet algorithme utilise trois opérandes d'entrées A, B et N où A et B sont sur 34 mots de 16 bits et N est sur 32 mots de 16 bits. La variable S est utilisée pour stocker les résultats intermédiaires, le résultat final est obtenu pour i=n+1, c'est-à-dire S_{n+1}

- La seule condition que requis cet algorithme est que β soit premier avec N. l'éxécution de cet algorithme peut être résumé dans les points suivants : [13]
- La détérmination de m_i qui est le LSB de la somme $(S_{i,0} + (a_i * b_0))$, cette dernière est obtenue par une simple réduction de cette somme en modulo β à chaque itération
- Une fois avoir calculer m_i . S_{i+1} $sera$ obtenu par l'expression : $[S_i + (a_i * B) + (m_i * N)] / \beta$.
- A chaque itération, les résultats intermédiaires S_{i+1} peuvent être supérieure a N , $S_{i+1} < (2*N)$, donc ils sont représentés sur n+1 mots. La soustraction finale est nécessaire pour assurer que le résultat de la multiplication de Montgomery soit strictement inferieure à N ($S_n < N$). Cette condition est très importante quand il s'agit d'une succession de multiplication modulaires, ou S sera utilisè comme opérande d'entrée lors de la prochaine opération.

Complexitè de l'algorithme :

En terme de complexité de calcul, comme cette algorithme est définie d'une manière générale pour une base $\beta=2^k$, ce qui signifie que les digits (a_i, m_i) sont représentés dans le système $\{0,1,\ldots, \beta-1\}$. Par conséquent, les calcules des termes $(a_i * B)$ et $(m_i * N)$ seront forcément éffféctués en utilisant des multiplieurs $k*n$, ainsi on peut considérer la complexité de ce algorithme équivalente à :

$$C_{algo2} = n*[(2\ mul(k,n)) + 2\ add(n+k+2) + mulBas(k)] + sous(n+1)$$

Le déroulement de l'algorithme [17] voir figure II.4

On peut donc résumer l' éxécution de cet algorithme comme suit :

a. Initialisation

Mise à zéro de la variable S , et lecture des opérandes A, B et N

c. Exécution de la boucle n+1 fois

Instruction 1 : Calcul de la variable m_i après une multiplication $a_i \times b$ (1bit×(n+2) bits) et une addition $S_i + a_i \times b$ sur (n+2 bits).

Instruction 2: Calcul de S_{i+1} après une multiplication $m_i \times N$ (1bit×512 bits) et une addition sur (n+2) bits.

La somme $S_i + a_i \times b$ étant précédemment calculée dans *Instruction 1,* il reste à calculer la division par $\beta = 2^k$ qui revient à un simple décalage des mots de k bits (16 bits dans notre cas)

d. ***Lecture du résultat de la multiplication*** : obtenu à la fin de la dernière itération dans S. voir figure II.4

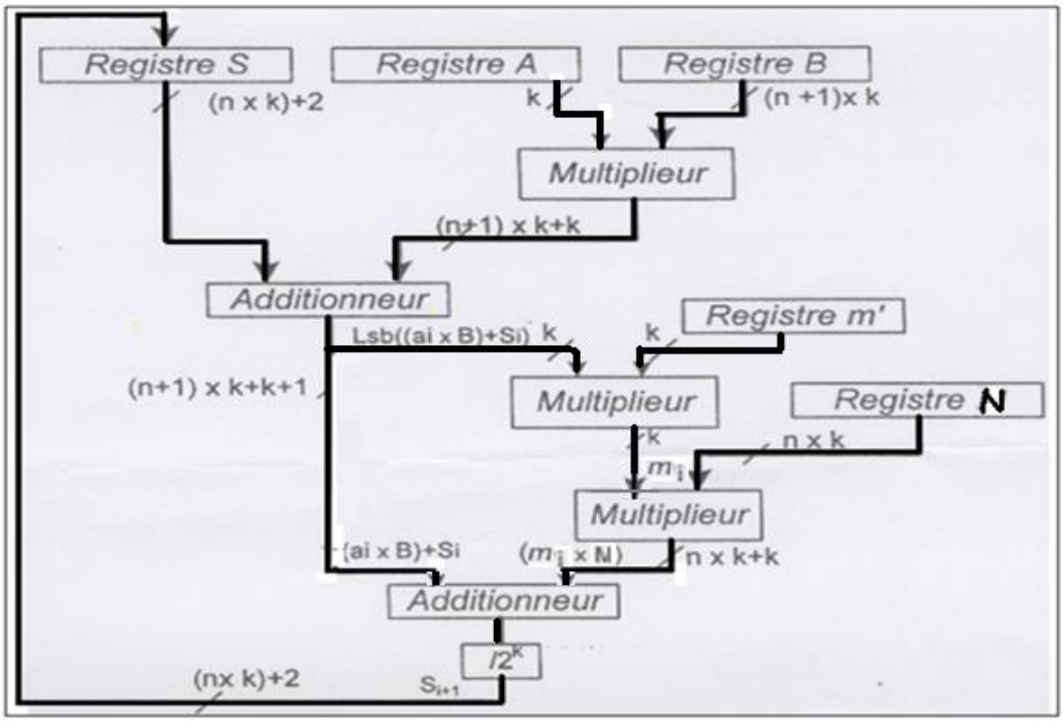


Figure II.4. Architecture interne de la MMME

II.3.2.2.2. Explication de l'algorithme de Montgomery MMME par Blocs [16] [17]

Dans le but d'éviter l'utilisation de long registres internes, l'approche proposé repose d'une part sur les Blocs Select-RAM disponibles sur la majorités des Circuits FPGA pour stocker les opérandes. D'autres part sur les chemins de propagation de retenues et les Blocs Multiplieurs 18 x 18 bits.

Les opérandes A, B, N sont stockés dans des mémoires à partir du mot le moins significatif au mot le plus significatifs. Pour atteindre la précision des opérandes qui est de *(n+1) * k bits*, une itération (i) nécessite (n+1) lectures mémoires. Les résultats intermédiaires peuvent être stockés dans des FIFO (First In, First Out). L'utilisation de ce mode de stockage est due au faite que le résultat de l'itération (i) sera utilisé comme entrée pour l'exécution de l'itération (i+1). Voir lgorithme. II.9

Algorithme II.9. Algorithme de la MMMSF ($\beta=2^k= 2^{16}$) [13]

Entrées: *A*= $\sum_{i=0}^{n} A[i] * 2^{i*k}$,*B*= $\sum_{i=0}^{n} B[j] * 2^{j*k}$,*N*= $\sum_{i=0}^{n} N[j] * 2^{j*k}$, *avec N[n]=0*

Pré-calculés *:* $m' = -m_0^{-1} \bmod \beta$, *avec* $\beta=2^k$ *(k=16 bits), gcd(N, β)=1*

Dans notre cas m'*=n_c* ***:*** *constante de multiplication (calculé en partie basse)*

m_0 *= est le mot le moins significatif du modulo N*

$r=2^{(n*k)+2}$ *avec k>2 (n= 32 mots)*

Variables intermédiaires : $S_{i+1} = \sum_{j=0}^{n} S[j]_{i+1} * 2^{j*k}$

$P1_i = \sum_{j=0}^{n} P1[j]_i * 2^{j*k}$ $C1_i = \sum_{j=0}^{n} C1[j]_i * 2^{j*k}$ $P2_i = \sum_{j=0}^{n} P2[j]_i * 2^{j*k}$

$Z_i = \sum_{j=0}^{n} Z[j]_i * 2^{j*k}$ $L_i = \sum_{j=0}^{n} L[j]_i * 2^{j*k}$ cy_1^j , cy_2^j , cy_3^j , cy_4^j

Sortie: $S = \sum_{j=0}^{n} S[j]_{n+1} * 2^{j*k} = A * B * R^{-1} \bmod N$

Début

1. $S_0 = \sum_{j=0}^{n} S[j]_0 * 2^{j*k} = 0$
2. $C1[-1] = C2[-1] = 0$
3. $cy_1^{-1} = cy_2^{-1}, = cy_3^{-1}, = cy_4^{-1} = 0$
4. *Pour* ***i*** *de* ***0*** *a* ***n*** *faire :*
5. $P1[0]_i$ *=A[i]*B[0]*
6. $Z[0]_i = P1[0]_i + S[0]_i$
7. $m_i = (Z[0]_i * m') \bmod 2^k$
8. *Pour* ***j*** *de* ***0*** *a* ***n*** *faire*
9. $(C1[j]_i , P1[j]_i)$ *=A[i]* B[j]*
10. $(cy_2^j 2^k , cy_1^j 2^k, Z[j]_i) = P1[j]_i + C1[j-1]_i + S[j]_i + cy_1^{j-1} + cy_2^{j-1}$
11. $(C2[j]_i , P2[j]_i) = m_i * N[j]$
12. $(cy_4^j 2^k , cy_3^j 2^k, L[j]_i) = Z[j]_i + P2[j]_i + C2[j-1]_i + S[j]_i + cy_3^{j-1} + cy_4^{j-1}$
13. $S[j]_{i+1} = L[j+1]_i$ $(S[j]_{i+1} = L[j+1]_i / 2^k)$

Fin pour

Retourn S = S_{n+1}

L'éxécution de cet algorithme est basé sur deux indice (i) et (j) désigne d'une part, la i^{eme} itération de l'algorithme et d'autre part, le i^{eme} mot A[i] de l'opérande A.

L'indice (j) permet de positionner les j^{eme} mots de B, N, S_i, C'est à dire , *B[j], N[j], S[j].*

Le déroulement de chaque itération (i) est décrit comme suit :[13]

A la fin de l'itération (i-1), ou j=n, l'exécution de l'itération (i) commence à partir de la lecture des mots *A[i], B[0], N[0] et S[0].* La lecture de *B[j], N[j] et S[j]* se répète tout au long de cette itération, en faisant incrémenter j de 0 a n. Les opérations arithmétiques sont exécutés sur une précision de k bits.

$P1[j]_i$ *:* est codé sur 2* k bits, car en général, la multiplication de deux nombres codés sur k bits donne comme résultat un nombre codé sur 2* k bits - -> $P1[j]_i$= *A[i]*B[j]*

$Z[j]_i$: est codé sur kbits car l'addition des trois opérandes codés sur k bits, donne un résultat codé sur k bits et deux retenues de poids 2^k.

$(cy_2^j 2^k\ ,\ cy_1^j 2^k, Z[j]_i) = P1[j]_i + C1[j-1]_i + S[j]_i + cy_1^{j-1} + cy_2^{j-1}$

cy_1^{j-1}, cy_2^{j-1} *:* correspondent aux retenus générés par les additions éffectuées à l'itération *(j-1),* les retenues générées à l'itération *(j)* sont du poids 2^{k*j}.

Aprés l'obtention du j^{eme}de Z_i, on passe au calcul de L_i dans la seconde étape de l'itération (i). Il est à noter que ce dernier est en fonction de m_i qui est determiné dés la parution du premier mot de Z_i et reste constant pour chaque itération (i).

En effet m_i est calculé à partir de la partie basse de la multiplication $Z[0]_i * m'$, ce qui signifie qu'il est codé sur k bits (voir ligne 7 de l'algorithme).

Dans cette deuxième étape de l'itération (i), on peut considérer que l'obtention de L_i est identique au calcule de Z_i, ceci est du au fait que l'expression (1) et (2) de l'algorithme.2.2 sont semblables. C'est à dire, elles sont constitués par le même types d'opérations (une multiplication suivie par une addition).

cy_3^{j-1}, cy_4^{j-1} *:* correspondent aux retenus générées par les additions effectuées à l'itération *(j-1),* les retenues générées à l'itération *(j)* sont de poids 2^{k*j}.

Finalement, une fois avoir obtenue L_j, le calcul des résultats intermédiaires est donné par la ligne (13) de l'algorithme II.9. ($S[j]_{i+1} = L[j+1]_i / 2^k$) est éffectué par un simple décalage de k bits vers les poids les plus faible de L_i . ainsi le j^{eme} mot de S_{i+1} en l'occurrence $S[j]_{i+1}$ correspond au $j-1^{eme}$ mot de L_i (voir ligne 13). La j^{eme} boucle est toujours utilisé pour décaler le résultat par un mot à droite (i.e : division par 2^k) .

L'exécution de l'algorithme MMMSF pour deux itérations successives j et j+1 est presenté dans la Figure II.5

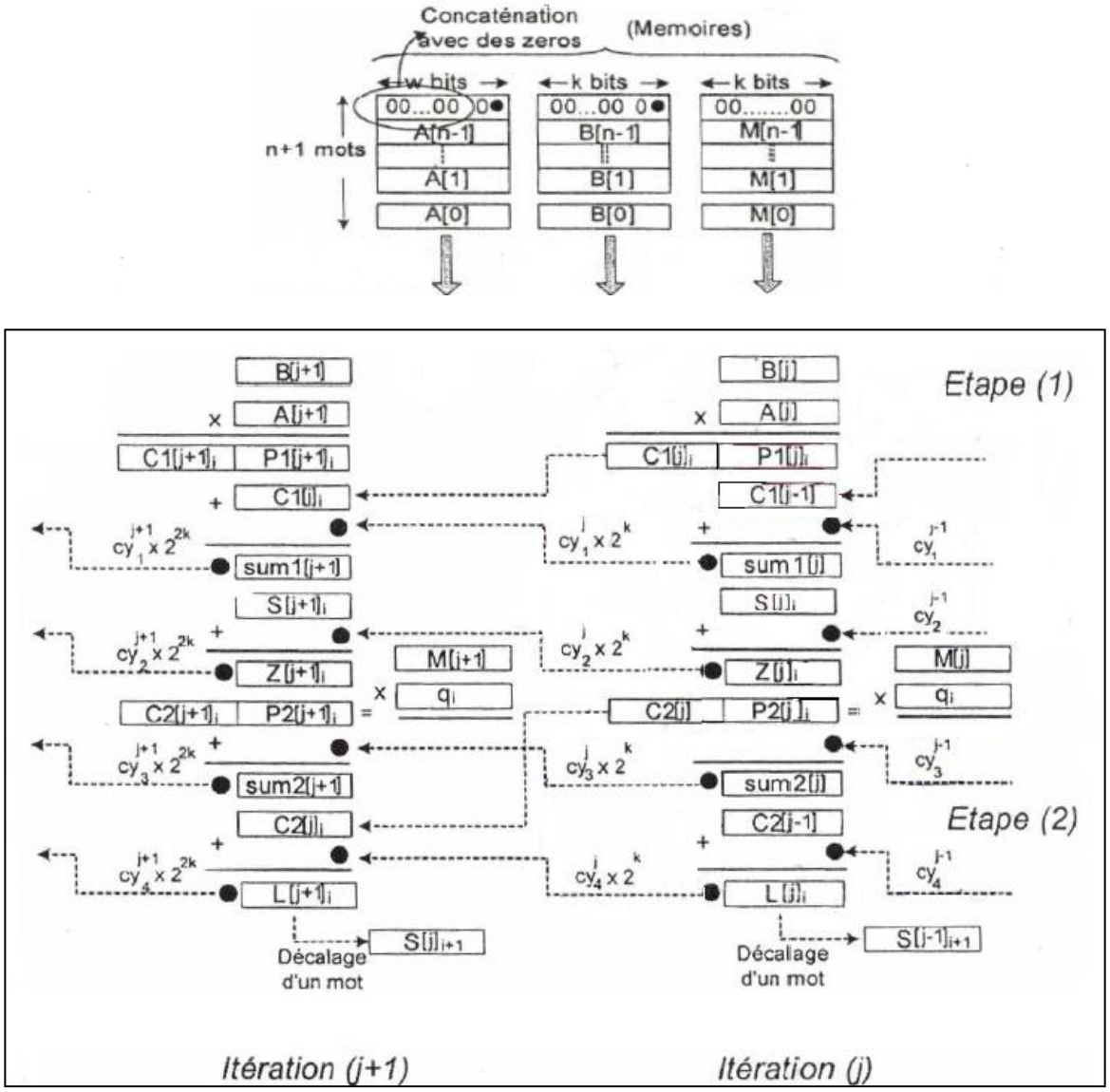


Figure.II.5 Exécution de l'algorithme de Montgomery MMMSF pour deux itérations (j) et (j+1)[13]

II.3.2.2.3. Architecture Systolique [17]

L'architecture systolique, introduite en 1978 par Kung et Leiserson, s'est révélé être un outil puissant pour la conception de processeurs intégrés spécialisés. Celle-ci est présentée comme étant un réseau de cellules élémentaires identiques et localement interconnectées (voir la figure II.6). Chaque cellule reçoit des données en provenance des cellules voisines, éffectue un calcul simple, puis transmet les résultats toujours aux cellules voisines, Dans un temps de cycle plus tard. Seules les cellules situées à la frontière du réseau communiquent avec le monde extérieur.

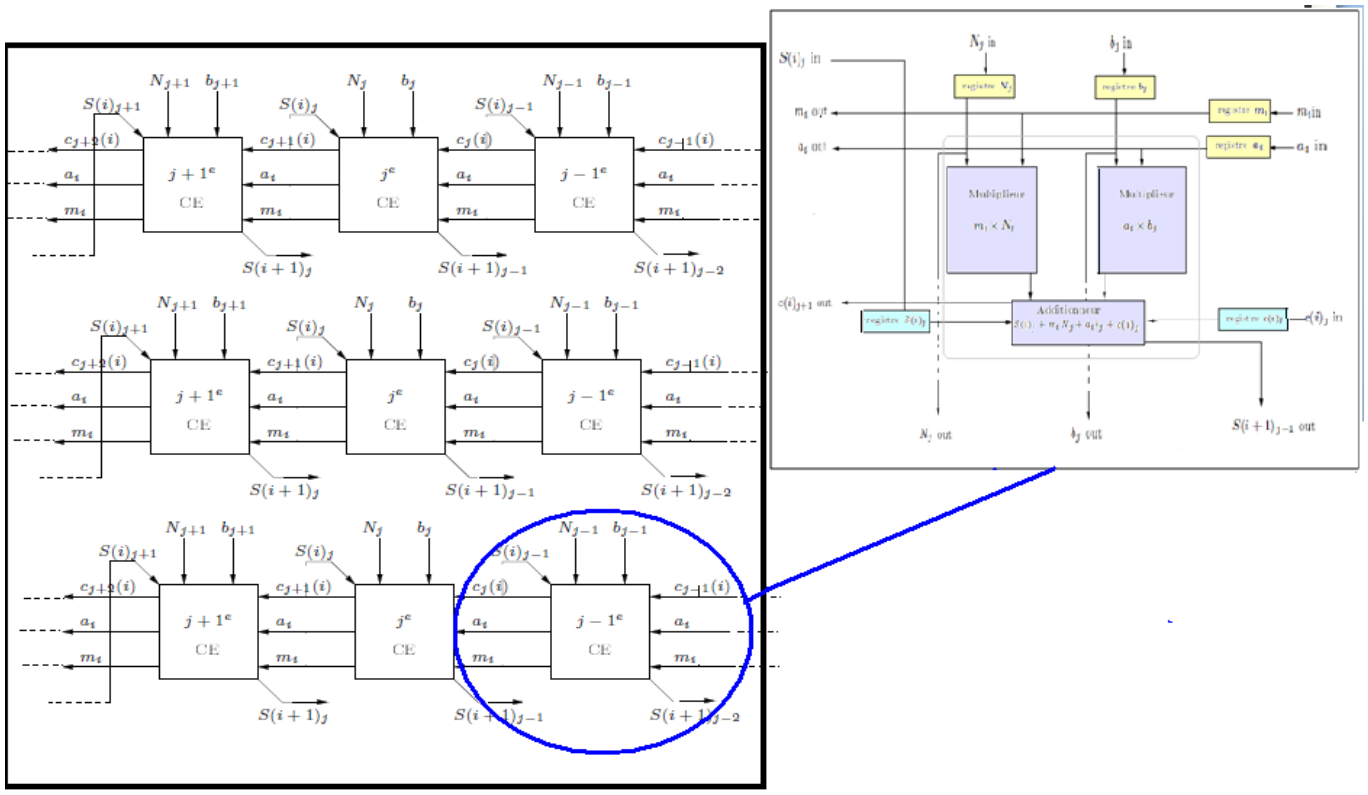


***Figure II.6. Pipeleine des Cellule elementaires de l'architecture Systolique du Bloc de multiplication de Montgomery* [17]**

II.3.2.2.3.1. Définition du PE [16] [17]

Le principal élément calculateur (PE) calcule le mot $S(i+1)_{j-1}$ du PE (j). Le bloc de multiplication est constitué de PE's dupliqués en cascade (n+1 fois) en ligne et en colonne. La duplication de toute la chaine verticalement constitue le calcul itératif du le mot $S(i+1)_{j-1}$.

L'architecture de base de ces éléments est illustrée par la figure II.7

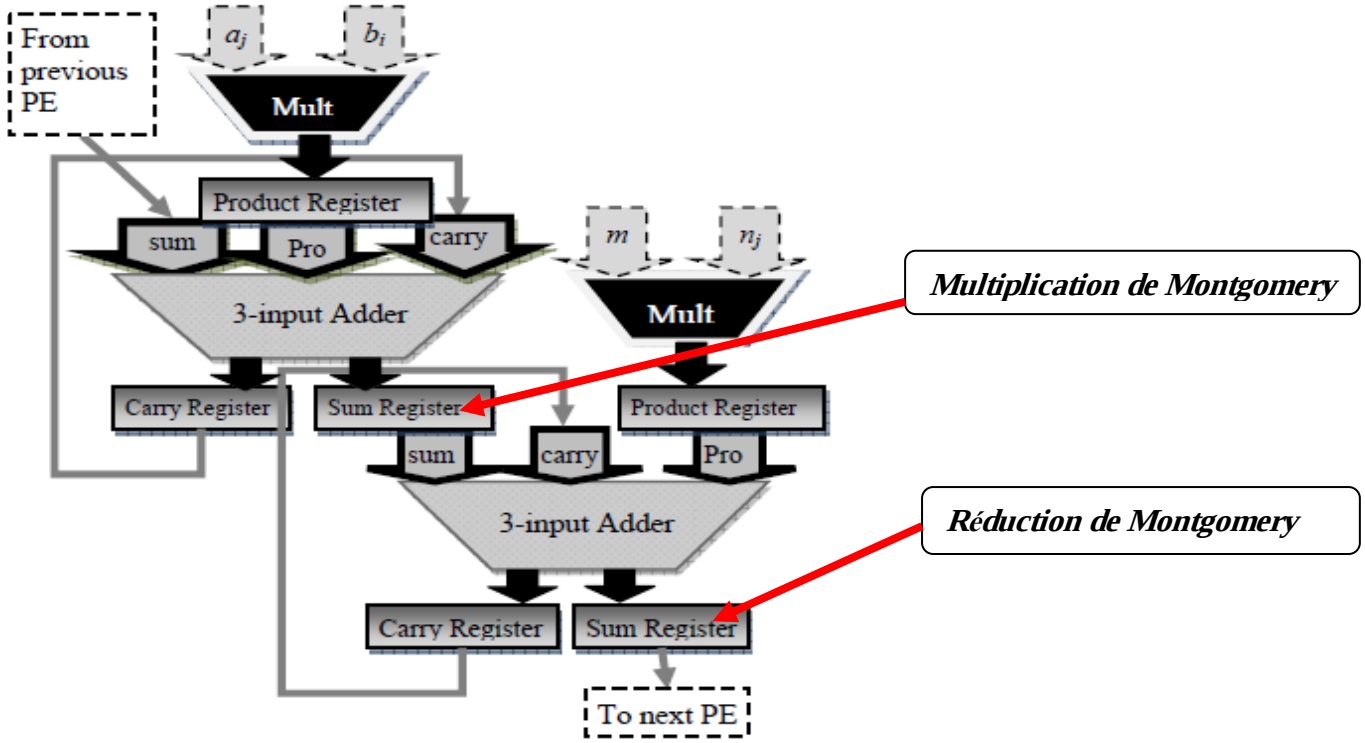


***Figure II.7 Description général de la Cellule élémentaire d'une architecture Systolique (PE) Processing élément* [16] [17]**

Le PE est l'élément responsable de l'éxécution d'une seule itération dans une boucle de l'algorithme MMMSF II.9. Chaque PE est composé de (2 Muliplieurs + 2 Additionneurs + 6 Registres de (16 bits)) voir (Figure .II.7)

La taille des registres : $a_i,\ m_i,\ bj,\ Nj,\ C(i)_j, S(i)_j$ est de 16 bits. On remarque ici la présence de deux sens de propagation différents : Si la j^{eme} cellule réalise cette opération, le résultat $S(i+1)_{j-1}$ est transmis a la cellule précédente et le carry $C(i)_{j+1}$ à la cellule suivante. Le PE calcule l'équation de Montgomery ci-dessous :

$$\boldsymbol{S(i+1)_{j-1} + C(i)_{j+1} = S(i)_j + a_i * bj + m_i * N_j + C(i)_j}$$

Quand le premier PE calcule la première itération de l'algorithme de Montgomery et génère le premier mot du résultat intermédiaire (le mot le moins significatifs), le prochain PE calcule la deuxième itération de la boucle de l'algorithme en tenant compte de la valeur précédante obtenue à partir du premier PE.

II.3.2.2.4. La méthode CIOS de Montgomery [18]

Il existe plusieurs méthodes algorithmiques pour exécuter la multiplication modulaire de Montgomery, comme il existe plusieurs méthodes pour faire la multiplication de Montgomery. Ces derniers sont classés en deux facteurs principaux :

1) ***Approche séparé ou intégré*** :

   Multiplication et réduction séparé : consiste en premier lieu à multiplier d'abord ($a_i * bj$), pour tout le nombre d'itérations existant dans l'architecture et par la suite procéder à la réduction en calculant le produit : ($m_i * N_j$) **.** Cette approche est appellé séparé car les deux opérations s' exécute séparément dans l'algorithme.

   Multiplication et réduction intégré : Consiste à alterner entre l'opération de la multiplication ($a_i * bj$) et celle de la Réduction ($m_i * N_j$) au sein d'une même itération (même boucle).

   Cette approche est appellé ''intégré''car les deux opérations s'éxécutent l'une après l'autre dans l'algorithme.

2) ***Operand/Product Scanning*** :

   Opérand Scanning : quand une boucle éxtérieure se déplace à travers tous les mots d'une seule opérande pour faire la multiplication.

   Product Scanning : quand une boucle éxtérieure se déplace à travers les produits des mots d'une seule opérande pour faire la multiplication

Le deuxième facteur est indépendant du premier. La multiplication peut avoir une forme { operand scanning , ou product scanning } et la Réduction peut avoir une autre forme { operand scanning , ou product scanning}, que pour l'approche séparé ou intégré.

Les différentes Méthodes algorithmiques de Montgomery

Les cinq méthodes existantes sur le marché ont été sélectionnées et analysées selon le ***temps d'analyse*** et ***l'espace occupé***.

Le temps d'analyse : est donné par le calcul du nombre total des multiplieurs et additionneurs et les opérations de lécture et d'écriture des mémoires (en terme de mots). Comme le montre la table.II.10

L'espace occupée : est donné par le calcul du nombre totale des mots utilisés pour un espace temporel.

Les Méthodes algorithmiques de Montgomery connues sur le marché sont de nombre 5 :

- Separated Operand Scanning (SOS) méthod
- Coarsely Integrated Operand Scanning (CIOS) method
- Finely Integrated Operand Scanning (FIOS) method
- Finely Integrated Product Scanning (FIPS) method
- Coarsely Integrated Hybrid Scanning (CIHS) method

Les méthodes présentés ci-dessus traitent le même nombre de mot et de taille, ce qui implique la même précision, mais par contre, un différent nombre d'opérations éffectués par chacun.

La méthode CIOS : est la méthode la plus populaire utilisée en Cryptographie RSA qui sert a Crypter des données. Elle permet de nous offrir une meilleure exécution hardware en tenant compte de l'avantage du **parallélisme** entre les blocs multiplieurs, ce qui l'a met au sommet de toutes les autres méthodes existantes sur le marché.

Cette méthode a été choisit pour sa grande rapidité (faster algorithme) qui est due aux critères suivants :

- Au lieu de calculer entiérement le produit de la multiplication ensuite, le produit de la réduction '' *l'approche séparé''* qui ralenttira l'excécusion de l'algorithme. La méthode CIOS vient pallier à ce probléme en éffectuant une alternance rapide entre la multiplication (**a * b)** et la réduction (***n *m)***, ce qui l'a rend plus rapide par rapport à d'autres algorithmes.

- Elle offre le plus petit ***temps d'analyse***, qui donne un nombres d'opérations de multiplications, d'additions et d'écriture, lécture memoires le plus petit par rapport au résultats obtenues dans d'autre méthodes, comme le montre la Table II.10 .En plus elle ne consomme pas beaucoup d'espace mémoire, c'est uniquement : ***s+3*** pour chaque opérande (dans notre cas : **s** = nombre de mots de 16 bits = 32 mots).

Tableau.II.10 Temps et surface obtenus pour chaque méthode de Montgomery [18]

Method	Multiplications	Additions	Reads	Writes	Space
SOS	$2s^2+s$	$4s^2+4s+2$	$6s^2+7s+3$	$6s^2+6s+2$	$2s+2$
CIOS	$\mathbf{2s^2+s}$	$\mathbf{4s^2+4s+2}$	$\mathbf{6s^2+7s+2}$	$\mathbf{2s^2+5s+1}$	$\mathbf{s+3}$
FIOS	$2s^2+s$	$5s^2+3s+2$	$7s^2+5s+2$	$3s^2+4s+1$	$s+3$
FIPS	$2s^2+s$	$6s^2+2s+2$	$9s^2+8s+2$	$5s^2+8s+1$	$s+3$
CIHS	$2s^2+s$	$4s^2+4s+2$	$6.5s^2+6.5s+2$	$3s^2+5s+1$	$s+3$

Comme la méthode CIOS a été le choix de notre algorithme, on s'est focalisé sur son étude afin d'en tirer le calcule du nombre total des opérations de multiplications, ou d'additions, ou Read , Write memory , effectuées par l'algorithme .

Le calcule de ces opérations se fait par le comptage du nombre d'itérations dans une boucle qui sera multiplié par le nombre d'opérations déroulante dans cette dernière , comme le montre la table II.11

Table II.11 Le Calcul du nombre total d'opérations exécutées par la méthode CIOS (Coarsley Integrated Scaning Operand) [18]

	Algorithme	Opération				itérations
		Mult	Add	Read	Write	
Boucle de Multiplication	for i = 0 to s-1 (s=32 mots)	-	-	-	-	-
	1. $C \leftarrow 0$	*0*	*0*	*0*	*0*	*s*
	2. for j = 0 to s-1	-	-	-	-	-
	a. $\{C, S\} \leftarrow tj + aj \times bi + C$	*1*	*2*	*3*	*0*	s^2
	b. $tj \leftarrow S$	*0*	*0*	*0*	*1*	s^2
	3. $\{C, S\} \leftarrow ts + C$	*0*	*1*	*1*	*0*	*s*
	4. $ts \leftarrow S;$	*0*	*0*	*0*	*1*	*s*
	5. $ts+1 \leftarrow C$	*0*	*0*	*0*	*1*	*s*
Boucle de Réduction	6. $m \leftarrow t_0 \times (n_0^{-1}) \bmod 2^w$	*1*	*0*	*2*	*1*	*s*
	7. $\{C, S\}$ $t0 + n0 \times m$	*1*	*1*	*3*	*0*	*s*
	8. for j = 1 to s-1	-	-	-	-	-
	a. $\{C, S\} \leftarrow tj + nj \times m + C$	*1*	*2*	*3*	*0*	*s(s-1)*
	b. $t_{j-1} \leftarrow S$	*0*	*0*	*0*	*1*	*s(s-1)*
	9. $\{C, S\} \leftarrow t_s + C$	*0*	*1*	*1*	*0*	*s*
	10. $t_{s-1} \leftarrow S$	*0*	*0*	*0*	*1*	*s*
	11. $t_s \leftarrow t_{s+1} + C$	*0*	*1*	*1*	*1*	*s*
	Soustraction Final	*0*	*2(s+1)*	*2(s+1)*	*s+1*	*1*
		$2s^2 + s$	$4s^2 + 4s + 2$	$6s^2 + 7s + 2$	$2s^2 + 5s + 1$	

II.3.2.2.5. L'architecture de la méthode CIOS [18], [16]

L'alternance entre les Blocs multiplication et réduction se fait successivement afin d'assurer la rapidité de l'algorithme, une fois que PE_0 génère le premier mot du résultat intermédiaire (c.a.d : le mot le moins significatif), PE_1 commence le calcul de la deuxième itération en tenant compte des résultats du PE_0 .

Tout les PEs transmettent les variables { $a_j , n_j , sum\ S\ et\ C$ } au PEs qui succèdent pour exploiter les données déjà utilisées et simplifier la connexion du Réseau. Afin de maintenir un flot de données continu entre les *PEs*, et compenser les temps perdue des pré-calcul, des FIFOs sont placées pour assurer cette tache.

Une fois que le résultat intermédiaire de l'opération du premier cycle sera obtenue, il sera stocké automatiquement dans une mémoire RAM, pour qu'il soit injecté encore une fois dans le premier PE du cycle suivant , et continue l'opération de la multiplication. Comme le montre la figure .II.8

L'objectif de mettre en place la mémoire RAM est d'économiser de l'espace sur la carte d'implémentation pour faire le calcul d'un nombre limitées de PEs , au lieu de travailler sur un nombre illimité de PEs dans l'architecture systolique, ce qui causera la lenteur de l'algorithme et ces difficultés pour l'implémentation .

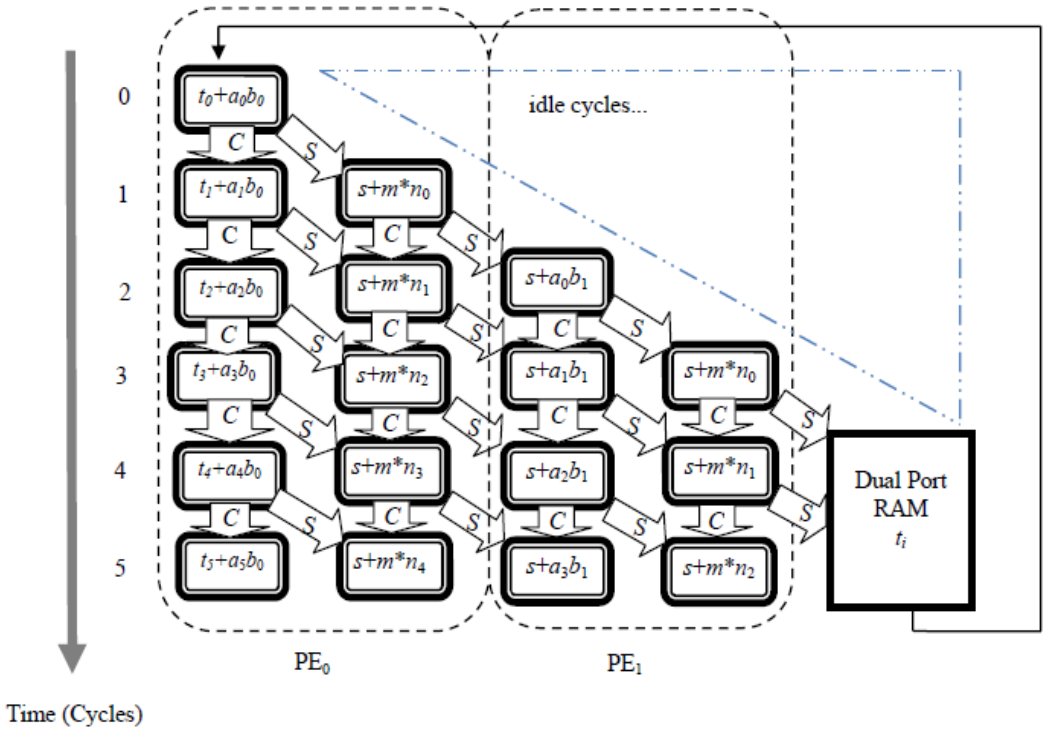


Figure II.8. Le principe de fonctionnement de la méthode CIOS [16]

II. 7. Conclusion

L'exponentiation modulaire qui est une suite de multiplication modulaire, est le coeur des algorithmes de la cryptographie à clé public tels que le RSA. Les méthodes existantes pour éffectuer cette opération se basent sur le principe d'optimiser soit le temps de calcul de la multiplication modulaire, soit le nombre nécessaire de multiplications. Ceci dit, la multiplication modulaire a un intérêt majeur sur le calcul de l'exponentiation modulaire et que toute amélioration dans son calcul est traduite dans l'exécution des protocoles de cryptographie. Dans ce rapport nous avons étudié l'arithmétique modulaire et abordé en particulier la multiplication modulaire de Montgomery. Les algorithmes à réduction sont alors plus intéressants dans les domaines qui exigent un temps de calcul réduit. La comparaison des algorithmes à réduction a savoir Barrett, Quisquarter et Montgomery, nous a permis de conclure que celui de Montgomery est le plus rapide. En effet, il est le mieux adapté au calcul intensif de l'exponentiation modulaire.

De l’étude menée dans ce chapitre, on tire la conclusion suivante : l’utilisation de l’algorithme de Montgomery CIOS associé à la méthode binaire MSB (Montgomery Ladder) est la solution la plus adaptée au calcul de l’exponentiation modulaire du protocole RSA de notre architecture. L’implémentation software de l’algorithme RSA sous C fera l’objet du prochain chapitre.

Chapitre III

Implémentation de RSA en langage C

III.1. Introduction

De nos jours, la protection et la sécurité des données est devenue un souci extrême des compagnies. Les systèmes cryptographique asymétriques définissent une paire de clé pour la transmission des données, l'une est publié a tout le monde, et utilisée pour crypter le texte clair afin d'en déduire le texte chiffré. Et l'autre est secrète, utilisé pour décrypter le texte chiffré afin d'en déduire le texte original. Le message crypté ne sera pas décrypté par n'importe quel personne possédant la clé publique par contre, uniquement par la personne possédant la clé secrète, c'est le principe de fonctionnement de RSA qui est donc l'algorithme à clé publique le plus répandu. Seulement que le problème majeur de RSA réside dans la factorisation d'un très grand nombre en deux nombres premiers, qui est devenue presque impossible au fur et à mesure que la taille de la clé augmente. Plus le nombre à factoriser augmente, la possibilité de factorisation diminue. Pour pallier à ce problème, la mise en œuvre de la bibliothèque GMP a été extrêmement nécessaire afin de gérer les grands nombres. GMP a été classé parmi les librairies les plus indispensables pour surmonter les problèmes de la cryptographie.

Dans ce chapitre, nous allons opté pour l'implémentation software de l'algorithme RSA sous C++ en utilisant la bibliothèque GMP , ainsi que l'analyse des performances de l'algorithme qui ont été enregistrées pendant les opération de cryptage et décryptage en modifiant le nombre de caractères crypté en même temps.

III.2. La bibliothèque GMP [20]

III.2.1. Définition de GMP [20]

GNU MP, également appelée **GMP :** GNU Multiple Précision est une bibliothèque qui permet de gérer les nombres entiers, rationnels et en virgule flottante de trés grande taille et d'éffectuer tout un tas de calculs rapidement. C'est une bibliothèque qui utilise des algorithmes très rapides, son avantage augmente avec la taille des opérandes utilisées au cours des opérations.

C'est une bibliothèque mathématique, elle regorge que des fonctions mathématiques tel que (+, -, *, /). C'est avec ***mpz_class*** et ***mpz_t*** c'est deux variables qui permettent d'utiliser simplement les opérateurs arithmétiques usuels (+, -, *, /).

Il existe plusieurs catégories de fonctions dans la bibliothèque GMP:

1. De haut niveau des fonctions arithmétiques entiers (***mpz***). Il ya environ 140 fonctions arithmétiques et logiques dans cette catégorie.

2. De haut niveau des fonctions arithmétiques rationnels (***mpq***). Cette catégorie se compose de 35 fonctions, mais toutes les fonctions arithmétiques entiers signés peuvent aussi être utilisées, en les appliquant au numérateur et au dénominateur séparément.
3. De haut niveau des fonctions arithmétiques à virgule flottante (***mpf***). Il s'agit de la catégorie de la fonction GMP à utiliser si le type C « double » ne donne pas suffisamment de précision pour une application. Il ya environ 65 fonctions dans cette catégorie.

Les fonctions permettent d'accélérer la multiplication modulaire dans GMP sont: la Fonction ***Modulo*** , ***Puissance*** et la ***division,*** nous commençant a étudier cette fonction de GMP et ses différents prototypes dans ce qui suit

III.2.2.Les opérateurs usuels, Les divisions [20], [18]

GMP nous donne tout un panel de fonctions sur la division, dont les prototypes primordiaux des trois fonctions sont définis ci-dessous :

- ***void mpz_fdiv_q (mpz_t q, mpz_t n, mpz_t d)*** ……*(1)*
- ***void mpz_fdiv_r (mpz_t r, mpz_t n, mpz_t d)*** ……*(2)*
- ***void mpz_fdiv_qr (mpz_t q, mpz_t r, mpz_t n, mpz_t d)*** ……*(3)*

Ces trois fonctions ont chacune leur propre rôle :

La première (1) permet d'obtenir le quotient de la division euclidienne de "n" par "d". La deuxième (2) permet d'obtenir le reste de la division euclidienne de "n" par "d". Et la troisième (3) permet d'obtenir le quotient et le reste de la division euclidienne de "n" par "d".

Maintenant que l'on sait à quoi servent ces fonctions, le prototype utilise des "***mpz_t***" et non des "***mpz_class***" "*mpz_t*" découle du fait que la bibliothèque était à l'origine en C et non pas C++. Il s'agit d'un des trois types de variables crées à l'origine de la GMP. . Pour s'en servir, il va falloir que l'on transforme notre objet "***mpz_class***" en une variable du type "***mpz_t***". C'est pour cela que la méthode ***get_mpz_t()*** a vu le jour.

III.2.3. Les operateurs de la multiplication modulaire [36]

La bibliothèque GMP est, certes, une bibliothèque qui sait gérer les très grands nombres, mais avant tout, elle est une bibliothèque mathématique. C'est pour cela qu'elle regorge de fonctions mathématiques. Certaines sont les mêmes que celles disponibles dans la bibliothèque mathématique standard. (Ceil(), floor(), sqrt(), abs(), etc.), Et d'autres servent au calcul de la multiplication / exponentiation modulaire tel que Les fonctions ***Modulo*** , ***Puissance*** qui seront évoquée par la suite.

III.2.3.1. La fonction Modulo

C'est sur cette fonction que repose tout le système RSA. Elle éffectue le calcul suivant : r = M mod(n), son prototype est ci-dessous :

void ***mpz_mod*** *(mpz_t* ***r****, mpz_t* ***M****, mpz_t* ***n*** *)* ……*(4)*

Sachant que les fonctions ***mpz_mod*** et ***mpz_fdiv_r*** (définie dans (2)) font le même travail ; cela dit que nous avons choisi le modulo pour utiliser la fonction prévue à cet effet.

III.2.3.2. La fonction Puissance

La fonction "puissance" est aussi une des fonctions à la base du système RSA. Cette fonction éffectue le calcul suivant : $r = M^{exp}$, L'exposant doit être un entier de type "unsigned long int". son prototype est ci-dessous :

void mpz_pow_ui *(mpz_t* ***r****, mpz_t* ***M****, unsigned long int* ***exp*** *)* ……..*(5)*

III.2.3.3. La fonction idéale pour RSA [36]

Nous pourrions très bien faire notre programme RSA en utilisant seulement les deux dernières fonctions que l'on vient de voir (4) et (5). Mais le problème est que ; calculer ***une puissance*** puis ***un modulo*** (l'un après l'autre) demande de nombreux calculs, et donc du temps. L'astuce consiste en faite à calculer ***le modulo et la puissance*** en même temps pour gagner beaucoup plus de temps.

Cette fonction éffectue le calcul de l'exponentiation modulaire suivant : $C = M^e \bmod (n)$, *c'est l'équation de cryptage de RSA,* son prototype est ci-dessous :

void mpz_powm *(* *mpz_t* ***C****, mpz_t* ***M****, mpz_t* ***e****, mpz_t* ***n*** *)*

La même fonction éffectue le calcul de l'exponentiation modulaire : $M = C^d \bmod (n)$, pour le décryptage de RSA. Son prototype est ci-dessous :

void mpz_powm *(mpz_t* ***M****, mpz_t* ***C****, mpz_t* ***d****, mpz_t* ***n*** *)*

Donc c'est à travers ces deux fonctions que l'opération du modulo et puissance seront accélérer lors du *Cryptage et Décryptage*, qui veut dire accélérer ***l'exponentiation Modulaire de l'algorithme RSA.***

III.3. L'implémentation de l'algorithme RSA en langage C basé sur la librairie GMP [Annexe A/ Algorithme RSA_1]

L'algorithme RSA comporte dans sa globalité 3 principales Fonctions de base :

1- La fonction de génération de clefs
2- La fonction de cryptage
3- La fonction de décryptage

La première fonction a pour tache la création des clés publiques et privés suivants les étapes ci-dessous :

* Générer deux grands nombres primaires p et q aléatoirement
* Calcul de n = p * q et x = (p-1) (q-1) ; p et q sont de tailles de 512 bits , ce qui fait que n = p * q est de taille de 1024 bits.
* Sélectionner l'entier e tel que : e (1<e<x) et le : gcd (e,x) = 1 ; la taille minimale de la clé publique est sur 17 bit équivalent a [65537 (décimale)= 10001(hexadécimale)]
* Calculer l'entier unique d: tel que e*d = 1(mod x)
* Déduire la Clé Publique : (e, n), et la clé privé: (d, n)

La deuxième fonction a pour tache d'accomplir l'opération de cryptage suivant l'équation :

* $C = M^e \bmod (n)$

La troisième fonction a pour tache d'accomplir l'opération de décryptage suivant l'équation :

* $M = C^e \bmod (n)$

Pour ce qui suit, nous trouverons la description des fonctions principales du programme RSA en langage C suivant l'ordre chronologique .

III.3.1. initializeGMP()

Cette fonction permet d'initialiser les entiers GMP (d, e, n, M, C) en utilisant la commande : *mpz_init.*

III.3.2. RSA_checkKeys()

Cette fonction permet de vérifier *l'existence* des deux clés publique et privée dans les fichiers $HOME/.rsapublic et $HOME/.rsaprivate

- ✓ la structure de la clé publique *:{ e_str,n_str }*

mpz_set_str (e, e_str, 10);
mpz_set_str(n, n_str, 10);

- ✓ la taille de la clé publique *:{ e_str, n_str }* des deux entiers GMP

e_str[100];
n_str[1000];

- ✓ la structure de la clé privée : { d_str,n_str }

mpz_set_str(d,d_str,10);
mpz_set_str(n,n_str,10);

- ✓ la taille de la clef privée : { d_str,n_str } ces deux entier GMP

d_str[1000];
n_str[1000];

On Remarque bien que la clé privée est de taille plus grande que celle de la clé publique. Pour des raisons de sécurité, elle doit être compliqué et de taille très grande.

III.3.3. RSA_generateKeys()

C'est une fonction de base pour le programme RSA, qui permet de créer les deux clés publique et privée, elle permet de modifier la constante '' Bitstrengh'' selon la taille du modulo n de RSA. C'est la première fonction qui permet d'accélérer la multiplication modulaire à travers les fonctions cités ci-dessous :

- ✓ enregistre le temps de début de génération de clés : c'est une fonction de la bibliothèque 'time.h'

if(gettimeofday(&tv1,&tz)!=0) => tv1,

- ✓ Le calcul de n = p * q par la fonction de multiplication modulaire de GMP

mpz_mul(n,p,q);

- ✓ Le calcul de x=(p-1)*(q-1) par la fonction de multiplication modulaire de GMP

mpz_mul(x ,p_minus_1, q_minus_1);

- ✓ Le calcul de e, par la fonction :

mpz_gcd_ui(gcd,x,e_int);

- ✓ Le calcul de d, par la fonction :

mpz_invert(d,e,x)==0 ;

- ✓ Enregistre le temps de fin de génération de clefs : c'est une fonction de la bibliothèque 'time.h'

if(gettimeofday(&tv2,&tz)!=0)=> tv2,

- ✓ Calcul du temps totale mis lors de la génération de cléf publique et privée :

timediff(&tv2,&tv1,&tvdiff);=> tv2-tv1 =tvdiff,

- ✓ Ecrire les clefs public et privé dans: $HOME/.rsapublic et $HOME/.rsaprivate

III.3.4. RSA_encrypt()

Cette fonction permet l'opération de Cryptage en utilisant la bibliothèque GMP ci-dessous qui contribue a l'accélération de l'exponentiation modulaire.

- ✓ Calcul du temps avant le cryptage :

if(gettimeofday(&tv1,&tz)!=0)

- ✓ Operation de Cryptage :

encrypt(stread,fout);

- ✓ Calcul du temps après le Cryptage :

if(gettimeofday(&tv2,&tz)!=0)

- ✓ Calcul du temps total du Cryptage :

timediff(&tv2,&tv1,&tvdiff);

- ✓ Fonction de Cryptage : $C = M^e \bmod (n)$

mpz_powm (C, M, e, n) ;

III.3.5. RSA_decrypt();

Cette fonction permet l'opération de décryptage en utilisant la bibliothéque GMP qui contribue à l'accélération de l'exponentiation modulaire.

- ✓ Calcul du temps avant le décryptage :

if(gettimeofday(&tv1,&tz)!=0)

- ✓ Fonction de décryptage : $M = C^d \bmod (n)$

mpz_powm (M, C, d, n) ;

- ✓ Calcul de temps après le décryptage :

if(gettimeofday(&tv2,&tz)!=0)

- ✓ Calcul de temps total du décryptage :

timediff(&tv2,&tv1,&tvdiff);

III.3.6. ClearGMP();

Nettoyer tout les entiers GMP (d, e, n, M, C) avec la fonction : *mpz_clear*.

III.4. L'implémentation de l'algorithme RSA sous Linux Ubunto [Annexe A]

L'algorithme RSA implémenté en langage C a été exécuté sous une machine Intel P4 Processor 1.73 GHz, de generation DELL D610, sous l'environnement Linux Ubunto, pour faire des tests de compilation.

Avant d'installer la bibliothèque GMP, il va falloir d'abord installer les packages du compilateur **GCC** pour pouvoir procéder a l'étape de l'installation du GMP.

III.4.1. Installation de la bibliothèque GMP

L'installation de la bibliothèque GMP nécessite les étapes suivantes :

- Télécharger le fichier ''gmp-5.0.2.zip''
- Lancer un terminal sous Linux Ubunto
- Taper **: tar -zxvf gmp-5.0.1.tar.bz2**
- Taper **: . /configure**
- Taper **: make**
- Taper **: sudo make install**

  Après l'installation de GMP, on procède par la suite a l'exécution de l'algorithme RSA

III.4.2. Exécution de l'algorithme RSA [Annexe A/ Algorithme RSA_2]

Pour l'exécution de l'algorithme RSA sous C, on doit mettre à notre disposition les fichiers ci-dessous, comme le montre la figure ci-dessous :

Pour l'opération de cryptage : voir figure (III.1)

- un fichier_a_ crypter qui contient les données à crypter sous format héxadecimal
- un fichier_crypté, qui contient le texte crypté : les données qui ont été cryptées à partir du fichier : fichier_a_crypter.

- Le fichier de cle public : contient la clé publique de cryptage (e,n) ainsi que le nombre de caractère à crypter en même temps.

Pour l'opération de décryptage : voir figure (III.1)

- un fichier_a_ décrypter qui contient le texte Crypté : contient les données déjà cryptées (sous format héxadecimal), a partir du fichier_crypté.
- un fichier_décrypté , qui contient le texte clair original, c.a.d ; les données décryptées à partir du fichier : fichier_a_décrypter.
- Le fichier de clé privée : contient la clé privé de décryptage (d,n) ainsi que le nombre de caractères à décrypter en même temps.

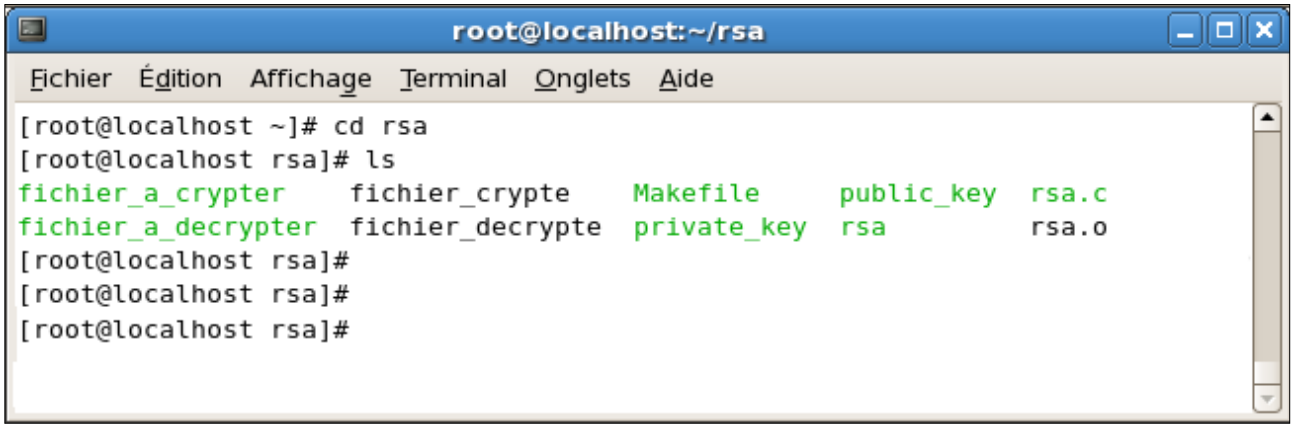


Figure III.1. Les fichiers de cryptage et décryptage et clefs de l'algorithme rsa

III.4.2.1. La compilation du code rsa.c [Annex A/ Algorithme_2]

Lors de la compilation de rsa, un menu s'affiche a l'écran, donnant le choix a l'utilisateur de soit :

1) Sélectionner 1 pour éffectuer l'opération de cryptage
2) Sélectionner 2 pour éffectuer l'opération de décryptage
3) Sélectionner 3 pour quitter

- Compiler le code rsa.c, on tapant la commande suivante, comme le montre la figure III.2

[root@localhost~]# gcc –g –o rsa -lgmp rsa.c

- Pour exécuter le code rsa, on tape la commande suivante, comme le montre la figure III.2

[root@localhost~]# . /rsa

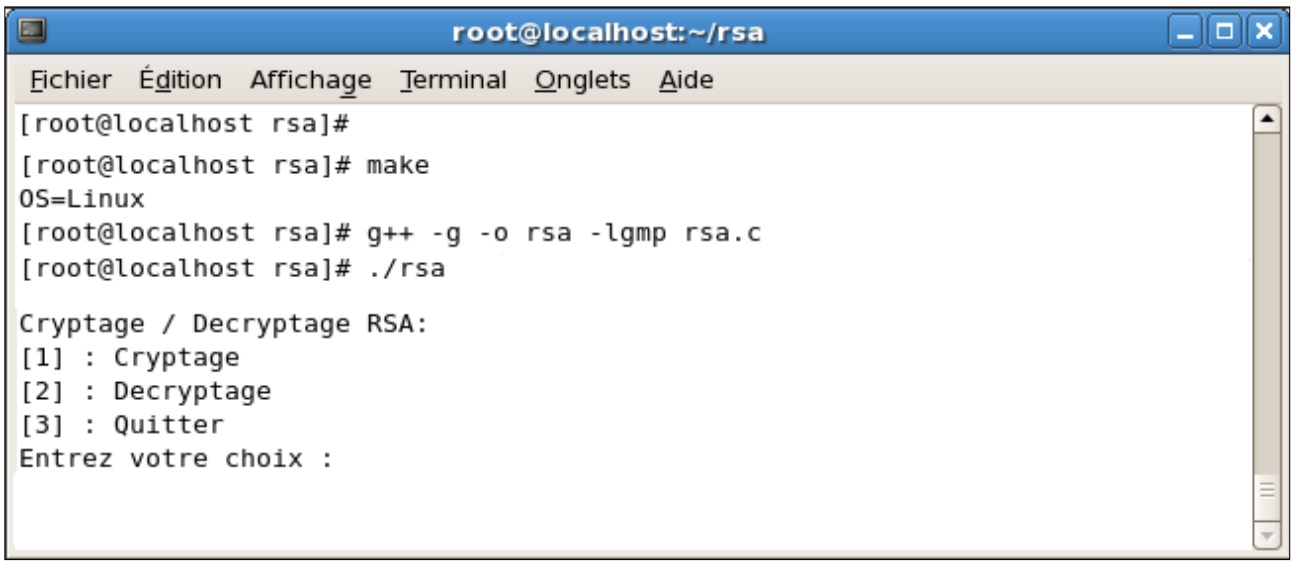


Figure III.2. Compilation du code rsa.c

III.4.2.1.1. Opération de cryptage :

Dans le menu de l'algorithme RSA, pour crypter le fichier_a_crypter**,** il faut selectionner **[1].** Ce dernier contient le texte clair sous format hexadecimal. Le fichier_crypté contiendra le texte crypté sous format hexadecimal y compris le temps enregistré lors de l'opération de cryptage. Le nombre de caractéres à crypter en même temps sera indiqué dans le fichier de clé public ou les deux valeurs de {e,n} seront affichées sous format hexadécimal aussi, comme le montre les figures ci-dessous.

Pour ce qui s'en suit, les temps de cryptage seront enregistrés pour toutes les tailles de clés allant de { 512, 1024, 2048 } bits, qui correspondent respectivement au nombre de caractéres à crypter en meme temps allant de { 64, 128, 256 }caractéres, sachant qu'un caractére est codé sur 8 bits.

Tous les vecteurs de test ont été choisis d'une manière aléatoire.

- **Clé=512 bits**

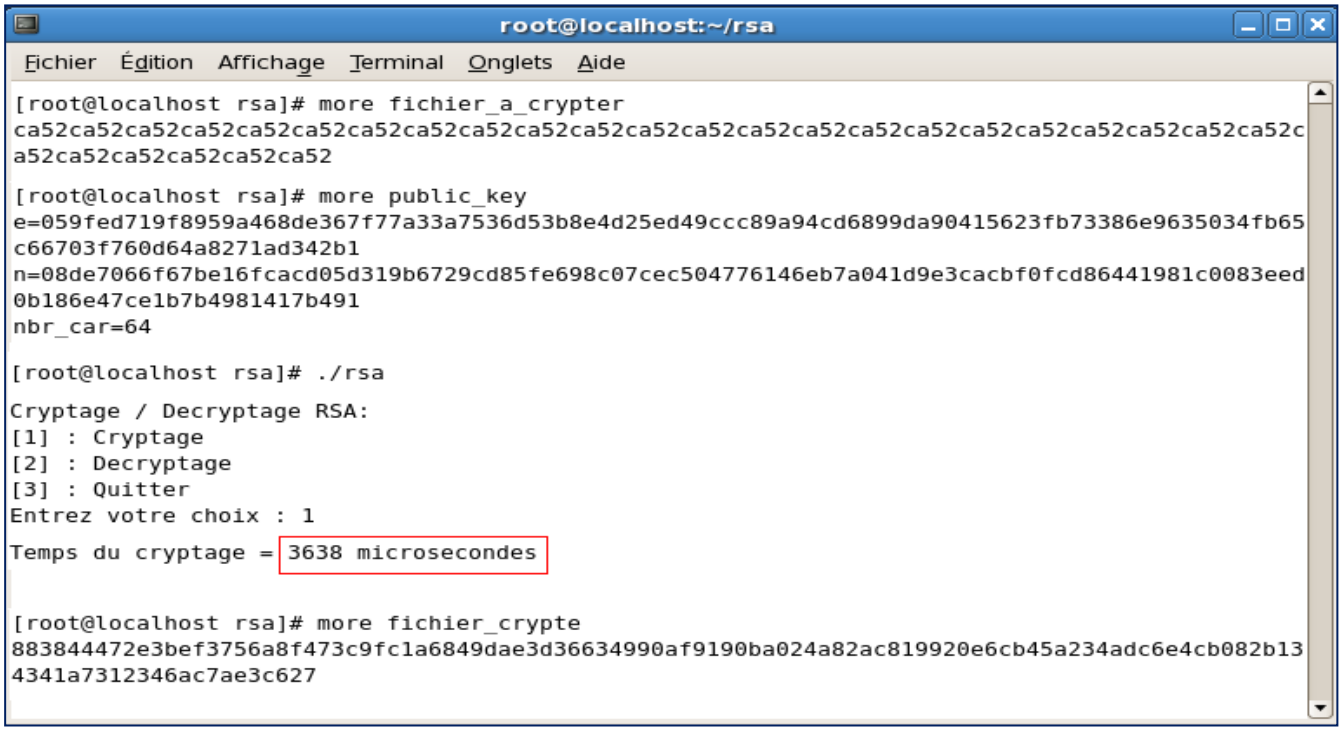


Figure III.3. L'opération de Cryptage rsa pour une clé de 512 bits

- **Clé=1024 bits**

Figure III.4. L'opération de Cryptage rsa pour une clé de 1024 bits

- **Clé=2048 bits**

```
Fichier  Édition  Affichage  Terminal  Onglets  Aide
[root@localhost rsa]# more fichier_a_crypter
ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52
ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52
ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52
ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52
ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52ca52
[root@localhost rsa]# more public_key
e=67cd484c9a0d8f98c21b65ff22839c6df0a6061dbceda7038894f21c6b0f8b35de0e827830cbe7ba6a56ad77c6eb517970790a
a0f4fe45e0a9b2f419da8798d6308474e4fc596cc1c677dca991d07c30a0a2c5085e217143fc0d073df0fa6d149e4e63f0175879
1c4b981c3d3db01bdffa253ba3c02c9805f61009d887db031967cd484c9a0d8f98c21b65ff22839c6df0a6061dbceda7038894f2
1c6b0f8b3567cd484c9a0d8f98c21b65ff22839c6df0a6061dbceda7038894f21c6b0f67cd484c9a0d8f98c21b65ff22839c6df0
a6061dbceda7038894f21c6b0f8b3567cd484c9a0d8f98c21b65ff22839c6df0a6061dbceda7038894f21c6b0f8b35
n=a9e167983f39d55ff2a093415ea6798985c8355d9a915bfb1d01da197026170fbda522d035856d7a986614415ccfb7b7083b09
c991b81969376df9651e7bd9a93324a37f3bbbaf460186363432cb07035952fc858b3104b8cc18081448e64f1cfb5d60c4e05c1f
53d37f53d86901f105f87a70d1be83c65f38cf1c2caa6aa7eb67cd484c9a0d8f98c21b65ff22839c6df0a6061dbceda7038894f2
1c6b0f8b3567cd484c9a0d8f98c21b65ff22839c6df0a6061dbceda7038894f21c6b0f67cd484c9a0d8f98c21b65ff22839c6df0
a6061dbceda7038894f21c6b0f8b3567cd484c9a0d8f98c21b65ff22839c6df0a6061dbceda7038894f21c6b0f8b35
nbr_car=256
[root@localhost rsa]# ./rsa
Cryptage / Decryptage RSA:
[1] : Cryptage
[2] : Decryptage
[3] : Quitter
Entrez votre choix : 1
Temps du cryptage = 154612 microsecondes
[root@localhost rsa]# more fichier_crypte
6e0a752a70ce520447a9eaa58ab2d7bae01b8e706a899f93580afae3ca39dcfb8a6326f5022118a8e4302b51ac00a68d4d311cc2
328b472959d6c8a5e3183c6dcfe4d8b6200331cc88834cf1757f40c05a2936a9d513f8581d20b2383f53697c00242be0299be5cb
9992a12e1979466b87454a2710ca0a35c46dfe09993c68b4ce5f3340219fa27c5c92b7688d24f9004571694a57d70f479dc7d824
7a9b96bc6f24e7c3bb744bb84d3cb32aac5f9e91267852b4724502e4f7d6b99e814ebd65e9af113e763c520c7104d7a460edd873
99e9e780388010eaf2a0cf17e57e7172f0fd5742af8ffeda1d51b460f13fec9a4a6df1ffa6f1e65302811046b6b8
```

Figure. III.5. L'opération de Cryptage rsa pour une clé de 2048 bits

Les temps de cryptage mentionnés dans le tableau III.1, ont été enregistrés précédemment pendant l'opération de cryptage sur le fichier_a_crypter pour des tailles de clefs variant de [512, 1024, 2048] bits avec respectivement des nombres de caractères cryptés en même temps variant de [64, 128, 256]

Tableau III.1. Temps de Cryptage enregistrés

Bit Strength	*Nombre de caractère*	*La clef de Génération*	*Temps de Cryptage (micro-seconde)*
512	***64***	***0.057984***	***3638***
1024	***128***	***0.465994***	***18877***
2048	***256***	***4.052181***	***154612***

En variant la constante ''Bitstrengh'' de 512 vers 2048 bits, nous constatons que les temps enregistrés pour le cryptage augmentent au fur et à mesure que cette constante augmente.

Parmi les résultats obtenus du tableau, nous concluons que le chiffrement ***RSA_512*** bits est le plus rapide vue son temps de cryptage, mais le moins sécurisé, car il ne représente pas une protection optimale contre les attaques sur RSA. Par contre, le chiffrement ***RSA_2048*** bits qui est le plus lent vue son temps de cryptage, il offre une sécurité optimale contre les attaques sur RSA, c'est pour cette raison qu'il est utilisé pour les situations critiques.

Se basant sur cette comparaison, le choix de la taille de la clé s'est porté sur le chiffrement ***RSA_1024*** bits qui offre le meilleur compromis entre vitesse et sécurité surtout pour une application comme la notre, où on essaye de réduire au minimum le temps de chiffrement/déchiffrement des données.

III.5. Conclusion

Le chiffrement RSA, est fondé sur l'hypothèse qu'il est très difficile de factoriser un nombre très grand en deux nombres premiers, ceci le met en tète des chiffrements à cléf publique.

L'implémentation software de l'algorithme RSA sous C au sein d'une machine Linux, présenté dans ce chapitre, a permis d'enregistrer des valeurs de temps de cryptage allant au alontour des micro-secondes, (les temps d'éxcécusion de la machine).

Cependant, étant donné sa lenteur, et ces temps d'exécutions trop élevé de la machine due au calcul d'exponentiation modulaire intensif, il est plus judicieux d'implémenter cet algorithme en hardware pour compenser cet inconvénient. Ceci est l'objet des prochains chapitres.

Chapitre IV

Etude de l'IP-Core rsa_512

IV.1. Introduction

Dans ce chapitre nous allons présenter la description matérielle de notre IP-Core RSA, que nous allons étudié pour le calcul de l'exponentiation modulaire binaire basée sur la multiplication de Montgomery. ainsi que la description des blocs de contrôle, et mémoires (Fifos) permettant le fonctionnement correcte de cette architecture.

Notre architecture est de type serial (les paquets de données sont introduit à l'entrée de l'IP-Core en série), qui est basée sur une structure parallèle permettant l'exécution de deux multiplications modulaires de
Montgomery en parallèle, ce qui a réduit énormément les temps de chiffrement et déchiffrement des messages.

Afin de pouvoir réaliser notre projet nous nous sommes interessé au circuit d'Open-Core'' RSA_512 bits''. Nous consacrons ce chapitre à son étude et la description de son fonctionnement

IV.2. Définition d'un IP core [32]

L'IP-Core est un bloc logique utilisé dans la conception d'application sur FPGA. Il a pour tâche d'accélérer l'exécution d'une opération qui consomme beaucoup de temps en software.

Dans notre travail, le SoC qu'on va réaliser, aura pour tâche de calculer et accélérer l'opération de la multiplication modulaire dans le but d'optimiser l'exponentiation modulaire.

L' IP-Core réalise une fonction spécifique, dont la complexité est variable. On retrouve des mémoires, des entrées/sorties, des processeurs sous forme d'IP core.

IV.2.1. Le langage utilisé pour le développement de l'IP-Core [25]

VHDL est le langage utilisé pour le développement de L'IP-Core. C'est est un langage de description matériel destiné à représenter le comportement ainsi que l'architecture d'un système électronique numérique. L'intérêt d'une telle description réside dans son caractère exécutable : une spécification décrite en VHDL peut être vérifiée par simulation, avant que la conception détaillée ne soit terminée. En outre, les outils de conception assistée par ordinateur permettant de passer directement d'une description fonctionnelle en VHDL à un schéma en porte logique ont révolutionné les méthodes de conception des circuits numériques, ASIC ou FPGA.

IV.2.2. Présentation de l'IP-Core RSA_512 [Annexe B]

Cet IP-Core permet de calculer l'exponentiation modulaire classique dont l'équation mathématique est de : $x^y \bmod m$. La figure V.1 montre l'interface de l'IP-Core ainsi que tout ses signaux d'entrée et de sortie.

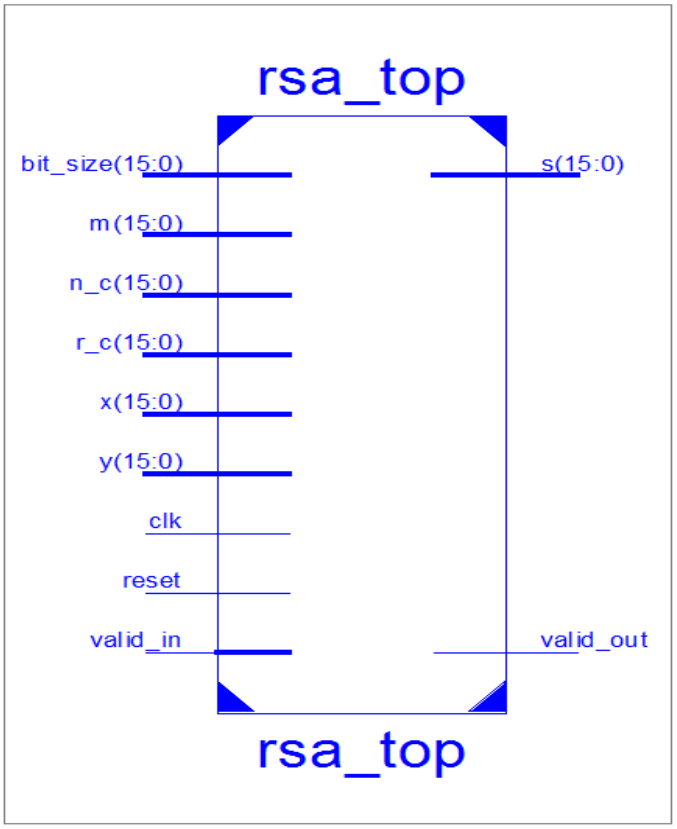


Figure VI.1 : Interface de l'IP core rsa_512 [36]

IV.2.2.1. Description des signaux d'entrée

bit_size : C'est une valeur constante qui spécifie la longueur de *y*, qui est soit la clé privée (est sur 512 bits) soit la clé publique (est d'une taille minimale est de 17 bits). Elle peut être calculée en utilisant log(*y*) avec *y* la clé utilisée pour crypter.

x: Cette variable représente le texte en entrée qui sera crypté. Le texte sera introduit de la forme suivante $x=a_{31}a_{30}.....a_0$ avec a_i un mot de 16 bits. Le mot le moins signifiant a_0 sera introduit le premier (premier cycle d'horloge).

y: Cette clé une fois introduite, va servir au cryptage du texte x. Elle est sur 17 bit lors du cryptage avec la clé publique et sur 512 bit lors du cryptage avec la clé privée

m : C'est le module m. Il est sur 512 bits

n_c : C'est une constante de 32 bits, utilisée pour accélérer l'exponentiation, elle dépend du module m seulement. Elle peut être calculée comme suit $n_c=(-m)^{-1} \bmod r$, avec r la constante de Montgomery. (r = 2^{16*32}, tel que 32 est le nombre de mots , et 16 est la taille du mot)

r_c : C'est une constante de 512 bits, utilisée pour accélérer l'exponentiation, elle est donné par $r_c = r^2 \bmod m$.

valid_in : Ce signal doit être activé à l'état haut lors de la prise des données.

IV.2.2.2. Description des signaux de sortie

S: Ceci est le résultat de l'exponentiation modulaire.

Valid_out : Comme son nom l'indique, elle nous informe quand la sortie *s* est prête.

IV.3. Methodologie d'étude et d'analyse de l'IP-Core RSA_512

L'architecture décrite dans notre IP-Core, est ***une architecture Serial*** (série), car les données d'entrées sont introduites en série (et non pas en parallèle) ; mot par mot constituant un flot de paquets en série de 32 mots de 16 bits .

L'IP-Core RSA_512 est composé de plusieurs modules consituant le module principal rsa_top. Ces derniers permettent d'accélérer l'exponontiation modulaire de l'algorithme RSA en s'appyant sur l'avantage de l'algorithme de Montgomery

L'étude de l'architecture globale a été élaboré d'une manière croissante. Nous commencons notre étude par le plus petit bloc élémentaire dans la multiplication de Montgomery ; qui est le bloc ''pe '', jusqu'au bloc principale ''rsa_top'' (le plus grand bloc) qui englobe la totalité des blocs existants.

IV.4. Principe de fonctionnement des modules de L'IP-Core RSA_top [Annexe B]

L'architecture globale de l'IP-Core RSA_top (module principale) qui calcule l'exponentiation modulaire est composée des Blocs ci-dessous :

- Deux blocs de multiplications de Montgomery : ***mon_1*** et ***mon_2*** servent à éffectuer le calcul de l'exponontiation binaire qui permet d'accomplir les deux opérations d'élévation au carré et de multiplication durant les 514 itérations. Une Fifo : ***fifo_res_out*** utilisée pour stocker les multiplications intermédiaires et une mémoire principal RAM (qui englobe deux mémoire : ***exp-mem_b***, ***n_mod-mem_b***) utilisée pour le stockage de l'exposant y.

- Chaque Bloc de Montgomery_mult : ***mon_1*** et ***mon_2 ;*** contient 7 blocs de Montgomery_step (PEs) pour le calcul du PE, et deux fifos :(Fifo_512_bram, Fifo_256_feedback) utilisée pour stocker les résultats intermédiaires.

- Chaque Bloc de ***Montgomery_step*** , contient le Bloc ***PE*** qui permet le calcule élémentaire de la multiplication et réduction de Montgomery. Plus le bloc ***''m_calc''*** qui permet le calcul de la constante de Montgomery ***n_c***. comme le montre la figure IV.1

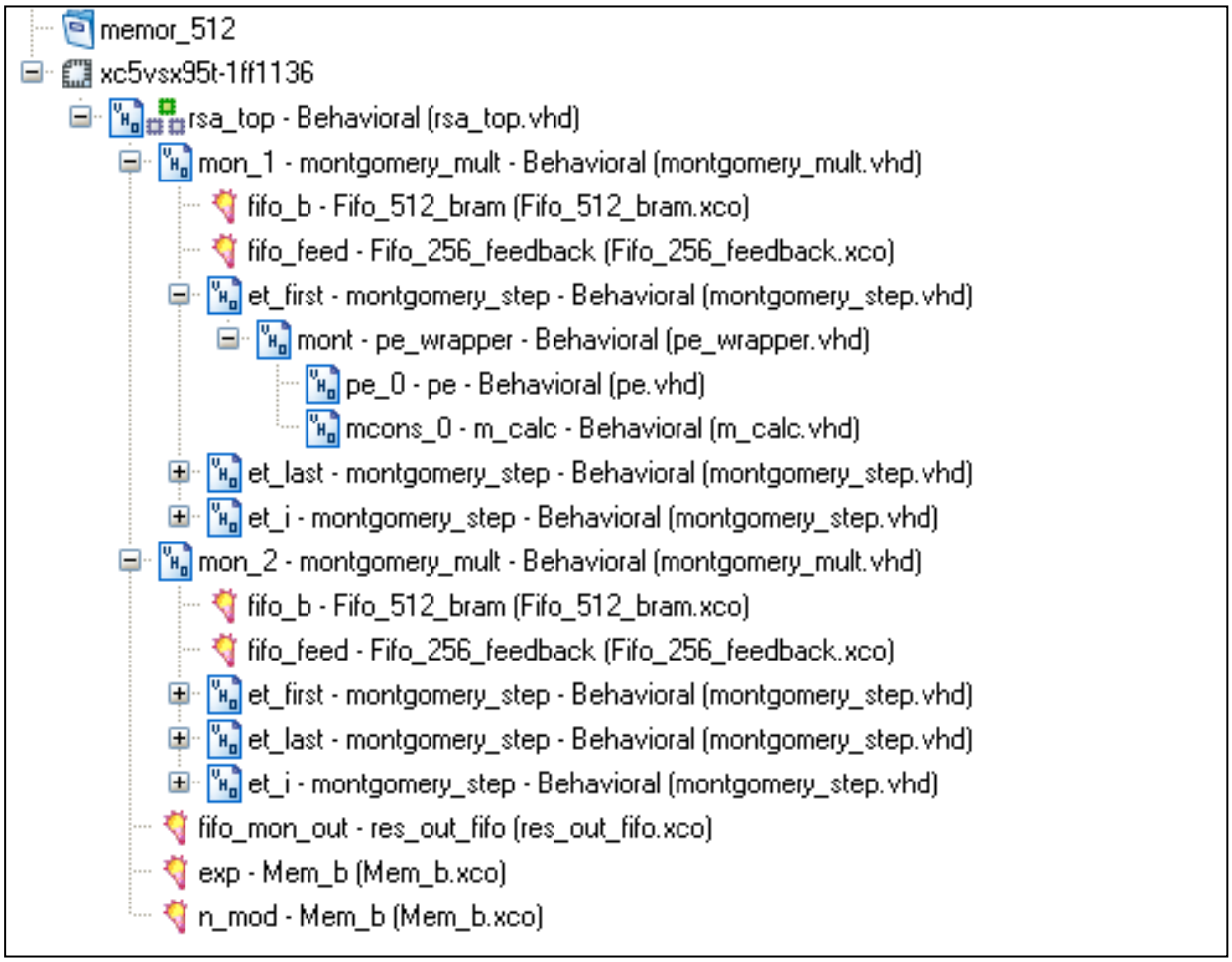


Figure VI.2. Les modules de l'IP-Core rsa_512

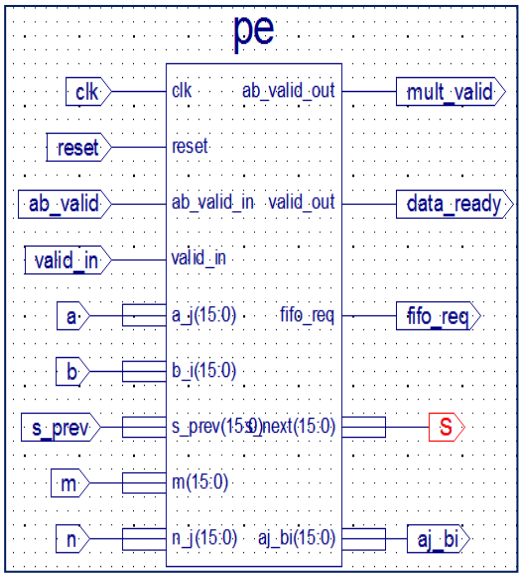
Dans ce qui suit, nous allons aborder l'explication détaillé des modules constituants l'IP-Core rsa_top y compris la machine d'état qui corresponde à la desccription de chaque circuit séquentiel contenant chaque module.

Chaque module sera representé par son schéma descriptif identifiant les entrées et sorties du bloc (y compris les signaux de contrôle), qui définissent eux-meme la tache (sous forme d'équation mathématique) assurée par le bloc au sein de l'architecture globale à travers sa description en vhdl

On commencera notre étude par le plus petit bloc élémentaire dans la multiplication de Montgomery : le bloc ''pe'', et finira par le bloc principale rsa_top (le plus grand bloc) qui englobe la totalité des autres modules et assure l'éxcécusion de l'exponontiation modulaire RSA.

IV.4.1 Module ''pe'' [Annexe B]

PE: appelé " **Processing Element**". C'est l'élément basique du modules rsa_top. Il est responsable du calcul de la multiplication et réduction de Montgomery dans chaque itération de l'algorithme. (voir Algorithme II.1).



prod_aj_bi = a_i * b_j

prod_nj_m = n_j *m

next_sum_1 <= prod_ai_bj + sum_1 (31 downto 16) + s_prev

next_sum_2 <= prod_nj_m + sum_2 (31 downto 16) + sum_1(15 downto 0)

S_next =S <= sum_2 (15 downto 0)

next_sum_2 = (a_j * b_i) + (n_j *m) + S_prev + sum_1 (31 downto 16) + sum_2 (31 downto 16)

s_prev: resultat du PE precedent a l'iteration j-1

sum_1 (31 downto 16): la partie haute du produit (a_i-1 * b_j-1) de l'itération j-1

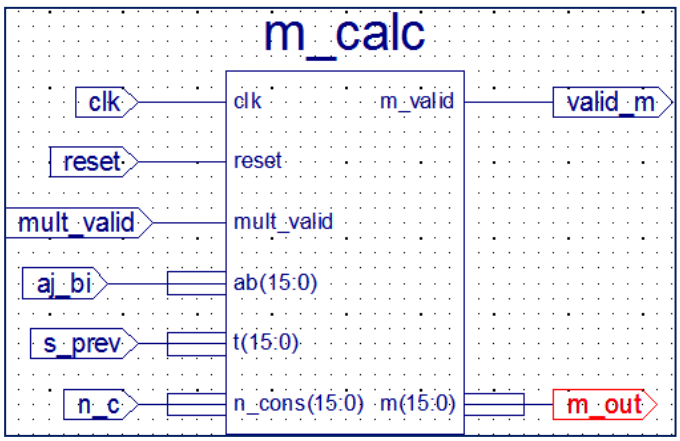
sum_2 (31 downto 16): la partie haute du produit (n_j-1 *m i-1) de l'itération j-1

sum_1 (31 downto 16)+ sum_2 (31 downto 16)= $C(i)_j$ .....de l'equation (*)

Figure VI.3. Schéma et description du bloc PE [36]

IV.4.2 Le module ''m_calc'' [Annexe B]

Ce module permet de calculer la constant de multiplication ''m'' qui est utilisé pour compléter le fonctionnement du module ''pe''. $m = [tj + (aj \times bi)] (n_0^{-1}) \bmod 2^w$



sum_res <= ab+t = ai_bj + s_prev

n_cons=n_c= $-m_0^{-1} \bmod r$ (r =constant de Montgomery , r= $2^{16*(32+1)}$)

m_0= est le mot le moins significatif du modulo n

mult <= (ai_bj + s_prev) * n_cons

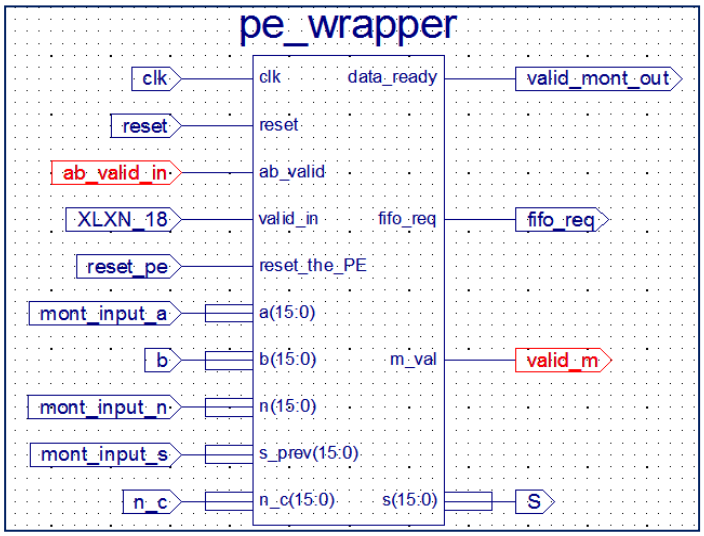
mult <= sum_res * n_cons

m <= mult (15 downto 0)

Figure VI.4. Schéma et description du bloc ''m_calc''[36]

IV.4.3. Le module ''pe_wrapper'' [Annexe B]

Pe_Wrapper permet de connecter les deux Bloc : ''Pe'' et ''m_calc'' comme le montre la figure.VI.6, son rôle est de finaliser le calcul du ''pe'' actuel afin de se préparer au calcul du ''pe'' Prochain. Il sauvegarde les résultats intermediares pour réduire le ''Critical Path''.



Le Bloc PE_Wrapper connecte le Blocs Pe et m_calc

valid_m =m_val = '1' => next_m <= m_out;

il permet de valider le calcul de m afin de pouvoir l'introdure dans le pe pour calculer le produit : **prod_nj_m = n_j *m**

ab_valid_in =ab_valid : indique que les les valeurs ont ete introduite dans le Pe correctement

reset_pe : sera mis a 1 pour accepter le PE suivant

Figure VI.5. Schéma et description du bloc ''pe_wrapper'' [36]

La figure VI.6 décrit le rôle du bloc ***PE*** et ***m_calc*** qui contribuent tous les deux à accomplir la multiplication et la réduction de Montgomery.

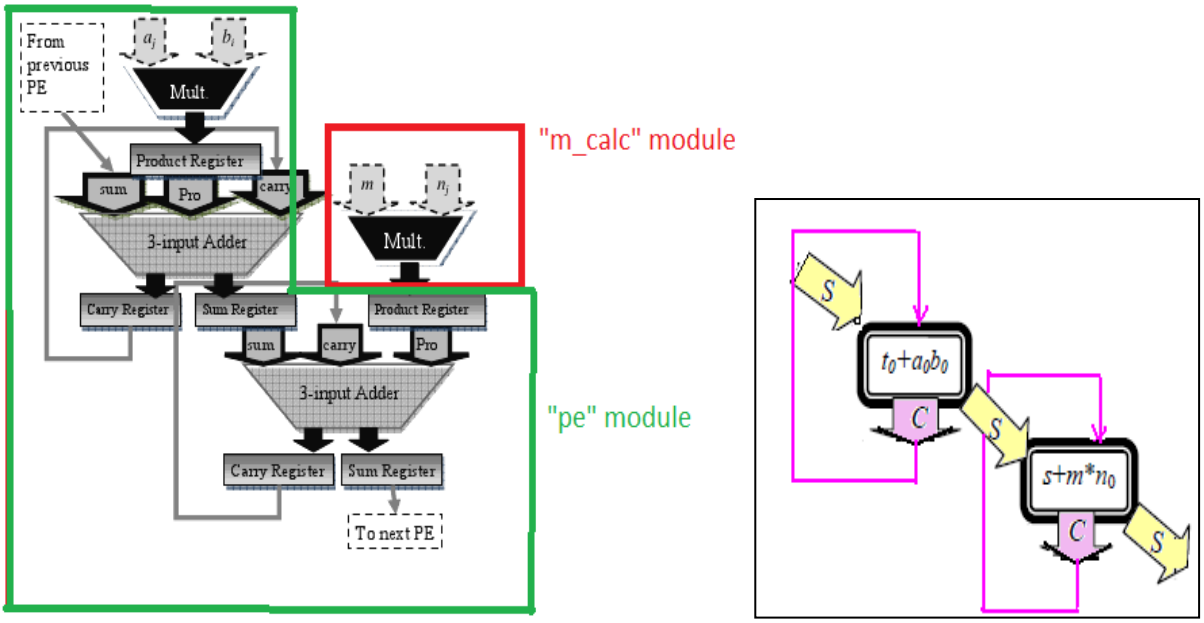


Figure VI.6. Schéma du bloc ''pe_wrapper'' dans l'architecture Systolique [36]

IV.4.4. Le module ''Montgomery_step'' [Annexe B]

Ce module sert à préparer les blocs de multiplication et d'addition de PE à la multiplication globale de Montgomery CIOS. Autrement dit sert à préparer le bloc ***Montgomery_mult*** à éxcércer la multiplication.

Il permet de charger les résultats intermédiaires issues des blocs de PE-Wrapper dans des registres de tailles de 16 bits * 3, afin de préparer les 32 mots de : a_i , n_i , et s_prev_i au calcul de l'architecture systolique comme le montre la Figure VI.7

Le compteur fait introduire 32 mots de 16 bits et rajoute 2 mots, pour donner 34 mots. Ces deux itérations sont mise à zéro, elles ont été rajoutées exprés pour éviter les débordements des Blocs d'additions.

Ce module calcule l'itération complète de la boucle totale de l'algorithme de Montgomery CIOS (déjà étudié dans le chapitre II)

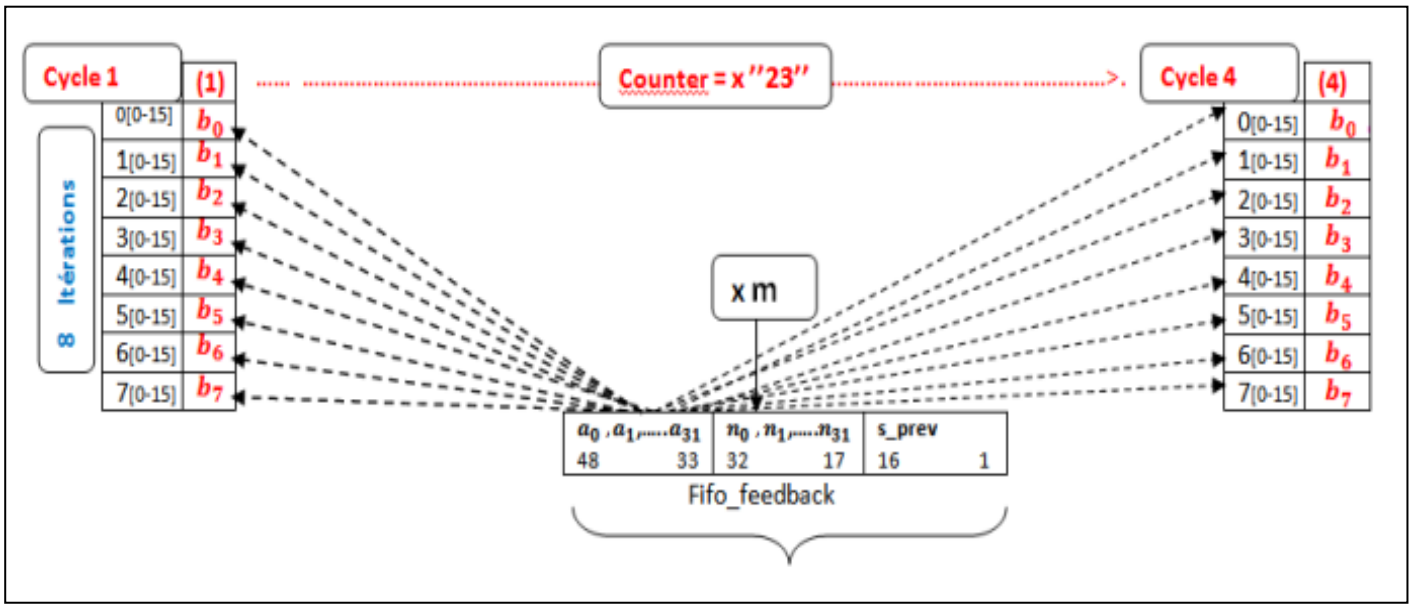


Figure VI.7. Le principe de fonctionnement du module''Montgomery_step'' [36]

Ce module accomplie l'itération complète de la première boucle de l'algorithme Montgomery CIOS pour les 32 mots, et prépare les opérations ci-dessous :

b0 --- > [t0+a0b0], [t1+a1b0], [t2+a2b0], . ……**[*t31+a31b0*]**

b1 --- > [s+a0b1], [s+a1b1], [s+a2b1], ... [s+a31b1]

…

b7 --- > [s+a0b7], [s+a1b7], [s+a2b7] .....[s+a31b7]

Nous maintenons les résultats de l'itération de la premier boucle de l'algorithme Montgomery CIOS, stoqués dans des registre 16*3, en attendant que le calcul de **m** sera génère.

Une fois que m_val = '1' => la valeur de **m** est calculé, on peux maintenant procéder au calcule de l'itération de la deuxième boucle de l'algorithme de Montgomery CIOS, comme le montre la figure.VI.7

b0 --- > s+m*n0, s+m*n1, s+m*n2 …….. s+m*n31

b1 --- > s+m*n0, s+m*n1, s+m*n2 …….. s+m*n31

….

b7 --- > s+m*n0, s+m*n1, s+m*n2 …….. s+m*n31

Le compteur est incrémenté 34 fois (32 +2) , pour faire passer les 32 mots de 16 bits, les deux additionnels mots sont à zéro (ces mots sont rajoutés pour éviter les débordements des blocs d'additions).

Le compteur incrémente de ''00'' a ''22'' = > de { 0 à 34 mots)

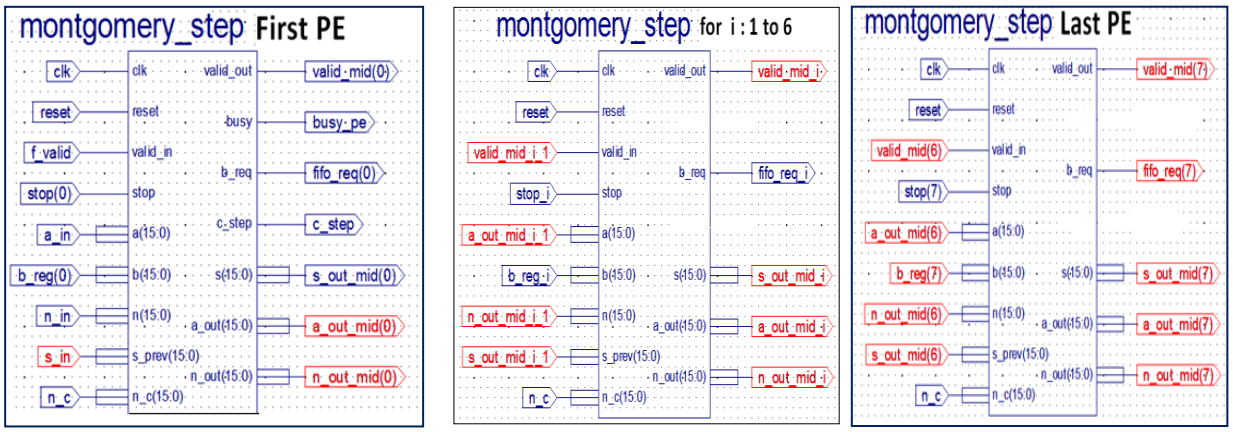


Figure VI.8. Schéma et description du bloc ''Montgomery_step'' [36]

En générale, le calcul du résultat final de l'architecture systolique est déduit après le cascadement d'un nombre illimités de PEs. Pour maintenir le process, une Fifo ou bien mémoire RAM a été mise en place pour stocker les résultats intermédiaires. Concernant notre architecture, le résultat est obtenue après 8 itérations (donc le cascadement de 8 PEs). Ce dernier sera stoqué dans une RAM, afin de faire une boucle de retour (feedback) sur le cycle prochain pour continuer le déroulement du reste des itérations, et balayer tout les mots jusqu'à atteindre 34 mots. Comme le montre la figure VI.9.

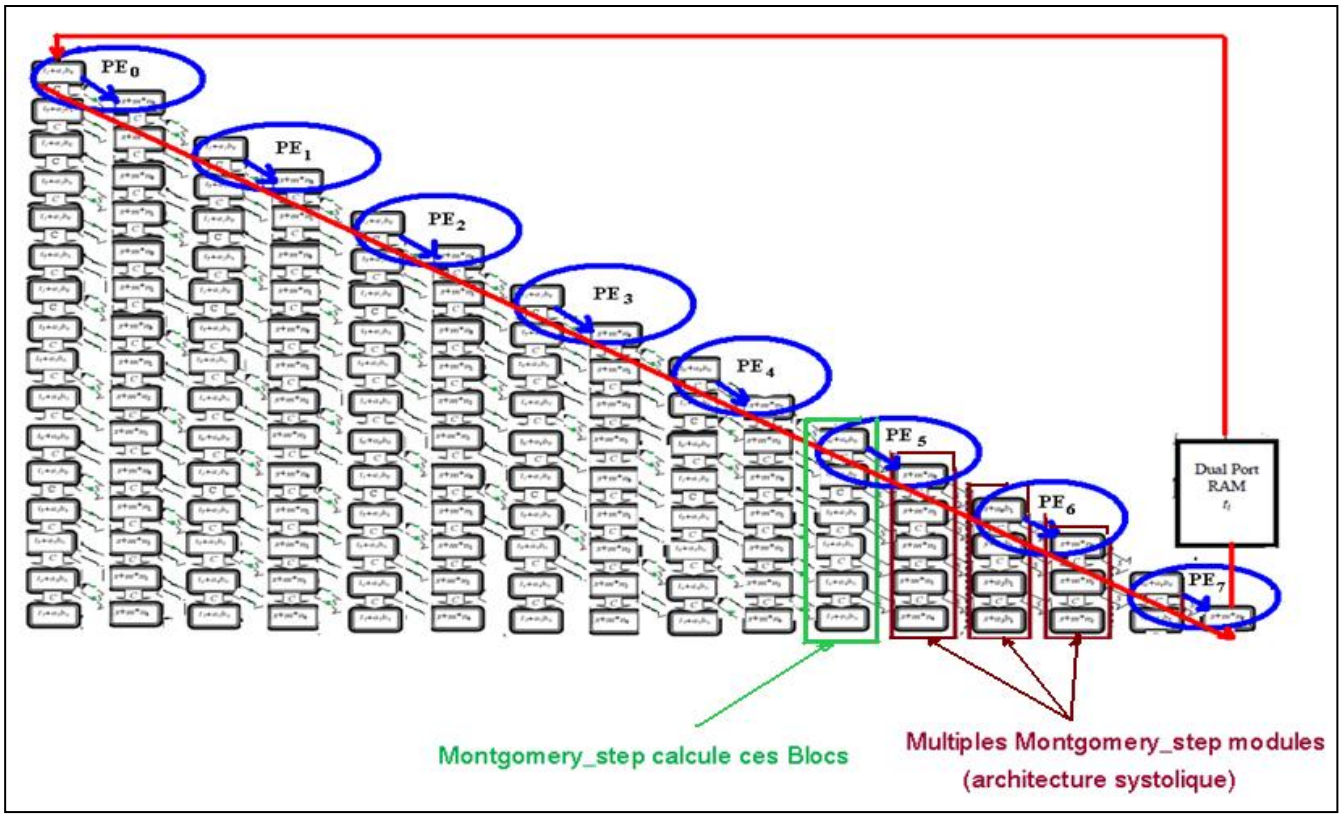


Figure VI.9. le module''Montgomery_step'' dans l'architecture systolique [36]

Le module '' Montgomery_step '' est le seule Bloc qui permet l'éxécution totale et complète de l'algorithme de Montgomery, car il finalise le déroulemnt des deux opérations suivantes en balayant toutes les itérations , comme le montre la figure VI.9

- Les multiplications de Montgomery : **[t0+a0b0], [t1+a1b0], [t2+a2b0],**……**[t31+a31b0]**
- Les Réductions de Montgomery : [**s+m*n0, s+m*n1, s+m*n2 …….. s+m*n31]**

IV.4.4. 1. La machine d'état du bloc ''Montgomery_step''

Dans un circuit séquentiel les transistions d'un état vers l'autre se font au fur et à mesure que le process se déroule, et l'algorithme s'éxécute. Dans l'algorithme de Montgomery décrit ci-dessus, le process passe par plusieurs états et transitions selon les étapes constituant l'algorithme et les valeurs d'entrées des signaux de contrôle (valid_in, valid_out, ..) y compris les registres et compteurs qui permettent le stockage et comptage des paquets de données pendant l'éxcécution de l'algorithme. Ce qui nous induit à présenter la machine d'états ci-dessous (voir figure VI.10) .

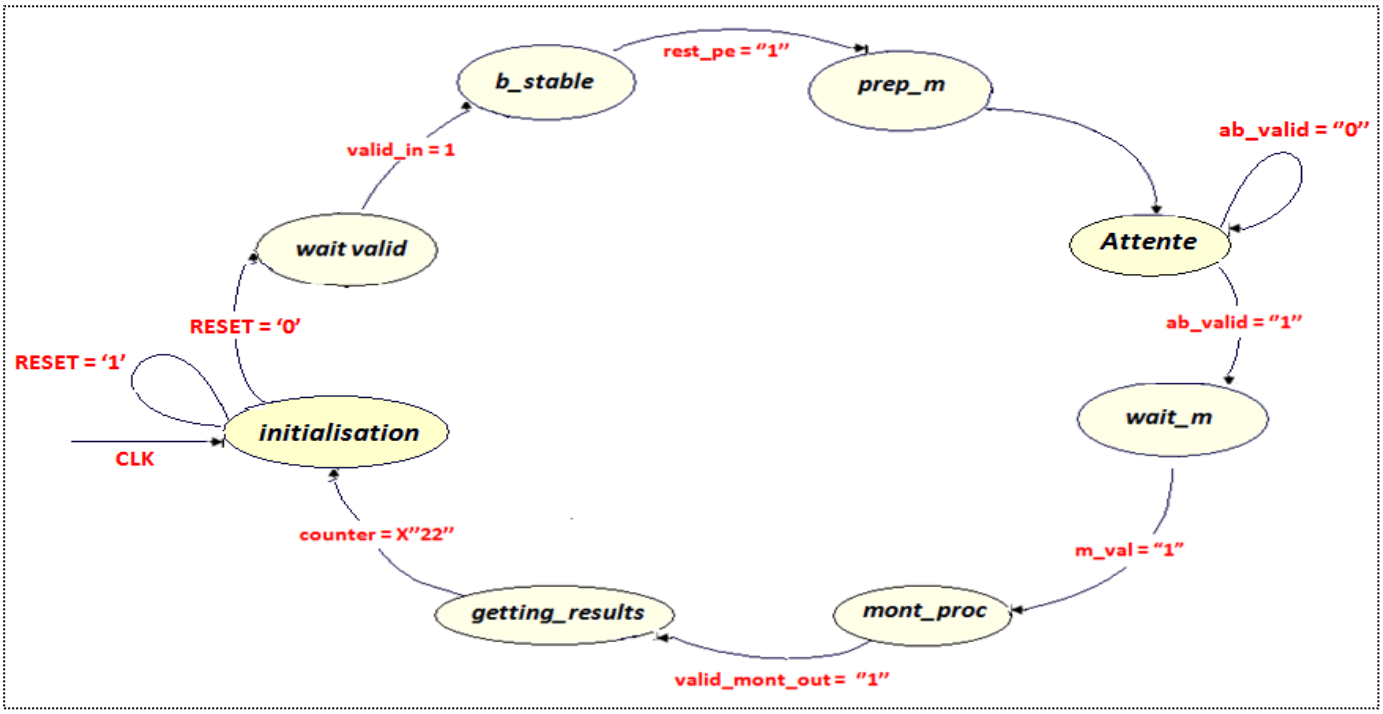


Figure VI.10. La machine d'état du bloc ''Montgomery_step''

Chacun des états présentés dans la figure ci-dessus a pour role bien précis pour le traitements des données dans l'IP-Core, et l'éxécusion des différentes étapes de l'algorithme de Montgomery, les états sont cités ci-dessous selon leur ordre de priorité et de fonction (voir Annexe B) :

1) wait_valid : le signal de controle qui permet d'informer que les 34 pquets de 16 bits ont été totalement traité par les 8 blocs de PE
2) b_stable : permet de stocker dans la fifo les 8 PEs du premier cycle pour se préparer au deuxiéme cycle
3) prep_m: préparer le calcule de la constant "m" dans le bloc PE
4) wait_m : synchroniser la constante " m" a la multiplication de Montgomery
5) mont_proc: introduire les résultats de la multiplication de Montgomery dans les registres aprpries
6) getting_results: afficher les résultats

IV.4.5. Le module ''Montgomery_mult''

C'est le module qui éffectue la multiplication de Montgomery, il est constitué des blocs suivants :

- 8 Blocs de **Montgomery_step**, composant un Cycle de l'architecture systolique contenant 8 PEs {0,.. ..7}, qui recouvrent 8 itérations en parallèle afin d'accélérer les sorties des Bloc Montgomery_mult comme le montre la Figure VI.10. Le résultat final de chaque cycle sera stocké dans la ''**fifo_feedback**''qui fera une boucle de retour au premier bloc de Montgomery_step (PE), pour continuer l'opération de multiplication. Comme le décrit la figure VI.11 , le nombre de Blocs de PEs cascadés en pipeline au sein du même cycle, peut être modifié de tel sorte qu'on doit pas dépasser les 8 PEs par cycle (ceci réduira la vitesse du core rsa_top).

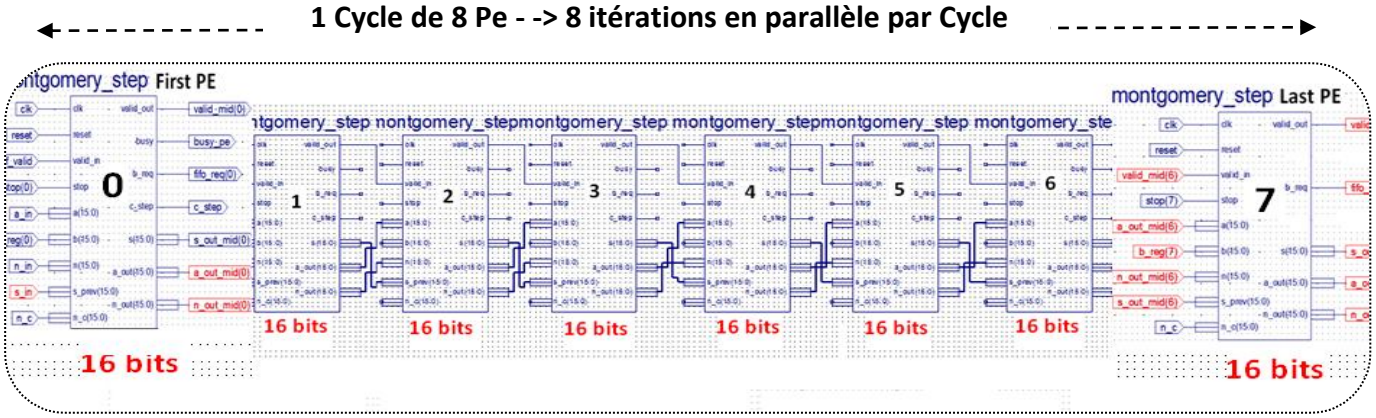


Figure VI.11. Cascadement des 8 Blocs de Montgomery _step au sein d'un Cycle [36]

- **fifo_512_bram** : { largeur = 16 bits , profondeur = 64 emplacements} permet de stocker les valeurs intermédiaires de b pendant l'opération de la multiplication de Montgomery (Figure VI.11)
- **fifo_256_feedback** : { largeur = 49 bits , profondeur = 32 emplacements} permet de stocker les valeurs intermédiaires du résultat final issu du Block PE_7 , aprés avoir passer par le cascadement des 8 Pes, et les reboucler au premier PE du prochain Cycle (PE_8) comme le montre la (Figure VI.12)

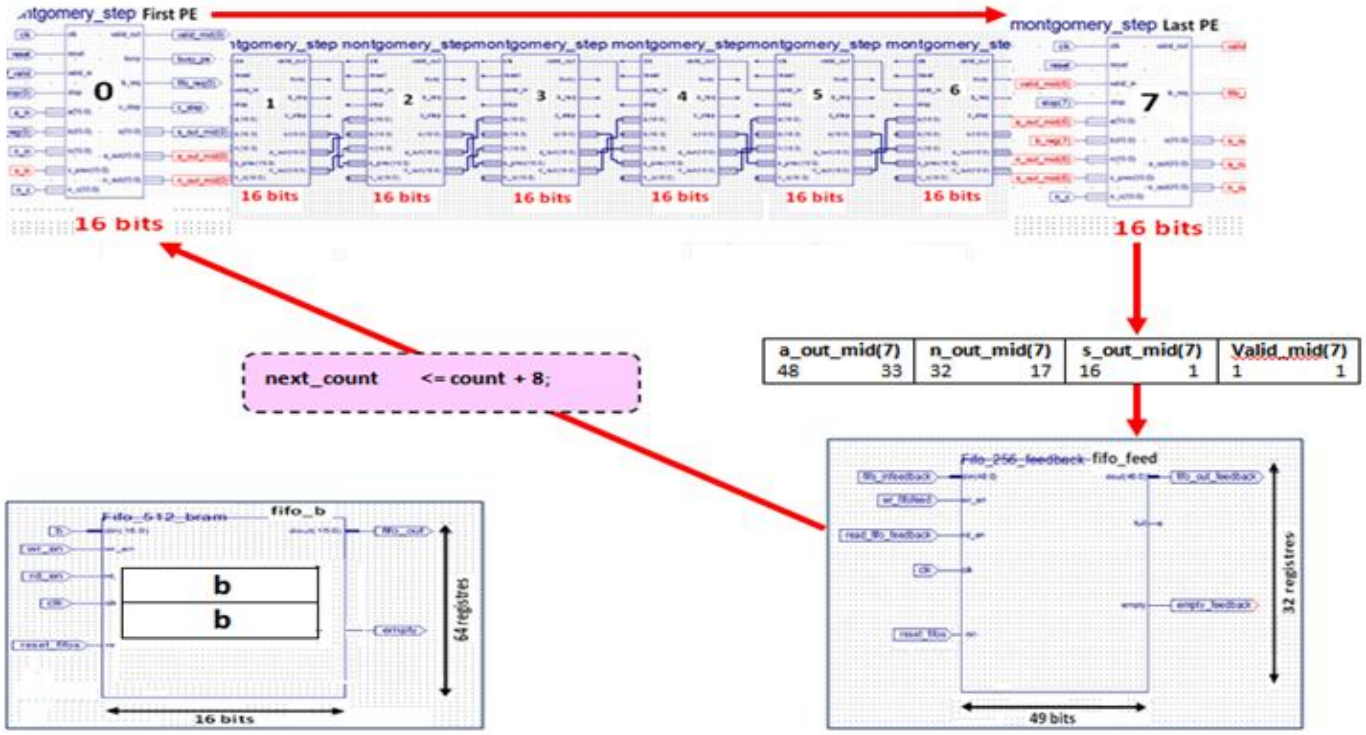


Figure VI.12. Principe de fonctionnement du Bloc Montgoemery_mult [36]

Des que le compteur sera incrementé de 8 itérations, le résultat de la fin du premier Cycle du Pipeline (PE_7), sera stoqué dans la fifo_feed, désque le valid_mid sera à '1'

```
rd_en   <= fifo_reqs(0) or fifo_reqs(1) or fifo_reqs(2) or fifo_reqs(3) or fifo_reqs(4) or fifo_reqs(5) or fifo_reqs(6) or fifo_reqs(7);
wr_en          <= '0';
fifo_in_feedback  <= a_out_mid(7)&n_out_mid(7)&s_out_mid(7)&valid_mid(7);          (the last PE)
read_fifo_feedback <= '0';
wr_fifofeed      <= '0';
```

Le compteur est incrémenté par ''8''à chaque fin du Cycle

```
if(reg_c_step = '1') then
next_count      <= count + 8;
end if;
```

La sortie du dernier PE du premier Cycle est l'entrée du premier PE du deuxiéme Cycle

```
wr_fifofeed <= valid_mid(7);
read_fifo_feedback <= '1';
a_in <= fifo_out_feedback(48 downto 33);
n_in <= fifo_out_feedback(32 downto 17);
s_in <= fifo_out_feedback(16 downto 1);
f_valid <= fifo_out_feedback(0);
if(empty_feedback='1') then
next_state <= process_data;
end if;
```

Et incrementé par ''1'' à chaque fin de traitement de PE pour faire passer les 34 mots (**x"22**", de l'architecture systolique

```
if(count_feedback = x"22") then
read_fifo_feedback <= '0';
next_state <= process_data;
end if;
```

Une fois que le traitement des 34 mots est fait, le résultat final sera affiché à la sortie du premier PE du dernier Cycle

```
if( count = x"20") then
s             <= s_out_mid(0);
valid_out     <= valid_mid(0);
end if;
```

Comme le nombre d'itérations par cycle dans l'architecture systolique du core rsa_top est de 8, donc pour traiter les 34 mots de 16 bits, il est nécessaire de répéter l'opération du cascadement des 8 blocs en quatre reprise (balayer les quatre Cycles de 8 PEs), afin de recouvrir les 34 itérations de Montgomery CIOS, et assurer le fonctionnement du Core à 512 bits (8*4 = 32 mots de 16 bits), comme le montre la figure ci-dessous (Figure.VI.13)

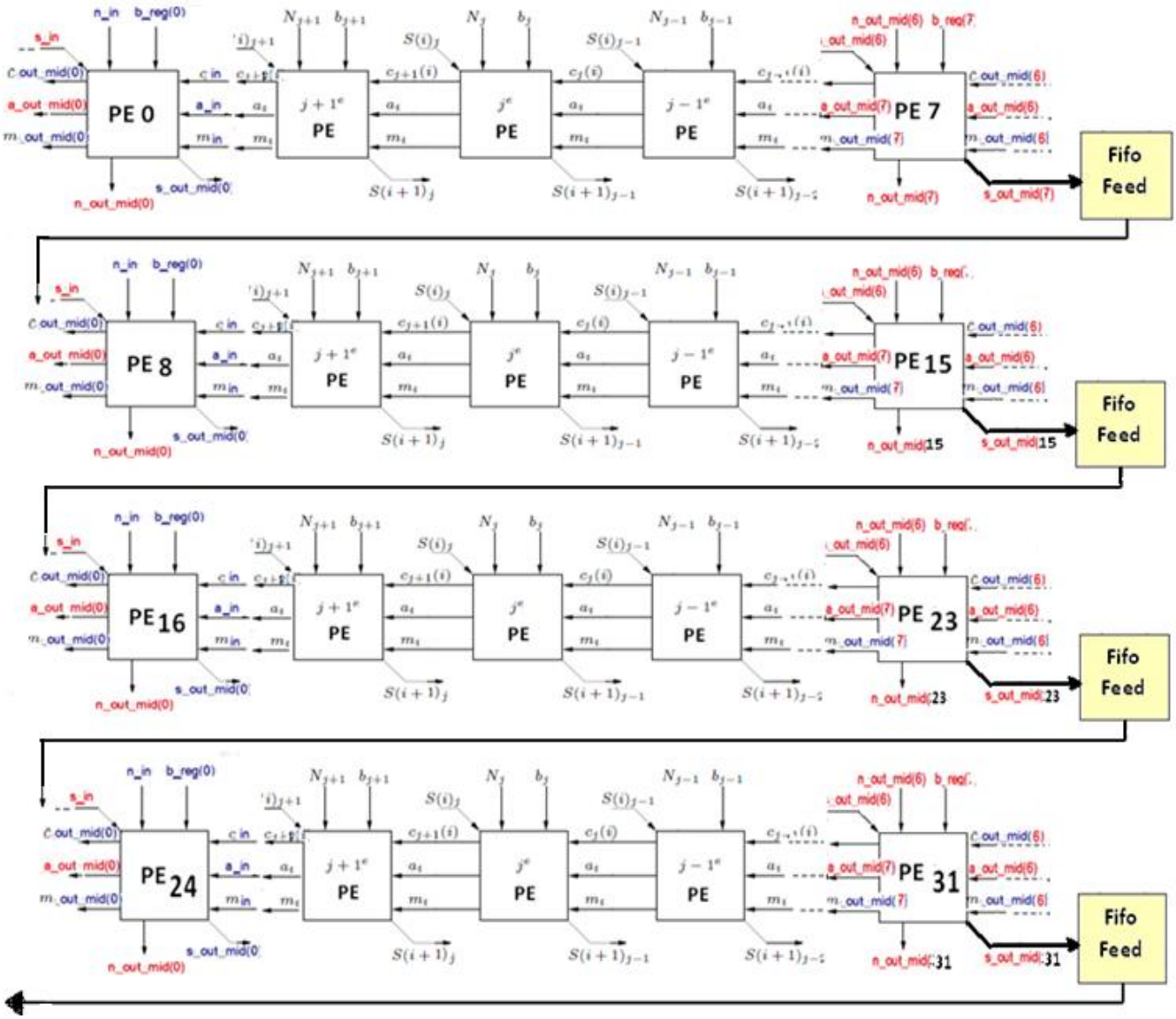


Figure VI.13. L'architecture Systolique (8 * 4) basé sur l'algorithme Montgomery CIOS [36]

IV.4.5. 1. La machine d'état du bloc ''Montgomery_mult''

Dans un circuit séquentiel les transitions d'un état vers l'autre se font au fur et à mesure que le process se déroule, et l'algorithme s'éxécute. Le déroulement des étapes de l'algorithme de Montgomery décrit ci-dessous (voir figure VI.14) passe par plusieurs états et transitions selon les étapes constituant l'algorithme et les valeurs d'entrées des signaux de contrôle (valid_in, valid_out, ..) y compris les

registres et compteurs qui permet le stockage et comptage des paquets de données pendant l'excécution de l'algorithme. Ce qui nous induit à présenter la machine d'états ci-dessous (voir figure VI.14) .

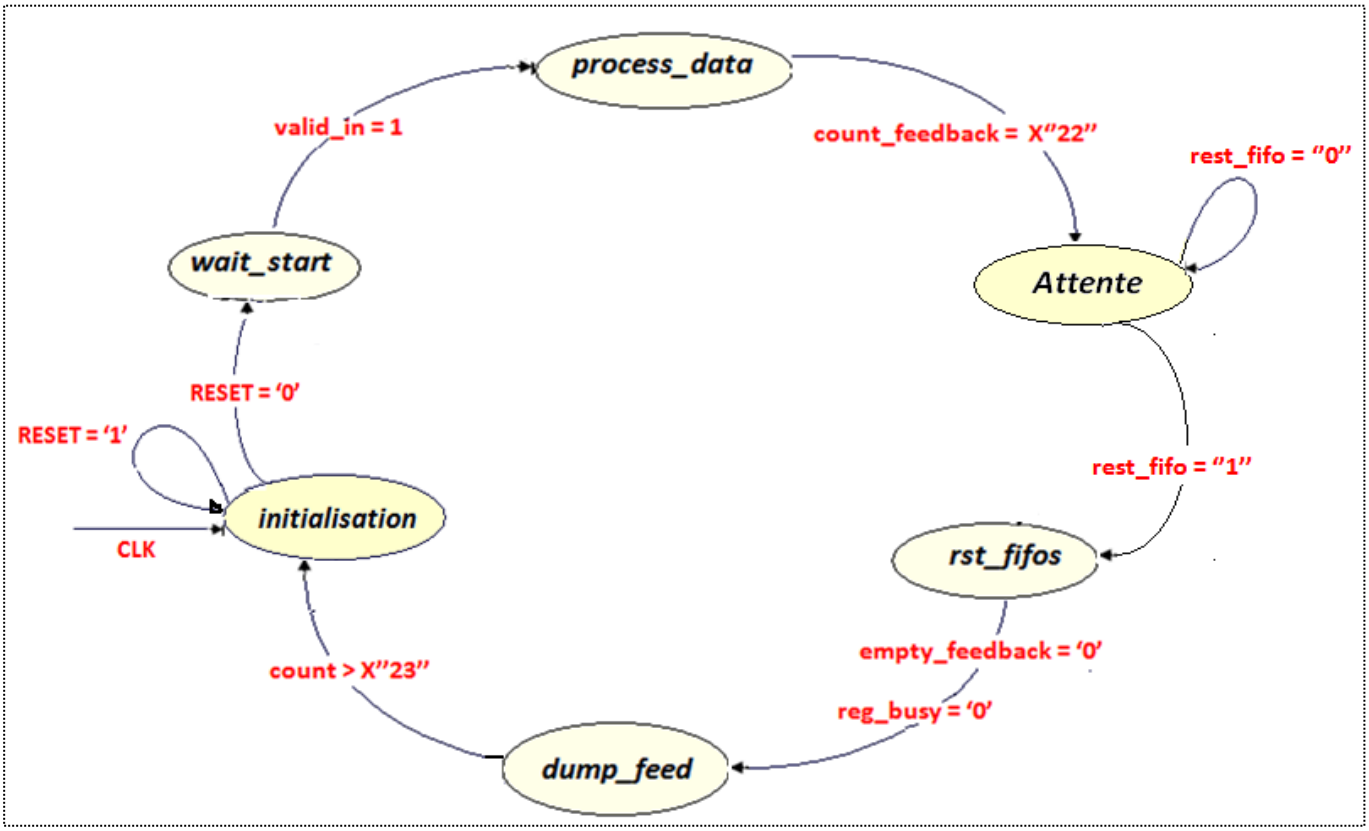


Figure VI.14. La machine d'état du bloc ''Montgomery_mult''

Chacun des états présentés dans la figure, a pour role bien précis pour le traitements des données dans l'IP-Core, et l'éxécusion des différentes étapes de l'algorithme de Montgomery, les états sont cités ci-dessous selon leur ordre de priorité et de fonction (voir Annexe B) :

1) wait_start : indique que le compteur a fini le comptage des 32 paquets .
2) process_data : C'est la ou l'algorithme de Montgomery s'exécute en calculant la multiplication des 32 paquets
3) Rst_fifos: réinitialiser les fifos afin de pouvoir stokcer les données intermédiaires au cours du traitement
4) Dump_feed : une fois que la fifo est vide, le compteur s'incrémente pour recevoir les paquets du prochain cycle

IV.4.6 Le module ''rsa_top'' [Annexe B]

C'est le module qui permet le calcule de l'équation de l'exponentiation modulaire : **S=x^ymod m,** en utilisant l'algorithme de l'exponentiation binaire de Montgomery (Montgomery Ladder Exponontiation) afin de calculer : x^y

C'est un algorithme ***Square and Multiply*.** Il utilise le même principe de l'exponentiation binaire pour chaque bloc de multiplication. Il permet d'accélérer la multiplication modulaire. Physiquement, il comporte :

- Deux blocs de CIOS de Montgomery, qui effectuent des multiplications en parallèle, l'un pour l'élévation au carrè et l'autre pour la multiplication et vis-versa.

- Une Fifo qui permet de lire et synchroniser les résultats sortants des blocs CIOS, en les concaténant et respectant l'ordre dans lesquels ont été générer afin de les mettre en ordre et les regénérer encore une fois pour les injecter dans les Blocks CIOS. Les deux opérations sont indépendantes et s'exécutent en parallèle. Comme le montre Figure VI.15

- Une mémoire RAM (Mem_exp) pour le stockage de l'exposant **y,** et un registre à décalage à droite y_i pour décaler l'exposant à chaque itération. Les entrées des blocs CIOS changent de fonction selon la nature du bit de l'exposant y_i, suivant la méthode '' **Left to Right** '' de l'exponentiation binaire (voir l'algorithme VI.1)

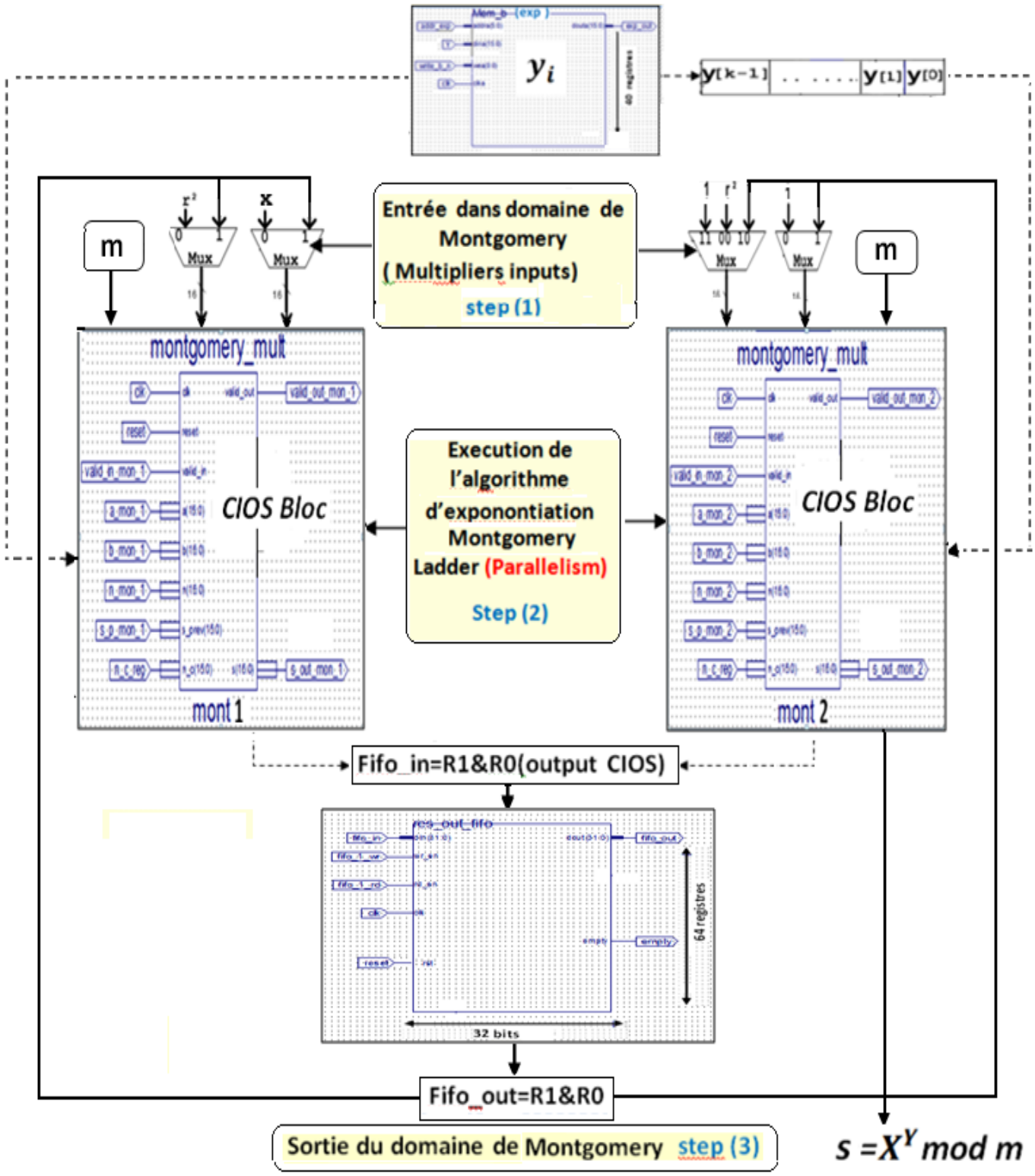


Figure VI.15. L'architecture globale du bloc rsa_top [36]

L'algorithme de l'exponentiation binaire de Montgomery Ladder a été implémenté dans le bloc principale rsa_top afin de bénéficier de l'avantage du parallélisme de ces deux bloc de multiplication de Montgomery et surtout qui permet une optimisation du temps d'exécution de 50%, de plus elle est simple à réaliser.

Le déroulement de l'algorithme de la méthode a été étalé de la manière suivante :

1.Etape initiale : Initialisation des variables R_0 et R_1 respectivement à 1et à x dans le domaine de Montgomery en réalisant les multiplications de Montgomery appliquées aux constantes 1 et x avec $r^2=2^{2(k*n)} \bmod N$ (r : est la constante de Montgomery , dans notre cas $r= 2^{(k*n)} \rightarrow r= 2^{(16*32)}$)

{ k= 16 bits(taille du paquet) , n =32 (nombre de mots)}

$$R_0 = Mon(1) = Montgomery\ (1, r^2) = r \bmod m$$

$$R_1 = Mon(X) = Montgomery\ (X, r^2) = X^* r \bmod m$$

***2. Exécution de la boucle* :** calcul itératif de l'exponentiation de Montgomery Ladder (Excécusion de la boucle)

pour $y_i=1$

$R_0 = Montgomery\ (R_0 , R_1 , M)$ (opération de multiplication)

$R_1 = Montgomery\ (R_1 , R_1 , M)$ (opération d'élévation au carré)

Pour $y_i=0$

$R_0 = Montgomery\ (R_0 , R_0 , M)$ (opération d'élévation au carré)

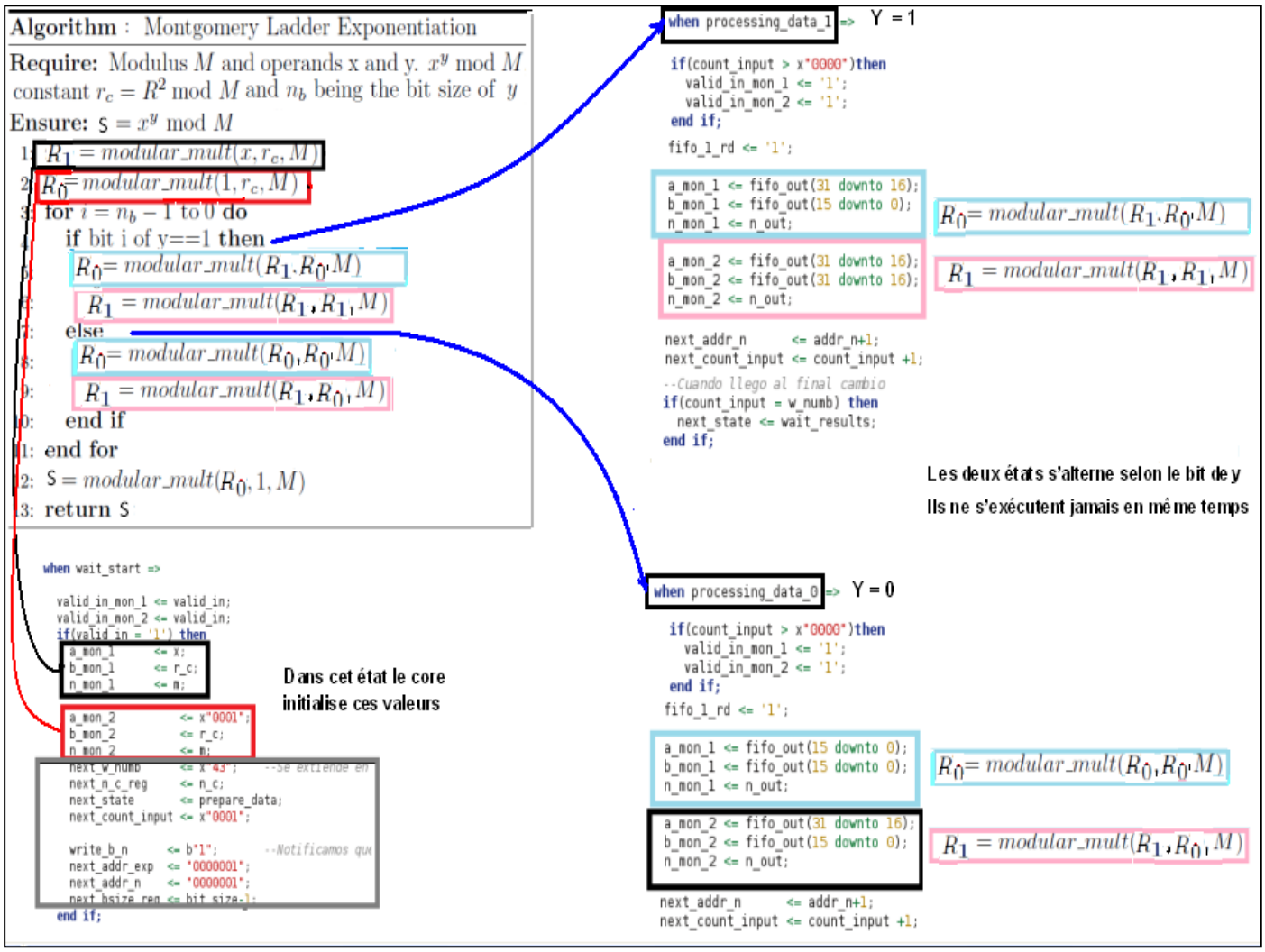
$R_1 = Montgomery\ (R_0, R_1 , M)$ (opération de multiplication)

3.Etape finale Sortir du domaine de Montgomery et le résultat obtenu est dans S à la fin de la boucle**.**

S= $Montgomery\ (R_0, 1, M)$ indépendamment de y_i

L'entrée au domaine de Montgomery Ladder et la sortie du domaine revient à faire deux multiplications supplémentaires ce qui signifie que l'exponentiation modulaire pour un exposant de k bits nécessitera (k+2) multiplications de Montgomery.

Algorithme VI.1. Description de l'algorithme d'exponontiation binaire Montgomery Ladder du module rsa_top [36]



IV.4.6.1. La machine d'état du bloc rsa_top

Dans un circuit sequentiel les transistions d'un état vers l'autre se font au fur et à mesure que le process se déroule. Dans l'algorithme de lexponontiation binaire de Montgomery Ladder décrit ci-dessus, le process passe par plusieurs états et transitions (numéroté par ordre de priorité et de tache) selon les étapes constituant l'excécusion de l'algorithme et les valeurs d'entrées des signaux de contrôle (valid_in, valid_out, ..) y compris les registres et compteurs qui permettent le stockage et comptage des paquets de données pendant l'éxécution de l'algorithme. Ce qui nous induit à présenter la machine d'états ci-dessous (voir figure VI.16) .

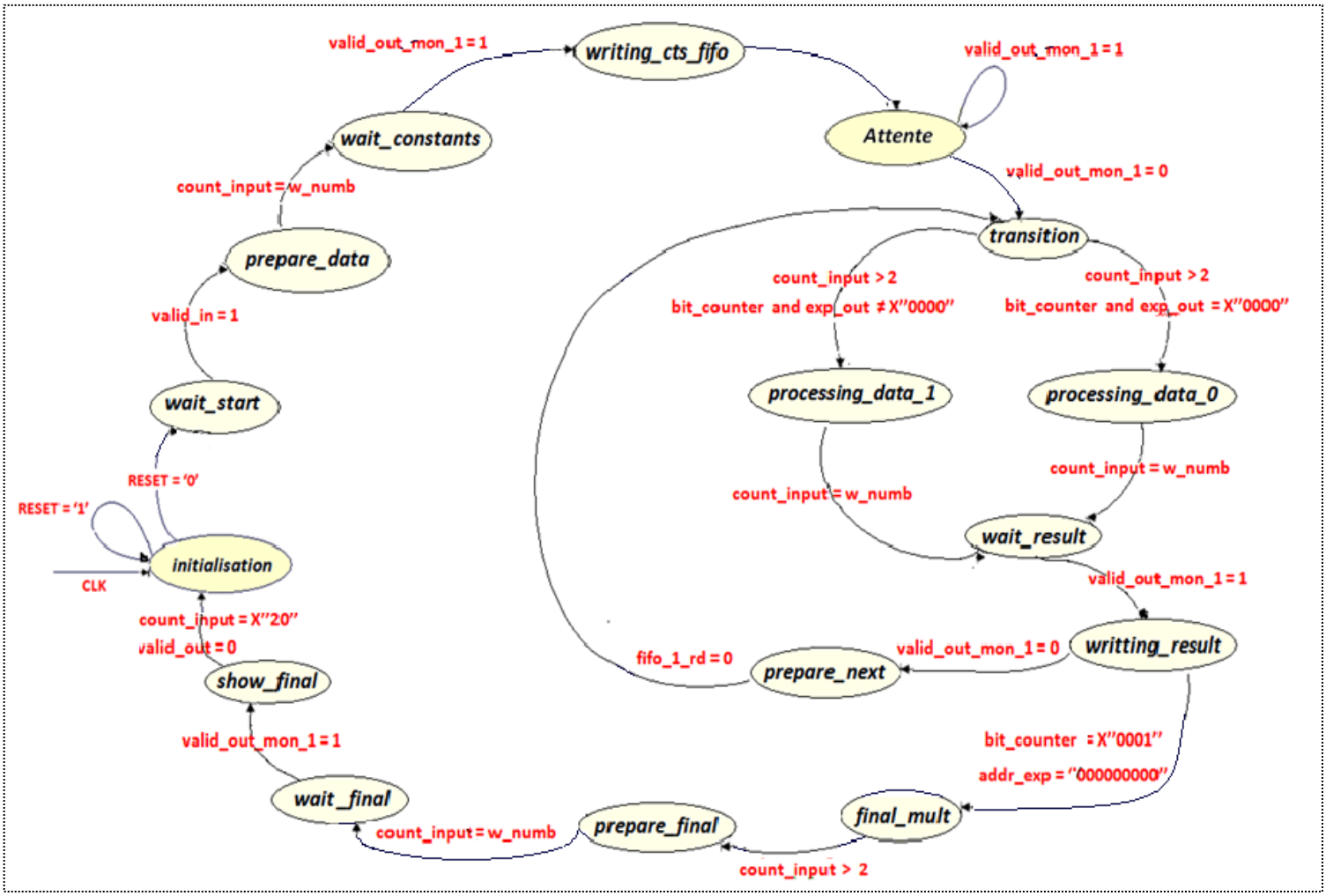


Figure VI.16. La machine d'état du bloc rsa_top

Chacun des états présentés dans la machine d'états a pour role bien précis pour le traitements des données dans l'IP-Core, et l'éxécusion des différentes étapes de l'algorithme de l'exponontiation binaire. Les états sont cités ci-dessous selon leur ordre de priorité et de fonction (voir Annexe B) :

1) Wait start: décrit l'entrée au domaine de Montgomery, c'est dans cet état ou le core initialise ces valeurs.
2) prepare_data : préparer les données d'entrées de l'IP-Core { x, y, m, r_c, n_c}
3) wait_constants : attendre les valeurs calculées des élévations au carrés et des multiplications des blocs mon_1 et mon_2. (c'est que l'algorithme de Montgomery commence à s'éxécuter)
4) writting_cts_fifo : stocker ces valeurs dans la fifo

5) transition: c'est la que les transition commence vers pd0 et pd1 selon la valeur de y (l'exposant)
6) - processing_data_0 : si y=0 , R_0 s'éléve au carre et par la suite le resultat sera multiplié a R_1 (voir l'algorithme VI.1)
   - processing_data_1 : si y=1 , R_1 s'éléve au carré et par la suite le résultat sera multiplié a R_0 (voir l'algorithme VI.1)
7) wait_result: attendre les résultat S_0 et S_1 déduits des blocs mon_1 et mon_2 (voir figure VI.13)
8) writting_result : concaténer et stocker les résultats précédants dans la fifo (voir figure VI.13)
9) final_mult : calculer la multiplication final de deux blocs de Montgomery
10) prepare_next : se préparer à l'état de transition si l'opération de multiplication n'est pas finie
11) prepare_final : préparer le résultat final déduit de l'un des bloc de Montgomery
12) wait_final : sortie du domaine de Montgomery
13) show_final : Afficher le résultat final de l'exponontiation binaire de Montgomery Ladder

IV.5 Conclusion

Dans ce chapitre, nous avons présenté notre IP-Core rsa_top et son principe de fonctionnement basé sur l'algorithme de l'exponentiation modulaire binaire s'appuyant sur la multiplication de Montgomery.

Notre architecture série (serial) est basée sur une structure parallèle permettant l'éxécution de deux multiplications modulaires de Montgomery en parallèle, ce qui a réduit énormément les temps de chiffrage et déchiffrage des messages.

La méthode binaire MSB est celle qui s'adapte le mieux à notre champ d'étude car elle nous permet de calculer les grands exposants, et elle nous offre la possibilité de calculer en parallèle la multiplication et l'élévation au carré qui permet une optimisation du temps d'exécution de 50%.

Pour utiliser au mieux les ressources offertes par les circuits FPGA, nous avons consacré le chapitre suivant à l'extension de l'IP-Core rsa_512 vers rsa_1024, afin de réduire au maximum le hardware nécessaire à notre architecture entre temps augumenter la taille de la clef et le nombre de paquet a traiter lors de l'opération du cryptage.

Le fonctionnement de l'architecture globale a été validé par des simulations fonctionnelles et temporelles et implémenté sur un circuit FPGA de Xilinx de la famille Virtex-5, en l'occurrence le circuit : xc5vsx95t-3ff1136. Les résultats de simulation et d'implémentation sont discutés dans le prochain chapitre.

Chapitre V

Simulation/Implémentation de l'IP-Core RSA_512/1024 sous ISE

V.1. Introduction

Dans ce chapitre, nous allons présenter les résultats d'implémentation de notre IP-Core rsa sur circuit FPGA, ainsi que les résultats de simulation fonctionnelles et temporelles.

Le flot de conception sur circuit FPGA de Xilinx se décompose en quatre étapes :

1. Description de l'architecture.
2. Simulation fonctionnelle.
3. Synthèse et implémentation.
4. Simulation temporelle et vérification.

L'architecture de notre IP-Core rsa_512 a été conçue sous l'environnement Foundation ISE 9.2i de Xilinx .

En premier lieu, l'extension de rsa_512 vers rsa_1024 a été élaborée afin de comparer les résultats obtenus des deux architectures. Dans une architecture serial, ya uniquement deux facteur qui changent lors de l'extension :

1) La latence (appelé temps de simulation, ou temps de réponse du circuit). C'est le temps enregistré pendant l'opération de cryptage, il est limité entre valid_in (signal de validation des données d'entrées) et valid_out (signal de validation de la sortie) .
   Dans notre cas la latence de rsa_1024 doit etre plus élevé que celle de rsa_512, car le temps pour traiter 64 paquets est bein évidemment plus grand que celui de 32 paquets.
2) Le deuxième facteur est la consommation de l'énergie (ou de puissance) qui est due aux temps de transitions de 1 à 0 dans un flot de bits. Ce facteur appelé « switching activity », et dans notre cas c'est vérifié, car plus le nombre de paquets est élevé dans un flot de données, plus les transitions augmentent, plus la consommation d'énergie augmente.

Une simulation fonctionnelle et temporelle a été effectuée à l'aide de (ModelSim) pour les deux architectures, qui donne des résultats sous forme de chronogrammes.

Après la simulation fonctionnelle, une phase de synthèse a permis de donner un rapport détaillé sur les ressources du circuit FPGA consommées ainsi que la fréquence maximale de fonctionnement.

Par la suite, une phase de placement et routage a éclaté schématiquement la surface consommée par l'architecture globale et a configuré les routages d'interconnexion entre les différents blocs des ressources du circuit FPGA.

Finalement, la simulation temporelle a permis de vérifier si le circuit obtenu respecte les contraintes temporelles et utilise les délais des portes et les délais des interconnexions pour calculer la vitesse maximale. La figure V.1 regroupe toutes les étapes d'une conception sur circuit FPGA.

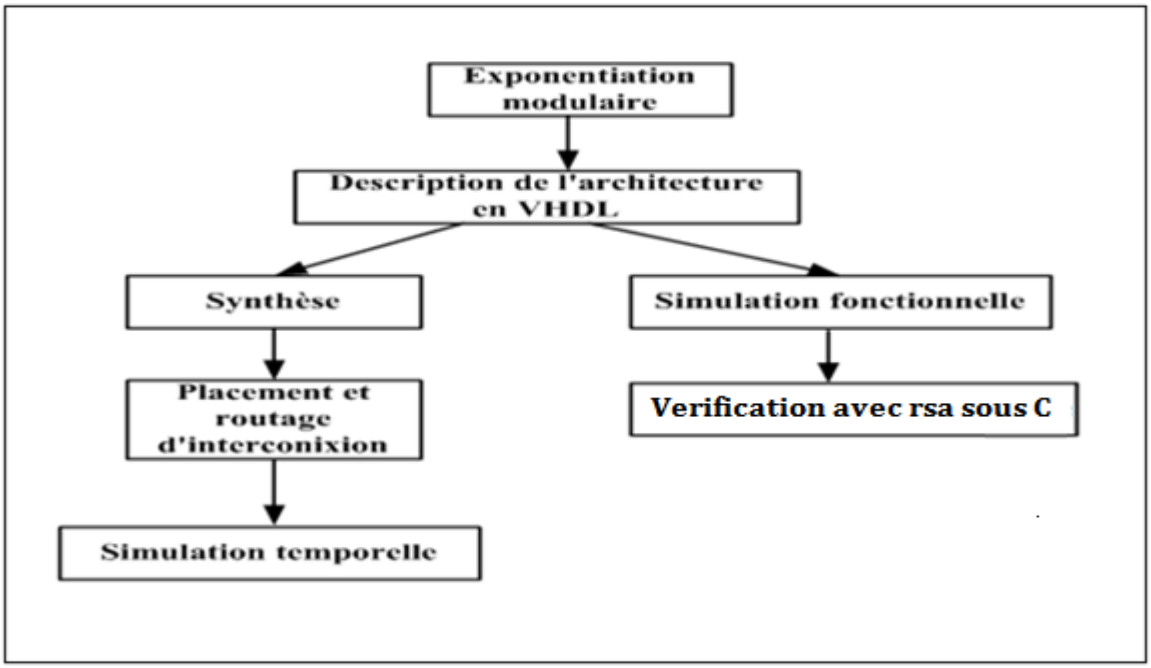


Figure V.1. Schéma des étapes de conception de notre architecture [36]

V.2. L'outil ISE

ISE ou *Integrated Software Environment* (ISE) est le logiciel de design de Xilinx qui nous permet de faire entrer un circuit (comme design) à travers *design entry* (*Xilinx device programming*). Le *ISE Project Navigator* va nous guider à développer et à suivre l'évolution de votre design à travers les différentes étapes dans le *ISE design flow*.

V.2.1. Les étapes de l'ISE design flow

V.2.1.1. Entrée du design

C'est la première des étapes dans l'*ISE design flow*. Durant cette étape, on crée les fichiers sources basés sur les objectifs du circuit désiré. On peut créer un fichier de haut niveau utilisant un langage de description matériel (Hardware Description Language-HDL-) comme VHDL, Verilog ou ABEL ou sinon en utilisant un schématique. Comme on peut utiliser plusieurs formats pour les fichiers sources de bas niveau dans notre design.

V.2.1.2. Synthèse

Après le *design entry* on passe à la synthèse. Durant cette étape, les fichiers VHDL et Verilog ou autres deviennent des fichiers netlist qui sont acceptés comme des entrées pour l'étape de l'implémentation.

V.2.1.3. Implémentation

Après la synthèse, nous exécutons le *design implémentation*, qui convertit le design (fichier) logique en un format de fichier physique qui pourra être chargé dans le circuit cible. A partir du navigateur de projets (*Project Navigator*) on peut exécuter le processus d'implémentation en une seule étape ou en plusieurs. Les processus de l'implémentation varient suivant qu'on cible un FPGA (Field Programmable Gate Array) ou un CPLD (Complex Programmable Logic Device).

V.2.1.4. Vérification

Le fonctionnement du circuit peut être vérifié en différents points dans le *design flow*. On peut utiliser un logiciel de simulation (*ModelSim*, *Xilinx ISE Simulator*,...) pour vérifier le fonctionnement et le timing du design ou une portion de celui-ci.

Le simulateur interprète le code VHDL ou Verilog en fonctionnement du circuit et donne des résultats logiques pour le HDL décrit afin de déterminer l'opération correcte du circuit. La simulation nous permet de créer et vérifier les fonctions complexes dans un temps relativement court. Nous pouvons aussi faire une vérification directe sur le circuit après l'avoir programmer.

V.2.1.5. Configuration du circuit

Après avoir générer un fichier de programmation, on passe à la configuration de notre circuit. Durant la configuration, on génère des fichiers de configuration et on charge les fichiers de programmation dans le circuit à partir de l'ordinateur. Comme le montre la figure V.1

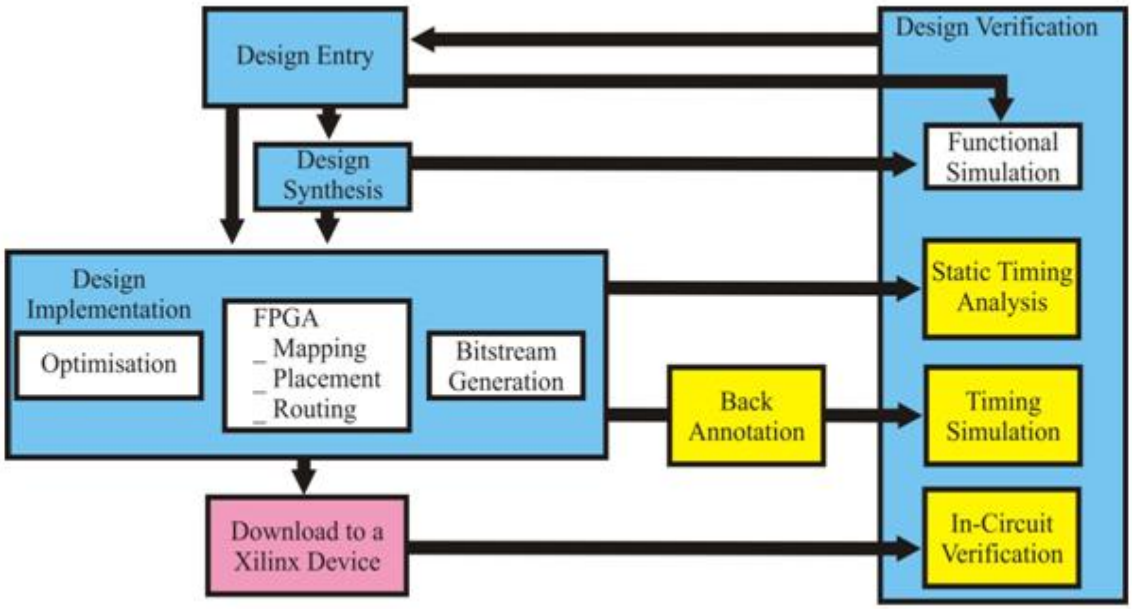


Figure V.1 Flot de conception Xilinx générique

Pour cela, les étapes que nous avons suivies pour l'implémentation de notre IP-Core rsa se résument comme suit :

- Simulation fonctionnelle par l'outil Modelsim de chacun des IP-cores (rsa_512, rsa_1024) séparément.
- Synthèse de l'architecture de chaque bloc séparément par l'outil de synthèse XST de Xilinx.
- Vérification des résultats de simulation de l'architecture globale avec l'outil Modelsim
- Implémentation de l'architecture globale des deux cores sur le circuit FPGA ***xc5vsx95t-3ff1136*** de la famille VIRTEX-V.

V.3. Extension de l'IP-Core rsa_512 vers rsa_1024 [Annexe C] [40]

L'extension de l'IP-Core d'une taille à l'autre se fait en paramétrant le design, par la modification des critères nécessaires permettant le paramétrage. Ces critères doivent répondre au conditions ci-dessous:

1) Le design doit être flexible pour qu'il soit intégrer dans une petite ou grande surface d'un FPGA en ajustant le nombre des PEs .
2) La longueur des bits des mots doit être paramétrique, afin d'utiliser toute la performance des multiplieurs.

3) Le design doit être paramétrable pour qu'il fonctionne avec des opérandes de différentes tailles { 512, 1024, 2048, ..4096 bit }.
4) L'implémentation doit résister aux chaines d'attaques
5) Les multiplieurs du circuit doivent fonctionner avec une fréquence maximale (c'est à dire une période minimale)
6) Tous les opérands doivent être stockés dans des blocks RAMs pour assurer une consommation minimale de la surface.

V.3.1. Les Composants Paramétrique d'une architecture (design) [16]

Suivant les critères cités ci-dessus, on peut paramétrer un design selon les besoins de notre application. Les critères majeur permettant de paramétrer une architecture sont cités ci-dessous :

1) Radix (R) : Ce paramètre détermine la taille d'un mot en bits (dans notre cas, elle est de 16 bits par mot), L'utilisation des mots de taille grandes : c.a.d : R grand >>, a un effet négative sur la fréquence car il génère de longue carry dans les additionneurs utilisés dans les PEs. Le Radix a un effet directe sur la fréquence du circuit, il est lié au délais du *chemin critique* dans les additionneurs. Ce paramètre doit être paramétrer de tel sorte d'avoir le grand avantage des blocs multiplieurs afin d'achever une meilleur performance du temps.

2) Nombre de mots : le Radix multiplié par le nombre de mots détermine la taille totale de l'opérande (Bit-length) pour notre cas : R = 16 et le nombre de mots = 32, le nombre de mots détermine la profondeur des Blocks RAM.

Rsa_512 → 16 * 32 = 512 bits
Rsa_1024 → 16 * 64 = 1024 bits

V.3.2. L'extension de l'IP-Core de 512 bit vers 1024bit [Annexe C]

Dans notre architecture Serial (série) , les données d'entrées sont introduites en série, 32 mots de 16 bits, qui sont introduit mot par mot en série, dans le cas de rsa_512 , et 64 mots de 16 bits dans le cas de rsa_1024. Donc la différence entre les deux cores se trouve uniquement dans le nombre de mots introduits en entrée de l'IPCore.

Dans une architecture serial, y'a uniquement deux facteurs qui changent lors de l'extension :

1) La latence (appelé temps de simulation, ou temps de réponse du circuit), et ce temps est celui enregistré pendant l'opération de cryptage, il est limité entre valid_in (signal de validation des données d'entrées) et valid_out (signal de validation des données de la sortie) .

2) Le deuxième facteur est la consommation de l'énergie (ou de puissance) qui est due aux temps de transitions de 1 a 0 dans un flot de bits. Ce facteur appelé « switching activity », et dans notre cas c'est vérifié, car plus on augmente le nombre de paquets dans un flot de données, plus les transitions augmente, plus la consommation d'énergie augmente.

Notre IP-Core est composé de 6 modules (composants) et de 4 Core-générateurs programmés en langage VHDL. Les modules décrivent le principe de fonctionnement de l'algorithme de Montgomery au sein des PEs , et les Core- générateurs décrivent le fonctionnement des Fifos et mémoires mise en place pour stocker et mémoriser les résultats intermédiaire de l'architecture, ces derniers sont paramétrés de tel sortes qu'ils aient la capacité de stocker une taille minimale de 32 paquets de 16 bits pour l'IP-core rsa_512 et 64 paquets de 16 bits pour rsa_1024.

V.3.2. 1. Paramétrage au niveau des composants de l'IP-Core (modules vhdl) [Annexe C / rsa_1024]

Pour étendre l'IP-Core d'une taille de **512 bit** à **1024 bit** , on doit modifier deux critéres :

- Le changement se fait dans les valeurs des compteurs « registres » au sein de tous les fichiers vhdl {rsa_top.vhd, Montgomery_mult, Montgomery_step, Pe_wrapper, Pe, m_calc} de l'IP-Core. Il s'agit de remplacer toutes les valeur qui étaient a **x"20"** (32 mots) dans rsa_512 par la valeur **x"40"** (64 mots) dans rsa_1024.

- Remplacer toutes les valeurs qui étaient à **x"23"** (35 mots) dans rsa_512 , par **x"43"** (67 mots) dans rsa_1024, qui explique l'addition de 3 mots (de valeurs 0) a été faite exprès pour éviter le débordement généré par les additionneurs des PEs.

V.3.2. 2. Paramétrage au niveau des Core-generateur de l'IP-Core (Coregens) [Annexe C / rsa_1024] :

Concernant les coregens, le changement se fait aussi dans les mémoires et les fifos plus précisément au niveau des profondeurs des mémoires, qui doit être minimum le double pour rsa_1024 afin de garantir suffisamment de l'espace pour stocker tout les mots de l'opérande.

Pour ce qui suit, nous allons procéder au paramétrage des core-générateurs pour migrer vers l'IP-Core rsa_1024.

V.3.2.2.1. Mémoire : Mem_b [Annexe B/C]

Cette mémoire est de largeur de 16 bits et de profondeur de 40 dans rsa_512 , elle sera remplacée par une mémoire de largeur de 16 bits et d'une profondeur supérieure a 40, on choisit 261 qui permettra largement de stocker entièrement les 64 paquets de 16 bits . voir Figure V.2

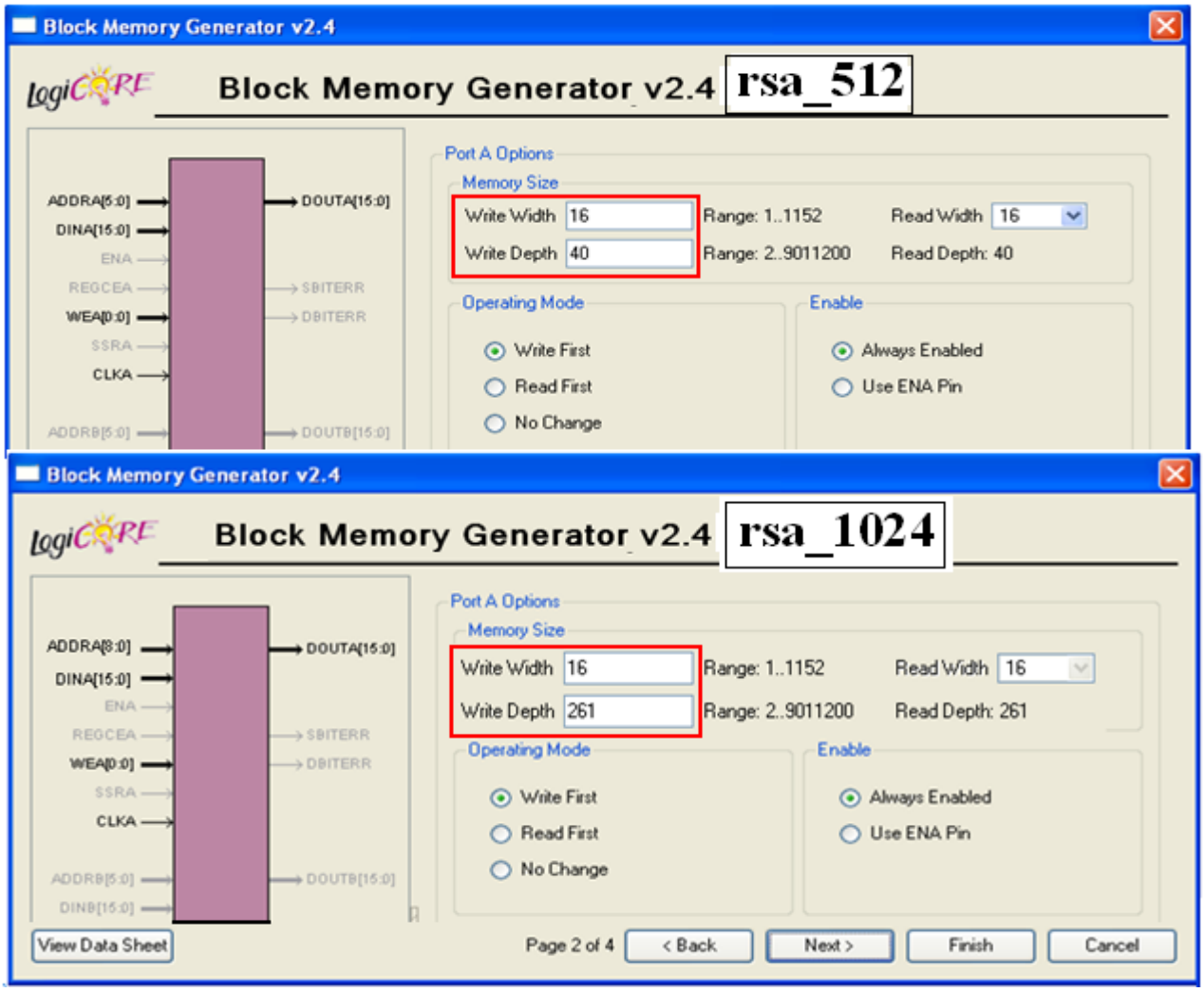


Figure V.2. Extension de la mémoire Mem_b de rsa_512 vers rsa_1024 [36]

V.3.2.2.2. res_out_fifo [Annexe B/C]

Cette Fifo est de largeur de 32 bits et de profondeur de 64 dans rsa_512 , elle sera remplacée par une Fifo de largeur de 32 bits et d'une profondeur cette fois-ci de 1024, ce qui permettra largement de stocker entièrement les 64 paquets de 16 bits. Comme le montre la Figure V.3

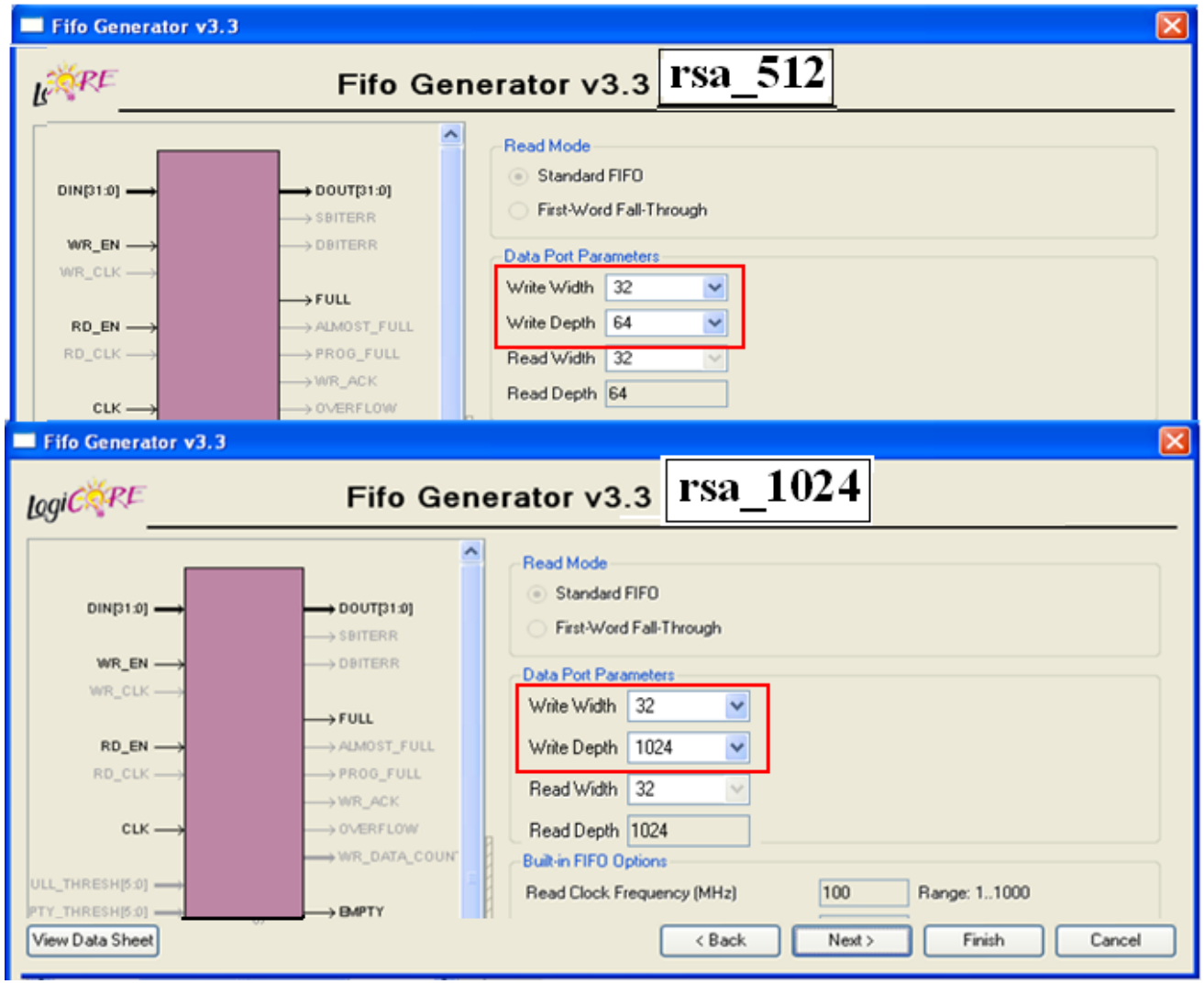


Figure V.3. Extension de la Fifo: res_out_fifo, de rsa_512 vers rsa_1024 [36]

V.3.2.2.3. fifo_512_bram [Annexe B/C]

Cette Fifo est de largeur de 16 bits et de profondeur de 64 dans rsa_512, elle sera remplacée par une Fifo de largeur de 16 bits et d'une profondeur cette fois-ci de 1024, ce qui permettra largement de stocker entièrement les 64 paquets de 16 bits . Comme le montre la Figure V.4

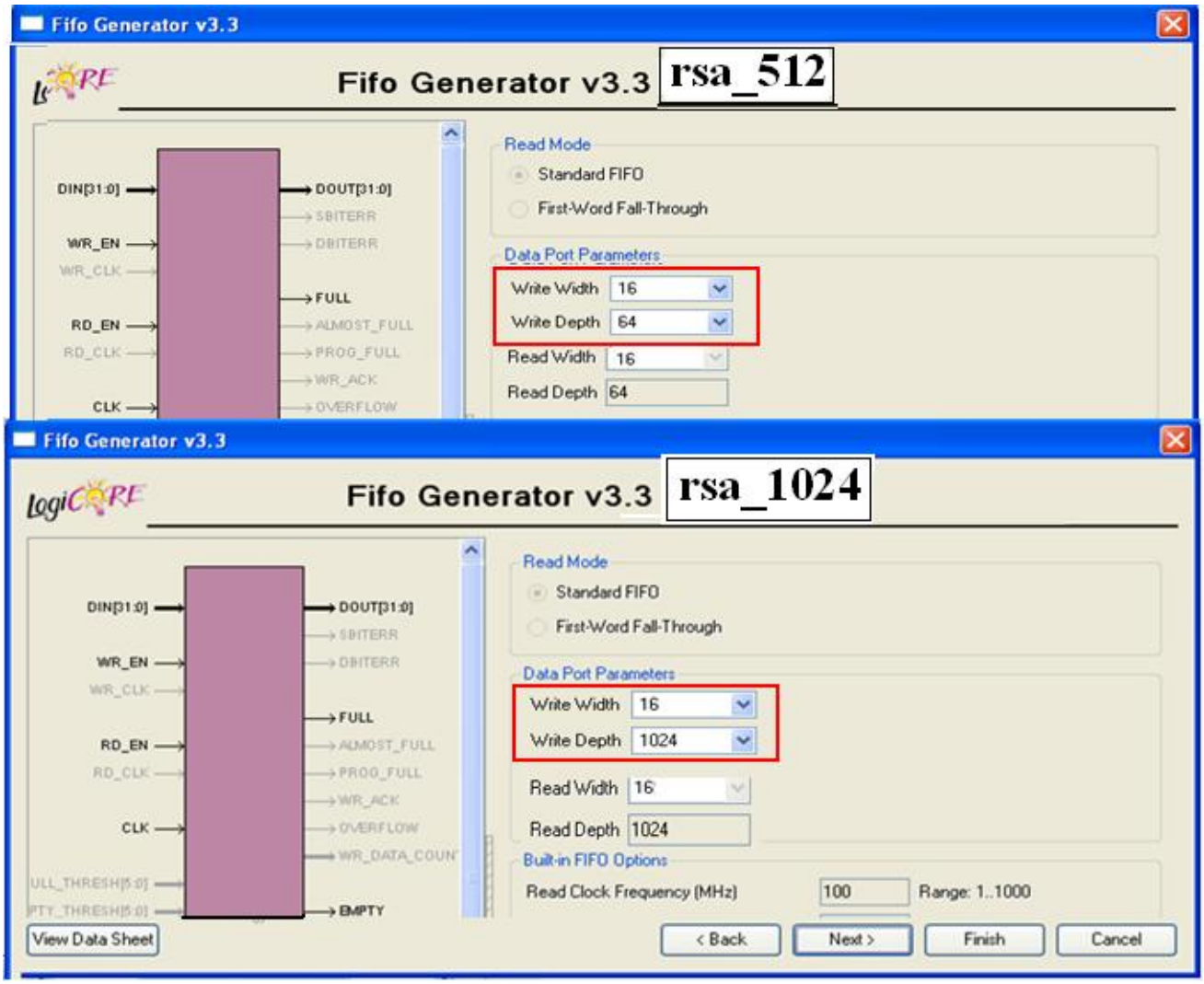


Figure V.4. Extension de la Fifo : fifo_512_bram de rsa_512 vers rsa_1024 [36]

V.3.2.2.4. fifo_256_feedback [Annexe B/C]

Cette Fifo est de largeur de 49 bits et de profondeur de 32 bits dans rsa_512, elle sera remplacée par une fifo de largeur de 49 bits et d'une profondeur cette fois-ci de 1024, ce qui permettra largement de stocker entièrement les 64 paquets de 16 bits . Comme le montre la Figure V.5

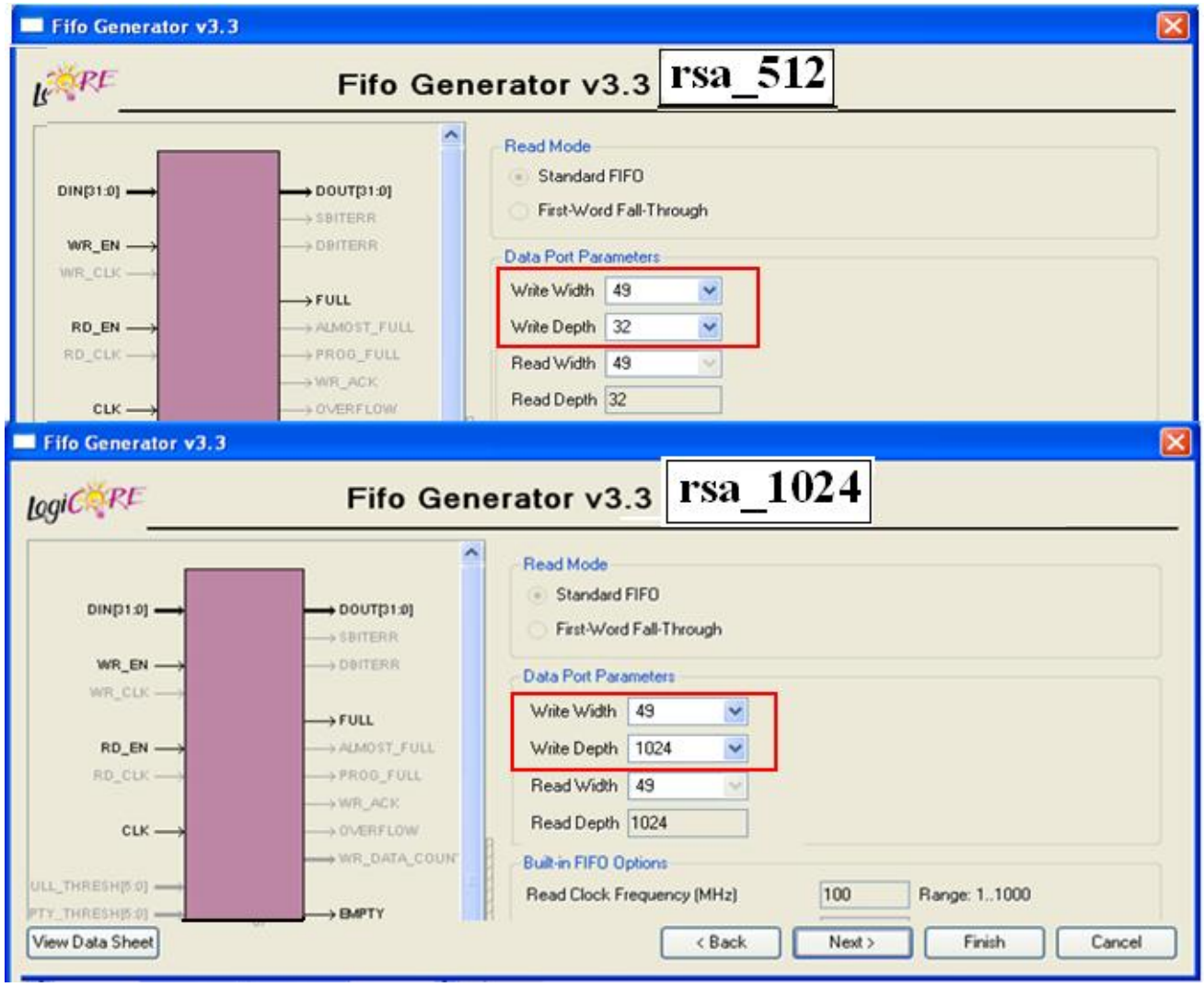


Figure V.5. Extension de la Fifo : fifo_256_feedback de rsa_512 vers rsa_1024 [36]

V.3.3. Simulation fonctionnelle des architecture rsa_512/ rsa_1024

La simulation fonctionnelle (comportementale) est une simulation qui permet de vérifier le fonctionnement du circuit sans prendre en compte les détails concernant le temps (timimg), elle n'a aucune relation avec le circuit cible FPGA, car elle n'est pas temporelle.

La simulation fonctionnelle permet de contrôler le comportement d'un design avant l'implémentation dans le composant cible. Dans le but de vérifier le bon fonctionnement des deux architectures, rsa_512 et rsa_1024, nous avons procéder à faire la simulation via l'outil Modelsim en utilisant le fichier de test « ***testbench***» qui est décrit en VHDL, et qui contient les opérandes d'entrées représentées par :x, y, m, r_c.., et s en sortie.

Afin de vérifier le fonctionnement correcte de l'IP-Core réalisant l'exponentiation modulaire et pour juger l'exactitude du résultat fourni par ***ModelSim***, nous allons refaire le calcul de l'exponontiation modulaire qui vérifie l'équation mathématique **S=** x^y ***modm*** en éxcécutant l'algorithme rsa sous C, afin de pouvoir comparer les résultats obtenus précédemment.

Et Après une comparaison des deux résultats { Modelsim et compilation sous C }, dans le cas ou ils sont bien identiques, ceci confirme que la description de notre l'architecture est valable. Le flot de vérification est montré sur la figure V.6.

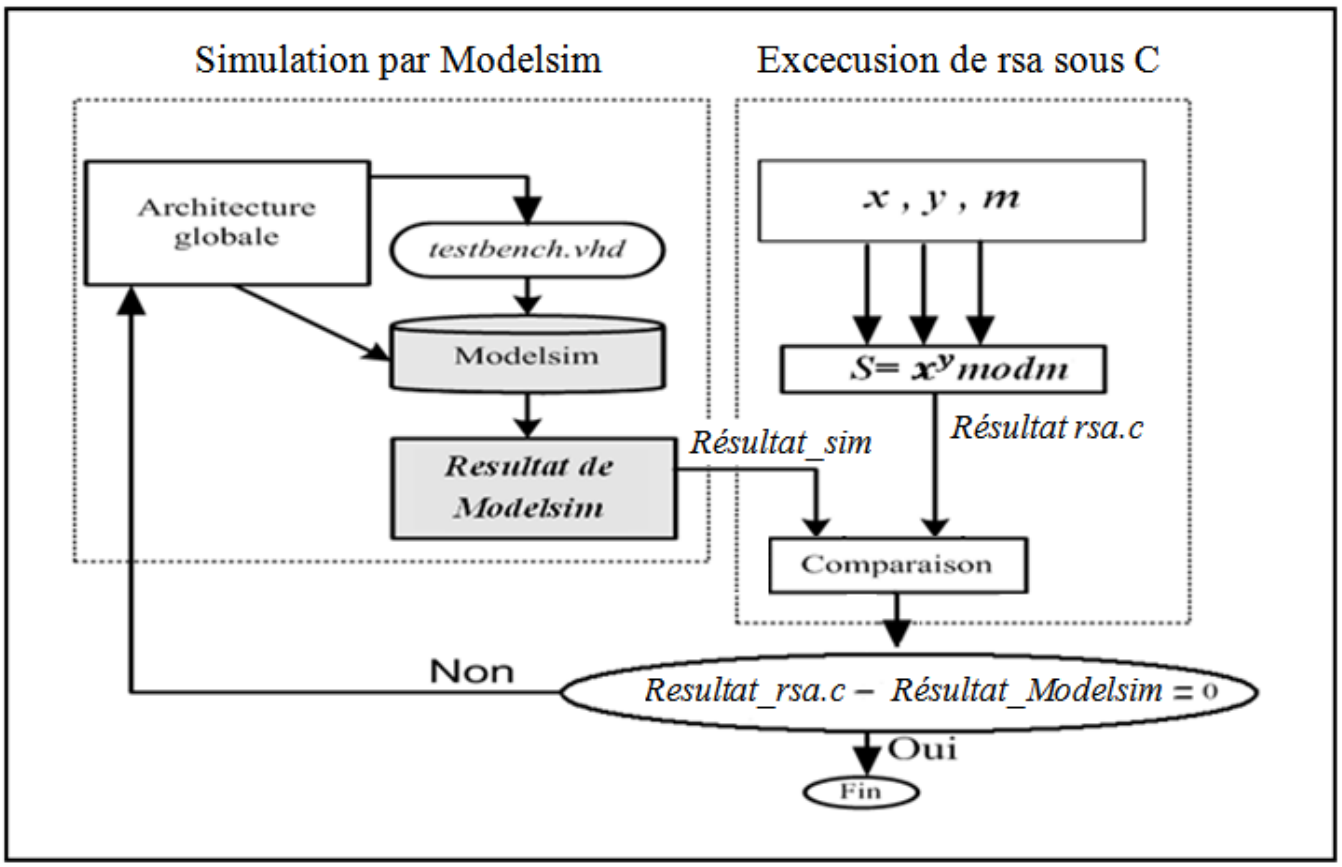


Figure .V.6. Organigramme de vérification de la Simulation fonctionnelle : Modelsim/rsa.c

Dans ce qui suit, nous allons évaluer les résultats de la simulation fonctionnelle des deux architectures.

V.3.3.1. Simulation fonctionnelle de l'architecture rsa_512 [Annexe B] [35]

La simulation fonctionnelle se fait via l'outil Modelsim en introduisant les valeurs { x, y, m, r_c, n_c } dans le fichier ***testbench,*** rsa_512. Ces valeurs d'entrées sont de taille 512 bits représentées en hexadécimal.

La possibilité de vérifier tous les vecteurs de test possible manullement est casiment impossiblesurtout pour des vecteurs ayant une taille allant jusqu'à 512 bits ou plus. Pour se faire, un compteur a été mise en oeuvre (programmé en VHDL) de tel sorte à balayer tous les vecteurs de test possibles ayant cette taille de clef [Annexe B/ Compteur_512]. Les combinaisons de test possibles seront jusqu'à 2^{512} test. Pour démontrer la faisabilité du process et l'exactitude du résultat, on s'est limité à deux vecteurs de test choisit d'une manière aléatoires pour les valeurs {x, y, m } afin de vérifier le fonctionnement de L'IP-Core en comparant les résultats obtenus par simulation de Modelsim à ceux enregistrés lors de la compilation de l'algorithme RSA sous C.

test 1

bit_size = x"0200"

X="*06c8061cf6df4a36c6e4d2cd8fc1d62e5a1268f496004e636af98e40df4a36c6e4d2cd8fc1d62e5a1268f496031d8fc1d62e5a1268f496004e636af98e40f3ad*"

Y="*059fed719f8959a468de367f77a33a7536d53b8e4d25ed49ccc89a94cd6899da90415623fb73386e9635034fb65ad5f248445a1c66703f760d64a8271ad342b1*"

m="*08de7066f67be16fcacd05d319b6729cd85fe698c07cec504776146eb7a041d9e3cacbf0fcd86441981c0083eed1f8f1b18393f0b186e47ce1b7b4981417b491*"

Calcule de r_c/ n_c **[Annexe B : Constant_gen.c]**

Le r_c et n_c dépendent uniquement du modulo ''m'', ils sont utilisés pour accélérer l'exponentiation modulaire, il sont pré-calculés suivant les équations suivantes :

r_c = r^2 mod m, (codé sur 512 bits) , calculé à travers le fichier "constant_gen.c" qui est exécuté en C sous Linux. (Compilé ci-dessous).

n_c = $-m^{-1}$ mod r (codé sur 16 bits) calculé à travers le fichier "constant_gen.c" qui est exécuté en C sous Linux. , comme le montre la figure ci-dessous.

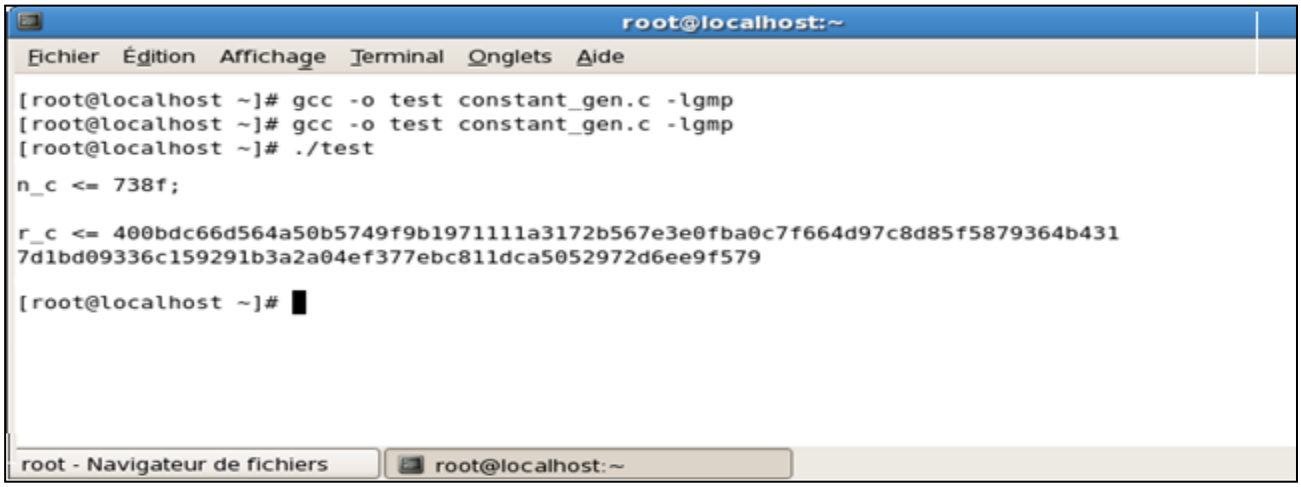


Figure .V.7.1. Calcul de r_c et n_c pour rsa_512 sous Linux

Pour cela, on déduit les valeurs de n_c et r_c :

n_c= "738f"
r_c="400bdc66d564a50b5749f9b1971111a3172b567e3e0fba0c7f664d97c8d85f5879364b4317d1bd09336c159291b3a2a04ef377ebc811dca5052972d6ee9f579"

V.3.3.1.1. Résultats obtenues par le Simulateur Modelsim

L'introduction des valeurs d'entrées sous forme de 32 paquets de 16 bits a été faite après 20 top d'horloge de tel sorte qu'un top d'horloge =1ns pour la simulation fonctionnelle par Modelsim, comme le montre la figure V.8

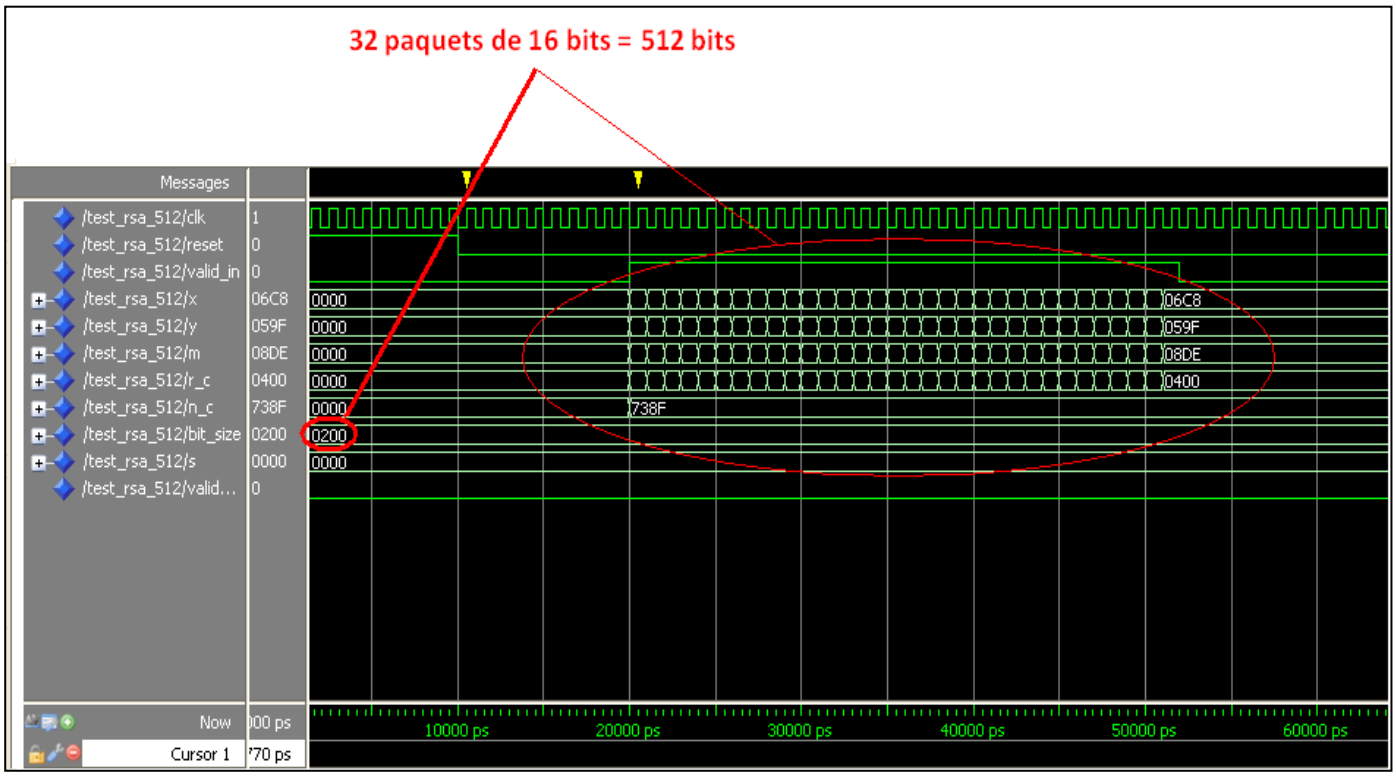


Figure .V.8.1. Les 32 paquets de 16 bits de l'IP rsa_512

Cette simulation de l'exponentiation modulaire a été réalisé pendant une durée de 200.000 ns, le résultat s a été enregistré plus précisémment à **195295, 4 ns**, comme le montre la figure V.10.1

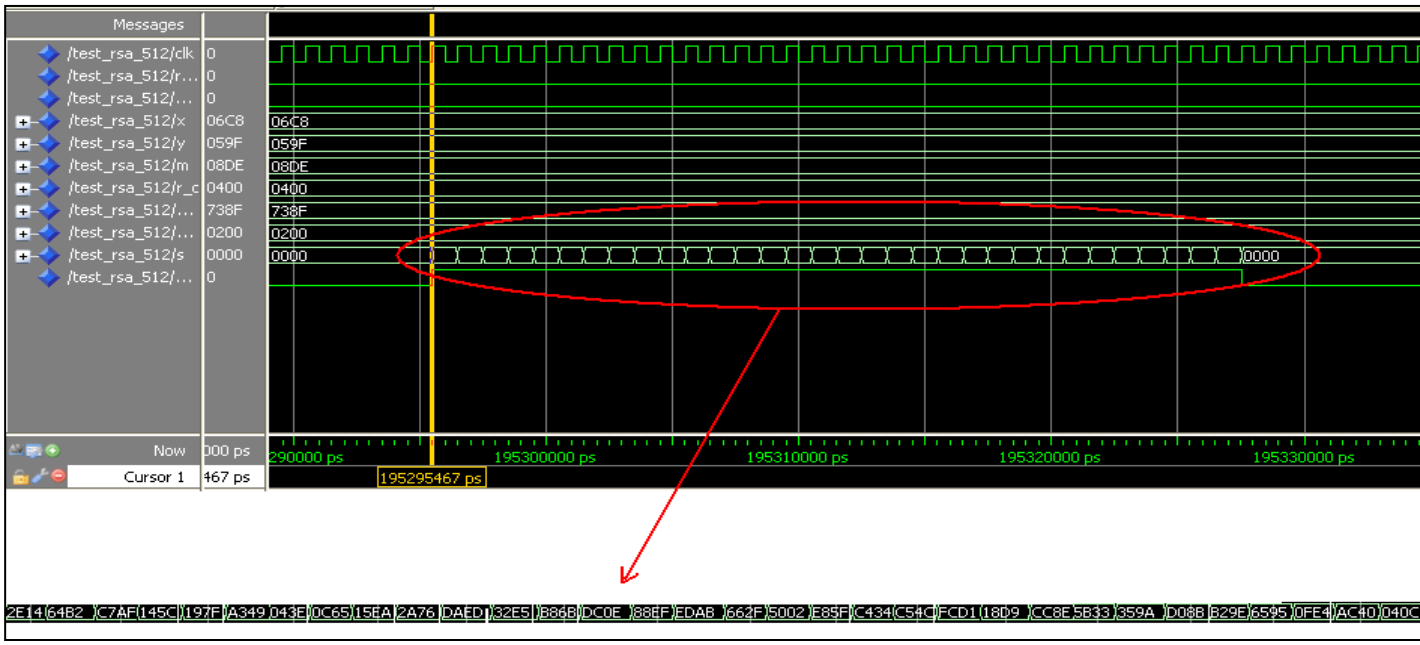


Figure .V.9.1. Le temps enregistré pour la simulation fonctionnelle de l'IP rsa_512

Le premier nombre qui apparait dans la sortie S est : **2E14**, après **64B2** et **C7AF,** et à la fin **: 040C,** le train de bit de S est affiché sur le Simulateur Modelsim ; du mot le moins significatif au mot le plus significatif, donc le résultat S doit être présenté de la manière suivante :

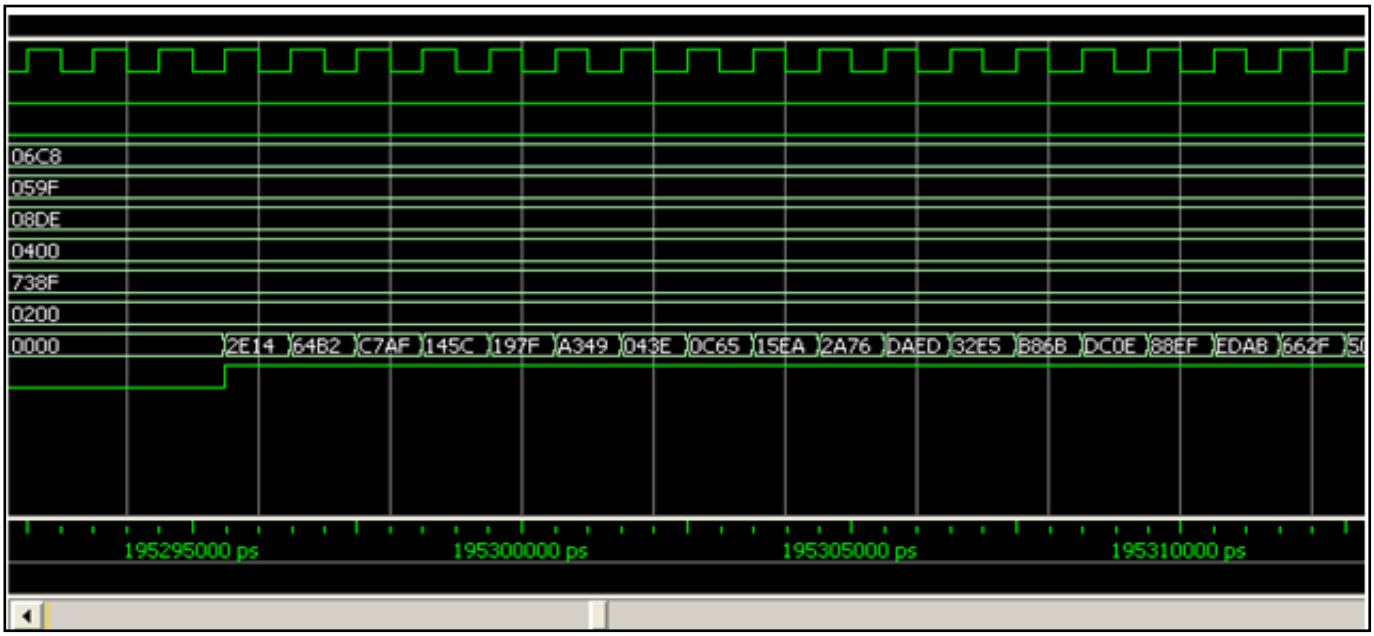


Figure .V.10.1. La sortie S obtenue pour la simulation fonctionnelle de l'IP rsa_512

S="040CAC400FE46595B29ED08B359A5B33CC8E18D9FCD1C54CC434E85F5002662FEDAB88E FDC0EB86B32E5DAED2A7615EA0C65043EA349197F145CC7AF64B22E14 "

V.3.3.1.2. Résultat obtenues par l'algorithme rsa sous C

Afin de vérifier l'éxactitude des résultats obtenus par le Simulateur Modelsim, on procède dans cette section à éxécuter l'algorithme rsa sous C (présenté dans le chapitre III) afin d'en déduire le résultat S à partir des valeurs d'entrées du fichier testbench { x, y, m : de taille 512 bit }et le comparer à celui du Simulateur Modelsim. Dans ce qui suit on met le texte clair dans le ''fichier_a_crypter'' = x, la clé public : y = e, et le modulo m = n, le résultat S (fichier_crypté) sera calculé à travers l'algorithme rsa, suivant l'équation d'exponontiation modulaire : (S = $x^y \ mod \ m$), comme le montre la Figure .V.11.1

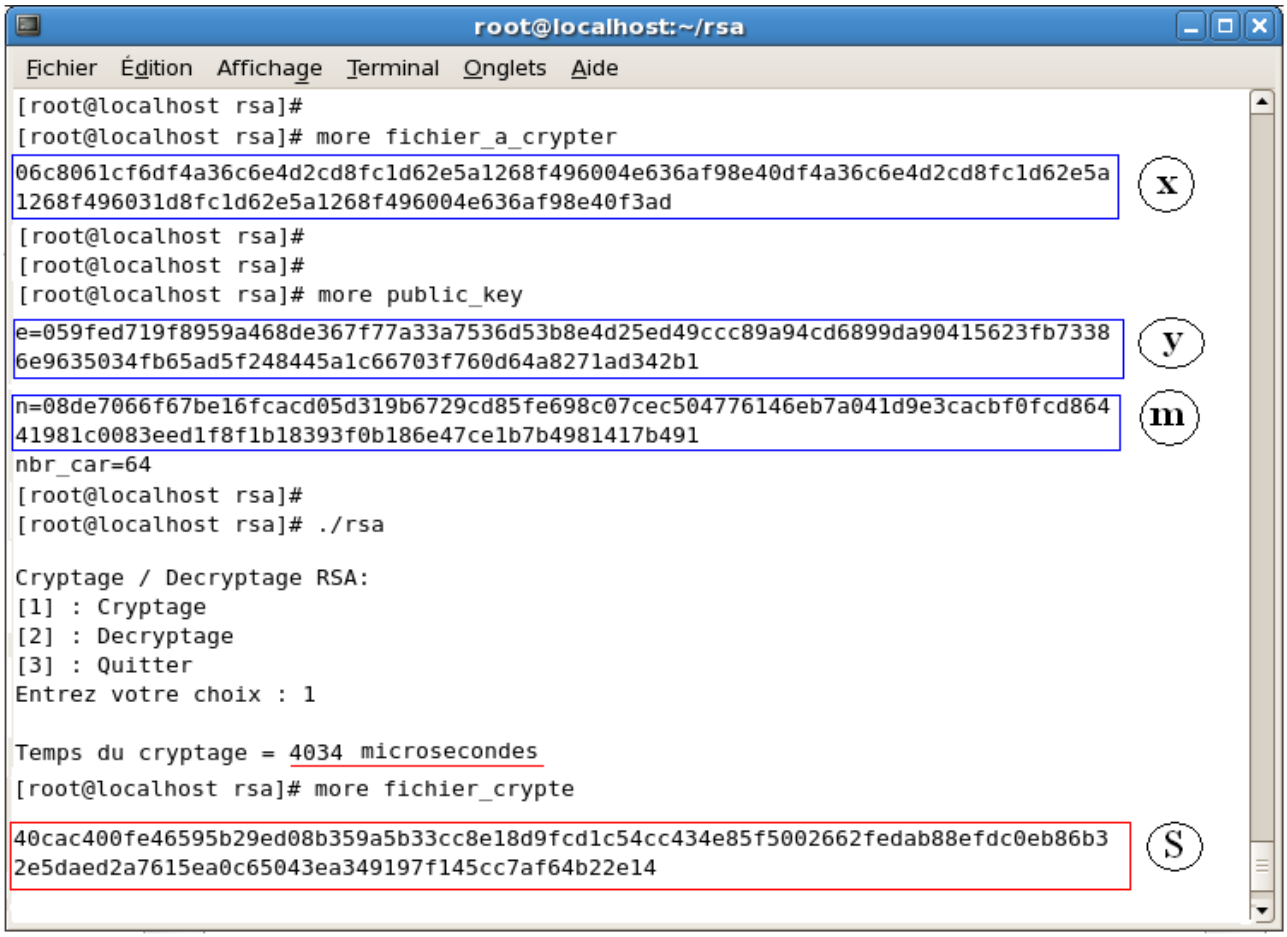


Figure .V.11.1. La sortie S obtenue par l'algorithme rsa sous C (clé=512 bit)

Par comparaison des deux résultats obtenus par Modelsim et l'algorithme rsa compilé sous C, on remarque l'égalité des deux valeurs du résultat S pour le premier test : test 1 ce qui confirme le bon fonctionnement de l'IP-Core rsa_512. Sauf que le temps enregistré pour l'opération de cryptage sous C est quasiment supérieure à celui déduit pour la simulation fonctionnelle du Core (**4034 us >> 195295, 4 ns**) c'est l'avantage de l'implémentation hardware de l'algorithme par rapport à son éxécusion sous le software Linux.

Test 2

```
bit_size  = x"0200"
x="b16b2148e9a2f9c6f44bb5c52e3c6c8061cf694145fafdb24402ad1819ea
cedf4a36c6e4d2cd8fc1d62e5a1268f496004e636af98e40f3adcfccb698f4e
80b9f"

y="308474e4fc596cc1c677dca991d07c30a0a2c5085e217143fc0d073df0fa
6d149e4e63f01758791c4b981c3d3db01bdffa253ba3c02c9805f61009d887d
b0319"
```

***m*="3324a37f3bbbaf46018636343 2cb07035952fc858b3104b8cc18081448e64f1cfb5d60c4e05c1f53d37f53d86901f105f87a70d1be83c65f38cf1c2caa6aa7eb"**

Calcule de r_c/ n_c [Annexe B : Constant_gen.c]

Les facteurs r_c et n_c dépendent uniquement du modulo ''m'', ils sont utilisés pour accélérer l'exponentiation modulaire, ils sont pré-calculés suivant les équations suivantes :

r_c = r^2 mod m, (codé sur 512 bits) , calculé à travers le fichier "constant_gen.c" qui est exécuté en C sous Linux. (Compilé ci-dessous).

n_c = $-m^{-1}$ mod r (codé sur 16 bits) calculé a travers le fichier "constant_gen.c" qui est exécuté en C sous Linux. comme le montre la figure ci-dessous :

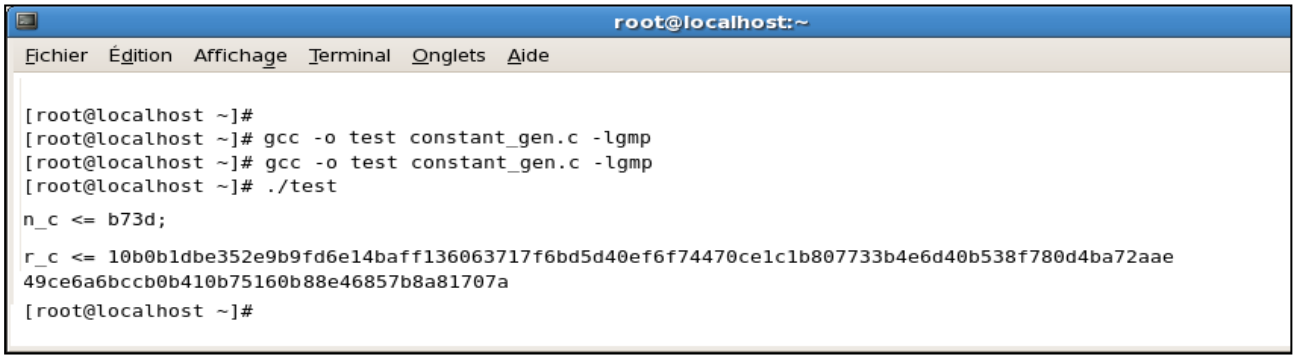


Figure .V.7.2. Calcul de r_c et n_c pour rsa_512 sous Linux

Pour cela, on déduit les valeurs de n_c et r_c :

n_c= "b73d"
r_c=
"10b0b1dbe352e9b9fd6e14baff136063717f6bd5d40ef6f74470ce1c1b807733b4e6d40b538f780d4ba72aae49ce6a6bccb0b410b75160b88e46857b8a81707a"

V.3.3.1.3. Résultat obtenues par le Simulateur Modelsim

L'introduction des valeurs d'entrées sous forme de 32 paquets de 16 bits a été faite après 20 top d'horloge tel que, un top d'horloge =1ns pour la simulation fonctionnelle par Modelsim, comme le montre la figure V.8.2

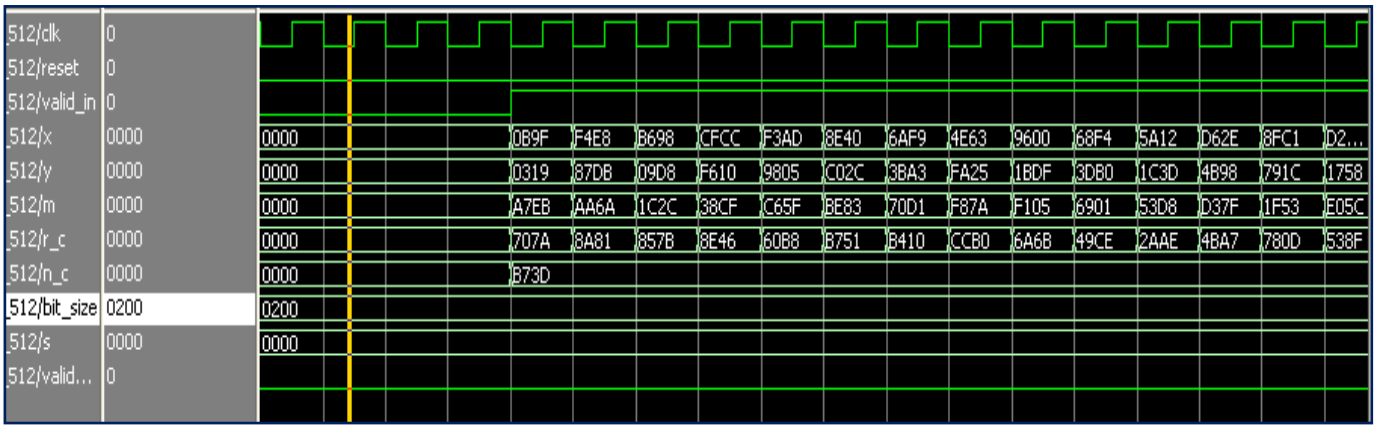


Figure .V.8.2. Les 32 paquets de 16 bits de l'IP rsa_512

Cette simulation de l'exponentiation modulaire a été réalisé pendant une durée de 200.000 ns, le résultat S a été enregistré plus précisément à **195295, 4 ns**, comme le montre la figure V.10.2

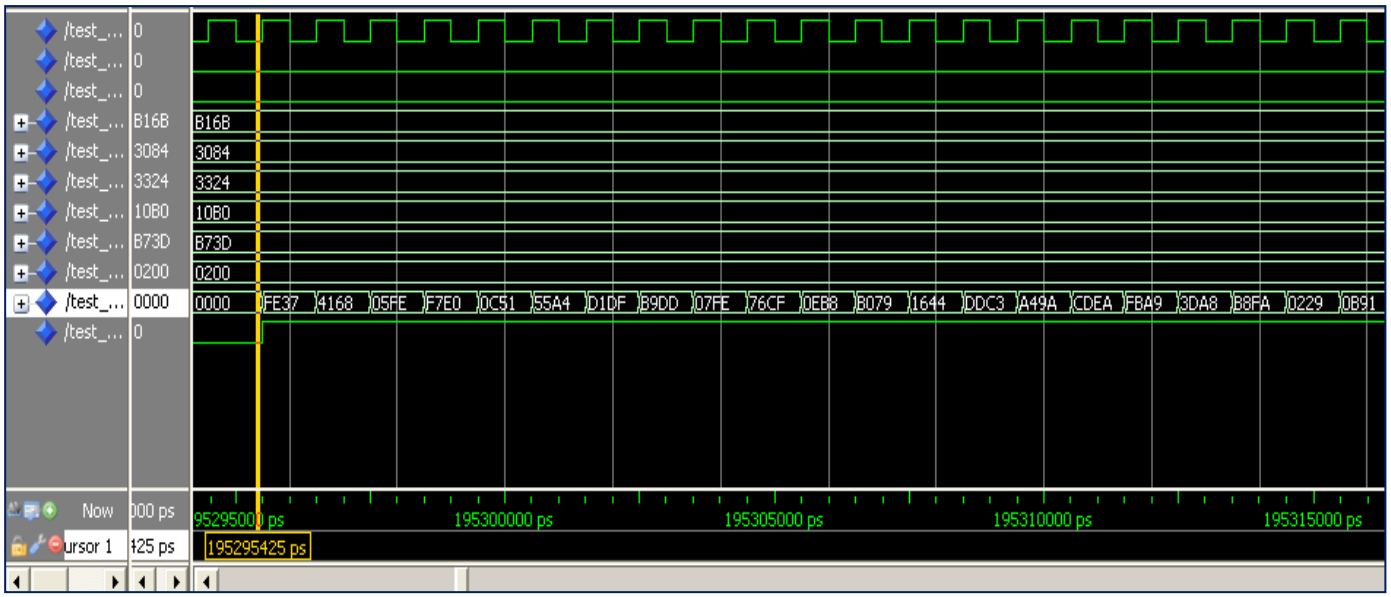


Figure .V.10.2. La sortie S obtenue pour la simulation fonctionnelle de l'IP rsa_512

Le premier nombre qui apparait dans la sortie S est :**FE37**, après **4168** et **05FE,** et à la fin **: 179A,** le train de bit de S est affiché sur le Simulateur Modelsim ; du mot le moins significatif au mot le plus significatif, donc le résultat S est ci-dessous :

S=

179ACA52F56ACDA41E0E261947AE4681B5EE3C9F8BF30B910229B8FA3DA8FBA9CDEAA49A

DDC31644B0790EB876CF07FEB9DDD1DF55A40C51F7E005FE4168FE37

V.3.3.1.4. Résultat obtenues par l'algorithme rsa sous C

Afin de vérifier l'éxactitude des résultats obtenus par le Simulateur Modelsim, on procède dans cette section à éxécuter l'algorithme rsa sous C (présenté dans le chapitre III) afin d'en déduire le résultat S à partir des valeurs d'entrées du fichier testbench { x, y, m : de taille 512 bit }et le comparer à celui du Simulateur Modelsim. Dans ce qui suit on met le texte clair dans le ''fichier_a_crypter'' = x, la clé public : y = e, et le modulo m = n, le résultat S(dans le fichier_ crypté) sera calculé à travers l'algorithme rsa, suivant l'équation d'exponontiation modulaire : (S = $x^y \ mod \ m$), comme le montre la Figure .V.11.2

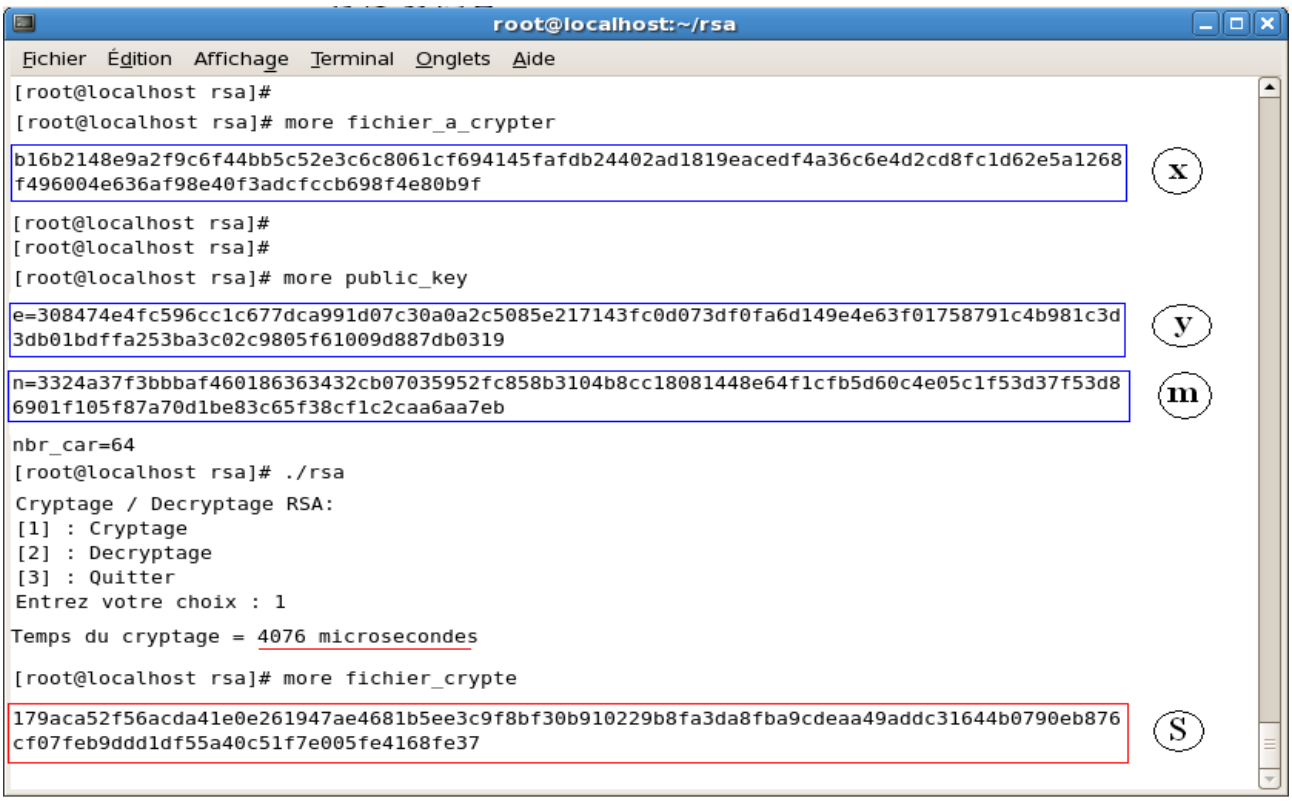


Figure .V.11.2. La sortie S obtenue par l'algorithme rsa sous C (clé=512 bit)

Par comparaison des deux résultats obtenus par Modelsim et l'algorithme rsa compilé sous C, on remarque l'égalité des deux valeurs du résultat S pour le deuxiéme test : test 2 ce qui confirme le bon fonctionnement de l'IP-Core rsa_512. Sauf que le temps enregistré pour

l'opération de cryptage sous C est quasiment supérieure à celui déduit pour la simulation fonctionnelle du Core (**4076 us >> 195295, 4 ns**), c'est l'avantage de l'implémentation hardware de l'algorithme par rapport à son éxécusion sous le software Linux**.**

V.3.3.2. Simulation fonctionnelle de l'architecture rsa_1024 [Annexe C] [51]

Les valeurs d'entrées introduites dans le fichier ''testbench'' rsa_1024, ont été seulement dupliquées par rapport au ceux introduites dans le fichier testbench'' rsa_512''pour le premier test : test1 et introduire des valeurs différentes pour le deuxiéme test : test2, elles sont de tailles 1024 bits représentées en hexadécimal, avec un bit_size **= x"0400":**

Test1

```
bit_size  = x"0400"

x =
"06c8061cf6df4a36c6e4d2cd8fc1d62e5a1268f496004e636af98e40df4a36
c6e4d2cd8fc1d62e5a1268f496031d8fc1d62e5a1268f496004e636af98e40f
3ad06c8061cf6df4a36c6e4d2cd8fc1d62e5a1268f496004e636af98e40df4a
36c6e4d2cd8fc1d62e5a1268f496031d8fc1d62e5a1268f496004e636af98e4
0f3ad"

Y =
"059fed719f8959a468de367f77a33a7536d53b8e4d25ed49ccc89a94cd6899
da90415623fb73386e9635034fb65ad5f248445a1c66703f760d64a8271ad34
2b1059fed719f8959a468de367f77a33a7536d53b8e4d25ed49ccc89a94cd68
99da90415623fb73386e9635034fb65ad5f248445a1c66703f760d64a8271ad
342b1"

M =
"08de7066f67be16fcacd05d319b6729cd85fe698c07cec504776146eb7a041
d9e3cacbf0fcd86441981c0083eed1f8f1b18393f0b186e47ce1b7b4981417b
49108de7066f67be16fcacd05d319b6729cd85fe698c07cec504776146eb7a0
41d9e3cacbf0fcd86441981c0083eed1f8f1b18393f0b186e47ce1b7b498141
7b491"
```

Calcule de r_c/ n_c **[Annexe C : Constant_gen.c]**

Le r_c et n_c dépendent uniquement du modulo ''m'', ils sont utilisés pour accélérer l'exponentiation modulaire, il sont pré-calculés suivant les équation suivantes :

r_c = r^2 mod m, (codé sur 1024 bits) , calculé a travers le fichier "constant_gen.c" qui est exécuté en C sous Linux. (Compilé ci-dessous).

n_c = $-m^{-1}$ mod r (codé sur 16 bits) calculé à travers le fichier "constant_gen.c" qui est exécuté en C sous Linux. , comme le montre la figure ci-dessous

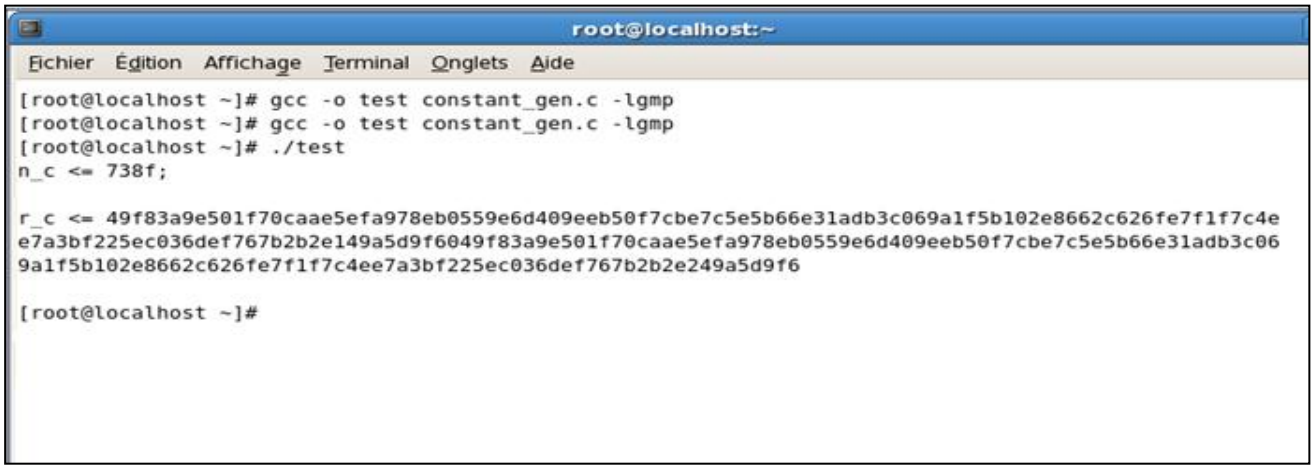


Figure .V.12.1. Calcul de r_c et n_c pour rsa_1024 sous Linux

Pour cela, on déduit les valeurs de n_c et r_c :

n_c= "738f"
r_c="49f83a9e501f70caae5efa978eb0559e6d409eeb50f7cbe7c5e5b66e31
adb3c060a1f5b102e8662c626fe7f1f7c4ee7a3bf225ec036def767b2b2e149
a5d9f6049f83a9e501f70caae5efa978eb0559e6d409eeb50f7cbe7c5e5b66e
31adb3c069a1f5b102e8662c626fe7f1f7c4ee7a3bf225ec036def767b2b2e2
49a5d9f6"

V.3.3.2.1. Résultats obtenues par le Simulateur Modelsim

L'introduction des valeurs d'entrées sous forme de 64 paquets de 16 bits a été faite après 20 top d'horloge de tel sorte qu'un top d'horloge =1 ns pour la simulation fonctionnelle par Modelsim, comme le montre la figure V.13.1

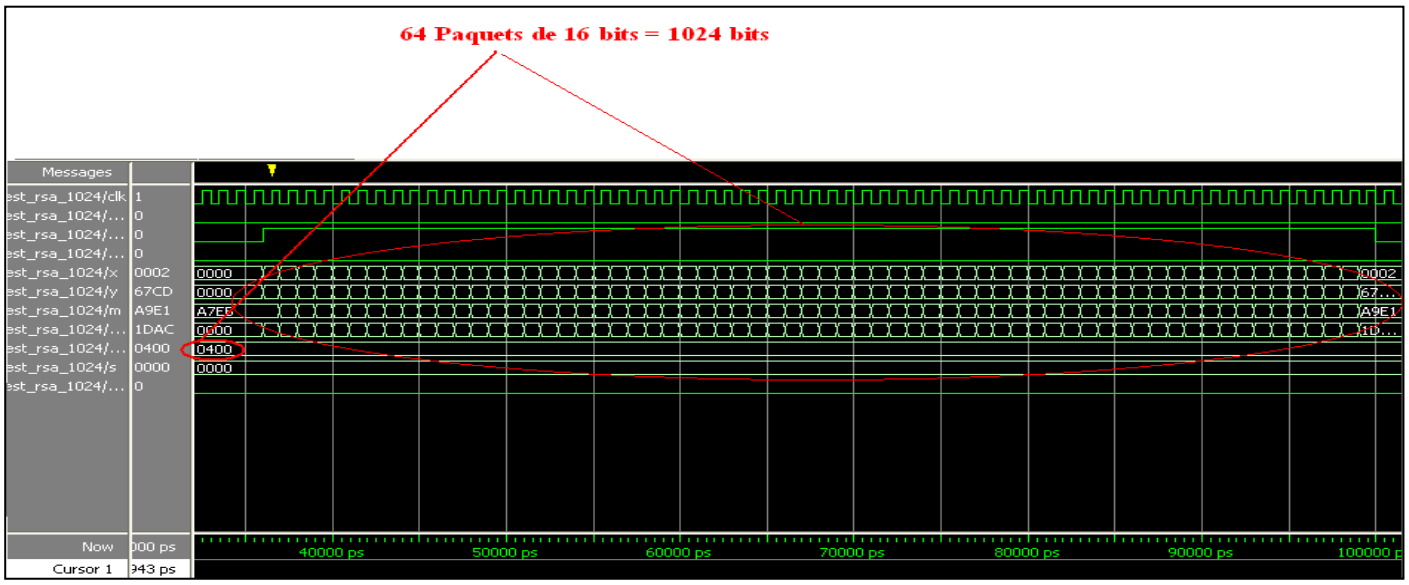


Figure .V.13.1. Les 64 paquets de 16 bits de l'IP rsa_1024

Cette simulation de l'exponentiation modulaire a été réalisé pendant une durée de 800.000 ns, le résultat S a été visualisé plus précisément à **759199, 4 ns**, comme le montre la figure V.14.1

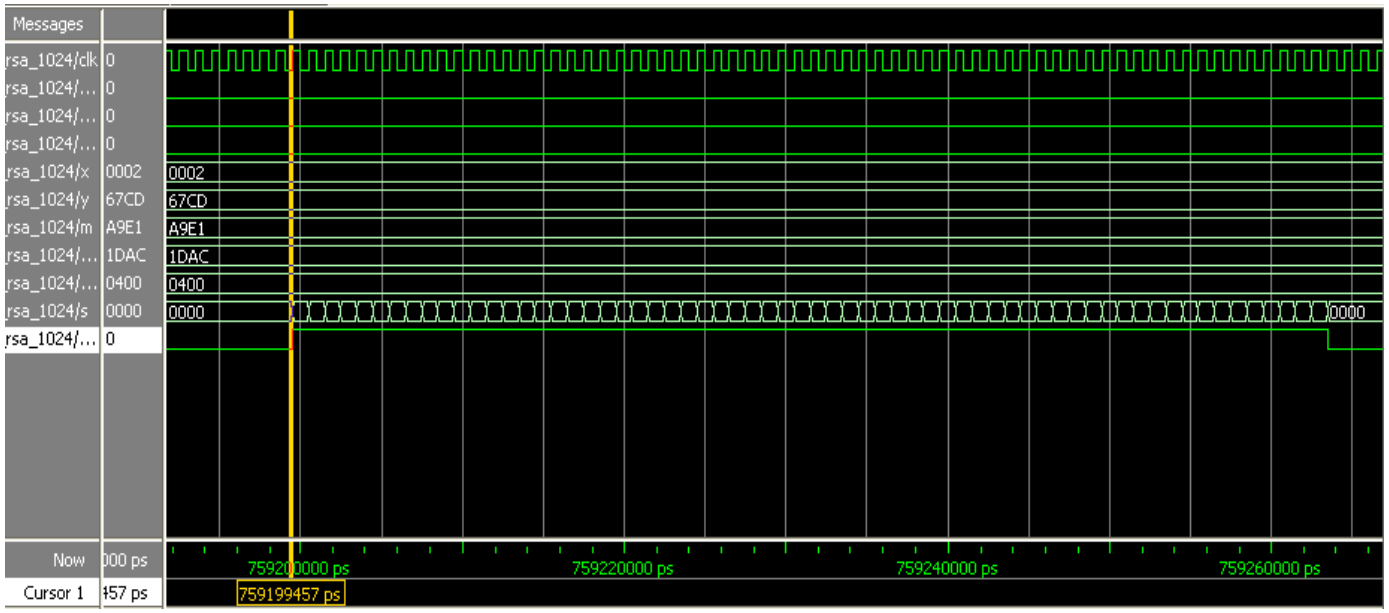


Figure .V.14.1. Le temps de la simulation fonctionnelle de l'IP rsa_1024

Le premier nombre qui apparait dans la sortie S est : **2863**, après **CEC1** et à la fin **: 0772,** le train de bit de S est affiché sur le Simulateur Modelsim ; du mot le moins significatif au mot le plus significatif, donc le résultat S doit être présenté de la manière suivante :

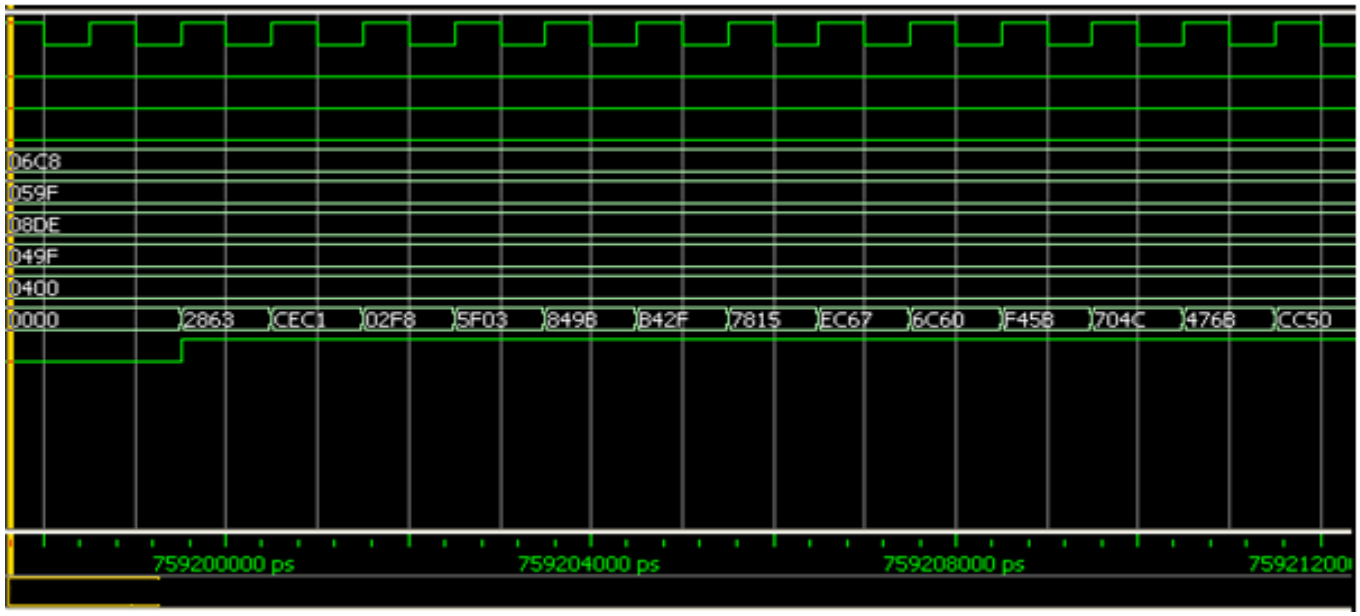


Figure .V.15.1. Le résultat de la simulation fonctionnelle de l'IP rsa_1024

S= "

0772F454FC4B1EB2206A72730BE31D931EAF53EF377A61E45EBD0DA38B458C2C07F4D193C7D9CC50476B704CF45B6C60EC677815B42F849B5F0302F8CEC12863 "

V.3.3.2.2. Résultat obtenues par l'algorithme rsa sous C

Afin de vérifier l'éxactitude des résultats obtenus par le Simulateur Modelsim, on procède dans cette section a éxécuter l'algorithme rsa sous C (présenté dans le chapitre III) afin d'en déduire le résultat S à partir des valeurs d'entrées du fichier testbench { x, y, m : de taille 1024 bit }et le comparer à celui du Simulateur Modelsim. Dans ce qui suit on met le texte clair dans le ''fichier_a_crypter'' = x, la clé public : y = e, et le modulo m = n, le résultat S(dans le fichier_ crypté) sera calculé à travers l'algorithme rsa, suivant l'équation d'exponontiation modulaire : (S = $x^y \ mod \ m$), comme le montre la Figure .V.16.1

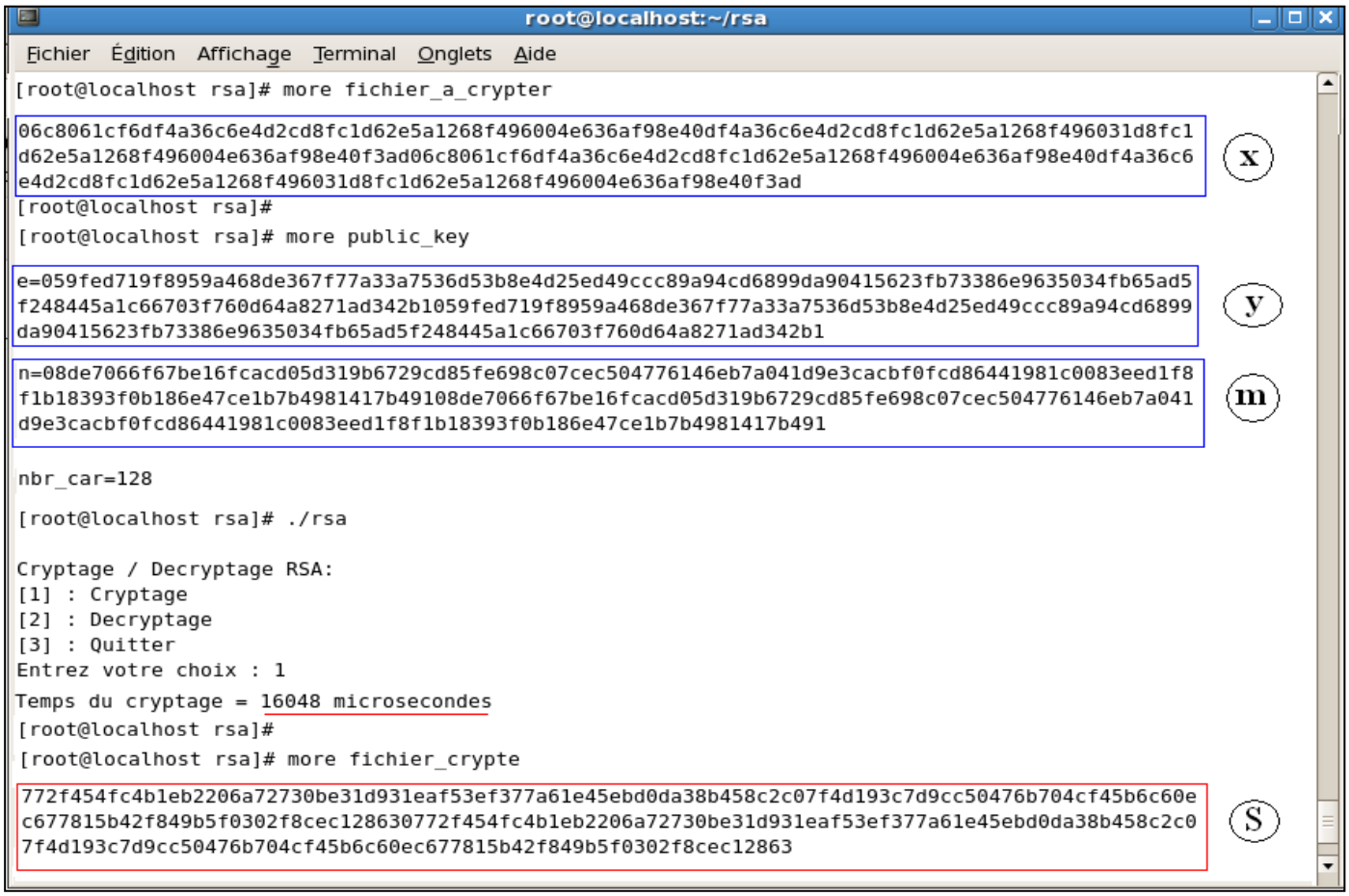


Figure .V.11.1. La sortie S obtenue par l'algorithme rsa sous C (clé=1024 bit)

Par comparaison des deux résultats obtenus par Modelsim et l'algorithme rsa compilé sous C, on remarque l'égalité des deux valeurs du résultat S pour le premier test : test 1 ce qui confirme le bon fonctionnement de l'IP-Core rsa_1024. Sauf que le temps enregistré pour l'opération de cryptage sous C est quasiment supérieure à celui déduit pour la simulation fonctionnelle du Core (**16048 us >> 759199, 4 ns**), c'est l'avantage de l'implémentation hardware de l'algorithme par rapport à son éxécusion sous le software Linux.

Test2

```
bit_size  = x"0400"
x = "
0002257f48fd1f1793b7e5e02306f2d3228f5c95adf5f31566729f132aa1200
9e3fc9b2b475cd6944ef191e3f59545e671e474b555799fe3756099f0449640
38b16b2148e9a2f9c6f44bb5c52e3c6c8061cf694145fafdb24402ad1819eac
edf4a36c6e4d2cd8fc1d62e5a1268f496004e636af98e40f3adcfccb698f4e8
0b9f"
```

Y *="*
67cd484c9a0d8f98c21b65ff22839c6df0a6061dbceda7038894f21c6b0f8b3
5de0e827830cbe7ba6a56ad77c6eb517970790aa0f4fe45e0a9b2f419da8798
d6308474e4fc596cc1c677dca991d07c30a0a2c5085e217143fc0d073df0fa6
d149e4e63f01758791c4b981c3d3db01bdffa253ba3c02c9805f61009d887db
0319"

M *= "*
a9e167983f39d55ff2a093415ea6798985c8355d9a915bfb1d01da197026170
fbda522d035856d7a986614415ccfb7b7083b09c991b81969376df9651e7bd9
a93324a37f3bbbaf460186363432cb07035952fc858b3104b8cc18081448e64
f1cfb5d60c4e05c1f53d37f53d86901f105f87a70d1be83c65f38cf1c2caa6a
a7eb"

Calcule de r_c/ n_c [Annexe C : Constant_gen.c]

Le r_c et n_c dépendent uniquement du modulo ''m'', ils sont utilisés pour accélérer l'exponentiation modulaire, il sont pré-calculés suivant les équation suivantes :

r_c = r^2 mod m, (codé sur 1024 bits) , calculé à travers le fichier "constant_gen.c" qui est exécuté en C sous Linux. (Compilé ci-dessous).

n_c = $-m^{-1}$ mod r (codé sur 16 bits) calculé à travers le fichier "constant_gen.c" qui est exécuté en C sous Linux. , comme le montre la figure ci-dessous

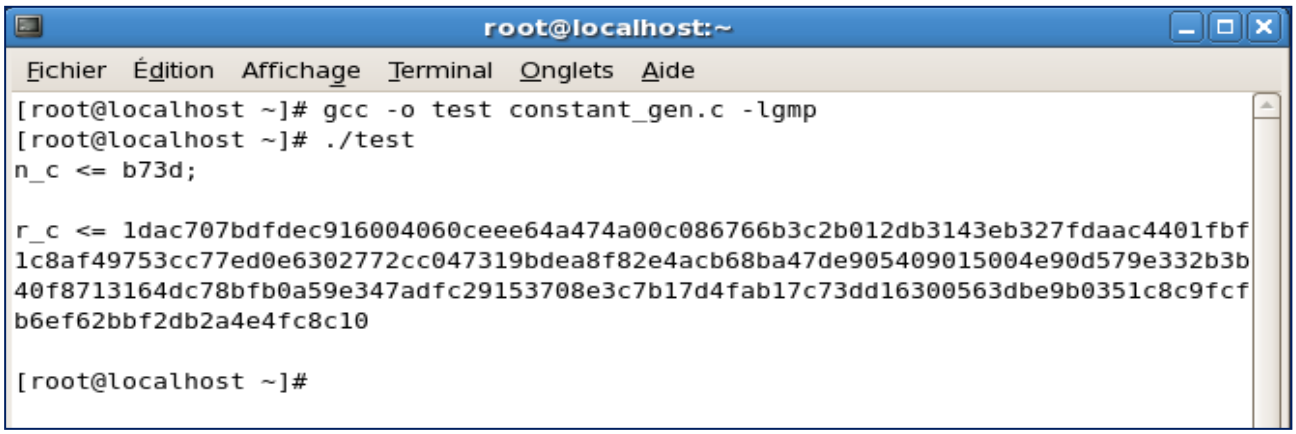


Figure .V.12.2. Calcul de r_c et n_c pour rsa_1024 sous Linux

Pour cela, on déduit les valeurs de n_c et r_c :

n_c= "b73d"
r_c="1dac707bdfdec916004060ceee64a474a00c086766b3c2b012db3143eb
327fdaac4401fbf1c8af49753cc77ed0e6302772cc047319bdea8f82e4acb68
ba47de905409015004e90d579e332b3b40f8713164dc78bfb0a59e347adfc29
153708e3c7b17d4fab17c73dd16300563dbe9b0351c8c9fcfb6ef62bbf2db2a
4e4fc8c10"

V.3.3.2.3. Résultats obtenus par le Simulateur Modelsim

L'introduction des valeurs d'entrées sous forme de 64 paquets de 16 bits a été faite après 20 top d'horloge de tel sorte qu'un top d'horloge =1 ns pour la simulation fonctionnelle par Modelsim, comme le montre la figure V.13.2

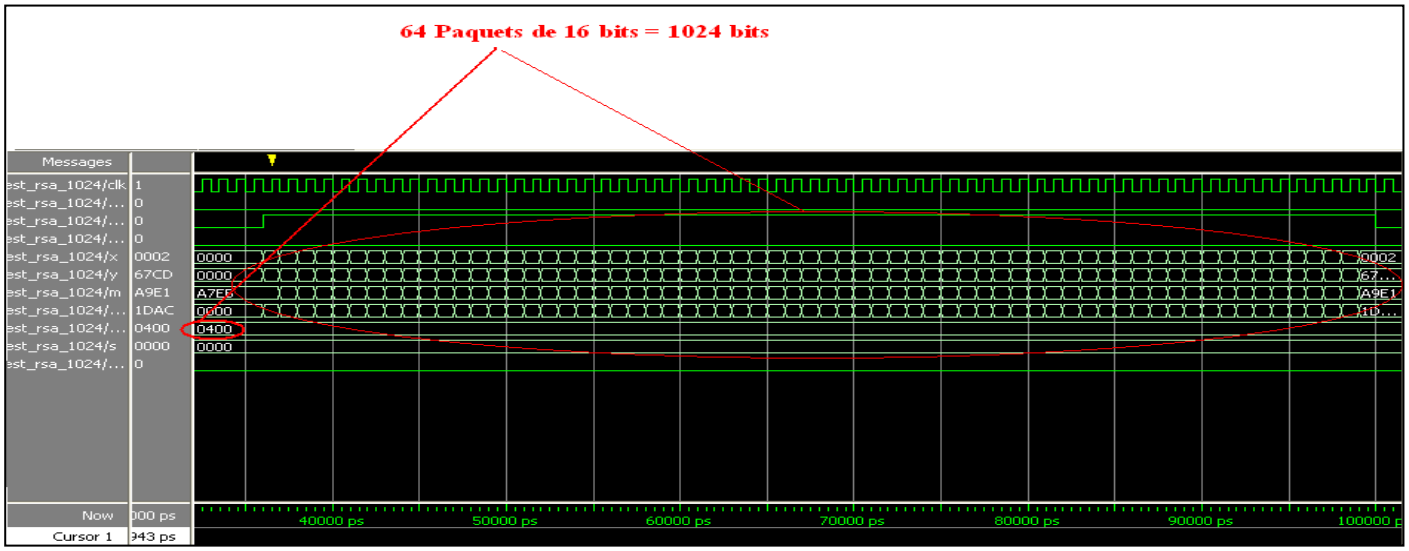


Figure .V.13.2. Les 64 paquets de 16 bits de l'IP rsa_1024

Cette simulation de l'exponentiation modulaire a été réalisé pendant une durée de 800.000 ns, le résultat S a été visualisé plus précisémment à **759199, 4 ns**, comme le montre la figure V.14.2

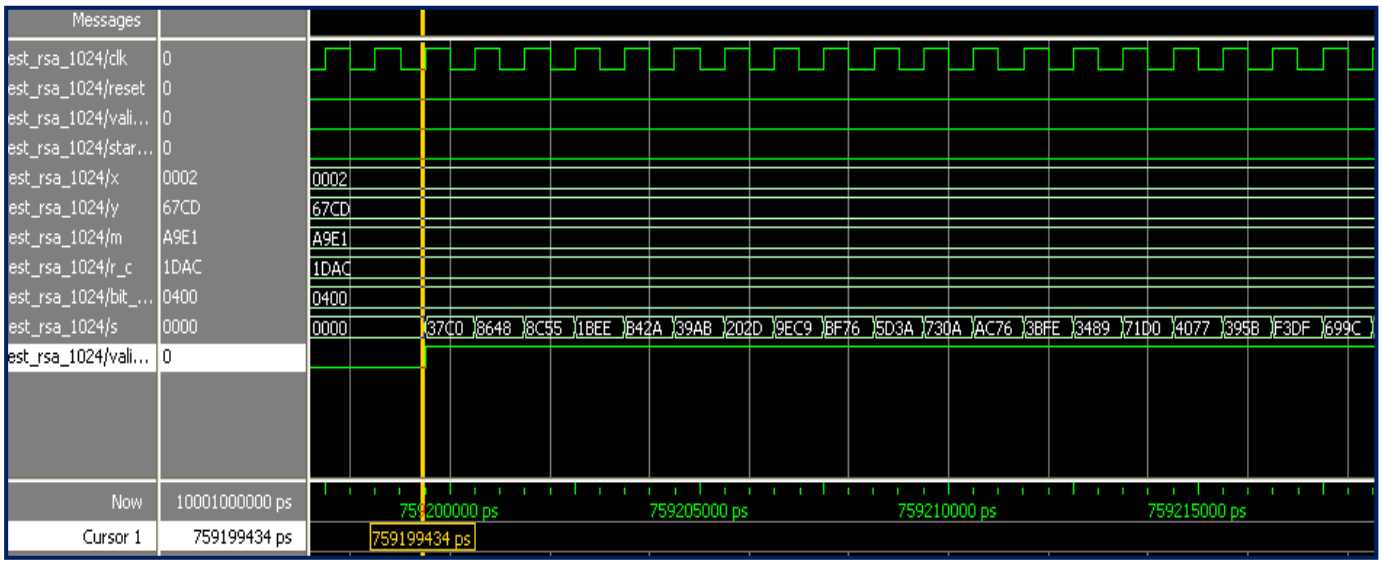


Figure .V.14.2. Le temps de la simulation fonctionnelle de l'IP rsa_1024

Le premier nombre qui apparait dans la sortie S est : **37C0** , après **8648** et à la fin **: A87D,** le train de bit de S est affiché sur le Simulateur Modelsim, du mot le moins significatif au mot le plus significatif, donc le résultat S doit être présenté de la manière suivante :

S= "
A87D9407F6AECB7B936D6C57C2E58DED4F1E2E39AC122F8200732D0A06477312F9D1D36FEF
436DD797232B6314E13EFCBFD5B402CB70C82A83CC3A9C6A96302FD1DA551DA3EC89A934B
CD2728E612690F847159ABC04DADB6876699CF3DF395B407771D034893BFEAC76730A5D3AB
F769EC9202D39ABB42A1BEE8C5586483 7C0 "

V.3.3.2.4. Résultat obtenues par l'algorithme rsa sous C

Afin de vérifier l'éxactitude des résultats obtenus par le Simulateur Modelsim, on procède dans cette section a éxécuter l'algorithme rsa sous C (présenté dans le chapitre III) afin d'en déduire le résultat S à partir des valeurs d'entrées du fichier testbench { x, y, m : de taille 1024 bit }et le comparer à celui du Simulateur Modelsim. Dans ce qui suit on met le texte clair dans le ''fichier_a_crypter'' = x, la clé public : y = e, et le modulo m = n, le résultat S(dans le fichier_ crypté) sera calculé à travers l'algorithme rsa, suivant l'équation d'exponontiation modulaire : (S = $x^y \ mod \ m$), comme le montre la Figure .V.15.2

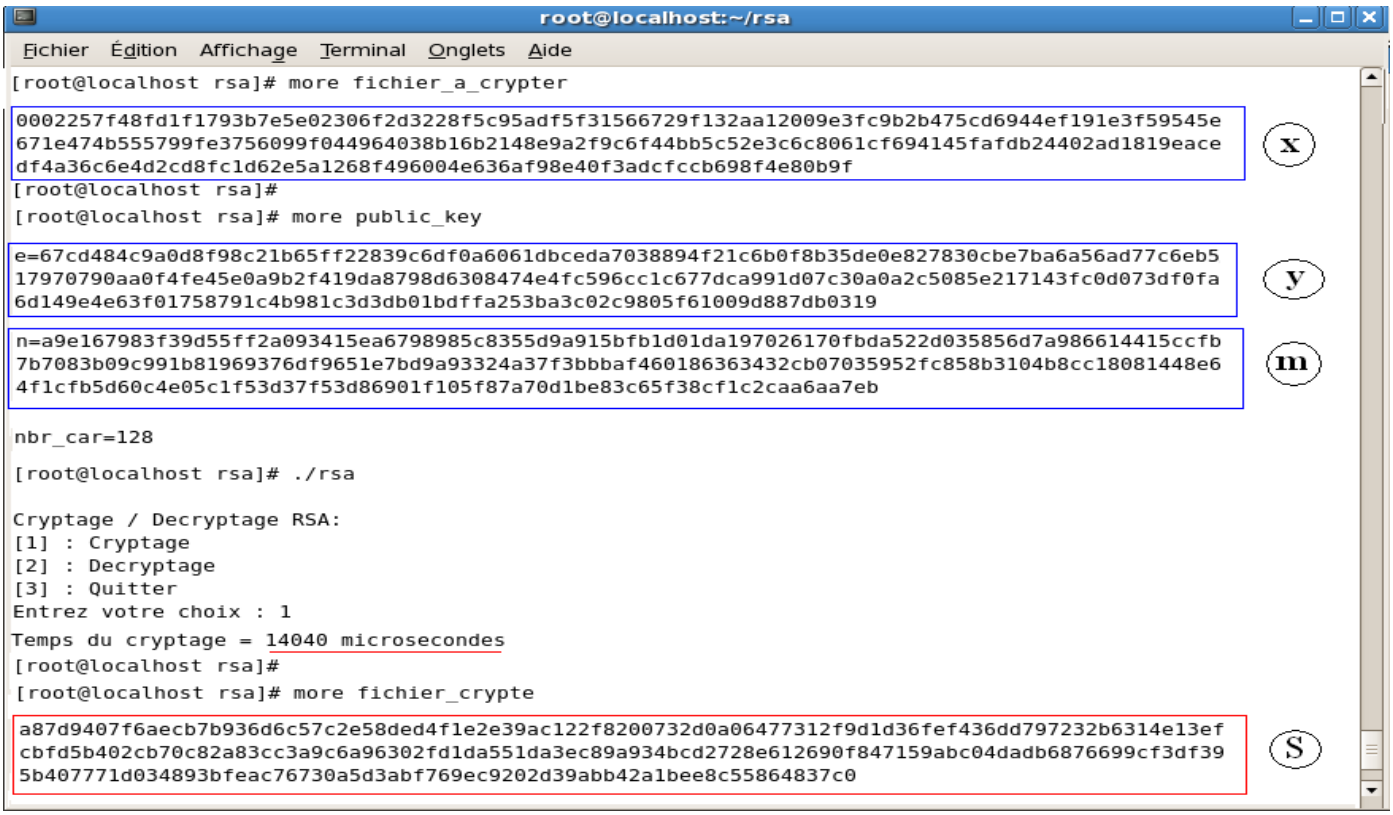


Figure .V.15.2. La sortie S obtenue par l'algorithme rsa sous C (clé=1024 bit)

Par comparaison des deux résultats obtenus par Modelsim et l'algorithme rsa compilé sous C, on remarque l'égalité des deux valeurs du résultat S pour le deuxiéme test : test 2 ce qui confirme le bon fonctionnement de l'IP-Core rsa_1024. Sauf que le temps enregistré pour l'opération de cryptage sous C est quasiment supérieure à celui déduit pour la simulation fonctionnelle du Core (**14040 us >> 759199, 4 ns**), c'est l'avantage de l'implémentation hardware de l'algorithme par rapport à son éxcécusion sous le software Linux**.**

Le tableau ci-dessous résume les résultats de simulation obtenus pour les deux cores :

Tableau V.1. Temps de simulation Fonctionnelle

La taille de la cle de l'IP-Core	Nombre de paquet de 16 bits	Temps de Simulation Fonctionnelle
Rsa_512	*32*	***195295, 4 ns***
Rsa_1024	*64*	***759199, 4 ns***

V.3.4. Synthèse de l'architecture rsa_512/ rsa_1024 [35]

Cette phase transforme la déscrpistion du core programmé en vhdl vers une déscription contenant que des composants de base (portes logiquet , LUT, CLB (porte logique). Elle a pour role de transformer un fichier vhdl en un fichier Netlist (porte logique, …), niveau d'abstraction : Cette phase n'a pas de relation avec FPGA

La synthèse permet de réaliser l'implémentation physique d'un projet. Le synthétiseur a pour rôle de convertir le projet en fonction du type du circuit FPGA cible utilisé, en portes logiques et bascules de base. L'outil ‘’View RTL Schematic’’ permet de visualiser les schémas électroniques équivalents générés par le synthétiseur.

C'est une étape complète de l'architecture (rsa_512/rsa_1024), elle est exécutée par l'outil de Synthèse ***SYNTHESIZE*** et permet d'avoir une estimation du taux d'occupation et du temps d'exécution de l'IP-core dans le Circuit FPGA. Pour ce faire, la carte FPGA X***ilinx Virtex-5 LXT/SXT/FXT,*** a été utilisée dans ce sens.

V.3.4. 1. Les résultats de Synthèse de l'architecture rsa_512 [35]

Le résultat de la synthèse de l'architecture rsa_512 (figure V.17) montre la consommation de notre architecture en termes de ressources physiques du FPGA, en indiquant le nombre de CLB, d'IOB et de bus d'interconnexion ainsi que le pourcentage des ressources consommées.

Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	10,832	58,880	18%	
Number used as Flip Flops	10,832			
Number of Slice LUTs	5,690	58,880	9%	
Number used as logic	5,654	58,880	9%	
Number of route-thrus	583	117,760	1%	
Slice Logic Distribution				
Number of occupied Slices	3,710	14,720	25%	
Number of LUT Flip Flop pairs used	12,747			
Number with an unused Flip Flop	1,915	12,747	15%	
Number with an unused LUT	7,057	12,747	55%	
Number of fully used LUT-FF pairs	3,775	12,747	29%	
Number of unique control sets	55			
IO Utilization				
Number of bonded IOBs	110	640	17%	
Specific Feature Utilization				
Number of BlockRAM/FIFO	5	244	2%	
Total Memory used (KB)	162	8,784	1%	
Number of BUFG/BUFGCTRLs	2	32	6%	
Number used as BUFGs	2			
Number of DSP48Es	48	640	7%	
Total equivalent gate count for design	720,585			
Additional JTAG gate count for IOBs	5,280			

Figure .V.17. Consommation des ressources physique de rsa_512

On note que la consommation de notre architecture est faible. Elle représente 25% de la capacité totale des cellules CLB, 18% du réseau d'interconnexion et 17% du nombre total d'IOB.

Vue les résultats ci-dessus, qui montre la faible consommation des ressources FPGA, on peut se préparer à l'étape de la simulation placement et routage et c'est la qu'on peut confirmer que notre IP-Core peut étre implémenter sur le circuit FPGA,(mais ca reste toujours une étape indépendante de l'FPGA)

V.3.4. 1. Les résultats de Synthèse de l'architecture rsa_1024 [35]

Le résultat de la synthèse de l'architecture rsa_1024 (figure **V.18**) montre la consommation de notre réalisation en termes de ressources physiques du FPGA, en indiquant le nombre de CLB, d'IOB et de bus d'interconnexion ainsi que le pourcentage des ressources consommées.

Device Utilization Summary				
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	11,103	58,880	18%	
Number used as Flip Flops	11,103			
Number of Slice LUTs	6,279	58,880	10%	
Number used as logic	6,204	58,880	10%	
Number used as exclusive route-thru	75			
Number of route-thrus	1,052	117,760	1%	
Slice Logic Distribution				
Number of occupied Slices	3,781	14,720	25%	
Number of LUT Flip Flop pairs used	13,176			
Number with an unused Flip Flop	2,073	13,176	15%	
Number with an unused LUT	6,897	13,176	52%	
Number of fully used LUT-FF pairs	4,206	13,176	31%	
Number of unique control sets	66			
IO Utilization				
Number of bonded IOBs	98	640	15%	
Specific Feature Utilization				
Number of BlockRAM/FIFO	7	244	2%	
Number using BlockRAM only	7			
Total Memory used (KB)	252	8,784	2%	
Number of BUFG/BUFGCTRLs	2	32	6%	
Number used as BUFGs	2			
Number of DSP48Es	50	640	7%	

Figure .V.18. Consommation des ressources physique de rsa_1024

On note que la consommation de notre architecture réalisé est toujours faible, malgré que la taille des opérandes a augmenté. Elle représente 25% de la capacité totale des cellules CLB, 18% du réseau d'interconnexion et 15% du nombre total d'IOB. Vue les résultats ci-dessus, qui montre la faible consommation des ressources FPGA, on peut se préparer à l'étape de la simulation placement et routage et c'est la qu'on peut confirmer que notre IP-Core peut étre implémenter sur le circuit FPGA,(mais ca reste toujours une étape indépendante de l'FPGA)

On remarque que les résultats de Synthèse sont les mêmes, que pour rsa_512, que pour rsa_1024, c'est l'avantage de l'architecture en série (serial). Plus la taille de la clé augmente plus les résultats de synthèse reste presque les memes, ce qui est optimale pour une taille allant jusqu'à .. 4096 bits.

Le tableau ci-dessous résume les résultats obtenus par l'outil Syntesize de l'IP-core rsa_512 et rsa_1024, au sein de la carte ***Virtex-5 LXT/SXT/FXT***:

Tableau V.2. Résultats de la synthèse et simulation fonctionnelle

Bit_size	*Nombre de Cycles par opération de cryptage*	*Temps de simulation (ns)*	*Surface consommé dans les slices*
512	***200.000***	***195295.4***	***25%***
1024	***800 000***	***759199, 4***	***25%***

V.3.5. L'implémentation de rsa_512 /rsa_1024 sur le Circuit FPGA

Une fois que l'on a vérifié la validité du fonctionnement du projet, on peut l'implémenter maintenant sur le circuit FPGA Virtex -5 xc5vsx95t-3ff1136. C'est la dernière étape dans la conception de notre architecture. Elle se fait par l'outil ***IMPLEMENT DESIGN*** qui permet de créer un rapport Place and Route qui nous donne les informations sur les ressources utilisées (nombre de CLBs, de Slices,…), y compris le routage et le délai d'éxécution.

Cette étape dépends fotement du FPGA, car c'est à ce niveau qu'on fait la projection du Netlist sur les ressources de l'FPGA

L'implémentation du circuit est divisée en quatre sous étapes:

- ✓ La transformation (*mapping*) : regrouper les composantes obtenues lors de la synthèse dans des blocs spécifiques du FPGA.
- ✓ La disposition (*placement*) : choisir des endroits spécifiques sur le FPGA où disposent les blocs utilisés, et choisir les pins du FPGA correspondant aux ports d'entrée et de sortie.
- ✓ Le routage (*routing*) : établir des connexions électriques entre les blocs utilisés.
- ✓ La configuration (*configuration*) : convertir toute cette information en un fichier pouvant être téléchargé sur le FPGA pour le programmer.

V.3.5. 1. Les résultats de l'implémentation de rsa_512 sur FPGA

Par rapport aux résultats de la Synthèse, se sont les résultats **réels** de l'implémentation du core rsa_512 sur le circuit FPGA : Virtex -5 xc5vsx95t-3ff1136.

On note que les résultats de l'implémentation sont faibles par rapport à ceux de la synthèse. Elle représente 18% de la capacité totale des cellules CLB, et 17% du nombre total d'IOB.

```
Device Utilization Summary:

   Number of BUFGs                           2 out of 32      6%
   Number of DSP48Es                        48 out of 640     7%
   Number of External IOBs                 110 out of 640    17%
      Number of LOCed IOBs                   0 out of 110     0%

   Number of RAMB18X2s                       1 out of 244     1%
   Number of RAMB18X2SDPs                    2 out of 244     1%
   Number of RAMB36SDP_EXPs                  2 out of 244     1%
   Number of Slice Registers             10832 out of 58880  18%
      Number used as Flip Flops          10832
      Number used as Latches                 0
      Number used as LatchThrus              0

   Number of Slice LUTS                   5690 out of 58880   9%
   Number of Slice LUT-Flip Flop pairs   12747 out of 58880  21%
```

Figure .V.19. Taux d'occupation de rsa_512 sur FPGA

V.3.5. 2. Les résultats de l'implémentation de rsa_1024 sur FPGA

Par rapport aux résultats de la Synthèse, se sont les résultats **réels** de l'implémentation du core rsa_1024 sur le circuit FPGA : Virtex -5 xc5vsx95t-3ff1136.

On note que les résultats de l'implémentation sont faibles par rapport à ceux de la synthèse. Elle représente 18% de la capacité totale des cellules CLB, et 15% du nombre total d'IOB.

```
Device Utilization Summary:

   Number of BUFGs                          2 out of 32      6%
   Number of DSP48Es                       50 out of 640     7%
   Number of External IOBs                 98 out of 640    15%
      Number of LOCed IOBs                  0 out of 98      0%

   Number of RAMB18X2s                      5 out of 244     2%
   Number of RAMB36_EXPs                    2 out of 244     1%
   Number of Slice Registers            11103 out of 58880  18%
      Number used as Flip Flops         11103
      Number used as Latches                0
      Number used as LatchThrus             0

   Number of Slice LUTS                  6279 out of 58880  10%
   Number of Slice LUT-Flip Flop pairs  13176 out of 58880  22%
```

Figure .V.20. Taux d'occupation de rsa_1024 sur FPGA

Pour les résultats des deux tableaux ci-dessus, on remarque que plus la taille de la clé augumente, plus le nombre de paquets augumente, plus l'architecture interne reste la meme. Le taux d'occupation du circuit au sein de cette architecture est presque le meme, malgré l'augumentation du nombre de paquets à traiter, c'est ce qui illumine l'avantage de l'architecture en série

V.3.6. La fréquence maximale du chemin critique [35]

Le chemin critique est le chemin le plus long empreinté par le signal afin de balayer tout les CLB existant dans le circuit FPGA partant de la bascule source vers la bascule destinataire. On prendras toujours la période minimale qui correspond à la fréquence maximale qui permet de piloter l'architecture de l'IP-Core. Autrement dit, choisir le chemin le plus long est calculé le temps minimale (période minimale) mis à travers tout le circuit FPGA en choisissant le chemin le plus long , cette procédure est appelé '' Critical Path''.

V.3.6.1 La fréquence maximale du core rsa_512 dans le circuit FPGA

Comme le montre la figure ci-dessous, la période minimale est de : **7.388 ns**

Delay: 7.388ns (data path - clock path skew + uncertainty)	
Source:	mon_2/et_last/state_FFd2_1 (FF)
Destination:	mon_2/et_last/mont/pe_0/prod_aj_bi_21 (FF)
Data Path Delay:	7.100ns (Levels of Logic = 3)
Clock Path Skew:	-0.253ns
Source Clock:	clk_BUFGP rising
Destination Clock:	clk_BUFGP rising
Clock Uncertainty:	0.035ns

Figure .V.21. Fréquence maximale du chemin critique de rsa_512

De ce fait, la fréquence maximale qui peut piloter ce circuit est de f_{max} = 1/ T_{min}

f_{max} : *fréquence maximale*

T_{min} : *Période minimale*

f_{max} = 1/ T_{min} = = > f_{max} = 1 / 7.388 ns =1 / 7.388 * 10^{-9} s = 0.136 * 10^{9} Hz = 0.136 * 10^{3} * 10^{6} . f_{max} = ***136 MHz***

V.3.6.2 La fréquence maximale du core rsa_1024 dans le Circuit FPGA [35]

Comme le montre la figure ci-dessous : La période minimale est de **: 6.45 ns**

Delay: 6.465ns (data path - clock path skew + uncertainty)	
Source:	mon_2/g1[5].et_i/state_FFd1 (FF)
Destination:	mon_2/g1[5].et_i/mont/pe_0/prod_aj_bi_3 (FF)
Data Path Delay:	6.360ns (Levels of Logic = 4)
Clock Path Skew:	-0.070ns
Source Clock:	clk_BUFGP rising
Destination Clock:	clk_BUFGP rising
Clock Uncertainty:	0.035ns

Figure .V.22. Fréquence maximale du chemin critique de rsa_1024

De ce fait, la fréquence maximale qui peut piloter ce circuit est de f_{max} = 1/ T_{min}

f_{max} : *fréquence maximale*

T_{min} : *Période minimale*

$f_{max} = 1/ T_{min}$ = = > f_{max} = *1 / 6.46 ns =1 / 6.46 * 10^{-9} s= 0.154 * 10^{9} Hz = 0.154 * 10^{3} * 10^{6}*

f_{max} = ***154 MHz***

V.3.7. La simulation temporelle

Contrairement à la simulation fonctionnelle qui fait piloter le circuit à n'importe qu'elle valeur de périodes (qui signifie n'importe quelle fréquence), dans notre cas $T = 1\,ns$. La simulation temporelle (simulation réelle), fait piloter le circuit FPGA à la fréquence maximale déjà calculé précédemment, qui correspond à la période minimale.

Dans notre cas :

- La période minimale pour rsa_512 est de : **7.38 ns**
- La période minimale pour rsa_1024 est de : **6.46 ns**

V.3.7.1. La simulation temporelle pour rsa_512

Apr és avoir passer par l'étape de l'implémentation, on peut passer à la simulation temporelle en choisissant la période minimale à 8 ns (> 7.38 ns) dans le fichier ''**testbench**'' dans la section *' Post-Route Simulation''* comme le montre la figure .V.23

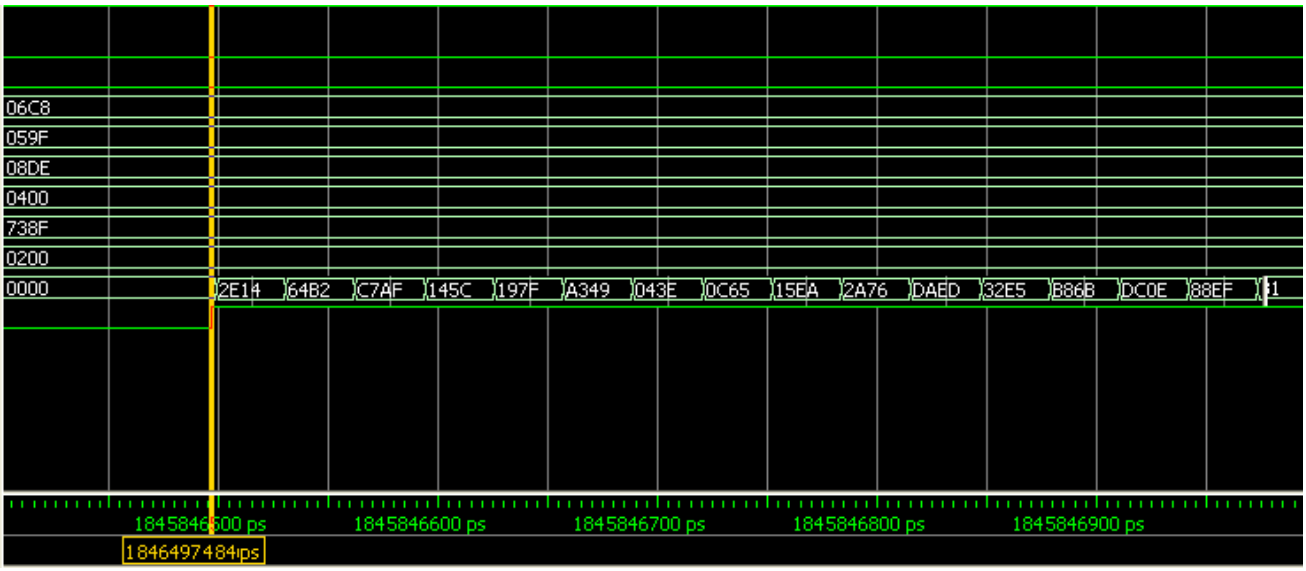


Figure .V.23 Le temps de Simulation temporelle de rsa_512

Le temps de simulation temporelle a été enregistré à **1. 846 us.**

Dans la simulation post& place and route (appelé simulation temporelle), la période a été fixé a 8 ns (> 7,3 ns : période du chemin critique de rsa_512), automatiquement le résultat de cette simulation sera approximativement huit fois celui de la simulation comportementale (qui est d'une période de 1 ns), mais réellement sera un peu plus élevé en y ajoutant le délai de différentes interconnexions et portes logiques.

Le temps de la simulation temporelle peut être estimé aproximativement en multipliant la période minimale (8 ns) par le temps de la simulation fonctionnelle (195295.4 ns)

= 8 ns * 195295.4 ns = 1 562 360 ns = **1.56 us** (résultat approximative)

Par comparaison avec le temps obtenu de la simulation temporelle : { 1.846 us ~ 1.56 us },

V.3.7.2. La simulation temporelle pour rsa_1024

Aprés avoir passer par l'étape de l'implémentation, on peut passer à la simulation temporelle en choisissant la période minimale à 7 ns (> 6.46 ns) dans le fichier ''**testbench**'' dans la section *'' Post-Route Simulation''* pour obtenir les résultats affichés dans la figure .V.24

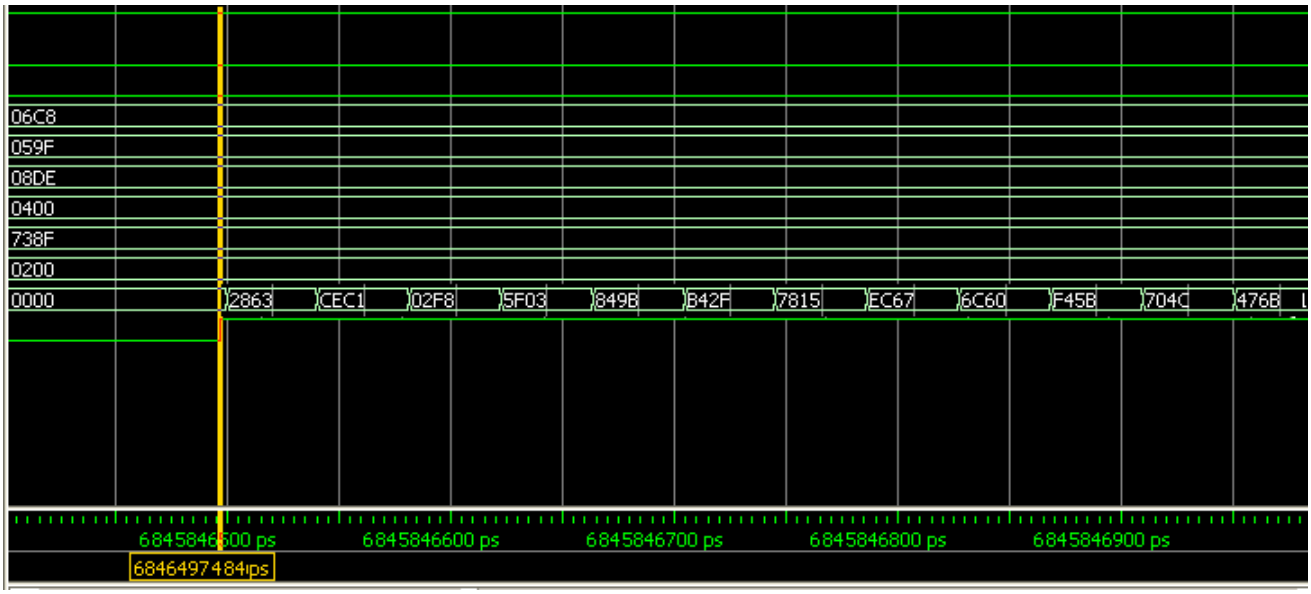


Figure .V.24. Le temps de Simulation temporelle de rsa_1024

Le temps de la simulation temporelle réelle a été enregistré à **6.866 us**

Le temps de simulation temporelle peut être estimé aproximativement en multipliant la période minimale (7 ns) par le temps de la simulation fonctionnelle (759199, 4 ns)

= 7 ns * 759199, 4 ns = 5 314 395.8 ns = **5.314 us**

Par comparaison avec le temps obtenu de la simulation temporelle : { 6.866 us ~ 5.314 us }

Chacune de ces étapes précédantes fait appel à un outil bien spécifique. Nous avons présenté également, les résultats de simulation fonctionnelle, synthèse, implémentation, ainsi que la simulation temporelle obtenue par l'outil Xilinx utilisant le circuit FPGA Virtex -5 xc5vsx95t-3ff1136, comme le montre le tableau V.3

Tableau V.3. Comparaison des résultats de simulation Fonctionnelle /Temporelle, et implementation pour rsa_512 et rsa_1024

Bit_size	*Temps de simulation Fonctionelle (**ns**)*	*Temps de simulation Temporelle (**us**)*	*Période Minimale du circuit (**ns**)*	*Fréquence de maximale du FPGA(**MHz**)*	*Nombre total d'IOB*	*Nombre total de CLB*
512	***195295.4***	***1. 846 us***	***7.38***	***136***	***17%***	***25%***
1024	***759199, 4***	***6.866 us***	***6.46***	***154***	***15%***	***25%***

V.3.8. Conclusion

Dans ce chapitre, nous avons suivi les différentes étapes d'un flot de conception sur circuit FPGA pour arriver à l'implémentation de notre architecture.

Selon les résultats du tableau, par comparaison on remarque que les temps de la simulation temporelle sont plus élevé que ceux de la simulation fonctionnelle, pour la simple raison que les premiers sont réelles et ont été enregistrés après l'implémentation de l'architecture dans le circuit FPGA. Par contre les deuxièmes sont comportementale, autrement dit imaginaire, n'ont aucune relation avec l'implémentation matérielle.

Les résultats de la simulation Fonctionelle / Temporelle (le temps de réponse dans l'architecture Sérial) sont tout à fait logique. La latence de rsa_512 est de 1.84 us et celle de rsa_1024 est de 6.86 us. L' augmentation de cette valeur, signifie que le temps de réponse de l'IP-Core rsa_1024 (64 paquets) est plus élevé que celui de l'IP-Core rsa_512 (32 paquets) - - - > c'est une specifité des critéres de l'architecture sérial.

Les résultats d'implémentation/Synthèse pour rsa_512, et rsa_1024 sont presque les mêmes concernant le taux d'occupation de la surface de l'IP-Core sur FPGA, ce qui met en évidence l'avantage de l'architecture serial qui consomme moins en surface mais beaucoup en temps d'éxcécusion.

Dans le tableau V.3, nous remarquons que les nombres des CLB et IOB consommés sur FPGA sont presque les même pour les deux architectures, car cette extension n'a pas affecté l'architecture interne de l'IP-Core en terme de surface, par contre a permis uniquement de faire augumenter le nombre de mots d'entrées, ce qui a modifié le nombre de registres, les compteurs, et nombre des PEs, pour donner un temps de réponse plus élevé.

La faible consommation en ressources physiques du FPGA, nous incite à étendre encore notre architecture vers le rsa_2048 et rsa_4096. ..,c'est l'avantage de l'architecture en série, plus la taille de la clé augumente, plus le nombre de paquets augumente, plus le taux d'occupation du circuit au sein de cette architecture est presque le meme, malgré l'augumentation du nombre de paquets à traiter.

Dans le chapitre qui suit, nous allons donner un aperçu sur les systèmes sur puce SOC (*system on chip*), les systèmes embarquées EDK / Microblaze.

Chapitre VI

Implémentation de l'IP-Core rsa_512 sur Soc sous EDK Microblaze

VI.1. Introduction [26]

De nos jours, les systèmes électroniques sont de plus en plus complexes, par conséquent, la nécessité d'une plus grande intégration est impérative. Dans notre application, on a adopté la solution SoC, cette dernière permet d'intégrer tout un système sur une seule puce.

Les circuits programmables se révèlent convenables pour de telles applications grâce à leur

grande capacité d'intégration et facilité de programmation. Les circuits FPGA viennent en tête des circuits programmables du fait qu'ils offrent des coûts de réalisation plus faibles et une grande flexibilité devant les technologies concurrentes : les microcontrôleurs et les ASIC.

Notre travail consiste à intégrer l'IP-Core RSA_512 dans une solution SoC en utilisant l'environnement de développement EDK fournit par Xilinx. Dans ce qui suit, on va détailler nos outils de développement et la démarche suivie pour la réalisation de notre application sur les systèmes sur puce SOC (*system on chip*), et une description de la carte utilisée pour la réalisation du projet, ainsi que les logiciels de XILINX utilisés.

VI.2. Les systèmes sur puce [27]

VI.2.1. Définition

Un SoC (*System On Chip*) est un circuit intégré complexe qui intègre la majorité des fonctionnalités d'un produit final, le tout, sur une seule puce. En général un SoC incorpore un ou plusieurs processeurs, de la mémoire sur puce, des fonctions d'accélération, mais aussi des interfaces avec des composants périphériques. La conception du SoC utilise l'approche du Co-design, il permet de combiner, à la fois, des composants logiciels (SW) et matériels (HW).

VI.2.2. Principe [26]

Les *System On Chip* représentent des blocs fonctionnels regroupés sur une même puce pour donner un système complet. L'approche SOC a été crée dans un premier temps pour le développement d'ASIC mais a été étendue pour le développement de FPGA.. Il permet des meilleures performances en termes de consommation, de vitesse et de surface. Mais la fabrication et le test sont des étapes longues et coûteuses

Partant du fait qu'un Soc est généralement constitué par l'assemblage d'un certain nombre de fonctions spécifiques, certaines bien connues, d'autres plus innovantes, le concept de réutilisabilité est basé sur le principe suivant.

Les ressources de conception étant devenues très précieuses, plutôt que d'en gaspiller à refaire ce qui a déjà été fait, il vaut mieux les concentrer sur les parties plus innovantes, celles qui font réellement la valeur du système, et s'appuyer sur la réutilisation pour les autres : c'est l'***IP*** **.**

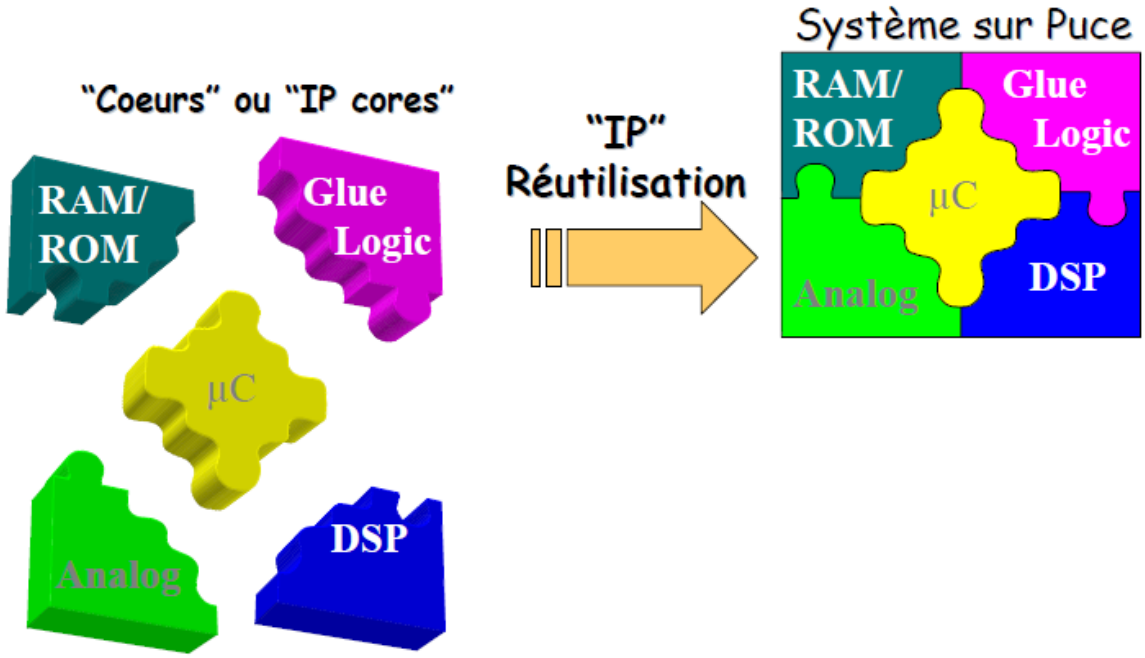


Figure VI.1 : Principe des SOCs [26]

VI.2.3. Flot de conception d'un SOC [27]

On peut distinguer trois différentes étapes lors du développement d'un SOC: le développement de la partie *hardware* ou matérielle qui consiste à choisir les différents éléments qui seront utilisés par le système en question. Le développement de la partie *software* ou logicielle qui se fera en utilisant le langage C/C++ pour décrire le comportement des éléments choisis préalablement. Puis la dernière étape consiste à implémenter la partie *hardware* sur le FPGA et éxécuter la partie *software* au sein de l'architecture implémentée. La figure ci-dessous présente le flot de conception d'un SOC en général.

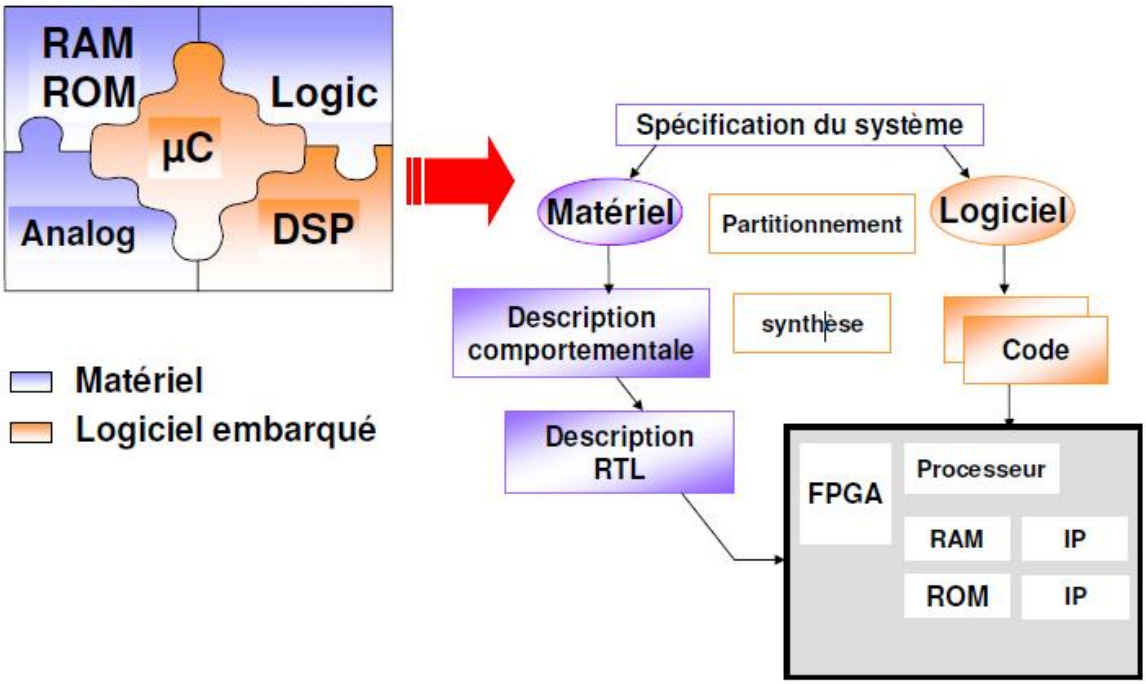


Figure VI.2 : Flot de conception d'un SOC [27]

Un bloc ou un composant IP est un composant virtuel qui peut apparaître sous différentes formes. L'élaboration d'un système SoC peut passer par l'utilisation d'un circuit FPGA.

Cependant, l'utilisation d'un circuit ASIC pour la conception d'un SoC présente un inconvénient dû au manque de flexibilité. En effet, les circuits ASIC réputés pour être des circuits figés, ne permettent pas de faire des modifications sur l'algorithme implémenté ou même sur ses
paramètres. Les FPGA, quant à eux, certes, ne sont pas aussi rapides que les ASIC, mais offrent une très grande flexibilité qui les privilégie pour des applications d'implémentations d'algorithmes. C'est le cas que nous aurons à traiter.

VI.3. Les circuits à logique programmable [27]

Actuellement, on trouve différentes familles de circuits programmables tels que les CPLD (*Complex Logic Programmable Device*) et les FPGA. La différence entre ces deux types de composants est structurelle. Les CPLD sont des composants pour la plupart

reprogrammables électriquement ou à fusibles, peu chers et très rapides (fréquence de fonctionnement élevée) mais avec une capacité fonctionnelle moindre que les FPGA.

Par contre, les FPGAs sont constitués de blocs mémoires vives, entièrement reconfigurables , elles sont structurés en LUT (*Look Up Table*), flip-flop, RAM et l'ensemble dispose d'un vaste système d'interconnexions. Le progrès de la conception des circuits électroniques permet d'avoir des composants toujours plus rapides et à plus haute densité d'intégration, ce qui permet de programmer des applications importantes.

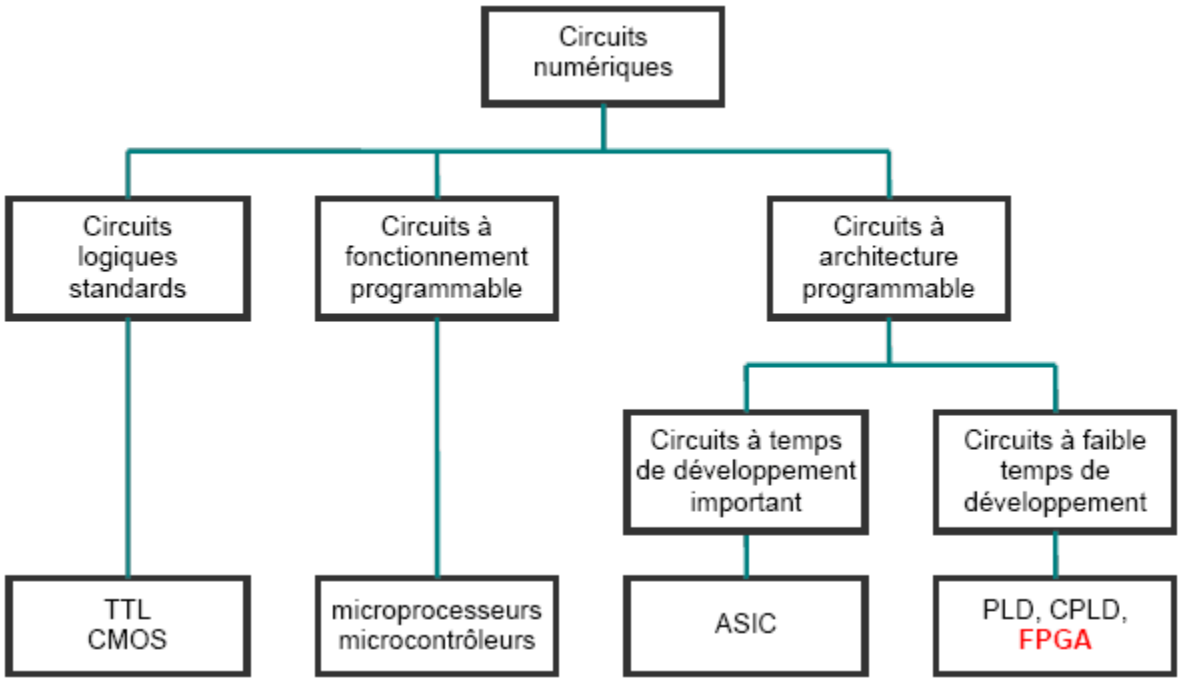


Figure VI.3 : Les différents circuits numériques [27]

VI.3.1. Les circuits FPGA [27]

Contrairement aux ordinateurs par exemple, les FPGA ne possèdent pas un programme résident, il faudrait les configurer à chaque mise sous tension, cette configuration peut se faire par la méthode Master/Slave (le FPGA est connecté à un ordinateur), où depuis une ROM que l'on pourrait lui connecter. Cette configuration se ramène au chargement d'un fichier appelé le bitstream, ce dernier décrit les interconnexions entres les constituants du FPGA (CLB et IOB). Ce fichier peut être créé par deux moyens :

- L'utilisation d'un langage de description matérielle (VHDL). Ce langage permet de synthétiser comment un matériel doit être implémenté. Dans ce cas, on est amené à utiliser l'outil de développement ISE de Xilinx.
- L'utilisation d'un processeur soft, qui offre plus de flexibilité et de rapidité de reconfiguration que les processeurs hard. Ce qui nous a conduit à l'utilisation de l'outil EDK de Xilinx. C'est l'intérêt de notre étude.

VI.3.1.1. L'architecture interne d'un circuit FPGA [29]

Les circuits FPGA possèdent une structure matricielle de deux types de blocs (ou cellules). Des blocs d'entrées/sorties et des blocs logiques programmables. Le passage d'un bloc logique à un autre se fait par un routage programmable. Certains circuits FPGA intègrent également des mémoires RAM, des multiplieurs et même des noyaux de processeurs. Actuellement deux fabricants mondiaux se disputent le marché mondial des FPGA : Xilinx et Altera.

De nombreux autres fabricants, de moindre envergure, proposent également leurs propres produits avec des technologies et des principes organisationnels différents. Dans ce qui suit, on va faire une description de l'architecture utilisée par Xilinx, car c'est sur des circuits Xilinx que nous allons implémenter notre programme.

L'architecture retenue par Xilinx se présente sous forme de deux couches, comme le montre la figure :

- Une couche appelée circuit configurable.
- Une couche réseau mémoire SRAM (Static Read Only Memory).

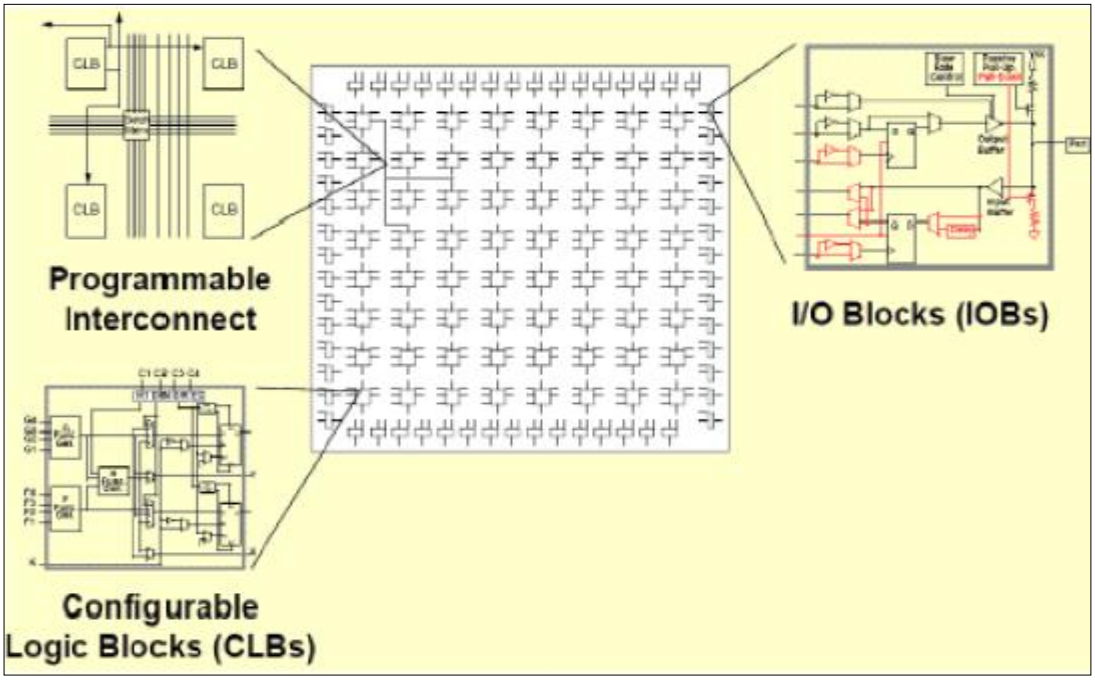


Figure VI.4 : Architecture interne du FPGA [29]

La couche dite (circuit configurable) est constituée d'une matrice de blocs logiques configurables (CLB) permettant de réaliser des fonctions combinatoires et des fonctions séquentielles. Tout autour de ces blocs logiques configurables, nous trouvons des blocs d'entrées/sorties (IOB) dont le rôle est de gérer les entrées-sorties réalisant l'interface avec les modules extérieurs (_Figure.VI.4). La programmation du circuit FPGA, appelé aussi LCA (Logic Cells Arrays), consistera en l'application d'un potentiel adéquat sur la grille de certains transistors à effet de champ servant à interconnecter les éléments des CLB et des IOB, affin de réaliser les fonctions souhaitées et d'assurer la propagation des signaux. Ces potentiels sont tout simplement mémorisés dans le réseau mémoire SRAM.

Réseau mémoire SRAM : la programmation d'un circuit FPGA est volatile, la configuration du circuit est donc mémorisée sur la couche réseau SRAM et stockée dans une ROM externe. Un dispositif interne permet à chaque mise sous tension de charger la SRAM interne à partir de la ROM. Ainsi on conçoit aisément qu'un même circuit puisse être exploité successivement avec des ROM différentes puisque sa programmation interne n'est jamais définitive.

VI.4. Présentation de la carte de développement [28]

VI.4.1. Xilinx

Xilinx est une entreprise américaine de semi-conducteurs. Inventeur du FPGA, Xilinx fait partie des plus grandes entreprises spécialisées dans le développement et la commercialisation de composants logiques programmables, et des services associés tels que les logiciels de CAO électroniques ; blocs de propriété intellectuelle réutilisables et formation. Il s'agit aussi du logiciel de développement.

VI.4.2. Eléments de la carte de développement

La Carte cible utilisée est la : ***Virtex-5 LXT/SXT/FXT.*** Elle consiste en un ensemble de différents périphériques montés autour d'un circuit FPGA (Figure VI.5). Ses principales caractéristiques sont les suivantes :

- ***Virtex4 LXT/SXT/FXT***, dans lequel est placé un processeur PPC405
- Mémoires : Elle supporte plusieurs types de mémoire. Elle contient 64 MB DDR-SDRAM, 8Mb SRAM, 64Mb flash et la 4kb IIC EEPROM;
- Les connecteurs: clavier, souris, microphone, port RS-232, ports USB, PC4 JTAG, port VGA, port Ethernet 10/100/1000 RJ-45, GPIO consiste en boutons et LED.
- Elle dispose aussi d'un port VGA, d'un port série, d'un port USB, d'interrupteurs, de diodes. Comme le montre la figure VI.5

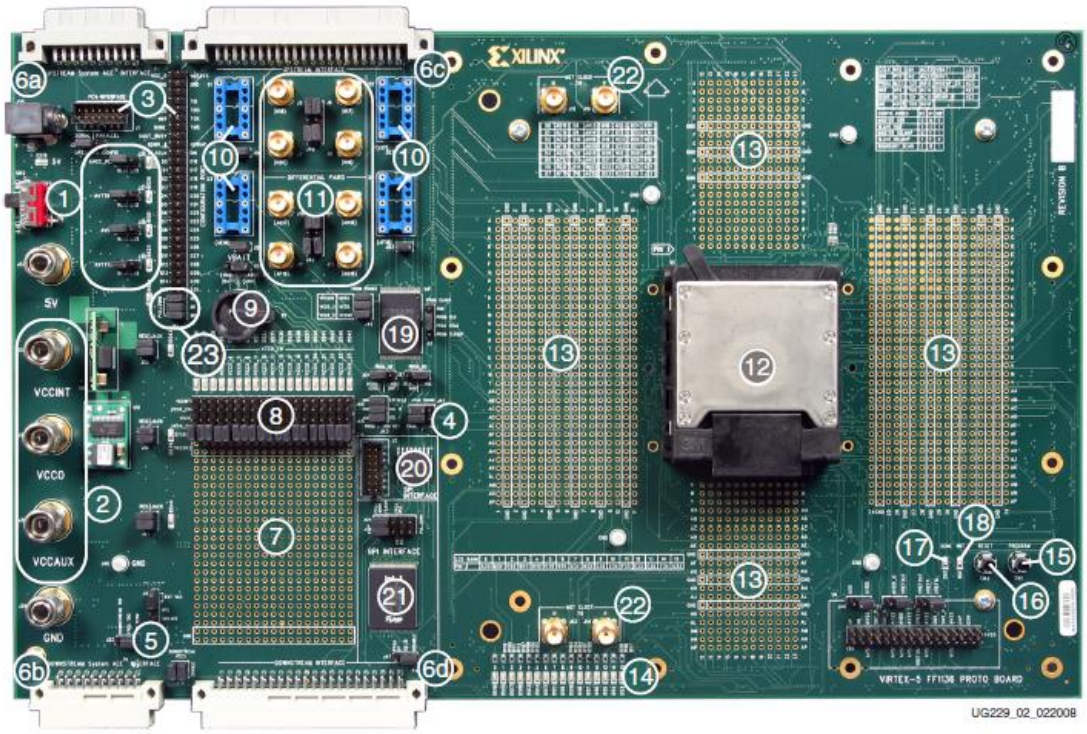


Figure VI .5 : la carte xilinx Virtex-5 LXT/SXT/FXT [28]

Le FPGA Virtex-5 possède 200000 portes logiques assemblables à souhait à l'aide du VHDL. Il dispose d'environ 170 entrées-sorties et fonctionne sous une tension de 1,5V.

Cette carte se programme via le port parallèle de l'ordinateur à l'aide de l'environnement de développement intégré de Xinlinx ISE.

La Virtex_5 inclut un bloc processeur IBM PowerPC 405. Ce processeur fonctionne à des fréquences supérieures à 300MHz. Il est couplé à des blocs BRAM par l'intermédiaire de contrôleurs OCM pour permettre des échanges rapides entre les données entrantes dans le circuit FPGA et celles traitées par le PowerPC. Sa consommation est faible, elle est de l'orde de 0,9 mW/MHz. Il possède 32 registres de 32 bits, un pipeline à 5 étages. Il intègre des unités de calcul avancées tel qu'un multiplieur et un diviseur. Il inclut la technologie de débogage et de traçage des signaux par liaison JTAG.

Les DCM's

Les DCMs sont les contrôleurs d'horloges du FPGA. Ils synchronisent les données extérieures avec un minimum de gigue et génèrent les horloges internes par multiplications ou divisions successives d'une fréquence de référence externe ou interne. Ils minimisent les décalages d'horloge provoqués par les délais de transmissions au sein du circuit.

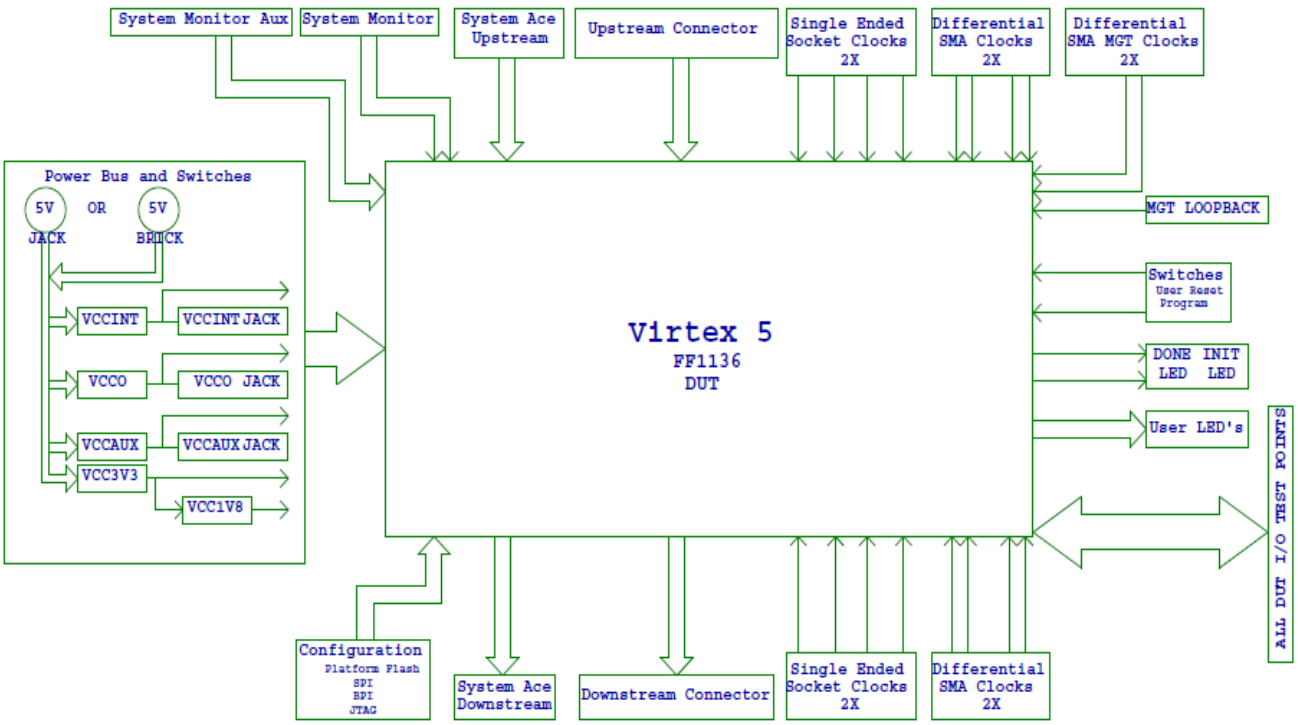


Figure VI .6 : Architecture interne de la carte Virtex 5 [28]

VI.5. L'outil de développement EDK [29]

''Embedded Développement Kit ''. EDK est un environnement de développement fourni par Xilinx destiné au développement d'une application complète embarquée et à son intégration sur un FPGA. EDK donne accès à tous les dimensionnements nécessaires à l'application que l'on souhaite créer.

L'environnement EDK fait appel au concept du Co-design. Il permet le développement hardware en parallèle à un développement software. Il inclut, principalement, la plateforme Xilinx XPS et le kit de développement software SDK « Software Développement Kit ».

VI.5.1. Le logiciel Xilinx Platform Studio (XPS) [52]

Pour la création et la vérification d'un système embarqué, Xilinx propose un ensemble d'outils regroupés dans XPS (Xilinx Platform Studio). La conception d'un système embarqué inclut typiquement les phases suivantes, (voir Figure VI.7)

Les processus supportés par l'outil XPS sont principalement :

- Création du fichier MHS
- Création du fichier MSS
- Réalisation de la matrice de connexion des différents bus
- L'adaptation des drivers, des librairies, et des contrôleurs d'interruptions
- La gestion des fichiers sources

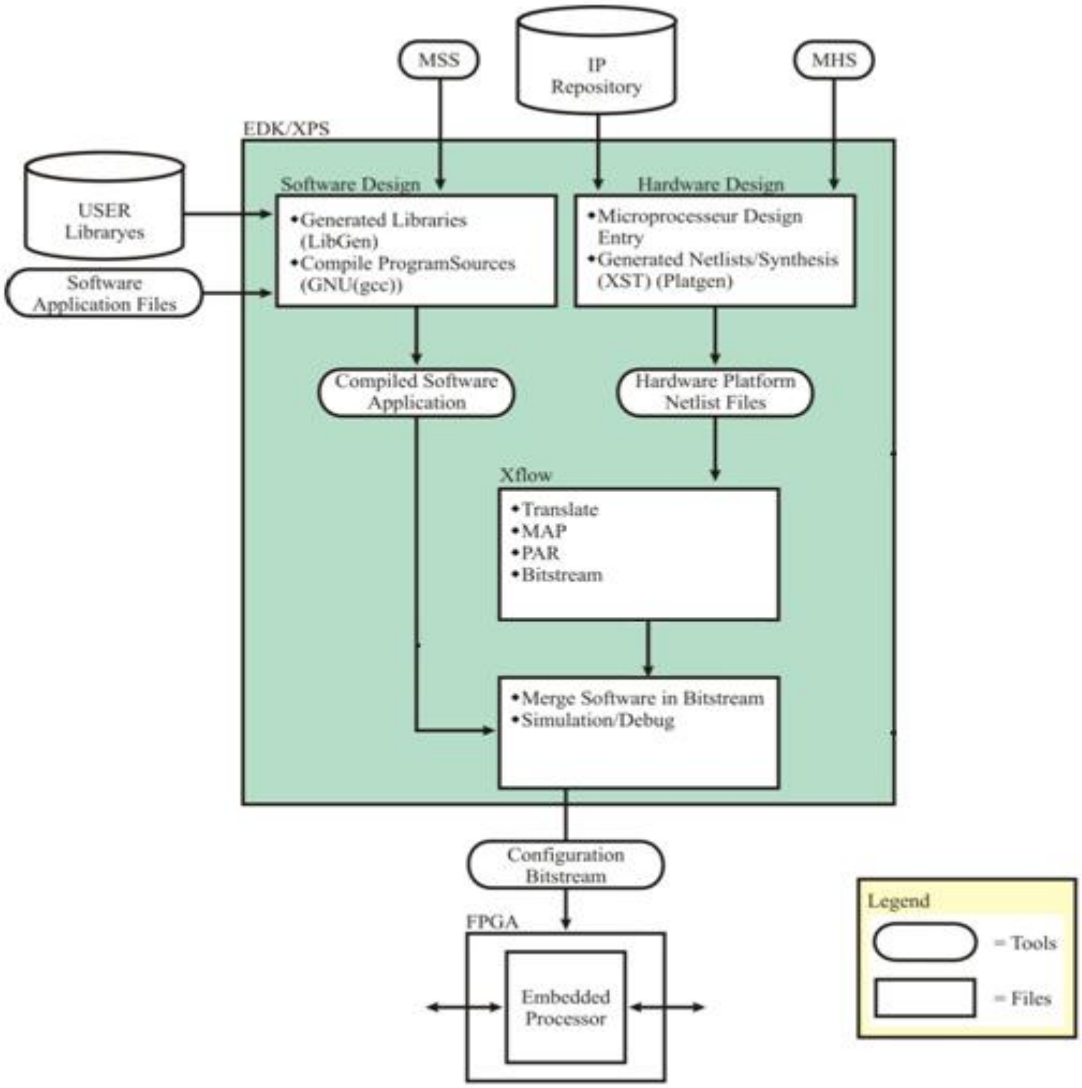


Figure VI .7 : Flot de Conception de XPS [52]

Le software XPS est composé de deux plateformes : La plateforme matérielle et la plateforme logicielle.

La plateforme matérielle est définie par le fichier MHS (Microprocessor Hardware Specification). L'utilisateur peut définir et ajouter ses propres périphériques. L'outil XPS fournit des moyens graphiques pour la création du fichier MHS. Celui-ci définit l'architecture, les connexions et le mapping des adresses du système. A partir du fichier MHS l'outil Platform Generator (PlatGen) crée la netlist du système sous différents formats (NGC, EDIF) et les wrappeurs HDL des niveaux TOP afin que l'utilisateur puisse intégrer ses composants au système.

La plateforme logicielle est définie par le fichier MSS (Microprocessor Software Specification). Ce fichier regroupe les drivers et les librairies pour l'adaptation des paramètres

des périphériques et ceux des processeurs, les routines d'interruption les composants d'entrées/sorties. Le fichier MSS est utilisé par l'outil Library Generator (LibGen) pour l'adaptation des drivers et librairies et les routines d'interruption. A noter que pour le cas d'un système d'exploitation par exemple Linux, le logiciel XPS ne génère qu'une partie des librairies nécessaires au noyau Linux. La compilation complète du noyau s'éffectue sous un environnement Linux et non sous XPS.

La création et la vérification d'une application logicielle passe d'abord par l'écriture du code en C, C++ ou assembleur qui va être exécuté sur les plateformes SW et HW. Ensuite ce code est compilé et linké à l'aide de l'outil GNU (d'autres outils peuvent aussi être utilisés) pour générer le fichier exécutable sous le format ELF (Executable and Link Format). Finalement, XMD et le débogueur de GNU (GDB) sont utilisés pour déboguer l'application.

VI.5.2. Le Processeur Microblaze [27] [28]

Le Microblaze est un soft processeur de 32 bits ayant une architecture Harvard à jeu d'instruction réduit (RISC), c'est-à-dire qu'il inclut des bus d'instructions et de données séparés, de 32 bits chacun fonctionnant à la même vitesse.

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

Le bus LMB est un bus synchrone utilisé principalement pour accéder aux blocks RAM inclus sur le FPGA. Il utilise un minimum de signaux de contrôle et protocole simple pour s'assurer d'accéder à la mémoire rapidement (un front d'horloge).

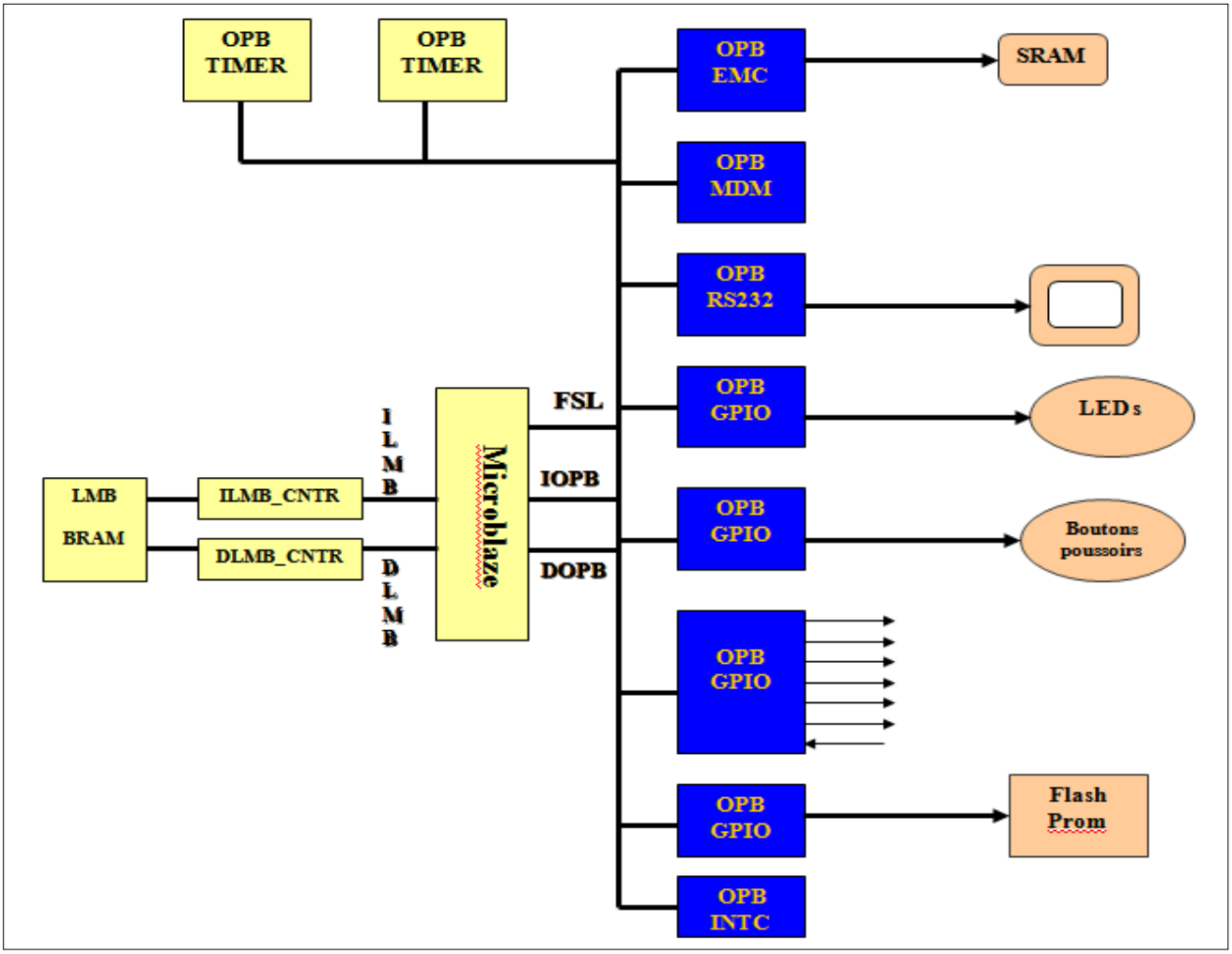


Figure VI.8 : Architecture du processeur microblaze [28]

Les caractéristiques générales du processeur MicroBlaze sont les suivantes :

- L'architecture du MicroBlaze utilise une architecture Harvard avec ses bus de programme et bus de données séparés.
- Le processeur utilise un jeu d'instruction réduit RISC (*Reduced Instruction Set Computer*).
- Le processeur possède un *pipeline* de 5 étages et de profondeur 3. Trois instructions sont exécutées en 9 cycles d'horloge.
- Les instructions sont de 32 bits avec 3 opérandes et 2 modes d'adressages.
- Le bus d'adresse est de 32 bits.

- Il possède 32 registres de 32 bits à usage général.

- Depuis la version 7, le processeur a une unité de gestion de mémoire virtuelle MMU (*Memory Management Unit*) optionnelle. Les parties en couleur grise sont optionnelles et montrent la configurabilité du processeur MicroBlaze, comme le montre la figure VI.9

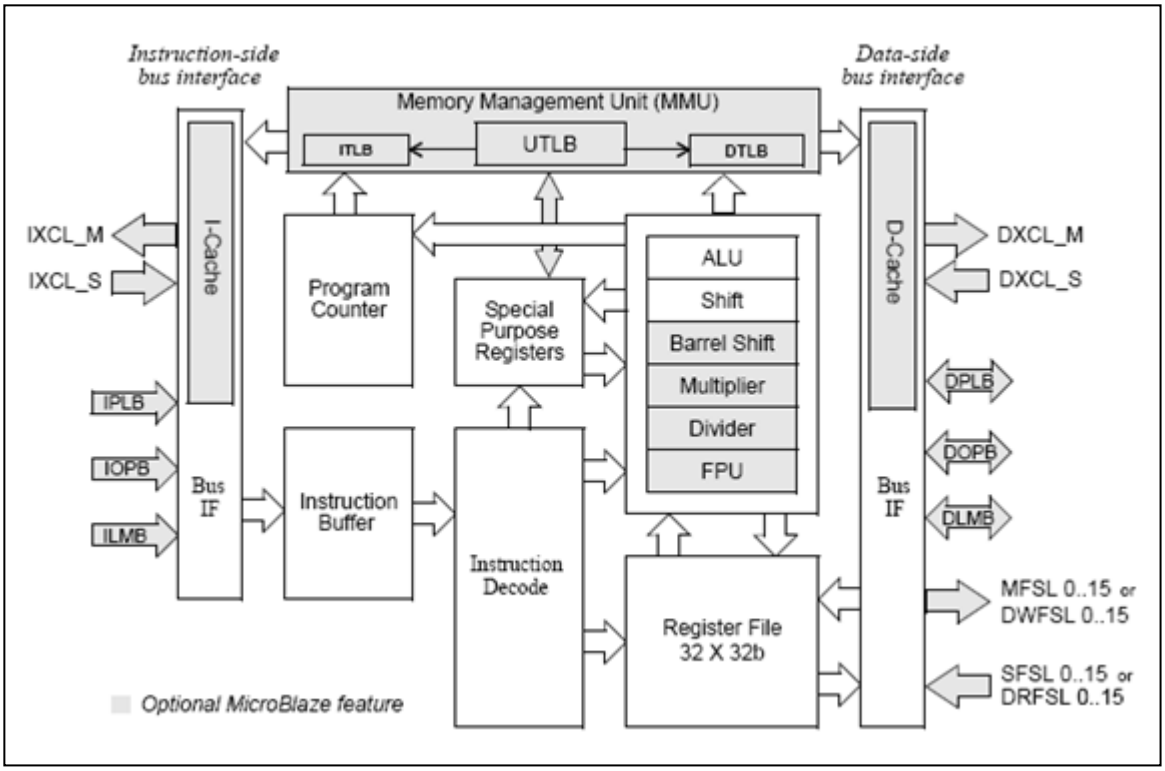


Figure VI.9. Les composants internes du Microblaze [28]

Les options matérielles *Barrel Shifter*, *Multiplier*, *Divider* et FPU (*Floating Point Unit*) permettent d'accélérer les traitements en implantant par matériel des opérations mathématiques (division, multiplication, opération sur des nombres réels).

Le *pipeline* se compose de 5 étages d'exécution :

1. Lecture de l'instruction dans le buffer d'instruction.

2. Décodage de l'instruction dans le décodeur d'instruction.

3. Accès mémoire pour la lecture des opérandes.

4. Calcul ou exécution de l'instruction dans l'ALU.

5. Ecriture du résultat dans la mémoire.

Dans la phase de construction du circuit SoC (*System on Chip*) avec l'outil XPS de Xilinx, il est possible d'inclure différents périphériques standards en utilisant le bus d'adresses/données PLB du processeur MicroBlaze :

· Mémoire.

· Timer.

· Liaison série UART.

· Interface écran LCD.

· E/S parallèles.

· Interfaces Ethernet.

· Interface Compact Flash.

· JTAG.

VI.5.3. Le bus FSL [30]

FSL est un bus de communication unidirectionnel utilisé pour assurer des communications
rapides entre deux éléments (IP-Cores) implémentés sur FPGA. Le MicroBlaze comporte 8 interfaces FSL de type Master et 8 de type Slave (entrées/sorties). Chaque interface FSL est unidirectionnelle (simplex) et met en oeuvre une FIFO (pour stocker les données) et des signaux de contrôle (FULL, EMPTY, WRITE, READ,…).

Les interfaces FSL ont une largeur de 32 bits. De plus, les mêmes canaux FSL peuvent être utilisés pour transmettre ou recevoir aussi bien des signaux de contrôle que des données. La performance de l'interface FSL peut atteindre 300 *MB/sec* pour une fréquence d'horloge de 150 Mhz.

Le Microblaze met aussi à la disposition de l'utilisateur plusieurs fonctions intéressantes, les plus utilisées sont : "**microblaze_bwrite_datafsl**" et "**microblaze_bread_datafsl**". Ces deux fonctions permettent d'échanger des données entre différents MicroBlazes, par exemple, en utilisant la FIFO déjà intégrée dans le bus FSL. Les deux fonctions bwrite et bread sont bloquantes, bwrite se bloque lorsque la FIFO du bus FSL

est saturée et bread se bloque lorsque la FIFO est vide. Il doit y avoir un bwrite pour débloquer la lecture.

Les caractéristiques principales de l'interface FSL sont:

- Une communication point à point unidirectionnelle
- Un mécanisme de communication non arbitré et non partagé
- Support de communication de contrôle et de données
- Communication FIFO
- La taille de la donnée peut être configurée
- Le bus FSL est dirigé par un Maître et peut commander un esclave.

Dans ce qui suit l'architecture détaillée d'un Bus FSL

VI.5.3.1. L'architecture d'un bus FSL [30]

Dans la figure ci-dessous décrit le Block Diagramme expliquant le rôle des Entrées / Sorties d'une interface FSL . Chaque interface FSL est unidirectionnelle (simplex) et met en oeuvre une FIFO (pour stocker les données) et des signaux de contrôle (FULL, EMPTY, WRITE, READ,…).

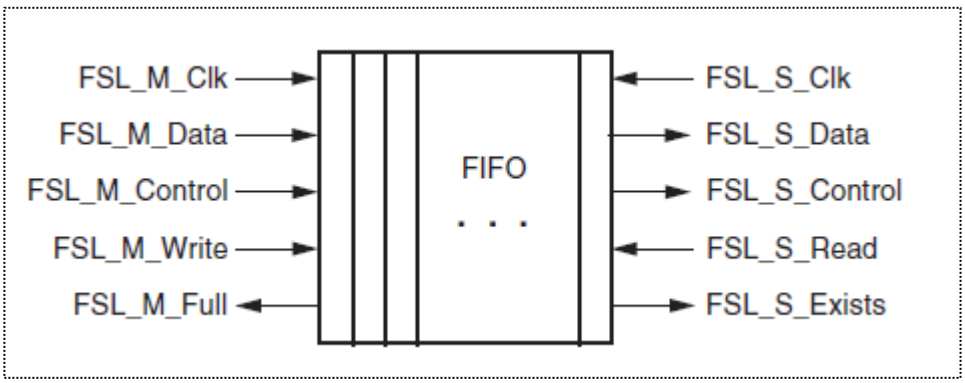


Figure. VI.10. Le Bloc diagramme d'un Block FSL [30]

Dans l tableau ci-dessous une description bréve du role de chaque signal entrant et sortant de l'interface FSL

Signal	***I /O***	***Description***
FSL_Clk	*I*	*L'horloge synchrone pour l'interface Maitre/ Esclave de FSL*
FSL_rst	*O*	*Système reset externe*
FSL_M_Clk	*I*	*L'horloge externe pour l'interface FSL Maitre dans le mode asynchrone*
FSL_M_Data	*I*	*La donnees d'entrees pour une interface FSL Maitre*
FSL_M_Control	*I*	*Le Bus de contrôle pourl'interface FSL Maitre*
FSL_M_Write	*I*	*Le bus qui contrôle l'ecriture pour interface Maitre*
FSL_M_Full	*O*	*Signal de sortie pour interface FSL Maitre , indiquant que la FiFo est pleine*
FSL_S_Clk	*I*	*L'horloge externe pour l'interface FSL Esclave dans le mode asynchrone*
FSL_S_Data	*O*	*La donnees de sorties vers une interface FSL Esclave*
FSL_S_Control	*O*	*Le Bus de contrôle pourl'interface FSL Esclave*
FSL_S_Read	*I*	*Le Bus qui contrôle la lecture dans linterface FSL Esclave*
FSL_S_Exists	*O*	*Le Bus qui supervise les donnes dans FSL Esclave*
FSL_Full	*O*	*Indique que la FIFO est pleine*

Figure. VI.11. Description des entrées/sorties du Block FSL [30]

VI.6. L'interfaçage de l' IP-core RSA_512 avec Microblaze via FSL[Annexe D]

VI.6. Definition : [30] [31]

L'interfaçage de L'IP-Core rsa_top avec EDK Microblaze se fait à travers les bus FSLs dans un système on Chip (Soc) comme le montre la figure VI.12.

Le choix a été porté sur FSL c'est le seule bus de transmission point to point qui offre une meilleure vitesse de transmission car il est Half-duplex (unidirectionnel) et prêt à recevoir des requêtes du Microblaze et des IP-Cores qui l'entoure, à tout moments. Microblaze utilise

FSL soit pour recevoir, soit pour émettre, d'où la notion Master/Slave. Microblaze permet d'établir 16 interfaces FSLs avec l'IP-Core { 8 Maitre } et { 8 Esclaves }.

Dans notre cas on utilise uniquement deux interfaces FSL. La première ipif FSL Slave reçoit des requêtes du Microblaze et les transmets à l'IP-Core rsa_top_512. Et la deuxième ipif Master reçoit des requêtes de L'IP-Core et les transmets au Microblaze, voir Figure VI.12.

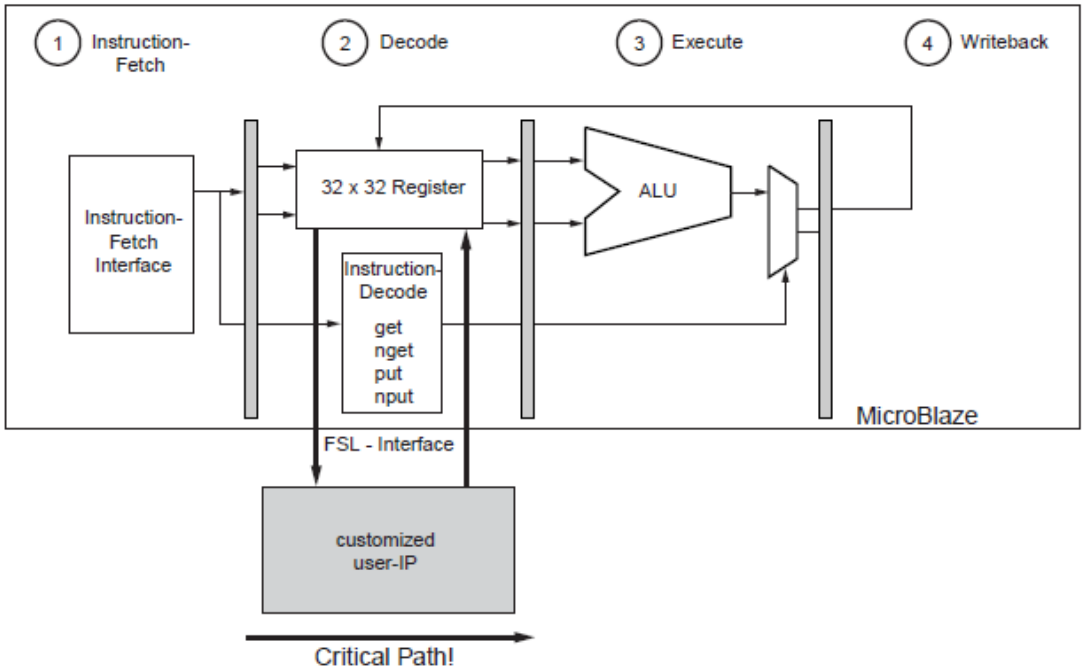


Figure. VI.12. Connexion de l'IP-Core au Microblaze a travers le bus FSL [31]

L'IP-Core est connecté au Microblaze comme on-chip système bus (IP connecté à un bus). L'IP est connecté en dehors de l'architecture de Microblaze. L' avantage de cette connexion est que si l'IP-Core prends 100 cycles de clock pour calculer un résultat complexe, Microblaze peut entre autres exécuter différentes application sans attendre à ce que les 100 cycles de clock finissent.

L'intégration du hardware (IP) dans le software (programmé en C) ne requis pas le code assembleur pour pouvoir l'intégrer, car les interfaces FSL contiennent des macros C prédéfinis (voir ci-dessous) qui sont utilisés pour envoyer des paramètres spécifiques à l'IP-Core et recevoir les résultats.

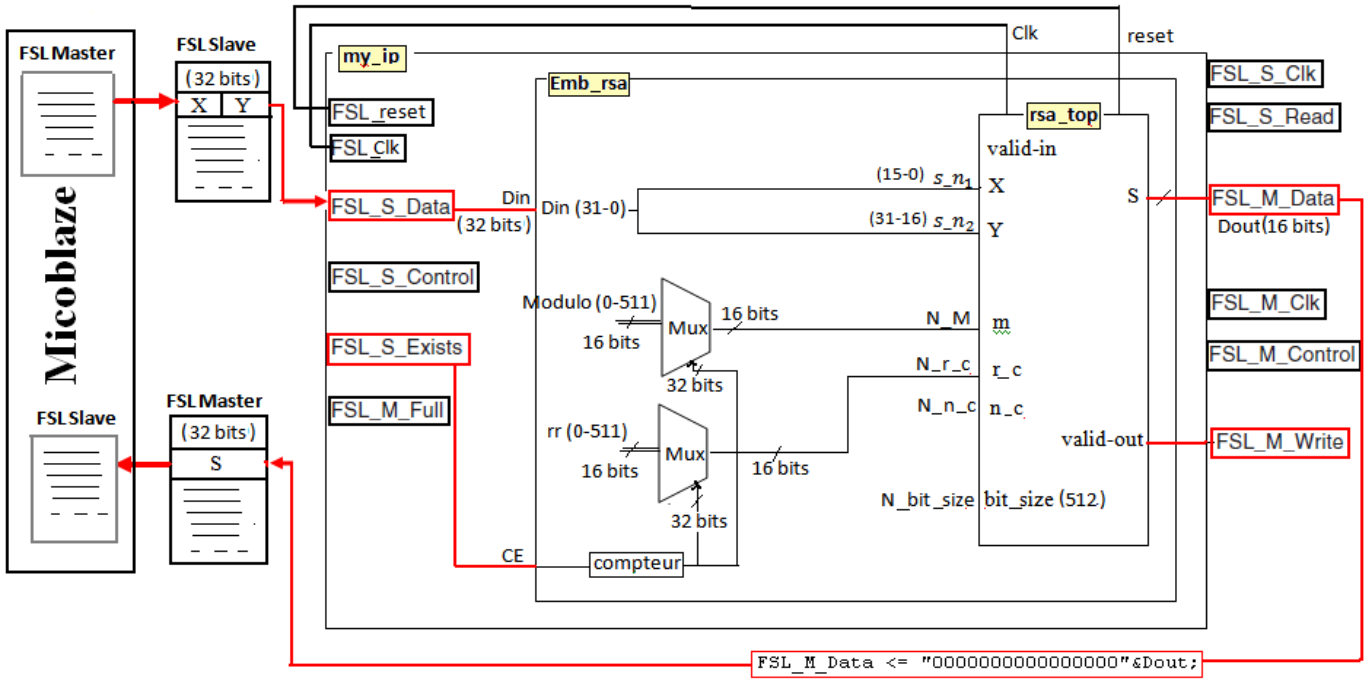


Figure. VI.12. Interfaçage de l'IP-Core rsa_top avec Microblaze [Annexe D]

Afin d'intégrer l'IP-Core rsa_top dans EDK Microblaze, et réaliser l'interfaçage via le protocole FSL, il est nécessaire de passer par les étapes suivantes (détaillé dans la figure VI.12) :

- Création du fichier ''**emb_rsa**'', qui permet d'ajouter '' **rsa_top**'' comme composant, de tel sorte à concaténer ''x'' et ''y'' qui sont de taille de 16 bits dans une seule entrée ''Din'' de 32 bits afin de le faire adapter au interface FSLs qui sont eux aussi de taille de 32 bits (y compris Microblaze). D'autre part, les deux valeurs : r_c et m ont été fixé comme des entrées de taille de 512 bits par des multiplexeurs afin de pouvoir les introduire par paquet de 16 bits.
- Création de l'enveloppe ''**my_ip**'' qui englobe le composant '' emb_rsa'' et qui est entouré par les connexions unidirectionnelles FSL, afin d'assurer l'interfaçage avec Microblaze.
- Quand le signal de contrôle valid_in est à 1, le Microblaze ''Master'' fournie les entrées ''x'' et ''y''à l'enveloppe ''**my_ip**''qui est considérée dans ce cas comme interface de FSL Slave.
- Quand le signal de contrôle valid_out est à 1, l'enveloppe ''**my_ip**''qui est dans ce cas ''Master'' fournie la sortie ''S'' au Microblaze ''slave''.

Dans ce qui suit, on abordera l'intégration de notre IP-Core ''**rsa_top**'' au sein du Soc, qui a pour tâche d'accélérer l'éxécution de l'exponentiation modulaire qui consomme beaucoup de temps en software mais moins de temps sous EDK Microblaze, en utilisant les outils de Xilinx XPS.

VI.6.1. Création de Project sous EDK : [Annexe D]

En s'appuyant sur les outils de XPS, l'implémentation du schéma de la figure VI.11, au sein du système embarquée, La création du Project BSB nomé Virtex_5 sous EDK, a été faite en choisissant la carte ''***xc5vsx95t-3ff1136''*** comme carte de développement, et composé du microprocesseur Microblaze relié à travers les bus PLB à un certain nombre de périphériques cités ci-dessous:

- UPS Uartlite : RS232
- LEDs _8Bits : XPS_GPIO
- LEDs_ 7 Segments : XPS_GPIO
- Push_Buttons_3Bits : XPS_GPIO
- DIP_Switches_8Bits

Le système contient deux types de mémoires

- BRAM = 32Kb (interne reliée à Microblaze avec le bus Local Memory Bus « LMB»)
- SRAM (externe reliée à Microblaze avec le bus PLB)

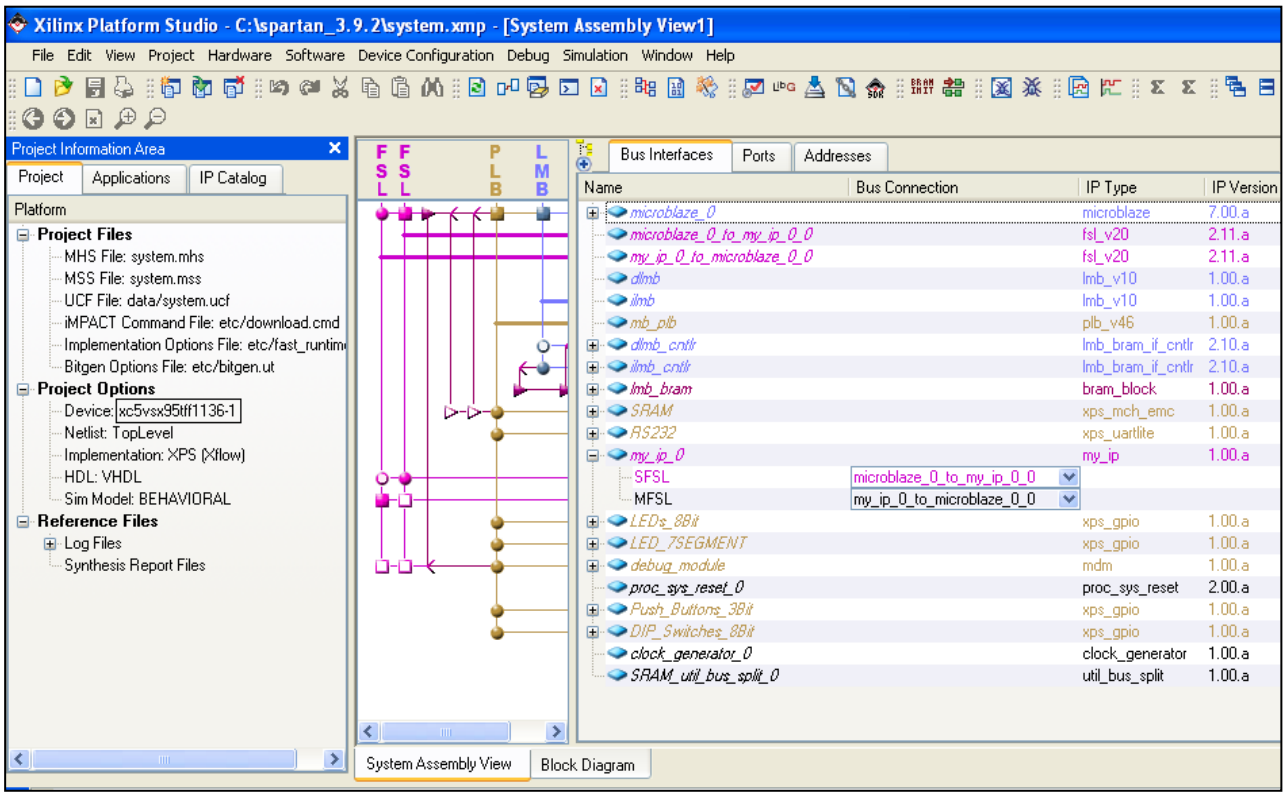


Figure VI.13. Création du Project BSB pour la carte de développement

VI.6.2. Création de l'enveloppe de l'IP-core sous XPS [Annexe D]

Afin d'intégrer notre IP-core rsa-512 au niveau de notre système, il est nécessaire de lui créer une enveloppe (Wrapper) nommé ‘’**my_ip**‘’ lui permettant de communiquer avec Microblaze suivant les étapes : Hardware → Create or import peripheral→ create template for

a new peripheral.

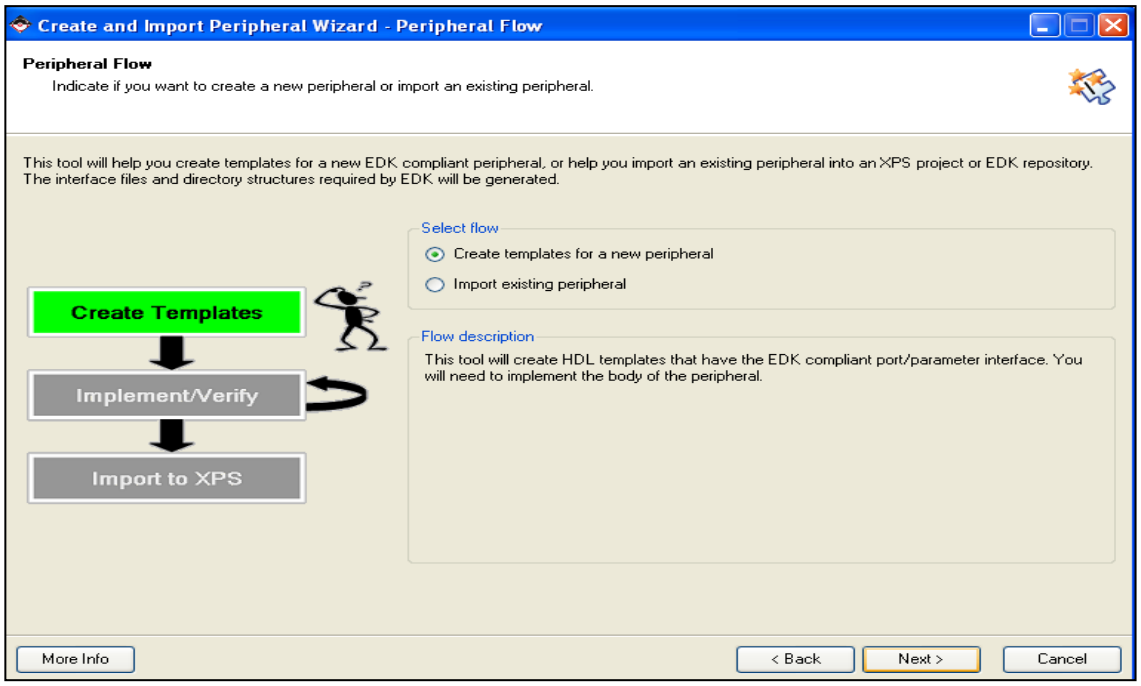


Figure VI.14. Création d'une nouvelle l'enveloppe

Création d'un nouveau périphérique, c'est l'enveloppe ‘’my_ip’’ qui englobera l'IP-Core ‘’rsa_top’’

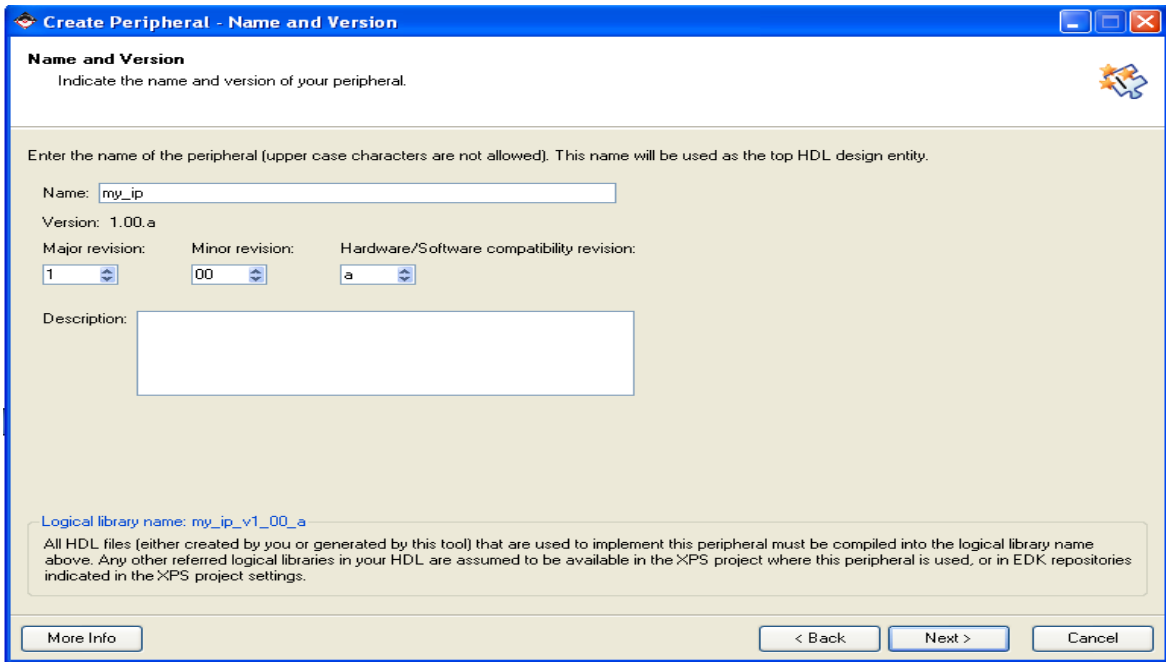


Figure VI.15.Nomme l'envellope ‘’my_ip’’

L'enveloppe ‘’my_ip’’ relira notre IP-Core à Microblaze à travers les bus FSLs

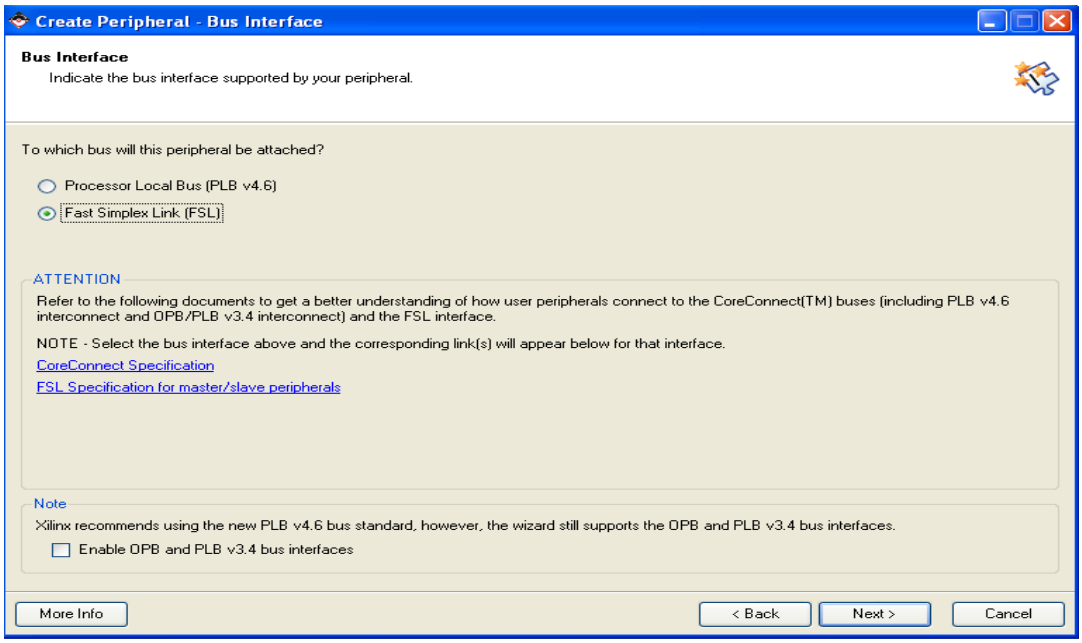


Figure VI.16. Le choix du bus FSL

Comme on a des entrées x,y sur 16 bits chacun et une sortie S sur 16 bits, donc on va choisir deux interfaces FSL, une pour l'entrée et l'autre pour la sortie :

- FSL input est sur 32 Bits, concatène les deux entrées {x=16 bits} et {y=16bits}
- FSL output est sur 32 Bits , contiendra la sortie S {S= 16 bits + 000…00}= 32 bits

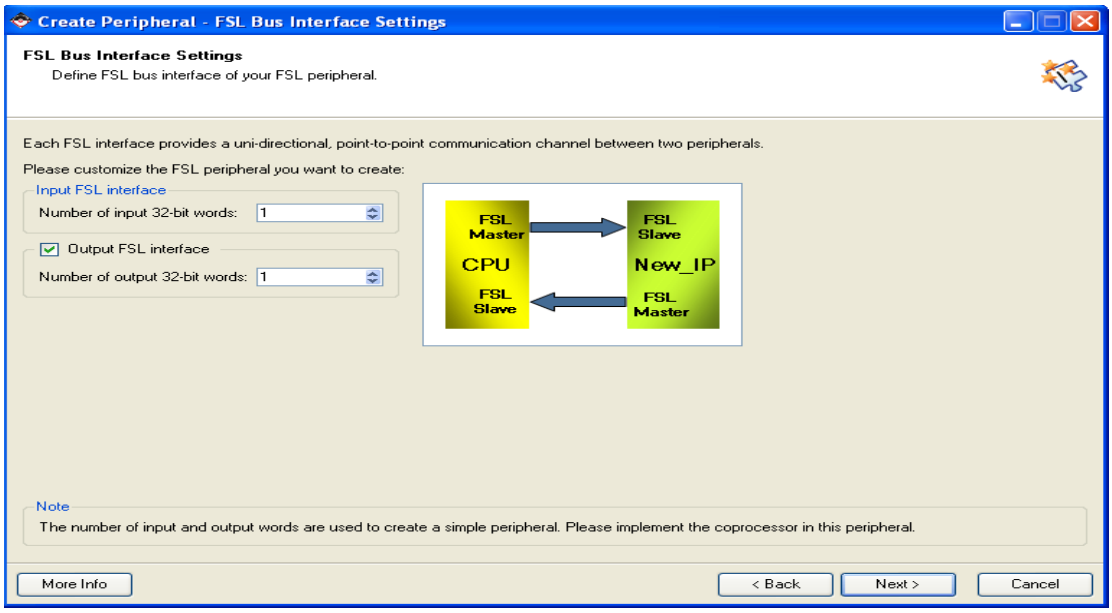


Figure VI.17. le choix de deux interface FSL ; Maitre /Esclave

Configuration de l'interface EDK pour paramétrer le design

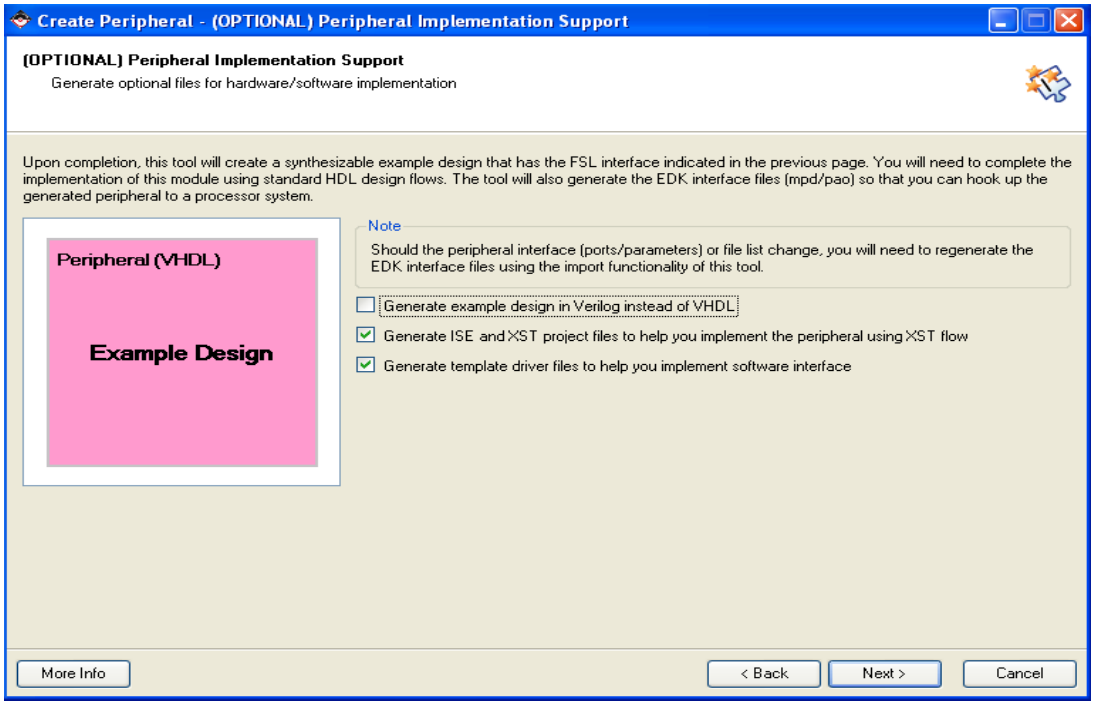


Figure VI.18. La génération de ISE et XST projets

C'est dans cette étape que la création de l'enveloppe ''my_ip'' est achevée

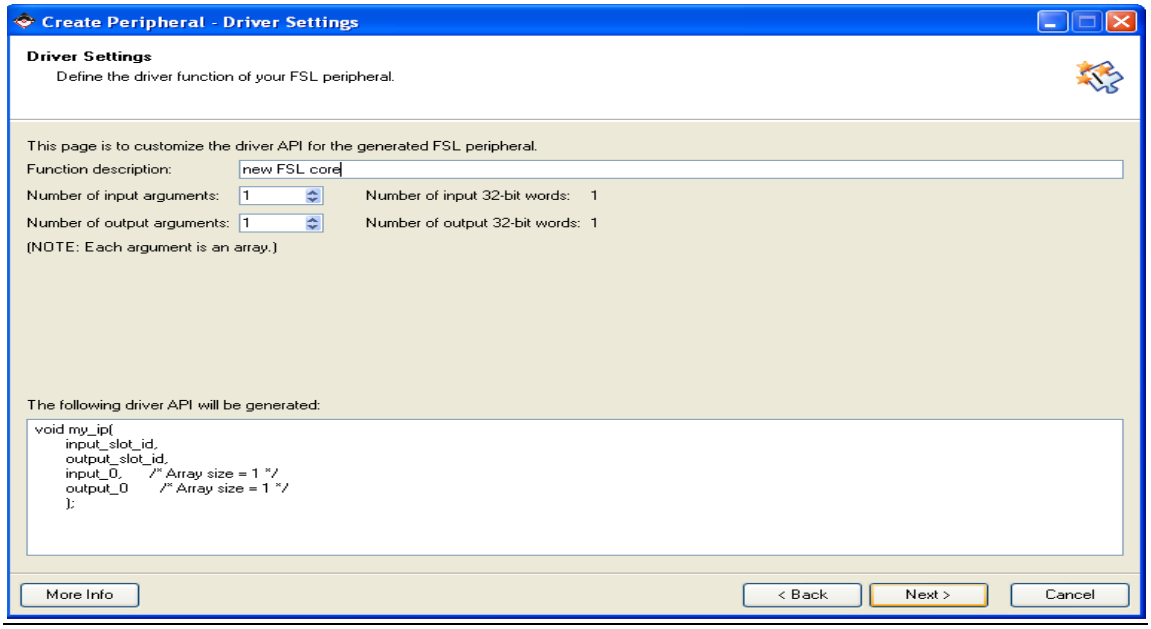


Figure VI.19. Création de l'enveloppe ''my_ip''

A présent l'enveloppe ''my_ip'' est visible au niveau de dans notre système. Elle a été rajoute a ''System Assembly View''

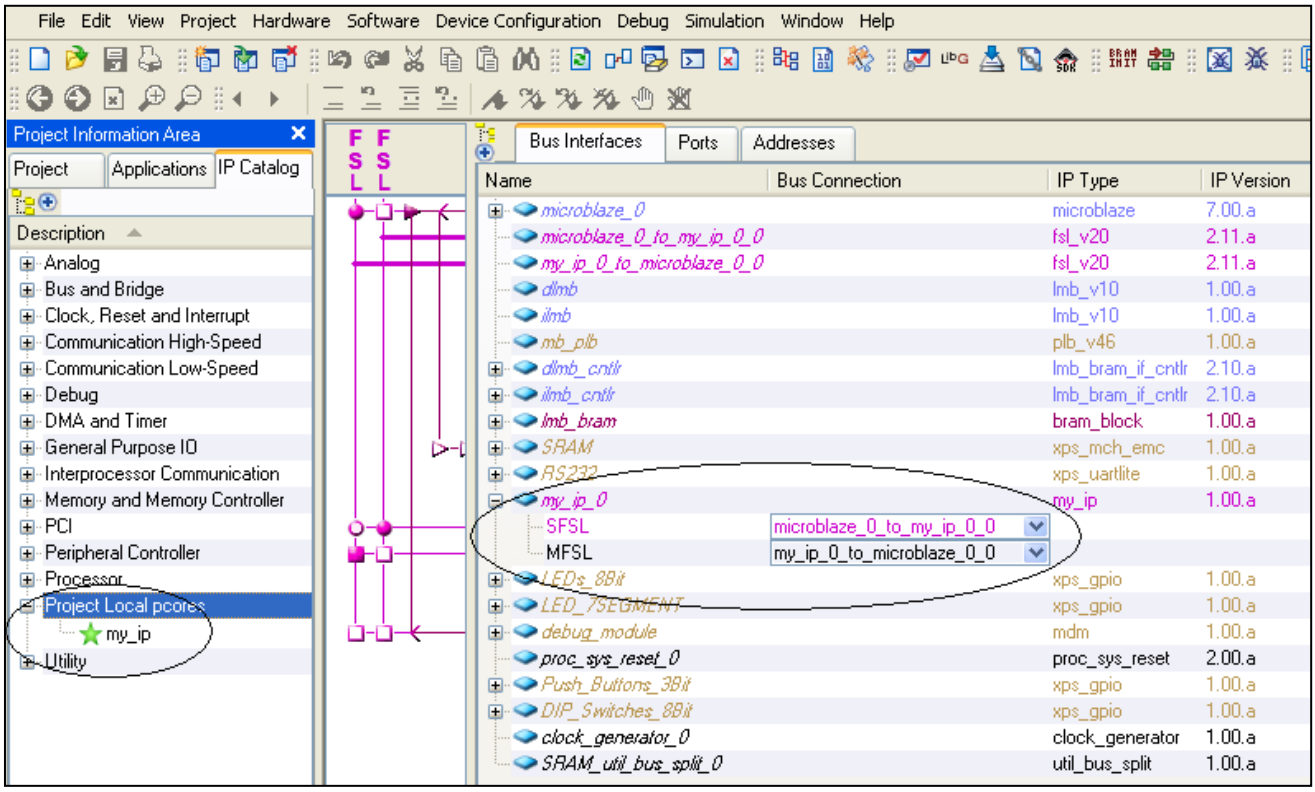


Figure VI.20. L'apparition de l'enveloppe ''my_ip'' dans ''System Assembly View''

Vérification du dossier : my_ip_v1_00_a qui a été créé lors la création de notre enveloppe.

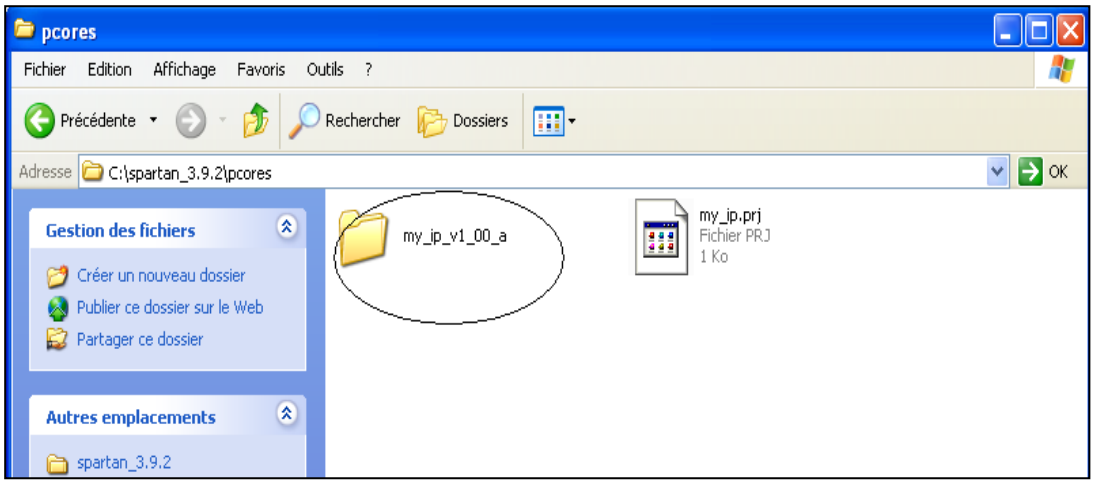


Figure VI.21. Le dossier de l'enveloppe ''my_ip''

VI.6.3. L'importation de l'IP-Core rsa_top dans le projet EDK [Annexe D]

Afin d'importer l'IP core rsa_top dans notre projet il faut copier ses fichiers sources dans un premier temps au niveau du répertoire suivant: C:\Virtex_5 \pcores\my_ip_v1_00_a\hdl\vhdl

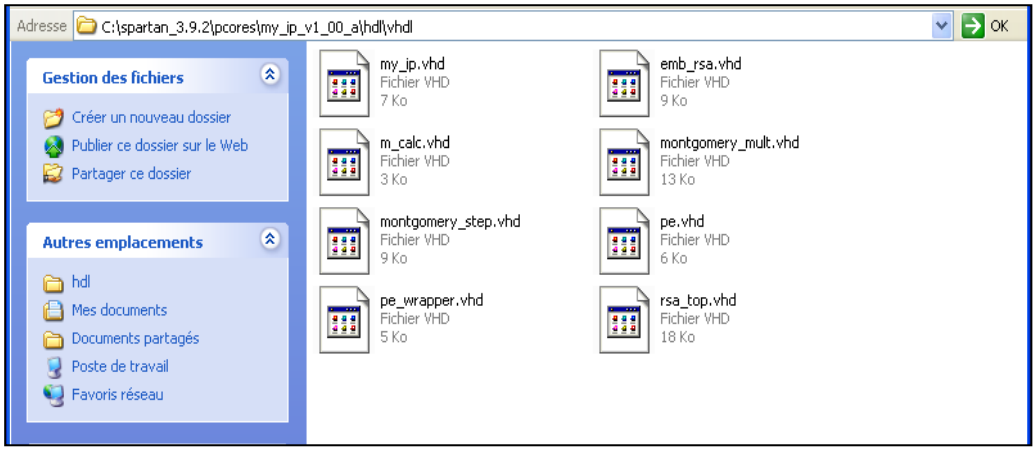


Figure VI.22.L' importation des modules de rsa_top dans le dossier de l'enveloppe

L'étape suivante consiste à mettre à jour le fichier : my_ip_v2_1_0.pao et s'assurer que tous les modules de rsa_top existent dans ce fichier , dont le chemin est : C:\Virtex_5 \pcores\my_ip_v1_00_a \data\my_ip_v2_1_0.pao

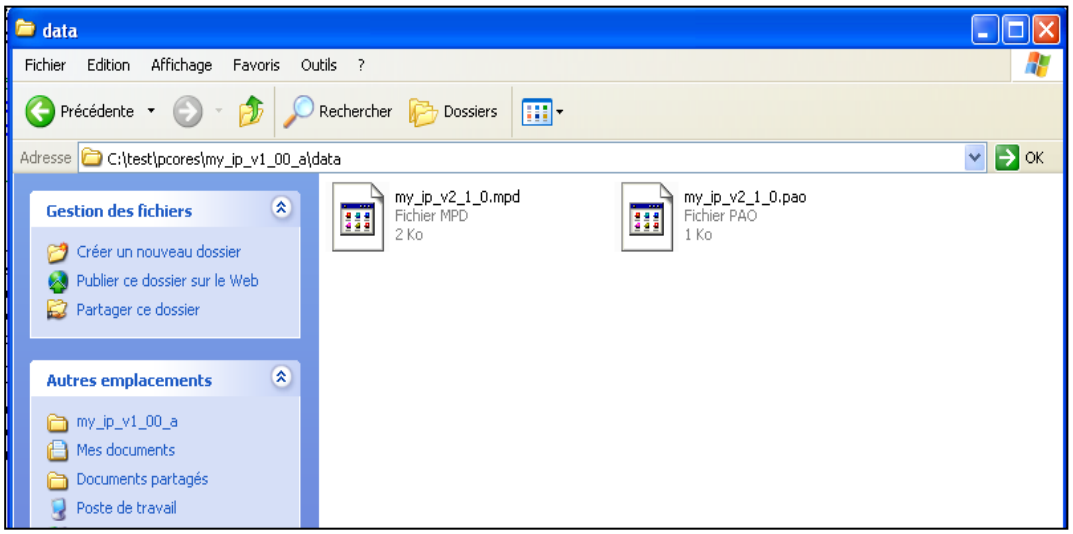


Figure VI.23. Confirmation de l'existence du fichier de l'enveloppe '' my_ip…pao''

Remplir le fichier ''my_ip_v2_1_0.pao'' par tous les modules de L'IP-Core'' rsa_top'' et les modules de l'enveloppe, comme le montre la figure ci-dessous :

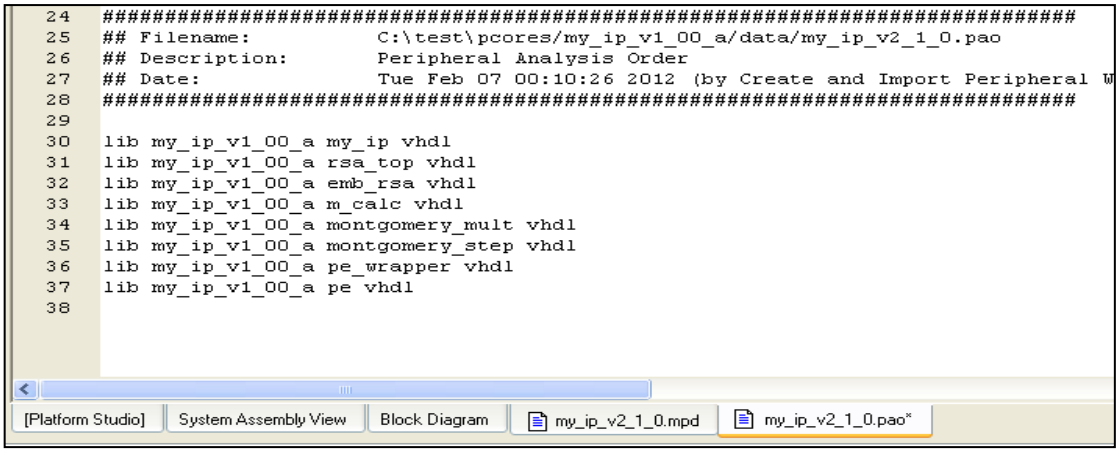


Figure VI.24. Les modules de l'IP-Core et de l'enveloppe

Coller L'IP-Core rsa_512 dans l'enveloppe pour que tout le circuit puisse le reconnaitre par la manière suivante :

Hardware -- > Create or import peripheral -- >import existing peripheral.

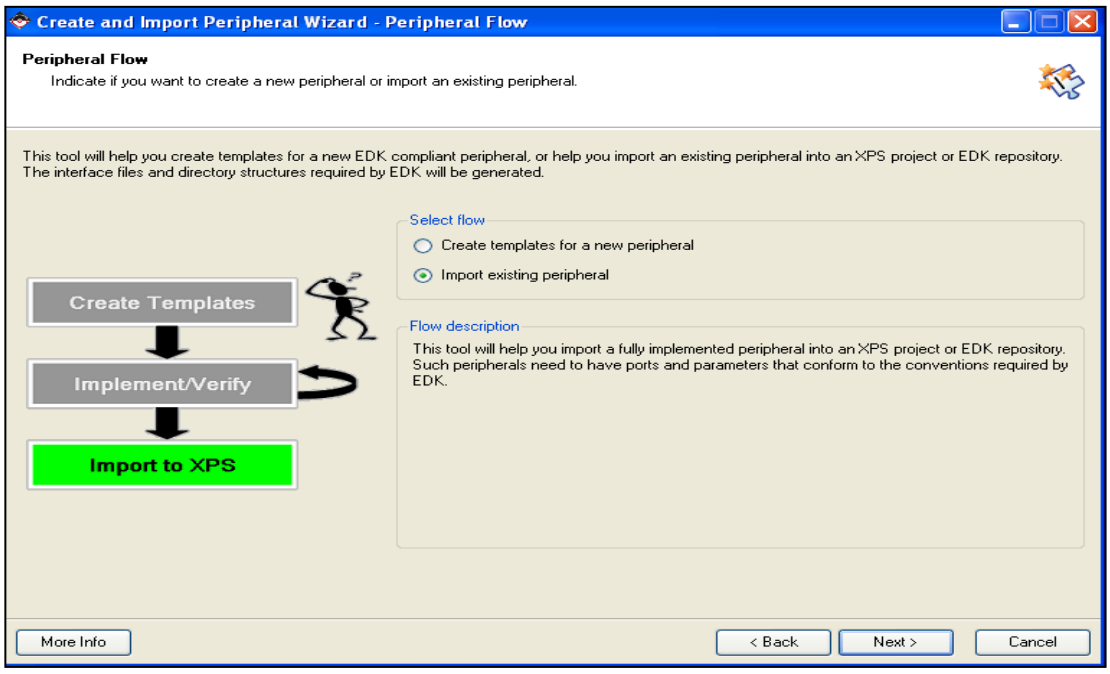


Figure VI.25. Importer L'IP-Core dans l'envellope''my_ip''

Préciser le nom et le chemin de l'enveloppe '' my_ip'' :

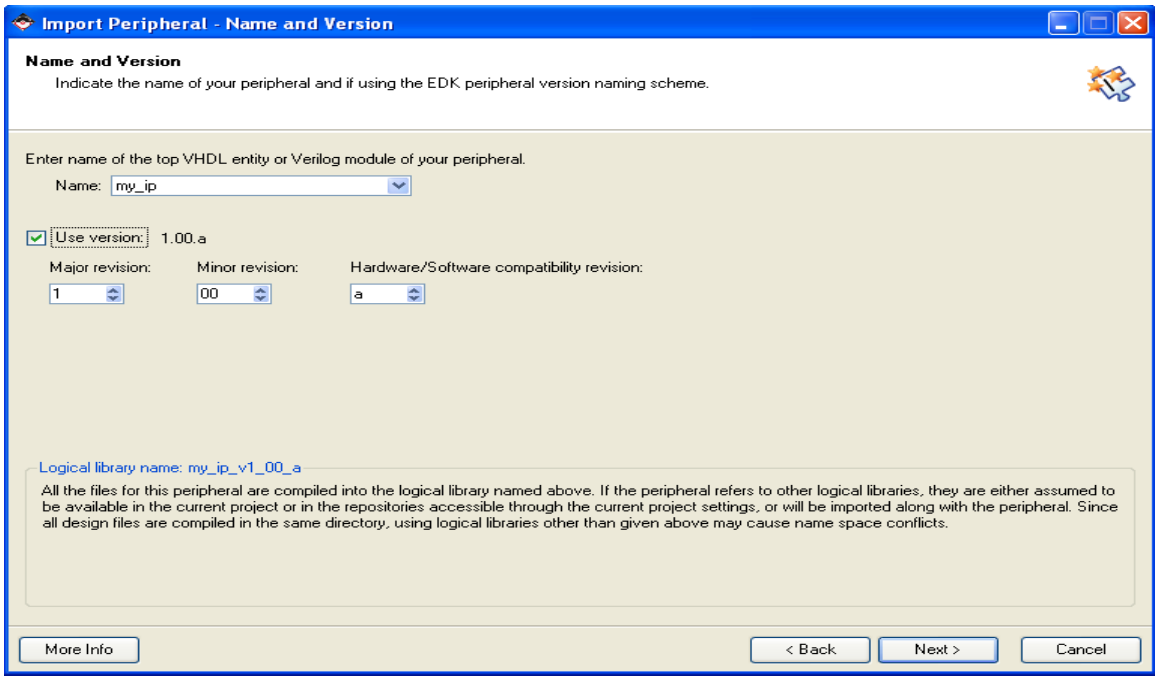


Figure VI.26. Préciser le nom de l'enveloppe lors de l'importation de L'IP-Core

Comme EDK ne reconnait pas l'extension '' xco'' de ISE , il l'a remplace par l'extension ngc , souvent utilisée pour la génération des netlist files

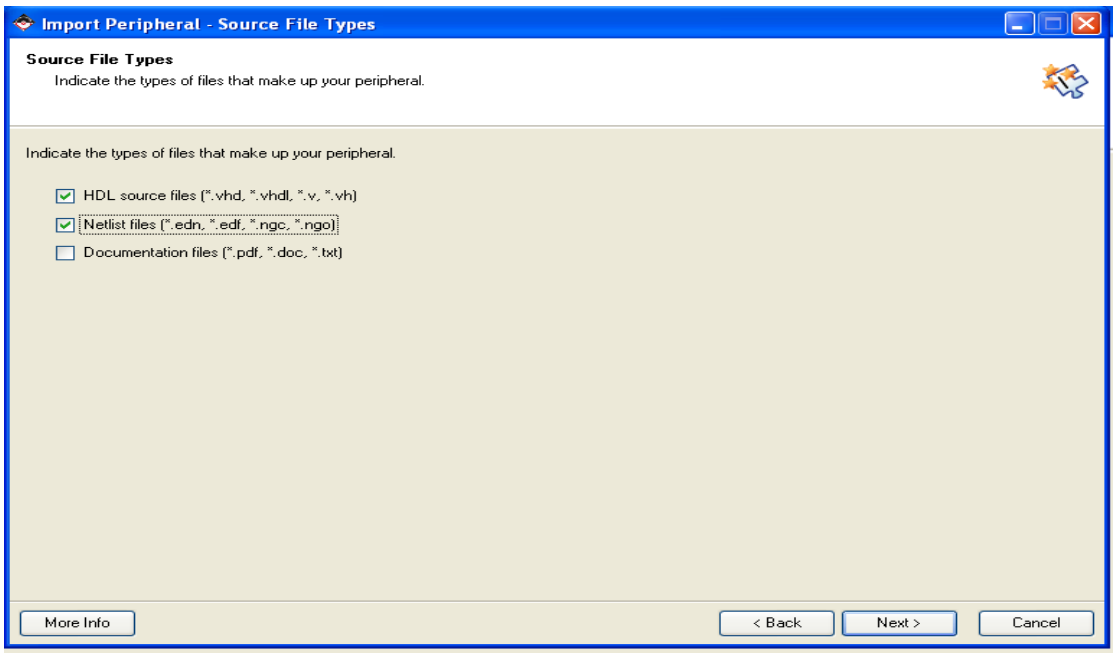


Figure VI.27. Les fichiers ''Netlist''

Préciser le nom de fichier ‘’’my_ip_v2_1_0.pao ‘’qui contient tous les modules de l'enveloppe, lors de son l'importation au sein de l'envellope

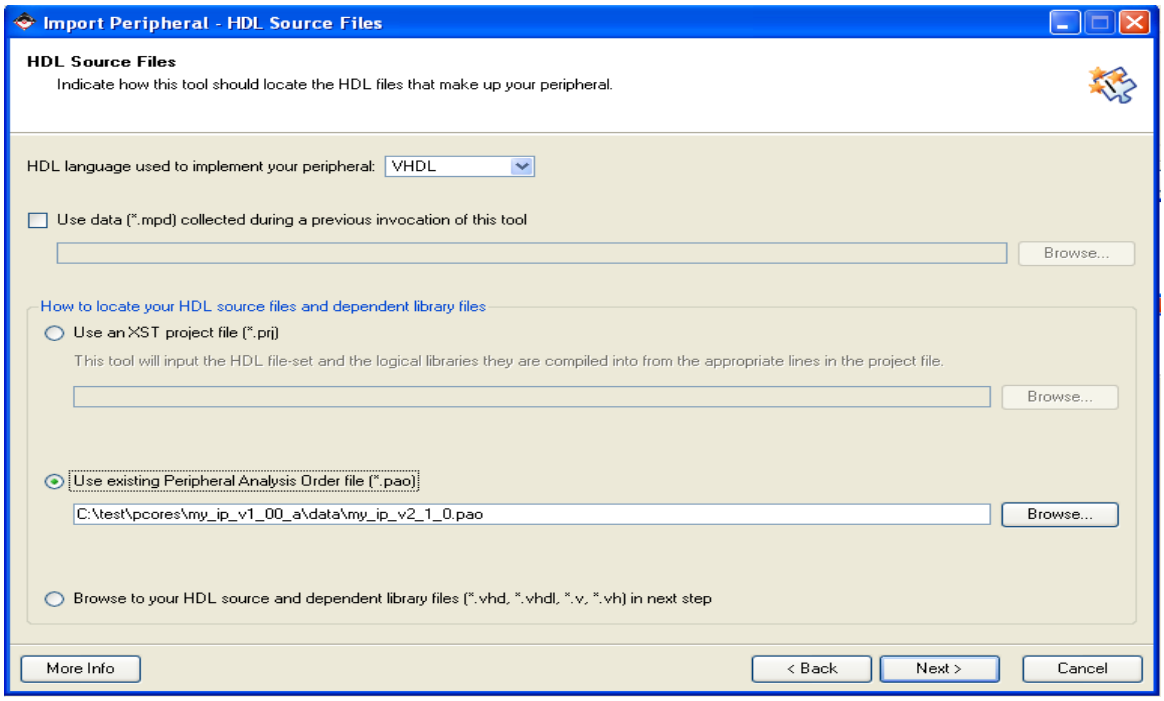


Figure VI.28. L'importation du fichier ‘’ my_ip….pao’’ dans l'enveloppe

Importer tous les modules existant dans l'enveloppe ‘’my_ip’’

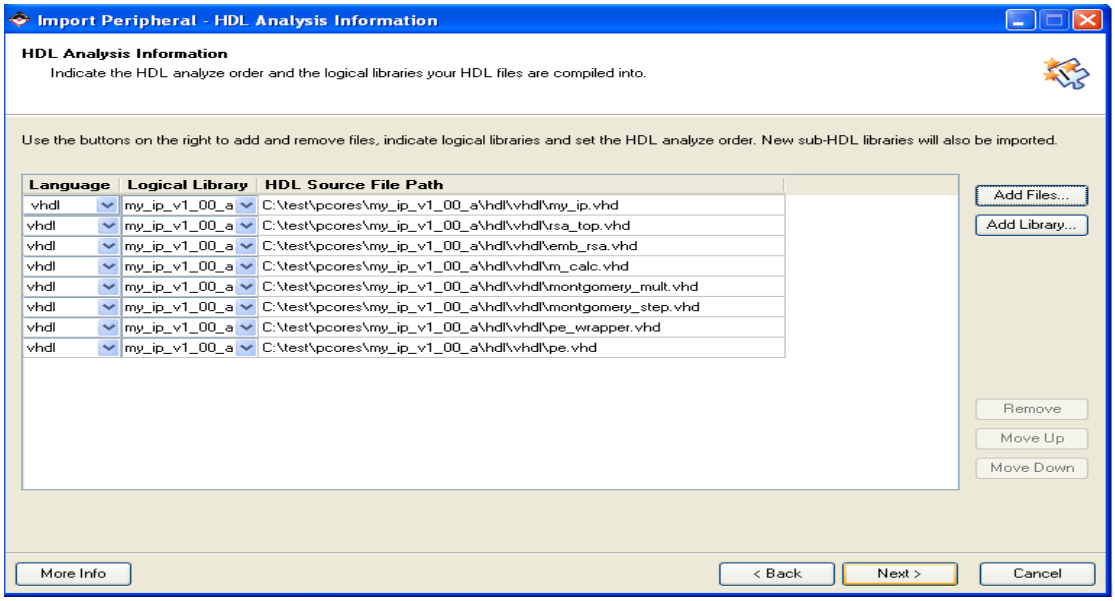


Figure VI.29. Importation des modules existant dans ‘’my_ip’’

Au moment de l'importation des modules de l'envellope dans le projet, des message d'erreurs s'affichent due à la programmation des fichiers VHDL, la correction a été faite au fur et à mesure pou faire adapter les modules de l'IP-Core à l'enveloppe.

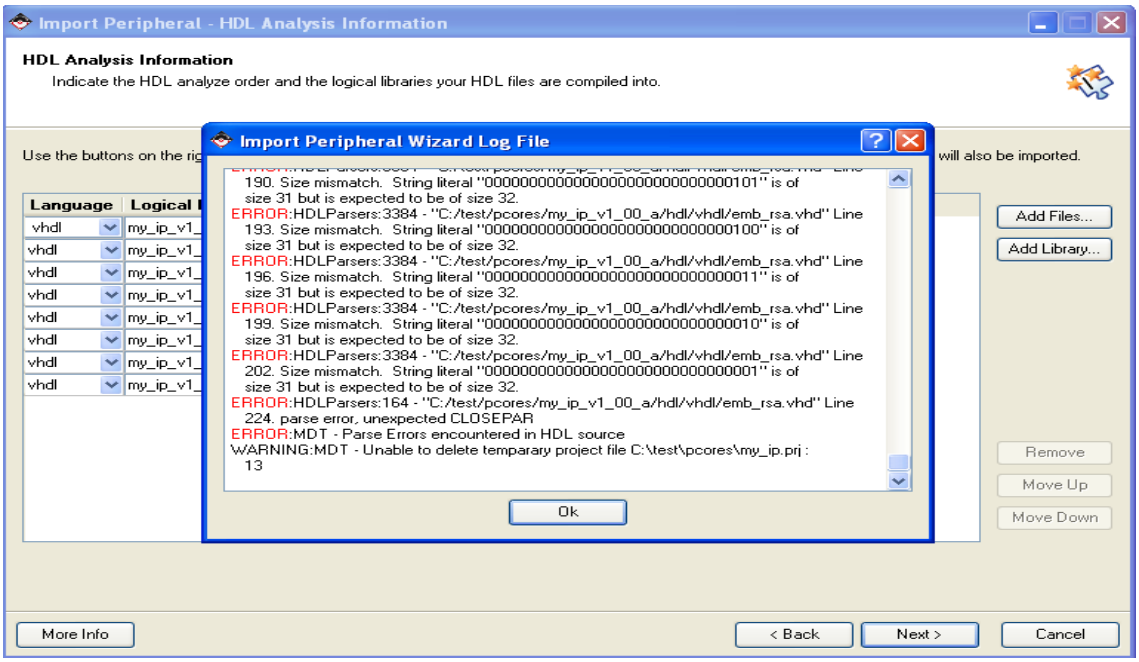


Figure VI.30. Correction des messages d'erreurs lors de l'importation des modules de L'IP-Core a l'envellope

Cette étape consiste à la configuration des interfaces FSL comme le montre la figure ci-dessous, Cocher : FSL Master (MFSL), et Cocher : FSL Slave (SFSL)

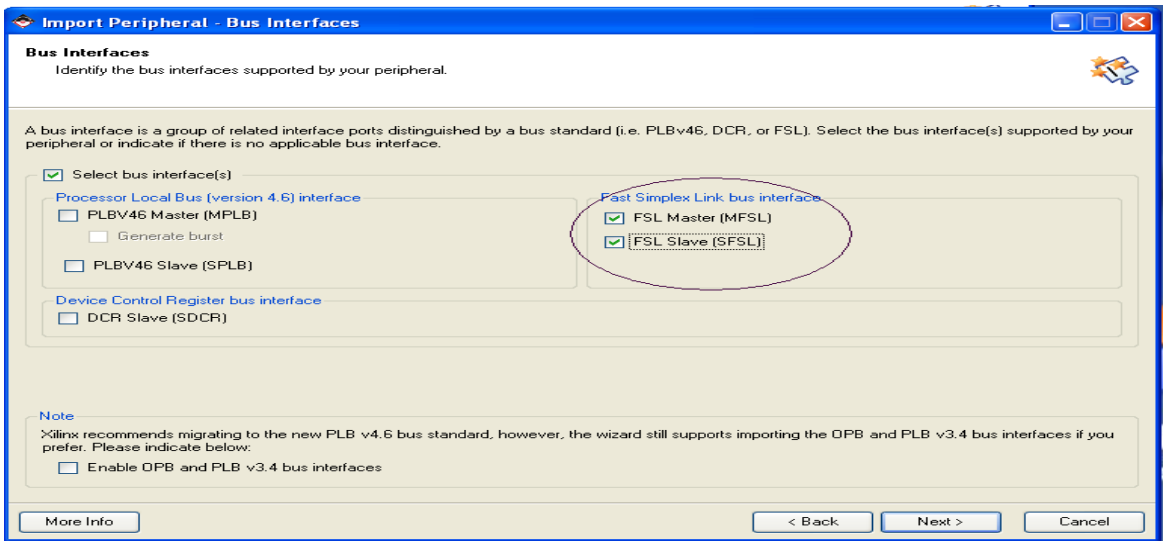


Figure VI.31. Configuration des interfaces FSL

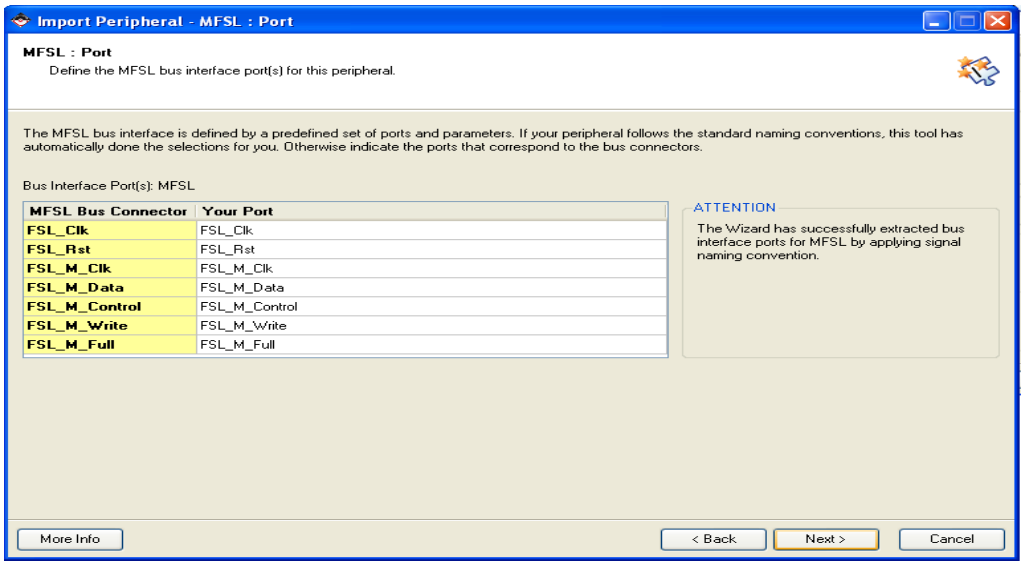


Figure VI.32. Configuration de l'interface MFSL

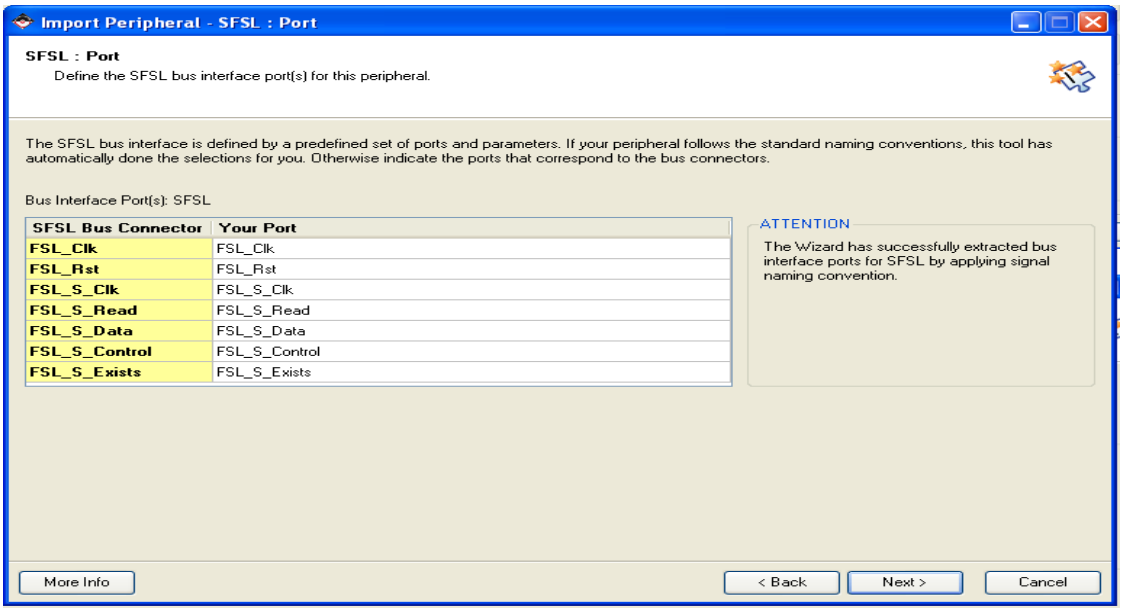


Figure VI.33. Configuration de l'interface SFSL

Sélectionner toutes les Entrées/Sorties des bus FSL dans l'enveloppe ''my_ip'' afin de procéder à l'interfaçage avec Micrblaze.

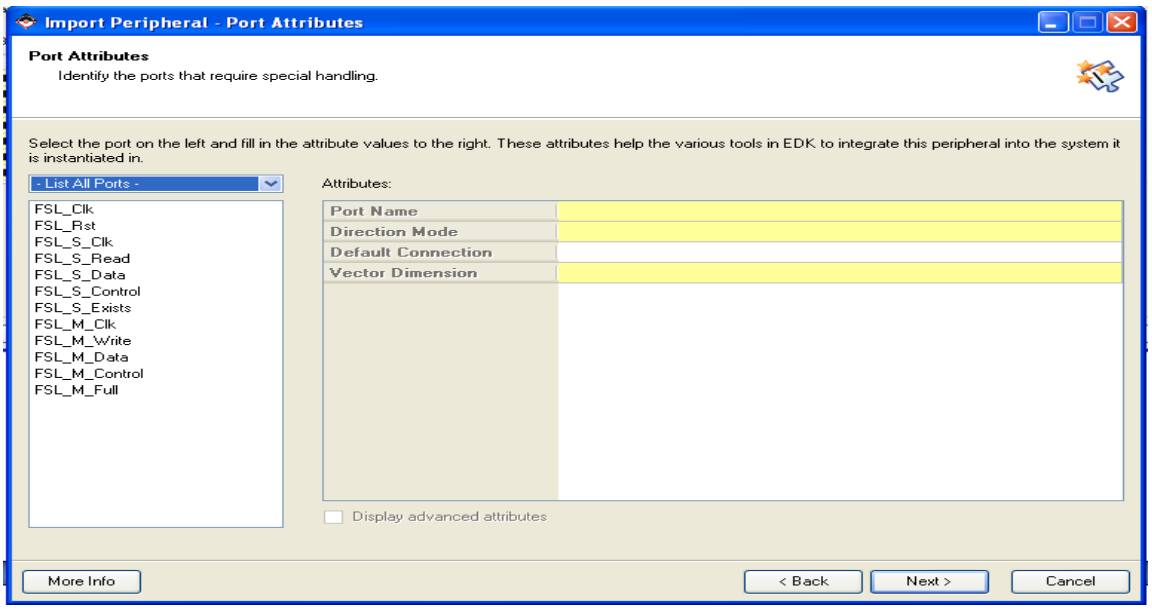


Figure VI.34. Les Entrées/Sorties de l'interface FSL

Importation des Core-générateurs de L'IP-Core rsa_top vers notre enveloppe '' my_ip'', qui était déjà crée dans le dossier ''Corgen'' : C:\Virtex_5\coregen

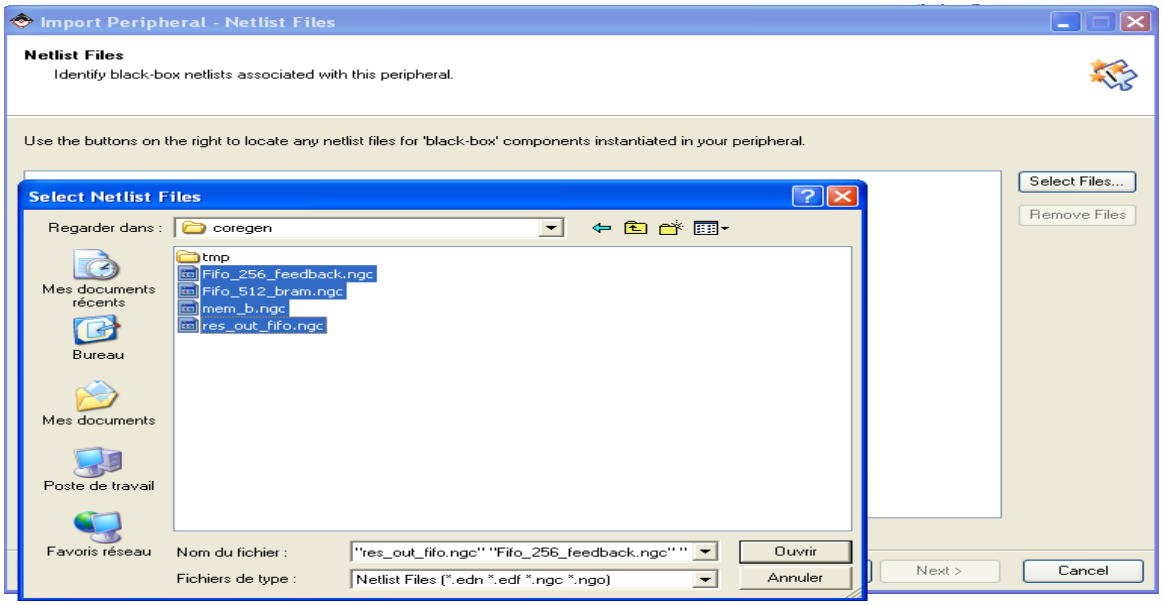


Figure VI.35. L'importation des Corgen vers l'enveloppe''my_ip''

Déduction des corgens sous forme de Netlist

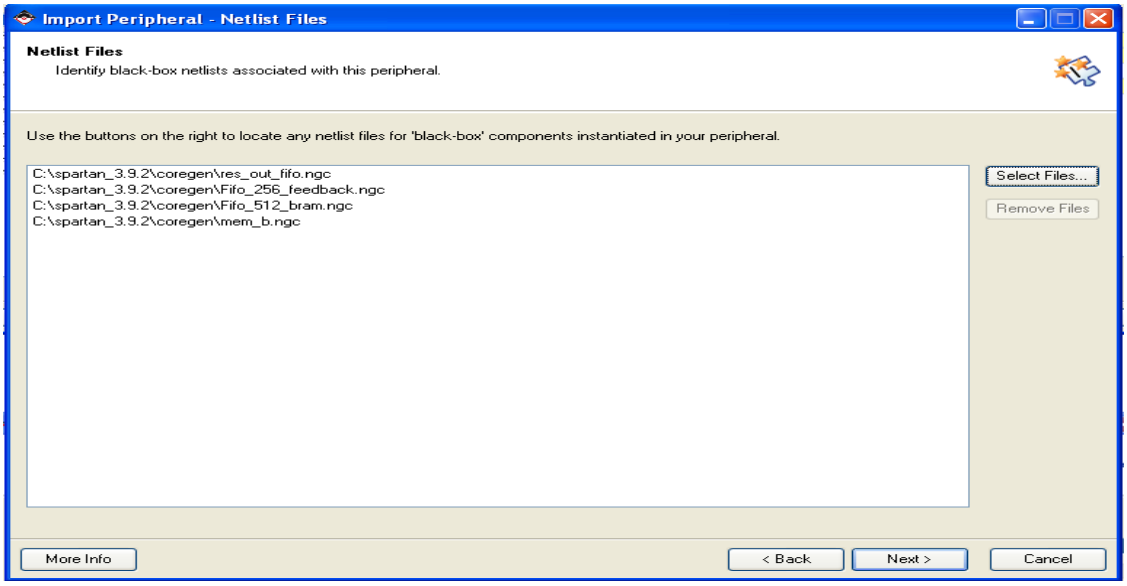


Figure VI.36. Les Corgens sous forme de Netlist

Aprés avoir importer tous les Modules et Corgens de L'IP-Core rsa_top, c'est maintenant que l'envellope '' my_ip'' sera rajouté manuelement dans le projet XPS, en cliquant sur :

Hardware -- > Configure Processor

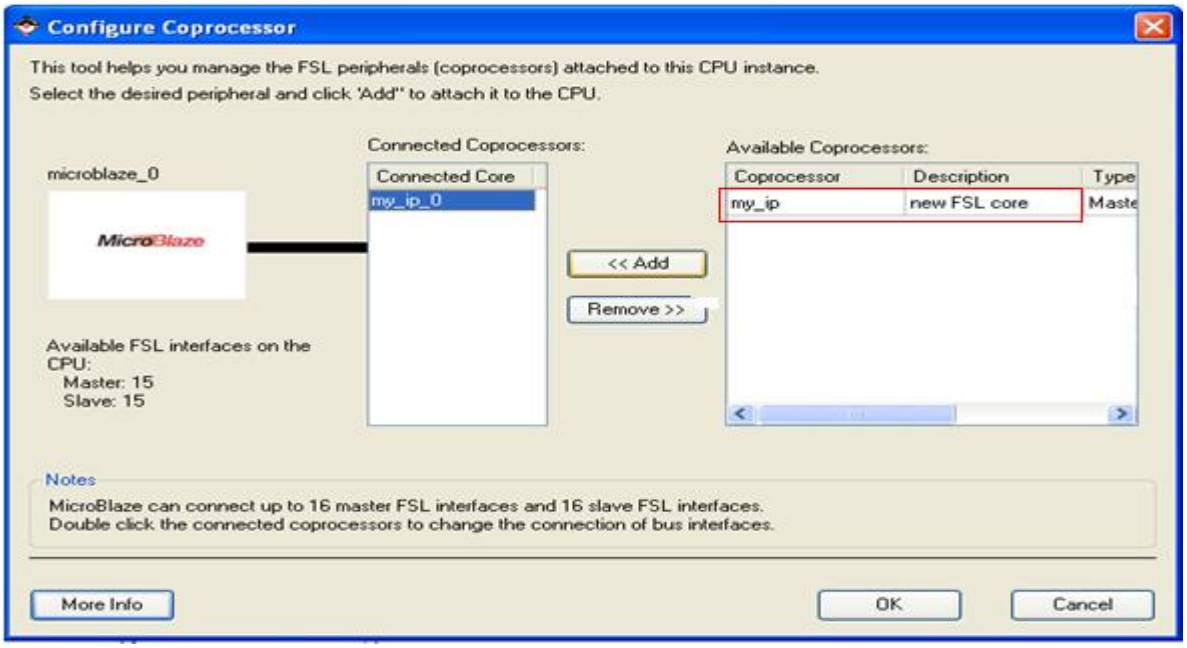


Figure VI.37. L'importation de l'enveloppe ''my_ip'' vers le project

C'est maintenant que l'enveloppe ''my _ip'' a été introduite dans notre projet XPS , comme le montre le block'' System AssemblyView''. Les connexions entre Microblaze et L'IP-Core sont indiquées ci-dessous :

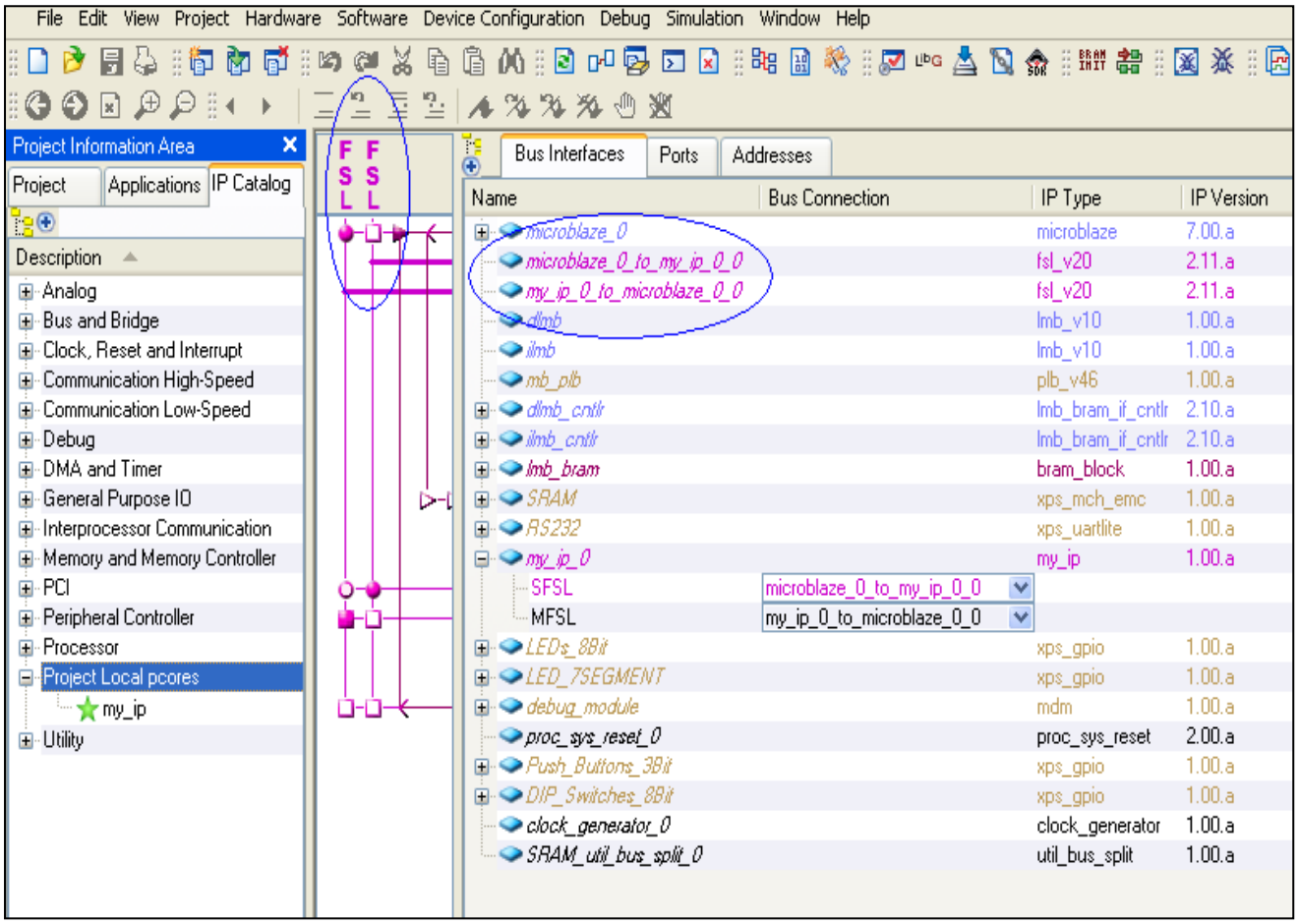


Figure VI.38 . La connexion de l'enveloppe ''my_ip'' avec Microblaze

L'interfaçage de l'enveloppe ''my_ip''y compris l'IP-Core rsa_top avec Microblaze dans le Block Diagramme, comme le montre le schéma ci-dessous,.

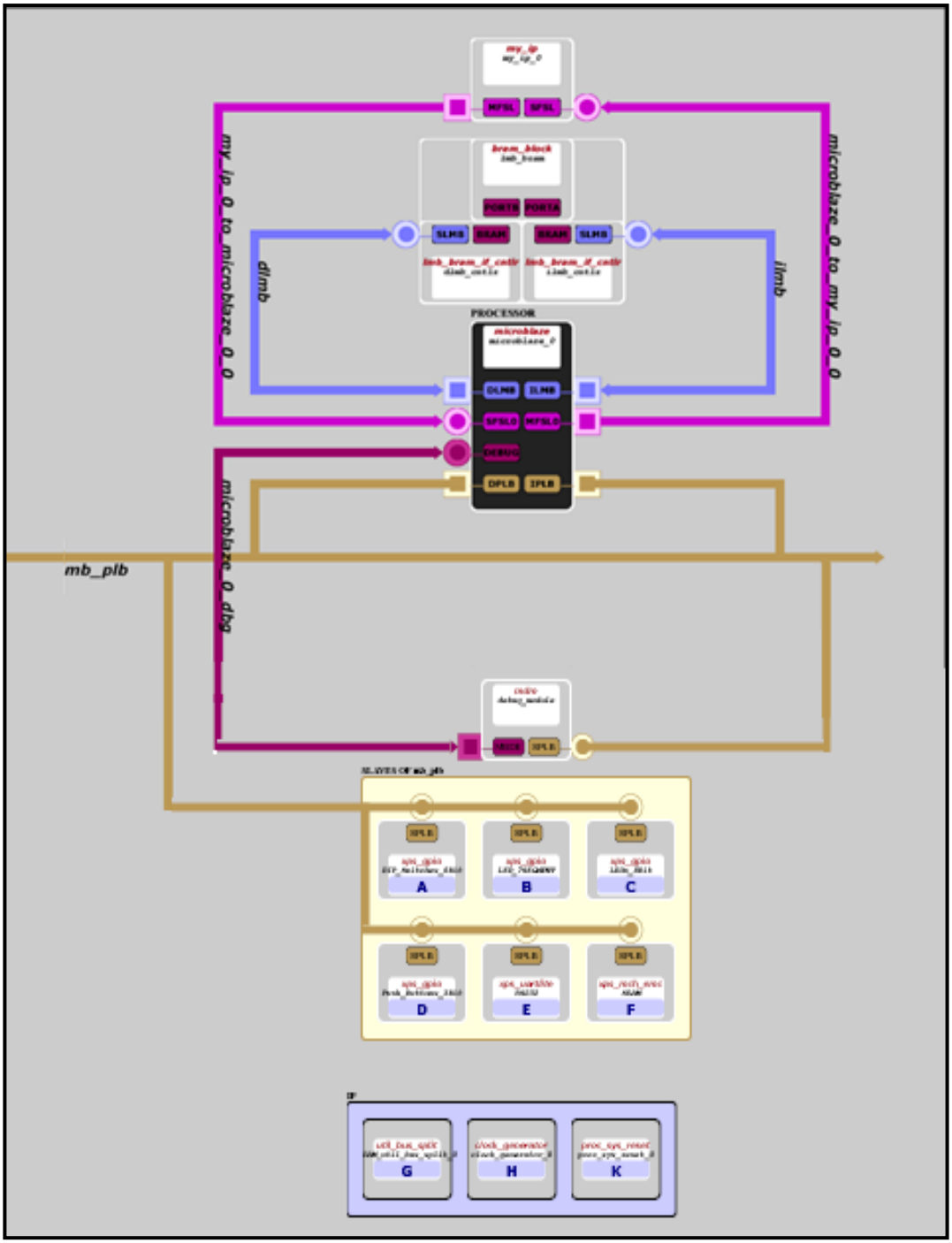


Figure VI.39 . L'intégration de l'IP-Core rsa_top dans le système embarqué

FSL est de taille 32 bits

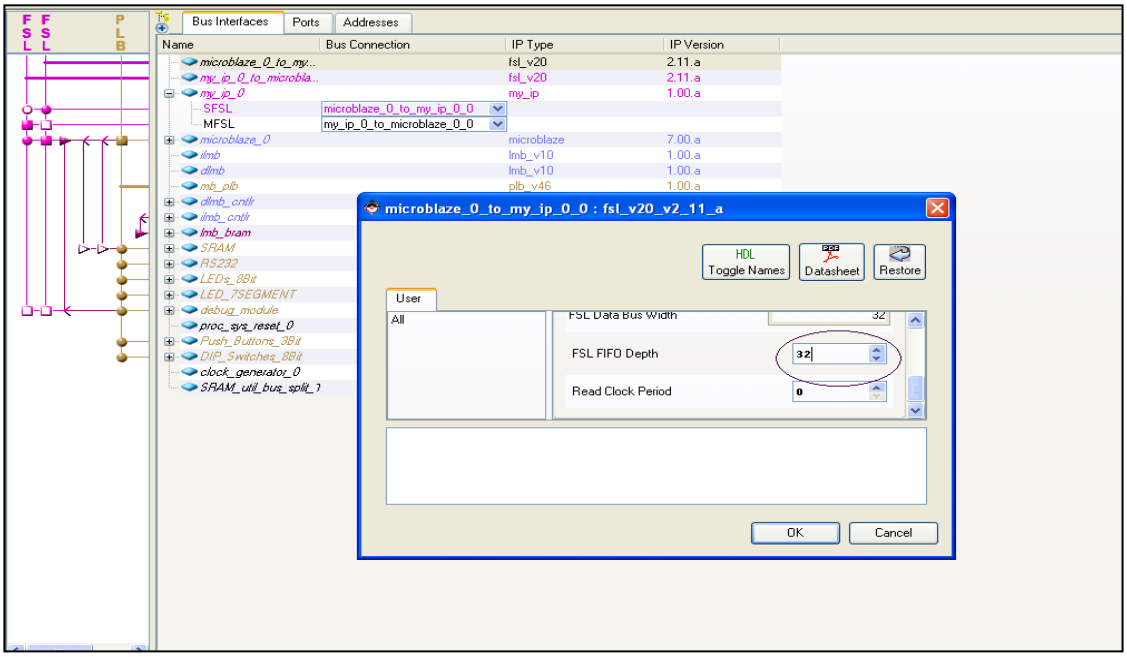


Figure VI.40 . Interfaces SFSL et MFSL de taille 32 bits

VI.6.4. Génération du Fichier testbench dans le projet : [Annexe D]

Génération du fichier testbench en programme C, qui nous permet d'introduire les données { x et y } de taille 16 bits. Ces valeurs seront introduites à travers les interfaces FSL par couplet de 16 bits afin de compléter un mot de 32 bits. La figure ci-dessous montre l'emplacement du couplet {x, y} dans le programme (C:\Virtex_5\TestApp_Memory.c) voir (Annexe D, EDK/XPS).

```
58
59      input_0[0] = 0x42B1F3AB;
60      input_0[1] = 0x1ad38e40;
61      input_0[2] = 0xa8276af9;
62      input_0[3] = 0x0d644e63;
63      input_0[4] = 0x3f769600;
64      input_0[5] = 0x667068f4;
65      input_0[6] = 0x5a1c5a12;
66      input_0[7] = 0x4844d62e;
67      input_0[8] = 0xd5f28fc1;
68      input_0[9] = 0xb65a031d;
69      input_0[10]= 0x034ff496;
70      input_0[11]= 0x96351268;
71      input_0[12] = 0x386e2e5a;
72      input_0[13] = 0xfb73c1d6;
73      input_0[14] = 0x5623cd8f;
74      input_0[15] = 0x9041e4d2;
75      input_0[16] = 0x99da36c6;
76      input_0[17] = 0xcd68df4a;
77      input_0[18] = 0x9a948e40;
78      input_0[19] = 0xccc86af9;
79      input_0[20] = 0xed494e63;
80      input_0[21] = 0x4d259600;
81      input_0[22] = 0x3b8e68f4;
82      input_0[23] = 0x36d55a12;
83      input_0[24] = 0x3a75d62e;
84      input_0[25] = 0x77a38fc1;
85      input_0[26] = 0x367fd2cd;
86      input_0[27] = 0x68dec6e4;
87      input_0[28] = 0x59a44a36;
88      input_0[29] = 0x9f89f6df;
89      input_0[30] = 0xed71061c;
90      input_0[31] = 0x059f06c8;
91
92    nputfsl(input_0[i],0);
93    //microblaze_nbwrite_datafsl(input_0[i],0);
```


Figure VI.41 . Le fichier '' testbench''

VI.6.5. Compilation et Simulation du fichier testbench via Modelsim

Choisir Modelsim comme simulateur pour générer le fichier testbench :

Project → Project Options → HDL and Simulation → ''Modelsim'' → cocher "generate testbench template"

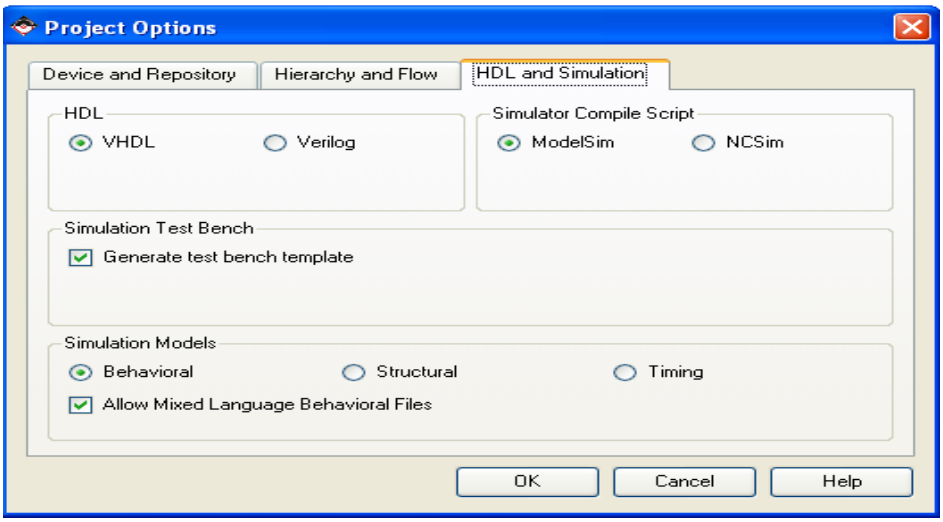


Figure VI.42 . Générer Le fichier du testbench via Modelsim

Compiler le fichier ''testbench'' comme il s'en suit :
Simulation →compile simulation Libraries → compilation Done ! a 100%

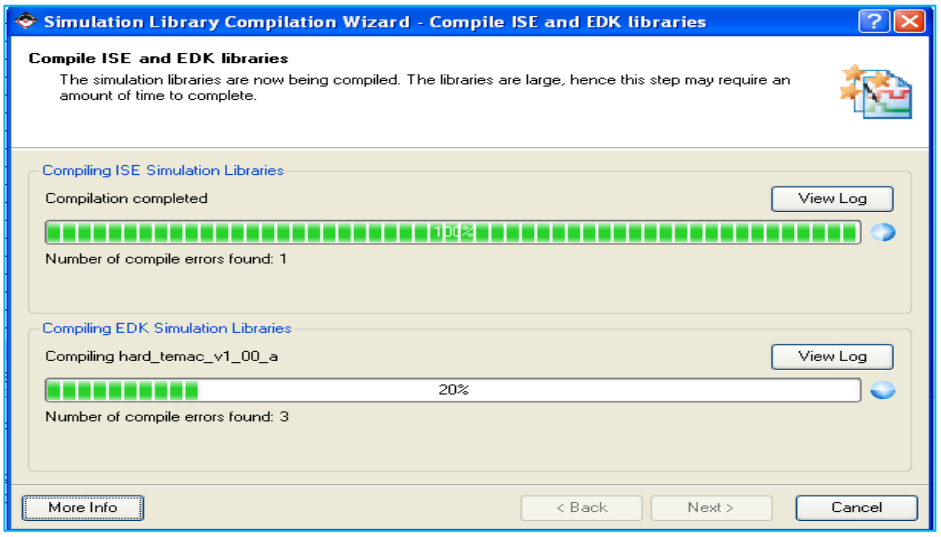


Figure VI.43 . Compilation fichier du testbench

Aprés que la compilation a été achevée à 100% et sans erreur, on procéde à la simulation en faisant appel à Modelsim :

Cliquer sur Simulation → launch HDL Simulator

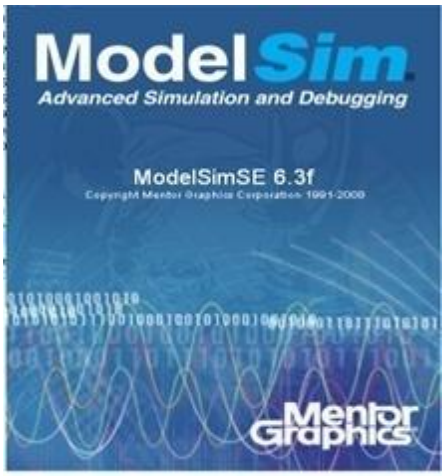


Figure VI.44 . Simulateur Modelsim

Dans Modelsim pour faire la simulation, il faut suivre les étapes ci-dessous :

Modelsim> C : Pour Compiler

Modelsim> S : Pour afficher les composants

Modelsim> w : pour afficher les chronogrammes.

Modelsim> run 4 ms : Pour simuler a 4ms

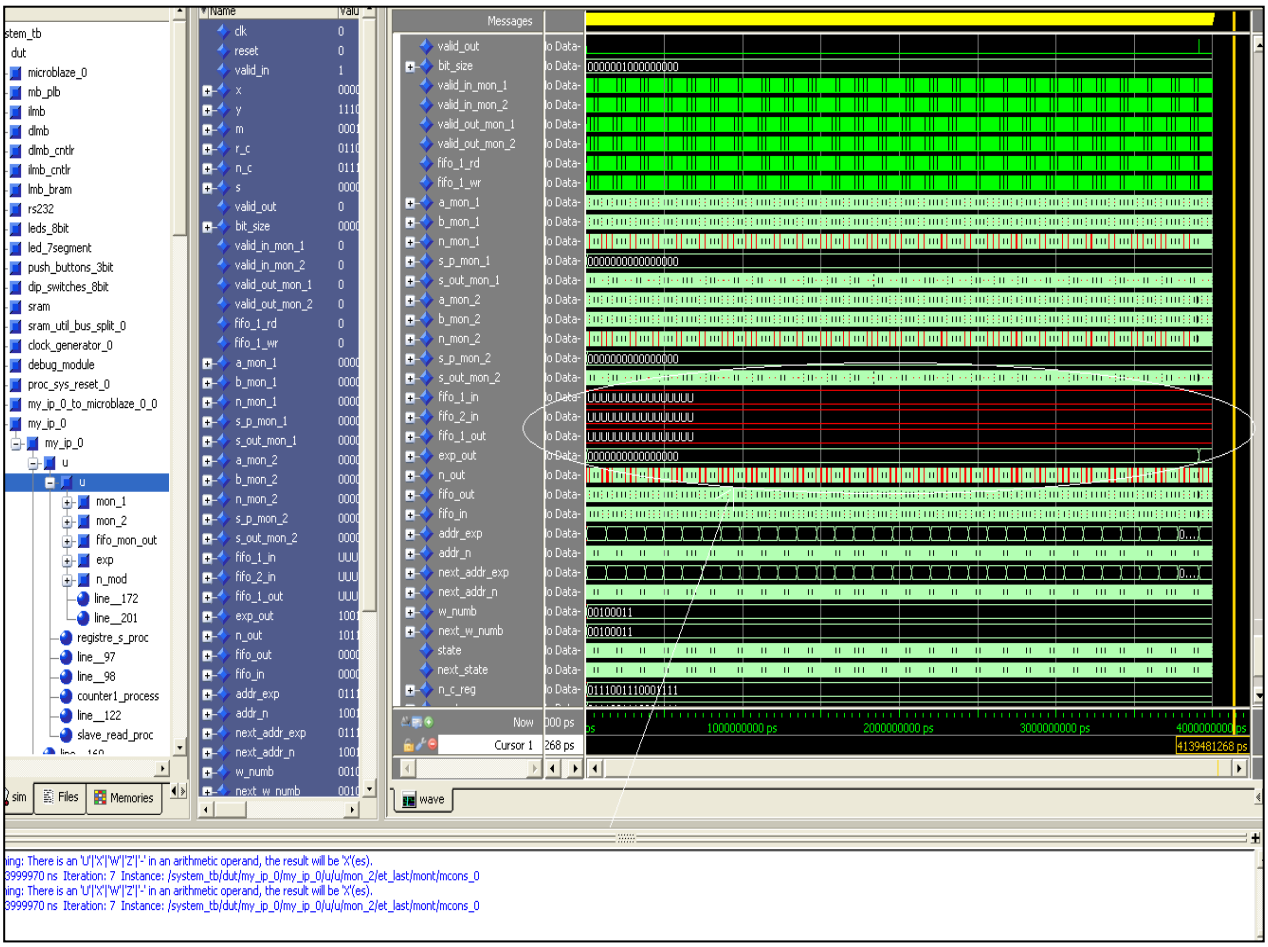


Figure VI.45 . Résultat de la simulation de l'IP-Core sous EDK Microblaze

Aprés 4 ms seconde, on ne retrouve pas la sortie S. Le problème réside dans le contrôle et paramétrage des Blocs Fifos et mémoires (Cerclées en rouge dans la figure VI.45), pendant les opérations de lecture et écriture.

Le rsa_top dans sa configuration en VHDL est conçu de tel sorte qu'à chaque top d'horloge, un mot de 16 bits constituant les données d'entrées (X, Y et le résultats intermédiaires) sont stockées dans la mémoire. Par contre, le process se déroule autrement sous EDK microblaze. Pour le contrôle du core rsa_top, les données d'entrées (X, Y, résultats intermédiaires) sont fournies mots par mots sur plusieurs tops d'horloges (et non pas à chaque top d'horloge). Ce qui a n'a pas permis d'obtenir les résultats de simulation. Et par conséquent de générer le fichier ''bitstream'' dans la carte de développement ciblé.

VI.6.6. Conclusion

Nous avons présenté dans ce chapitre l'outil de développement de notre application sous EDK Microblaze. On a abouti à intégrer note IP Core rsa_top dans un système embarqué, System en chip (Soc), en utilisant les outils du logiciel XPS. L'interfaçage de Microblaze avec l'IP-Core à travers les bus FSL a été réalisé avec succès. Les résultats de simulation n'ont pas été obtenus à cause du probléme du top d'horloge qui s'est présenté dans le fonctionnement du processeur Microblaze. Autrement dit, les instructions de notre programme C/C+ (qui est stoqué dans la BRAM) prennent plusieurs tops d'horloge pour s'éxcécuter, par contre, rsa_top utilise un seul top d'hologe pour traiter un paquet de 16 bits.

Pour remédier à ce problème de top d'horloge présenté dans EDK, des solutions ont été proposées à ce propos :

1) Mettre en place un circuit d'entrée : Fifo (circuit de contrôle écrit en vhdl), comme interface entre le microblaze et l'IP-Core ; qui va jouer le rôle d'un régulateur. Il permettras de recevoir et stocker les données d'entrées provenant de Microblaze, et par la suite les transférer vers l'IP-Core à chaque top d'horloge.
   Mettre en place un autre circuit de sortie : Fifo (circuit de contrôle : écrit en vhdl), qui va jouer le rôle d'un régulateur aussi. Il recevras la sortie de l'IP-Core (et qui est délivré a chaque top d'horloge) et par la suite les transférer au microblaze sur plusieur tops d'horloges.

2) Diminuer l'horloge de L'IP-Core (car il est trés rapide par rapport a microblaze). Si une instruction microblaze prend 10 tops d'horloge pour s' éxcécuter, l'horloge de notre IP doit étre fixé à une période de taille égale à 10 fois celle de microblaze pour le rendre plus long que notre IP Core, pour le synchroniser avec microblaze et pour que les données rentre à chaque top dhorloge.

Conclusion Générale

Ce projet a pour but l' étude et mise en ouvre de l'IP-Core rsa_top sur FPGA et son adaptation à une solution Soc.

Comme nous l'avons montré dans nos précédents chapitres, l'algorithme RSA présente l'avantage d'une cryptanalyse plus difficile. Cependant, les temps de calcul restent longs si on envisage son utilisation dans un système nécessitant des traitements sur des flux de données.

Pour y remédier, deux orientations sont possibles. Elles peuvent être complémentaires. La première orientation est la recherche d'algorithme introduisant des simplifications au niveau de calcul tenant compte de la spécificité des opérateurs à mettre en œuvre. C'est ainsi que dans le cas de RSA, l'algorithme de Montgomery apporte un gain appréciable dans le calcul de l'exponentiation modulaire. Ce qui a retenu notre choix.

La méthode binaire MSB est celle qui s'adapte le mieux à notre champ d'étude car elle nous permet de calculer les grands exposants. Elle nous offre la possibilité de calculer en parallèle la multiplication et l'élévation au carré donc il y a une optimisation du temps d'éxécution de 50%, de plus elle est simple à réaliser.

Après que les choix des méthodes d'exponentiation et de multiplication aient été fixés, l'architecture réalisant l'exponentiation modulaire binaire basée sur la multiplication de Montgomery a été conçue. Celle-ci est basée sur une structure parallèle qui exécute deux multiplications modulaires en même temps ce qui a permis de réduire les temps de chiffrage/déchiffrage des messages.

Seulement que le problème majeur de RSA réside dans la factorisation d'un très grand nombre en deux nombres premiers, qui est devenue presque impossible au fur et à mesure que la taille de la clé augmente. Pour pallier à ce problème, nous avons mis en œuvre la bibliothèque GMP qui était extrêmement indispensable afin de gérer les grands nombres et le problème de factorisation de RSA. L'intégration de la bibliothèque GMP dans l'implémentation de l'algorithme RSA a apporté une grande optimisation sur les performances

de l'algorithme qui ont été déduites lors de la compilation sous Linux et donner des résultats considérables au temps d'exécution de la machine.

Afin de comparer les résultats obtenus précédemment, on a opté pour l'étude de l'IP-Core rsa_top_512 et son extension vers rsa_top_1024 et son implémentation sur la carte de développement Virtex_5 dans le but de faire une étude comparative sur les résultats de simulation et d'implémentation des deux cores sur ISE Xilinx.

Les résultats acquis ont été idéals et meilleurs, par rapport a ceux obtenus a partir de l'algorithme RSA a base de GMP. A cause de l'implémentation hardware du core rsa_top basé sur les algorithmes accélérateurs mise en œuvre.

Afin d'optimiser encore les résultats précédants, L'implémentation du core rsa_top a été intégrée au sein du systèmes embarqués EDK Microblaze qui a présenté des problèmes lors de la simulation comme le montre le tableau ci-dessous.

Bit_size	*Temps de Compilation Sous Linux Ubuntu(**us**)*	*Temps de simulation Fonctionnelle (**ns**) ISE*	*Temps de simulation Temporelle (**us**) ISE*	*Temps de Simulation Sous EDK Microblaze*
512	***70700***	***195295.4***	***1. 846***	*/*
1024	***77626***	***759199, 4***	***6.846***	*/*

Conclusion Générale

Ce projet a pour but l' étude et mise en ouvre de l'IP-Core rsa_top sur FPGA et son adaptation à une solution Soc.

Comme nous l'avons montré dans nos précédents chapitres, l'algorithme RSA présente l'avantage d'une cryptanalyse plus difficile. Cependant, les temps de calcul restent longs si on envisage son utilisation dans un système nécessitant des traitements sur des flux de données.

Pour y remédier, deux orientations sont possibles. Elles peuvent être complémentaires. La première orientation est la recherche d'algorithme introduisant des simplifications au niveau de calcul tenant compte de la spécificité des opérateurs à mettre en œuvre. C'est ainsi que dans le cas de RSA, l'algorithme de Montgomery apporte un gain appréciable dans le calcul de l'exponentiation modulaire. Ce qui a retenu notre choix.

La méthode binaire MSB est celle qui s'adapte le mieux à notre champ d'étude car elle nous permet de calculer les grands exposants. Elle nous offre la possibilité de calculer en parallèle la multiplication et l'élévation au carré donc il y a une optimisation du temps d'éxécution de 50%, de plus elle est simple à réaliser.

Après que les choix des méthodes d'exponentiation et de multiplication aient été fixés, l'architecture réalisant l'exponentiation modulaire binaire basée sur la multiplication de Montgomery a été conçue. Celle-ci est basée sur une structure parallèle qui exécute deux multiplications modulaires en même temps ce qui a permis de réduire les temps de chiffrage/déchiffrage des messages.

Seulement que le problème majeur de RSA réside dans la factorisation d'un très grand nombre en deux nombres premiers, qui est devenue presque impossible au fur et à mesure que la taille de la clé augmente. Pour pallier à ce problème, nous avons mis en œuvre la bibliothèque GMP qui était extrêmement indispensable afin de gérer les grands nombres et le problème de factorisation de RSA. L'intégration de la bibliothèque GMP dans l'implémentation de l'algorithme RSA a apporté une grande optimisation sur les performances de l'algorithme qui ont été déduites lors de la compilation sous Linux et donner des résultats considérables au temps d'exécution de la machine.

Afin de comparer les résultats obtenus précédemment, on a opté pour l'étude de l'IP-Core rsa_top_512 et son extension vers rsa_top_1024 ainsi que son implémentation sur la carte de développement Virtex_5 dans le but de faire une étude comparative sur les résultats de simulation et d'implémentation des deux cores sur ISE Xilinx.

Les résultats acquis ont été idéals et meilleurs, par rapport a ceux obtenus a partir de l'algorithme RSA a base de GMP. A cause de l'implémentation hardware du core rsa_top basé sur les algorithmes accélérateurs mise en œuvre.

Afin d'optimiser encore les résultats précédants, L'implémentation du core rsa_top a été intégrée au sein du systèmes embarqués EDK Microblaze qui a présenté des problèmes lors de la simulation comme le montre le tableau ci-dessous.

Bit_size	*Temps de Compilation Sous Linux Ubuntu(**us**)*	*Temps de simulation Fonctionnelle (**ns**) ISE*	*Temps de simulation Temporelle (**us**) ISE*	*Temps de Simulation Sous EDK*
512	***70700***	***195295.4***	***1. 846***	*/*
1024	***77626***	***759199, 4***	***6.846***	*/*

Bibliographie

[1] Douglas Stinson, Cryptographie Théorie et Pratique, International Thomson Publishing France, PARIS, 1996.

[2] Alfred J. Menezes, Paul C. van Oorschot et Scott A. Vanstone, "Handbook of Applied Cryptography", by CRC Press, 1996, 816 pages, ISBN: 0-8493-8523-7.

[3] H.Bessalah, N.Anane, M.Issad, K.messaoudi, " Cryptographie à clé publique : RSA, ECC", Juillet 07, rapport de recherche, CDTA.

[4] Thomas Plantard "Arithmétique modulaire pour la Cryptographie" thèse de PHD, Université de Montpellier II, 2005.

[5] P. Fournaris, O. Koufopavlou, "A New RSA Encryption Architecture and Hardware Implementation based on Optimized Montgomery Multiplication*" in proceedings of* 2005 *IEEE International Symposium on Circuits and Systems(ISCAS* 2005*)*, Kobe, May 23 -26, Japan, 2005.

[6] Chinuk Kim, "VHDL Implementation of Systolic Modular Multiplication on RSA Cryptosystem" Master of Science (Computer Science) at The City College of the City University

[7] V. Bunim, M. Schimmler, B.Tolg "A Complexity-Effective Version of Montgomery's Algorihm" Institute for Computer Engineering and Communication Networks, Technical University of Braunschweig, Germany. 2003.

[8] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signature and Public-key Cryptosystems*,"Comm. ACM,* 21(2), 120-126, 1978.

[9] P. L. Montgomery, "Modular Multiplication without Trial Division," *Math. Computation, vol.* 44, *pp.*519-521, 1985.

[10] C. McIvor, M. McLoone, J.V. McCanny. "FPGA Montgomery Multiplier Architectures - a Comparison", 12th IEEE Symposium on Field-Programmable Custom Computing Machines (*FCCM*

[11] Ç. K. Koc, T. Acar, B. S. Kaliski: "Analyzing and Comparing Montgomery Multiplication Algorithms". *IEEE Micro, Vol.* 16,*No.* 3, *pp.* 26-33, June 1996.

[12] N. Mentens, K. Sakiyama, L.Batina, I. Verbauwhede, B.Preneel, "FPGA-Oriented Secure Data Path Design Implementation of a Public Key Coprocessor", *FPL* 2006,IEEE, pp. 133-138, 2006.

[13] Djamakebir Nawel, Seksaoui Yasmine, M.Issad, K.messaoudi. ''Implémentation de l'exponentiation modulaire du protocole RSA sur circuit FPGA '', Université de Blida

[14] C. McIvor, M. Mcloone, J. N. McCanny, A. Daly, W.Marnane, "Fast Montgomery Modular Multiplication and RSA Cryptographic Processor Architectures", 37*th Annual Asilomar Conference on*

[15] A. Daly and W. Marnane, "Efficient Architectures for implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic", *in proc. of 10th International symposium on FPGA's*, 2002.

[16] Ersin Öksüzoğlu, Erkay Savaş, ''Parametric, Secure and Compact Implementation of RSA on FPGA'' *Sabanci University, Istanbul, TURKEY*

[17] S. B. Ors, L. Batina, B. Preneel and J. Vandawalle, “Hardware Implementation of a Montgomery Modular Multiplier in a Systolic Array”, *International Parallel and Distributed processing symposium (IPDPS '*03*)*, 2003.

[18] S. H. Tang, K. S. Tsui, P. H. W. Leong, “Modular Exponentiation using Parallel Multipliers” , *Proc of the 2003IEEE International Conference on Field ProgrammableTechnology (FTP* 2003*), pp. 52-59* (2003).

[19] R.Beguenane,J.L.Beuchat, J.M.Muller, S.Simard, “Modular Multiplication of Large Integers on FPGA” , INRIA 5668, 2005

[20] Rajorshi Biswas, Shibdas Bandyopadhyay, Anirban Banerjee ,A fast implementation of the RSA algorithm using the GNU MP library, Indian Institute of Information Technology, Calcutta

[21] Daniel M. Gordon, *A survey of fast exponentiation methods* Journal of algorithms, 27, 1998, 126-146.

[22] Alvaro Bernal Noréna, "Conception et Etude d’une architecture Numérique de haute performance pour le calcul de la fonction exponentielle modulaire", Thèse de doctorat de l’INPG Octobre 1999.

[23] N.Takagi, “A radix-4 modular multiplication algorithm for modular exponentiation”, IEEE Transactions on computers, 41(8): 949-956, August 1992.

[24] P.L.Montgomery, Modular Multiplication without trial division, Mathematics of computation, 44 (170): 519-521, Apr 1985.

[25] VHDL Du langage au circuit, du circuit au langage par J.Weber et M.Meaudre - Masson

[26] Y. Huang, W. T.Cheng, « Using Embedded Infrastructure IP for SOC Post-Silicon Verification», DAC 2003, June 2-6, 2003, Anaheim, California, USA.

[27] Reuse Methodology Manual For System On Chip Designs, Michael Keating and Pierre Bricaud – Kluwer Academic Publishers 1999

[28] D. Suzuki ,’’EDK 9.2 MicroBlaze Tutorial in Virtex-5’’ *CHES* 2007, *LNCS* 4727, *pp.*272-288, 2007.

[29] “EDK Concepts, Tools, and Techniques”, Embedded Development Kit 9.2i, Mai 2007.

[30] “Fast Simplex Link (FSL) Bus (v2.11a)”, Product Specification, Juin 2007.

[31] Hans-Peter Rosinger, “Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link (FSL) Channel”, XAPP529, Mai 2004.

Webographie:

[33] Newsgroup : comp.lang.vhdl

[34] http://www.xilinx.com

[35] http://www.xilinx.com/ *User Guide*: Virtex-5 LXT/SXT/FXT FPGA Prototype

[36] http://www.opencores.org

[38] http://www-rocq.inria.frwhoAnne.Canteautcrypto_modern.pdf

[39] http://www.crypto.freezee.org/crypto/cours.html

[40] http://www.rsa.com/

Annexe

Annexe A

L'algorithme 1 : RSA en langage C (étudié)

```
#include <stdio.h>
#include <gmp.h>
#include <string.h>
#include <stdlib.h>

typedef struct{
  mpz_t e;
  mpz_t n;
}public_key;

typedef struct{
  mpz_t d;
  mpz_t n;
}private_key;

/*****************************************************************/
void init_public_key(public_key *puKey, int *nbr_car)
{
  FILE *fp;
  char data[1000];
  fp=fopen("public_key","r");
  fscanf(fp,"e=%s\n",data);
  mpz_init_set_str(puKey->e,data,16);
  data[0]='\0';
  fscanf(fp,"n=%s\n",data);
  mpz_init_set_str(puKey->n,data,16);
  fscanf(fp,"nbr_car=%d\n",nbr_car);
  fclose(fp);
}
/*****************************************************************/
void init_private_key(private_key *prKey,int *nbr_car)
{
  FILE *fp;
  char data[1000];
  fp=fopen("private_key","r");
  fscanf(fp,"d=%s\n",data);
  mpz_init_set_str(prKey->d,data,16);
  data[0]='\0';
  fscanf(fp,"n=%s\n",data);
  mpz_init_set_str(prKey->n,data,16);
  fscanf(fp,"nbr_car=%d\n",nbr_car);
  fclose(fp);
}
/*****************************************************************/
void clear_public(public_key *puKey)
{
  mpz_clear(puKey->e);
  mpz_clear(puKey->n);
}
/*****************************************************************/
void clear_private(private_key *prKey)
{
```

```
  mpz_clear(prKey->d);
  mpz_clear(prKey->n);
}
/*****************************************************************/
void crypter(public_key puKey,int nbr_oct_cr)
{
  FILE *fp,*fp_cr;
  char* buffer;
  char buffer_2[1000];
  mpz_t x,y;
  int n;

  mpz_init(y);
  buffer=(char*)malloc((nbr_oct_cr)*sizeof(char));
  fp=fopen("fichier_a_crypter","r");
  fp_cr=fopen("fichier_crypte","w");
  while(!feof(fp))
    {
      n=(int)fread(buffer,sizeof(char), nbr_oct_cr,fp);
      if(n!=1)
 {
   if(n<nbr_oct_cr)
     {
       while(n<=nbr_oct_cr)
       {
         buffer[n-1]='0';
         n++;
       }
     }
   buffer[n]='\0';
   mpz_init_set_str(x,buffer,16);
   mpz_powm (y, x, puKey.e, puKey.n);
   mpz_get_str (buffer_2, 16, y);
   fprintf(fp_cr,"%s",buffer_2);
 }
    }
  fclose(fp_cr);
  fclose(fp);
  free(buffer);
  mpz_clear(x);
  mpz_clear(y);
}
/*****************************************************************/
void decrypter(private_key prKey,int nbr_oct_dec)
{
  FILE *fp,*fp_dec;
  char* buffer;
  char buffer_2[1000];
  mpz_t x,y;
  int n;

  mpz_init(y);
  buffer=(char*)malloc((nbr_oct_dec)*sizeof(char));
  fp=fopen("fichier_a_decrypter","r");
  fp_dec=fopen("fichier_decrypte","w");
  while(!feof(fp))
    {
      n=(int)fread(buffer,sizeof(char), nbr_oct_dec,fp);
      if(n!=1)
 {
   if(n<nbr_oct_dec)
```

```
      {
        while(n<=nbr_oct_dec)
        {
          buffer[n-1]='0';
          n++;
        }
      }
    buffer[n]='\0';
    mpz_init_set_str(x,buffer,16);
    mpz_powm (y, x, prKey.d, prKey.n);
    mpz_get_str (buffer_2, 16, y);
    fprintf(fp_dec,"%s",buffer_2);
 }
     }
  fclose(fp_dec);
  fclose(fp);
  free(buffer);
  mpz_clear(x);
  mpz_clear(y);
}
/*****************************************************************/
int main(void)
{
  public_key puKey;
  private_key prKey;
  int nbr_car;
  char reponse;
  struct timeval tempo1, tempo2;

  long elapsed_utime;
  long elapsed_mtime;
  long elapsed_seconds;
  long elapsed_useconds;

  printf("\nCryptage / Decryptage RSA:\n");
  printf("[1] : Cryptage\n");
  printf("[2] : Decryptage\n");
  printf("[3] : Quitter\n");
  printf("Entrez votre choix : ");
  reponse=getchar();
  switch(reponse)
    {
    case '1': {
      init_public_key(&puKey,&nbr_car);
      gettimeofday(&tempo1, NULL);
      crypter(puKey,nbr_car*2);
      gettimeofday(&tempo2, NULL);
      elapsed_seconds  = tempo2.tv_sec  - tempo1.tv_sec;
      elapsed_useconds = tempo2.tv_usec - tempo1.tv_usec;
      elapsed_utime = (elapsed_seconds) * 1000000 + elapsed_useconds;
      printf("\nTemps du cryptage = %ld microsecondes\n", elapsed_utime);
      clear_public(&puKey);
   }
      break;
    case '2':{
      init_private_key(&prKey,&nbr_car);
      gettimeofday(&tempo1, NULL);
      decrypter(prKey,nbr_car*2);
      gettimeofday(&tempo2, NULL);
      elapsed_seconds  = tempo2.tv_sec  - tempo1.tv_sec;
```

```
      elapsed_useconds = tempo2.tv_usec - tempo1.tv_usec;
      elapsed_utime = (elapsed_seconds) * 1000000 + elapsed_useconds;
      printf("\nTemps du decryptage = %ld microsecondes\n", elapsed_utime);
      clear_private(&prKey);
    }
      break;
    case '3':break;
    default:printf("Erreur de saisie!!\n");
    }
  return 0;
}
```

L'algorithme 2 : RSA en langage C (implémenté)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <gmp.h>
#include <sys/time.h>


#define BITSTRENGTH  1024               /* size of modulus (n) in bits */
#define PRIMESIZE    (BITSTRENGTH / 2)  /* size of the primes p and q  */


/* Declare global variables */

mpz_t d,e,n;
mpz_t M,c;


/* Declare time-related variables */

struct timeval tv1,tv2;
struct timeval tvdiff;
struct timezone tz;


/* Declare core routines */

void RSA_generateKeys();
int  RSA_checkKeys();
void RSA_encrypt();
void RSA_decrypt();


/* Initialization related routines */

void initializeGMP();
void clearGMP();
void initializeRandom();


/* Timing routine */
void timediff(struct timeval*,struct timeval*,struct timeval*);
```

```
/* Helper routines */
inline void process(char*);
inline void encrypt(char*,FILE*);



/* Main subroutine */
int main()
{
    /* Initialize the GMP integers first */
    initializeGMP();

    /*
    *  Check existence of key files : ~/.rsapublic
    *  and ~/.rsaprivate else generate new keys and file
    */

    if(!RSA_checkKeys())
    {
        printf("Creating new RSA Key Files...\n\n");
        RSA_generateKeys();
    }

    /* Show menu */
    int choice = -1;

    while(choice!=3)
    {
        printf("\n\n");

        printf("\n1. Encrypt... ");
        printf("\n2. Decrypt... ");
        printf("\n3. Quit... ");

        printf("\n\nEnter your choice (1-3) : ");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1  :   RSA_encrypt();
                        break;

            case 2  :   RSA_decrypt();
                        break;

            case 3  :   return(0);

            default :   printf("\nUndefined choice.");
        }
    }

    /* Clear the GMP integers */
    clearGMP();

    return 0;
}



void initializeGMP()
{
```

```
    /* Initialize all the GMP integers once and for all */

    mpz_init(d);
    mpz_init(e);
    mpz_init(n);

    mpz_init(M);
    mpz_init(c);
}



void clearGMP()
{
    /* Clean up the GMP integers */

    mpz_clear(d);
    mpz_clear(e);
    mpz_clear(n);

    mpz_clear(M);
    mpz_clear(c);
}



void initializeRandom()
{
    /* This initializes the random number generator */

    /* sleep for one second (avoid calls in the same second) */
    sleep(1);

    /* Set seed for rand() by system time() ... */
    unsigned int time_elapsed;
    time((time_t*)&time_elapsed);
    srand(time_elapsed);
}



void timediff(struct timeval* a,struct timeval* b,struct timeval* result)
{
    /* This function calculates and returns the time
    *  difference between two timeval structs
    */

    (result)->tv_sec  = (a)->tv_sec  - (b)->tv_sec;
    (result)->tv_usec = (a)->tv_usec - (b)->tv_usec;

    if((result)->tv_usec < 0)
    {
        --(result)->tv_sec;
        (result)->tv_usec += 1000000;
    }
}



int RSA_checkKeys()
{
```

```
    /* This function checks whether the keys exist
    *  in the file ~/.rsaprivate and ~/.rsapublic
    */

    char publicFile[100];
    char privateFile[100];

    strcpy(publicFile,getenv("HOME"));
    strcpy(privateFile,getenv("HOME"));

    strcat(publicFile,"/.rsapublic");
    strcat(privateFile,"/.rsaprivate");

    FILE* fpublic  = fopen(publicFile,"r");
    FILE* fprivate = fopen(privateFile,"r");

    If ((!fpublic) || (!fprivate))
    {
        /* Key files do not exist */
        return 0;
    }


    printf("\nUsing RSA Key Files : \n\n");
    printf("\nPublic Key File  : %s",publicFile);
    printf("\nPrivate Key File : %s",privateFile);

    char d_str[1000];
    char e_str[100];
    char n_str[1000];


    /* Get keys */
    fscanf(fpublic,"%s\n",e_str);
    fscanf(fpublic,"%s\n",n_str);

    fscanf(fprivate,"%s\n",d_str);

    mpz_set_str(d,d_str,10);
    mpz_set_str(e,e_str,10);
    mpz_set_str(n,n_str,10);

    fclose(fpublic);
    fclose(fprivate);

    return 1;
}



void RSA_generateKeys()
{
    /* This function creates the keys. The basic algorithm is...
    *
    *  1. Generate two large distinct primes p and q randomly
    *  2. Calculate n = pq and x = (p-1)(q-1)
    *  3. Select a random integer e (1<e<x) such that gcd(e,x) = 1
    *  4. Calculate the unique d such that ed = 1(mod x)
    *  5. Public key pair : (e,n), Private key pair : (d,n)
    *
    */
```

```
/* initialize random seed */
initializeRandom();

/* first, record the start time */
if(gettimeofday(&tv1,&tz)!=0)
    printf("\nWarning : could not gettimeofday() !");

/*
*  Step 1 : Get two large (512 bits) primes.
*/

mpz_t p,q;

mpz_init(p);
mpz_init(q);

char* p_str = new char[PRIMESIZE+1];
char* q_str = new char[PRIMESIZE+1];

p_str[0] = '1';
q_str[0] = '1';

for(int i=1;i<PRIMESIZE;i++)
    p_str[i] = (int)(2.0*rand()/(RAND_MAX+1.0)) + 48;

for(int i=1;i<PRIMESIZE;i++)
    q_str[i] = (int)(2.0*rand()/(RAND_MAX+1.0)) + 48;

p_str[PRIMESIZE] = '\0';
q_str[PRIMESIZE] = '\0';

mpz_set_str(p,p_str,2);
mpz_set_str(q,q_str,2);

mpz_nextprime(p,p);
mpz_nextprime(q,q);

mpz_get_str(p_str,10,p);
mpz_get_str(q_str,10,q);

printf("Random Prime 'p' = %s\n",p_str);
printf("Random Prime 'q' = %s\n",q_str);

/*
*  Step 2 : Calculate n (=pq) ie the 1024 bit modulus
*  and x (=(p-1)(q-1)).
*/

char n_str[1000];

mpz_t x;

mpz_init(x);


/* Calculate n... */

mpz_mul(n,p,q);

mpz_get_str(n_str,10,n);
```

```
printf("\nn = %s\n",n_str);


/* Calculate x... */

mpz_t p_minus_1,q_minus_1;

mpz_init(p_minus_1);
mpz_init(q_minus_1);

mpz_sub_ui(p_minus_1,p,(unsigned long int)1);
mpz_sub_ui(q_minus_1,q,(unsigned long int)1);

mpz_mul(x,p_minus_1,q_minus_1);


/*
*  Step 3 : Get small odd integer e such that gcd(e,x) = 1.
*/

mpz_t gcd;
mpz_init(gcd);

/*
*  Assuming that 'e' will not exceed the range
*  of a long integer, which is quite a reasonable
*  assumption.
*/

unsigned long int e_int = 65537;

while(true)
{
    mpz_gcd_ui(gcd,x,e_int);

    if(mpz_cmp_ui(gcd,(unsigned long int)1)==0)
        break;

    /* try the next odd integer... */
    e_int += 2;
}

mpz_set_ui(e,e_int);


/*
*  Step 4 : Calculate unique d such that ed = 1(mod x)
*/


char d_str[1000];

if(mpz_invert(d,e,x)==0)
{
    printf("\nOOPS : Could not find multiplicative inverse!\n");
    printf("\nTrying again...");
    RSA_generateKeys();

}

mpz_get_str(d_str,10,d);
```

```
printf("\n\n");

/*
*  Print the public and private key pairs...
*/

printf("\nPublic Keys (e,n): \n\n");
printf("\nValue of 'e' : %ld",e_int);
printf("\nValue of 'n' : %s ",n_str);

printf("\n\n");

printf("\nPrivate Key : \n\n");
printf("\nValue of 'd' : %s",d_str);

/* get finish time of key generation */
if(gettimeofday(&tv2,&tz)!=0)
    printf("\nWarning : could not gettimeofday() !");

timediff(&tv2,&tv1,&tvdiff);

printf("\nKey Generation took (including I/O) ...\n");
printf("\n%-15s : %ld","Seconds",tvdiff.tv_sec);
printf("\n%-15s : %ld","Microseconds",tvdiff.tv_usec);


/* Write values to file $HOME/.rsapublic and $HOME/.rsaprivate */

char publicFile[100];
char privateFile[100];

strcpy(publicFile,getenv("HOME"));
strcpy(privateFile,getenv("HOME"));

strcat(publicFile,"/.rsapublic");
strcat(privateFile,"/.rsaprivate");

FILE* fpublic  = fopen(publicFile,"w");
FILE* fprivate = fopen(privateFile,"w");

if((!fpublic) || (!fprivate))
{
    fprintf(stderr,"FATAL: Could not write to RSA Key Files!");
    exit(1);
}

/* Write ~/.rsapublic */
fprintf(fpublic,"%ld\n",e_int);
fprintf(fpublic,"%s\n",n_str);

/* Write ~/.rsaprivate */
fprintf(fprivate,"%s\n",d_str);

fclose(fpublic);
fclose(fprivate);

printf("\nWrote RSA Key Files ...\n");
printf("\nPublic Key File  : %s",publicFile);
printf("\nPrivate Key File : %s",privateFile);
```

```
    /* clean up the gmp mess */
    mpz_clear(p);
    mpz_clear(q);
    mpz_clear(x);
    mpz_clear(p_minus_1);
    mpz_clear(q_minus_1);
    mpz_clear(gcd);

}



void RSA_encrypt()
{
    /* The RSA Encryption routine */


    printf("RSA Encryption :\n\n");

    char pubkeyfile[200];   /* file containing public key */
    char infile[200];    /* filename to encrypt */
    char outfile[200];    /* filename to decrypt */
    FILE *fin,*fout;    /* file pointers */
    FILE *fpublic;        /* file pointer to public keyfile */

    int num;        /* number of characters
                    to encrypt together */

    int i;            /* string index */
    char chread;        /* character read */
    char stread[1000];    /* string read */

    char e_str[100],n_str[1000];

    /* Get public keys of recipient */
    printf("\nEnter file containing public key of recipient");
    printf("\nType '.' to use your own public key : ");

    strcpy(pubkeyfile,"");
    scanf("%s",pubkeyfile);

    if(strcmp(pubkeyfile,"."))
    {
        fpublic = fopen(pubkeyfile,"r");

        if(!fpublic)
        {
            fprintf(stderr,"FATAL: Could not read %s!",pubkeyfile);
            return;
        }

        fscanf(fpublic,"%s\n",e_str);
        fscanf(fpublic,"%s\n",n_str);

        mpz_set_str(e,e_str,10);
        mpz_set_str(n,n_str,10);

    }
    else
        printf("\nWARNING : Encrypting using your own public key!\n");
```

```
printf("\nEnter filename to encrypt :");
scanf("%s",infile);

fin = fopen(infile,"r");

if(!fin)
{
    fprintf(stderr,"FATAL : Could not open %s for reading",infile);
    return;
}

printf("Enter filename to encrypt to :");
scanf("%s",outfile);

fout = fopen(outfile,"w");

if(!fout)
{
    fprintf(stderr,"FATAL : Could not open %s for writing",outfile);
    return;
}

printf("Enter number of characters to encrypt together :");
scanf("%d",&num);

if(num<1 || num>100)
{
    /* safest numbers are in the range 1-100 for 1024 bit RSA */

    if(num<1)
    {
        fprintf(stderr,"Invalid input!");
        return;
    }
    else
        fprintf(stderr,"WARNING : Possibly out of range!");
}


/* Get time before encryption */
if(gettimeofday(&tv1,&tz)!=0)
    fprintf(stderr,"\nWARNING : could not gettimeofday() !");


i = 0;
chread = 'a';

do
{

    chread = fgetc(fin);

    if(chread==EOF)
    {
        if(i!=0)
        {
            stread[i] = '\0';
            encrypt(stread,fout);
        }
    }
    else if(i==num-1)
```

```
        {
            stread[i] = chread;
            stread[i+1] = '\0';
            encrypt(stread,fout);
            i = 0;

        }
        else
        {
            stread[i] = chread;
            i++;
        }

    }while(chread!=EOF);

    /* Get time after encryption */
    if(gettimeofday(&tv2,&tz)!=0)
        printf("\nWarning : could not gettimeofday() !");

    timediff(&tv2,&tv1,&tvdiff);

    printf("\nEncryption took...\n");

    printf("\n%-15s : %ld","Seconds",tvdiff.tv_sec);
    printf("\n%-15s : %ld","Microseconds",tvdiff.tv_usec);

    fclose(fin);
    fclose(fout);
}



inline void encrypt(char* msg,FILE* fout)
{
    /* This function actually does the encrypting of each message */

    unsigned int i;
    int tmp;
    char tmps[4];
    char* intmsg = new char[strlen(msg)*3 + 1];


    /* Here, (mpz_t) M is the messsage in gmp integer
    *  and (mpz_t) c is the cipher in gmp integer */

    char ciphertext[1000];

    strcpy(intmsg,"");

    for(i=0;i<strlen(msg);i++)
    {
        tmp = (int)msg[i];

        /* print it in a 3 character wide format */
        sprintf(tmps,"%03d",tmp);

        strcat(intmsg,tmps);
    }

    mpz_set_str(M,intmsg,10);
```

```
    /* free memory claimed by intmsg */
    delete [] intmsg;

    /* c = M^e(mod n) */
    mpz_powm(c,M,e,n);

    /* get the string representation of the cipher */
    mpz_get_str(ciphertext,10,c);

    /* write the ciphertext to the output file */
    fprintf(fout,"%s\n",ciphertext);
}



void RSA_decrypt()
{
    /* The RSA decryption routine */

    printf("RSA Decryption:\n\n");

    char file[200];         /* filename of file to decrypt */
    FILE* fp;         /* file pointer to decrypt */
    char ciphertext[1000];    /* ciphertext */

    /* Here, (mpz_t) c is the cipher in gmp integer
    *  and (mpz_t) M is the message in gmp integer */

    char decrypted[1000];    /* decypted text */

    printf("Enter name of file to decrypt :");
    scanf("%s",file);


    fp = fopen(file,"r");

    if(!fp)
    {
        fprintf(stderr,"FATAL : Could not open %s for reading",file);
        return;
    }

    /* Get time before decryption */
    if(gettimeofday(&tv1,&tz)!=0)
        printf("\nWarning : could not gettimeofday() !");

    while(fscanf(fp,"%s\n",ciphertext)>0)
    {
        mpz_set_str(c,ciphertext,10);

        /* M = c^d(mod n) */
        mpz_powm(M,c,d,n);

        mpz_get_str(decrypted,10,M);

        process(decrypted);
    }

    /* Get time after decription */
    if(gettimeofday(&tv2,&tz)!=0)
```

```
        printf("\nWarning : could not gettimeofday() !");

    timediff(&tv2,&tv1,&tvdiff);

    printf("\nDecryption took... (including output)\n");
    printf("\n%-15s : %ld","Seconds",tvdiff.tv_sec);
    printf("\n%-15s : %ld","Microseconds",tvdiff.tv_usec);
}



inline void process(char* str)
{
    /* This function shows the decrypted integer
    *  message as an understandable text string
    */

    unsigned int i=0;
    int tmpnum;
    char strmod[1000];

    /* make the message length an integral multiple
    *  of 3 by adding zeroes to the left if required
    */

    if(strlen(str)%3 == 1)
    {
        strcpy(strmod,"00");
        strcat(strmod,str);
    }
    else if(strlen(str)%3 == 2)
    {
        strcpy(strmod,"0");
        strcat(strmod,str);
    }
    else
        strcpy(strmod,str);


    while(i<=strlen(strmod)-3)
    {
        tmpnum = strmod[i] - 48;
        tmpnum = 10*tmpnum + (strmod[i+1] - 48);
        tmpnum = 10*tmpnum + (strmod[i+2] - 48);

        i += 3;

        printf("%c",tmpnum);
    }
}
```

Annexe B

IP-Core RSA _512 (32 mots de 16 bits)

rsa_top.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
library UNISIM;
use UNISIM.VCOMPONENTS.all;


entity rsa_top is
 port(
  clk      : in  std_logic;
  reset    : in  std_logic;
  valid_in : in  std_logic;
  x        : in  std_logic_vector(15 downto 0);
  y        : in  std_logic_vector(15 downto 0);
  m        : in  std_logic_vector(15 downto 0);
  r_c      : in  std_logic_vector(15 downto 0);-- constant de Montgomery:
r^2modm
  n_c      : in  std_logic_vector(15 downto 0); -- constante de la
multiplication de Montgomery
  s        : out std_logic_vector( 15 downto 0);-- s=x^ymodm
  valid_out: out std_logic;
  bit_size : in  std_logic_vector(15 downto 0); -- (log2(y))
    );
 end rsa_top;

architecture Behavioral of rsa_top is


-- Deux Block de Multiplicateur de Montgomery qui fonctionne en même temps
  component montgomery_mult is
                       port(
                          clk       : in  std_logic;
                          reset     : in  std_logic;
                          valid_in  : in  std_logic;
                          a         : in  std_logic_vector(15 downto 0);
                          b         : in  std_logic_vector(15 downto 0);
                          n         : in  std_logic_vector(15 downto 0);
                          s_prev    : in  std_logic_vector(15 downto 0);
                          n_c       : in  std_logic_vector(15 downto 0);
                          s         : out std_logic_vector(15 downto 0);
                          valid_out : out std_logic
);
  end component;


-- Memoire pour stocker l'exposant y et le modulo m
  component Mem_b
    port (
      clka  : in  std_logic;
      wea   : in  std_logic_vector( 0 downto 0);
      addra : in  std_logic_vector( 5 downto 0);
```

```
      dina  : in  std_logic_vector(15 downto 0);
      douta : out std_logic_vector(15 downto 0));
  end component;


-- fifos pour stocker les multiplications intermédiaires/partielles
  component res_out_fifo
    port (
      clk   : in  std_logic;
      rst   : in  std_logic;
      din   : in  std_logic_vector(31 downto 0);
      wr_en : in  std_logic;
      rd_en : in  std_logic;
      dout  : out std_logic_vector(31 downto 0);
      full  : out std_logic;
      empty : out std_logic);
  end component;


  signal valid_in_mon_1, valid_in_mon_2, valid_out_mon_1, valid_out_mon_2,
  fifo_1_rd, fifo_1_wr : std_logic;

  signal a_mon_1, b_mon_1, n_mon_1, s_p_mon_1, s_out_mon_1, a_mon_2,
b_mon_2, n_mon_2, s_p_mon_2, s_out_mon_2, fifo_1_in, fifo_2_in, fifo_1_out,
exp_out, n_out : std_logic_vector(15 downto 0);

  signal fifo_out, fifo_in : std_logic_vector(31 downto 0);

  signal addr_exp, addr_n, next_addr_exp, next_addr_n : std_logic_vector
  (5 downto 0);

  type state_type is (wait_start, prepare_data, wait_constants,
writting_cts_fifo, processing_data_0, processing_data_1, wait_results,
transition, prepare_next, writting_results, final_mult, show_final,
prepare_final, wait_final);
  signal state, next_state               : state_type;
  signal w_numb, next_w_numb             : std_logic_vector(7 downto 0);

--Les signaux de registres
  signal n_c_reg, next_n_c_reg           : std_logic_vector(15 downto 0);

--Comptage des données qui passent par les multiplicateurs
  signal count_input, next_count_input, bit_counter, next_bit_counter :
  std_logic_vector(15 downto 0);

  signal bsize_reg, next_bsize_reg       : std_logic_vector(15 downto 0);
  signal write_b_n                       : std_logic_vector(0  downto 0);


begin


  mon_1 : montgomery_mult port map(
    clk       => clk,
    reset     => reset,
    valid_in  => valid_in_mon_1,
    a         => a_mon_1,
    b         => b_mon_1,
    n         => n_mon_1,
    s_prev    => s_p_mon_1,
    n_c       => n_c_reg,
```

```
  s         => s_out_mon_1,
  valid_out => valid_out_mon_1
  );

mon_2 : montgomery_mult port map(
  clk       => clk,
  reset     => reset,
  valid_in  => valid_in_mon_2,
  a         => a_mon_2,
  b         => b_mon_2,
  n         => n_mon_2,
  s_prev    => s_p_mon_2,
  n_c       => n_c_reg,
  s         => s_out_mon_2,
  valid_out => valid_out_mon_2
  );


fifo_mon_out : res_out_fifo port map (
    clk   => clk,
    rst   => reset,
    din   => fifo_in,
    wr_en => fifo_1_wr,
    rd_en => fifo_1_rd,
    dout  => fifo_out
    );

exp : Mem_b port map (
  clka  => clk,
  wea   => write_b_n,
  addra => addr_exp,
  dina  => y,
  douta => exp_out);

n_mod : Mem_b port map (
  clka  => clk,
  wea   => write_b_n,
  addra => addr_n,
  dina  => m,
  douta => n_out);



process(clk, reset)
begin

  if(clk = '1' and clk'event) then

    if(reset = '1')then
      state       <= wait_start;
      n_c_reg     <= (others => '0');
      w_numb      <= (others => '0');
      count_input <= (others => '0');
      addr_exp    <= (others => '0');
      addr_n      <= (others => '0');
      bit_counter <= (others => '0');
      bsize_reg   <= (others => '0');
    else
      state       <= next_state;
      n_c_reg     <= next_n_c_reg;
      w_numb      <= next_w_numb;
```

```
      count_input <= next_count_input;
      addr_exp    <= next_addr_exp;
      addr_n      <= next_addr_n;
      bit_counter <= next_bit_counter;
      bsize_reg   <= next_bsize_reg;
    end if;
  end if;
end process;

process(state, bsize_reg, n_c_reg, valid_in, x, n_c, r_c, m, y, w_numb,
count_input, addr_exp, addr_n, s_out_mon_1, s_out_mon_2, bit_size,
valid_out_mon_1, bit_counter, exp_out, fifo_out, n_out)

variable mask : std_logic_vector(3 downto 0);

begin

  --Mettre ajour les registres
  next_state       <= state;
  next_n_c_reg     <= n_c_reg;
  next_w_numb      <= w_numb;
  next_count_input <= count_input;
  next_bsize_reg   <= bsize_reg;

  --Les entrées de Montgomerys.
  valid_in_mon_1   <= '0';
  valid_in_mon_2   <= '0';
  a_mon_1          <= (others => '0');
  b_mon_1          <= (others => '0');
  n_mon_1          <= (others => '0');
  a_mon_2          <= (others => '0');
  b_mon_2          <= (others => '0');
  n_mon_2          <= (others => '0');
  s_p_mon_1        <= (others => '0');
  s_p_mon_2        <= (others => '0');

  --Control des fifos
  fifo_1_rd        <= '0';
  fifo_in          <= (others => '0');
  fifo_1_wr        <= '0';

  --Control de mémoires (exposant y et modulo m)
  write_b_n        <= b"0";
  next_addr_exp    <= addr_exp;
  next_addr_n      <= addr_n;
  next_bit_counter <= bit_counter;

  --Les sorties du rsa_top
  valid_out        <= '0';
  s                <= (others => '0');
  case state is

    when wait_start =>

      valid_in_mon_1 <= valid_in;
      valid_in_mon_2 <= valid_in;
      if(valid_in = '1') then
        a_mon_1      <= x;
        b_mon_1      <= r_c;
        n_mon_1      <= m;
```

```
    a_mon_2          <= x"0001";
    b_mon_2          <= r_c;
    n_mon_2          <= m;
    next_w_numb      <= x"23"; --3 mots mise a “0” sont rajoutes,
    afin d’eviter les debordement des multiplieurs/additionneurs
    next_n_c_reg     <= n_c;
    next_state       <= prepare_data;
    next_count_input <= x"0001";
    write_b_n      <= b"1"; -- la valeur de l’exposant « y »
    next_addr_exp  <= "000001";
    next_addr_n    <= "000001";
    next_bsize_reg <= bit_size-1;
  end if;

when prepare_data =>
  next_count_input <= count_input+1;
  valid_in_mon_1   <= '1';
  valid_in_mon_2   <= '1';
  if(valid_in = '1') then
    a_mon_1        <= x;
    b_mon_1        <= r_c;
    n_mon_1        <= m;
    b_mon_2        <= r_c;
    n_mon_2        <= m;
    write_b_n     <= b"1";
    next_addr_exp <= addr_exp+1;
    next_addr_n   <= addr_n+1;
  end if;
  if(count_input = w_numb) then
    next_state    <= wait_constants;
    next_addr_n   <= (others => '0');

    next_addr_exp                          <= bsize_reg(9 downto 4);
    --Decodeur pour etablir le masque
    mask := bsize_reg(3 downto 0);
    case (mask) is
      when "0000"        => next_bit_counter <= "0000000000000001";
      when "0001"        => next_bit_counter <= "0000000000000010";
      when "0010"        => next_bit_counter <= "0000000000000100";
      when "0011"        => next_bit_counter <= "0000000000001000";
      when "0100"        => next_bit_counter <= "0000000000010000";
      when "0101"        => next_bit_counter <= "0000000000100000";
      when "0110"        => next_bit_counter <= "0000000001000000";
      when "0111"        => next_bit_counter <= "0000000010000000";
      when "1000"        => next_bit_counter <= "0000000100000000";
      when "1001"        => next_bit_counter <= "0000001000000000";
      when "1010"        => next_bit_counter <= "0000010000000000";
      when "1011"        => next_bit_counter <= "0000100000000000";
      when "1100"        => next_bit_counter <= "0001000000000000";
      when "1101"        => next_bit_counter <= "0010000000000000";
      when "1110"        => next_bit_counter <= "0100000000000000";
      when "1111"        => next_bit_counter <= "1000000000000000";
      when others        =>
    end case;
    next_count_input                       <= (others => '0');
  end if;

  --Attendre les sorties de Montgomery
when wait_constants =>

  --Ecrire des donnees dans la fifo
```

```
  if(valid_out_mon_1 = '1') then
    fifo_1_wr        <= '1';
    fifo_in          <= s_out_mon_1 & s_out_mon_2;
    next_count_input <= count_input+1;
    next_state       <= writting_cts_fifo;
  end if;

  --Ecrire les deux constantes initiales dans la fifo
when writting_cts_fifo =>
  fifo_1_wr        <= valid_out_mon_1;
  next_count_input <= count_input+1;
  if(count_input < x"20") then
    fifo_in        <= s_out_mon_1 & s_out_mon_2;
  end if;

  --Commencer la multiplication
  if(valid_out_mon_1 = '0') then
    next_count_input <= (others => '0');
    next_state       <= transition;
  end if;


when transition =>
  next_count_input <= count_input+1;

  if(count_input > 2) then
    next_count_input <= (others => '0');
                                  --fifo_1_rd <= '1';
                                  --next_addr_n <= addr_n+1;
    if((bit_counter and exp_out) = x"0000") then
      next_state     <= processing_data_0;
    else
      next_state     <= processing_data_1;
    end if;

  end if;


 --Exécuter multiplications successives (entrée dans la boucle de
  Montgomery)
when processing_data_1 =>

  if(count_input > x"0000")then
    valid_in_mon_1 <= '1';
    valid_in_mon_2 <= '1';
  end if;


  fifo_1_rd <= '1';

  a_mon_1 <= fifo_out(31 downto 16);
  b_mon_1 <= fifo_out(15 downto 0);
  n_mon_1 <= n_out;
  a_mon_2 <= fifo_out(31 downto 16);
  b_mon_2 <= fifo_out(31 downto 16);
  n_mon_2 <= n_out;
  next_addr_n      <= addr_n+1;
  next_count_input <= count_input +1;

  if(count_input = w_numb) then
    next_state <= wait_results;
```

```
  end if;

when processing_data_0 =>

  if(count_input > x"0000")then
    valid_in_mon_1 <= '1';
    valid_in_mon_2 <= '1';
  end if;

  fifo_1_rd <= '1';

  a_mon_1 <= fifo_out(15 downto 0);
  b_mon_1 <= fifo_out(15 downto 0);
  n_mon_1 <= n_out;

  a_mon_2 <= fifo_out(31 downto 16);
  b_mon_2 <= fifo_out(15 downto 0);
  n_mon_2 <= n_out;

  next_addr_n      <= addr_n+1;
  next_count_input <= count_input +1;

  if(count_input = w_numb) then
    next_state       <= wait_results;
    next_count_input <= (others => '0');
  end if;

when wait_results =>

  --Ecrire les données dans la fifo
  if(valid_out_mon_1 = '1') then
    fifo_1_wr        <= '1';
    fifo_in          <= s_out_mon_2 & s_out_mon_1;
    next_count_input <= x"0001";
    next_state       <= writting_results;
  end if;

when writting_results =>

  next_addr_n      <= (others => '0');
  fifo_1_wr        <= valid_out_mon_1;
  next_count_input <= count_input+1;
  if(count_input < x"20") then
    fifo_in        <= s_out_mon_2 & s_out_mon_1;
  end if;

  --necessaire pour la multiplication
  if(valid_out_mon_1 = '0') then
    next_count_input   <= (others => '0');
    next_state         <= prepare_next;
                          --Calcule du bit de l'exposant
                          --Décalage du masque
    next_bit_counter   <= '0'&bit_counter(15 downto 1);
    if(bit_counter = x"0001")
    then
      next_addr_exp    <= addr_exp -1;
      next_bit_counter <= "1000000000000000";
    end if;
    if((bit_counter = x"0001") and addr_exp = "000000000")
    then
      next_state <= final_mult;
```

```
        next_count_input <= (others => '0');
        next_addr_exp <= (others => '0');
      end if;
    end if;

  when prepare_next =>
    next_state <= transition;
    next_count_input <= (others => '0');
    fifo_1_rd <= '0';

  when final_mult =>
    next_count_input <= count_input+1;

    if(count_input > 2) then
      next_count_input <= (others => '0');
      next_state <= prepare_final;
    end if;

  when prepare_final =>

    if(count_input > x"0000")then
      valid_in_mon_1 <= '1';
    end if;

    fifo_1_rd <= '1'; --Sortie de Montgomery

    a_mon_1 <= fifo_out(15 downto 0);
    if(count_input = x"0001") then
      b_mon_1 <= x"0001";
    end if;
    n_mon_1 <= n_out;

    next_addr_n <= addr_n+1;
    next_count_input <= count_input +1;

    --Afficher les resultatas finaux de Montgomery
    if(count_input = w_numb) then
      next_state <= wait_final;
      next_count_input <= (others =>'0');
    end if;

  when wait_final =>

    if(valid_out_mon_1 = '1') then
      valid_out <= '1';
      s <= s_out_mon_1;
      next_state <= show_final;
      next_count_input <= count_input +1;
    end if;

  when show_final =>
    valid_out <= '1';
    s <= s_out_mon_1;
    next_count_input <= count_input +1;

   if(count_input = x"20") then
      valid_out <= '0';
      next_state <= wait_start;
    end if;

end case;
```

```
  end process;

end Behavioral;
```

Montgomery_mult.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity montgomery_mult is

  port(
    clk       : in  std_logic;
    reset     : in  std_logic;
    valid_in  : in  std_logic;
    a         : in  std_logic_vector(15 downto 0);
    b         : in  std_logic_vector(15 downto 0);
    n         : in  std_logic_vector(15 downto 0);
    s_prev    : in  std_logic_vector(15 downto 0);
    n_c       : in  std_logic_vector(15 downto 0);
    s         : out std_logic_vector(15 downto 0);
    valid_out : out std_logic
    );

end montgomery_mult;

architecture Behavioral of montgomery_mult is

  component montgomery_step is
                        port(
                         clk       : in  std_logic;
                         reset     : in  std_logic;
                         valid_in  : in  std_logic;
                         a         : in  std_logic_vector(15 downto 0);
                         b         : in  std_logic_vector(15 downto 0);
                         n         : in  std_logic_vector(15 downto 0);
                         s_prev    : in  std_logic_vector(15 downto 0);
                         n_c       : in  std_logic_vector(15 downto 0);
                         s         : out std_logic_vector(15 downto 0);
                         valid_out : out std_logic;
                         busy      : out std_logic;
                         b_req     : out std_logic;
                         a_out     : out std_logic_vector(15 downto 0);
                         n_out     : out std_logic_vector(15 downto 0);
                         c_step    : out std_logic;
                         stop      : in  std_logic
                                 );
  end component;

  component fifo_512_bram
    port (
      clk   : in  std_logic;
      rst   : in  std_logic;
      din   : in  std_logic_vector(15 downto 0);
      wr_en : in  std_logic;
      rd_en : in  std_logic;
      dout  : out std_logic_vector(15 downto 0);
```

```
      full  : out std_logic;
      empty : out std_logic);
  end component;

  component fifo_256_feedback
    port (
      clk   : in  std_logic;
      rst   : in  std_logic;
      din   : in  std_logic_vector(48 downto 0);
      wr_en : in  std_logic;
      rd_en : in  std_logic;
      dout  : out std_logic_vector(48 downto 0);
      full  : out std_logic;
      empty : out std_logic);
   end component;

  type arr_dat_out is array(0 to 7) of std_logic_vector(15 downto 0);
  type arr_val is array(0 to 7) of std_logic;
  type arr_b is array(0 to 7) of std_logic_vector(15 downto 0);

  signal b_reg, next_b_reg                                          :
arr_b;
  signal valid_mid, fifo_reqs, fifo_reqs_reg, next_fifo_reqs_reg, stops
:arr_val;
  signal a_out_mid, n_out_mid, s_out_mid                            :
arr_dat_out;

 --Signaux des fifos
  signal wr_en, rd_en, empty : std_logic;
  signal fifo_out            : std_logic_vector(15 downto 0);

  signal fifo_out_feedback, fifo_in_feedback : std_logic_vector(48 downto
0);
  signal read_fifo_feedback, empty_feedback  : std_logic;

  --Les entrees du premier PE
  signal a_in, s_in, n_in : std_logic_vector(15 downto 0);
  signal f_valid, busy_pe : std_logic;

  signal c_step, reg_c_step : std_logic;

  --Les signaux du conteurs
  signal count, next_count  : std_logic_vector(7 downto 0);

  signal wr_fifofeed : std_logic;

  type state_type is (rst_fifos,wait_start, process_data, dump_feed);
  signal state, next_state                    : state_type;
  signal reg_busy                             : std_logic;
  signal reset_fifos                          : std_logic;
  signal count_feedback, next_count_feedback : std_logic_vector(15 downto
0);

begin

--Fifo pour calculer le b
  fifo_b : fifo_512_bram port map (
    clk   => clk,
    rst   => reset_fifos,
    din   => b,
    wr_en => wr_en,
```

```
    rd_en => rd_en,
    dout  => fifo_out,
    empty => empty
    );

--Fifo pour le feedback du premier PE
  fifo_feed : fifo_256_feedback port map (
    clk   => clk,
    rst   => reset_fifos,
    din   => fifo_in_feedback,
    wr_en => wr_fifofeed,
    rd_en => read_fifo_feedback,
    dout  => fifo_out_feedback,
    empty => empty_feedback
    );

--Primer PE
  et_first : montgomery_step port map(
    clk       => clk,
    reset     => reset,
    valid_in  => f_valid,
    a         => a_in,
    b         => b_reg(0),
    n         => n_in,
    s_prev    => s_in,
    n_c       => n_c,
    s         => s_out_mid(0),
    valid_out => valid_mid(0),
    busy      => busy_pe,
    b_req     => fifo_reqs(0),
    a_out     => a_out_mid(0),
    n_out     => n_out_mid(0),
    c_step    => c_step,
    stop      => stops(0)
    );

--Dernier PE
  et_last : montgomery_step port map(
    clk       => clk,
    reset     => reset,
    valid_in  => valid_mid(6),
    a         => a_out_mid(6),
    b         => b_reg(7),
    n         => n_out_mid(6),
    s_prev    => s_out_mid(6),
    n_c       => n_c,
    s         => s_out_mid(7),
    valid_out => valid_mid(7),
    b_req     => fifo_reqs(7),
    a_out     => a_out_mid(7),
    n_out     => n_out_mid(7),
    stop      => stops(7)
    );

  g1     : for i in 1 to 6 generate
    et_i : montgomery_step port map(
      clk       => clk,
      reset     => reset,
      valid_in  => valid_mid(i-1),
      a         => a_out_mid(i-1),
      b         => b_reg(i),
```

```
    n         => n_out_mid(i-1),
    s_prev    => s_out_mid(i-1),
    n_c       => n_c,
    s         => s_out_mid(i),
    valid_out => valid_mid(i),
    b_req     => fifo_reqs(i),
    a_out     => a_out_mid(i),
    n_out     => n_out_mid(i),
    stop      => stops(i)
    );

end generate g1;


process(clk, reset)
begin

  if(clk = '1' and clk'event) then

    if(reset = '1')then
      state               <= wait_start;
      count_feedback      <= (others => '0');
      reg_busy            <= '0';
      for i in 0 to 7 loop
        b_reg(i)          <= (others => '0');
        fifo_reqs_reg (i) <= '0';
        count             <= (others => '0');
        reg_c_step        <= '0';

      end loop;
    else
      state               <= next_state;
      reg_busy            <= busy_pe;
      count_feedback      <= next_count_feedback;
      for i in 0 to 7 loop
        b_reg(i)          <= next_b_reg(i);
        fifo_reqs_reg (i) <= next_fifo_reqs_reg(i);
        count             <= next_count;
        reg_c_step        <= c_step;
      end loop;
    end if;
  end if;

end process;


process(fifo_reqs_reg, fifo_out, b, fifo_reqs, b_reg, state, empty)
begin


  for i in 0 to 7 loop
    next_b_reg(i)         <= b_reg(i);
    next_fifo_reqs_reg(i) <= fifo_reqs(i);
  end loop;

  if(state = wait_start) then
    next_b_reg(0)           <= b;
    next_fifo_reqs_reg(0)   <= '0';
    for i in 1 to 7 loop
      next_b_reg(i)         <= (others => '0');
      next_fifo_reqs_reg(i) <= '0';
```

```
      end loop;
    else
      for i in 0 to 7 loop
        if(fifo_reqs_reg(i) = '1' and empty = '0') then
          next_b_reg(i)       <= fifo_out;
        end if;
      end loop;
    end if;
  end process;


  process( valid_in, b, state, fifo_reqs, a_out_mid, n_out_mid, s_out_mid,
valid_mid, a, s_prev, n, busy_pe, empty_feedback, fifo_out_feedback, count,
reg_c_step, reset, reg_busy, count_feedback )

begin

    rd_en              <= fifo_reqs(0) or fifo_reqs(1) or fifo_reqs(2) or
fifo_reqs(3) or fifo_reqs(4) or fifo_reqs(5) or fifo_reqs(6) or
fifo_reqs(7);
    next_state         <= state;
    wr_en              <= '0';
    fifo_in_feedback   <=
a_out_mid(7)&n_out_mid(7)&s_out_mid(7)&valid_mid(7);
    read_fifo_feedback <= '0';
    wr_fifofeed        <= '0';
    --Le PE primaire
    a_in               <= a;
    s_in               <= s_prev;
    n_in               <= n;
    f_valid            <= valid_in;
    reset_fifos        <= reset;
    --Les sorties
    s                  <= (others => '0');
    valid_out          <= '0';

    --Incrementer le compteur
    next_count          <= count;
    next_count_feedback <= count_feedback;
    --incrementer le compteur dans le pipeleine
    if(reg_c_step = '1') then
      next_count        <= count + 8;
    end if;
        if( count = x"20") then
      s                 <= s_out_mid(0);
      valid_out         <= valid_mid(0);
    end if;

    for i in 0 to 7 loop
      stops(i) <= '0';
    end loop;

    case state is
      when wait_start =>
        --reset_fifos   <= '1';
        if(valid_in = '1') then
          reset_fifos <= '0';
          next_state  <= process_data;
        --wr_en <= '1';
        end if;
```

```
      when process_data =>
        wr_fifofeed <= valid_mid(7);
        if(valid_in = '1') then
          wr_en     <= '1';
        end if;
        if(empty_feedback = '0' and reg_busy = '0') then
          read_fifo_feedback <= '1';
          next_state <= dump_feed;
          next_count_feedback <= x"0000";
        end if;


        if( count >  x"23") then
          next_state <= wait_start;
                              --y
          for i in 0 to 7 loop
            stops(i) <= '1';
          end loop;
          next_count <= (others => '0');
        end if;

        --Vider la fifo de feedback
      when dump_feed =>
        if(empty_feedback='0')then
          next_count_feedback <= count_feedback+1;
        end if;
        wr_fifofeed <= valid_mid(7);
        read_fifo_feedback <= '1';
        a_in <= fifo_out_feedback(48 downto 33);
        n_in <= fifo_out_feedback(32 downto 17);
        s_in <= fifo_out_feedback(16 downto 1);
        f_valid <= fifo_out_feedback(0);
        if(empty_feedback='1') then
          next_state <= process_data;

        end if;
        if(count_feedback = x"22") then
          read_fifo_feedback <= '0';
          next_state <= process_data;
        end if;
       when rst_fifos =>
           next_state <= wait_start;
              reset_fifos   <= '1';
    end case;

  end process;
end Behavioral;
```

Montgomery_step.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;


entity montgomery_step is
```

```
  port(
    clk       : in  std_logic;
    reset     : in  std_logic;
    valid_in  : in  std_logic;
    a         : in  std_logic_vector(15 downto 0);
    b         : in  std_logic_vector(15 downto 0);
    n         : in  std_logic_vector(15 downto 0);
    s_prev    : in  std_logic_vector(15 downto 0);
    n_c       : in  std_logic_vector(15 downto 0);
    s         : out std_logic_vector(15 downto 0);
    valid_out : out std_logic;
    busy      : out std_logic;
    b_req     : out std_logic;
    a_out     : out std_logic_vector(15 downto 0);
    n_out     : out std_logic_vector(15 downto 0);
    c_step    : out std_logic;
    stop      : in  std_logic
    );
end montgomery_step;

architecture Behavioral of montgomery_step is

  component pe_wrapper
    port(
      clk          : in  std_logic;
      reset        : in  std_logic;
      ab_valid     : in  std_logic;
      valid_in     : in  std_logic;
      a            : in  std_logic_vector(15 downto 0);
      b            : in  std_logic_vector(15 downto 0);
      n            : in  std_logic_vector(15 downto 0);
      s_prev       : in  std_logic_vector(15 downto 0);
      n_c          : in  std_logic_vector(15 downto 0);
      s            : out std_logic_vector(15 downto 0);
      data_ready   : out std_logic;
      fifo_req     : out std_logic;
      m_val        : out std_logic;
      reset_the_PE : in  std_logic);
      end component;

  --Entrees

  signal ab_valid              : std_logic;
  signal valid_mont, fifo_read, m_val, valid_mont_out, reset_pe :
std_logic;


 --Sorties

  type state_type is (wait_valid, wait_m, mont_proc, getting_results,
prep_m,
  b_stable);
  signal state, next_state     : state_type;
  signal counter, next_counter : std_logic_vector(7 downto 0);
    --il compte les mots qui sont sortis pour aller coupure

  signal mont_input_a, mont_input_n, mont_input_s: std_logic_vector(15
downto 0);
  signal reg_constant, next_reg_constant, next_reg_input, reg_input :
std_logic_vector(47 downto 0);
```

```
  signal reg_out, reg_out_1, reg_out_2, reg_out_3, reg_out_4        :
std_logic_vector(31 downto 0);
  signal next_reg_out                   : std_logic_vector(31 downto 0);

  signal reg_input_1, reg_input_2, reg_input_3, reg_input_4, reg_input_5 :
std_logic_vector(47 downto 0);


begin

  mont : pe_wrapper port map (
    clk          => clk,
    reset        => reset,
    ab_valid     => ab_valid,
    a            => mont_input_a,
    b            => b,
    n            => mont_input_n,
    s_prev       => mont_input_s,
    n_c          => n_c,
    s            => s,
    valid_in     => valid_mont,
    data_ready   => valid_mont_out,
    m_val        => m_val,
    reset_the_PE => reset_pe
    );

  process(clk, reset)
  begin
    if(clk = '1' and clk'event) then

      if(reset = '1')then
        state        <= wait_valid;
        counter      <= (others => '0');
        reg_constant <= (others => '0');
        reg_input    <= (others => '0');
        reg_input_1  <= (others => '0');
        reg_input_2  <= (others => '0');
        reg_input_3  <= (others => '0');
        reg_input_4  <= (others => '0');

        reg_out   <= (others => '0');
        reg_out_1 <= (others => '0');
        reg_out_2 <= (others => '0');
        reg_out_3 <= (others => '0');
        reg_out_4 <= (others => '0');

      else
        reg_input   <= next_reg_input;
        reg_input_1 <= reg_input;
        reg_input_2 <= reg_input_1;
        reg_input_3 <= reg_input_2;
        reg_input_4 <= reg_input_3;
        reg_input_5 <= reg_input_4;

        reg_out   <= reg_input_4(47 downto 32) & reg_input_4(31 downto 16);
        reg_out_1 <= reg_out;
        reg_out_2 <= reg_out_1;
        reg_out_3 <= reg_out_2;
        reg_out_4 <= reg_out_3;

        state        <= next_state;
```

```
        counter      <= next_counter;
        reg_constant <= next_reg_constant;
      end if;
    end if;
  end process;

  process(state, valid_in, m_val, a, n, s_prev, counter, valid_mont_out,
stop, reg_constant,
  reg_input_5, reg_out_4)
  begin
    --reset_fifo <= '0';
    next_reg_input <= a&n&s_prev;
    --next_reg_out <= a&n;

    a_out <= reg_out_4(31 downto 16);
    n_out <= reg_out_4(15 downto 0);

    next_state   <= state;
    next_counter <= counter;
    --write_fifos <= valid_in;
    ab_valid     <= '0';
    valid_mont   <= '0';
    valid_out    <= '0';
    reset_pe     <= '0';
    busy         <= '1';
    b_req        <= '0';
    c_step       <= '0';

    mont_input_a      <= (others => '0');
    mont_input_n      <= (others => '0');
    mont_input_s      <= (others => '0');
    next_reg_constant <= reg_constant;

    case state is
      when wait_valid =>
        busy     <= '0';
        reset_pe <= '1';
        if(valid_in = '1') then
          b_req             <= '1';
          next_state        <= b_stable;
          next_reg_constant <= a&n&s_prev;
        end if;


        when b_stable =>
        next_state          <= prep_m;
      when prep_m   =>
        mont_input_a        <= reg_constant(47 downto 32);
               --Ajouter au sensitivity
        mont_input_n        <= reg_constant(31 downto 16);
        mont_input_s        <= reg_constant(15 downto 0);
        ab_valid            <= '1';
        next_state          <= wait_m;



            when wait_m   =>
        --Nous maintenons les entrées pour calculer le m
        mont_input_a <= reg_constant(47 downto 32);
        --Ajouter au sensitivity
        mont_input_n <= reg_constant(31 downto 16);
```

```
        mont_input_s <= reg_constant(15 downto 0);


        if (m_val = '1')    -- m (qi) est pret
              valid_mont   <= '1';
          next_state   <= mont_proc;
          mont_input_a <= reg_input_5(47 downto 32);
          mont_input_n <= reg_input_5(31 downto 16);
          mont_input_s <= reg_input_5(15 downto 0);

        end if;

      when mont_proc =>

        valid_mont   <= '1';
        mont_input_a <= reg_input_5(47 downto 32);
        mont_input_n <= reg_input_5(31 downto 16);
        mont_input_s <= reg_input_5(15 downto 0);

        if(valid_mont_out = '1') then

          next_counter <= x"00";
          next_state   <= getting_results;
        end if;

      when getting_results =>

        valid_out    <= '1';
        next_counter <= counter+1;
        valid_mont   <= '1';

        mont_input_a <= reg_input_5(47 downto 32);
        mont_input_n <= reg_input_5(31 downto 16);
        mont_input_s <= reg_input_5(15 downto 0);

        if(counter = (x"22")) then -- 34 mot de 16 bits
          next_state <= wait_valid;
          c_step <= '1';
          reset_pe <= '1';
        end if;

    end case;

    if(stop='1') then
      next_state <= wait_valid;
      --reset_fifo <= '1';
      reset_pe <= '1';
    end if;

  end process;

end Behavioral;
```

pe_wrapper.vhd

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity pe_wrapper is

  port(
    clk          : in  std_logic;
    reset        : in  std_logic;
    ab_valid     : in  std_logic;
    valid_in     : in  std_logic;
    a            : in  std_logic_vector(15 downto 0);
    b            : in  std_logic_vector(15 downto 0);
    n            : in  std_logic_vector(15 downto 0);
    s_prev       : in  std_logic_vector(15 downto 0);
    n_c          : in  std_logic_vector(15 downto 0);
    s            : out std_logic_vector(15 downto 0);
    data_ready   : out std_logic;
    fifo_req     : out std_logic;
    m_val        : out std_logic;
    reset_the_PE : in  std_logic);


end pe_wrapper;

architecture Behavioral of pe_wrapper is

  component pe is
                port ( clk          : in  std_logic;
                       reset        : in  std_logic;
                       a_j          : in  std_logic_vector(15 downto 0);
                       b_i          : in  std_logic_vector(15 downto 0);
                       s_prev       : in  std_logic_vector(15 downto 0);
                            --la valeur precedente de la somme S
                       m            : in  std_logic_vector(15 downto 0);
                       n_j          : in  std_logic_vector(15 downto 0);
                       s_next       : out std_logic_vector(15 downto 0);
                       aj_bi        : out std_logic_vector(15 downto 0);
                       -- aj_bi=a*b
                      ab_valid_in   : in  std_logic;
                      valid_in      : in  std_logic;
                            ab_valid_out : out std_logic;
                       valid_out    : out std_logic;
                       fifo_req     : out std_logic);
  end component;

  component m_calc is
                    port(
                      clk        : in  std_logic;
                      reset      : in  std_logic;
                      ab         : in  std_logic_vector (15 downto 0);
                      t          : in  std_logic_vector (15 downto 0);
                      n_cons     : in  std_logic_vector (15 downto 0);
                      m          : out std_logic_vector (15 downto 0);
                      mult_valid : in  std_logic;
                      m_valid    : out std_logic);
  end component;

  signal aj_bi, m, next_m, m_out          : std_logic_vector(15 downto 0);
  signal mult_valid, valid_m, valid_m_reg : std_logic;
```

```
begin

  pe_0 : pe port map(
    clk           => clk,
    reset         => reset,
    a_j           => a,
    b_i           => b,
    s_prev        => s_prev,
    m             => m,
    n_j           => n,
    s_next        => s,
    aj_bi         => aj_bi,
    ab_valid_in   => ab_valid,
    valid_in      => valid_in,
    ab_valid_out => mult_valid,
    valid_out     => data_ready,
    fifo_req      => fifo_req);

  mcons_0 : m_calc port map(
    clk         => clk,
    reset       => reset,
    ab          => aj_bi,
    t           => s_prev,
    n_cons      => n_c,
    m           => m_out,
    mult_valid => mult_valid,
    m_valid     => valid_m);

  process(clk, reset)
  begin
    if(clk='1' and clk'event) then

      if(reset='1')then
        m <= (others=> '0' );
        valid_m_reg <= '0';
      else
        m <= next_m;
        valid_m_reg <= valid_m;
      end if;
    end if;
  end process;

  process(m_out,valid_m, valid_m_reg, m)
  begin
    m_val <= valid_m_reg;
    if(valid_m = '1' and valid_m_reg ='0') then
      next_m <= m_out;
    else
      next_m <= m;
    end if;
  end process;

end Behavioral;
```

pe.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;


entity pe is
  port ( clk          : in  std_logic;
         reset        : in  std_logic;
         a_j          : in  std_logic_vector(15 downto 0);
         b_i          : in  std_logic_vector(15 downto 0);
         s_prev       : in  std_logic_vector(15 downto 0);
         m            : in  std_logic_vector(15 downto 0);
         n_j          : in  std_logic_vector(15 downto 0);
         s_next       : out std_logic_vector(15 downto 0);
         aj_bi        : out std_logic_vector(15 downto 0);
         ab_valid_in  : in  std_logic;
         valid_in     : in  std_logic;
         ab_valid_out : out std_logic;
         valid_out    : out std_logic;
         fifo_req     : out std_logic);
end pe;

architecture Behavioral of pe is

  signal prod_aj_bi, next_prod_aj_bi, mult_aj_bi                    :
std_logic_vector(31 downto 0);
  signal prod_nj_m, next_prod_nj_m, mult_nj_m, mult_nj_m_reg        :
std_logic_vector(31 downto 0);
  signal sum_1, next_sum_1                                          :
std_logic_vector(31 downto 0);
  signal sum_2, next_sum_2                                          :
std_logic_vector(31 downto 0);
  signal ab_valid_reg, valid_out_reg, valid_out_reg2, valid_out_reg3 :
std_logic;
  signal n_reg, next_n_reg, s_prev_reg, next_s_prev_reg, ab_out_reg  :
std_logic_vector(15 downto 0);
  --signal prod_aj_bi_out, next_prod_aj_bi_out : std_logic_vector(15 downto
0);

begin


  mult_aj_bi <= a_j * b_i;
  mult_nj_m  <= n_reg *m;


  process(clk, reset)
  begin

    if(clk = '1' and clk'event)
    then
      if(reset = '1') then
        prod_aj_bi     <= (others => '0');
        prod_nj_m      <= (others => '0');
        sum_1          <= (others => '0');
        sum_2          <= (others => '0');
        ab_valid_reg   <= '0';
        n_reg          <= (others => '0');
```

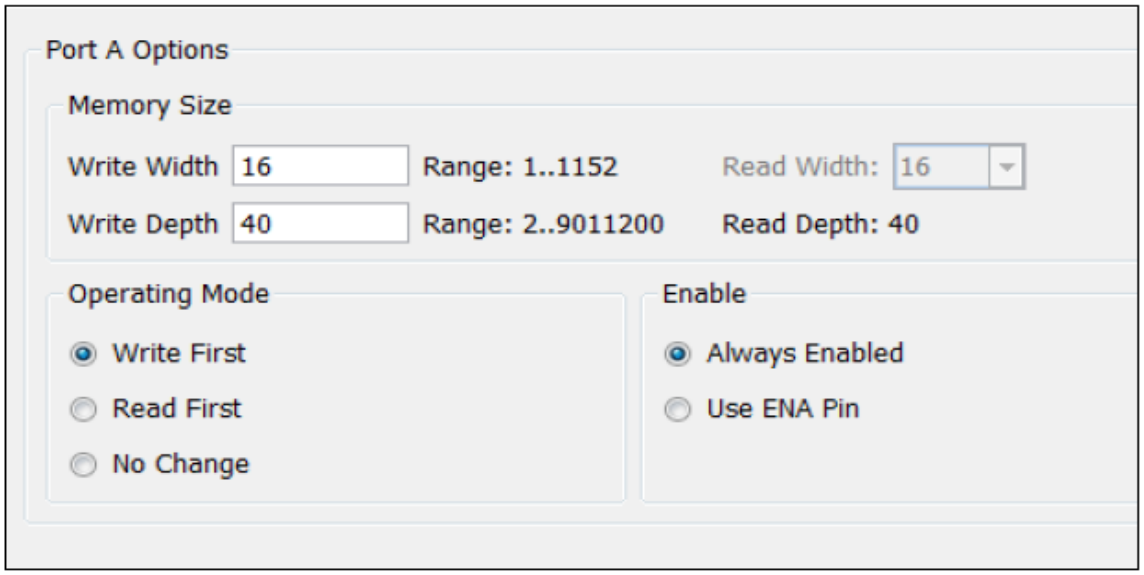
```
        valid_out_reg  <= '0';
        valid_out_reg2 <= '0';
        valid_out_reg3 <= '0';
        s_prev_reg     <= (others => '0');
      else
        --prod_aj_bi_out <= next_prod_aj_bi_out;
        prod_aj_bi     <= next_prod_aj_bi;
        prod_nj_m      <= next_prod_nj_m;
        sum_1          <= next_sum_1;
        sum_2          <= next_sum_2;
        ab_valid_reg   <= ab_valid_in;
        ab_out_reg     <= mult_aj_bi(15 downto 0);
        n_reg          <= next_n_reg;
        valid_out_reg  <= valid_in;
        valid_out_reg2 <= valid_out_reg;
        valid_out_reg3 <= valid_out_reg2;
        s_prev_reg     <= next_s_prev_reg;
        --mult_nj_m_reg <= mult_nj_m;
      end if;
    end if;

  end process;

  process(s_prev, prod_aj_bi, prod_nj_m, sum_1, sum_2, mult_aj_bi,
mult_nj_m, valid_in, ab_valid_reg, n_j, n_reg, valid_out_reg3, s_prev_reg,
ab_out_reg)
  begin
    ab_valid_out      <= ab_valid_reg;
    aj_bi             <= ab_out_reg(15 downto 0);
    s_next            <= sum_2     (15 downto 0);
    fifo_req          <= valid_in;
    valid_out         <= valid_out_reg3;
    next_sum_1        <= sum_1;
    next_sum_2        <= sum_2;
    next_prod_nj_m    <= prod_nj_m;
    next_prod_aj_bi   <= prod_aj_bi;
    next_n_reg        <= n_reg;
    next_s_prev_reg   <= s_prev_reg;
    if(valid_in = '1') then
      next_s_prev_reg <= s_prev;
      next_n_reg      <= n_j;
      next_prod_aj_bi <= mult_aj_bi;
      next_prod_nj_m  <= mult_nj_m;
      next_sum_1      <= prod_aj_bi+sum_1(31 downto 16)+s_prev_reg;
      next_sum_2      <= prod_nj_m+sum_2 (31 downto 16) + sum_1(15 downto
0);
    else
      next_s_prev_reg <= (others => '0');
      next_n_reg      <= (others => '0');
      next_prod_aj_bi <= (others => '0');
      next_prod_nj_m  <= (others => '0');
      next_sum_1      <= (others => '0');
      next_sum_2      <= (others => '0');
    end if;

  end process;

end Behavioral;
```

m_calc.vhd

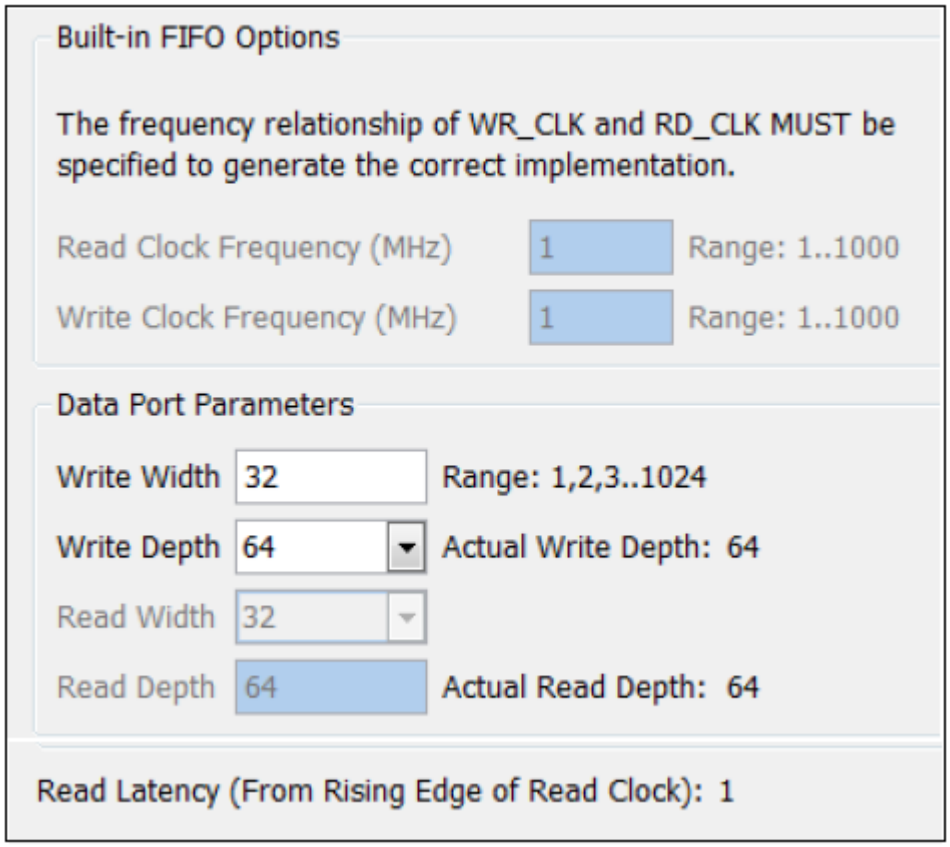
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity m_calc is

  port(
    clk        : in  std_logic;
    reset      : in  std_logic;
    ab         : in  std_logic_vector (15 downto 0);
    t          : in  std_logic_vector (15 downto 0);
    n_cons     : in  std_logic_vector (15 downto 0);
    m          : out std_logic_vector (15 downto 0);
    mult_valid : in  std_logic;
    m_valid    : out std_logic);
end m_calc;

architecture Behavioral of m_calc is

  signal sum_res, next_sum_res      : std_logic_vector(15 downto 0);
  signal mult_valid_1, mult_valid_2 : std_logic;
  signal mult                       : std_logic_vector(31 downto 0);
begin

  mult <= sum_res * n_cons;

  process(clk, reset)
  begin

    if(clk = '1' and clk'event) then

      if(reset = '1') then
        sum_res      <= (others => '0');
        mult_valid_1 <= '0';
        mult_valid_2 <= '0';
      else
        sum_res      <= next_sum_res;
        mult_valid_1 <= mult_valid;
        mult_valid_2 <= mult_valid_1;

      end if;

    end if;

  end process;

  process(ab, t, mult_valid_2)
  begin
    m            <= mult(15 downto 0);
    next_sum_res <= ab+t;
    m_valid      <= mult_valid_2;
  end process;

end Behavioral;
```

Coregenerateurs de rsa_top_512

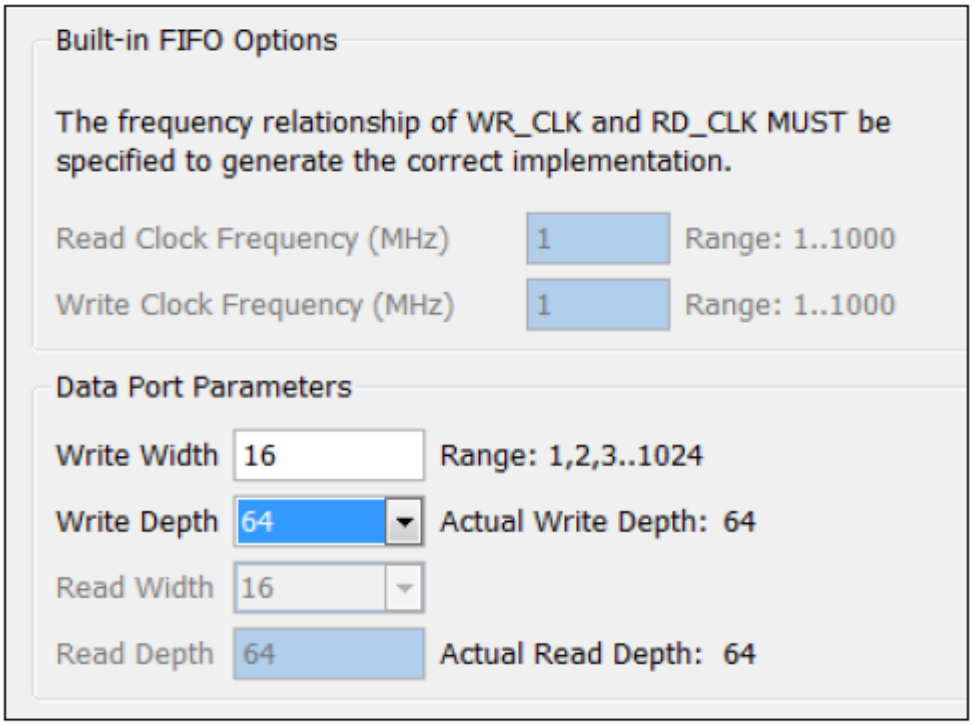
Mem_b



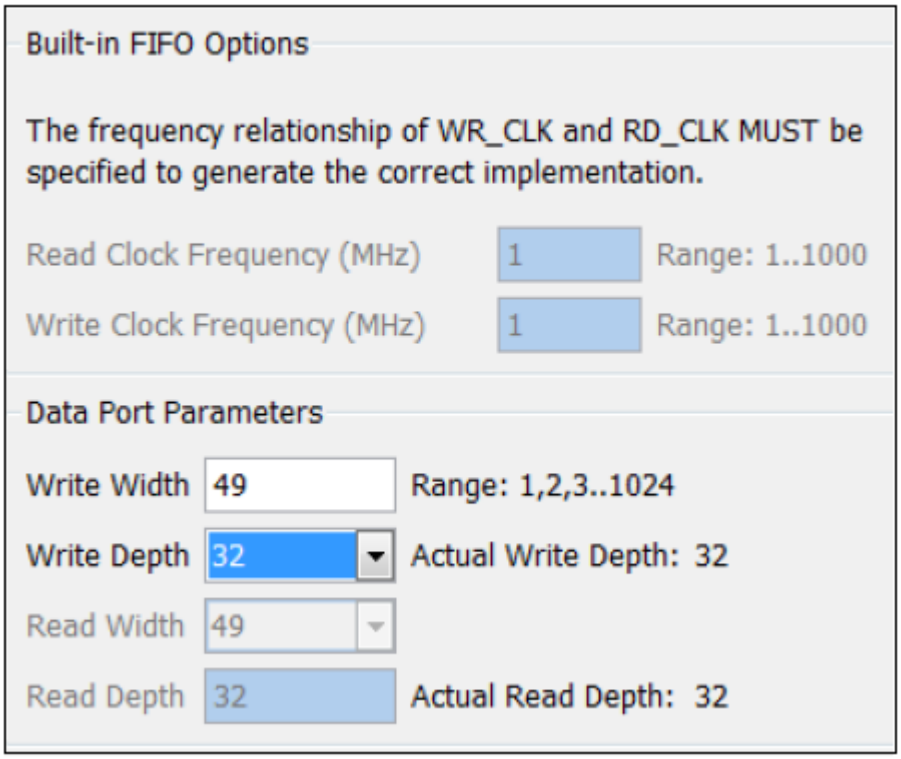
res_out_fifo



Fifo_512_bram



Fifo_256_feedback



testbench : test_rsa_512 (32 mots de 16 bits)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity test_rsa_512 is
end test_rsa_512;

architecture behavior of test_rsa_512 is

  component rsa_top
    port(
      clk       : in  std_logic;
      reset     : in  std_logic;
      valid_in  : in  std_logic;
      x         : in  std_logic_vector(15 downto 0);
      y         : in  std_logic_vector(15 downto 0);
      m         : in  std_logic_vector(15 downto 0);
      r_c       : in  std_logic_vector(15 downto 0);
      n_c       : in  std_logic_vector(15 downto 0);
      s         : out std_logic_vector(15 downto 0);
      valid_out : out std_logic;
      bit_size  : in  std_logic_vector(15 downto 0)
      );
  end component;


  --Entrees
  signal clk       : std_logic                      := '0';
  signal reset     : std_logic                      := '0';
  signal valid_in  : std_logic                      := '0';
  signal x         : std_logic_vector(15 downto 0) := (others => '0');
  signal y         : std_logic_vector(15 downto 0) := (others => '0');
  signal m         : std_logic_vector(15 downto 0) := (others => '0');
  signal r_c       : std_logic_vector(15 downto 0) := (others => '0');
  signal n_c       : std_logic_vector(15 downto 0) := (others => '0');
  signal bit_size  : std_logic_vector(15 downto 0) := x"0200";
  --Sorties
  signal s         : std_logic_vector(15 downto 0);
  signal valid_out : std_logic;

  -- Clock period definitions
  constant clk_period : time := 1ns;

  begin

  -- Instantiate the Unit Under Test (UUT)
  uut : rsa_top port map (
    clk       => clk,
    reset     => reset,
    valid_in  => valid_in,
    x         => x,
    y         => y,
    m         => m,
    r_c       => r_c,
    n_c       => n_c,
    s         => s,
    valid_out => valid_out,
```

```
  bit_size  => bit_size
  );

-- Clock process definitions
clk_process : process
begin
  clk <= '0';
  wait for clk_period/2;
  clk <= '1';
  wait for clk_period/2;
end process;


  -- Stimulus process
  stim_proc : process
  begin
  valid_in <= '0';
  -- hold reset state for 100ms.
  reset    <= '1';
  wait for 10ns;
  reset    <= '0';
  wait for clk_period*10;

  -- insert stimulus here

  --n_c and valid signal and the r_c constant are also required
  n_c      <= x"738f";
  valid_in <= '1';
  x        <= x"f3ad";
  y        <= x"42b1";
  m        <= x"b491";
  r_c      <= x"f579";
  wait for clk_period;
  x        <= x"8e40";
  y        <= x"1ad3";
  m        <= x"1417";
  r_c      <= x"6ee9";
  wait for clk_period;
  x        <= x"6af9";
  y        <= x"a827";
  m        <= x"b498";
  r_c      <= x"972d";
  wait for clk_period;
  x        <= x"4e63";
  y        <= x"0d64";
  m        <= x"e1b7";
  r_c      <= x"5052";
  wait for clk_period;
  x        <= x"9600";
  y        <= x"3f76";
  m        <= x"e47c";
  r_c      <= x"1dca";
  wait for clk_period;
  x        <= x"68f4";
  y        <= x"6670";
  m        <= x"b186";
  r_c      <= x"bc81";
  wait for clk_period;
  x        <= x"5a12";
  y        <= x"5a1c";
  m        <= x"93f0";
```

```
r_c      <= x"377e";
wait for clk_period;
x        <= x"d62e";
y        <= x"4844";
m        <= x"b183";
r_c      <= x"04ef";
wait for clk_period;
x        <= x"8fc1";
y        <= x"d5f2";
m        <= x"f8f1";
r_c      <= x"3a2a";
wait for clk_period;
x        <= x"031d";
y        <= x"b65a";
m        <= x"eed1";
r_c      <= x"291b";
wait for clk_period;
x        <= x"f496";
y        <= x"034f";
m        <= x"0083";
r_c      <= x"c159";
wait for clk_period;
x        <= x"1268";
y        <= x"9635";
m        <= x"981c";
r_c      <= x"9336";
wait for clk_period;
x        <= x"2e5a";
y        <= x"386e";
m        <= x"6441";
r_c      <= x"1bd0";
wait for clk_period;
x        <= x"c1d6";
y        <= x"fb73";
m        <= x"fcd8";
r_c      <= x"317d";
wait for clk_period;
x        <= x"cd8f";
y        <= x"5623";
m        <= x"cbf0";
r_c      <= x"64b4";
wait for clk_period;
x        <= x"e4d2";
y        <= x"9041";
m        <= x"e3ca";
r_c      <= x"8793";
wait for clk_period;
x        <= x"36c6";
y        <= x"99da";
m        <= x"41d9";
r_c      <= x"85f5";
wait for clk_period;
x        <= x"df4a";
y        <= x"cd68";
m        <= x"b7a0";
r_c      <= x"7c8d";
wait for clk_period;
x        <= x"8e40";
y        <= x"9a94";
m        <= x"146e";
r_c      <= x"64d9";
```

```
wait for clk_period;
x        <= x"6af9";
y        <= x"ccc8";
m        <= x"4776";
r_c      <= x"c7f6";
wait for clk_period;
x        <= x"4e63";
y        <= x"ed49";
m        <= x"ec50";
r_c      <= x"fba0";
wait for clk_period;
x        <= x"9600";
y        <= x"4d25";
m        <= x"c07c";
r_c      <= x"e3e0";
wait for clk_period;
x        <= x"68f4";
y        <= x"3b8e";
m        <= x"e698";
r_c      <= x"b567";
wait for clk_period;
x        <= x"5a12";
y        <= x"36d5";
m        <= x"d85f";
r_c      <= x"3172";
wait for clk_period;
x        <= x"d62e";
y        <= x"3a75";
m        <= x"729c";
r_c      <= x"111a";
wait for clk_period;
x        <= x"8fc1";
y        <= x"77a3";
m        <= x"19b6";
r_c      <= x"1971";
wait for clk_period;
x        <= x"d2cd";
y        <= x"367f";
m        <= x"05d3";
r_c      <= x"9f9b";
wait for clk_period;
x        <= x"c6e4";
y        <= x"68de";
m        <= x"cacd";
r_c      <= x"b574";
wait for clk_period;
x        <= x"4a36";
y        <= x"59a4";
m        <= x"e16f";
r_c      <= x"4a50";
wait for clk_period;
x        <= x"f6df";
y        <= x"9f89";
m        <= x"f67b";
r_c      <= x"6d56";
wait for clk_period;
x        <= x"061c";
y        <= x"ed71";
m        <= x"7066";
r_c      <= x"bdc6";
wait for clk_period;
```

```
    x         <= x"06c8";
    y         <= x"059f";
    m         <= x"08de";
    r_c       <= x"0400";
    wait for clk_period;
    valid_in <= '0';

--valid_in <='0';
    wait for clk_period*200000;

    wait;
  end process;
  END;
```

Constant_gen.c (calcul de r_c et n_c a partir du modulo ‘’m’’)

```
#include <stdio.h>
#include <gmp.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#define uint16 unsigned short int
#define uint32 unsigned int
#define uint64 unsigned long long int
#define WORD_SIZE 32


uint32 getMpzSize(mpz_t n)
{

   return (mpz_size(n)*2);

}

uint16 getlimb(mpz_t n, int i)
{
   uint32 aux=mpz_getlimbn(n,i/2);
   if(i%2 == 0)
   return (uint16) (aux&0xffff);
   return (uint16) (aux>>16);

}


int main()
{
    int i;

    mpz_t m,x,y,r,r_aux, n_cons, zero, recons;

    char *template;

      mpz_init_set_str(m,"8de7066f67be16fcacd05d319b6729cd85fe698c07cec5047
      76146eb7a041d9e3cacbf0fcd86441981c0083eed1f8f1b18393f0b186e47ce1b7b49
      81417b491",16); //mpz_init_set_str(m,"c3217fff",16);

    mpz_init(r);
    mpz_init(r_aux);
```

```
	mpz_init(n_cons);
	mpz_init_set_ui(zero,0);

	//Calculo de la constante r_c et n_c de montgomery
	mpz_ui_pow_ui(r,2,16*(getMpzSize(m)+1));
	mpz_mul(r_aux,r,r);
	mpz_mod(r_aux,r_aux,m);

	mpz_sub(n_cons,zero,m);
	mpz_invert(n_cons,n_cons,r);

	printf    ("n_c <= %x;\n\n", (uint32)mpz_getlimbn(n_cons,0)&0xffff);
	gmp_printf("r_c <= %Zx\n\n", r_aux);

	return 0;
}
```

Compteur_ 512 bits (introduit des vecteurs de test jusqu'a 2^{512} combinaisons)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity COUNT512 is

  port(

        DIN          : in          std_logic_vector(511 downto 0);
        CLK          : in          std_logic;
        LOAD         : in          std_logic;
        RESET        : in          std_logic;
        DOUT         : out         std_logic_vector(511 downto 0)

                );

end COUNT512;

architecture behavior of COUNT512 is

begin

        clk_proc:process(CLK, RESET)
        variable RET:unsigned(511 downto 0):=
"000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000";
        Variable COUNT:unsigned(511 downto 0):=
```

```
"00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000";


            begin

            if (RESET ='1' or RET = '1' ) then
            COUNT                                                                   :=
"00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000";
            elsif (CLK ='1' and CLK 'event) then

                        if LOAD = '1' then
                        COUNT := DIN;
                        else COUNT := COUNT + 1;

            end if;
            if                              (COUNT                                  =
"11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111") then
            RET <= '1';
            else
            RET <= '0';
            end if;
            end if;
            DOUT <= COUNT;
            end process clk_proc;

end behavior;
```

Annexe C

IP-Core RSA _1024 (64 mots de 16 bits)

rsa_top.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
library UNISIM;
use UNISIM.VCOMPONENTS.all;


entity rsa_top is
 port(
  clk      : in  std_logic;
  reset    : in  std_logic;
  valid_in : in  std_logic;
  x        : in  std_logic_vector(15 downto 0);
  y        : in  std_logic_vector(15 downto 0);
  m        : in  std_logic_vector(15 downto 0);
  r_c      : in  std_logic_vector(15 downto 0);-- constant de Montgomery:
r^2modm
  s        : out std_logic_vector( 15 downto 0);-- s=x^ymodm
  valid_out: out std_logic;
  bit_size : in  std_logic_vector(15 downto 0); -- (log2(y))
    );
 end rsa_top;

architecture Behavioral of rsa_top is

-- Composant n_c
component n_c_core

    port (clk    : in  std_logic;
          m_lsw  : in  std_logic_vector(15 downto 0);
          ce     : in  std_logic;
          n_c    : out std_logic_vector(15 downto 0);
          done   : out std_logic
          );
  end component;

-- Deux Block de Multiplicateur de Montgomery qui fonctionne en même temps
  component montgomery_mult
                   port(
                   clk       : in  std_logic;
                   reset     : in  std_logic;
                   valid_in  : in  std_logic;
                   a         : in  std_logic_vector(15 downto 0);
                   b         : in  std_logic_vector(15 downto 0);
                   n         : in  std_logic_vector(15 downto 0);
                   s_prev    : in  std_logic_vector(15 downto 0);
                   n_c       : in  std_logic_vector(15 downto 0);
                   s         : out std_logic_vector(15 downto 0);
                   valid_out : out std_logic
);
  end component;
```

```
-- Memoire pour stocker l'exposant y et le modulo m
  component Mem_b
    port (
      clka  : in  std_logic;
      wea   : in  std_logic_vector( 0 downto 0);
      addra : in  std_logic_vector( 8 downto 0);
      dina  : in  std_logic_vector(15 downto 0);
      douta : out std_logic_vector(15 downto 0));
  end component;


-- fifos pour stocker les multiplications intermédiaires/partielles
  component res_out_fifo
    port (
      clk   : in  std_logic;
      rst   : in  std_logic;
      din   : in  std_logic_vector(31 downto 0);
      wr_en : in  std_logic;
      rd_en : in  std_logic;
      dout  : out std_logic_vector(31 downto 0);
      full  : out std_logic;
      empty : out std_logic);
  end component;


  signal valid_in_mon_1, valid_in_mon_2, valid_out_mon_1, valid_out_mon_2,
  fifo_1_rd, fifo_1_wr : std_logic;

  signal a_mon_1, b_mon_1, n_mon_1, s_p_mon_1, s_out_mon_1, a_mon_2,
b_mon_2,
  n_mon_2, s_p_mon_2, s_out_mon_2, fifo_1_in, fifo_2_in, fifo_1_out,
exp_out,
  n_out : std_logic_vector(15 downto 0);

  signal fifo_out, fifo_in : std_logic_vector(31 downto 0);

  signal addr_exp, addr_n, next_addr_exp, next_addr_n : std_logic_vector
  (8 downto 0);

  type state_type is (wait_start, prepare_data,
wait_constants,writting_cts_fifo,
  processing_data_0, processing_data_1, wait_results, transition,
prepare_next,
  writting_results, final_mult, show_final, prepare_final, wait_final);
  signal state, next_state                      : state_type;
  signal w_numb, next_w_numb                    : std_logic_vector(15
downto 0);

--Les signaux de registres
  signal n_c_reg, next_n_c_reg                  : std_logic_vector(15
downto 0);

--Comptage des données qui passent par les multiplicateurs
  signal count_input, next_count_input, bit_counter, next_bit_counter :
  std_logic_vector(15 downto 0);

  signal bsize_reg, next_bsize_reg       : std_logic_vector(15 downto 0);
  signal write_b_n                       : std_logic_vector(0  downto 0);
```

```
  signal n_c_o    : std_logic_vector(15 downto 0);
  signal n_c      : std_logic_vector(15 downto 0);
  signal n_c_load : std_logic;


begin


n_c1 : n_c_core  port map (
      clk   => clk,
      m_lsw => m,
      ce    => start_in,
      n_c   => n_c_o,
      done  => n_c_load
      );


  mon_1 : montgomery_mult port map(
    clk       => clk,
    reset     => reset,
    valid_in  => valid_in_mon_1,
    a         => a_mon_1,
    b         => b_mon_1,
    n         => n_mon_1,
    s_prev    => s_p_mon_1,
    n_c       => n_c_reg,
    s         => s_out_mon_1,
    valid_out => valid_out_mon_1
    );

  mon_2 : montgomery_mult port map(
    clk       => clk,
    reset     => reset,
    valid_in  => valid_in_mon_2,
    a         => a_mon_2,
    b         => b_mon_2,
    n         => n_mon_2,
    s_prev    => s_p_mon_2,
    n_c       => n_c_reg,
    s         => s_out_mon_2,
    valid_out => valid_out_mon_2
    );


  fifo_mon_out : res_out_fifo port map (
      clk   => clk,
      rst   => reset,
      din   => fifo_in,
      wr_en => fifo_1_wr,
      rd_en => fifo_1_rd,
      dout  => fifo_out
      );

  exp : Mem_b port map (
    clka  => clk,
    wea   => write_b_n,
    addra => addr_exp,
    dina  => y,
    douta => exp_out);

  n_mod : Mem_b port map (
```

```
  clka  => clk,
  wea   => write_b_n,
  addra => addr_n,
  dina  => m,
  douta => n_out);



process(clk, reset)
begin

  if(clk = '1' and clk'event) then

    if(reset = '1')then
      state       <= wait_start;
      n_c_reg     <= (others => '0');
      w_numb      <= (others => '0');
      count_input <= (others => '0');
      addr_exp    <= (others => '0');
      addr_n      <= (others => '0');
      bit_counter <= (others => '0');
      bsize_reg   <= (others => '0');
    else
      if(n_c_load = '1') then
        n_c       <= n_c_o;
      end if;
      state       <= next_state;
      n_c_reg     <= next_n_c_reg;
      w_numb      <= next_w_numb;
      count_input <= next_count_input;
      addr_exp    <= next_addr_exp;
      addr_n      <= next_addr_n;
      bit_counter <= next_bit_counter;
      bsize_reg   <= next_bsize_reg;
    end if;
  end if;
end process;

process(state, bsize_reg, n_c_reg, valid_in, x, n_c, r_c, m, y, w_numb,
count_input, addr_exp, addr_n, s_out_mon_1, s_out_mon_2, bit_size,
valid_out_mon_1, bit_counter, exp_out, fifo_out, n_out)

variable mask : std_logic_vector(3 downto 0);

begin

  --Mettre ajour les registres
  next_state       <= state;
  next_n_c_reg     <= n_c_reg;
  next_w_numb      <= w_numb;
  next_count_input <= count_input;
  next_bsize_reg   <= bsize_reg;

  --Les entrées de Montgomerys.
  valid_in_mon_1   <= '0';
  valid_in_mon_2   <= '0';
  a_mon_1          <= (others => '0');
  b_mon_1          <= (others => '0');
  n_mon_1          <= (others => '0');
  a_mon_2          <= (others => '0');
  b_mon_2          <= (others => '0');
```

```
n_mon_2           <= (others => '0');
s_p_mon_1         <= (others => '0');
s_p_mon_2         <= (others => '0');

--Control des fifos
fifo_1_rd         <= '0';
fifo_in           <= (others => '0');
fifo_1_wr         <= '0';

--Control de mémoires (exposant y et modulo m)
write_b_n         <= b"0";
next_addr_exp     <= addr_exp;
next_addr_n       <= addr_n;
next_bit_counter <= bit_counter;

--Les sorties du rsa_top
valid_out         <= '0';
s                 <= (others => '0');
case state is

  when wait_start =>

    valid_in_mon_1 <= valid_in;
    valid_in_mon_2 <= valid_in;
    if(valid_in = '1') then
      a_mon_1       <= x;
      b_mon_1       <= r_c;
      n_mon_1       <= m;

      a_mon_2           <= x"0001";
      b_mon_2           <= r_c;
      n_mon_2           <= m;
      next_w_numb       <= x"0043"; --3 mots mise a "0" sont rajoutes,
      afin d'eviter les debordement des multiplieurs/additionneurs
      next_n_c_reg      <= n_c;
      next_state        <= prepare_data;
      next_count_input <= x"0001";
      write_b_n       <= b"1"; -- la valeur de l'exposant « y »
      next_addr_exp   <= "000000001";
      next_addr_n     <= "000000001";
      next_bsize_reg <= bit_size-1;
    end if;

  when prepare_data =>
    next_count_input <= count_input+1;
    valid_in_mon_1   <= '1';
    valid_in_mon_2   <= '1';
    if(valid_in = '1') then
      a_mon_1         <= x;
      b_mon_1         <= r_c;
      n_mon_1         <= m;
      b_mon_2         <= r_c;
      n_mon_2         <= m;
      write_b_n      <= b"1";
      next_addr_exp <= addr_exp+1;
      next_addr_n    <= addr_n+1;
    end if;
    if(count_input = w_numb) then
      next_state     <= wait_constants;
      next_addr_n    <= (others => '0');
```

```
    next_addr_exp                          <= bsize_reg(12 downto 4);
    --Decodeur pour etablir le masque
    mask := bsize_reg(3 downto 0);
    case (mask) is
      when "0000"        => next_bit_counter <= "0000000000000001";
      when "0001"        => next_bit_counter <= "0000000000000010";
      when "0010"        => next_bit_counter <= "0000000000000100";
      when "0011"        => next_bit_counter <= "0000000000001000";
      when "0100"        => next_bit_counter <= "0000000000010000";
      when "0101"        => next_bit_counter <= "0000000000100000";
      when "0110"        => next_bit_counter <= "0000000001000000";
      when "0111"        => next_bit_counter <= "0000000010000000";
      when "1000"        => next_bit_counter <= "0000000100000000";
      when "1001"        => next_bit_counter <= "0000001000000000";
      when "1010"        => next_bit_counter <= "0000010000000000";
      when "1011"        => next_bit_counter <= "0000100000000000";
      when "1100"        => next_bit_counter <= "0001000000000000";
      when "1101"        => next_bit_counter <= "0010000000000000";
      when "1110"        => next_bit_counter <= "0100000000000000";
      when "1111"        => next_bit_counter <= "1000000000000000";
      when others        =>
    end case;
    next_count_input                       <= (others => '0');
  end if;

  --Attendre les sorties de Montgomery
when wait_constants =>

  --Ecrire des donnees dans la fifo
  if(valid_out_mon_1 = '1') then
    fifo_1_wr        <= '1';
    fifo_in          <= s_out_mon_1 & s_out_mon_2;
    next_count_input <= count_input+1;
    next_state       <= writting_cts_fifo;
  end if;

  --Ecrire les deux constantes initiales dans la fifo
when writting_cts_fifo =>
  fifo_1_wr        <= valid_out_mon_1;
  next_count_input <= count_input+1;
  if(count_input < x"40") then
    fifo_in        <= s_out_mon_1 & s_out_mon_2;
  end if;

  --Commencer la multiplication
  if(valid_out_mon_1 = '0') then
    next_count_input <= (others => '0');
    next_state       <= transition;
  end if;


when transition =>
  next_count_input <= count_input+1;

  if(count_input > 2) then
    next_count_input <= (others => '0');
                                --fifo_1_rd <= '1';
                                --next_addr_n <= addr_n+1;
    if((bit_counter and exp_out) = x"0000") then
      next_state     <= processing_data_0;
    else
```

```
      next_state     <= processing_data_1;
    end if;

  end if;


 --Exécuter multiplications successives (entrée dans la boucle de
  Montgomery)
when processing_data_1 =>

  if(count_input > x"0000")then
    valid_in_mon_1 <= '1';
    valid_in_mon_2 <= '1';
  end if;


  fifo_1_rd <= '1';

  a_mon_1 <= fifo_out(31 downto 16);
  b_mon_1 <= fifo_out(15 downto 0);
  n_mon_1 <= n_out;
  a_mon_2 <= fifo_out(31 downto 16);
  b_mon_2 <= fifo_out(31 downto 16);
  n_mon_2 <= n_out;
  next_addr_n      <= addr_n+1;
  next_count_input <= count_input +1;

  if(count_input = w_numb) then
    next_state <= wait_results;
  end if;

when processing_data_0 =>

  if(count_input > x"0000")then
    valid_in_mon_1 <= '1';
    valid_in_mon_2 <= '1';
  end if;

  fifo_1_rd <= '1';

  a_mon_1 <= fifo_out(15 downto 0);
  b_mon_1 <= fifo_out(15 downto 0);
  n_mon_1 <= n_out;

  a_mon_2 <= fifo_out(31 downto 16);
  b_mon_2 <= fifo_out(15 downto 0);
  n_mon_2 <= n_out;

  next_addr_n      <= addr_n+1;
  next_count_input <= count_input +1;

  if(count_input = w_numb) then
    next_state       <= wait_results;
    next_count_input <= (others => '0');
  end if;

when wait_results =>

  --Ecrire les données dans la fifo
  if(valid_out_mon_1 = '1') then
    fifo_1_wr        <= '1';
```

```
    fifo_in          <= s_out_mon_2 & s_out_mon_1;
    next_count_input <= x"0001";
    next_state       <= writting_results;
  end if;

when writting_results =>

  next_addr_n      <= (others => '0');
  fifo_1_wr        <= valid_out_mon_1;
  next_count_input <= count_input+1;
  if(count_input < x"40") then
    fifo_in        <= s_out_mon_2 & s_out_mon_1;
  end if;

  --necessaire pour la multiplication
  if(valid_out_mon_1 = '0') then
    next_count_input   <= (others => '0');
    next_state         <= prepare_next;
                                --Calcule du bit de l'exposant
                                --Décalage du masque
    next_bit_counter   <= '0'&bit_counter(15 downto 1);
    if(bit_counter = x"0001")
    then
      next_addr_exp    <= addr_exp -1;
      next_bit_counter <= "1000000000000000";
    end if;
    if((bit_counter = x"0001") and addr_exp = "000000000")
    then
      next_state <= final_mult;
      next_count_input <= (others => '0');
      next_addr_exp <= (others => '0');
    end if;
  end if;

when prepare_next =>
  next_state <= transition;
  next_count_input <= (others => '0');
  fifo_1_rd <= '0';

when final_mult =>
  next_count_input <= count_input+1;

  if(count_input > 2) then
    next_count_input <= (others => '0');
    next_state <= prepare_final;
  end if;

when prepare_final =>

  if(count_input > x"0000")then
    valid_in_mon_1 <= '1';
  end if;

  fifo_1_rd <= '1'; --Sortie de Montgomery

  a_mon_1 <= fifo_out(15 downto 0);
  if(count_input = x"0001") then
    b_mon_1 <= x"0001";
  end if;
  n_mon_1 <= n_out;
```

```
        next_addr_n <= addr_n+1;
        next_count_input <= count_input +1;

        --Afficher les resultatas finaux de Montgomery
        if(count_input = w_numb) then
          next_state <= wait_final;
          next_count_input <= (others =>'0');
        end if;

      when wait_final =>

        if(valid_out_mon_1 = '1') then
          valid_out <= '1';
          s <= s_out_mon_1;
          next_state <= show_final;
          next_count_input <= count_input +1;
        end if;

      when show_final =>
        valid_out <= '1';
        s <= s_out_mon_1;
        next_count_input <= count_input +1;
        if(count_input = x"40") then
          valid_out <= '0';
          next_state <= wait_start;
        end if;

    end case;

  end process;

end Behavioral;
```

Montgomery_mult.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity montgomery_mult is

  port(
    clk       : in  std_logic;
    reset     : in  std_logic;
    valid_in  : in  std_logic;
    a         : in  std_logic_vector(15 downto 0);
    b         : in  std_logic_vector(15 downto 0);
    n         : in  std_logic_vector(15 downto 0);
    s_prev    : in  std_logic_vector(15 downto 0);
    n_c       : in  std_logic_vector(15 downto 0);
    s         : out std_logic_vector(15 downto 0);
    valid_out : out std_logic
    );
```

```
end montgomery_mult;

architecture Behavioral of montgomery_mult is

  component montgomery_step is
                    port(
                      clk       : in  std_logic;
                      reset     : in  std_logic;
                      valid_in  : in  std_logic;
                      a         : in  std_logic_vector(15 downto 0);
                      b         : in  std_logic_vector(15 downto 0);
                      n         : in  std_logic_vector(15 downto 0);
                      s_prev    : in  std_logic_vector(15 downto 0);
                      n_c       : in  std_logic_vector(15 downto 0);
                      s         : out std_logic_vector(15 downto 0);
                      valid_out : out std_logic;
                      busy      : out std_logic;
                      b_req     : out std_logic;
                      a_out     : out std_logic_vector(15 downto 0);
                      n_out     : out std_logic_vector(15 downto 0);
                      c_step    : out std_logic;
                      stop      : in  std_logic
                                 );
  end component;

  component fifo_512_bram
    port (
      clk   : in  std_logic;
      rst   : in  std_logic;
      din   : in  std_logic_vector(15 downto 0);
      wr_en : in  std_logic;
      rd_en : in  std_logic;
      dout  : out std_logic_vector(15 downto 0);
      full  : out std_logic;
      empty : out std_logic);
  end component;

  component fifo_256_feedback
    port (
      clk   : in  std_logic;
      rst   : in  std_logic;
      din   : in  std_logic_vector(48 downto 0);
      wr_en : in  std_logic;
      rd_en : in  std_logic;
      dout  : out std_logic_vector(48 downto 0);
      full  : out std_logic;
      empty : out std_logic);
   end component;

  type arr_dat_out is array(0 to 7) of std_logic_vector(15 downto 0);
  type arr_val is array(0 to 7) of std_logic;
  type arr_b is array(0 to 7) of std_logic_vector(15 downto 0);

  signal b_reg, next_b_reg                                          :
arr_b;
  signal valid_mid, fifo_reqs, fifo_reqs_reg, next_fifo_reqs_reg, stops
:arr_val;
  signal a_out_mid, n_out_mid, s_out_mid                             :
arr_dat_out;
```

```
 --Signaux des fifos
  signal wr_en, rd_en, empty : std_logic;
  signal fifo_out            : std_logic_vector(15 downto 0);

  signal fifo_out_feedback, fifo_in_feedback : std_logic_vector(48 downto
0);
  signal read_fifo_feedback, empty_feedback  : std_logic;

  --Les entrees du premier PE
  signal a_in, s_in, n_in : std_logic_vector(15 downto 0);
  signal f_valid, busy_pe : std_logic;

  signal c_step, reg_c_step : std_logic;

  --Les signaux du conteurs
  signal count, next_count  : std_logic_vector(15 downto 0);

  signal wr_fifofeed : std_logic;

  type state_type is (rst_fifos,wait_start, process_data, dump_feed);
  signal state, next_state                    : state_type;
  signal reg_busy                             : std_logic;
  signal reset_fifos                          : std_logic;
  signal count_feedback, next_count_feedback : std_logic_vector(15 downto
0);

begin

--Fifo pour calculer le b
  fifo_b : fifo_512_bram port map (
    clk   => clk,
    rst   => reset_fifos,
    din   => b,
    wr_en => wr_en,
    rd_en => rd_en,
    dout  => fifo_out,
    empty => empty
    );

--Fifo pour le feedback du premier PE
  fifo_feed : fifo_256_feedback port map (
    clk   => clk,
    rst   => reset_fifos,
    din   => fifo_in_feedback,
    wr_en => wr_fifofeed,
    rd_en => read_fifo_feedback,
    dout  => fifo_out_feedback,
    empty => empty_feedback
    );

--Primer PE
  et_first : montgomery_step port map(
    clk       => clk,
    reset     => reset,
    valid_in  => f_valid,
    a         => a_in,
    b         => b_reg(0),
    n         => n_in,
    s_prev    => s_in,
    n_c       => n_c,
    s         => s_out_mid(0),
```

```
    valid_out => valid_mid(0),
    busy      => busy_pe,
    b_req     => fifo_reqs(0),
    a_out     => a_out_mid(0),
    n_out     => n_out_mid(0),
    c_step    => c_step,
    stop      => stops(0)
    );

--Dernier PE
  et_last : montgomery_step port map(
    clk       => clk,
    reset     => reset,
    valid_in  => valid_mid(6),
    a         => a_out_mid(6),
    b         => b_reg(7),
    n         => n_out_mid(6),
    s_prev    => s_out_mid(6),
    n_c       => n_c,
    s         => s_out_mid(7),
    valid_out => valid_mid(7),
    b_req     => fifo_reqs(7),
    a_out     => a_out_mid(7),
    n_out     => n_out_mid(7),
    stop      => stops(7)
    );

  g1     : for i in 1 to 6 generate
    et_i : montgomery_step port map(
      clk       => clk,
      reset     => reset,
      valid_in  => valid_mid(i-1),
      a         => a_out_mid(i-1),
      b         => b_reg(i),
      n         => n_out_mid(i-1),
      s_prev    => s_out_mid(i-1),
      n_c       => n_c,
      s         => s_out_mid(i),
      valid_out => valid_mid(i),
      b_req     => fifo_reqs(i),
      a_out     => a_out_mid(i),
      n_out     => n_out_mid(i),
      stop      => stops(i)
      );

  end generate g1;


  process(clk, reset)
  begin

    if(clk = '1' and clk'event) then

      if(reset = '1')then
        state               <= wait_start;
        count_feedback      <= (others => '0');
        reg_busy            <= '0';
        for i in 0 to 7 loop
          b_reg(i)          <= (others => '0');
          fifo_reqs_reg (i) <= '0';
          count             <= (others => '0');
```

```
          reg_c_step        <= '0';

        end loop;
      else
        state               <= next_state;
        reg_busy            <= busy_pe;
        count_feedback      <= next_count_feedback;
        for i in 0 to 7 loop
          b_reg(i)          <= next_b_reg(i);
          fifo_reqs_reg (i) <= next_fifo_reqs_reg(i);
          count             <= next_count;
          reg_c_step        <= c_step;
        end loop;
      end if;
    end if;

  end process;


  process(fifo_reqs_reg, fifo_out, b, fifo_reqs, b_reg, state, empty)
  begin


    for i in 0 to 7 loop
      next_b_reg(i)          <= b_reg(i);
      next_fifo_reqs_reg(i) <= fifo_reqs(i);
    end loop;

    if(state = wait_start) then
      next_b_reg(0)           <= b;
      next_fifo_reqs_reg(0)   <= '0';
      for i in 1 to 7 loop
        next_b_reg(i)         <= (others => '0');
        next_fifo_reqs_reg(i) <= '0';
      end loop;
    else
      for i in 0 to 7 loop
        if(fifo_reqs_reg(i) = '1' and empty = '0') then
          next_b_reg(i)       <= fifo_out;
        end if;
      end loop;
    end if;
  end process;


  process( valid_in, b, state, fifo_reqs, a_out_mid, n_out_mid, s_out_mid,
valid_mid, a, s_prev, n, busy_pe, empty_feedback, fifo_out_feedback, count,
reg_c_step, reset, reg_busy, count_feedback )

begin

    rd_en              <= fifo_reqs(0) or fifo_reqs(1) or fifo_reqs(2) or
fifo_reqs(3) or fifo_reqs(4) or fifo_reqs(5) or fifo_reqs(6) or
fifo_reqs(7);
    next_state         <= state;
    wr_en              <= '0';
    fifo_in_feedback   <=
a_out_mid(7)&n_out_mid(7)&s_out_mid(7)&valid_mid(7);
    read_fifo_feedback <= '0';
    wr_fifofeed        <= '0';
    --Le PE primaire
```

```
a_in               <= a;
s_in               <= s_prev;
n_in               <= n;
f_valid            <= valid_in;
reset_fifos        <= reset;
--Les sorties
s                  <= (others => '0');
valid_out          <= '0';

--Incrementer le compteur
next_count          <= count;
next_count_feedback <= count_feedback;
--incrementer le compteur dans le pipeleine
if(reg_c_step = '1') then
  next_count        <= count + 8;
end if;
    if( count = x"40") then
  s                 <= s_out_mid(0);
  valid_out         <= valid_mid(0);
end if;

for i in 0 to 7 loop
  stops(i) <= '0';
end loop;

case state is
  when wait_start =>
    --reset_fifos   <= '1';
    if(valid_in = '1') then
      reset_fifos <= '0';
      next_state  <= process_data;
    --wr_en <= '1';
    end if;

  when process_data =>
    wr_fifofeed <= valid_mid(7);
    if(valid_in = '1') then
      wr_en     <= '1';
    end if;
    if(empty_feedback = '0' and reg_busy = '0') then
      read_fifo_feedback <= '1';
      next_state <= dump_feed;
      next_count_feedback <= x"0000";
    end if;


    if( count >  x"43") then
      next_state <= wait_start;
                        --y
      for i in 0 to 7 loop
        stops(i) <= '1';
      end loop;
      next_count <= (others => '0');
    end if;

    --Vider la fifo de feedback
  when dump_feed =>
    if(empty_feedback='0')then
      next_count_feedback <= count_feedback+1;
    end if;
    wr_fifofeed <= valid_mid(7);
```

```
        read_fifo_feedback <= '1';
        a_in <= fifo_out_feedback(48 downto 33);
        n_in <= fifo_out_feedback(32 downto 17);
        s_in <= fifo_out_feedback(16 downto 1);
        f_valid <= fifo_out_feedback(0);
        if(empty_feedback='1') then
          next_state <= process_data;

        end if;
        if(count_feedback = x"42") then
          read_fifo_feedback <= '0';
          next_state <= process_data;
        end if;
       when rst_fifos =>
           next_state <= wait_start;
              reset_fifos   <= '1';
    end case;

  end process;
end Behavioral;
```

Montgomery_step

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;


entity montgomery_step is
  port(
    clk       : in  std_logic;
    reset     : in  std_logic;
    valid_in  : in  std_logic;
    a         : in  std_logic_vector(15 downto 0);
    b         : in  std_logic_vector(15 downto 0);
    n         : in  std_logic_vector(15 downto 0);
    s_prev    : in  std_logic_vector(15 downto 0);
    n_c       : in  std_logic_vector(15 downto 0);
    s         : out std_logic_vector(15 downto 0);
    valid_out : out std_logic;
    busy      : out std_logic;
    b_req     : out std_logic;
    a_out     : out std_logic_vector(15 downto 0);
    n_out     : out std_logic_vector(15 downto 0);
    c_step    : out std_logic;
    stop      : in  std_logic
    );
end montgomery_step;

architecture Behavioral of montgomery_step is

  component pe_wrapper
    port(
      clk          : in  std_logic;
      reset        : in  std_logic;
      ab_valid     : in  std_logic;
      valid_in     : in  std_logic;
      a            : in  std_logic_vector(15 downto 0);
      b            : in  std_logic_vector(15 downto 0);
```

```
      n            : in  std_logic_vector(15 downto 0);
      s_prev       : in  std_logic_vector(15 downto 0);
      n_c          : in  std_logic_vector(15 downto 0);
      s            : out std_logic_vector(15 downto 0);
      data_ready   : out std_logic;
      fifo_req     : out std_logic;
      m_val        : out std_logic;
      reset_the_PE : in  std_logic);
      end component;

  --Entrees

  signal ab_valid                                                    :
std_logic;
  signal valid_mont, fifo_read, m_val, valid_mont_out, reset_pe :
std_logic;


 --Sorties

  type state_type is (wait_valid, wait_m, mont_proc, getting_results,
prep_m,
  b_stable);
  signal state, next_state     : state_type;
  signal counter, next_counter : std_logic_vector(15 downto 0);
    --il compte les mots qui sont sortis pour aller coupure

  signal mont_input_a, mont_input_n, mont_input_s: std_logic_vector(15
downto 0);
  signal reg_constant, next_reg_constant, next_reg_input, reg_input :
std_logic_vector(47 downto 0);
  signal reg_out, reg_out_1, reg_out_2, reg_out_3, reg_out_4        :
std_logic_vector(31 downto 0);
  signal next_reg_out                              : std_logic_vector(31
downto 0);

  signal reg_input_1, reg_input_2, reg_input_3, reg_input_4, reg_input_5 :
std_logic_vector(47 downto 0);


begin

  mont : pe_wrapper port map (
    clk          => clk,
    reset        => reset,
    ab_valid     => ab_valid,
    a            => mont_input_a,
    b            => b,
    n            => mont_input_n,
    s_prev       => mont_input_s,
    n_c          => n_c,
    s            => s,
    valid_in     => valid_mont,
    data_ready   => valid_mont_out,
    m_val        => m_val,
    reset_the_PE => reset_pe
    );

  process(clk, reset)
  begin
    if(clk = '1' and clk'event) then
```

```
      if(reset = '1')then
        state        <= wait_valid;
        counter      <= (others => '0');
        reg_constant <= (others => '0');
        reg_input    <= (others => '0');
        reg_input_1  <= (others => '0');
        reg_input_2  <= (others => '0');
        reg_input_3  <= (others => '0');
        reg_input_4  <= (others => '0');

        reg_out   <= (others => '0');
        reg_out_1 <= (others => '0');
        reg_out_2 <= (others => '0');
        reg_out_3 <= (others => '0');
        reg_out_4 <= (others => '0');

      else
        reg_input   <= next_reg_input;
        reg_input_1 <= reg_input;
        reg_input_2 <= reg_input_1;
        reg_input_3 <= reg_input_2;
        reg_input_4 <= reg_input_3;
        reg_input_5 <= reg_input_4;

        reg_out   <= reg_input_4(47 downto 32) & reg_input_4(31 downto 16);
        reg_out_1 <= reg_out;
        reg_out_2 <= reg_out_1;
        reg_out_3 <= reg_out_2;
        reg_out_4 <= reg_out_3;

        state        <= next_state;
        counter      <= next_counter;
        reg_constant <= next_reg_constant;
      end if;
    end if;
  end process;

  process(state, valid_in, m_val, a, n, s_prev, counter, valid_mont_out,
stop, reg_constant, reg_input_5, reg_out_4)
  begin
    --reset_fifo <= '0';
    next_reg_input <= a&n&s_prev;
    --next_reg_out <= a&n;

    a_out <= reg_out_4(31 downto 16);
    n_out <= reg_out_4(15 downto 0);

    next_state   <= state;
    next_counter <= counter;
    --write_fifos <= valid_in;
    ab_valid     <= '0';
    valid_mont   <= '0';
    valid_out    <= '0';
    reset_pe     <= '0';
    busy         <= '1';
    b_req        <= '0';
    c_step       <= '0';

    mont_input_a      <= (others => '0');
    mont_input_n      <= (others => '0');
```

```
mont_input_s      <= (others => '0');
next_reg_constant <= reg_constant;

case state is
  when wait_valid =>
    busy     <= '0';
    reset_pe <= '1';
    if(valid_in = '1') then
      b_req            <= '1';
      next_state       <= b_stable;
      next_reg_constant <= a&n&s_prev;
    end if;


    when b_stable =>
    next_state          <= prep_m;
  when prep_m   =>
    mont_input_a        <= reg_constant(47 downto 32);
           --Ajouter au sensitivity
    mont_input_n        <= reg_constant(31 downto 16);
    mont_input_s        <= reg_constant(15 downto 0);
    ab_valid            <= '1';
    next_state          <= wait_m;



        when wait_m   =>
    --Nous maintenons les entrées pour calculer le m
    mont_input_a <= reg_constant(47 downto 32);
    --Ajouter au sensitivity
    mont_input_n <= reg_constant(31 downto 16);
    mont_input_s <= reg_constant(15 downto 0);


    if (m_val = '1')   -- m (qi) est pret
          valid_mont   <= '1';
      next_state   <= mont_proc;
      mont_input_a <= reg_input_5(47 downto 32);
      mont_input_n <= reg_input_5(31 downto 16);
      mont_input_s <= reg_input_5(15 downto 0);

    end if;

  when mont_proc =>

    valid_mont   <= '1';
    mont_input_a <= reg_input_5(47 downto 32);
    mont_input_n <= reg_input_5(31 downto 16);
    mont_input_s <= reg_input_5(15 downto 0);

    if(valid_mont_out = '1') then

      next_counter <= x"0000";
      next_state   <= getting_results;
    end if;

  when getting_results =>

    valid_out    <= '1';
    next_counter <= counter+1;
    valid_mont   <= '1';
```

```
        mont_input_a <= reg_input_5(47 downto 32);
        mont_input_n <= reg_input_5(31 downto 16);
        mont_input_s <= reg_input_5(15 downto 0);

        if(counter = (x"0042")) then -- 34 mot de 16 bits
          next_state <= wait_valid;
          c_step <= '1';
          reset_pe <= '1';
        end if;

    end case;

    if(stop='1') then
      next_state <= wait_valid;
      --reset_fifo <= '1';
      reset_pe <= '1';
    end if;

  end process;

end Behavioral;
```

pe_wrapper.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity pe_wrapper is

  port(
    clk          : in  std_logic;
    reset        : in  std_logic;
    ab_valid     : in  std_logic;
    valid_in     : in  std_logic;
    a            : in  std_logic_vector(15 downto 0);
    b            : in  std_logic_vector(15 downto 0);
    n            : in  std_logic_vector(15 downto 0);
    s_prev       : in  std_logic_vector(15 downto 0);
    n_c          : in  std_logic_vector(15 downto 0);
    s            : out std_logic_vector(15 downto 0);
    data_ready   : out std_logic;
    fifo_req     : out std_logic;
    m_val        : out std_logic;
    reset_the_PE : in  std_logic);


end pe_wrapper;

architecture Behavioral of pe_wrapper is

  component pe is
                port ( clk          : in  std_logic;
                       reset        : in  std_logic;
                       a_j          : in  std_logic_vector(15 downto 0);
                       b_i          : in  std_logic_vector(15 downto 0);
```

```
                        s_prev       : in  std_logic_vector(15 downto 0);
                              --la valeur precedente de la somme S
                        m            : in  std_logic_vector(15 downto 0);
                        n_j          : in  std_logic_vector(15 downto 0);
                        s_next       : out std_logic_vector(15 downto 0);
                        aj_bi        : out std_logic_vector(15 downto 0);
                        -- aj_bi=a*b
                        ab_valid_in  : in  std_logic;
                        valid_in     : in  std_logic;
                              ab_valid_out : out std_logic;
                        valid_out    : out std_logic;
                        fifo_req     : out std_logic);
  end component;

  component m_calc is
                      port(
                        clk        : in  std_logic;
                        reset      : in  std_logic;
                        ab         : in  std_logic_vector (15 downto 0);
                        t          : in  std_logic_vector (15 downto 0);
                        n_cons     : in  std_logic_vector (15 downto 0);
                        m          : out std_logic_vector (15 downto 0);
                        mult_valid : in  std_logic;
                        m_valid    : out std_logic);
  end component;

  signal aj_bi, m, next_m, m_out          : std_logic_vector(15 downto 0);
  signal mult_valid, valid_m, valid_m_reg : std_logic;

begin

  pe_0 : pe port map(
    clk          => clk,
    reset        => reset,
    a_j          => a,
    b_i          => b,
    s_prev       => s_prev,
    m            => m,
    n_j          => n,
    s_next       => s,
    aj_bi        => aj_bi,
    ab_valid_in  => ab_valid,
    valid_in     => valid_in,
    ab_valid_out => mult_valid,
    valid_out    => data_ready,
    fifo_req     => fifo_req);

  mcons_0 : m_calc port map(
    clk        => clk,
    reset      => reset,
    ab         => aj_bi,
    t          => s_prev,
    n_cons     => n_c,
    m          => m_out,
    mult_valid => mult_valid,
    m_valid    => valid_m);

  process(clk, reset)
  begin
    if(clk='1' and clk'event) then
```

```
      if(reset='1')then
        m <= (others=> '0' );
        valid_m_reg <= '0';
      else
        m <= next_m;
        valid_m_reg <= valid_m;
      end if;
    end if;
  end process;

  process(m_out,valid_m, valid_m_reg, m)
  begin
    m_val <= valid_m_reg;
    if(valid_m = '1' and valid_m_reg ='0') then
      next_m <= m_out;
    else
      next_m <= m;
    end if;
  end process;

end Behavioral;
```

pe.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;


entity pe is
  port ( clk          : in  std_logic;
         reset        : in  std_logic;
         a_j          : in  std_logic_vector(15 downto 0);
         b_i          : in  std_logic_vector(15 downto 0);
         s_prev       : in  std_logic_vector(15 downto 0);
         m            : in  std_logic_vector(15 downto 0);
         n_j          : in  std_logic_vector(15 downto 0);
         s_next       : out std_logic_vector(15 downto 0);
         aj_bi        : out std_logic_vector(15 downto 0);
         ab_valid_in  : in  std_logic;
         valid_in     : in  std_logic;
         ab_valid_out : out std_logic;
         valid_out    : out std_logic;
         fifo_req     : out std_logic);
end pe;

architecture Behavioral of pe is

  signal prod_aj_bi, next_prod_aj_bi, mult_aj_bi                    :
std_logic_vector(31 downto 0);
  signal prod_nj_m, next_prod_nj_m, mult_nj_m, mult_nj_m_reg        :
std_logic_vector(31 downto 0);
  signal sum_1, next_sum_1                                          :
std_logic_vector(31 downto 0);
```

```
  signal sum_2, next_sum_2                                          :
std_logic_vector(31 downto 0);
  signal ab_valid_reg, valid_out_reg, valid_out_reg2, valid_out_reg3 :
std_logic;
  signal n_reg, next_n_reg, s_prev_reg, next_s_prev_reg, ab_out_reg  :
std_logic_vector(15 downto 0);
  --signal prod_aj_bi_out, next_prod_aj_bi_out : std_logic_vector(15 downto
0);

begin


  mult_aj_bi <= a_j * b_i;
  mult_nj_m  <= n_reg *m;


  process(clk, reset)
  begin

    if(clk = '1' and clk'event)
    then
      if(reset = '1') then
        prod_aj_bi     <= (others => '0');
        prod_nj_m      <= (others => '0');
        sum_1          <= (others => '0');
        sum_2          <= (others => '0');
        ab_valid_reg   <= '0';
        n_reg          <= (others => '0');
        valid_out_reg  <= '0';
        valid_out_reg2 <= '0';
        valid_out_reg3 <= '0';
        s_prev_reg     <= (others => '0');
      else
        --prod_aj_bi_out <= next_prod_aj_bi_out;
        prod_aj_bi     <= next_prod_aj_bi;
        prod_nj_m      <= next_prod_nj_m;
        sum_1          <= next_sum_1;
        sum_2          <= next_sum_2;
        ab_valid_reg   <= ab_valid_in;
        ab_out_reg     <= mult_aj_bi(15 downto 0);
        n_reg          <= next_n_reg;
        valid_out_reg  <= valid_in;
        valid_out_reg2 <= valid_out_reg;
        valid_out_reg3 <= valid_out_reg2;
        s_prev_reg     <= next_s_prev_reg;
        --mult_nj_m_reg <= mult_nj_m;
      end if;
    end if;

  end process;

  process(s_prev, prod_aj_bi, prod_nj_m, sum_1, sum_2, mult_aj_bi,
mult_nj_m, valid_in, ab_valid_reg, n_j, n_reg, valid_out_reg3, s_prev_reg,
ab_out_reg)
  begin
    ab_valid_out      <= ab_valid_reg;
    aj_bi             <= ab_out_reg(15 downto 0);
    s_next            <= sum_2     (15 downto 0);
    fifo_req          <= valid_in;
    valid_out         <= valid_out_reg3;
    next_sum_1        <= sum_1;
```

```
    next_sum_2        <= sum_2;
    next_prod_nj_m    <= prod_nj_m;
    next_prod_aj_bi   <= prod_aj_bi;
    next_n_reg        <= n_reg;
    next_s_prev_reg   <= s_prev_reg;
    if(valid_in = '1') then
      next_s_prev_reg <= s_prev;
      next_n_reg      <= n_j;
      next_prod_aj_bi <= mult_aj_bi;
      next_prod_nj_m  <= mult_nj_m;
      next_sum_1      <= prod_aj_bi+sum_1(31 downto 16)+s_prev_reg;
      next_sum_2      <= prod_nj_m+sum_2 (31 downto 16) + sum_1(15 downto
0);
    else
      next_s_prev_reg <= (others => '0');
      next_n_reg      <= (others => '0');
      next_prod_aj_bi <= (others => '0');
      next_prod_nj_m  <= (others => '0');
      next_sum_1      <= (others => '0');
      next_sum_2      <= (others => '0');
    end if;


  end process;

end Behavioral;
```

m_calc.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity m_calc is

  port(
    clk        : in  std_logic;
    reset      : in  std_logic;
    ab         : in  std_logic_vector (15 downto 0);
    t          : in  std_logic_vector (15 downto 0);
    n_cons     : in  std_logic_vector (15 downto 0);
    m          : out std_logic_vector (15 downto 0);
    mult_valid : in  std_logic;
    m_valid    : out std_logic);
end m_calc;

architecture Behavioral of m_calc is



  signal sum_res, next_sum_res      : std_logic_vector(15 downto 0);
  signal mult_valid_1, mult_valid_2 : std_logic;
  signal mult                       : std_logic_vector(31 downto 0);
begin
```

```
  mult <= sum_res * n_cons;

  process(clk, reset)
  begin

    if(clk = '1' and clk'event) then

      if(reset = '1') then
        sum_res      <= (others => '0');
        mult_valid_1 <= '0';
        mult_valid_2 <= '0';
      else
        sum_res      <= next_sum_res;
        mult_valid_1 <= mult_valid;
        mult_valid_2 <= mult_valid_1;

      end if;

    end if;

  end process;

  process(ab, t, mult_valid_2)
  begin
    m            <= mult(15 downto 0);
    next_sum_res <= ab+t;
    m_valid      <= mult_valid_2;
  end process;

end Behavioral;
```

Core : n_c

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

--library UNISIM;
--use UNISIM.VComponents.all;

entity n_c_core is
  port (clk   : in  std_logic;
        m_lsw : in  std_logic_vector(15 downto 0);
        ce    : in  std_logic;
        n_c   : out std_logic_vector(15 downto 0);
        done  : out std_logic
        );
end n_c_core;

architecture Behavioral of n_c_core is

  type stateNC_type is (stNC_idle, stNC_step1, stNC_step2,
                        stNC_step3, stNC_step4, stNC_fin);

  signal stateNC       : stateNC_type;
  signal NC_complete   : std_logic := '0';
  signal NC_start      : std_logic := '0';
  signal LSW_M         : std_logic_vector(15 downto 0);
```

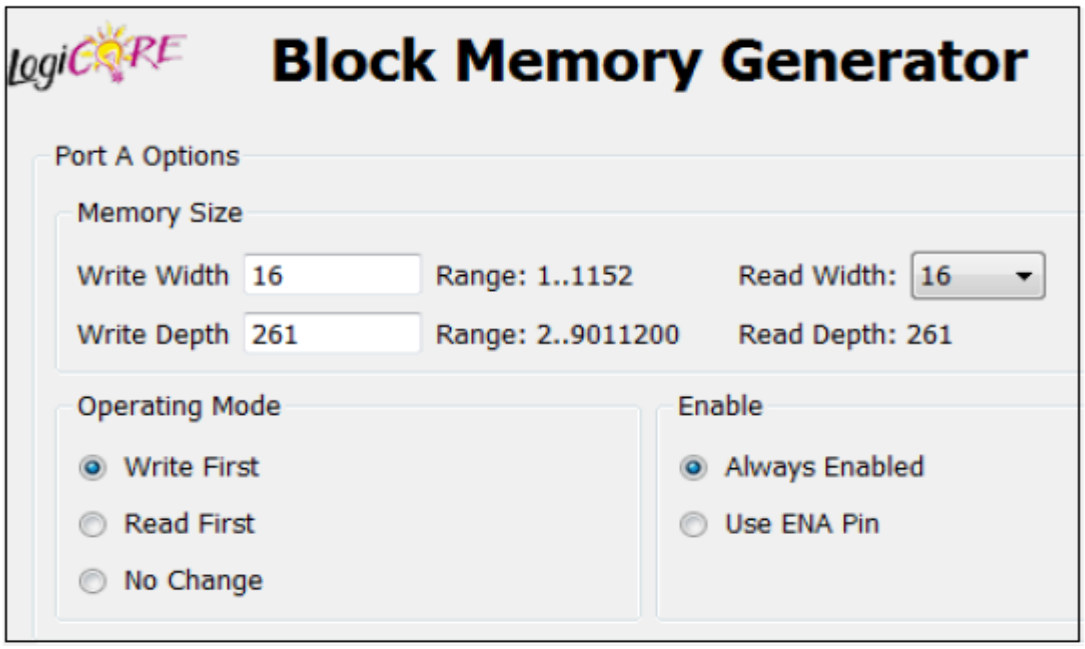
```
  signal adr_tbl       : std_logic_vector(2 downto 0);
  signal X0_tbl        : std_logic_vector(3 downto 0);
  signal Z0_tbl        : std_logic_vector(3 downto 0);
  signal X1_tbl        : std_logic_vector(3 downto 0);
  signal V1x9          : std_logic_vector(3 downto 0);
  signal TforNC        : std_logic_vector(15 downto 0);
  signal not_TforNCPl3 : std_logic_vector(15 downto 0);
  signal NC            : std_logic_vector(15 downto 0);
  signal t_NC          : std_logic_vector(15 downto 0);
  signal t_NC_out      : std_logic_vector(15 downto 0);
  signal b2equalb1     : std_logic;


-- dummy signals for simulation
  signal DUMMY_SIM0 : std_logic_vector(19 downto 0);
  signal DUMMY_SIM1 : std_logic_vector(19 downto 0);
  signal mul1, mul2 : std_logic_vector(35 downto 0);
begin

  mul1     <= ("00"&t_NC)*("00"&LSW_M);
  TforNC   <= mul1(15 downto 0);
  mul2     <= ("00"&t_NC)*("00"&not_TforNCPl3);
  t_NC_out <= mul2(15 downto 0);

-- TforNC_inst : MULT18X18
-- port map (
-- P(15 downto 0) => TforNC,
-- P(35 downto 16) => DUMMY_SIM0,       --only for sim, normally open
--      A(15 downto 0)  => t_NC,
--      A(17 downto 16) => "00",
--      B(15 downto 0)  => LSW_M,
--      B(17 downto 16) => "00"
--      );

-- NC_inst : MULT18X18
-- port map (
-- P(15 downto 0) => t_NC_out,
-- P(35 downto 16) => DUMMY_SIM1,       --only for sim, normally open
--      A(15 downto 0)  => t_NC,
--      A(17 downto 16) => "00",
--      B(15 downto 0)  => not_TforNCPl3,
--      B(17 downto 16) => "00"
--      );
-------------------------------------------
  WRITELSWM_PROCESS : process(clk, NC_complete)
  begin
    if(NC_complete = '1') then
      NC_start                      <= '0';
    elsif rising_edge(clk) then
      if(ce = '1') then
        LSW_M                       <= m_lsw;
        NC_start                    <= '1';
      end if;
    end if;
  end process WRITELSWM_PROCESS;
-------------------------------------------
  X0_ROM            : process(adr_tbl)
  begin
    case adr_tbl is
      when "000"          => X0_tbl <= X"F";
      when "001"          => X0_tbl <= X"5";
```

```
      when "010"          => X0_tbl <= X"3";
      when "011"          => X0_tbl <= X"9";
      when "100"          => X0_tbl <= X"7";
      when "101"          => X0_tbl <= X"D";
      when "110"          => X0_tbl <= X"B";
      when others         => X0_tbl <= X"1";
    end case;
  end process X0_ROM;
-------------------------------------------------------------------------------
  Z0_ROM            : process(adr_tbl)
  begin
    case adr_tbl is
      when "000"          => Z0_tbl <= X"F";
      when "001"          => Z0_tbl <= X"5";
      when "010"          => Z0_tbl <= X"3";
      when "011"          => Z0_tbl <= X"4";
      when "100"          => Z0_tbl <= X"C";
      when "101"          => Z0_tbl <= X"5";
      when "110"          => Z0_tbl <= X"3";
      when others         => Z0_tbl <= X"1";
    end case;
  end process Z0_ROM;
-------------------------------------------------------------------------------
  X1_ROM            : process(b2equalb1, LSW_M, Z0_tbl, V1x9)
  begin
    if(b2equalb1 = '0') then            -- b1==b2
      X1_tbl                        <= LSW_M(7 downto 4) + Z0_tbl;
    else                                -- b1 != b2
      X1_tbl                        <= V1x9 + Z0_tbl;
    end if;
  end process X1_ROM;
-------------------------------------------------------------------------------
  STATE_NC_PROCESS  : process(clk)
  begin
    if rising_edge(clk) then
      if(NC_start = '0') then
        NC_complete                 <= '0';
        stateNC                     <= stNC_idle;
        done                        <= '0';
      else
        case stateNC is
          when stNC_idle  =>
            done                    <= '0';
            stateNC                 <= stNC_step1;
            t_NC                    <= X"00" & X1_tbl & X0_tbl;
          when stNC_step1 =>
            t_NC                    <= t_NC_out;
            stateNC                 <= stNC_step2;
          when stNC_step2 =>
            t_NC                    <= t_NC_out;
            stateNC                 <= stNC_step3;
          when stNC_step3 =>
            t_NC                    <= t_NC_out;
            stateNC                 <= stNC_step4;
          when stNC_step4 =>
            t_NC                    <= t_NC_out;
            stateNC                 <= stNC_fin;
          when stNC_fin   =>
```

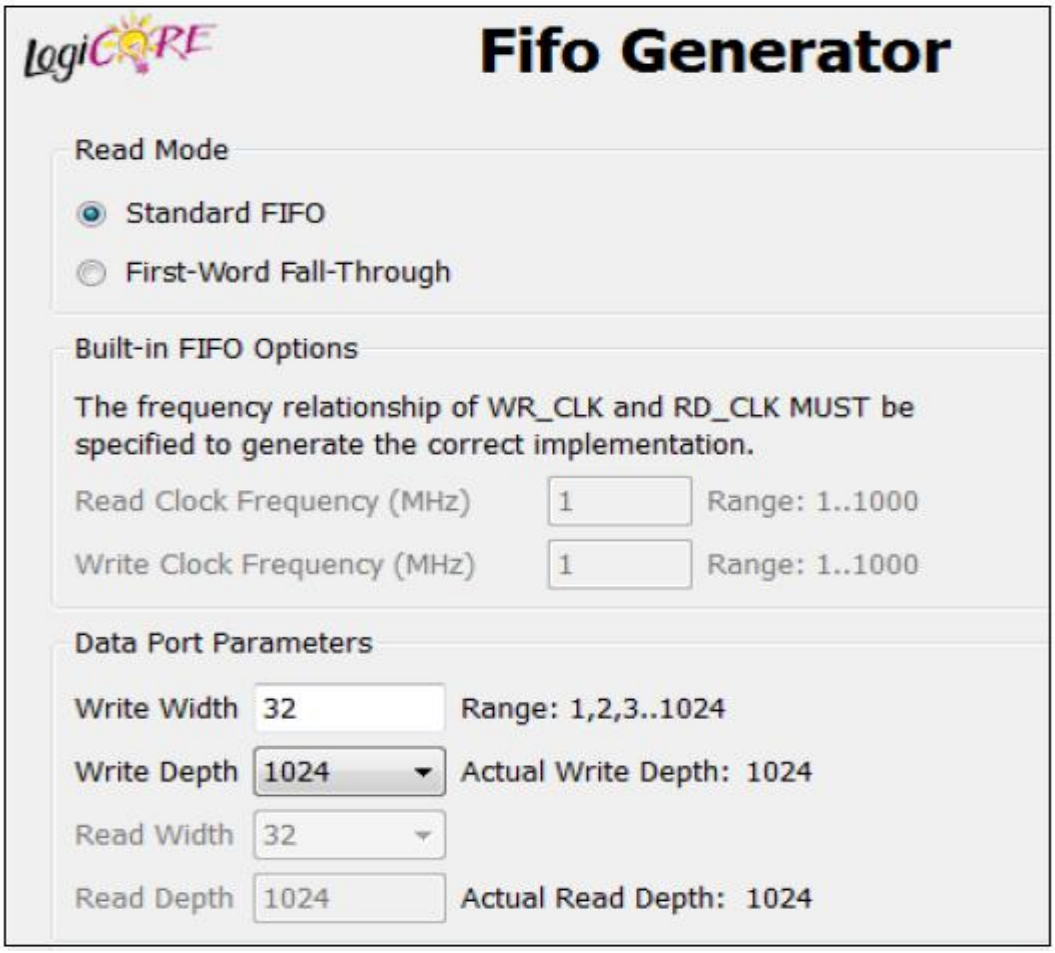
```
            NC_complete             <= '1';
            done                    <= '1';
            stateNC                 <= stNC_idle;
            NC                      <= (not (t_NC(15 downto 1))) & '1';
          when others      =>
            stateNC                 <= stNC_idle;
        end case;
      end if;
    end if;
  end process STATE_NC_PROCESS;
-------------------------------------------------------------------------------
----
  not_TforNCPl3                     <= (not TforNC) + 3;
  adr_tbl                           <= LSW_M(3 downto 1);
  V1x9                              <= (LSW_M(4) & "000") + LSW_M(7 downto
4);
  b2equalb1                         <= LSW_M(5) xor LSW_M(6);
  n_c                               <= NC;
end Behavioral;
```

Coregenerateurs de rsa_top_1024

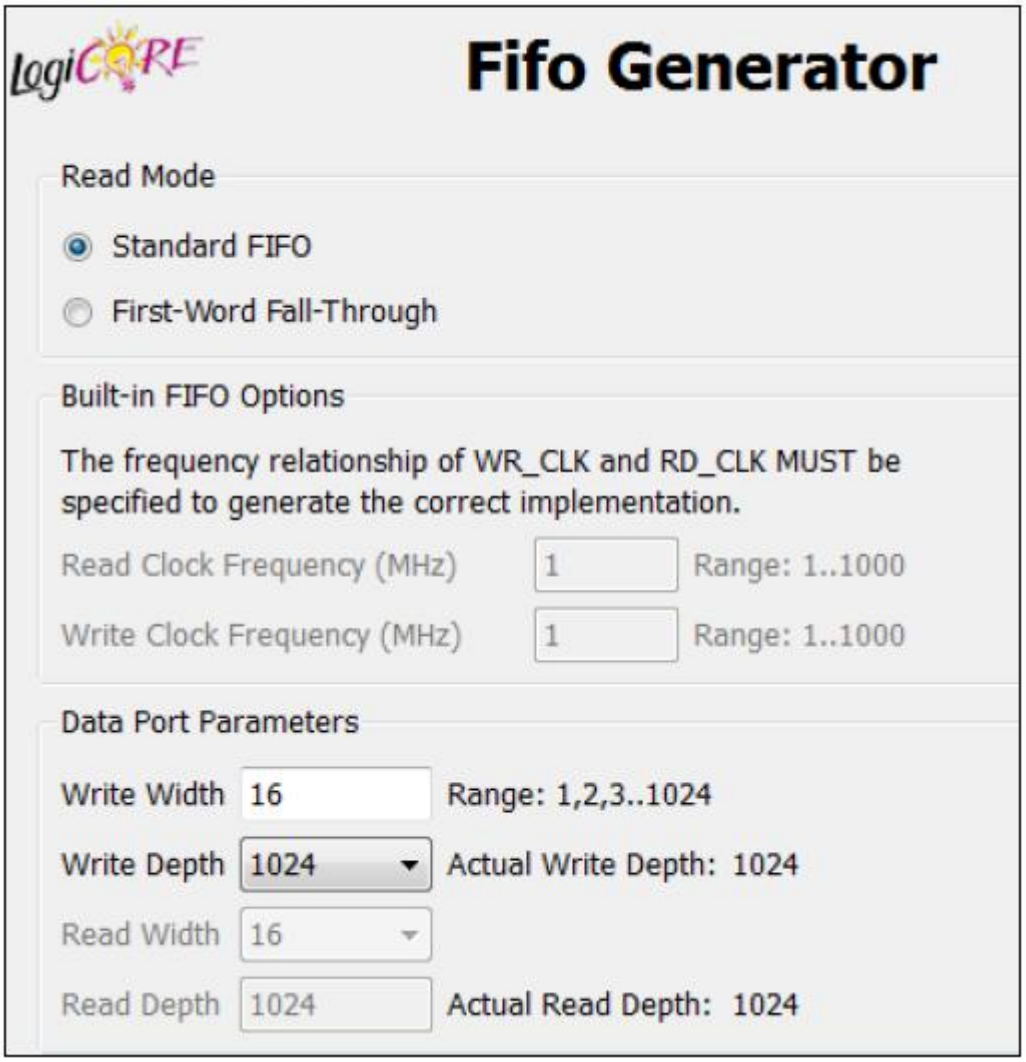
Mem_b



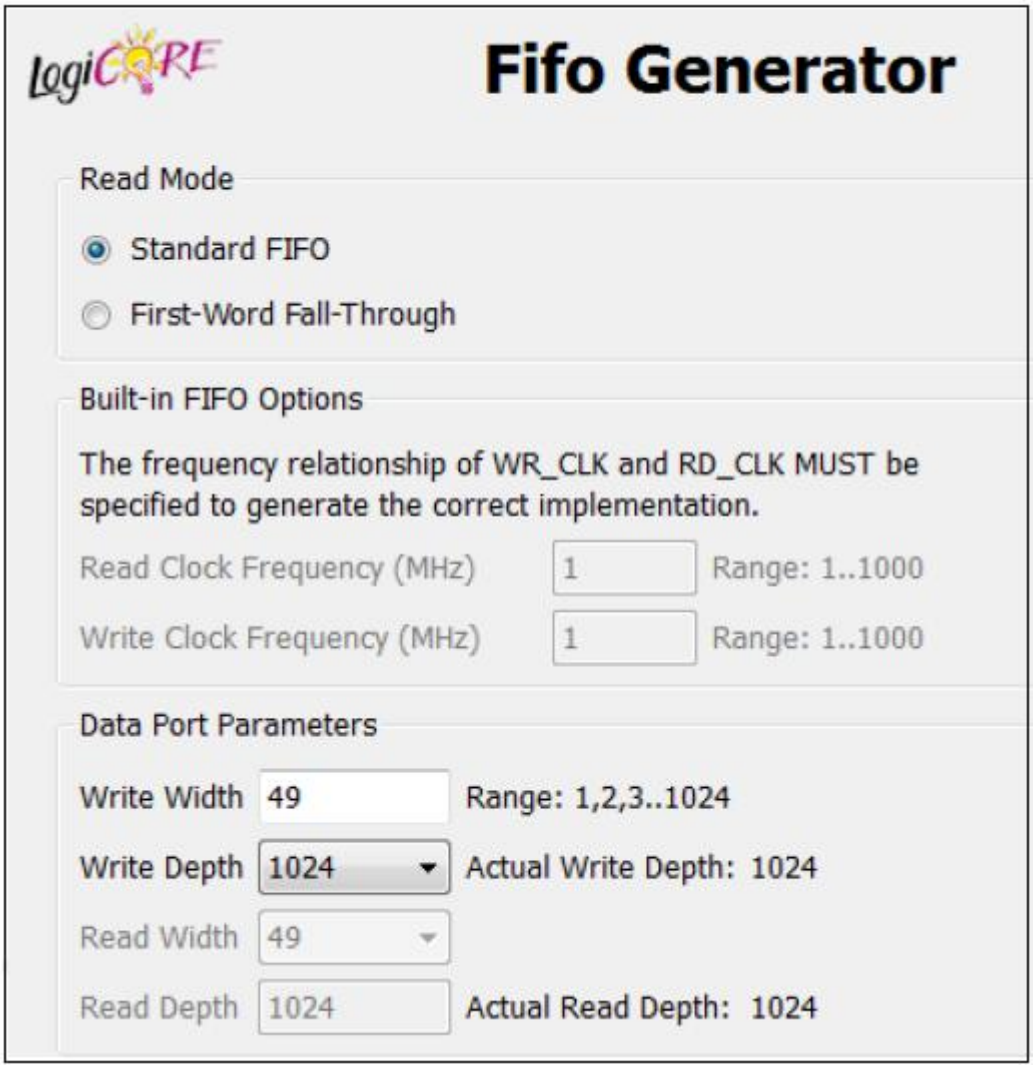
res_out_fifo


LogiCORE
Fifo Generator
Read Mode
Standard FIFO
First-Word Fall-Through
Built-in FIFO Options
The frequency relationship of WR_CLK and RD_CLK MUST be specified to generate the correct implementation.
Read Clock Frequency (MHz)
1
Range: 1..1000
Write Clock Frequency (MHz)
1
Range: 1..1000
Data Port Parameters
Write Width
32
Range: 1,2,3..1024
Write Depth
1024
Actual Write Depth: 1024
Read Width
32
Read Depth
1024
Actual Read Depth: 1024

Fifo_512_bram



Fifo_256_feedback



Testbench_rsa_1024 (64 mots de 16 bits)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY test_rsa_1024 IS
END test_rsa_1024;

ARCHITECTURE behavior OF test_rsa_1024 IS

    -- Component Declaration for the Unit Under Test (UUT)

  component rsa_top
    port(
```

```
      clk       : in  std_logic;
      reset     : in  std_logic;
      valid_in  : in  std_logic;
      start_in  : in  std_logic;
      x         : in  std_logic_vector(15 downto 0);
      y         : in  std_logic_vector(15 downto 0);
      m         : in  std_logic_vector(15 downto 0);
      r_c       : in  std_logic_vector(15 downto 0);
      s         : out std_logic_vector(15 downto 0);
      valid_out : out std_logic;
      bit_size  : in  std_logic_vector(15 downto 0)
      );
  end component;


  --Entrees
  signal clk       : std_logic                      := '0';
  signal reset     : std_logic                      := '0';
  signal valid_in  : std_logic                      := '0';
  signal start_in  : std_logic;
  signal x         : std_logic_vector(15 downto 0) := (others => '0');
  signal y         : std_logic_vector(15 downto 0) := (others => '0');
  signal m         : std_logic_vector(15 downto 0) := (others => '0');
  signal r_c       : std_logic_vector(15 downto 0) := (others => '0');
  signal bit_size  : std_logic_vector(15 downto 0) := x"0400";
  --Outputs
  signal s         : std_logic_vector(15 downto 0);
  signal valid_out : std_logic;

   -- Clock period definitions
  constant clk_period : time := 1ns;

BEGIN


  uut : rsa_top port map (
    clk       => clk,
    reset     => reset,
    valid_in  => valid_in,
    start_in  => start_in,
    x         => x,
    y         => y,
    m         => m,
    r_c       => r_c,
    s         => s,
    valid_out => valid_out,
    bit_size  => bit_size
    );

   -- Clock process definitions
   clk_process :process
   begin
            clk <= '0';
            wait for clk_period/2;
            clk <= '1';
            wait for clk_period/2;
   end process;


   stim_proc: process
   begin
```

```
     valid_in<='0';
  -- hold reset state for 100ms.
        reset <= '1';
  wait for 10ns;
  reset <= '0';
  wait for clk_period*10;

  -- insert stimulus here
m        <= x"b491";
start_in <= '1';
wait for clk_period;
start_in <= '0';
wait for clk_period*15;

valid_in <= '1';
x        <= x"f3ad";
y        <= x"42b1";
m        <= x"b491";
r_c      <= x"d9f6";
wait for clk_period;
x        <= x"8e40";
y        <= x"1ad3";
m        <= x"1417";
r_c      <= x"49a5";
wait for clk_period;
x        <= x"6af9";
y        <= x"a827";
m        <= x"b498";
r_c      <= x"b2e2";
wait for clk_period;
x        <= x"4e63";
y        <= x"0d64";
m        <= x"e1b7";
r_c      <= x"67b2";
wait for clk_period;
x        <= x"9600";
y        <= x"3f76";
m        <= x"e47c";
r_c      <= x"def7";
wait for clk_period;
x        <= x"68f4";
y        <= x"6670";
m        <= x"b186";
r_c      <= x"c036";
wait for clk_period;
x        <= x"5a12";
y        <= x"5a1c";
m        <= x"93f0";
r_c      <= x"225e";
wait for clk_period;
x        <= x"d62e";
y        <= x"4844";
m        <= x"b183";
r_c      <= x"a3bf";
wait for clk_period;
x        <= x"8fc1";
y        <= x"d5f2";
m        <= x"f8f1";
r_c      <= x"4ee7";
wait for clk_period;
x        <= x"031d";
```

```
y        <= x"b65a";
m        <= x"eed1";
r_c      <= x"1f7c";
wait for clk_period;
x        <= x"f496";
y        <= x"034f";
m        <= x"0083";
r_c      <= x"fe7f";
wait for clk_period;
x        <= x"1268";
y        <= x"9635";
m        <= x"981c";
r_c      <= x"c626";
wait for clk_period;
x        <= x"2e5a";
y        <= x"386e";
m        <= x"6441";
r_c      <= x"8662";
wait for clk_period;
x        <= x"c1d6";
y        <= x"fb73";
m        <= x"fcd8";
r_c      <= x"102e";
wait for clk_period;
x        <= x"cd8f";
y        <= x"5623";
m        <= x"cbf0";
r_c      <= x"1f5b";
wait for clk_period;
x        <= x"e4d2";
y        <= x"9041";
m        <= x"e3ca";
r_c      <= x"069a";
wait for clk_period;
x        <= x"36c6";
y        <= x"99da";
m        <= x"41d9";
r_c      <= x"db3c";
wait for clk_period;
x        <= x"df4a";
y        <= x"cd68";
m        <= x"b7a0";
r_c      <= x"e31a";
wait for clk_period;
x        <= x"8e40";
y        <= x"9a94";
m        <= x"146e";
r_c      <= x"5b66";
wait for clk_period;
x        <= x"6af9";
y        <= x"ccc8";
m        <= x"4776";
r_c      <= x"7c5e";
wait for clk_period;
x        <= x"4e63";
y        <= x"ed49";
m        <= x"ec50";
r_c      <= x"7cbe";
wait for clk_period;
x        <= x"9600";
y        <= x"4d25";
```

```
m        <= x"c07c";
r_c      <= x"b50f";
wait for clk_period;
x        <= x"68f4";
y        <= x"3b8e";
m        <= x"e698";
r_c      <= x"09ee";
wait for clk_period;
x        <= x"5a12";
y        <= x"36d5";
m        <= x"d85f";
r_c      <= x"e6d4";
wait for clk_period;
x        <= x"d62e";
y        <= x"3a75";
m        <= x"729c";
r_c      <= x"0559";
wait for clk_period;
x        <= x"8fc1";
y        <= x"77a3";
m        <= x"19b6";
r_c      <= x"78eb";
wait for clk_period;
x        <= x"d2cd";
y        <= x"367f";
m        <= x"05d3";
r_c      <= x"efa9";
wait for clk_period;
x        <= x"c6e4";
y        <= x"68de";
m        <= x"cacd";
r_c      <= x"aae5";
wait for clk_period;
x        <= x"4a36";
y        <= x"59a4";
m        <= x"e16f";
r_c      <= x"f70c";
wait for clk_period;
x        <= x"f6df";
y        <= x"9f89";
m        <= x"f67b";
r_c      <= x"e501";
wait for clk_period;
x        <= x"061c";
y        <= x"ed71";
m        <= x"7066";
r_c      <= x"83a9";
wait for clk_period;
x        <= x"06c8";
y        <= x"059f";
m        <= x"08de";
r_c      <= x"049f";
wait for clk_period;
x        <= x"f3ad";
y        <= x"42b1";
m        <= x"b491";
r_c      <= x"d9f6";
wait for clk_period;
x        <= x"8e40";
y        <= x"1ad3";
m        <= x"1417";
```

```
r_c      <= x"49a5";
wait for clk_period;
x        <= x"6af9";
y        <= x"a827";
m        <= x"b498";
r_c      <= x"b2e1";
wait for clk_period;
x        <= x"4e63";
y        <= x"0d64";
m        <= x"e1b7";
r_c      <= x"67b2";
wait for clk_period;
x        <= x"9600";
y        <= x"3f76";
m        <= x"e47c";
r_c      <= x"def7";
wait for clk_period;
x        <= x"68f4";
y        <= x"6670";
m        <= x"b186";
r_c      <= x"c036";
wait for clk_period;
x        <= x"5a12";
y        <= x"5a1c";
m        <= x"93f0";
r_c      <= x"225e";
wait for clk_period;
x        <= x"d62e";
y        <= x"4844";
m        <= x"b183";
r_c      <= x"a3bf";
wait for clk_period;
x        <= x"8fc1";
y        <= x"d5f2";
m        <= x"f8f1";
r_c      <= x"4ee7";
wait for clk_period;
x        <= x"031d";
y        <= x"b65a";
m        <= x"eed1";
r_c      <= x"1f7c";
wait for clk_period;
x        <= x"f496";
y        <= x"034f";
m        <= x"0083";
r_c      <= x"fe7f";
wait for clk_period;
x        <= x"1268";
y        <= x"9635";
m        <= x"981c";
r_c      <= x"c626";
wait for clk_period;
x        <= x"2e5a";
y        <= x"386e";
m        <= x"6441";
r_c      <= x"8662";
wait for clk_period;
x        <= x"c1d6";
y        <= x"fb73";
m        <= x"fcd8";
r_c      <= x"102e";
```

```
wait for clk_period;
x        <= x"cd8f";
y        <= x"5623";
m        <= x"cbf0";
r_c      <= x"1f5b";
wait for clk_period;
x        <= x"e4d2";
y        <= x"9041";
m        <= x"e3ca";
r_c      <= x"069a";
wait for clk_period;
x        <= x"36c6";
y        <= x"99da";
m        <= x"41d9";
r_c      <= x"db3c";
wait for clk_period;
x        <= x"df4a";
y        <= x"cd68";
m        <= x"b7a0";
r_c      <= x"e31a";
wait for clk_period;
x        <= x"8e40";
y        <= x"9a94";
m        <= x"146e";
r_c      <= x"5b66";
wait for clk_period;
x        <= x"6af9";
y        <= x"ccc8";
m        <= x"4776";
r_c      <= x"7c5e";
wait for clk_period;
x        <= x"4e63";
y        <= x"ed49";
m        <= x"ec50";
r_c      <= x"7cbe";
wait for clk_period;
x        <= x"9600";
y        <= x"4d25";
m        <= x"c07c";
r_c      <= x"b50f";
wait for clk_period;
x        <= x"68f4";
y        <= x"3b8e";
m        <= x"e698";
r_c      <= x"09ee";
wait for clk_period;
x        <= x"5a12";
y        <= x"36d5";
m        <= x"d85f";
r_c      <= x"e6d4";
wait for clk_period;
x        <= x"d62e";
y        <= x"3a75";
m        <= x"729c";
r_c      <= x"0559";
wait for clk_period;
x        <= x"8fc1";
y        <= x"77a3";
m        <= x"19b6";
r_c      <= x"78eb";
wait for clk_period;
```

```
    x        <= x"d2cd";
    y        <= x"367f";
    m        <= x"05d3";
    r_c      <= x"efa9";
    wait for clk_period;
    x        <= x"c6e4";
    y        <= x"68de";
    m        <= x"cacd";
    r_c      <= x"aae5";
    wait for clk_period;
    x        <= x"4a36";
    y        <= x"59a4";
    m        <= x"e16f";
    r_c      <= x"f70c";
    wait for clk_period;
    x        <= x"f6df";
    y        <= x"9f89";
    m        <= x"f67b";
    r_c      <= x"e501";
    wait for clk_period;
    x        <= x"061c";
    y        <= x"ed71";
    m        <= x"7066";
    r_c      <= x"83a9";
    wait for clk_period;
    x        <= x"06c8";
    y        <= x"059f";
    m        <= x"08de";
    r_c      <= x"049f";
   wait for clk_period;
valid_in <='0';


      wait;
   end process;

END;
```

Constant_gen.c (calcul de r_c et n_c a partir du modulo ‘’m’’)

```
#include <stdio.h>
#include <gmp.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

#define uint16 unsigned short int
#define uint32 unsigned int
#define uint64 unsigned long long int
#define WORD_SIZE 32


uint32 getMpzSize(mpz_t n)
{

   return (mpz_size(n)*2);
```

```
}

uint16 getlimb(mpz_t n, int i)
{
   uint32 aux=mpz_getlimbn(n,i/2);
   if(i%2 == 0)
   return (uint16) (aux&0xffff);
   return (uint16) (aux>>16);

}


int main()
{
    int i;

    mpz_t m,x,y,r,r_aux, n_cons, zero, recons;

    char *template;

      mpz_init_set_str(m,
      "08de7066f67be16fcacd05d319b6729cd85fe698c07cec504776146eb7a041d9e3ca
      cbf0fcd86441981c0083eed1f8f1b18393f0b186e47ce1b7b4981417b49108de7066f
      67be16fcacd05d319b6729cd85fe698c07cec504776146eb7a041d9e3cacbf0fcd864
      41981c0083eed1f8f1b18393f0b186e47ce1b7b4981417b491",16);
      //mpz_init_set_str(m,"c3217fff",16);

    mpz_init(r);
    mpz_init(r_aux);
    mpz_init(n_cons);
    mpz_init_set_ui(zero,0);

    //Calculo de la constante r_c et n_c de montgomery
    mpz_ui_pow_ui(r,2,16*(getMpzSize(m)+1));
    mpz_mul(r_aux,r,r);
    mpz_mod(r_aux,r_aux,m);

    mpz_sub(n_cons,zero,m);
    mpz_invert(n_cons,n_cons,r);

    printf    ("n_c <= %x;\n\n", (uint32)mpz_getlimbn(n_cons,0)&0xffff);
    gmp_printf("r_c <= %Zx\n\n", r_aux);

    return 0;
}
```

Annexe D : EDK /XPS

Module : emb_rsa

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity emb_rsa is
 port(
    clk       : in  std_logic;
    reset     : in  std_logic;
    valid_in  : in  std_logic;
    CE        : in  std_logic;
    Din       : in  std_logic_vector (31 downto 0);
--  m         : in  std_logic_vector (15 downto 0);
--  r_c       : in  std_logic_vector (15 downto 0;
--  n_c       : in  std_logic_vector (15 downto 0);
    s         : out std_logic_vector (15 downto 0);
    valid_out : out std_logic
    );
end emb_rsa;

architecture Behavioral of emb_rsa is

component rsa_top
    port(
      clk       : in  std_logic;
      reset     : in  std_logic;
      valid_in  : in  std_logic;
      x         : in  std_logic_vector(15 downto 0);
      y         : in  std_logic_vector(15 downto 0);
      m         : in  std_logic_vector(15 downto 0);
      r_c       : in  std_logic_vector(15 downto 0);
      n_c       : in  std_logic_vector(15 downto 0);
      s         : out std_logic_vector(15 downto 0);
      valid_out : out std_logic;
      bit_size  : in  std_logic_vector(15 downto 0)
      );
  end component;

   signal cnt         : integer;
   signal N_cnt_out   :  std_logic_vector(0 to 31);
   signal s_n1        :  std_logic_vector(15 downto 0);
   signal s_n2        :  std_logic_vector(15 downto 0);
   signal N_M         :  std_logic_vector(15 downto 0);

   constant Modulo    :  std_logic_vector(511 downto 0):
="0000100011011110011100000110011011110110011110111110000101101111110010101
1001101000001011101001100011001101101100111001010011100110110000101111111110
0110100110001100000001111100111011000101000001000111011101100001010001101111
0101101111010000001000001110110011110001111001010110010111111000011111110011
0110000110010001000001100110000001110000000000100000111110111011010001111111
0001111000110110001100000111001001111110000101100011000011011100100011111100
111000011011011110110100100110000001010000010111101101001001 0001"; --
valeur de m en binaire
```

```
   signal N_r_c      : std_logic_vector(15  downto 0);
   constant rr       : std_logic_vector(511 downto 0):
="0000010000000000010111101110001100110110101010110010010100101000010110101011
1110100100111111001101100011001011100010001000100011010001100010111001010110
0101011001111110001111100000111110111010000011000111111101100110010011011000
1011111001000110110000101111101011000011110010011011001001011010000110001010
1111010001101111010000100100110011011011000001010110010010100100011011001111
0100010101000000100111011110011011101111110101111001000000100011101110010100
0101000001010010100101110010110101101110111010011111010101111001";  --
valeur de r_c en binaire

   constant N_n_c    : std_logic_vector(15 downto 0)   :="0111001100011111";
   constant N_bit_size : std_logic_vector(15 downto 0) :="0000001000000000";

begin

s_n1<=Din (15 downto 0);
s_n2<=Din (31 downto 16);


-- Procedure du multiplexeur
-- garder m et r_c comme constants de 512 bits

CE<='0';

COUNTER1_PROCESS: process (Clk,cnt,CE,RESET )

begin
  if RESET = '1' then
      cnt <= 0;
            elsif Clk'event and Clk = '1' then
            if CE='1' then
         cnt <= cnt + 1;
         if cnt=5 then
              N_rst_operation<='1';
         end if;
      end if;
  end if;

end process COUNTER1_PROCESS;

SLAVE_Read_PROC : process( Clk,N_rst_operation,N_cnt_out ) is
begin
if N_rst_operation = '0' then
        S_out <= (others => '0');
    elsif Clk'event and Clk = '1' then
        case N_cnt_out is
                  when "00000000000000000000000000100000" =>
                        N_M<= modulo(496 to 511);
                        N_r_c<= rr(496 to 511);
                  when "00000000000000000000000000011111" =>
                        N_M<= modulo(480 to 495);
                        N_r_c<= rr(480 to 495);
                  when "00000000000000000000000000011110" =>
                        N_M<= modulo(464 to 479);
                        N_r_c<= rr(464 to 479);
                  when "00000000000000000000000000011101" =>
                        N_M<= modulo(448 to 463);
                        N_r_c<= rr(448 to 463);
                  when "00000000000000000000000000011100" =>
                        N_M<= modulo(432 to 447);
```

```
      N_r_c<= rr(432 to 447);
when "00000000000000000000000000011011" =>
      N_M<= modulo(416 to 431);
      N_r_c<= rr(416 to 431);
when "00000000000000000000000000011010" =>
      N_M<= modulo(400 to 415);
      N_r_c<= rr(400 to 415);
when "00000000000000000000000000011001" =>
      N_M<= modulo(384 to 399);
      N_r_c<= rr(384 to 399);
when "00000000000000000000000000011000" =>
      N_M<= modulo(368 to 383);
      N_r_c<= rr(368 to 383);
when "00000000000000000000000000010111" =>
      N_M<= modulo(352 to 367);
      N_r_c<= rr(352 to 367);
when "00000000000000000000000000010110" =>
      N_M<= modulo(336 to 351);
      N_r_c<= rr(336 to 351);
when "00000000000000000000000000010101" =>
      N_M<= modulo(320 to 335);
      N_r_c<= rr(320 to 335);
when "00000000000000000000000000010100" =>
      N_M<= modulo(304 to 319);
      N_r_c<= rr(304 to 319);
when "00000000000000000000000000010011" =>
      N_M<= modulo(288 to 303);
      N_r_c<= rr(288 to 303);
when "00000000000000000000000000010010" =>
      N_M<= modulo(272 to 287);
      N_r_c<= rr(272 to 287);
when "00000000000000000000000000010001" =>
      N_M<= modulo(256 to 271);
      N_r_c<= rr(256 to 271);
when "00000000000000000000000000010000" =>
      N_M<= modulo(240 to 255);
      N_r_c<= rr(240 to 255);
when "00000000000000000000000000001111" =>
      N_M<= modulo(224 to 239);
      N_r_c<= rr(224 to 239);
when "00000000000000000000000000001110" =>
      N_M<= modulo(208 to 223);
      N_r_c<= rr(208 to 223);
when "00000000000000000000000000001101" =>
      N_M<= modulo(192 to 207);
      N_r_c<= rr(192 to 207);
when "00000000000000000000000000001100" =>
      N_M<= modulo(176 to 191);
      N_r_c<= rr(176 to 191);
when "00000000000000000000000000001011" =>
      N_M<= modulo(160 to 175);
      N_r_c<= rr(160 to 175);
when "00000000000000000000000000001010" =>
      N_M<= modulo(144 to 159);
      N_r_c<= rr(144 to 159);
when "00000000000000000000000000001001" =>
      N_M<= modulo(128 to 143);
      N_r_c<= rr(128 to 143);
when "00000000000000000000000000001000" =>
      N_M<= modulo(112 to 127);
      N_r_c<= rr(112 to 127);
```

```
                    when "00000000000000000000000000000111" =>
                          N_M<= modulo(96 to 111);
                          N_r_c<= rr(96 to 111);
                    when "00000000000000000000000000000110" =>
                          N_M<= modulo(80 to 95);
                          N_r_c<= rr(80 to 95);
                    when "00000000000000000000000000000101" =>
                          N_M<= modulo(64 to 79);
                          N_r_c<= rr(64 to 79);
                    when "00000000000000000000000000000100" =>
                          N_M<= modulo(48 to 63);
                          N_r_c<= rr(48 to 63);
                    when "00000000000000000000000000000011" =>
                          N_M<= modulo(32 to 47);
                          N_r_c<= rr(32 to 47);
                    when "00000000000000000000000000000010" =>
                          N_M<= modulo(16 to 31);
                          N_r_c<= rr(16 to 31);
                    when "00000000000000000000000000000001" =>
                          N_M<= modulo(0 to 15);
                          N_r_c<= rr(0 to 15);
                    when others => null;
        end case;
    end if;
end process;


U: rsa_top
  port map(
    clk       => clk,
    reset     => reset,
    valid_in  => '1',
    x         => s_n1,
    y         => s_n2,
    m         => N_M,
    r_c       => N_r_c,
    n_c       => N_n_c,
    s         => s,
    valid_out => valid_out,
    bit_size  => N_bit_size,
    );

end Behavioral;
```

Module : my_ip

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity my_ip is
      port
      (
            -- DO NOT EDIT BELOW THIS LINE
            -- Bus protocol ports, do not add or delete.
            FSL_Clk        : in     std_logic;
```

```
            FSL_Rst         : in     std_logic;
            FSL_S_Clk       : out    std_logic;
            FSL_S_Read      : out    std_logic;
            FSL_S_Data      : in     std_logic_vector(0 to 31);
            FSL_S_Control   : in     std_logic;
            FSL_S_Exists    : in     std_logic;
            FSL_M_Clk       : out    std_logic;
            FSL_M_Write     : out    std_logic;
            FSL_M_Data      : out    std_logic_vector(0 to 31);
            FSL_M_Control   : out    std_logic;
            FSL_M_Full      : in     std_logic
            -- DO NOT EDIT ABOVE THIS LINE
      );

attribute SIGIS : string;
attribute SIGIS of FSL_Clk   : signal is "Clk";
attribute SIGIS of FSL_S_Clk : signal is "Clk";
attribute SIGIS of FSL_M_Clk : signal is "Clk";

end my_ip;


architecture EXAMPLE of my_ip is

   -- Total number of input data.
signal Dout : std_logic_vector(15 downto 0);


component emb_rsa
 port(
    clk       : in  std_logic;
    reset     : in  std_logic;
    CE        : in  std_logic;
    valid_in  : in  std_logic;
    x         : in  std_logic_vector(15 downto 0);
    y         : in  std_logic_vector(15 downto 0);
    s         : out std_logic_vector(15 downto 0);
    valid_out : out std_logic;
    );
end emb_rsa;


begin
   -- CAUTION:
   -- The sequence in which data are read in and written out should be
   -- consistent with the sequence they are written and read in the
   -- driver's my_ip.c file

U:
port map(
    clk        => FSL_clk,
    reset      => FSL_RST,
       CE         => FSL_S_Exists,
    valid_in   => '1',
    din        => FSL_S_Data,
    s         =>  Dout,
    valid_out =>  open
    );

   FSL_S_Read  <= FSL_S_Exists   when state = Read_Inputs   else '0';
   FSL_M_Write <= not FSL_M_Full when state = Write_Outputs else '0';
```

```
    FSL_M_Data <= "0000000000000000"&Dout;

 process (FSL_Clk)
 begin
    if FSL_Clk'event and FSL_Clk='0' then
          FSL_S_Read <= FSL_S_Exists;
    end if;
 end process;

 FSL_M_Control<='0';

 end architecture EXAMPLE;
```

Generation du Fichier testbench dans la Bram suivant le chemin : C:\Virtex_5\TestApp_Memory.c

```
* Xilinx EDK 9.1 EDK_J.19
 *
 * This file is a sample test application
 *
 * This application is intended to test and/or illustrate some
 * functionality of your system.  The contents of this file may
 * vary depending on the IP in your system and may use existing
 * IP driver functions.  These drivers will be generated in your
 * XPS project when you run the "Generate Libraries" menu item
 * in XPS.
 *
 * Your XPS project directory is at:
 *    D:\my_project_FPGA_EDK\multiplication_modulaire\edf_project_fsl\
 */


// Located in: microblaze_0/include/xparameters.h
#include "xparameters.h"
#include "mb_interface.h"
#include "xbasic_types.h"
#include "xuartlite.h"
#include "xutil.h"
#include "stdio.h"
#include "xstatus.h"
#include "xio.h"


int main(void){

//int input_0[0];
//int i;

Xuint32 i;

int temp[3];

int input_0[12];

 for (i=0;i<12;i++){
```

```
      input_0[0]  = 0x42b1f3ad;
      input_0[1]  = 0x1ad38e40;
      input_0[2]  = 0xa8276af9;
      input_0[3]  = 0x0d644e63;
      input_0[4]  = 0x3f769600;
      input_0[5]  = 0x667068f4;
      input_0[6]  = 0x5a1c5a12;
      input_0[7]  = 0x4844d62e;
      input_0[8]  = 0xd5f28fc1;
      input_0[9]  = 0xb65a031d;
      input_0[10] = 0x034ff496;
      input_0[11] = 0x96351268;
      input_0[12] = 0x386e2e5a;
      input_0[13] = 0xfb73c1d6;
      input_0[14] = 0x5623cd8f;
      input_0[15] = 0x9041e4d2;
      input_0[16] = 0x99da36c6;
      input_0[17] = 0xcd68df4a;
      input_0[18] = 0x9a948e40;
      input_0[19] = 0xccc86af9;
      input_0[20] = 0xed494e63;
      input_0[21] = 0x4d259600;
      input_0[22] = 0x3b8e68f4;
      input_0[23] = 0x36d55a12;
      input_0[24] = 0x3a75d62e;
      input_0[25] = 0x77a38fc1;
      input_0[26] = 0x367fd2cd;
      input_0[27] = 0x68dec6e4;
      input_0[28] = 0x59a44a36;
      input_0[29] = 0x9f89f6df;
      input_0[30] = 0xed71061c;
      input_0[31] = 0x059f06c8;



    nputfsl(input_0[i],0);
  //  microblaze_nbwrite_datafsl(input_0[i],0);
        xil_printf("0x%08x \n\r ",input_0[i]);
  //    XUartLite_SendByte(XPAR_RS232_BASEADDR,input_0[i]);

  };

      for(i = 0; i < 3; i++)
    {
              ngetfsl(temp[i],0);
        //    microblaze_nbread_datafsl(temp[i],0);
              //xil_printf("i = %d\r\n", i);
              //microblaze_nbread_cntlfsl(temp,0);
              xil_printf("Read: 0x%08x \n\r", temp[i]);

     //   XUartLite_SendByte(XPAR_RS232_BASEADDR,temp[i]);


              //    for (i=0;i<3;i++){
              //    microblaze_nbread_datafsl(output_0[i],0);
              //xil_printf("%08x \n\r",output_0[i]);
  };
while(1);
//return 1;
}
```