République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université IBN KHALDOUN de Tiaret

MEMOIRE DE MAGISTER

Spécialité : Informatique
Option : Informatique **R**épartie et **M**obile **(IRM)**

Présenté par :
Mr. CHETOUANE Mohamed

Thème :

« Crypto système ECC embarqué sur circuit FPGA ».

Devant le jury d'examen :

			Président
Mme. Benbouzid si Tayeb Fatima	Maître de Conférences A	ESI	Encadreur
Mr. ANANE Mohamed	Maître de Conférences B	ESI	Co-encadreur
			Examinateur
			Examinateur

Soutenance le : XX/XX/2016.

ملخص:
في هذا العمل، نقوم بتحسين, تصميم و تنفيذ نظام التشفير بالمفتاح العمومي ECC محمل علي دارة منطقية البرمجة (FPGA) ل Xilinx. تعتبر أحجام المفاتيح المستخدمة 192 بت.

الضرب القياسي وضرب الوحدات هما العمليتان الرئيسيتان للتشفير و فك التشفير في نظام الECC. تم استخدام احداثيات جاكوبيان من أجل تفادي عملية القسمة العددية و هي جد مكلفة. كما تم استخدام اثنان من مضاعفات مونتغمري بالتوازي و الذي يعطي نتائج أفضل من حيث الوقت المستغرق للتنفيذ و المساحة المشغولة على دارة FPGA .كما استعمل ضاغط 4:2 للمضاعف للقضاء على انتشار العدد المتحفظ في عملية الضرب.
تم التصميم المادي لعملية الضرب القياسي و تنفيذه على دارة FPGA بواسطة ISE 12.2 خاصة ب Xilinx.

الكلمات المفتاحية : نظام التشفير, ECC , FPGA , الضرب القياسي , ضاغط 4:2.

Résumé

Dans ce travail, nous présentons l'optimisation, la conception et l'implémentation du crypto système à clé publique ECC embarqué sur un circuit FPGA de Xilinx. Les tailles des clés considérées sont de 192 bits.

Les multiplications scalaire et modulaire sont les opérations principales du chiffrement/déchiffrement ECC. L'utilisation des coordonnées Jacobiennes pour l'additionnement et doublement élimine l'inversion modulaire qui est très couteuse et donne un meilleur coût en termes de complexité calculatoire. Nous avons utilisé deux multiplieurs de Montgomery en parallèle, ce qui a donné un bon compromis entre le temps d'exécution et la surface occupé. Pour optimiser le multiplieur en termes d'architecture, nous avons utilisé un compresseur 4 : 2 pour éliminer la propagation des retenues dans le multiplieur.

Nous avons réalisé la conception matérielle de la multiplication scalaire et son implémentation sur circuit FPGA avec l'outil ISE 12.2 de Xilinx.

Mots-clés: Crypto système, ECC, FPGA, Multiplication scalaire, Compresseur 4 : 2.

Abstract

In this work, we present an optimization, design and implementation of public key crypto system ECC embedded on FPGA circuit of Xilinx. The key sizes considered are of 192 bits.

The scalar and modular multiplications are the main operations of encryption/decryption in ECC. The use of Jacobian coordinates for addition and doubling eliminates the modular inversion which is a very expensive operation and provides a lower cost in terms of execution time and occupied area. To optimize the multiplier in terms of architecture a compressor 4: 2 was used to eliminate carries propagation in the multiplier.

We have designed hardware architecture for the scalar multiplication and presented its implementation on FPGA with the ISE 12.2 tool of Xilinx.

Keywords: Crypto system, ECC, FPGA, Scalar multiplication, compressor 4 : 2.

Remerciements

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

Nous tenons aussi à exprimer nos remerciements les plus chaleureux à notre directeurs de thèse Mme Benbouzid si Tayeb Fatima et Mr. Anane Mohamed, pour leur précieuse aides, pour leur suivi rigoureux et leur disponibilté pendant notre travail.

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

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

Table des matières

CHAPITRE III : La multiplication scalaire

CHAPITRE IV : Architecture hardware de la multiplication scalaire

CHAPITRE V : Résultats de simulation et d'implémentation

Introduction générale

L'évolution des besoins de l'humanité poussé la rechercher scientifique et technologiques à trouver des nouveaux solutions où des solutions très amélioré dans tous les domaines, l'informatique et l'électronique représentent des domaines très dynamiques riches en nouvelles découvertes et applications. Ces domaines se caractérisent par la tendance à la miniaturisation et l'automatisation. Des systèmes compacts, légers, économiques, ergonomiques et fiables.

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

En revanche, la cryptanalyse est l'étude des procédé cryptographique, dans le but de trouver des faiblesses, en particulier de décrypter des messages chiffrés sans connaitre la clé de déchiffrement. Pour que la cryptographie résiste à la cryptanalyse, des crypto systèmes sont composés par diverses fonctions complexes qui assurent non seulement la sécurité des données mais aussi leur intégrité, authenticité,….

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

Aujourd'hui les circuits FPGA (Field Programmable Gate Array) sont des solutions pour remplacer les ASIC (Application Specific Integrated Circuit) et les processeurs personnalisées, l'obstacle devant la technologie ASIC c'est le manque de souplesse, et l'impossibilité de faire des changements après la fabrication. Une erreur de conception ou un changement dans un modèle de communication standard nécessitera une nouvelle conception du système. La souplesse cherchée par les concepteurs des ASICs ne peut exister qu'en utilisant les circuits programmables.

Réduire les coûts, augmenter les performances et la fiabilité d'un circuit, diminuer les délais de mise en place et réduire l'encombrement des composants sur circuits, grâce aux cellules SRAM, il est facile de reconfigurer un FPGA autant de fois pour implanter les fonctionnalités désirées.

Dans ce travail, on s'intéresse au cryptosystème à clé publique l'ECC (Elliptic Crypthography Curve). L'intérêt de l'ECC, des clés bien plus petites que RSA (une clé ECC de 160 bits est aussi robuste qu'une clé RSA de 1024 bits), celui-là étant donc plus adapté à des environnements à puissance réduite (tels que les cartes à puce).

Le cryptosystème ECC est basé sur l'opération de la multiplication scalaire, qui une opération complexe basé sur une série d'additionnement et de doublement des points sur la courbe elliptique. Ces deux opérations à son rôle basé sur les opérations modulaires de base (multiplication modulaire, addition/soustraction modulaire et l'inversion modulaire)

L'objectif de ce travail est de proposer une architecture hardware optimiser en termes de complexité calculatoire et en terme de temps d'exécution pour effectuer la multiplication scalaire, puis on implémente cette opération sur un circuit FPGA afin d'atteindre une meilleure performance.

Nous avons organisé notre mémoire en cinq chapitres répartis comme suit : le premier chapitre est consacré aux généralités sur la cryptographie et les limites de crypto système RSA. Dans le deuxième chapitre, l'utilisation des courbes elliptiques dans la cryptographie et le fonctionnement d'un crypto système ECC. Le troisième chapitre présente le choix des coordonnées des points pour réduire la complexité calculatoire par conséquent réduire le nombre d'opérations. Le quatrième chapitre est réservé à l'architecture hardware des différents modules utilisés pour l'implémentation de la multiplication scalaire sur le circuit FPGA et la possibilité de parallélisé les opérations pour minimiser le temps d'exécution. Le dernier chapitre est consacré à la présentation des résultats de simulation et l'implémentation de l'opération de la multiplication scalaire.

CHAPITRE I

Généralités sur la cryptographie

1.1- Introduction

La cryptographie moderne a été développée pour atteindre certains buts tels que: l'authentification, la confidentialité, l'intégrité des données, la non répudiation, le contrôle d'accès et la gestion des clés. La cryptographie doit satisfaire ces principales fonctions de sécurité.

La cryptographie est essentiellement basée sur les mathématiques en utilisant des clés, et se partage en deux types : la cryptographie symétrique qui utilise une même clé pour le chiffrement/déchiffrement et la cryptographie asymétrique qui utilise une paire de clés : une clé publique pour le chiffrement et une clé privée pour le déchiffrement.

Dans ce premier chapitre, nous donnons des notions sur la cryptographie et ses deux types, puis nous abordons les principaux algorithmes cryptographiques asymétriques en présentant le RSA avec ses avantages et inconvénients. Enfin, nous terminons par une conclusion.

1.2- Terminologie sur la cryptographie

- **Chiffrement** : c'est la méthode ou l'algorithme utilisé pour transformer un texte en clair en un texte chiffré.

- **Déchiffrement** : c'est la méthode ou l'algorithme utilisé pour transformer un texte chiffré en un texte en clair.

- **Clé** : c'est le secret utilisé pour chiffrer un texte ou déchiffrer le texte chiffré.

- **Cryptosystème**: c'est l'ensemble des clés possibles (espace de clés), des textes clairs et chiffrés possibles associés à un algorithme donné. La figure 1.1 représente le schéma d'un cryptosystème.

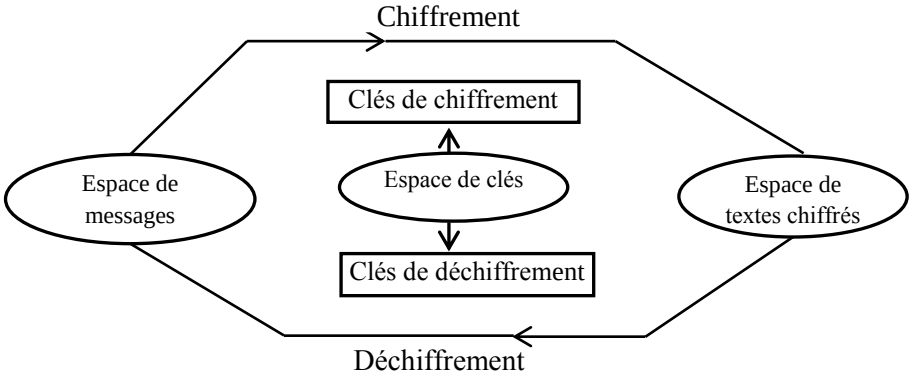


Figure 1.1 : Schéma d'un crypto système.

Un cryptosystème est en réalité un triplet d'algorithmes :

- L'un générant les clés
- L'autre pour chiffrer les messages
- Le troisième pour déchiffrer les messages.

1.3- La cryptographie

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

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

Selon la notion de clé pour le chiffrement et déchiffrement, on peut distinguer deux types de cryptographie : la cryptographie symétrique et la cryptographie asymétrique.

1.3.1-La cryptographie symétrique

Aussi appelé chiffrement à clé privée ou chiffrement à clé secrète, elle consiste à utiliser la même clé pour le chiffrement/déchiffrement qui doit rester secrète. Son principal avantage est la rapidité du calcul car elle utilise des clés de petites tailles et se base sur des opérations simples : additions et décalages. Mais elle présente deux inconvénients majeurs qui sont l'échange de la clé secrète via un canal sécurisé non disponible et la gestion des clés où pour que *n* personnes puissent communiquer, il faut gérer *n(n-1)/2* clés (une pour chaque paire de personnes).

La figure 1.2 montre le principe du chiffrement symétrique.

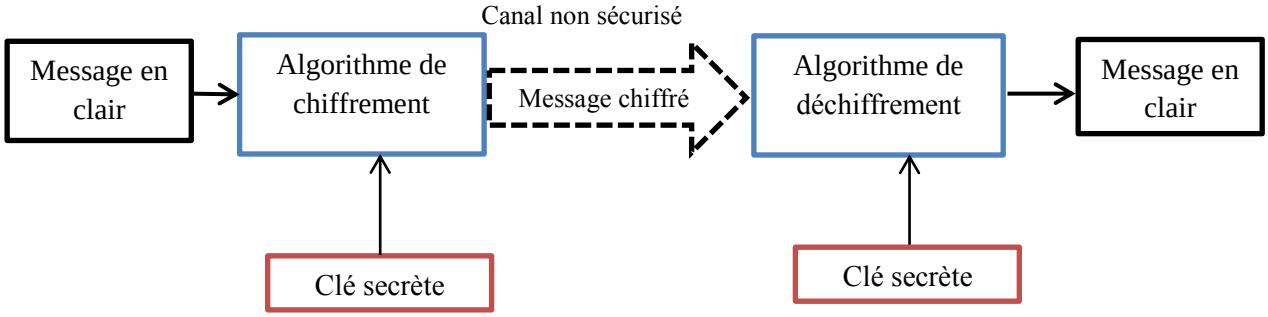


Figure 1.2 : Principe du chiffrement symétrique.

Il existe deux types d'algorithmes de chiffrement symétrique. Le premier type, qui est le plus utilisé, est le chiffrement par blocs, où les données sont traitées par bloc de bits et les standard les plus utilisés sont le DES (Data Encryption Standard) et l'AES (Advanced Encryption Standard). Le second type est le chiffrement à flot. Celui-ci permet après une période d'initialisation de chiffrer bit par bit [1, 2].

1.3.2-La cryptographie asymétrique

La cryptographie asymétrique ou à clé publique est couramment utilisée pour désigner une méthode de chiffrement d'un message en utilisant une *clé publique* pour obtenir un message chiffré qui sera transféré via un canal non sécurisé. Ce message chiffré sera déchiffré en utilisant une *clé privée* (ou secrète) pour retrouver le message d'origine comme le montre la figure 1.3.

Dans ce type de cryptographie, les problèmes du transfert de la clé secrète et de la gestion des clés posés par la cryptographie à clé secrète sont réglés.

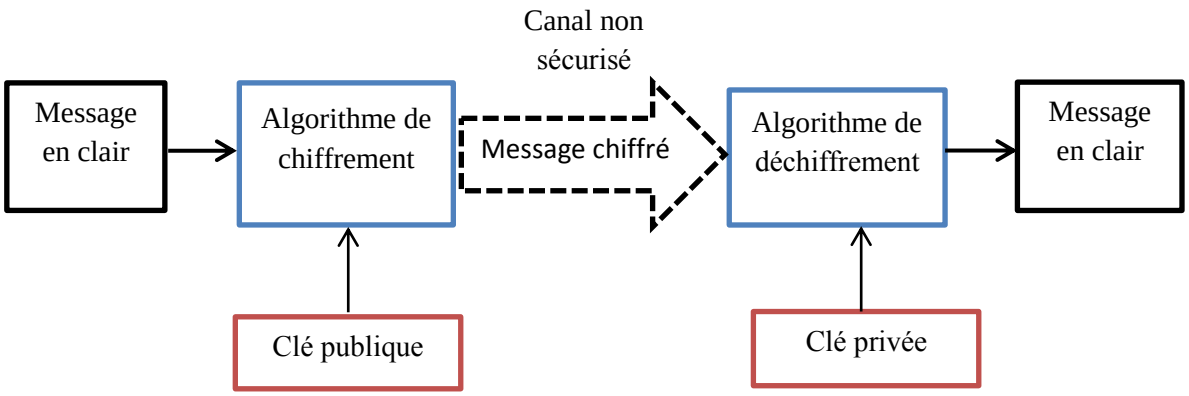


Figure 1.3 : Principe du chiffrement asymétrique.

1.4- La sécurité des cryptosystèmes à clé publique

La sécurité d'un cryptosystème à clé publique est dite calculatoire car il n'est pas possible de retrouver l'information secrète à partir des informations transmises publiquement sans faire un quelconque calcul d'une complexité exagérée.

Cette sécurité est basée sur des problèmes mathématiques considérés comme difficiles à résoudre tels que :

- La factorisation de grands nombres entiers : exemple du cryptosystème RSA ;

- Le calcul du logarithme discret dans le groupe multiplicatif d'un corps fini : le protocole d'échange de clés de Diffie-Hellman et le cryptosystème El Gamal ;
- Le calcul du logarithme discret dans le groupe additif des points d'une courbe elliptique définie sur un corps fini : ECC.

1.5- Les principaux algorithmes à clés publiques

1.5.1-Le protocole d'échange de clés de Diffie-Hellman

Ce n'est pas un système à clé publique, il s'agit d'un protocole qui permet à deux communicants qui partagent seulement de l'information publique d'établir une clé commune [3]. La sécurité se fonde sur la difficulté de calculer des logarithmes discrets.

Principe du protocole

Soit **A** et **B**, les deux entités qui veulent s'échanger une clé **K** :

- **A** génère un nombre premier $\boldsymbol{p}$ et un générateur **a** ($1 < \mathbf{a} < \boldsymbol{p}$ et $\boldsymbol{p}$ et **a** sont publics).
- **A** génère un nombre aléatoire : $\boldsymbol{X_A}$ ($< \boldsymbol{p}$), et **B** génère un autre nombre aléatoire : $\boldsymbol{X_B}$ ($< \boldsymbol{p}$) (gardés secrets).
- **A** calcule $\boldsymbol{y_A} = \mathbf{a}^{X_A}[\mathbf{mod}\ \boldsymbol{p}]$, et **B** calcule $\boldsymbol{y_B} = \mathbf{a}^{X_B}[\mathbf{mod}\ \boldsymbol{p}]$ ($\mathbf{y_A}$ et $\mathbf{y_B}$ sont publics).
- **A** calcule $\mathbf{K_{AB}} = \mathbf{y_B^{X_A}}\ \mathbf{[mod}\ \boldsymbol{p}\mathbf{]}$, et **B** calcule $\mathbf{K_{BA}} = \mathbf{y_A^{X_B}}\ \mathbf{[mod}\ \boldsymbol{p}\mathbf{]}$
- **A** et **B** partagent ainsi la même clé $\mathbf{K_{AB}} = \mathbf{K_{BA}}$.

Si quelqu'un a espionné : $\boldsymbol{p}$, $\boldsymbol{a}$, $\boldsymbol{y_A}$ et $\boldsymbol{y_B}$, pour obtenir **K**, il doit pouvoir calculer $\mathbf{x_A}$. Autrement dit, il doit pouvoir résoudre l'équation : $\boldsymbol{X_A} = \mathbf{log}_{\mathbf{a}}^{y_A}\mathbf{[mod\ p]}$

On appelle ceci résoudre le problème du logarithme discret. Quand les valeurs de $\boldsymbol{p}$, $\boldsymbol{a}$ et $\mathbf{y_A}$ sont très grandes, il s'agit d'un problème très difficile.

1.5.2-L ’algorithme d’El Gamal

C’est un système à clé publique basé sur le problème de Diffie-Hellman. Il fut inventé par Taher El Gamal [3]**.** Il est basé sur la difficulté de calculer des logarithmes discrets. Le problème du logarithme discret consiste à retrouver un entier λ tel que : $\mathbf{h} = \boldsymbol{g}^{\boldsymbol{\lambda}}\ \mathbf{mod}\ \boldsymbol{p}$**.**

Principe de l'algorithme

L'algorithme est présenté en trois étapes, soient **X** et **Y**, les deux parties qui veulent communiquer :

1- Génération de la clé

- **X** génère un grand nombre premier ***p*** et un générateur ***g*** tel que : $2 \leq$ **g** $\leq$ ***p***-1.
- **X** choisit de façon aléatoire un exposant **a** $\in \{0,1,2,\dots,P-1\}$ et on calcule : **A = g^{a} mod *p*, la clé publique** est (**A**), **la clé sécrète** est l'exposant **a** pour l'entité **X**.

De même pour **Y**, on choisit un exposant aléatoire **b**$\in \{0,1,2,\dots,P-1\}$ et on calcule : **B = g^{b} mod *p***, l**a clé publique** est (**B**), **La clé sécrète** est l'exposant **b** pour l'entité **Y**.

2- Chiffrement

Si l'entité **Y** veut chiffrer un message **m**, on calcule **c = A^{b}m mod *p*.**

Donc le code chiffré comprend :

- La clé publique de **Y** : (**B = g^{b} mod *p***).
- Le message chiffré (**c = A^{b} m mod *p***).

3- Déchiffrement

A la réception de (**B**, **c**), on divise **c** par **B^{a} mod *p*** comme suit : **m = c/ B^{a} mod *p*.**

1.5.3- L'algorithme R.S.A

RSA (Rivest-Shamir-Adleman), inventé en 1978 [4], est le cryptosystème le plus utilisé de nos jours. Sa sécurité dépend de la difficulté à trouver la factorisation du produit de deux grands nombres premiers.

Principe de l'algorithme

L'algorithme est en trois étapes résumées comme suit :

1- Génération des clés

- On génère deux grands nombres premiers ***P*** et ***Q***.
- On calcule ***N***. Il doit s'agir d'une valeur assez élevée, c'est le produit de ***P*** et ***Q***. La taille de ***N*** peut varier : 512 bits, 768, 1024, 2048…
- On choisit un très grand entier ***E***, relativement premier avec $\phi(n) =$[(***P***-1) $*$ (***Q***-1)]. La clé publique sera formée par (***E***, ***N***).

- On choisit ensuite un $\boldsymbol{D}$ tel que : $\boldsymbol{E * D \equiv 1\ mod\ [\phi(n)]}$, la clé privée sera donnée par ($\boldsymbol{D}$, $\boldsymbol{N}$).
- On jette $\boldsymbol{P}$ et $\boldsymbol{Q}$, il faut les détruire pour éviter les fuites.

2- **Chiffrement :** Pour chiffrer un message **M**, on calcule : $\mathbf{C = M^E\ mod\ N}$.

3- **Déchiffrement :** Pour déchiffrer un message chiffré **C**, on calcule : $\mathbf{M = C^D\ mod\ N}$.

Le principe de fonctionnement du protocole RSA est représenté sur la figure 1.4.

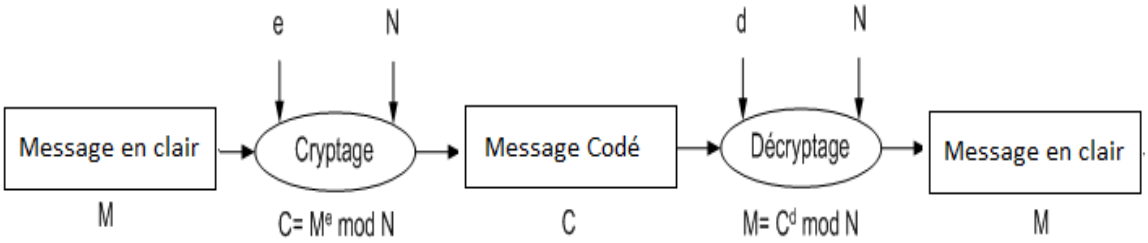


Figure 1.4 : Schéma bloc du principe de fonctionnement du RSA.

1.5.4-Les limites du R.S.A

Le RSA est le premier algorithme à clé publique publié; il est le plus cryptanalyse depuis les années 80. Dès qu'il est cassé, on augmente la taille de sa clé (taille du modulo N). Actuellement, les experts recommandent d'utiliser des nombres N de 1024 bits (309 chiffres décimaux) pour l'usage commercial est de 2048 bits (617 chiffres décimaux) pour une garantie se prolongeant sur une longue période de temps. Cette longueur tend à grandir dans le futur en même temps que la cryptanalyse et le pouvoir de calcul des machines progressent.

L'augmentation de la taille du modulo N induit beaucoup de calculs qui sont l'élévation d'un gros nombre à une grosse puissance modulo N. C'est-à-dire calculer $M^{\boldsymbol{E}}$ mod $\boldsymbol{N}$ connue sous : *l'exponentiation modulaire* qui est une suite de multiplications modulaires et est utilisée simultanément dans le chiffrement et le déchiffrement.

Cela pose des problèmes aux concepteurs de circuits sécurisés en particulier les systèmes embarqués qui ont des ressources limitées en terme de taille mémoire, d'énergie et de puissance de calcul.

Ces limites ont incité à chercher un nouveau cryptosystème basé sur les courbes elliptiques (ECC) dont les clés sont de petites tailles comparées aux clés de RSA avec une sécurité égale.

1.6- Conclusion

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

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

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

Dans le prochain chapitre, nous présentons en détail l'utilisation des courbes elliptiques dans la cryptographie, le cryptosystème ECC et leurs avantages par rapport au RSA.

CHAPITRE II

Les courbes elliptiques

2.1- Introduction

L'utilisation de la cryptographie basée sur les courbes elliptiques définies sur un corps fini a été suggérée en 1985 par deux chercheurs Miller [5] et Koblitz [6]. L'intérêt de l'ECC reposant sur le problème du logarithme discret qui est très difficile à résoudre et ECC requiert, à niveau de sécurité équivalent, des clés bien plus petites que le RSA (une clé ECC de 160 bits est aussi robuste qu'une clé RSA de 1024 bits), celui-là étant donc plus adapté à des environnements à puissance réduite (tels que les cartes à puce).

Dans ce chapitre, nous donnons des notions sur les groupes et les corps ainsi que quelques propriétés importantes des courbes elliptiques notamment l'équation algébrique à laquelle obéissent ces dernières. Nous détaillons aussi les paramètres d'une courbe elliptique, les avantages offerts par ECC par rapport au RSA et la justification du choix des paramètres en termes de sécurité et performance. Enfin, nous montrons comment utiliser les courbes elliptiques pour faire le chiffrement et le déchiffrement.

2.2- Notion de groupe

2.2.1- La loi du groupe

Soit (G, *) un ensemble muni d'une loi de composition interne *, on dit que G est un ***groupe*** si les trois conditions suivantes sont vérifiées :

1. G est associative, c'est à dire : $\forall f, g, h \in G, f * (g * h) = (f * g) * h$
2. G admet un élément neutre, c'est-à-dire : $\exists e \in G$ tel que $\forall g \in G, g * e = e * g = g$
3. Tout élément g de G admet un élément symétrique pour la loi *, c'est-à-dire : $\forall g \in G$, $\exists g' \in G$, tel que $g * g' = g' * g = e$.
4. Si de plus la loi * est commutative c'est-à-dire : $\forall g, h \in G, g * h = h * g$, on dit alors que (G, *) est un ***groupe commutatif*** (ou abélien).

2.2.2- L'ordre d'un groupe

L'*ordre* d'un groupe est le nombre de ses éléments. On le note $|G|$. Un groupe est dit fini si le nombre de ses éléments est fini. Si le groupe n'est pas fini, il est dit infini.

2.2.3- L'ordre d'un élément g du groupe G

L'*ordre* d'un élément $g \in G$ est le plus petit entier $k > 1$, s'il existe, tel que $g^k = e$ avec e est l'élément neutre de G. sinon on dit que l'ordre de g est infini, on le note *ord(g)*, l'*ordre* de g égal aussi à l'*ordre* du sous-groupe de G qu'il engendre.

2.2.4- Le sous-groupe ⟨g⟩ engendré par g

Si (G, *) est un groupe fini et $g \in G$, on note $\langle g \rangle$ le sous-groupe engendré par g, où $\langle g \rangle = \{g^{-2}, g^{-1}, 1, g, g^2, g^3, \dots\}$.

2.2.5- Un groupe cyclique

Si G est un groupe d'*ordre n*, avec $g \in G$ est aussi d'*ordre n* on dit que *G* est un *groupe cyclique* et *g* est un *générateur* de *G* on note $G =< g >$.

Exemple

Soit $\mathbf{F_{11}^*} = (\mathbb{Z}/11\mathbb{Z})^* = \{1, 2, \dots, 10\}$ le groupe multiplicatif de restes modulo 11, Dans $\mathbf{F_{11}^*}$nous avons $2^0 = 1, 2^1 = 2, 2^2 = 4, 2^3 = 8, 2^4 = 5, 2^5 = 10, 2^6 = 9, 2^7 = 7, 2^8 = 3, 2^9 = 6,$ mais $2^{10} = 1$ et on commence à répéter les éléments, donc ord(2)= 10 et il génère tout le groupe. $< 2 > = \{1, 2, \dots, 10\}$.

2.3- Notion de corps

2.3.1- La loi de corps

Un corps est muni de deux lois l'addition (+) et la multiplication (*), satisfaisant les propriétés suivantes:

1. (𝔽, +) est un groupe abélien (avec l'addition) et l'élément neutre est 0.
2. (𝔽 \ {0}, *) est un groupe abélien (avec multiplication) et l'élément neutre est 1.
3. La loi (*) est distributive par rapport à (+) tel que : x * (y + z) = x * y + x * z pour tout x, y, z ∈ 𝔽.

2.3.2- L'ordre d'un corps

L'*ordre* d'un corps fini est le nombre des éléments de ce corps, si l'ensemble 𝔽 est fini, alors le corps est dit fini.

Un corps fini est de la forme $\mathbb{F}_q$ tel que q = p^m où p est un nombre premier on le dit de *caractéristique p* de 𝔽 et m est un entier positif.

2.3.3- Le corps premier $\mathbb{F}_p$

Si m = 1 alors $\mathbb{F}_p$ est un *corps premier* de caractéristique *p* [8].

Un corps premier $\mathbb{F}_p$ contient *p* éléments: $\{0, 1, 2, \dots, p-1\}$, pour chaque *p* premier il existe un seul corps premier $\mathbb{F}_p$.

2.3.4- Les opérations sur un corps premier $\mathbb{F}_p$

Les opérations sur le corps premier $\mathbb{F}_p$ sont définies comme suit :

- **Addition(+)**

Si (a, b) ∈ $\mathbb{F}_p$ alors a + b = r avec r ∈ [0, p -1] et r est le reste de division (a + b) sur p on appelle cet opération *addition modulo p* et on écrit a + b ≡ r (mod *p*), l'élément neutre de l'addition est 0.

- **Multiplication(*)**

Si (a, b) ∈ $\mathbb{F}_p$ alors a * b = s avec s ∈ [0, p -1] et s est le reste de division (a * b) sur p on appelle cette opération *multiplication modulo p* et on écrit a * b ≡ s (mod *p*), l'élément neutre de la multiplication est 1.

- **Soustraction**(-)

La soustraction est définie comme étant une addition pour (a, b) ∈ $\mathbb{F}_p$, a – b = a + (-b) où [b + (-b) = 0 (mod p)].

- **Division(/)**

La division est définie comme étant une multiplication pour (a, b) ∈ $\mathbb{F}_p$, a/b = a * b^{-1} où [b * b^{-1} = 1(mod *p*)].

2.4- Les courbes elliptiques sur $\mathbb{R}$

2.4.1- Définition d'une courbe elliptique

D'une manière générale, sur $\mathbb{R}$, une courbe elliptique E est considérée comme l'ensemble des couples (x, y) tels que : $y^2 = x^3 + ax + b$, où (a, b) ∈ $\mathbb{R}$ et où le déterminant de E noté Δ est non nul, avec $\Delta = -16\,(4a^3 + 27b^2)$; la condition $\Delta \neq 0$ assure qu'il n'existe pas de points sur la courbe admettant deux tangentes ou plus.

Pour dessiner une courbe, pour a et b fixés, on calcule y tel que : $y = \sqrt{x^3 + ax + b}$

Par exemple, pour a = -3 et b = 0.6, on a la courbe donnée par l'équation : $y^2 = x^3 - 3x + 0.6$, comme le montre la figure 2.1.

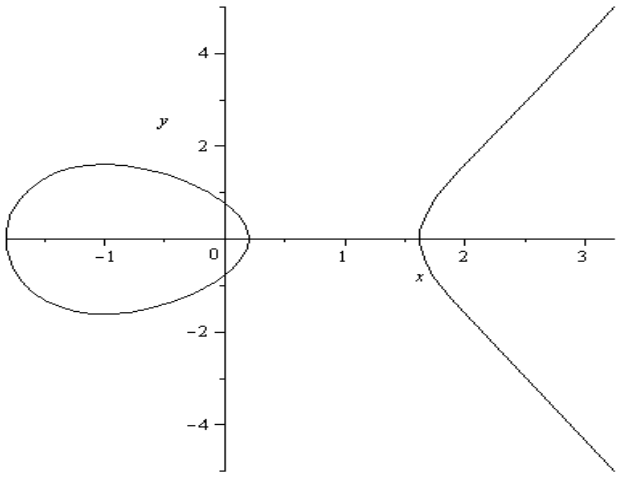


Figure 2.1 : Schéma de la courbe elliptique $\mathbf{y^2 = x^3 - 3x + 0.6}$

2.4.2- L'addition de deux points P et Q

Si P (x_P, y_P) et Q (x_P, y_P) sont des points distincts tels que P ≠ - Q, on détermine R = P + Q = (x_R, y_R) comme suit :

$$\begin{cases} x_R = (\lambda^2 - x_p - x_Q) \\ y_R = (\lambda(x_p - x_R) - y_P) \end{cases}, \text{ Avec } \lambda = \left(\frac{y_Q - y_P}{X_Q - X_P}\right)$$

2.4.3-Le doublement d'un point P

Si $y_P \neq 0$, on détermine 2P = R = (x_R, y_R)

$$\begin{cases} x_R = (\lambda^2 - 2x_p) \\ y_R = (\lambda(x_p - x_R) - y_P) \end{cases}, \textbf{ Avec } \lambda = \left(\frac{3X_P^2 + a}{2y_P}\right)$$

2.5- Les courbes elliptiques sur des corps finis $\mathbb{F}_p$

Les applications cryptographiques utilisent les courbes elliptiques sur les corps finis $\mathbb{F}_p$ au lieu de l'ensemble ℝ parce qu'on a besoin des calculs rapides et exacts.

2.5.1- Une courbe elliptique *E* définie sur $\mathbb{F}_p$

Une courbe elliptique E sur un corps fini $\mathbb{F}_p$, comprend tous les points (x, y) qui satisfont l'équation de la courbe elliptique modulo *p* : $y^2 \bmod p = (x^3 + ax + b) \bmod p$, où $(a, b) \in \mathbb{F}_p$. Si $(4a^3 + 27b^2) \bmod p \neq 0$, alors la courbe elliptique peut être utilisée pour former un *groupe elliptique*.

Un groupe de courbe elliptique définie sur $\mathbb{F}_p$ est constitué des points de la courbe elliptique correspondante, avec un point spécial O appelé point à l'infini, l'ensemble des points d'une courbe elliptique E définie sur $\mathbb{F}_p$ est noté E($\mathbb{F}_p$).

On peut vérifier que (E, +) est bien un g*roupe*. Notamment, il peut être montré que :

1. L'addition (+) est associatif : (**P1** + **P2**) + **P3** = **P1** + (**P2** + **P3**),
2. Il existe un point à l'infini **O** ∈ E avec **P1** + **O** = **O** + **P1** = **P1**,
3. Pour tout élément **P1** ∈ E, **−P1** ∈ E tel que **P1** + **−P1** = **−P1** + **P1** = **O**,

La démonstration mathématique de ces différents points est détaillée notamment dans le rapport technique de Joye [7].

Exemple

Le tracé de la courbe : $y^2 = x^3 + x + 1$ avec $a = 1$ et $b = 1$ sur le corps F_{23}, est montré sur la figure 2.2.

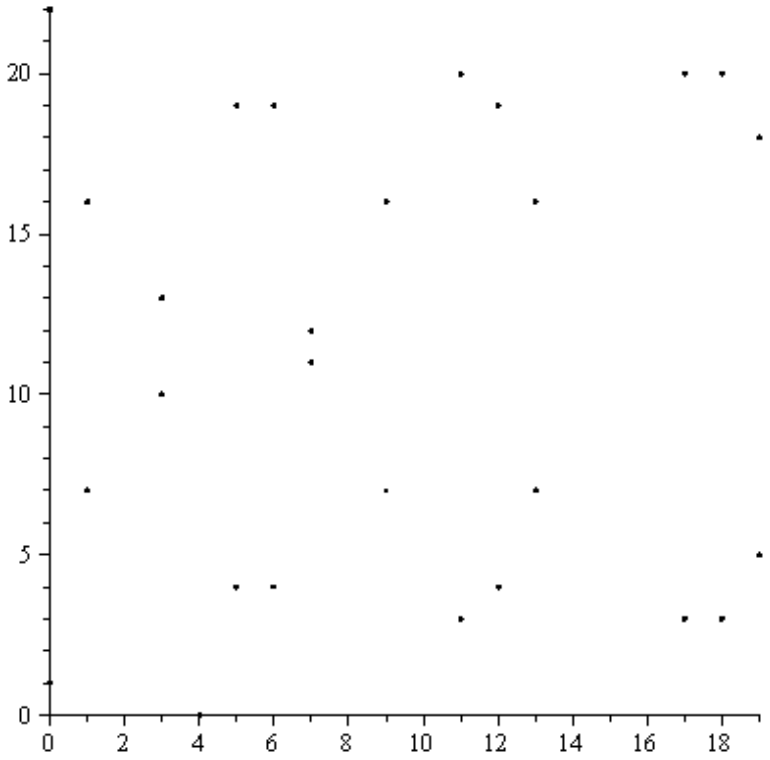


Figure 2.2 : Schéma de la courbe elliptique $\mathbf{y^2 = x^3 + x + 1}$ sur $\mathbf{F_{23}}$.

2.5.2- Les opérations sur les courbes elliptiques sur un corps fini $\mathbb{F}_p$

2.5.2.1- Addition de deux points P et Q sur $\mathbb{F}_p$

Les formules de calcul de l'addition et doublement sont les mêmes que dans, $\mathbb{R}$ mais en modulo p.

On note que l'opposé d'un point P (x_P, y_P) est P (x_P, $-y_P$), si P et Q sont des points distincts tels que P ≠ - Q, on détermine R = P + Q = (x_R, y_R) comme suit :

$$\begin{cases} x_R = (\lambda^2 - x_p - x_Q) \bmod p \\ y_R = (\lambda(x_p - x_R) - y_P) \bmod p \end{cases}, \text{ avec } \lambda = \left(\frac{y_Q - y_P}{X_Q - X_P}\right) \bmod p$$

2.5.2.2- Doublement d'un point P sur $\mathbb{F}_p$

Si $y_P \bmod p \neq 0$, on détermine 2P = R = (x_R, y_R)

$$\begin{cases} x_R = (\lambda^2 - 2x_p) \bmod p \\ y_R = (\lambda(x_p - x_R) - y_P) \bmod p \end{cases}, \text{ Avec } \lambda = \left(\frac{3X_P^2 + a}{2y_P}\right) \bmod p$$

2.5.3- Nombre de points d'une courbe elliptique définie sur un corps fini $\mathbb{F}_p$

Le nombre de points d'une courbe elliptique définie sur un corps fini (noté #E($\mathbb{F}_p$)), appelé aussi ordre du groupe E sur $\mathbb{F}_p$ est donné par le théorème de *Hasse* comme suit:

$$p + 1 - 2\sqrt{p} \leq \#E(\mathbb{F}_p) \leq p + 1 - 2\sqrt{p} \text{ [8].}$$

De ce théorème, on a #E($\mathbb{F}_p$) = $p + 1 - t$ avec $|t| \leq 2\sqrt{p}$, on appelle t la trace du Frobenius.

Ce théorème ne fournit qu'un encadrement du nombre de points de la courbe. Or en cryptographie, compter le nombre des points de la courbe elliptique est une étape très importante pour choisir des courbes sûres.

2.5.4- Utilisation des courbes elliptiques pour la cryptographie

Soit E une courbe elliptique définie sur un corps fini $\mathbb{F}_p$ et un point P ∈ E($\mathbb{F}_p$) avec un ordre *n*, et un autre point Q ∈ E($\mathbb{F}_p$), trouver $k \in [0, n-1]$ tel que Q = k P.

On dit que k est le *logarithme discret* de Q dans la base P et on le note $k = \log_p Q$. Pour utiliser les courbes elliptiques en cryptographie, il faut trouver un problème difficile à résoudre.

Considérons l'équation **Q** = ***k*** **P**, il est facile de calculer Q connaissant ***k*** et P, mais, il est difficile de déterminer ***k*** si on connait Q et P. La sécurité de ce crypto-système repose sur le problème du logarithme discret pour les courbes elliptiques (*Elliptic Curve Discrete Logarithm Problem, ECDLP*).

2.6- Les avantages des ECC

2.6.1- La taille de clés

Le problème du crypto-système RSA est que le niveau de sécurité croit avec la taille de la clé. La longueur de cette clé pose donc un problème au niveau des calculs, de la vérification des clés, de l'authentification et du stockage des données.

Le tableau 2.1 montre la taille de clés recommandée par le NIST pour l'ECC vers RSA [9].

Clés (bits)		Taux ECC/RSA
ECC	RSA	
192	1024	1 : 05
224	2048	1 : 09
256	3072	1 : 12
384	7680	1 : 20
521	15360	1 : 29

Tableau 2.1 : Taille des clés des ECC et RSA.

2.6.2- La performance

Pour vérifier la performance des algorithmes basés sur les ECC, quelques tests ont été faits pour le chiffrement d'un fichier texte de 1 Ko en les comparants aux résultats de l'algorithme de chiffrement asymétrique RSA pour le même niveau de sécurité. Les résultats sont représentés sur la figure 2.3.

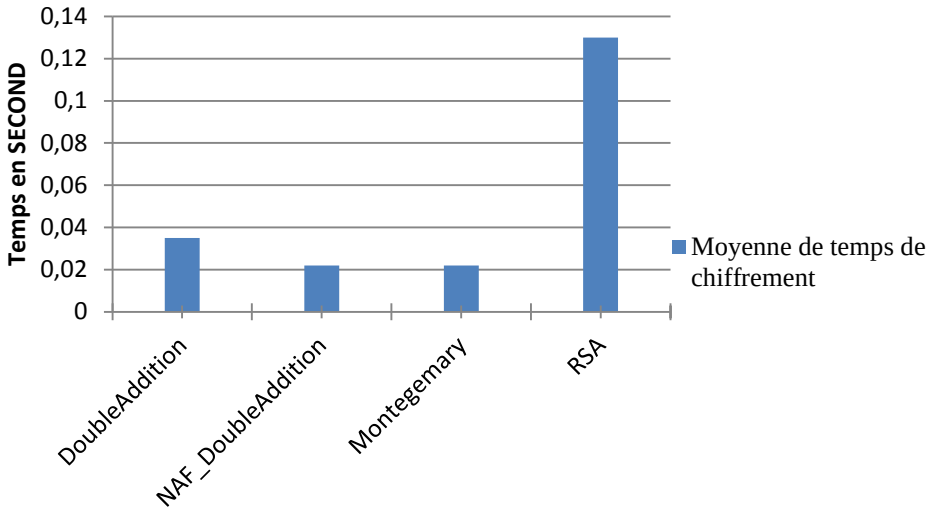


Figure 2.3 : Temps de chiffrement de différents algorithmes.

Ce schéma montre une grande différence par rapport au RSA où les algorithmes utilisés pour le chiffrement sur une courbe elliptique sont très optimisés de l'ordre de quelques millisecondes par rapport à l'algorithme RSA [10].

2.7- Le choix des paramètres de la courbe elliptique

Pour réaliser un cryptosystème ECC, il faut d'abord fixer les paramètres de la courbe. Ces paramètres doivent assurer la sécurité du cryptosystème contre les attaques mathématiques en premier lieu et avoir une arithmétique très efficace.

2.7.1- Les paramètres d'une courbe elliptique *E*

Les paramètres d'une courbe elliptique E définie sur un corps premier $\mathbb{F}_p$ sont sextuple *T*, tels que *T = (p, a, b, G, n, h)* avec :

- *p* est l'ordre de corps $\mathbb{F}_p$.
- *a* et *b* sont deux coefficients appartenant à $\mathbb{F}_p$, permettant de définir l'équation de la courbe elliptique sur le corps $\mathbb{F}_p$: $y^2 = x^3 + ax + b$.
- $G = (x_G, y_G)$ est un point générateur appartenant à $E(\mathbb{F}_p)$ avec x_G, y_G appartenant à $\mathbb{F}_p$.
- *n* est l'ordre du point *G*.
- *h* est le cofacteur = $\#E(\mathbb{F}_p)/n$.

2.7.2- Choix et la génération des paramètres en termes d'efficacité

Une bonne stratégie pour générer les paramètres d'une courbe elliptique est de les construire aléatoirement et de s'assurer qu'elles vérifient les contraintes de sécurité et d'efficacité.

Les corps finis $\mathbb{F}_p$ sont les corps les plus populaires pour les cryptosystèmes ECC. Un corps $\mathbb{F}_p$ est un corps premier de caractéristique *p*, qui contient *p* éléments: $\{0, 1, 2, \dots, p-1\}$, pour chaque *p* premier, il existe un seul corps premier $\mathbb{F}_p$[8].

La première étape consiste à donner une taille de *p* qui détermine la sécurité d'un cryptosystème ECC qui varie entre 160 et 512 bits [11].

2.7.2.1- Le choix de *p*

Lorsqu'on utilise $\mathbb{F}_p$ dans des systèmes à courbes elliptiques, il existe différents choix de *p*:

2.7.2.1.1- Les nombres premiers généraux

Ces nombres premiers peuvent être générés aléatoirement [11] et la mise en œuvre peut utiliser la réduction de Montgomery ou la réduction de Barrett, cela permet une implémentation efficace en utilisant d'autres systèmes cryptographiques tels que le RSA.

2.7.2.1.2-Les nombres premiers de Mersenne

Un nombre de Mersenne est un entier *p* de la forme $p = 2^n - 1$ avec $n \in \mathrm{N}^*$.

L'intérêt des nombres de Mersenne vient de leur forme si particulière qui offre une équivalence intéressante pour la réduction : $2^n \equiv 1 \pmod p$, par exemple pour n = 5, on a *p* = 31, $230 \bmod 31 = (7*2^5 + 6) \bmod 31 = (7 + 6) \bmod 31 = 13$.

Grâce à cette propriété, la réduction est énormément simplifiée.

Le problème avec les nombres de Mersenne est sa faible densité, il existe au maximum un seul nombre de Mersenne de n bits ainsi les protocoles ECC utilisent des nombres premiers de tailles supérieures à 100 bits et aucun premier de Mersenne ne se situe dans l'intervalle $[2^{160}, 2^{256}]$. Cet intervalle qui est d'une importance pour la cryptographie à base de courbes elliptiques conduit à proposer des généralisations sur les nombres premiers de Mersenne.

2.7.2.1.2.1-Pseudo-Mersennes

Crandall [12] propose des nombres Pseudo Mersenne de la forme $p = 2^n - c$ avec $n \in \mathrm{N}^*$ et $c < 2^{n/2}$, on peut utiliser la relation $2^n \equiv c \pmod p$ pour la réduction.

Les nombres Pseudo Mersenne évitent le principal défaut des nombres de Mersenne qui ne sont pas assez nombreux pour pouvoir fournir un nombre de Mersenne premier pour tout n.

Il faut noter que Crandall a déposé un brevet en 1992 concernant cette classe de nombres, ce qui peut limiter l'utilisation du pseudo Mersennes.

2.7.2.1.2.2-Les nombres de Mersenne Généralisés

Dans un autre sens, Solinas [13] introduit le concept des nombres premiers de Mersenne généralisées. Un nombre de Mersenne Généralisé est un entier p de la forme $p = P(2^k)$ où P est un polynôme unitaire de degré n tel que ses coefficients P_i appartiennent à $\{-1, 0, 1\}$ et que son second terme du plus haut degré soit de degré inférieur ou égal à n/2.

Plusieurs nombres de Mersenne Généralisés sont conseillés par le NIST [8] pour l'implantation des protocoles sur les courbes elliptiques.

Nous détaillons l'utilisation de p dont la taille est 192 bits. Pour cette taille, on a $p = 2^{192} - 2^{64} - 1$, ce dernier peut être défini par le polynôme $P(t) = t^3 - t - 1$ évalué par $t = 2^{64}$ en considérant les entiers modulo p de taille $3k = 192$ bit ou $k = 64$.

L'entier à réduire $S < p^2$ de taille $6k = 384$ bit écrits sous la forme :

$S = \sum_{j=0}^{5} s_j 2^{j64} = s_5 2^{320} + s_4 2^{256} + s_3 2^{192} + s_2 2^{128} + s_1 2^{64} + s_0$, où chaque s_j est de 64 bits. On peut écrire $S = (s_5 \parallel s_4 \parallel s_3 \parallel s_2 \parallel s_1 \parallel s_0)$.

$r = S \bmod (2^{192} - 2^{64} - 1)$ est obtenue en additionnant les quatre entiers de 192 bit :

$(s_5 \parallel s_5 \parallel s_5)_{2^{64}}$, $(s_4 \parallel s_4 \parallel 0)_{2^{64}}$, $(0 \parallel s_3 \parallel s_3)_{2^{64}}$, et $(s_2 \parallel s_1 \parallel s_0)_{2^{64}}$ et en effectuant des soustractions successives de p jusqu'à ce que r est soit inférieur à p [13] comme le montre l'algorithme 2.1.

Algorithme 2.1 : réduction modulaire sur $p = 2^{192} - 2^{64} - 1$

Entrée : S, p avec $0 \leq S < p^2$
Sortie : r tel que $r = S \bmod p$
Début
S = s(2^{64})
A := $(s_2 s_1 s_0)_{2^{64}}$
B := $(0\ s_3 s_3)_{2^{64}}$
C := $(s_4 s_4 0)_{2^{64}}$
D := $(s_5 s_5 s_5)_{2^{64}}$
R := A + B + C + D
r := R mod p
Fin

L'approche consiste à choisir un nombre de Mersenne généralisé pour accélérer la phase de réduction modulaire efficacement, mais elle conduit toutefois à un manque de flexibilité. En effet, lorsqu'on implante matériellement une méthode de réduction relative à un modulo ayant la forme d'un Mersenne, cette architecture n'est plus valable si ce modulo change.

2.7.2.2- Le choix de *a*

Les courbes standardisées utilisent des courbes elliptiques avec $a = -3$, ce choix est justifié par la réduction du nombre de multiplications en coordonnées Jacobéennes (*J*) (X, Y, Z) et en coordonnées Jacobéennes modifiées(J^c) (X, Y, Z, aZ^4) d'une multiplication modulaire.

Pour l'opération de doublement et dans l'expression $M = 3X_1^2 + aZ_1^4$ on a trois multiplications modulaires, si $a = -3$ alors $M = 3X_1^2 - 3Z_1^4 = 3(X_1 - Z_1^2)(X_1 + Z_1^2)$ qui réduit le nombre de multiplications modulaires à deux[15,16].

2.7.2.3- La génération de b

L'algorithme qui génère le coefficient *b* a comme entrées le module *p* et en sortie le coefficient *b* à générer.

Le coefficient *b* doit satisfaire la condition $b^2 r \equiv a^3 \pmod{p}$ où la variable *r* est différente de 0 et (-27/4) cette condition pour éviter les courbes super singulières [14], si $a = -3$ alors le coefficient *b* satisfait :

$$b^2 r \equiv -27 \pmod{p}.$$

2.7.2.4- Choix du point générateur G

Tout point dans la courbe d'ordre *n* est un point générateur, donc le choix de *G* reste libre [15].

2.7.3- Les contraintes de sécurité contre les attaques mathématiques

Les paramètres de la courbe elliptique *E* doivent être choisis afin de résister à toutes les attaques connues sur l'ECDLP. Si G est un groupe et *P* est un point générateur de ce groupe, *Q* est un autre élément de G, nous voulons déterminer l'entier $k = \log_P Q$ tel que Q = *kP* et $k \in [0, n-1]$ où *n* est l'ordre de *P*, on cite les attaques mathématiques sur ECDLP.

2.7.3.1- Attaque Pohlig-Hellman

L'attaque Pohlig-Hellman est réussie si l'ordre du groupe $\#E(\mathbb{F}_p)$ est connu. Donc $\#E(\mathbb{F}_p)$ peut être factorisé en de nombres premiers utilisant une méthode de factorisation; c'est à dire divise $\#E(\mathbb{F}_p)$ en petits nombres premiers.

Dans ce cas, l'ECDLP peut être résolu en résolvant l'ECDLP en petits groupes d'ordre premier en parallèle, puis en utilisant la technique des restes chinois pour résoudre le système obtenu et pour éviter cette attaque, il faut que *n* soit un nombre *premier* et *h* = 1 , 2 , 3 , ou 4 où $\#E(\mathbb{F}_p) = hn$.

2.7.3.2- Attaque Pollard's rho

Appelée attaque « Pollard's ρ », nécessite de l'ordre de $\sqrt{n}$ opérations. L'idée de cette méthode est de trouver deux paires *(c, d)* et *(c', d')* modulo *n* tel que : $cP + dQ = c'P + d'Q$, après on a $(c - c')P = (d - d')Q$, donc $Q = \frac{c - c'}{d - d'}P$, et enfin $k \equiv \frac{c - c'}{d - d'} (mod\ n)$.

La méthode consiste à choisir au hasard $c, d \in [0, n-1]$, après on calcule $cP + dQ$, et on stocke le triplet $(c, d, cP + dQ)$ dans un tableau, puis générer au hasard $c, d \in [0, n-1]$ et comparer le résultat avec les autres résultats jusqu'à obtenir une coïncidence tel que $cP + dQ = c'P + d'Q$ et $(c, d) \neq (c', d')$. Afin d'éviter cette attaque au minimum on devrait avoir $n > 2^{160}$.

2.7.3.3- Attaque sur les courbes anormales

Les courbes elliptiques avec $\#E(\mathbb{F}_p) = p$ sont appelées des courbes anormales. Cette attaque est un succès avec ce genre de courbe par le théorème de Hasse où nous avons $\#E(\mathbb{F}_p) = p + 1 - t$ avec t désignant la trace de E sur $\mathbb{F}_p$. Ainsi $\#E(\mathbb{F}_p) = p$ est équivalent à $t = 1$, donc le problème du logarithme discret sur E peut être résolu en un temps linéaire (attaque de Smart). L'idée de l'attaque consiste à soulever les points *Q* et *P* aux points $\tilde{Q}$ et $\tilde{P}$ sur la courbe elliptique $\tilde{E}/\mathbb{Q}_p$, où $\mathbb{Q}_p$ désigne le corps des nombres p-adiques puis résoudre ECDLP. Il faut que $\#E(\mathbb{F}_p) \neq p$ pour que *E* ne soit pas anormal.

2.7.3.4- Attaque de Menezes - Okamoto - Vanstone (MOV)

Si *v* est le plus petit entier tel que $n \mid (p^v - 1)$, alors grâce au couplage de Weil, on peut ramener le problème du logarithme discret sur la courbe elliptique ECDLP à un problème de logarithme discret sur le corps fini $\mathbb{F}p^v$, c'est l'attaque de MOV. Pour éviter cette attaque-il faut que *n* ne divise pas $(p^k - 1)$ pour $1 \leq k \leq 20$.

2.8- Le fonctionnement du crypto-système ECC

Soit l'entité **A** qui doit envoyer un message **m** chiffré à l'entité **B** qui doit à son tour déchiffrer le message.

D'abord, on fixe les paramètres d'une la courbe elliptique *E* à utiliser, où ses paramètres $T = (p, a, b, G, n)$ vérifient les contraintes du paragraphe 2.7.3.

Le fonctionnement se fait en trois étapes [17] :

3.2.1- La génération des clés

1- L'entité **B** génère un entier **d** inférieur à *n* qui sera sa **clé privée** et puis calcule sa **clé publique** $P_B = \mathbf{d} \cdot G$.

2- L'entité **B** envoie les paramètres de la courbe *E* et sa clé publique (P_B) à l'entité **A**, comme le montre la figure 2.3.

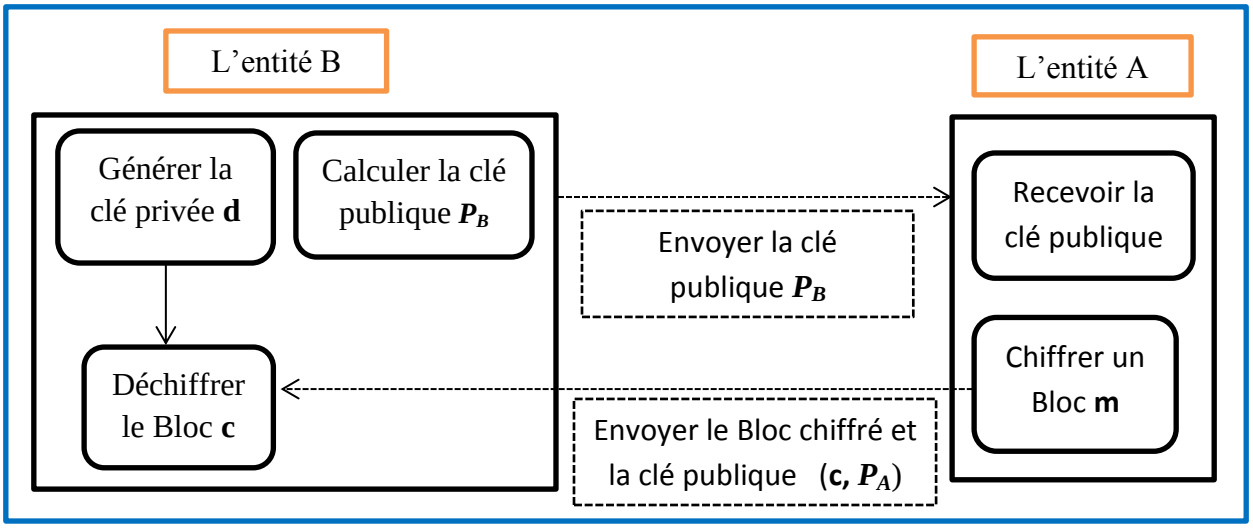


Figure 2.3 : Génération et échange des clés dans ECC.

3.2.2- Le chiffrement

1- L'entité **A** détermine aléatoirement un nombre entier positif **k** inférieur à *n* qui sera **sa clé privée**, puis calcule sa **clé publique** $P_A = \mathbf{k} \cdot G$.

2- L'entité **A** calcule la clé secrète entre **A** et **B** : $P_s(x_s, y_s) = \mathbf{k} \cdot P_B$.

3- Pour chiffrer le message **m,** on calcule $\mathbf{c} = (m \times x_s) \bmod p$.

4- L'entité **A** envoie [P_A, **c**] à l'entité **B**, comme le montre la figure 2.4.

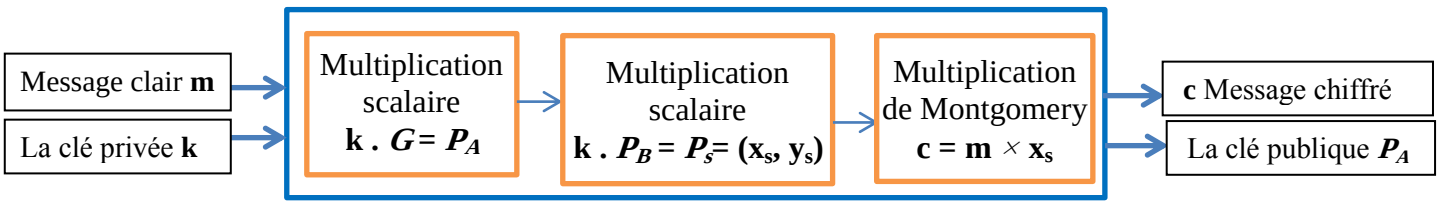


Figure 2.4 : Chiffrement d'un message m avec ECC.

3.2.3- Le déchiffrement

L'entité **B** reçoit $[P_A, c]$ de l'entité **A** et procède comme suit pour déchiffrer le message **m** :

1- L'entité **B** calcule la clé secrète entre **A** et **B** : $P_s(x_s, y_s) = \mathbf{d} \cdot P_A$.

2- L'entité **B** calcule la multiplicative inverse de $x_s = x_s^{-1}$.

3- L'entité **B** calcule $m = c \times x_s^{-1} \bmod p$, comme le montre la figure 2.5.

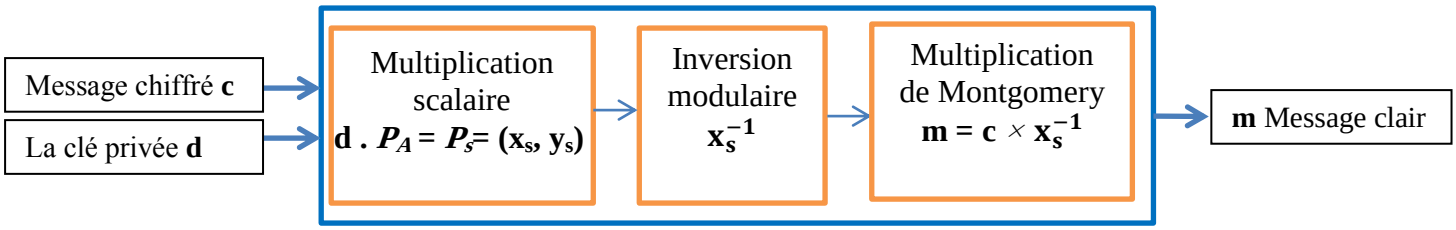


Figure 2.5 : Déchiffrement d'un message chiffré c avec ECC.

2.8- Conclusion

Les cryptosystèmes ECC sont plus adaptés pour des systèmes qui ont des ressources limitées telles que la taille de mémoire et une faible puissance de calcul car ces systèmes utilisent une clé de petite taille par rapport au RSA.

La sécurité du cryptosystème ECC est basée sur l'ECDLP, qui est un problème mathématique difficile à résoudre si les paramètres de la courbe sont bien choisis.

L'opération principale à effectuer afin de concevoir un cryptosystème basé sur les courbes elliptiques est la multiplication scalaire.

Dans le prochain chapitre, nous présenterons en détails l'opération de la multiplication scalaire et ses opérations arithmétiques modulaires de base.

CHAPITRE III

La multiplication scalaire

3.1- Introduction

La principale opération à effectuer lors d'un protocole utilisant les courbes elliptiques est la multiplication d'un point par un scalaire. Étant donné un entier *k* et un point *P* sur une courbe *E*, l'opération consiste à calculer le point *[k]•P*. Ce calcul s'effectue à partir d'une chaîne de calculs faisant intervenir les opérations élémentaires sur la courbe (addition de points et doublement d'un point).

Ce chapitre est consacré à la multiplication scalaire et comprendra trois parties :

1- La première partie traitera des différents algorithmes réalisant cette multiplication en fonction de nombre d'additions et de doublement de points sur la courbe.
2- La deuxième partie concerne les différents systèmes de coordonnées représentant les points d'une courbe elliptique en axant sur leurs influences sur la réduction du nombre d'opérations modulaires nécessaires à cette multiplication.
3- Enfin, la dernière partie, comporte une comparaison des différents systèmes de coordonnées pour choisir le système de coordonnées le plus adéquat en termes de complexité calculatoire.

3.2- Définition de la multiplication scalaire

Si *E* est une courbe elliptique définie sur un corps fini $\mathbb{F}_p$, la multiplication entre un nombre entier *k* et un point $P \in E(\mathbb{F}_p)$ donne un nouveau point *Q*. Cette opération est appelée la multiplication scalaire et nécessite une série d'opérations : addition de points et doublement du point *P* pour obtenir *Q*, et chacune de ces opérations nécessite un ensemble d'opérations modulaires dans le corps fini $\mathbb{F}_p$ qui sont : l'addition, la soustraction, la multiplication et la division comme le montre la figure 3.1.

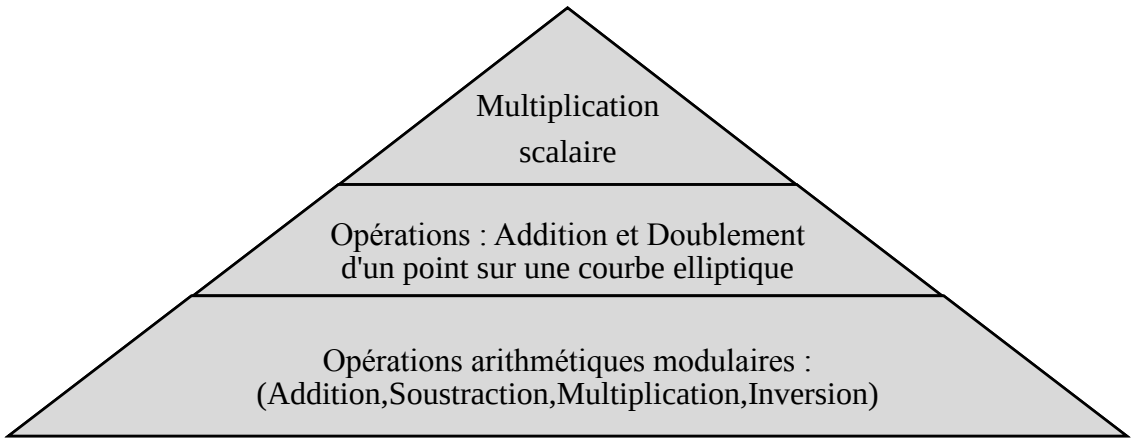


Figure 3.1 : Hiérarchie de la multiplication scalaire.

3. 3- Les algorithmes de la multiplication scalaire

Dans la littérature, il existe trois types d'algorithmes pour calculer la multiplication scalaire : les algorithmes binaires, les algorithmes de forme non adjacente et les algorithmes à fenêtres glissantes.

3.3.1- Les algorithmes binaires

Ces algorithmes sont basés sur la conversion du scalaire k en binaire et il existe deux types d'algorithmes :

3.3.1.1- Doublement-et-Addition de gauche à droite

Cette méthode calcule $[k]\bullet P$ en effectuant une série de doublements et d'additions et consiste à parcourir les bits de k en partant du poids fort de k : on effectue ainsi un doublement pour chaque bit de k, suivi d'une addition pour chaque bit non nul voir l'algorithme 3.1 [18].

Algorithme 3.1 : « Doublement-et-Addition de gauche à droite »

Entrées: $k = (k_{t-1}, \ldots, k_1, k_0)_2$, $P \in E(F_p)$.
Sorties: $Q = [k]\bullet P$.
1. $Q \leftarrow P$.
2. Pour i de $(t-2)$ jusqu'à 0 faire
   2.1 $Q \leftarrow 2Q$. [Doublement]
   2.2 Si $k_i = 1$ alors $Q \leftarrow Q + P$. [Addition]
3. Retourner (Q).

Si k est représenté sur t bits, l'algorithme « Doublement-et-Addition de gauche à droite » implique en moyenne (t-1) doublements et ((t-1)/2) additions.

Exemple

Dans cet exemple on présente le déroulement de l'algorithme précédent, on a choisi le scalaire k de tel sorte que le nombre de 1 et de 0 soient presque les mêmes :

- $k = (1720)_{10} = (11010111000)_2$,
- $t = 11$
- le nombre de doublement = 10 = *t-1*,
- le nombre d'Addition = 5 = *(t-1)/2*,

Nous remarquons pour cet algorithme que le nombre de doublement est égal à la taille de k en bits alors que le nombre d'additions équivaut au nombre de bits non nul.

3.3.1.2- Doublement-et-Addition de droite à gauche

Cet algorithme a le même principe que le précédent, mais les bits de k sont parcourus à partir du poids faible de k, voir l'algorithme 3.2 [18].

Algorithme 3.2 : « Doublement-et-Addition de droite à gauche »

Entrées: $k = (k_{t-1}, \ldots, k_1, k_0)_2$, $P \in E(F_p)$.
Sorties: $Q = k.P$.
1. $Q \leftarrow \infty$.
2. Pour i de 0 à $t-1$ faire
   2.1 Si $k_i = 1$ alors $Q \leftarrow Q + P$. [Addition]
   2.2 $P \leftarrow 2P$. [Doublement]
3. Retourner (Q).

Cet algorithme a la même complexité calculatoire que l'algorithme précédent, néanmoins, celui-ci a l'avantage d'exécuter les opérations d'addition et de doublement (lignes 2.1 et 2.2 de l'algorithme) en parallèle.

Exemple : On prend $k = (1720)_{10} = (11010111000)_2$, on a donc $t = 11$

- le nombre de doublement = 11 = t,
- le nombre d'addition = 6 ≈ $(t)/2$ = 11/2.

3.3.2- Les algorithmes de forme non adjacente (NAF)

Nous avons vu que le nombre d'additions à effectuer, dans les deux algorithmes précédents, dépend directement du nombre des k_i non nuls. Une façon de gagner en efficacité consisterait alors à rendre l'écriture binaire de k avec plus de k_i nuls où cette forme est appelée forme non-adjacente. Dans ce qui suit, nous présentons deux types d'algorithmes basés sur cette représentation.

3.3.2.1- Algorithme utilisant une forme non adjacente binaire (NAF_2)

Pour effectuer une multiplication scalaire, il faut d'abord convertir le scalaire k en forme non adjacente binaire (NAF_2). Cette forme utilise trois digits : {0, 1,-1}, ce qui réduit le nombre de 1 dans la représentation de k, néanmoins elle introduit un nouveau chiffre dans la représentation qui est le (-1) et qui sera représenté par $\bar{1}$. L'algorithme 3.3 montre comment calculer cette forme [19].

Algorithme 3.3 : « calcul de la forme NAF binaire »

Entrées **:** entier positif k.
Sorties **:** $NAF_2(k)$.
1. $i \leftarrow 0$.
2. Tant que $k \geq 1$ faire
   2.1 Si k est impair alors: $k_i \leftarrow 2 - (k \bmod 4)$, $k \leftarrow k - k_i$;

2.2 Sinon: $k_i \leftarrow 0$.
2.3 $k \leftarrow k/2$, $i \leftarrow i+1$.
3. Retourner $(k_{i-1}, k_{i-2}, \ldots, k_1, k_0)$.

L'algorithme 3.4 utilise la forme non-adjacente pour le calcul de la multiplication scalaire, le chiffre $\bar{1}$ de cette représentation correspond à une soustraction de point comme montré sur la ligne 3.3 de l'algorithme [19]. Cette dernière a le même coût qu'une addition parce-que l'opposé d'un point $P(x, y)$ est $\bar{P}$ $(x, -y)$.

Algorithme 3.4 : « Méthode NAF binaire pour la multiplication scalaire »

Entrées: entier positif k, $P \in E(F_p)$.
Sorties: $Q = [k] \bullet P$.
1. utiliser l'algorithme 3.3 calculer $NAF_2(k) = \sum_{i=0}^{l-1} d_i 2^i$.
2. $Q \leftarrow P$.
3. Pour i *de* $(l-2)$ jusqu'a 0 faire
3.1 $Q \leftarrow 2Q$. [Doublement]
3.2 Si $k_i = 1$ alors $Q \leftarrow Q + P$. [Addition]
3.3 Si $k_i = -1$ alors $Q \leftarrow Q - P$. [Soustraction]
4. Retourner(Q).

Le coût de la multiplication scalaire en utilisant cet algorithme est en moyenne de t doublements et $(t/3)$ Additions.

Exemple
Le déroulement de l'algorithme avec $k = (763)_{10} = (1011111011)_2$ donne :

- $NAF_2(k) = (10\bar{1}0000\bar{1}0\bar{1})$,
- $t = 11$,
- le nombre de doublement = 10 = t-1,
- le nombre d'addition et soustraction = 3 ≈ (t-1)/3.

3.3.2.2- Algorithme utilisant la fenêtre de forme non-adjacente *w*NAF

Un entier k est en forme wNAF écrit $k = \sum_{i=0}^{l-1} k_i 2^i$ où $|k_i| < 2^{w-1}$, où w est la taille de la fenêtre, l'algorithme 3.5 montre comment calculer la forme wNAF [20].

Algorithme 3.5 : « calcul de la forme *w*NAF »
Entrées **:** la taille de la fenêtre w, un entier positif k.
Sorties **:** $NAF_w(k)$.
1. $i \leftarrow 0$.
2. Tant que $k \geq 1$ faire
2.1 Si k est impair alors: $k_i \leftarrow k$ **mods** 2^w, $k \leftarrow k - k_i$;
2.2 Sinon: $k_i \leftarrow 0$.
2.3 $k \leftarrow k/2$, $i \leftarrow i+1$.

3. Retourner $(k_{i-1}, k_{i-2}, \ldots, k_1, k_0)$.

L'algorithme 3.6 présente le calcul de la fonction mods de l'algorithme 3.5.

Algorithme 3.6 : « calcul de la fonction mods »

Entrées: la taille de la fenêtre *w*, entier d
Sorties: d mods 2^w
1. Si **(**d mod 2^w) ≥ (2^w/2) Alors
2. Retourner (d mod 2^w) – 2^w
3. Sinon Retourner d mod 2^w
5. Fin si

Le calcul de la multiplication scalaire par la méthode wNAF est donné par l'algorithme 3.7 [20].

Algorithme 3.7 : « Méthode *w*NAF pour la multiplication scalaire »

Entrées: la taille de la fenêtre *w*, entier positif *k*, $P \in E(\mathrm{F}q)$.
Sorties: $Q = [k] \bullet P$.
1. Utiliser l'algorithme 3.5, calculer $\mathrm{NAF}_w(k) = \sum_{i=0}^{l-1} k_i 2^i$.
2. Calculer $P_i = i\,P$ pour $i \in \{1, 3, 5, \ldots, 2^{(w-1)} - 1\}$.
3. $Q \leftarrow \infty$.
4. Pour *i* de (l −1) jusqu'à 0 faire
   4.1 $Q \leftarrow 2Q$. [Doublement]
   4.2 Si $k_i \neq 0$ alors
   4.3 Si $k_i > 0$ alors $Q \leftarrow Q + P_{ki}$. [Addition]
   4.4 Sinon $Q \leftarrow Q - P_{-ki}$. [Soustraction]
5. Retourner(Q).

Le coût de la multiplication scalaire en utilisant *w*NAF est de *t* doublement et en moyenne [t/(w+1)] addition et soustraction parce que le nombre de bits non nuls dans la représentation *w*NAF est en moyenne [t/(w+1)][18], on remarque que le nombre d'additions/soustractions diminue lorsque la taille de la fenêtre augmente mais le nombre de pré-calculs augmente aussi, alors plus de mémoire est exigé pour stocker les pré-calculs ce qui rend la taille de la fenêtre dépendante de la mémoire disponible.

Exemple

Pour k =1720 et la taille de la fenêtre w = 4 on a donc :

- $P_i = \{P_1, P_3, P_5, P_7\}$,
- $\mathrm{NAF}_4(k)$ = $100\bar{3}0007000$,
- l = 12,
- le nombre de Doublement = 12,
- le nombre d'addition et soustraction = 3.

3.3.3- Les algorithmes à fenêtres glissantes

Ces algorithmes sont des améliorations des algorithmes binaires et NAF où au lieu de parcourir les bits de *k* un par un pour le calcul de l'addition et le doublement, *k* est considéré par bloc de bits selon la taille de la fenêtre choisie. Il existe deux types d'algorithmes :

3.3.3.1- Algorithmes à fenêtres glissantes pour la NAF

Cet algorithme utilise la forme non adjacente, son principe est de parcourir les digits de *k* en blocs de taille *w*. Si la valeur du bloc courant est impair, on effectue une addition sinon on réduit la taille du bloc (*w-1*) jusqu'à trouver une valeur impaire, et on effectue un doublement pour chaque bit de *k*, voir l'algorithme 3.8 [21].

Algorithme 3.8 : « Multiplication scalaire par fenêtres glissantes avec NAF »

Entrées: la taille de la fenêtre *w*, entier positif *k*, $P \in E(\mathrm{F}p)$.
Sorties: $Q = [k] \bullet P$.
1. Utilisation de l'algorithme 3.5, calculer $\mathrm{NAF}_w(k) = \sum_{i=0}^{l-1} k_i 2^i$.
2. Calcul de $P_i = iP$ pour $i \in \{1,3, \ldots, [2(2^w - (-1)^w) / 3 - 1]\}$.
3. $Q \leftarrow \infty$, $i \leftarrow (l - 1)$.
4. Tant que $i \geq 0$ faire
   4.1 Si $k_i = 0$ alors $t \leftarrow 1$, $u \leftarrow 0$;
   4.2 Sinon : rechercher le plus grand $t \leq w$ tel que $u \leftarrow (k_i, \ldots, k_{i-t+1})$ est impair.
   4.3 $Q \leftarrow 2^t Q$.
   4.4 Si $u > 0$ alors $Q \leftarrow Q + P_u$; Sinon si $u < 0$ alors $Q \leftarrow Q - P_{-u}$.
   4.5 $i \leftarrow (i - t)$.
5. Retourner (Q).

Le nombre de doublement pour cet algorithme est *l* qui est le même avec l'algorithme *w*NAF et le nombre d'additions/soustractions s'améliore par rapport *w*NAF qui est $l/(w + v(w))$ où $v(w) = 4/3 - [(-1)^w / (3 * 2^{w-2})]$ et $v(w)$ est la longueur moyenne d'une série de zéros entre les fenêtres[18], mais le nombre de pré-calcul P_i tel que $i \in \{1,3, \ldots, [2(2^w - (-1)^w)/3 - 1]\}$ augmente par rapport aux algorithmes *w*NAF [8] .

Exemple
Pour k =1720 et la taille de la fenêtre $w = 4$ on a donc :

- $P_i = \{P_1, P_3, P_5, P_7, P_9\}$,
- $\mathrm{NAF}_4(k) = 1000\bar{3}0007000$,
- $l = 12$,
- le nombre de doublement = 12,
- le nombre d'addition et soustraction = 3.

3.3.3.2- Algorithme utilisant la fenêtre glissante pour la forme binaire

L'algorithme 3.9 est un cas particulier de l'algorithme 3.8 où le même principe est utilisé mais avec une représentation binaire du scalaire *k*.

Algorithme 3.9 : « Méthode à fenêtres glissantes de forme binaire »

Entrées: la taille de la fenêtre *w*, entier positif $k = \sum_{i=0}^{l-1} k_i 2^i$, $P \in E(\mathrm{F}q)$.
Sorties: $Q = [k] \bullet P$.
1. Calculer $P_i = iP$ pour $i \in \{2, 3,5, \ldots, (2^w - 1)\}$.
2. $Q \leftarrow P$, $i \leftarrow (l - 2)$.
3. Tant que $i \geq 0$ faire
    3.1 Si $k_i = 0$ alors $Q \leftarrow 2Q$, $i \leftarrow (i - 1)$;
    3.2 Sinon: rechercher le plus petit entier t, tel que : $i - t + 1 \leq w$, et $k_t = 1$.
        3.2.1 Pour $j = 1$ à $(i - t + 1)$ do $Q \leftarrow 2Q$.
        3.2.2 $h_i \leftarrow (k_i\, k_{i-1} \ldots k_t)_2$.
        3.2.3 $Q \leftarrow Q + P_{h_i}$.
        3.2.4 $i \leftarrow (t - 1)$.
4. Retourner(Q).

Le nombre de doublement pour cet algorithme est toujours égal à *l* qui est le même que celui de l'algorithme à fenêtres glissantes avec la forme NAF. Le nombre d'additions dans le pire des cas est égal à $[l/w]$ et le coût du pré calcul évalué par un doublement est de $(2^w - 1)/2$ Additions [22].

Exemple
Pour $k = 1720 = (11010111000)_2$ et la taille de la fenêtre $w = 4$:

- $P_i = \{P_1, P_3, P_5, P_7, P_9, P_{11}, P_{13}, P_{15}\}$,
- $l = 11$,
- le nombre de doublements = 11,
- le nombre d'additions = 2.

3.4- Les systèmes de coordonnées

Dans ce qui a précédé, nous avons étudié la complexité calculatoire de la multiplication scalaire en termes de nombre d'additions et de doublements nécessaires. Dans ce qui suit, nous allons étudier la complexité calculatoire de ces deux opérations en termes d'opérations élémentaires telles que l'addition/soustraction modulaire et la multiplication modulaire. Comme l'addition et le doublement de points sont relatifs à des systèmes de coordonnées, nous allons alors exprimer cette complexité calculatoire pour chacun des systèmes pour en évaluer les performances et en choisir les meilleurs.

3.4.1- Les coordonnées Affines (*A*)

On dit qu'un point *P* de la courbe *E* est en coordonnées affines s'il est donné par un couple $P = (x, y)$ d'éléments de *E* vérifiant l'équation $y^2 = x^3 + ax + b$, définissant *E*.

3.4.1.1- Addition en coordonnées Affines

Soit $P(x_P, y_P)$ et $Q(x_Q, y_Q)$ deux points de *E* donnés par leurs coordonnées affines, les coordonnées de $R(x_R, y_R) = P + Q$ sont données par les formules suivantes :

$$\begin{cases} x_R = (\lambda^2 - x_p - x_Q) \bmod p \\ y_R = (\lambda(x_p - x_R) - y_P) \bmod p \end{cases}, \text{ Avec } \lambda = \left(\frac{y_Q - y_P}{X_Q - X_P}\right) \bmod p$$

Si on suppose que l'élévation au carrée est une multiplication, une opération d'additionnement coûte : six additions modulaires, une inversion modulaire et trois multiplications modulaires [23].

3.4.1.2- Doublement en coordonnées Affines

Soit $P(x_P, y_P)$ un point de *E* tel que : $y_P \bmod p \neq 0$, on détermine le doublement de point *P* par $2P = R(x_R, y_R)$ où :

$$\begin{cases} x_R = (\lambda^2 - 2x_p) \bmod p \\ y_R = (\lambda(x_p - x_R) - y_P) \bmod p \end{cases}, \text{ Avec } \lambda = \left(\frac{3x_P^2 + a}{2y_P}\right) \bmod p$$

Une opération de doublement en coordonnées Affines coûte : quatre additions modulaires, une inversion modulaires et trois multiplications modulaires [23].

3.4.2- Les coordonnées Projectives (*P*)

L'utilisation des coordonnées projectives permet d'éviter de calculer des inversions modulaires qui ont un coût très important en termes de multiplications modulaires. En coordonnées projectives, un point *P* de la courbe est représenté par un triplet (X, Y, Z) correspondant au point affine (X/Z; Y/Z). Le tableau 3.1 résume les propriétés de ce système de coordonnées ainsi que ceux que nous allons étudier par la suite.

P : Projective, *J* : Jacobienne, J^c : Jacobienne Chudnovesky, J^m : Jacobienne Modifiée

Coor	P en coordonnées	(x, y) =	Equation de la courbe	∞
P	(X, Y, Z)	(X/Z, Y/Z)	$Y^2Z = X^3 + aXZ^2 + bZ^3$	(0, 1, 0)
J	(X, Y, Z)	$(X/Z^2, Y/Z^3)$	$Y^2 = X^3 + aXZ^4 + bZ^6$	(1, 1, 0)
J^c	(X, Y, Z, Z^2, Z^3)	$(X/Z^2, Y/Z^3)$	$Y^2 = X^3 + aXZ^4 + bZ^6$	(1, 1, 0)
J^m	(X, Y, Z, aZ^4)	$(X/Z^2, Y/Z^3)$	$Y^2 = X^3 + aXZ^4 + bZ^6$	(1, 1, 0)

Tableau 3.1 : Propriétés des différents systèmes de coordonnées.

3.4.2.1- Addition en coordonnées Projectives

Soit P et Q deux points en coordonnées projectives tels que : $P(X_1; Y_1; Z_1)$, $Q(X_2; Y_2; Z_2)$. L'additionnement de ces deux points en coordonnées Projectives est donnée par un nouveau point R : $P + Q = R(X_3; Y_3; Z_3)$, où [23]:

$$X_3 = vA;$$
$$Y_3 = u(v^2X_1Z_2 - A) - v^3Y_1Z_2; \quad \text{Avec} \quad u = Y_2Z_1 - Y_1Z_2;$$
$$Z_3 = v^3Z_1Z_2; \qquad v = X_2Z_1 - X_1Z_2;$$
$$A = u^2Z_1Z_2 - v^3 - 2v^2X_1Z_2;$$

3.4.2.2- Doublement en coordonnées Projectives

Soit P et Q deux points en coordonnées projectives tels que : $P(X_1; Y_1; Z_1)$. Le doublement de ce point en coordonnées projectives est : $2P = R(X_3; Y_3; Z_3)$ où [23]:

$$X_3 = 2hs; \qquad w = aZ_1^2 + 3X_1^2;$$
$$Y_3 = w(4B - h) - 8Y_1^2s^2; \quad \text{Avec} \quad s = Y_1Z_1;$$
$$Z_3 = 8s^3; \qquad B = X_1Y_1s;$$
$$h = w^2 - 8B.$$

L'addition de deux points dans ces coordonnées nécessite quatorze multiplications modulaires alors qu'un doublement nécessite que quatorze multiplications modulaires.

3.4.3- Les coordonnées Jacobiennes (J)

En considérant qu'un point P de la courbe représenté par un triplet (X, Y, Z) en coordonnées Jacobiennes correspond au point représenté en coordonnées affines (X/Z^2, Y/Z^3) comme montré sur le tableau 3.1. Les coordonnées Jacobiennes standard (J) correspondent aux (X, Y, Z), les coordonnées Jacobiennes Chudnovsky(J^c) sont données par (X, Y, Z, Z^2, Z^3) et les coordonnées Jacobiennes modifiées(J^m) sont données par (X, Y, Z, aZ^4). Il n'y a donc plus d'inversions modulaires à calculer pour effectuer l'addition et le doublement. Il faut noter que dans tous les cas, l'opposé du point (X, Y, Z) s'écrit (X, −Y, Z).

3.4.3.1- Addition en coordonnées Jacobiennes

Soit P et Q deux points en coordonnées Jacobiennes tels que : $P(X_1; Y_1; Z_1)$, $Q(X_2; Y_2; Z_2)$, l'additionnement de ces deux points en coordonnées Jacobiennes est donnée par: $P + Q = R(X_3; Y_3; Z_3)$ où [24]:

$$X_3 = -H^3 - 2U_1H_2 + r^2; \qquad U_1 = X_1Z_2^2; \qquad S_2 = Y_2Z_1^3;$$
$$Y_3 = -S_1H^3 + r(U_1H^2 - X_3); \quad \textbf{Avec} \quad U_2 = X_2Z_1^2; \qquad H = U_2 - U_1;$$
$$Z_3 = Z_1Z_2H; \qquad S_1 = Y_1Z_2^3; \qquad r = S_2 - S_1;$$

3.4.3.2- Doublement en coordonnées Jacobiennes

Soit P est un point en coordonnées Jacobiennes tels que : $P(X_1; Y_1; Z_1)$. Le doublement de ce point en coordonnées Jacobiennes $2P = R\ (X_3; Y_3; Z_3)$ est montré par les formules ci-dessous où [24]:

$$X_3 = T;$$
$$Y_3 = -8Y_1^4 + M(S - T); \quad \text{Avec} \quad S = 4X_1Y_1^2\ ;$$
$$Z_3 = 2Y_1Z_1; \qquad M = 3X_1^2 + aZ_1^4;$$
$$T = -2S + M^{\ 2}.$$

L'addition de deux points pour les coordonnées Jacobiennes nécessitent seize multiplications modulaires alors que le doublement en coordonnées Jacobiennes nécessite dix multiplications modulaires. On remarque que si la valeur du coefficient de la courbe a = -3 le doublement nécessitera seulement huit multiplications modulaires alors que pour a = 0 le coût diminue à sept multiplications modulaires.

3.4.3.3- Les coordonnées Jacobiennes Chudnovsky

Ces coordonnées donnent une amélioration pour l'additionnement qui nécessite quatorze multiplications modulaires et onze multiplications modulaires pour le doublement. Quant aux formules de calcul ; ils sont présenté comme suit :

3.4.3.3.1- Addition en coordonnées Jacobiennes Chudnovsky

Soit $P(X_1; Y_1; Z_1; Z_1^2; Z_1^3\ )$ et $Q(X_2; Y_2; Z_2; Z_2^2; Z_2^3\ )$ deux points en coordonnées Chudnovsky, l'additionnement de ces deux point $P + Q = R(X_3; Y_3; Z_3; Z_3^2; Z_3^3)$ est donné par [24]:

$$X_3 = -H^3 - 2U_1H^2 + r^2;$$
$$Y_3 = -S_1H^3 + r(U_1H^2 - X_3);$$
$$Z_3 = Z_1Z_2H;$$
$$Z_3^2 = Z_3^2;$$
$$Z_3^3 = Z_3^3;$$

Avec

$$U_1 = X_1(Z_2^2\ ); \qquad S_2 = Y_2(Z_1^3\ );$$
$$U_2 = X_2(Z_1^2\ ); \qquad H = U_2 - U_1;$$
$$S_1 = Y_1(Z_2^3\ ); \qquad r = S_2 - S_1;$$

3.4.3.3.2- Doublement en coordonnées Jacobienne Chudnovsky

Soit $P(X_1; Y_1; Z_1; Z_1^2; Z_1^3)$ un point en coordonnées Jacobienne Chudnovsky, le doublement de ce point $2P = R(X_3; Y_3; Z_3; Z_3^2; Z_3^3\ )$ est donné par les formules suivantes [24]:

$X_3 = T;$

$Y_3 = -8Y_1^4 + M(S - T);$ $\quad S = 4X_1Y_1^2;$

$Z_3 = 2Y_1Z_1;$ Avec $\quad M = 3X_1^2 + a(Z_1^2)^2;$

$Z_3^2 = Z_3^2;$ $\quad T = -2S + M^2.$

$Z_3^3 = Z_3^3;$

3.4.3.4- Les coordonnées Jacobiennes Modifiées

Ces coordonnées donnent une amélioration pour le doublement qui nécessite huit multiplications modulaires et dix-neuf multiplications modulaires pour l'addition. Les formules de calcul comportent une modification des formules de coordonnées Jacobiennes standard.

3.4.3.4.1- Addition en coordonnées Jacobiennes modifiées

Si on a : $P = (X_1; Y_1; Z_1; aZ_1^4)$, $Q = (X_2; Y_2; Z_2; aZ_2^4)$ deux points en coordonnées Jacobiennes modifiées. L'additionnement de ces points donne un nouveau point R : $P + Q = R$ $(X_3; Y_3; Z_3; aZ_3^4)$ où les formules des différentes coordonnées du point R sont donnés par [24] :

$X_3 = -H^3 - 2U_1H^2 + r^2;$

$Y_3 = -S_1H^3 + r(U_1H^2 - X_3);$

$Z_3 = Z_1Z_2H;$ **Avec**

$aZ_3^4 = aZ_3^4;$

$U_1 = X_1(Z_2^2);$ $\quad S_2 = Y_2(Z_1^3);$

$U_2 = X_2(Z_1^2);$ $\quad H = U_2 - U_1;$

$S_1 = Y_1(Z_2^3);$ $\quad r = S_2 - S_1.$

3.4.3.4.1- Doublement en coordonnées Jacobienne modifié

Si on a : $P = (X_1; Y_1; Z_1; aZ_1^4)$ un point en coordonné Jacobienne modifié, le doublement de ce point $2P = R$ $(X_3; Y_3; Z_3; aZ_3^4)$ est donné par les formules suivantes [24] :

$X_3 = T;$ $\quad S = 4X_1Y_1^2;$

$Y_3 = M(S - T) - U;$ **Avec** $\quad U = 8Y_1^4;$

$Z_3 = 2Y_1Z_1;$ $\quad M = 3X_1^2 + (aZ_1^4);$

$aZ_3^4 = 2U(aZ_1^4);$ $\quad T = -2S + M^2.$

3.4.4- La conversion entre les systèmes de coordonnées

Un point en coordonnées affines peut être converti à n'importe quel système de coordonnées sans inversion où multiplication, alors que la conversion de projective entre

coordonnées a besoin de multiplications et d'inversions selon le cas. Le tableau 3.2 résume les conversions des coordonnées affines et projectives en d'autres coordonnées.

Affine (x, y) ⟶	Projective(x, y, 1)
Affine (x, y) ⟶	Jacobienne(x, y, 1)
Affine (x, y) ⟶	Jacobienne Chudnovsky(x, y, 1, 1, 1)
Affine (x, y) ⟶	Jacobienne Modifié (x, y, 1, a)
Projective(x, y, z) ⟶	Affine (xz^{-1}, yz^{-1})
Projective(x, y, z) ⟶	Jacobienne (xz, yz^2, z)
Projective(x, y, z) ⟶	C. Jacobienne (xz, yz^2, z, z^2, z^3)
Projective(x, y, z) ⟶	M. Jacobienne (xz, yz^2, z, az^4)

Tableau 3.2 : Conversions des coordonnées affine et projective en d'autres coordonnées [25].

3.5 – Réduction de la complexité calculatoire de la multiplication scalaire

A fin de réduire la complexité calculatoire, nous avons comparé les coûts des différentes systèmes de coordonnées; le tableau 3.3 résume le coût d'additionnement et doublement en fonction des multiplications et inversions modulaires pour les différentes systèmes de coordonnées. Dans cette comparaison on a supposé que le coût d'une élévation au carré est le même qu'une multiplication modulaire, et nous n'allons pas prendre en compte la complexité calculatoire des additions modulaires du fait que celle-ci est négligeable devant celle de la multiplication modulaire.

M : multiplication modulaire, I : inversion modulaire.

Système de coordonnées	**Additionnement**	**Doublement**
Affine	1I + 3M	1I + 4M
Projective	14M	12M
Jacobienne standard	16M	10M
Jacobienne Chudnovsky	14M	11M
Jacobienne Modifié	19M	8M

Tableau 3.3 : Coût du doublement et de l'addition pour chaque système de coordonnées [25].

Tous les coordonnées projectives et Jacobiennes ne nécessitent pas d'inversion modulaire pour effectuer l'additionnement ou le doublement, qui une opération très couteuse [26], d'où les coordonnées affines, qui utilise l'opération d'inversion, sont à éviter.

On remarque aussi dans le tableau 3.3 que les coordonnées Jacobiennes modifiées donnent un meilleur coût pour un doublement comparé à celui des coordonnées projectives. Les coordonnées Jacobienne Chudnovsky donnent un meilleur coût pour l'additionnement; cela implique l'utilisation des coordonnées mixtes : Jacobiennes modifiées pour le doublement et projectives où Jacobiennes Chudnovsky pour l'additionnement.

Concernant le choix entre les coordonnées projectives où Jacobiennes Chudnovsky pour l'additionnement en se basant sur le tableau 3.4, ce dernier résume le coût de conversion entre les différents systèmes de coordonnées en fonction de la multiplication modulaire et l'inversion modulaire.

M : Multiplication, I : Inversion, S : élévation carré.

De à	**Affine**	**Projective**	**Jacobienne**	**C. Jacobienne**	**M. Jacobienne**
Affine	-	-	-	-	-
Projective	2M + 1I	-	2M + 1S	3M + 1S	3M + 2S
Jacobienne	3M + 1S + 1I	2M + 1S	-	1M + 1S	1M + 2S
C. Jacobienne	3M + 1S + 1I	1M	-	-	1M + 1S
M. Jacobienne	3M + 1S + 1I	2M + 1S	-	1M + 1S	-

Tableau 3.4 : Coût de conversion entre les systèmes de coordonnées [25].

Si on choisit les coordonnées Jacobiennes modifiés pour le doublement, donc les coordonnées Jacobiennes Chudnovsky sont les plus adaptés où la conversion du J^m vers J^c et la conversion J^c vers J^m coûtent en total quatre multiplications modulaires, à l'opposé des coordonnées projectives où le coût total vaut cinq multiplications modulaires.

3.6- Conclusion

Dans ce chapitre, nous avons présenté trois types d'algorithmes pour effectuer la multiplication scalaire et nous avons évalué la complexité calculatoire de ces algorithmes en fonction du nombre d'additions et de doublements.

Ensuite, nous avons étudié les différents systèmes de coordonnées afin de choisir ceux présentant le minimum d'opérations pour l'exécution de l'additionnement et le doublement. Ces systèmes de coordonnées ont été évalués en fonction des multiplications modulaires.

Les coordonnées Jacobiennes et projectives permettent d'éviter l'inversion modulaire qui est très coûteuse en terme de complexité calculatoire et par conséquent le hardware consommé.

Et enfin nous avons conclu que les coordonnées mixtes (J^c pour l'additionnement *et* J^m pour le doublement) représentent les meilleurs choix en termes de réduction de la complexité calculatoire.

Dans le chapitre suivant, nous allons discuter les différentes architectures possibles pour l'implémentation de l'opération de chiffrement du protocole ECC à savoir la multiplication scalaire en se basant sur le parallélisme des opérations modulaires requises pour le doublement et l'addition de point.

CHAPITRE IV

Architecture de la multiplication scalaire

4.1- Introduction

Dans le chapitre précédent, nous avons conclu que les coordonnées mixtes (J^m) et (J^c) sont le meilleur choix en termes de complexité calculatoire. Alors, dans ce chapitre, nous allons axer sur l'architecture hardware de la multiplication scalaire en utilisant ces coordonnées.

En premier lieu, nous allons étudier la possibilité de paralléliser les opérations modulaires (multiplication et addition) pour le doublement en J^m et l'additionnement en J^c afin d'atteindre un temps d'exécution réduit. Ensuite, nous détaillerons l'architecture hardware des opérations modulaires : multiplication, inversion, addition et soustraction qui sont les opérations de base de la multiplication scalaire. Enfin nous proposons des architectures performantes pour l'additionnement et le doublement pour les combiner afin d'obtenir une architecture globale pour la multiplication scalaire.

4.2- Amélioration de la multiplication scalaire en termes de temps d'exécution

Pour améliorer le temps d'exécution de la multiplication scalaire, nous avons étudié la possibilité de paralléliser les opérations arithmétiques modulaires (multiplication, addition) de l'additionnement et du doublement en utilisant un seul multiplieur, deux et trois multiplieurs pour comparer les temps d'exécution et choisir le nombre de multiplieurs offrant le meilleur délai.

4.2.1- La structure du doublement en coordonnées J^m

Comme nous l'avons vu précédemment, le doublement d'un point $P(X_1; Y_1; Z_1; aZ_1^4)$ en coordonnées J^m donne un nouveau point $R(X_3; Y_3; Z_3; aZ_3^4)$ où les formules de ces coordonnées sont comme suit:

$X_3 = T;$

$Y_3 = M(S - T) - U;$ Avec $S = 4X_1Y_1^2$;

$Z_3 = 2Y_1Z_1;$ $U = 8Y_1^4;$

$aZ_3^4 = 2U(aZ_1^4);$ $M = 3X_1^2 + (aZ_1^4);$

Selon l'analyse des formules ci-dessus, nous avons extrait le séquencement des multiplications modulaires (V_1 à V_8) et les additions/soustractions modulaires (D_1 à D_{14}) d'un doublement comme montré ci-dessous :

1. $V_1 \leftarrow Y_1^2 = Y_1 * Y_1$ $D_1 \leftarrow 2X_1 = X_1 + X_1$
2. $V_2 \leftarrow Y_1^4 = V_1 * V_1$ $D_2 \leftarrow 4X_1 = D_1 + D_1$
3. $V_3 \leftarrow 4X_1Y_1^2 = D_2 * V1 =$ **S** $D_3 \leftarrow 2Y_1^4 = V_2 + V_2$
4. $V_4 \leftarrow X_1^2 = X_1 * X_1$ $D_4 \leftarrow 4Y_1^4 = D_3 + D_3$
5. $V_5 \leftarrow Y_1 * Z_1$ $D_5 \leftarrow 8Y_1^4 = D_4 + D_4 =$ **U**
6. $V_6 \leftarrow 8Y_1^4 aZ_1^4 = D_5 * aZ_1^4$ $D_6 \leftarrow 2X_1^2 = V_4 + V_4$

7.		$D_7 \leftarrow 3X_1^2 = D_6 + V_4$
8.		$D_8 \leftarrow D_7 + aZ_1^4 = 3X_1^2 + aZ_1^4 = \boldsymbol{M}$
9.	$V_7 \leftarrow (3X_1^2 + aZ_1^4)^2 = (D_8)^2$	$D_9 = \boldsymbol{Z_3} \leftarrow 2V_5 = 2\,Y_1Z_1$
10.		$D_{10} \leftarrow 2 * V_3 = 8\,X_1Y_1^2 = 2S$
11.		$D_{11} = \boldsymbol{X_3} \leftarrow V_7 - D_{10} = \boldsymbol{T}$
12.		$D_{12} \leftarrow V_3 - X_3$
13.	$V_8 \leftarrow (8Y_1^4 + aZ_1^4)(4X_1Y_1^2 - X_3) = D_{12} * D_8$	$D_{13} = aZ_3^4 \leftarrow 2V_6 = 16\,Y_1^4\,aZ_1^4$
14.		$D_{14} = \boldsymbol{Y_3} \leftarrow V_8 - D_5 = M(S - T) - U$

4.2.1.1- Doublement en J^m avec un seul multiplieur

La figure 4.1 présente le séquencement du doublement de point $P(X_1; Y_1; Z_1; aZ_1^4)$ en utilisant un seul multiplieur et un seul additionneur. Dans cette première figure, le chemin critique équivaut au délai de huit multiplications modulaires (MM) et six additions modulaires (AM), alors que les autres additions/soustractions modulaires n'entrent pas dans le chemin critique vu qu'elles sont couvertes par les MM dont le délai est bien plus important que celui de l'addition/soustraction modulaire.

4.2.1.2- Doublement en J^m avec deux multiplieurs

La figure 4.2 présente le séquencement d'un doublement du point $P(X_1; Y_1; Z_1; aZ_1^4)$ en utilisant deux multiplieurs et un seul additionneur. Dans cette deuxième structure, le chemin critique équivaut au délai de quatre multiplications et cinq additions modulaires. Avec l'utilisation de deux MM, le délai des additions modulaires a été réduit, vu que durant la multiplication plusieurs additions peuvent être exécutées séquentiellement. Nous avons reporté de la littérature, qu'en moyenne le délai de l'AM est 1/13 fois moins important que celui de la MM [24]. Ceci nous a conduit à exécuter quatre AM en parallèle avec la MM.

4.2.1.3- Doublement en J^m avec trois multiplieurs

La figure 4.3 présente le séquencement d'un doublement du point $P(X_1; Y_1; Z_1; aZ_1^4)$ en utilisant trois multiplieurs et un seul additionneur. Le délai de cette structure équivaut à celui du calcul de trois MM et onze AM. L'utilisation de quatre multiplieurs n'est pas intéressante parce que on ne peut pas faire en parallèle plus de trois multiplications MM.

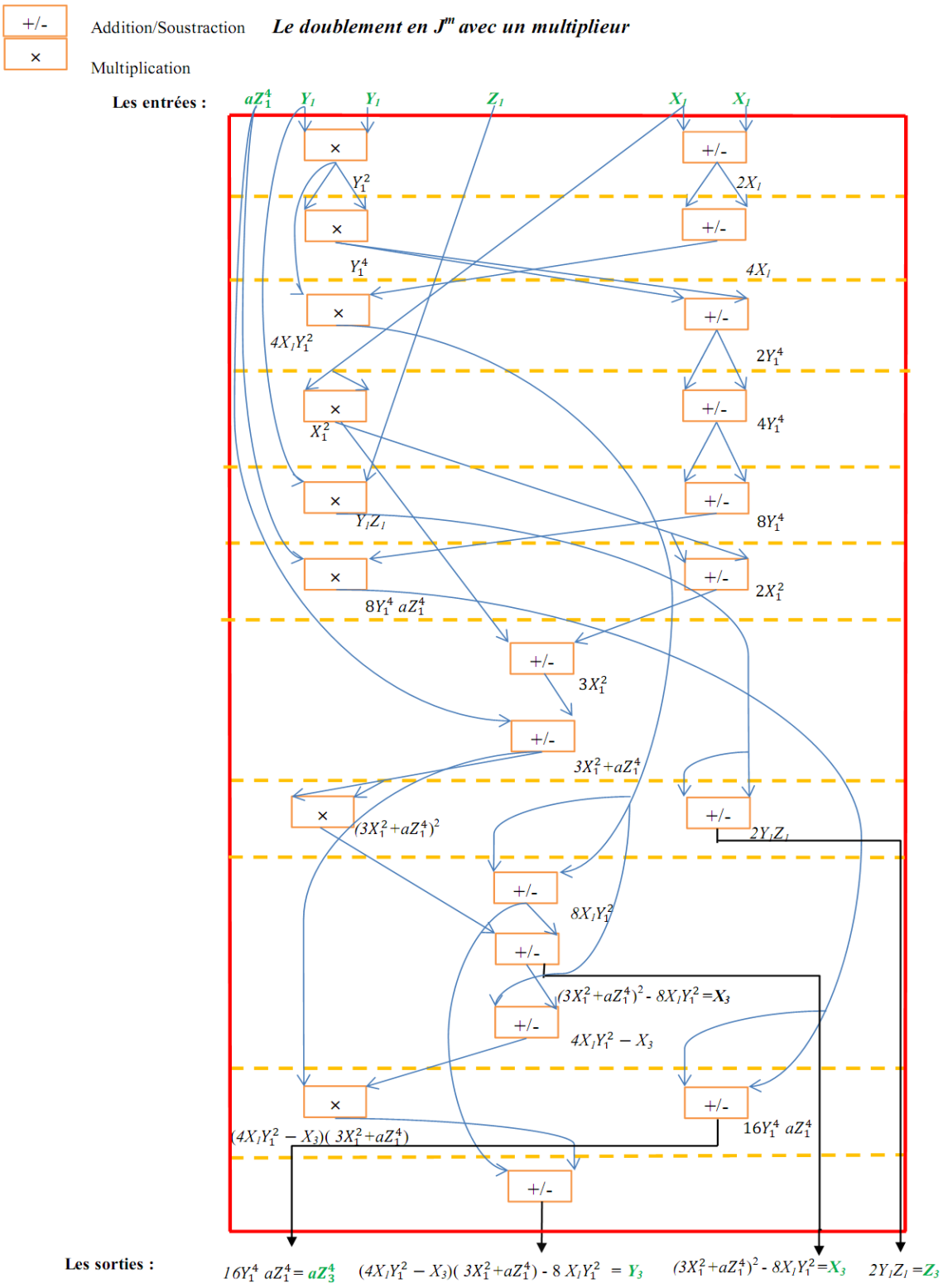


Figure 4.1 : Doublement en J^m avec un seul multiplieur

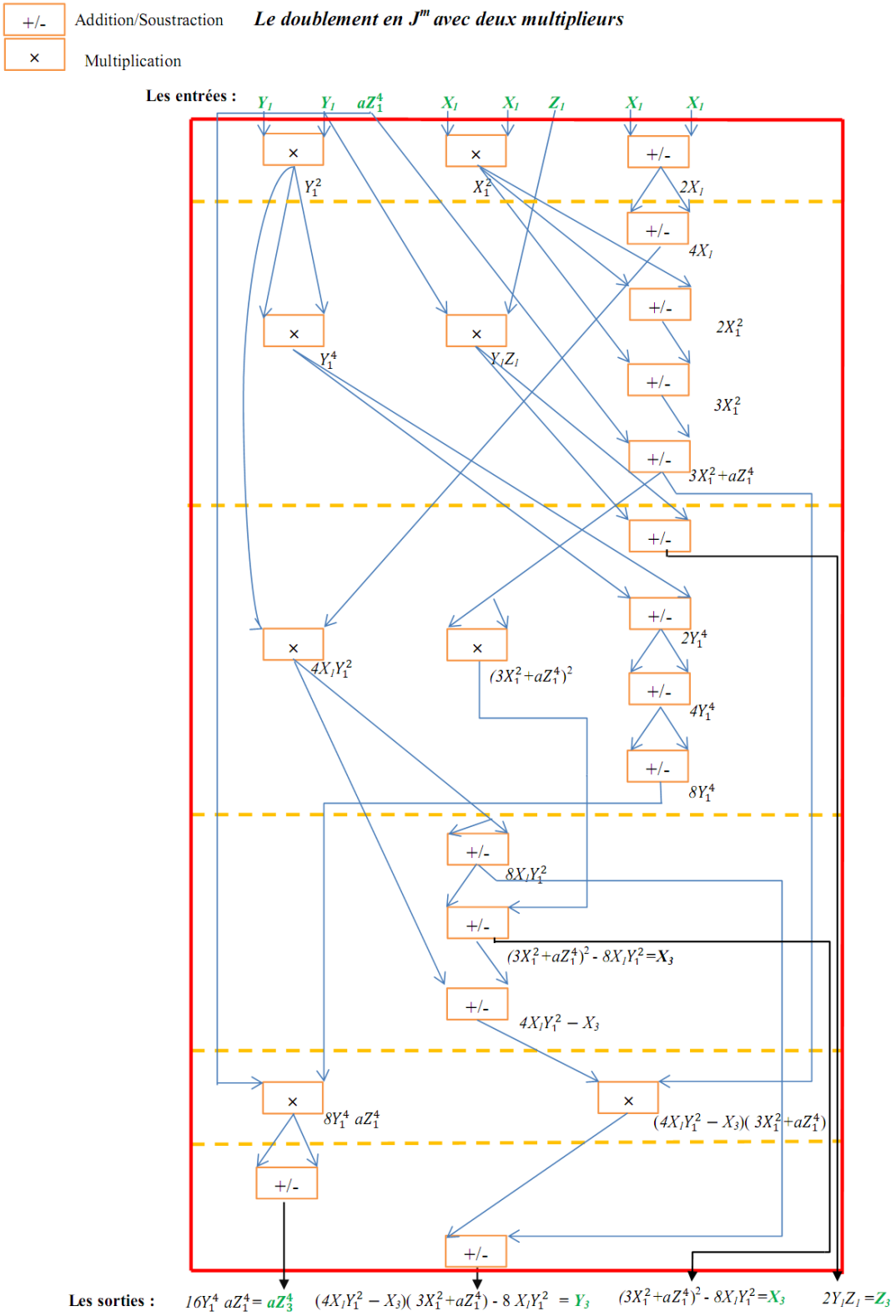


Figure 4.2 : Doublement en J^m avec deux multiplieurs

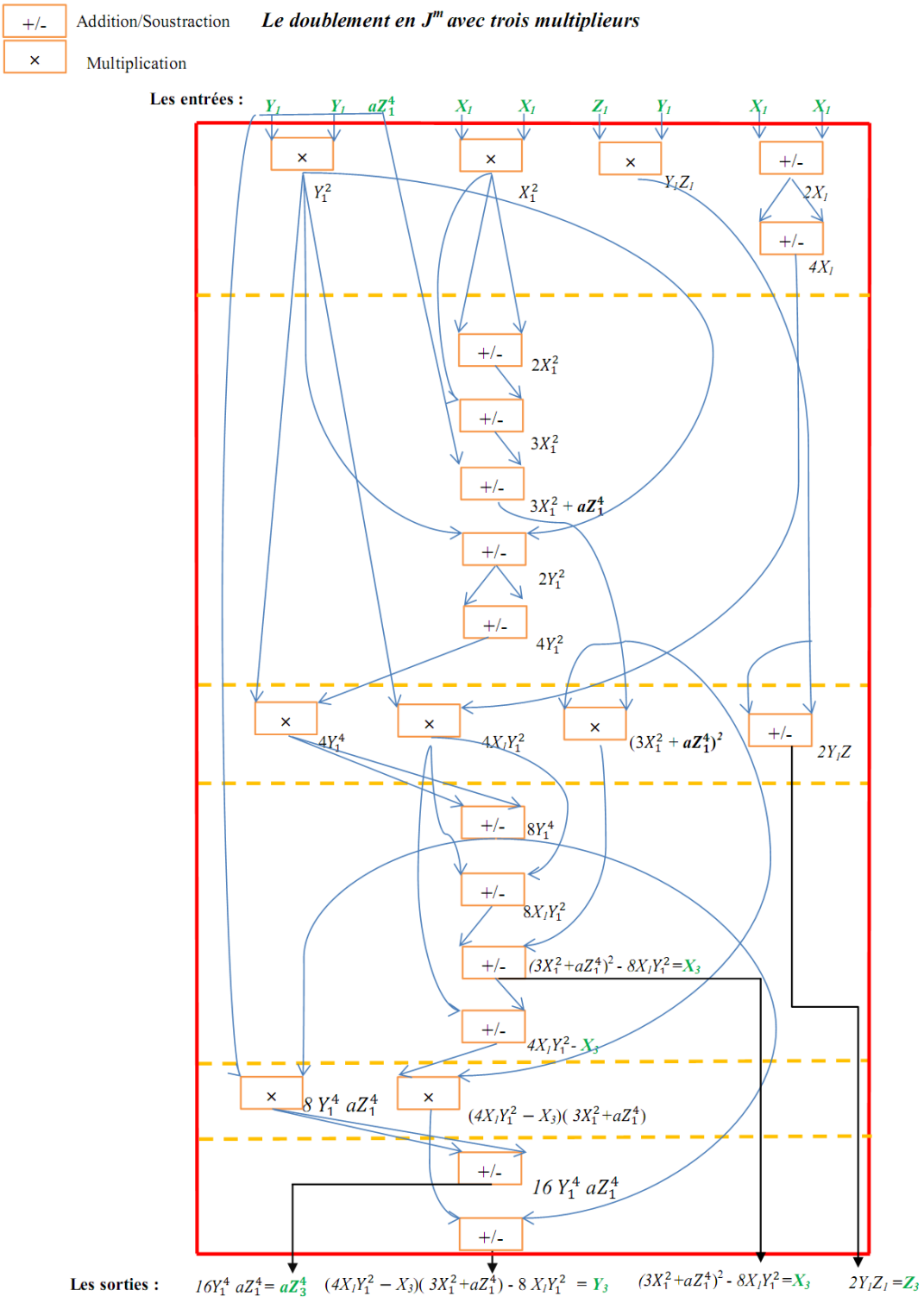


Figure 4.3 : Doublement en J^m avec trois multiplieurs.

4.2.2- La structure de l'addition en coordonnées J^c

L'addition de deux points $G_1(X_1; Y_1; Z_1; Z_1^2; Z_1^3)$ et $G_2(X_2; Y_2; Z_2; Z_2^2; Z_2^3)$ en coordonnées Jacobiennes Chudnovsky donne un nouveau point $G_3 =(X_3; Y_3; Z_3; Z_3^2; Z_3^3)$ où ses coordonnées sont comme suit :

$X_3 = -H^3 - 2U_1H^2 + r^2;$

$Y_3 = -S_1H^3 + r(U_1H^2 - X_3);$

$Z_3 = Z_1Z_2H;$

$Z_3^2 = Z_3^2;$

$Z_3^3 = Z_3^3;$

Avec

$U_1 = X_1 (Z_2^2);$

$U_2 = X_2(Z_1^2);$

$S_1 = Y_1(Z_2^3);$

$S_2 = Y_2(Z_1^3);$

$H = U_2 - U_1;$

$r = S_2 - S_1;$

Selon l'analyse des formules ci-dessus, nous avons extrait le séquencement des multiplications modulaires (V_1 à V_{14}) et les additions/soustractions modulaires (D_1 à D_6) de l'addition comme suit :

1. $V_1 \leftarrow X_1 Z_2^2 = U_1$
2. $V_2 \leftarrow X_2 Z_1^2 = U_2$
3. $V_3 \leftarrow Y_1 Z_2^3 = S_1$ $\quad D_1 \leftarrow V_2 - V_1 = H$
4. $V_4 \leftarrow Y_2 Z_1^3 = S_2$
5. $V_5 \leftarrow (D_1)^2 = H^2$ $\quad D_2 \leftarrow V_4 - V_3 = r$
6. $V_8 \leftarrow V_1 V_5 = U_1H^2$
7. $V_9 \leftarrow V_5 D_1 = H^3$ $\quad D_3 \leftarrow 2V_8 = 2U_1H^2$
8. $V_6 \leftarrow Z_1 Z_2$
9. $V_{10} \leftarrow V_6 D_1 = Z_1 Z_2 H = Z_3$ $\quad D_4 \leftarrow V_5 + D_3 = H^3 + 2U_1H^2$
10. $V_7 \leftarrow (D_2)^2 = r^2$
11. $V_{11} \leftarrow V_3V_9 = S_1 H^3$ $\quad D_5 = X_3 \leftarrow V_7 - D_4 = -H^3 - 2U_1H^2 + r^2$
12. $\quad D_6 = V_8 - X_3 = U_1H^2 - X_3$
13. $V_{12} \leftarrow D_2D_6 = r(U_1H^2 - X_3)$
14. $\quad D_7 = Y_3 \leftarrow V_{12} - V_{11} = -S_1H^3 + r(U_1H^2 - X_3)$
15. $V_{13} = (V_{10})^2 = Z_3^2$
16. $V_{14} = V_{10} V_{13} = Z_3^3$

4.2.2.1- Additionnement en J^c avec un seul multiplieur

La figure 4.4 présente le séquencement de l'additionnement des points $G_1(X_1; Y_1; Z_1; Z_1^2; Z_1^3)$ et $G_2(X_2; Y_2; Z_2; Z_2^2; Z_2^3)$ en utilisant un seul multiplieur et un seul additionneur. Dans cette structure, le délai du chemin critique équivaut celui de quatorze multiplications et de deux additions modulaires.

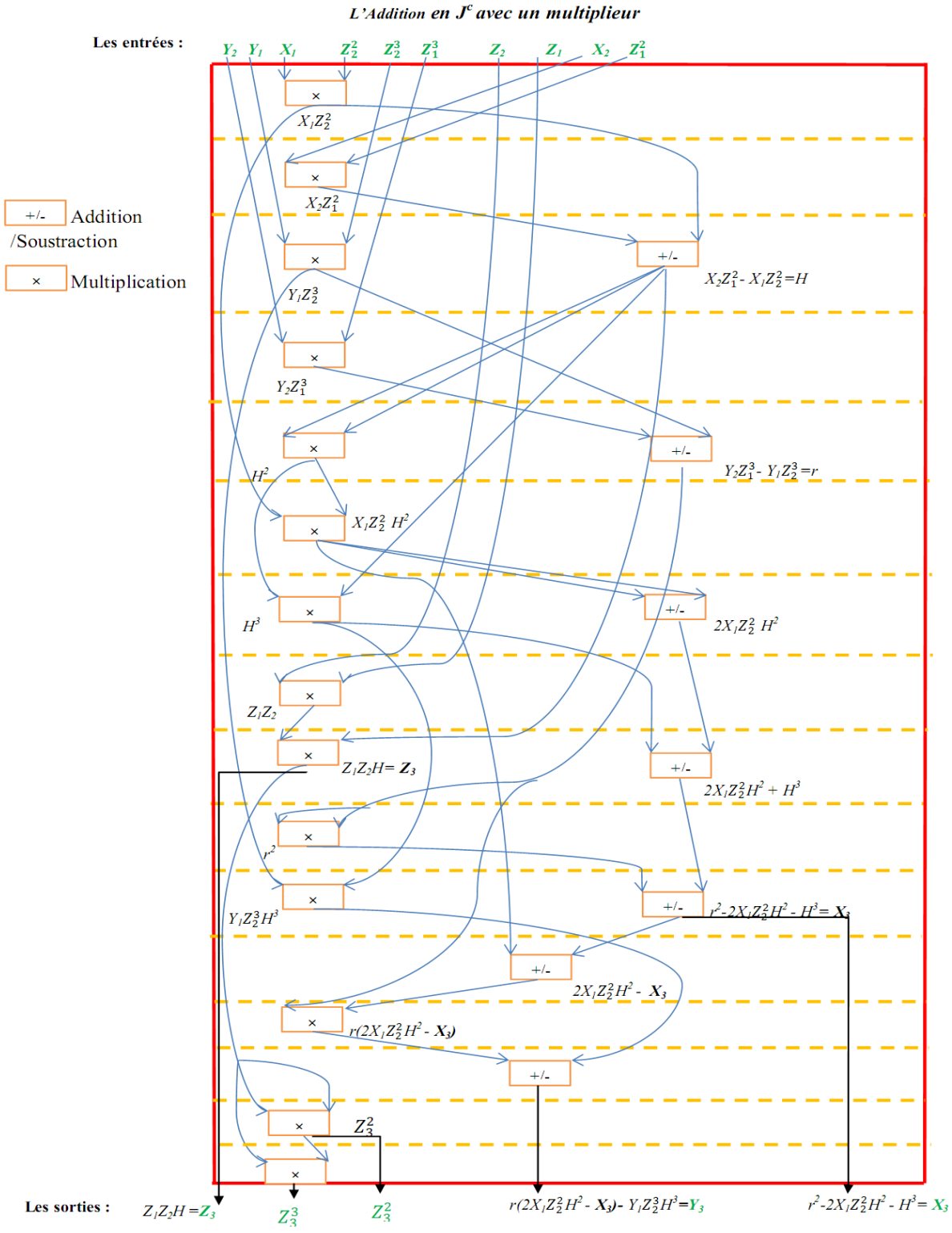


Figure 4.4 : Additionnement en J^c avec un seul multiplieur.

4.2.2.2- Additionnement en J^c avec deux multiplieurs

La figure 4.5 présente le séquencement d'additionnement de deux points $G_1(X_1; Y_1; Z_1; Z_1^2; Z_1^3)$ et $G_2(X_2; Y_2; Z_2; Z_2^2; Z_2^3)$ en utilisant deux multiplieurs et un seul additionneur. Dans cette structure, le délai du chemin critique équivaut à celui de sept multiplications modulaires.

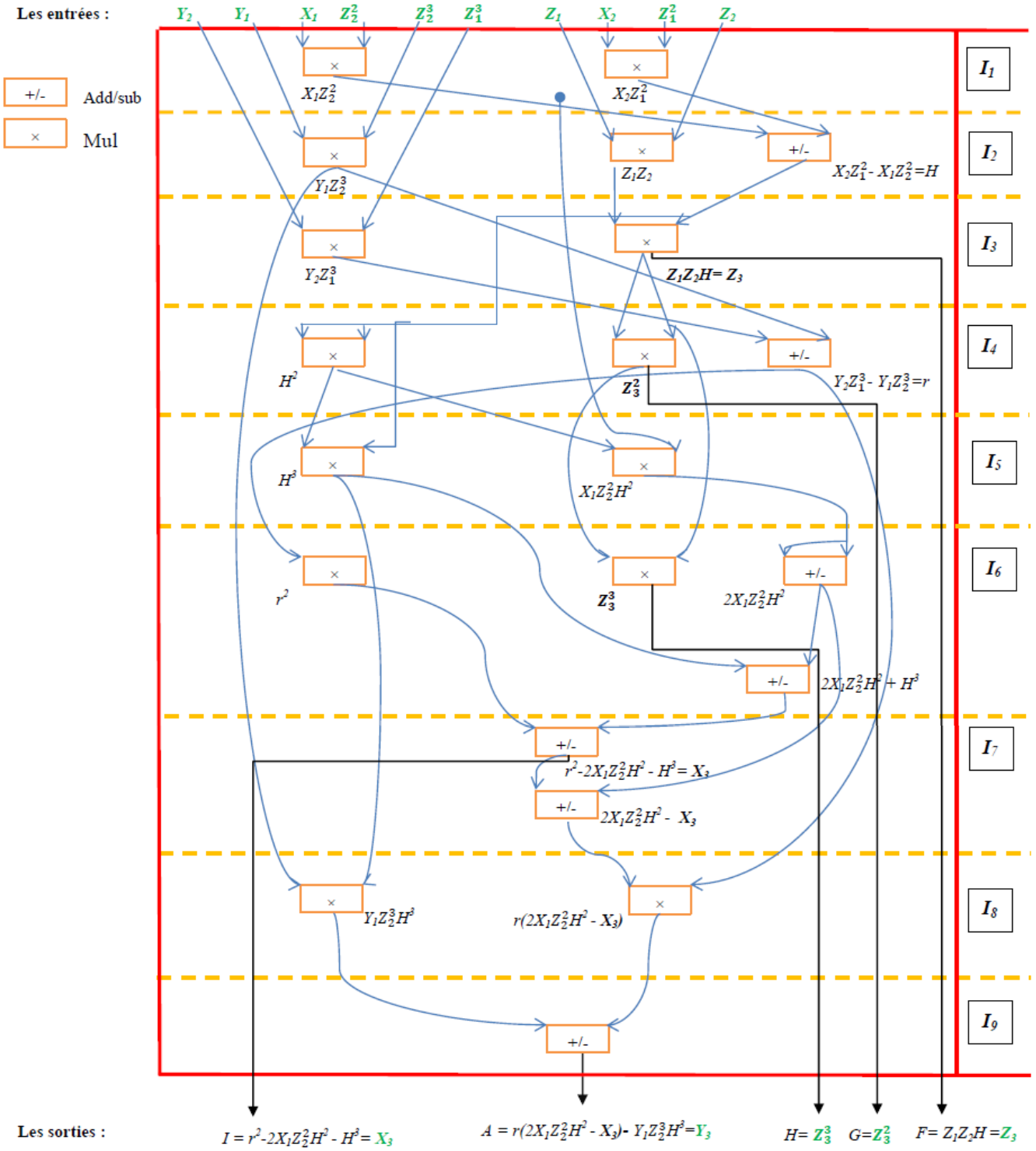


Figure 4.5 : Additionnement en J^c avec deux multiplieurs.

4.2.2.3- Additionnement en J^c avec trois multiplieurs

La figure 4.6 présente le séquencement de l'additionnement des points $G_1(X_1; Y_1; Z_1; Z_1^2; Z_1^3)$ et $G_2(X_2; Y_2; Z_2; Z_2^2; Z_2^3)$ en utilisant trois multiplieurs et un seul additionneur. Il en résulte un délai qui équivaut à celui de trois multiplications et de onze additions modulaires.

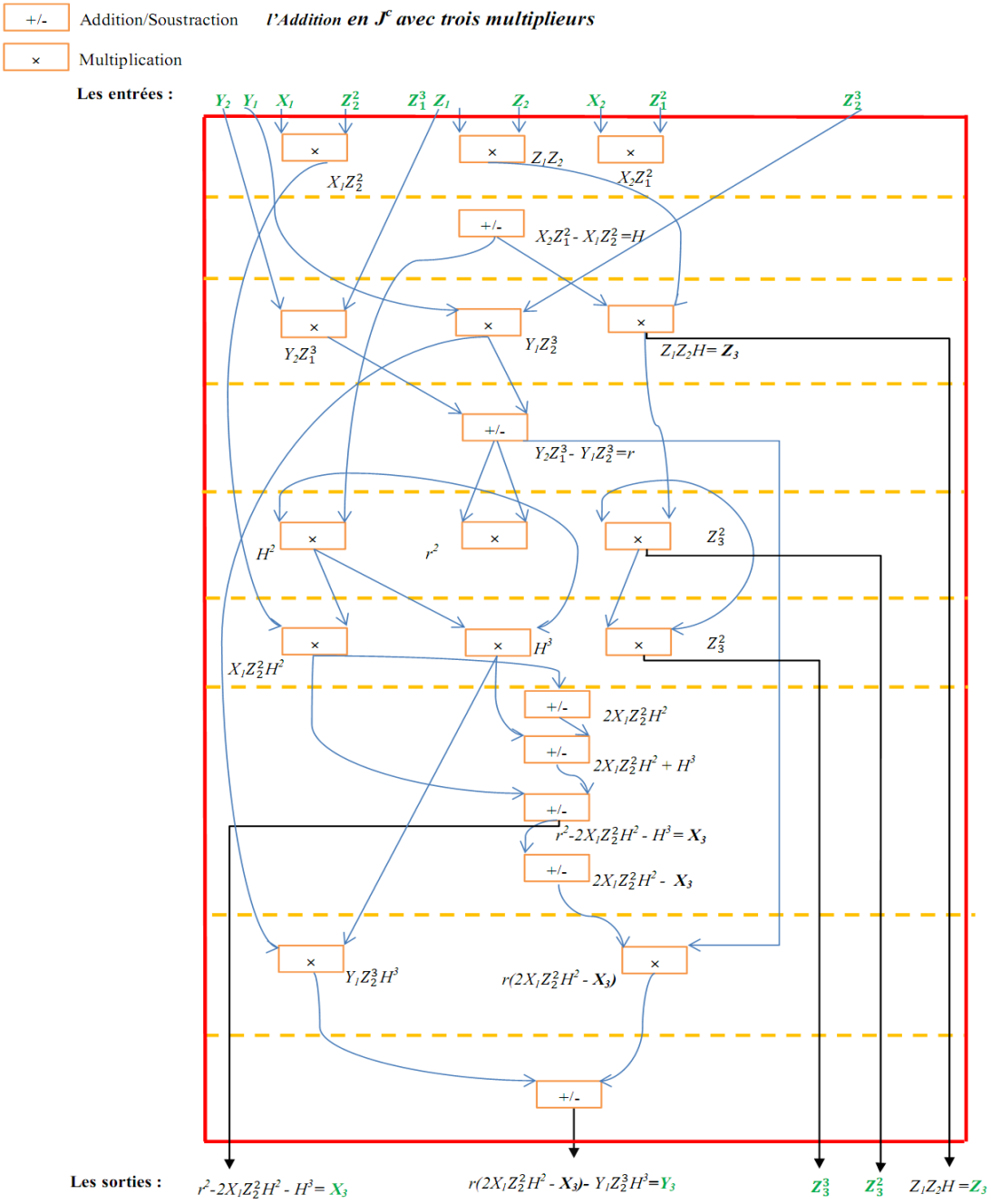


Figure 4.6 : Additionnement en J^c avec trois multiplieurs.

Le tableau 4.1 récapitule les délais de chacune des structures d'architectures présentées ci-dessus pour le doublement et l'addition de points.

T_{MM} : le temps d'une multiplication modulaire T_{AM} : le temps d'une addition/soustraction modulaire

Structure	Délai du Doublement	Délai de l'Addition
1 MM et 1 AM	$8* T_{MM} + 6 * T_{AM}$	$14 * T_{MM} + 2 * T_{AM}$
2 MM et 1 AM	$4* T_{MM} + 5 * T_{AM}$	$7 * T_{MM} + 4 * T_{AM}$
3 MM et 1 AM	$3* T_{MM} + 11 * T_{AM}$	$5 * T_{MM} + 7 * T_{AM}$

Tableau 4.1: Délai des différentes structures pour le doublement et l'addition de points.

Selon le tableau 4.1 le meilleur compromis Surface/Pérformance est dans la deuxième structures qui utilise deux multiplieurs MM et un additionneur AM. Le délai d'un doublement équivaut à celui du calcul de 4×MM et de 5×AM, alors que le délai de l'addition de points équivaut celui du calcul de 7×MM et de 4×AM. Cette configuration sera adoptée pour la conception de notre architecture. Nous remarquons que dans la structure adoptée, nous utilisons deux opérations qui sont la multiplication modulaire at l'addition modulaire, dans ce qui suit, nous nous pencherons sur l'optimisation des architectures de ces deux opérations en termes de pérformance et de surface occupée.

4.3- Opérations arithmétiques modulaires

Le multiplieur modulaire ainsi que l'additionneur modulaire représentent les deux modules de base de notre crypto système ECC. La performance de ce dernier dépend étroitement des performances de ces deux modules avec un degré moindre pour l'additionneur.

4.3.1- La multiplication modulaire

La multiplication modulaire représente la cellule de base pour le crypto système ECC. Elle consiste à effectuer la multiplication de deux nombres X et Y et prendre le reste S obtenu de la division du produit (X×Y) par un troisième nombre *p*, appelé modulo. Cette opération est donnée par l'expression *:* $Z = (X \times Y)$ *mod p.*

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

L'algorithme le plus populaire pour la multiplication modulaire est la méthode de Montgomery [27]. L'approche de Montgomery évite la division qui est le goulot d'étranglement pour d'autres algorithmes.

4.3.1.1- La multiplication Montgomery

En 1985, Montgomery a introduit une méthode très efficace pour effectuer la multiplication modulaire avec un *p* fixé [27].

A l'origine, la multiplication modulaire de Montgomery calcule non pas $S = X \times Y \ mod\ p$, mais une réduction avec un facteur supplémentaire, c'est-à-dire $Z = X \times Y \times R^{-1} \ Mod\ p$ où $R = 2^n$ et n est le nombre de bits de *p*.

La multiplication de Montgomery offre deux avantages majeurs : le premier est que les opérations utilisées sont des additions et des décalages, le deuxième est la longueur des résultats intermédiaires d'une taille fixe de n + 1 bits. L'idée de base de Montgomery est l'addition d'un multiple de *p* pour les résultats intermédiaires ne modifie pas la valeur du résultat final qui est calculée modulo *p*, où *p* est un nombre impair. Ainsi, le bit le moins significatif (LSB) du résultat intermédiaire est inspecté. S'il vaut 1, c'est à dire le résultat intermédiaire est impair, on ajoute *p* pour le rendre paire. Ce nombre est divisé par deux sans reste. Cette division par deux réduit le résultat intermédiaire à (n + 1) bits de nouveau.

L'inconvénient de cette méthode est que la multiplication modulaire de Montgomery est effectuée en trois étapes : la premier est de convertir X et Y dans le domaine de Montgomery qui consiste à calculer $mon(X) = (X \times R^2 \times R^{-1}) = (X \times R)$, $mon(Y) = (Y \times R^2 \times R^{-1}) = (Y \times R)$, puis on calcule la multiplication modulaire dans le domaine de Montgomery : $Montgomery\ (X, Y, p)$ et enfin faire sortir le résultat du domaine de Montgomery par $Montgomery\ (Z, 1, p)$.

En donnant X et Y de n bits, l'algorithme 4.1 donne le résultat de la multiplication modulaire de Montgomery $Z = X*Y*R^{-1} \bmod p$.

Algorithme 4.1 : «Multiplication modulaire de Montgomery»

Entrées : $X, Y < p, p, n \quad X = \sum_{i=0}^{n} x_i . 2^i, \ R=2^n$
Sorties : $Z = X \times Y \times R^{-1} \bmod p$
1. $k := 0$
2. for i = 0 to (n-1) do;
   2.1. $a := k + x_i \times Y$
   2.2. Si $(a \bmod 2) = 0$ alors $k := a/2$
   2.3. Sinon $k := (a + p)/2$
   Fin si
3. Si $k \geq p$ alors $Z := k - p$
   Sinon $Z := k$
Fin si;

4.3.1.2- La multiplication Montgomery avec compresseur 4 : 2

Une optimisation de l'algorithme 4.1 de Montgomery avec un compresseur 4:2 donné dans l'algorithme 4.2 qui permet la réduction de la propagation des retenues des additions [28].

Algorithme 4.2 : « Multiplication de Montgomery avec compresseur 4:2 »

Entrées : X, Y< p, p, n $\quad X = \sum_{i=0}^{n} x_i . 2^i,\ R=2^n$
Sorties : Z = X×Y×R^{-1} mod p
1. $S(0) = (S_s, S_c)(0) = 0$;
2. for i = 0 to ($n-1$) do;
   2.1. $q_i = \big((x_i \times y_0) + S_s(i)_0 + S_c(i)_0\big) mod 2$;
   2.2. $(S_s, S_c)(i+1) = (S_s(i) + S_c(i) + x_i.Y + q_i.p)/2$;
3. T := $S_s(n-1) + S_c(n-1)$
4. Si T ≥ p alors $Z := T - p$
   Sinon $Z :=$ T
Fin si;

La figure 4.7 représente l'architecture de la multiplication de Montgomery avec un compresseur 4:2. Cette architecture se compose de quatre registres : le registre Y contenant la valeur du multiplicande, le multiplicateur est stocké dans le registre X qui est un registre décalage à un bit pour chaque itération et les registres S, C qui stockent les résultats du compresseur sous forme redondante S pour la somme et C pour la retenue. Un CPA-192 (Carry Propagate Adder) donne le résultat final après l'addition de S et C. Un autre additionneur CPA-193 calcule la soustraction finale en additionnant (–p) au résultat final et enfin un multiplexeur pour sélectionner la valeur de Z.

Le compresseur 4:2 est un circuit à cinq entrées (A, B, C, D, Cin) et trois sorties (Sum, Carry, Cout) où les sorties représentent la somme des cinq entrées comme le montre la figure 4.8(a) et le compresseur 4:2 peut être implémenté en utilisant deux CSA (Carry Save Adder) comme le montre la figure 4.8(b).

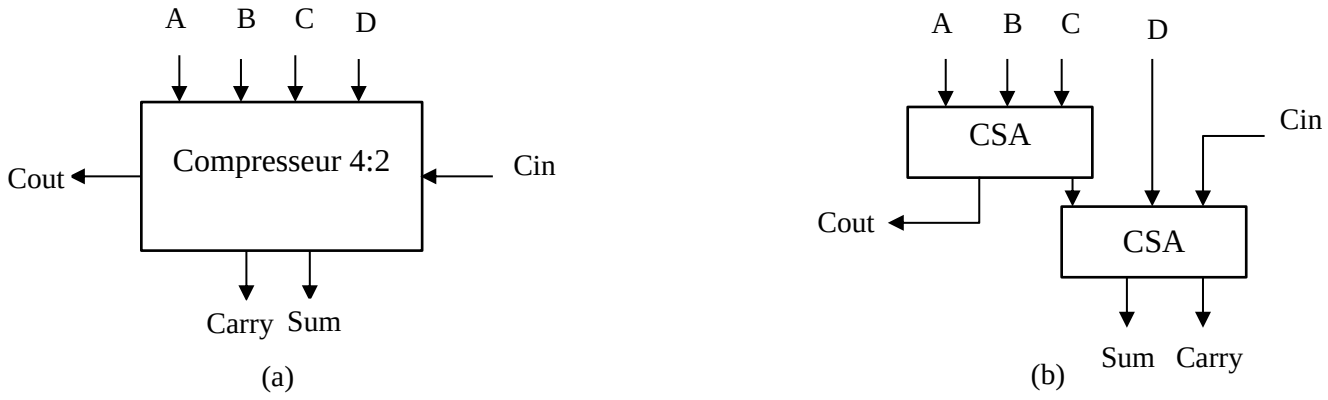


Figure 4.8 : Représentation d'un compresseur 4:2.

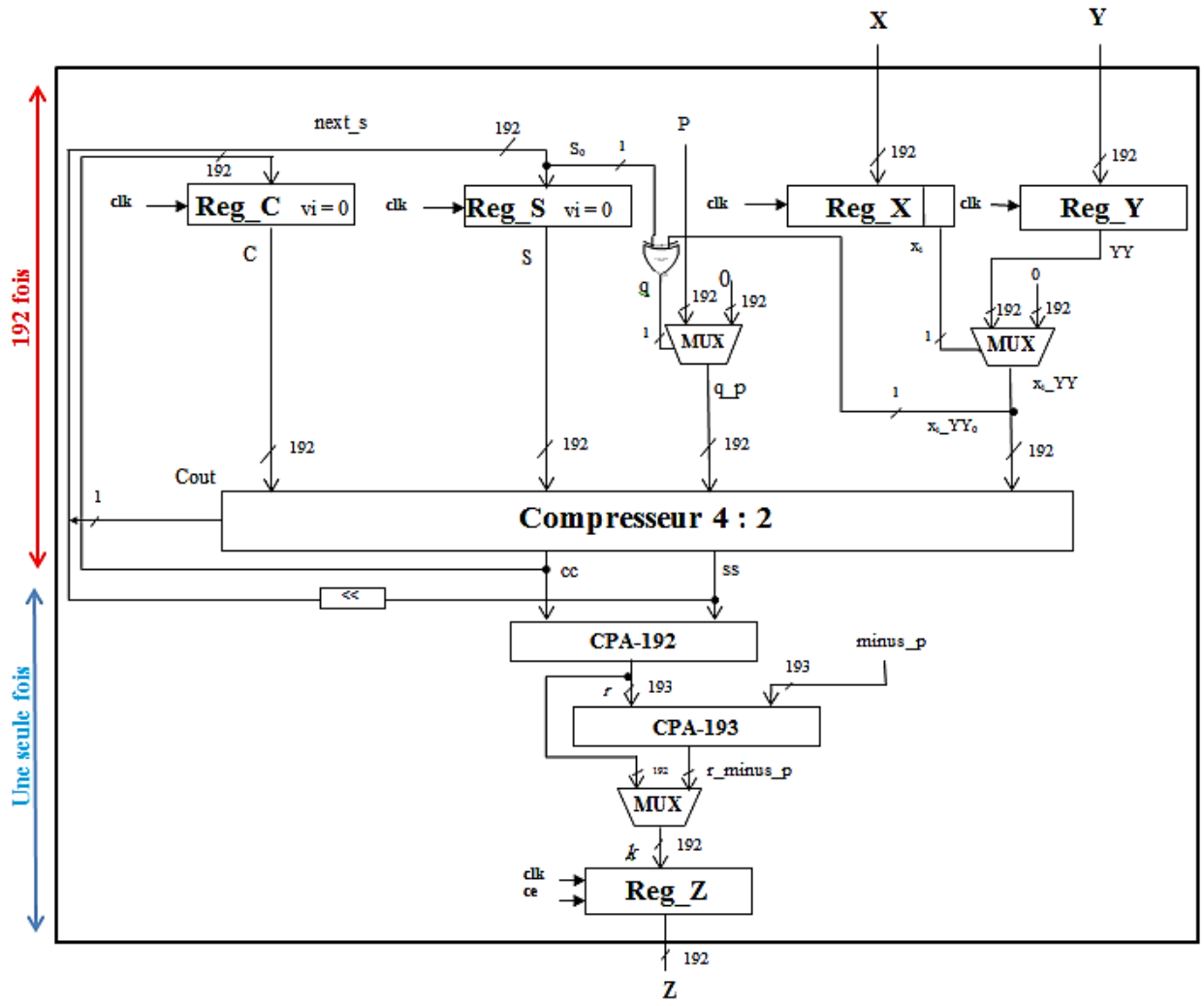


Figure 4.7 : Architecture de la multiplication de Montgomery avec compresseur 4 : 2.

La table de vérité qui correspond au compresseur 4 : 2 est décrite comme suit :

Entrées				Cin = 0		Cin = 1		Cout
A	B	C	D	Carry	Sum	Carry	Sum	
0	0	0	0	0	0	0	1	0
0	0	0	1	0	1	1	0	0
0	0	1	0					
0	1	0	0					
1	0	0	0					
0	0	1	1	0	0	0	1	1
0	1	1	0					
1	1	0	0					
0	1	0	1					
1	0	1	0					
1	0	0	1					
0	1	1	1	0	1	1	0	1
1	1	1	0					
1	1	0	1					
1	1	1	0					
1	1	1	1	1	0	1	1	1

Tableau 4.2 : Table de vérité du compresseur 4:2.

Le module de la multiplication de Montgomery sera utilisé comme un bloc dans l'architecture du crypto système ECC les autres modules comme le montre la figure 4.9.

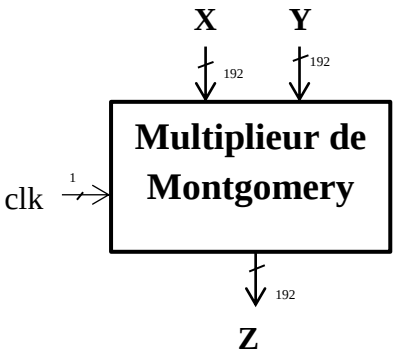


Figure 4.9 : Bloc de multiplication de Montgomery.

4.3.2- L'addition et la soustraction modulaire

Afin de calculer le résultat d'une addition et soustraction modulaire, on utilise les algorithmes 4.3 et 4.4 respectivement décrits ci-dessous.

L'addition modulaire de A et B (pour $A, B< p$) consiste à soustraire le module p de la somme $(A + B)$ si $(A+B)\geq p$, sinon en renvoie $(A+B)$.

Algorithme 4.3 : « Addition modulaire »

Entrées : p, $A = (a_{n-1} \cdots a_1 a_0) < p$, $B = (b_{n-1} \cdots b_1 b_0) < p$.
Sorties : $S = A + B \pmod p$.
1. $S \longleftarrow A + B$
2. $S' \longleftarrow S - p$
3. Si $S' < 0$ alors Retourner S
4. Sinon Retourner S'
5. fin si

De même pour la soustraction modulaire de A et B (pour $A, B< p$) consiste à additionner p à la soustraction $(A - B)$ si $(A - B) < 0$, sinon en renvoie $(A – B)$.

Algorithme 4.4 : « Soustraction modulaire »

Entrées : p, $A = (a_{n-1} \cdots a_1 a_0) < p$, $B = (b_{n-1} \cdots b_1 b_0) < p$.
Sorties : $S = A - B \pmod p$.
1. $S \longleftarrow A - B$
2. $S' \longleftarrow S + p$
3. Si $S < 0$ Alors Retourner S'
4. Sinon Retourner S
5. fin si

La figure 4.10 représente l'architecture de l'Additionneur/Soustracteur modulaires dans un même bloc. Le calcul de (x + y) ou de (x - y) se fait selon le signe de l'opération :

add_sub = 0 ⟹ une addition,

add_sub = 1 ⟹ une soustraction.

Nous gardons le résultat dans sum1 puis nous calculons [sum1-*p*] si l'opération est une addition ou [sum1+*p*] si l'opération est une soustraction. Ceci est déterminé par le signe inverse de add_sub. Nous gardons le résultat dans sum2 pour enfin choisir le résultat final en utilisant un multiplexeur 2:1 selon la fonction Sel = (NOT(add_sub) AND carry2) OR (add_sub AND NOT(carry1)), où la fonction Sel est construite selon la table de vérité du tableau 4.3.

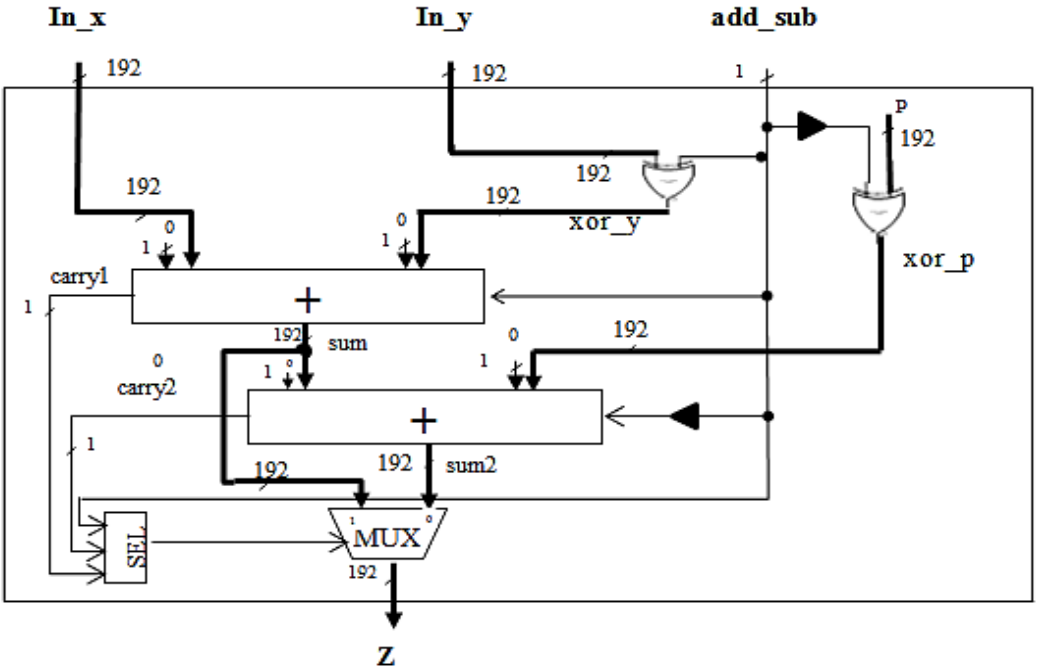


Figure 4.10 : L'architecture de l'addition/soustraction modulaire.

add_sub	Carry 1	Carry 2	Sel
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Tableau 4.3 : Table de vérité de la fonction sel.

De même, le module d'addition/soustraction modulaire sera utilisé comme un bloc dans les autres modules comme le montre la figure 4.11.

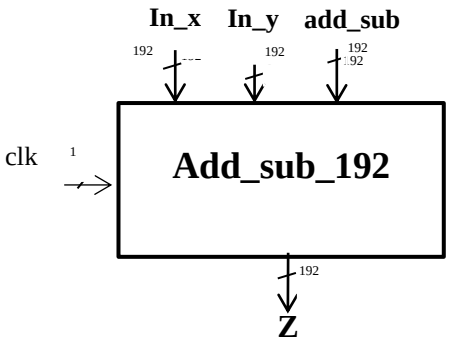


Figure 4.11 : Le bloc d'addition/soustraction modulaire.

4.3.3- L'inversion modulaire

4.3.3.1- Petit théorème de Fermat

Du petit théorème de Fermat, nous savons que si p est un nombre premier alors : $a^{p-1} \equiv 1 \bmod p$. Avec ce théorème, l'inversion modulaire peut être facilement calculé en multipliant les côtés par a^{-1} est on a donc : $a^{-1} \equiv a^{p-2} \bmod p$. Cela peut évidemment être mis en œuvre dans le matériel par une exponentiation modulaire, et donc à l'aide d'une série de multiplications modulaires : c'est une méthode d'inversion facile à implanter à l'aide du module de multiplication de Montgomery [29].

4.3.3.2- Les algorithmes d'exponentiation modulaire

Il existe deux types d'exponentiation modulaire : le premier consiste à parcourir les bits de l'exposant à partir du poids fort MSB au poids faible LSB ou de gauche à droite (L-R pour Left-to-Right) et pour chaque bit nous réalisons une élévation au carré et nous ajoutons une multiplication seulement si le bit est différent de zéro. Dans cette méthode ces deux opérations se réalisent en séquentiel (algorithme 4.5) [30].

Algorithme 4.5 : « Méthode binaire L–R pour calculer a^{-1} »

Entrées : $a, p, e = p - 2 = (e_0 e_1 \dots e_{n-1})_2$
Sorties : $C = a^{-1} \, mod \, p$
Début
1. $C = 1, \; S = a$
2. Pour i de 0 jusqu'a $n - 1$ faire
   2.1. Si $(e_i = 1)$ alors $C = S \times C \bmod p$
   2.2. $S = S \times S \bmod p$
3. Retourner C

Fin.

La deuxième méthode est la R-L pour (Right-to-Left), consiste à parcourir les bits à partir du LSB et pour chaque bit nous effectuons une élévation au carré qui sera suivie d'une multiplication seulement si le bit est non nul, les deux opérations dans cette méthode se font en parallèle (algorithme 4.6) [30].

Algorithme 4.6 : « Méthode binaire R-L pour calculer a^{-1} »

Entrées : $a, p, e = p - 2 = (e_0 e_1 \dots e_{n-1})_2$
Sorties : $C = a^{-1} \; mod \; p$
Début
1. Si e_{n-1} = 1 alors $C = a$ sinon $C = 1$ fin si
2. Pour i de $n - 2$ jusqu'a 0 faire
   2.1. $C = C^2 \bmod p$
   2.2. Si $(e_i = 1)$ alors $C = C \times a \bmod p$ fin si
3. Retourner C
Fin.

4.3.3.3 – L'architecture de l'inversion modulaire

L'opération d'inversion modulaire est utilisée une seule fois dans une multiplication scalaire pour convertir les coordonnées Jacobiennes en coordonnées Affine. Comme nous avons choisi d'utiliser deux multiplieurs pour l'additionnement et le doublement, donc l'algorithme 4.6 est le plus adapté parce qu'il supporte le parallélisme.

La figure 4.12 représente l'implémentation de l'inversion modulaire correspondante à Algorithme 4.6. Cette architecture utilise le module de multiplication de Montgomery avec deux registres S, C pour stocker les résultats intermédiaires et un registre à décalage E d'un bit qui contient la valeur de l'exposant (*p-2*) ainsi qu'un contrôleur permettant au multiplexeur de sélectionner la valeur de $\boldsymbol{a}$ et au registre C de s'initialiser avec « 1 » si l'opération est à sa première itération.

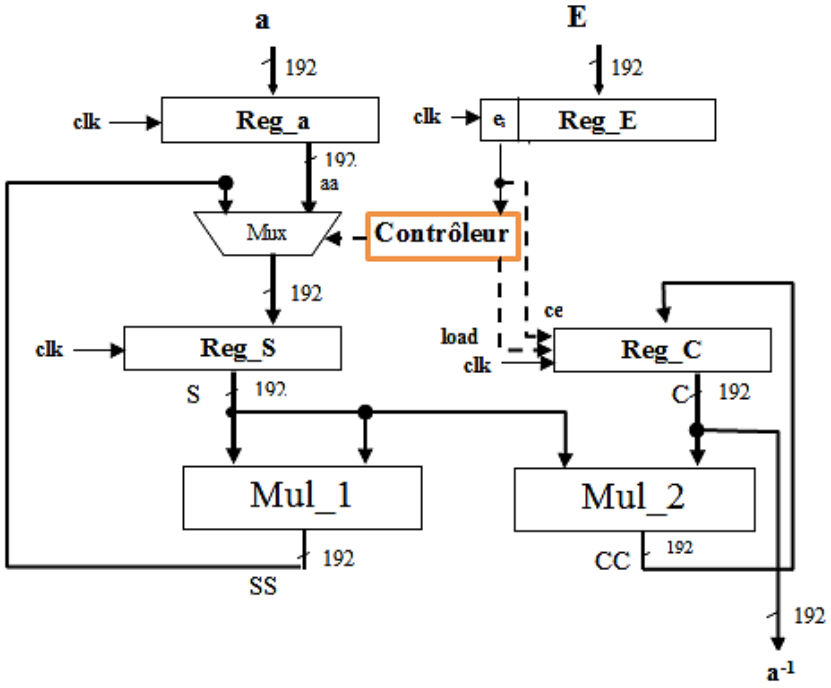


Figure 4.12 : L'architecture de l'inversion modulaire.

4.4- Architecture du doublement et de l'addition de points

Les opérations d'additionnement et le doublement de points sont basées sur les opérations modulaires (addition/soustraction, multiplication) et comme on a vu dans la section 4.2 le meilleur temps d'exécution est obtenu par l'utilisation de deux multiplieurs et d'un additionneur.

La figure 4.13 représente l'architecture d'additionnement/doublement où nous avons utilisé dix registres pour stocker les résultats intermédiaires, et pour chaque registre on a placé un multiplexeur 8:1 pour sélectionner l'un des résultats issus du premier multiplieur, du deuxième multiplieur, l'additionneur/soustracteur ou garde la valeur initiale (Aiguilleur de sortie). L'aiguilleur d'entrée se compose de six multiplexeurs 16:1, afin d'orienter les données du registres vers les deux multiplieurs et l'additionneur/soustracteur. Les figures 4.15 et la 4.16 présentent la circulation des données dans les registres pour un additionnement et un doublement respectivement.

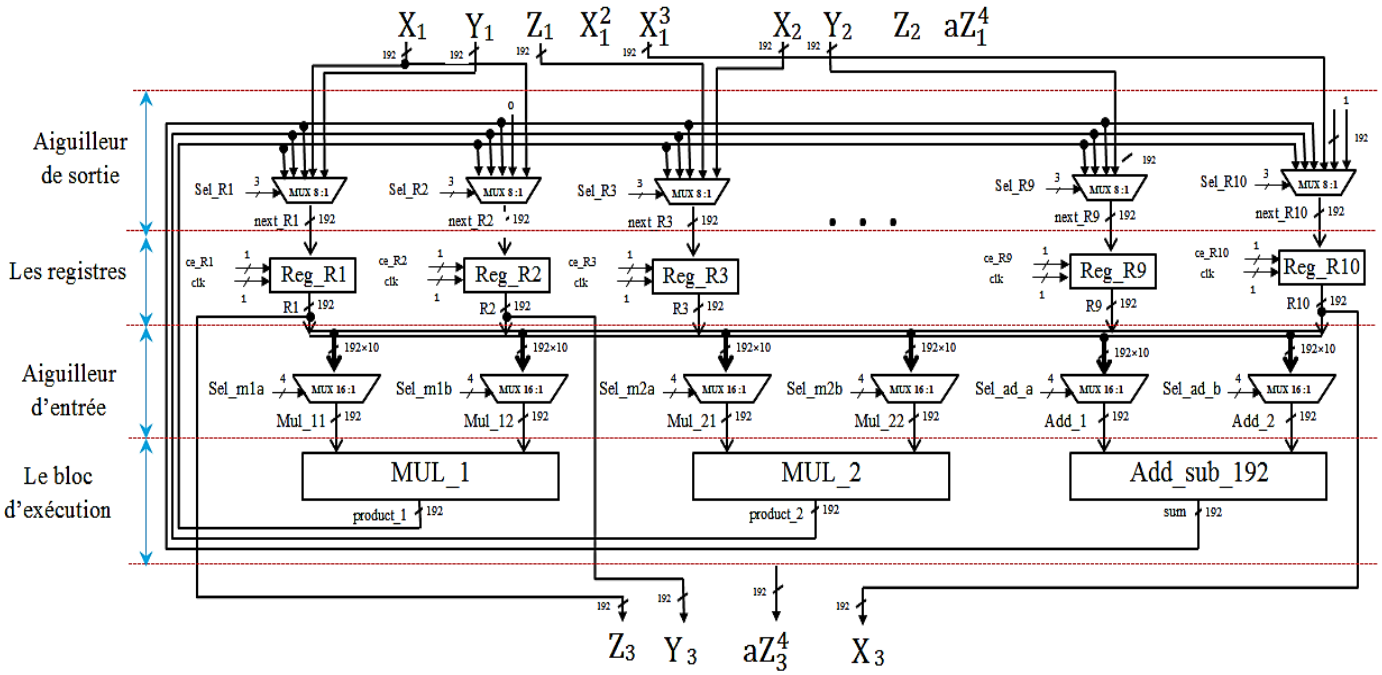


Figure 4.13 : L'architecture détaillée du bloc Mul_add.

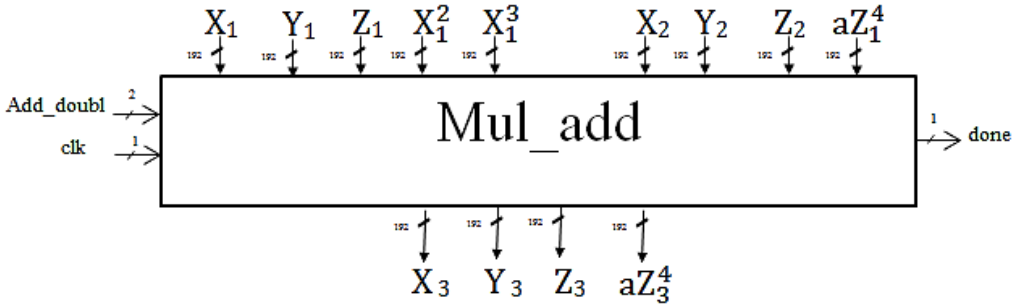


Figure 4.14 : L'architecture globale bloc Mul_add.

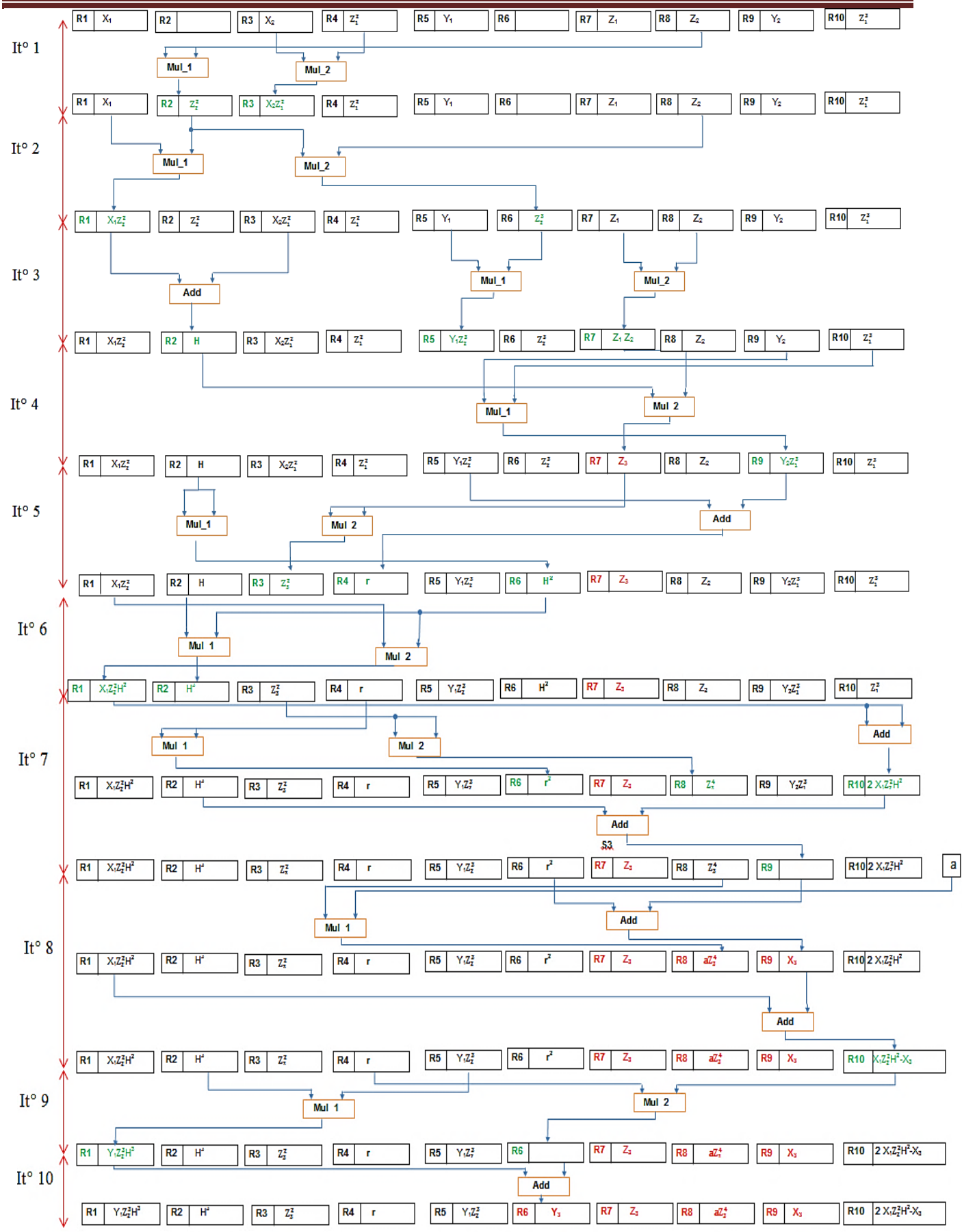


Figure 4.15 : La circulation des données pour un additionnement.

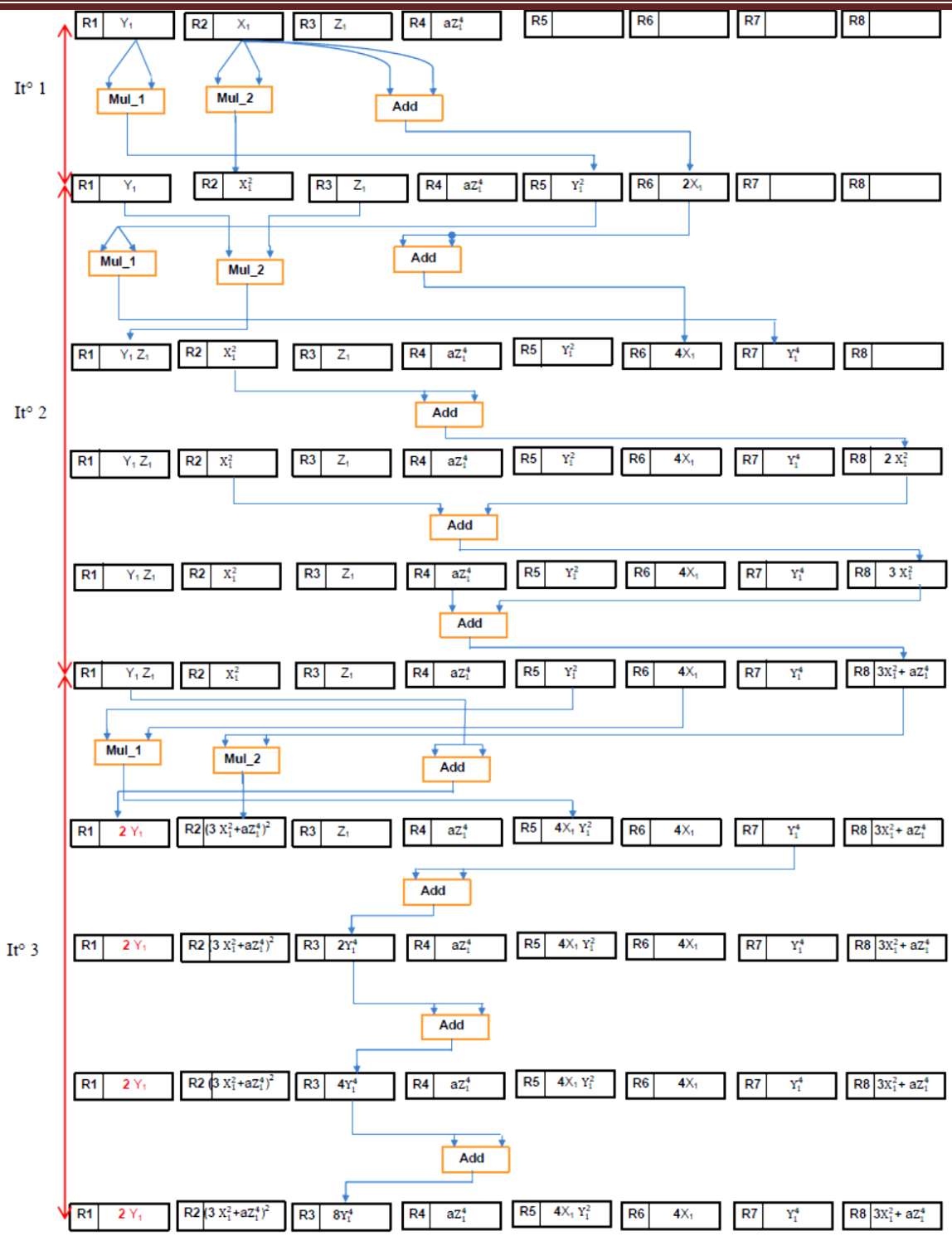

It° 1
It° 2
It° 3
Mul_1
Mul_2
Add
R1 Y1
R2 X1
R3 Z1
R4 aZ1^4
R5
R6
R7
R8
R1 Y1
R2 X1^2
R3 Z1
R4 aZ1^4
R5 Y1^2
R6 2X1
R7
R8
R1 Y1 Z1
R2 X1^2
R3 Z1
R4 aZ1^4
R5 Y1^2
R6 4X1
R7 Y1^4
R8
R1 Y1 Z1
R2 X1^2
R3 Z1
R4 aZ1^4
R5 Y1^2
R6 4X1
R7 Y1^4
R8 2 X1^2
R1 Y1 Z1
R2 X1^2
R3 Z1
R4 aZ1^4
R5 Y1^2
R6 4X1
R7 Y1^4
R8 3 X1^2
R1 Y1 Z1
R2 X1^2
R3 Z1
R4 aZ1^4
R5 Y1^2
R6 4X1
R7 Y1^4
R8 3X1^2+ aZ1^4
R1 2 Y1
R2 (3 X1^2+aZ1^4)^2
R3 Z1
R4 aZ1^4
R5 4X1 Y1^2
R6 4X1
R7 Y1^4
R8 3X1^2+ aZ1^4
R1 2 Y1
R2 (3 X1^2+aZ1^4)^2
R3 2Y1^4
R4 aZ1^4
R5 4X1 Y1^2
R6 4X1
R7 Y1^4
R8 3X1^2+ aZ1^4
R1 2 Y1
R2 (3 X1^2+aZ1^4)^2
R3 4Y1^4
R4 aZ1^4
R5 4X1 Y1^2
R6 4X1
R7 Y1^4
R8 3X1^2+ aZ1^4
R1 2 Y1
R2 (3 X1^2+aZ1^4)^2
R3 8Y1^4
R4 aZ1^4
R5 4X1 Y1^2
R6 4X1
R7 Y1^4
R8 3X1^2+ aZ1^4

Figure 4.16 : La circulation des données pour un doublement.

4.5- L'architecture de la multiplication scalaire

La figure 4.17 présente un enchainement de l'algorithme 3.1 du chapitre 3 : «Doublement-et-Addition de gauche à droite» pour effectuer la multiplication scalaire, ces étapes sont détaillées comme suit :

1- La conversion du point *P* des coordonnées Affines aux coordonnées Jacobiennes modifiées :

$P_{\text{affine}} \rightarrow P_{\text{Jacobienne Modifiée}}$

$X_{P_{JM}} = X_{P_A}$
$Y_{P_{JM}} = Y_{P_A}$
$Z_{P_{JM}} = 1$
$aZ^4_{P_{JM}} = a$

2- Entrée dans le domaine de Montgomery par la conversion des quatre coordonnées du point *P* au domaine de Montgomery pour exécuter quatre multiplications modulaires de Montgomery, telles que :

$X_{P_{JM}}$ = Montgomery ($X._{P_A}$, R^2), ou $R = 2^n$ le nombre de bits de *p* (le modulo).
$Y_{P_{JM}}$ = Montgomery (Y_{P_A}, R^2),
$Z_{P_{JM}}$ = Montgomery (1, R^2),
$aZ^4_{P_{JM}}$ = Montgomery (a, R^2).

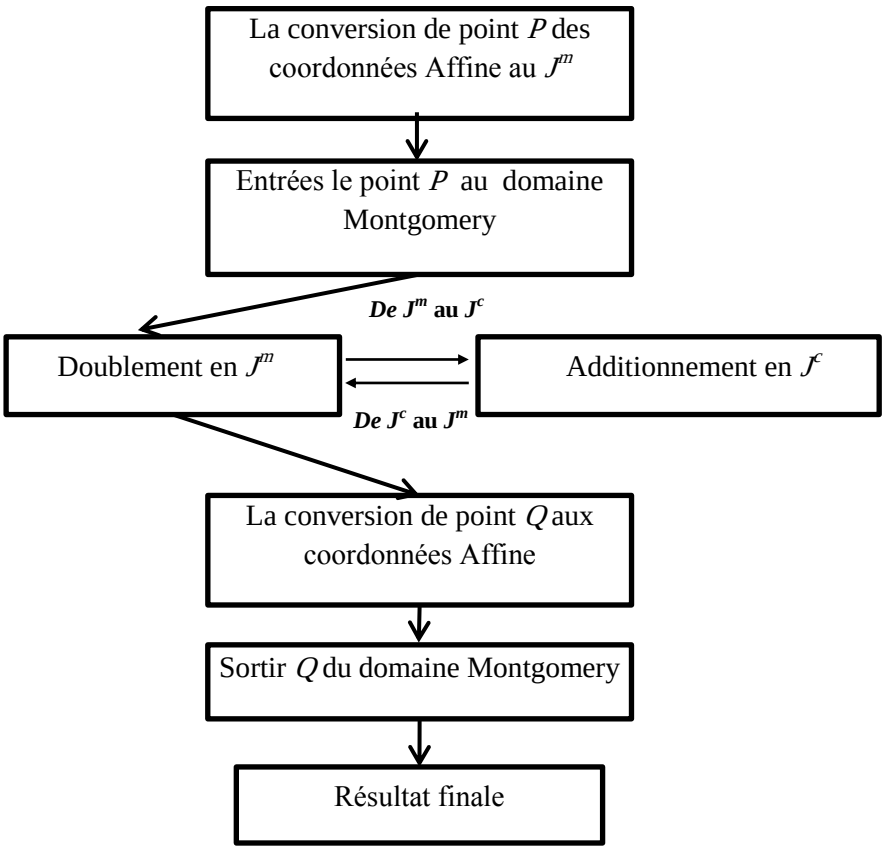


Figure 4.17 : Le séquencement de l'opération de la multiplication scalaire.

3- La figure 4.18 montre l'architecture globale de la multiplication scalaire, où le contrôleur doit parcourir les bits selon l'entrée du registre *k* de MSB au LSB, si $k_i = 0$ alors on fait un doublement seulement et si $k_i = 1$ alors on fait un doublement suivie d'un additionnement.

Le calcul se fait en coordonnées J^m pendant l'opération seulement pour l'additionnement le calcule est en J^c et le résultat est converti en J^m : Q Jacobienne Chudnovsky → Q Jacobienne modifé :

$Z_{Jm}^4 = Z_{Jc}^2 * Z_{Jc}^2$ élévation au carré, $aZ_3^4 = aZ_3^4$ multiplication modulaire;

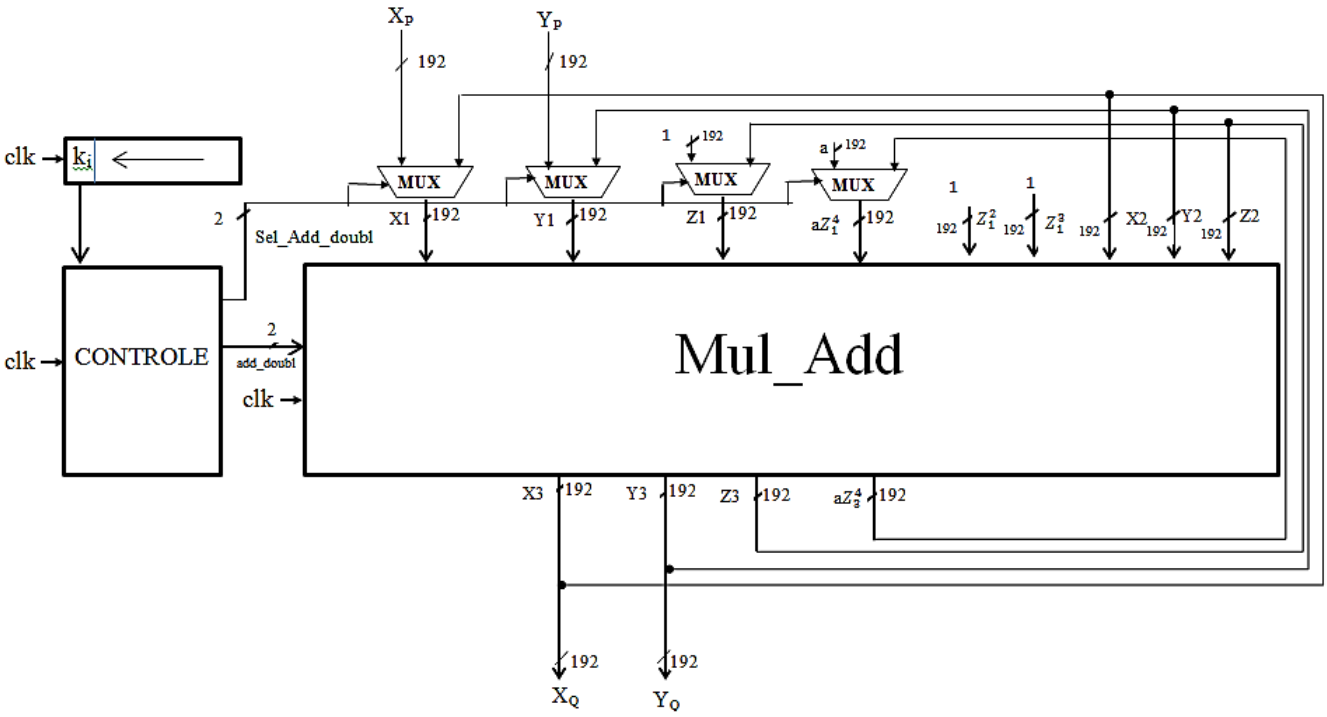


Figure 4.18 : L'architecture de l'opération de la multiplication scalaire.

4- La conversion du point *Q* des coordonnées Jacobiennes Modifiées (X, Y, Z, aZ_1^4) aux coordonnées Affines (x, y) se fait avec : $x = X/Z^2$, $y = Y/Z^3$ avec $Z \neq 0$ par une inversion modulaire (z^{-1}), une élévation au carré avec le multiplieur de Montgomery $(z^{-1})^2$ et trois multiplications modulaires avec le multiplieur de Montgomery $x(z^{-1})^2$, $(z^{-1})^3$, $y(z^{-1})^3$.

5- Sortir le résultat du domaine de Montgomery, pour convertir un point du domaine de Montgomery à la représentation normale ; il suffit d'effectuer une multiplication modulaire de X_Q = Montgomery (X_Q, 1), Y_Q = Montgomery (Y_Q, 1), et enfin nous obtenons le résultat final.

4.6- Conclusion

Dans ce chapitre, nous avons proposé une architecture hardware pour les différents blocs de la multiplication scalaire.

Nous avons commencé par la comparaison entre les différentes configurations possibles pour l'additionnement et le doublement afin de choisir la bonne configuration qui donne le meilleur résultat en terme de temps d'éxecution et en terme de surface occupée par le matériel sur le circuit FPGA.

Puis, nous avons détaillé l'architecture hardwares, les opérations modulaires de base (addition/soustraction, multiplication et inversion). Le multiplieur de Montgomery représente le noyeau de notre architecture, pour son amélioration nous avons utilisé un compresseur 4:2 afin d'éliminer la propagation des carrés.

Nous avons proposé une architecture pour l'additionnement et le doublement en utilisant deux multiplieurs et un additionneur et enfin, nous avons conçu l'architecture globle pour la multiplication scalaire.

CHAPITRE V

Résultats de Simulation et d'Implémentation

5.1- Introduction

Dans ce chapitre, nous allons présenter les résultats d'implémentation sur circuit FPGA de l'architecture permettant le calcul de la multiplication scalaire basée sur : la multiplication de Montgomery, l'addition modulaire et l'inversion modulaire; ainsi que les résultats de simulation de sa fonctionnalité.

En premier lieu, nous allons présenter l'environnement hardware (circuit FPGA) où sera implémentée notre architecture qui permet d'obtenir de bonnes performances du moment que les plateformes hardwares permettent d'atteindre des débits de traitement (chiffrement ou déchiffrement) très élevés par rapport aux implémentations softwares.

L'architecture développée dans ce travail a été conçue dans l'environnement ISE 12.2 de Xilinx. Dans le but de vérifier son bon fonctionnement, des simulations fonctionnelles et une comparaison avec un modèle mathématique de référence ont été effectuées, en utilisant respectivement les outils : ISim (M.63c) et Maple 13.

L'architecture a été décrite en langage VHDL selon l'approche Top to Down qui consiste en la décomposition de l'architecture en blocs modulaires. Chaque bloc peut à son tour être aussi décomposé en un ensemble de sous blocs fonctionnels, et ainsi de suite jusqu'au dernier niveau de la hiérarchie qui peut être un module composé d'un opérateur de base.

5.2- La définition de circuit FPGA

Un FPGA (Field Programmable Gate Array ou "réseaux logiques programmables") est un circuit logique reconfigurable ce qui permet de le reprogrammer par l'utilisateur afin d'exécuter les fonctions souhaitées.

Ce circuit offre deux avantages majeurs la reconfigurabilité et la puissance. La reconfigurabilité est une propriété essentielle qui donne plus de flexibilité au circuit, que ce soit la maintenance, mise à jour ou même on peut changer carrément l'application implémentée : tout revient à une simple reconfiguration des ressources logiques du circuit, et la puissance des FPGAs apparaître à leur capacité d'exécuter des fonctions parallèles simultanément. Le schéma de la figure 5.1 montre à quoi ressemble la structure interne du circuit FPGA.

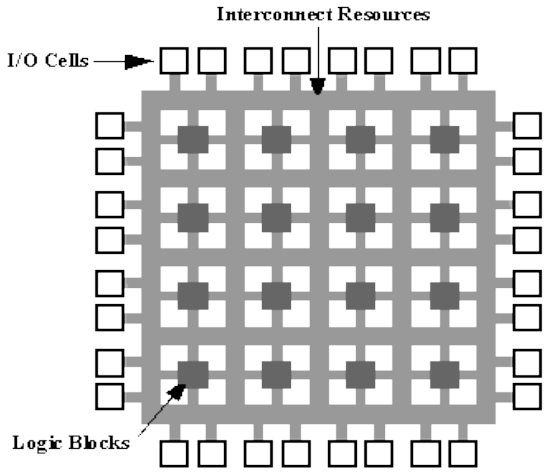


Figure 5.1 : Structure interne d'un circuit FPGA.

Cette architecture est constituée de trois éléments fonctionnels programmables fondamentaux :

- ✓ **Les CLBs (Blocs Logiques Configurables) :** chaque bloc est constitué d'un ensemble de LUTs (Look Up Tables) sur lesquels on implémente les applications. Ces blocs donnent la caractéristique de reconfigurabilité aux circuits FPGA.
- ✓ **Les IOBs (Blocs d'entrée/sortie) :** contrôlent le flot des données entre les pins d'E/S et la logique interne du circuit.
- ✓ **Les interconnexions :** sont très importantes car elles transmettent le signal d'un point à un autre.

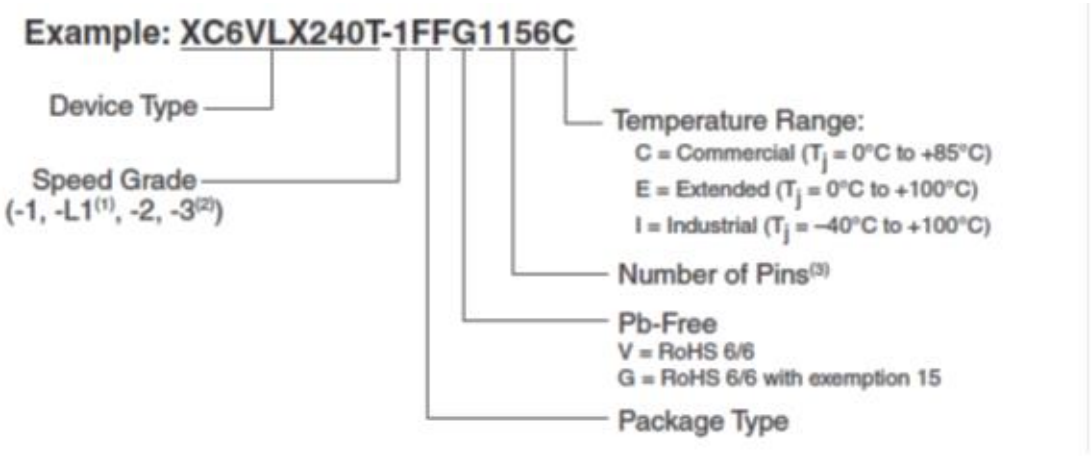
5.3- Etude de la famille Virtex-6

Les FPGAs de la famille Virtex-6 de Xilinx sont des circuits logiques programmables très flexibles, proposés sous trois familles :

1. Virtex-6 LXT FPGA qui est une logique de haute performance avec une connectivité série avancée.
2. Virtex-6 SXT FPGA: offre un traitement le plus élevé du signal et une capacité avec une connectivité série avancée.
3. Virtex-6 HXT FPGA: possède la plus haute bande passante de série connectivité. Elle est construite sur une technologie 40 nm [30].

5.3.1- Informations sur les Virtex-6

Voici un exemple de référence de Virtex-6 et le détail de sa nomenclature [30]:



5.3.2- Caractéristiques des FPGAs Virtex-6

Le tableau suivant résume les caractéristiques pour la famille Virtex-6 [30] :

Device	Logic Cells	Configurable Logic Blocks (CLBs)		DSP48E1 Slices[2]	Block RAM Blocks			MMCMs[4]	Interface Blocks for PCI Express[5]	Ethernet MACs[6]	Maximum Transceivers		Total I/O Banks[7]	Max User I/O[8]
		Slices[1]	Max Distributed RAM (Kb)		18 Kb[3]	36 Kb	Max (Kb)				GTX	GTH		
XC6VLX75T	74,496	11,640	1,045	288	312	156	5,616	6	1	4	12	0	9	360
XC6VLX130T	128,000	20,000	1,740	480	528	264	9,504	10	2	4	20	0	15	600
XC6VLX195T	199,680	31,200	3,040	640	688	344	12,384	10	2	4	20	0	15	600
XC6VLX240T	241,152	37,680	3,650	768	832	416	14,976	12	2	4	24	0	18	720
XC6VLX365T	364,032	56,880	4,130	576	832	416	14,976	12	2	4	24	0	18	720
XC6VLX550T	549,888	85,920	6,200	864	1,264	632	22,752	18	2	4	36	0	30	1200
XC6VLX760	758,784	118,560	8,280	864	1,440	720	25,920	18	0	0	0	0	30	1200
XC6VSX315T	314,880	49,200	5,090	1,344	1,408	704	25,344	12	2	4	24	0	18	720
XC6VSX475T	476,160	74,400	7,640	2,016	2,128	1,064	38,304	18	2	4	36	0	21	840
XC6VHX250T	251,904	39,360	3,040	576	1,008	504	18,144	12	4	4	48	0	8	320
XC6VHX255T	253,440	39,600	3,050	576	1,032	516	18,576	12	2	2	24	24	12	480
XC6VHX380T	382,464	59,760	4,570	864	1,536	768	27,648	18	4	4	48	24	18	720
XC6VHX565T	566,784	88,560	6,370	864	1,824	912	32,832	18	4	4	48	24	18	720

5.3.3- Les blocs logiques de base, CLBs (Blocs Logiques Configurables)

Chaque CLB est constitué de quatre SLICEs interconnectées. Ces slices sont disposées en paire : celle de gauche est appelée SLICEM et celle de droite SLICEL, ces deux éléments contiennent des LUTs (Look Up Table), deux éléments d'enregistrement et des multiplexeurs. Le tableau suivant résume les ressources logiques pour un CLB (Virtex-6) [31] [32].

Slices	LUTs	Flip-Flops	Arithmetic and Carry Chains	Distributed RAM(1)	Shift Registers(1)
2	8	16	2	256 bits	128 bits

5.4- ISE (Integrated Software Environment) de Xilinx

Tous les fabricants de FPGA proposent des outils CAO (**C**onception **A**ssistée par **O**rdinateur) pour configurer leurs circuits, par exemple pour la société Altera c'est QUARTUS et pour Xilinx c'est ISE.

ISE est un logiciel de programmation des produits Xilinx : CPLD (**C**omplex **P**rogrammable **L**ogique **D**evise), FPGA Spartan et Virtex …, téléchargeable gratuitement sur le site de Xilinx. Il intègre différents outils de CAO (**C**onception **A**ssistée par **O**rdinateur). Il dispose de [33] :

- Un éditeur de texte et de schéma.
- Un compilateur VHDL et Verilog.
- Un simulateur
- D'outils pour la gestion des contraintes temporelles.
- D'outils pour la synthèse.
- D'outils pour la vérification.
- D'outils pour l'implémentation sur FPGA et CPLD.

ISE est un outil de développement complet pour toutes les gammes de produits Xilinx, la Figure 5.2 identifie les principaux éléments de l'interface du Navigateur du Projet ISE 12.2 que nous avons utilisé dans ce projet.

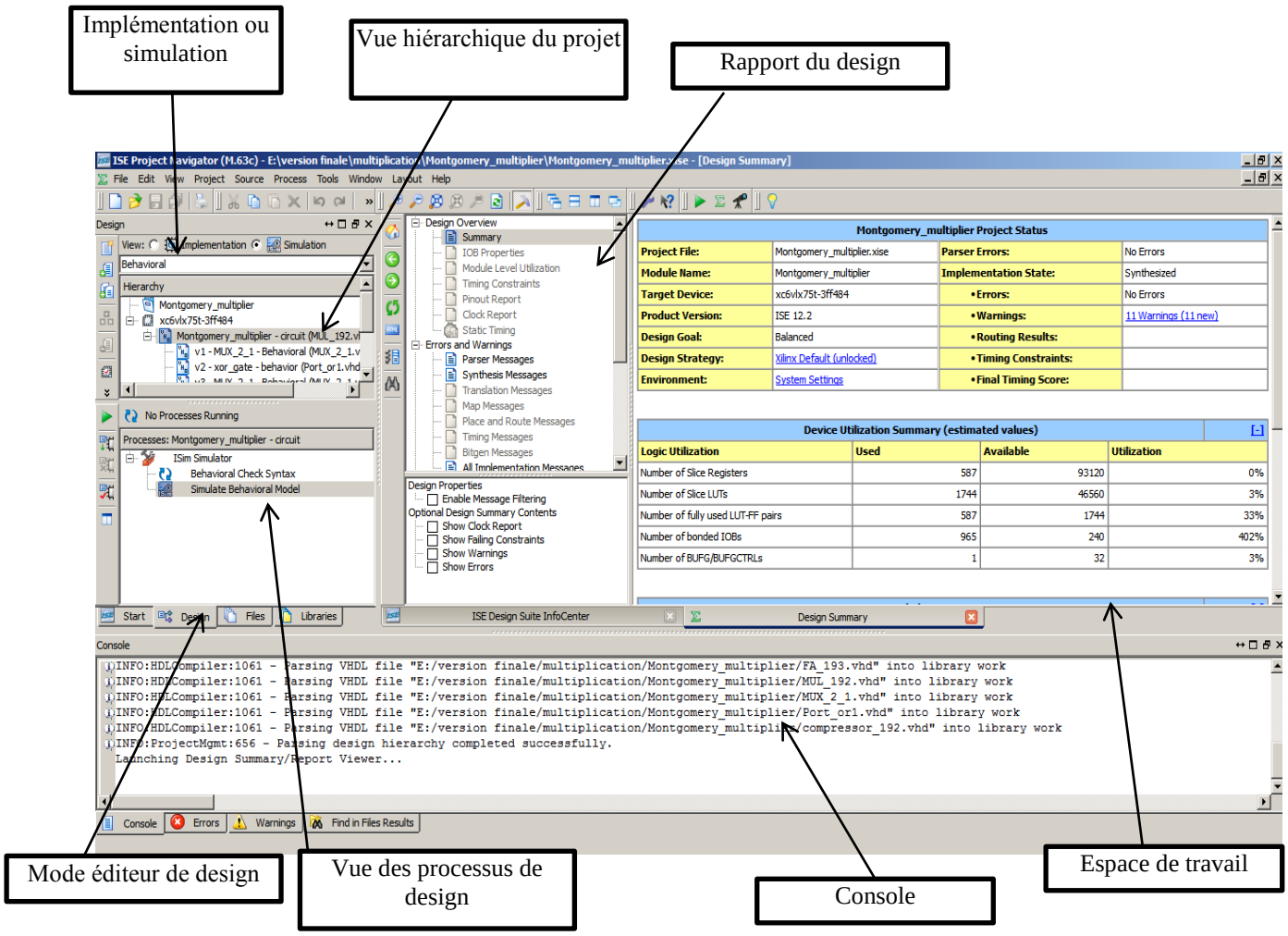


Figure 5.2 : L'interface principale d'ISE 12.2.

5.5- Etapes d'implémentation d'une spécification VHDL sur un FPGA

Pour implémenter une spécification VHDL dans le circuit FPGA, sur la plate-forme ISE trois étapes essentielles sont à suivre tel que le représente la figure 5.3 [33].

1. Description en VHDL des différents blocs composant l'architecture de calcul de la multiplication scalaire, à savoir : la multiplication de Montgomery, l'addition/soustraction modulaire, l'inversion modulaire, l'additionnement et le doublement d'un point [34]. La description de ces modules en VHDL est montrée dans l'annexe B.

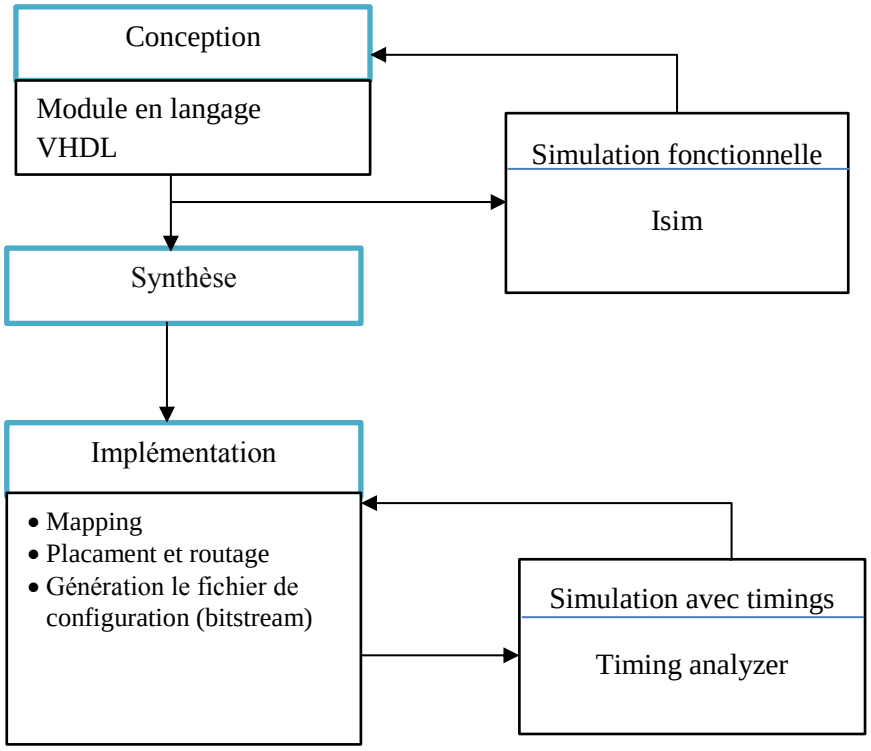


Figure 5.3 : Les étapes d'implémentation d'une spécification VHDL sur FPGA.

2. Simulation fonctionnelle par l'outil lSim de chacun de ces blocs et la vérification des résultats de simulation par l'outil Maple. Les procédures Maple utilisées sont présentées dans l'annexe A.
3. Synthèse de l'architecture par l'outil de synthèse XST de Xilinx.
4. Implémentation de l'architecture de la multiplication scalaire sur le circuit FPGA XC6VHX565T de la famille Virtex-6.

5.7- Les résultats de simulation et d'implémentation

Pour les tests, nous avons choisi la courbe elliptique *P-192* recommandée par le NIST (National Institute of Standards and Technology) où ses paramètres sont :

- a = -3,
- b = 2455155546008943817740293915197451784769108058161191238065,
- p = 6277101735386680763835789423207666416083908700390324961279.
- G = [205276475889337534449667902599269120263560324975669085786 2, 3235495846547773372963571948817489147168146215941690951212].
- n = 6277101735386680763835789423176059013767194773182842284081.

5.7.1- Multiplication de Montgomery à 192 bits

Afin de tester le bon fonctionnement du multiplieur de Montgomery, plusieurs simulations ont été faites avec des valeurs différentes ou les résultats de simulation vont être comparés avec les résultats donnés par l'outil Maple. Voici les valeurs utilisées pour un essai et ses résultats:

- x = 1226758474491164491549484325355196824873939855883911122940,
- y = 564991259338670531452547282180034069476687523495825638 2827,
- p = 627710173538668076383578942320766641608390870039032496 1279,
- r = 18446744073709551617,
- $x*r \bmod p$ = $(246116993320801334955585651344134838573263262126613880 1842)_{10}$,
- $y*r \bmod p$ = $(627710173538668076383578942320766641608390870039032496 1279)_{10}$,
- $x*r \bmod p$ = $(35D0430CC3F8E8920D1C8509CB92388E095436BF2FD6E208)_{16}$,
- $y*r \bmod p$ = $(645FD0249CD9A7977B513F19C1B1CB02FD5D45F3CA1FA6B2)_{16}$,
- $x*y*r \bmod p$ = $(99128EBCEE28A1C24E83B6A5CD29F4CA4D4A00951738A07C)_{16}$.

A)- Résultats de simulation avec (Isim)

Comme nous allons vu dans le chapitre précédent le multiplieur de Montgomery donne comme résultat $x*y*r^{-1}$, pour cela on a multiplié les valeurs de x et y par la valeur r avec Maple (entrée dans le domaine de Montgomery) et par conséquent le résultat obtenue reste dans le domaine de Montgomery c'est à dire multiplié par r. Toutes les valeurs de la figure suivante sont présenté en hexadécimale.

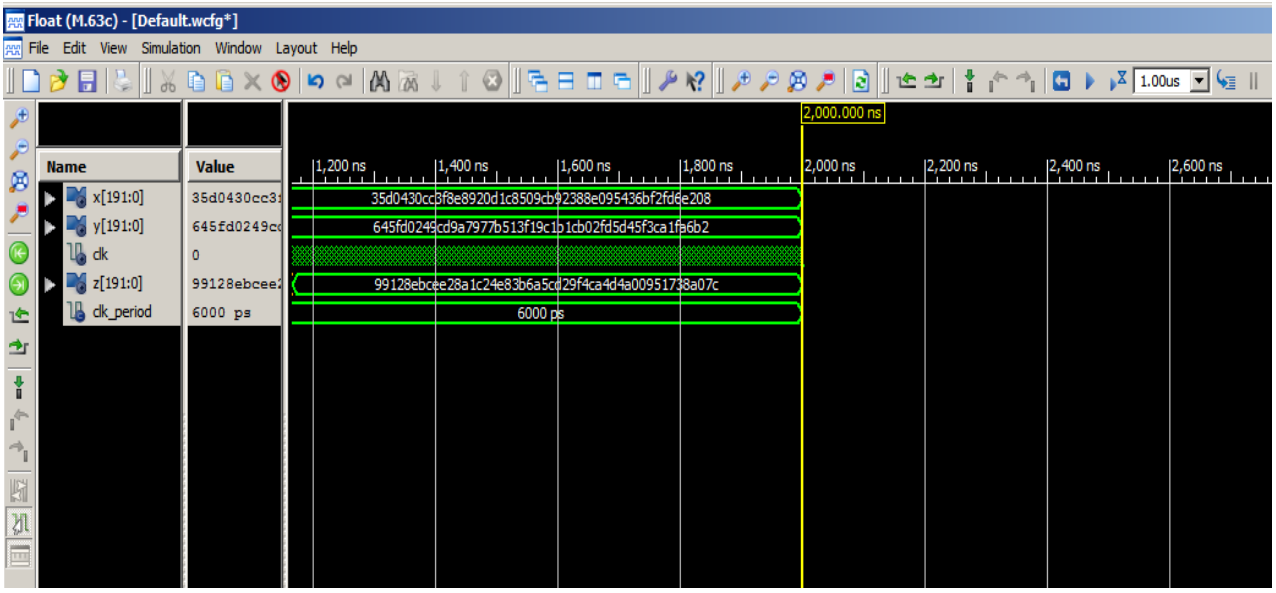


Figure 5.4 : Résultat de simulation du multiplieur de Montgomery.

B)- Résultats de Maple

Au début nous avons utilisé Maple pour entrées les valeurs de *x* et *y* dans le domaine de Montgomery, puis on a calculée la multiplication modulaire dans le domaine de Montgomery et en fin on a converti le résultat finale en hexadécimale. La figure suivante présente les résultats obtenue.

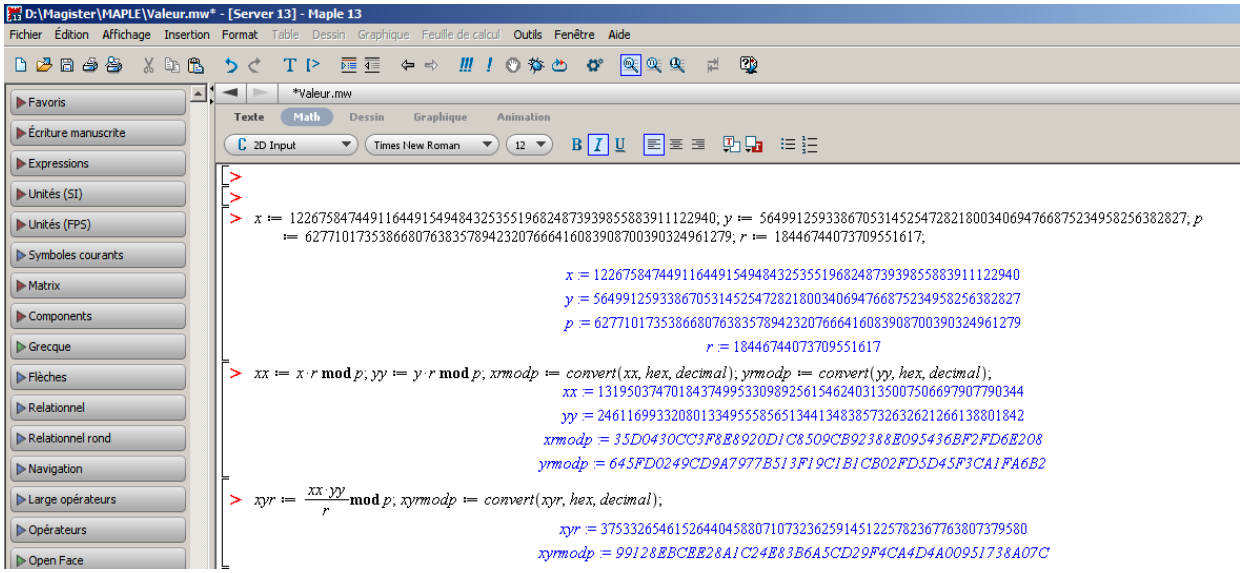


Figure 5.5 : Résultats de vérification de la multiplication modulaire avec Maple.

C)- Résultats de synthèse avec XST

La figure suivante montre le schéma RTL (Register Transfer Level) pour le module de la multiplication de Montgomery, le RTL est généré à la fin de l'étape de synthèse, il permet d'avoir un aperçu du circuit tout au début du processus d'implémentation.

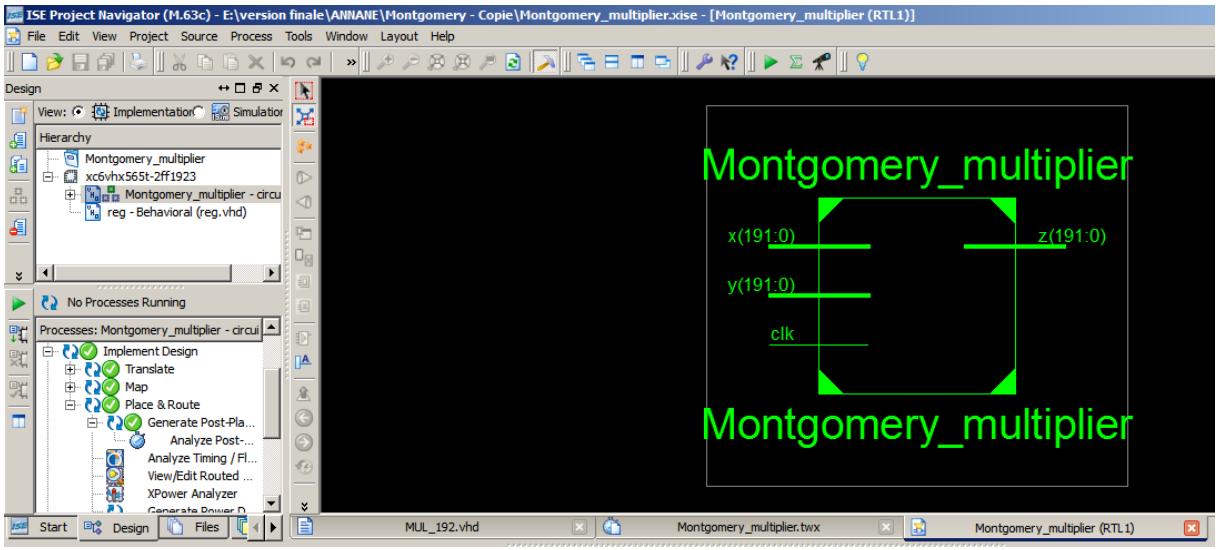


Figure 5.6 : Le schéma RTL du multiplieur de Montgomery.

D)- Résultats Timing analyser

La détermination du temps globale du multiplieur de Montgomery est faite en deux phases à l'aide de Timing analyser : en premier étape nous avons mesuré le temps écoulé d'un seul cycle à partir des deux registres X et Y jusqu'à la sortie du compresseur 4 : 2 comme montre la Figure 4.7, le résultat est montré dans la figure suivante :

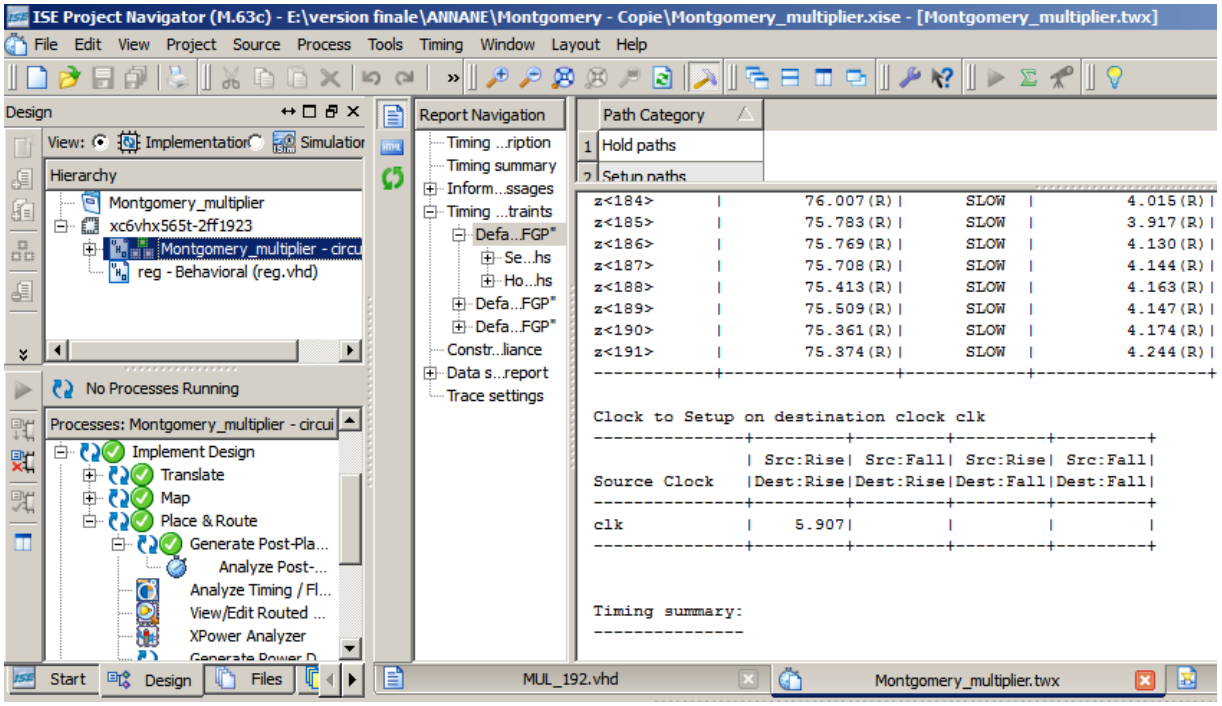


Figure 5.7 : Le temps d'un cycle de la première phase du multiplieur de Montgomery.

Le temps d'un seul cycle pour le module du multiplieur de Montgomery est 5,9 ns, donc pour 192 itérations le temps est 1133 ns. La deuxième étape est de mesurer le temps à partir des deux registres X et Y jusqu'à le registre Z comme montre la figure 4.7, le résultat obtenue est montré dans la figure 5.8. Le temps du compresseur 4 : 2 au registre Z est 2,85 ns donc le temps total pour une multiplication modulaire est 1135 ns.

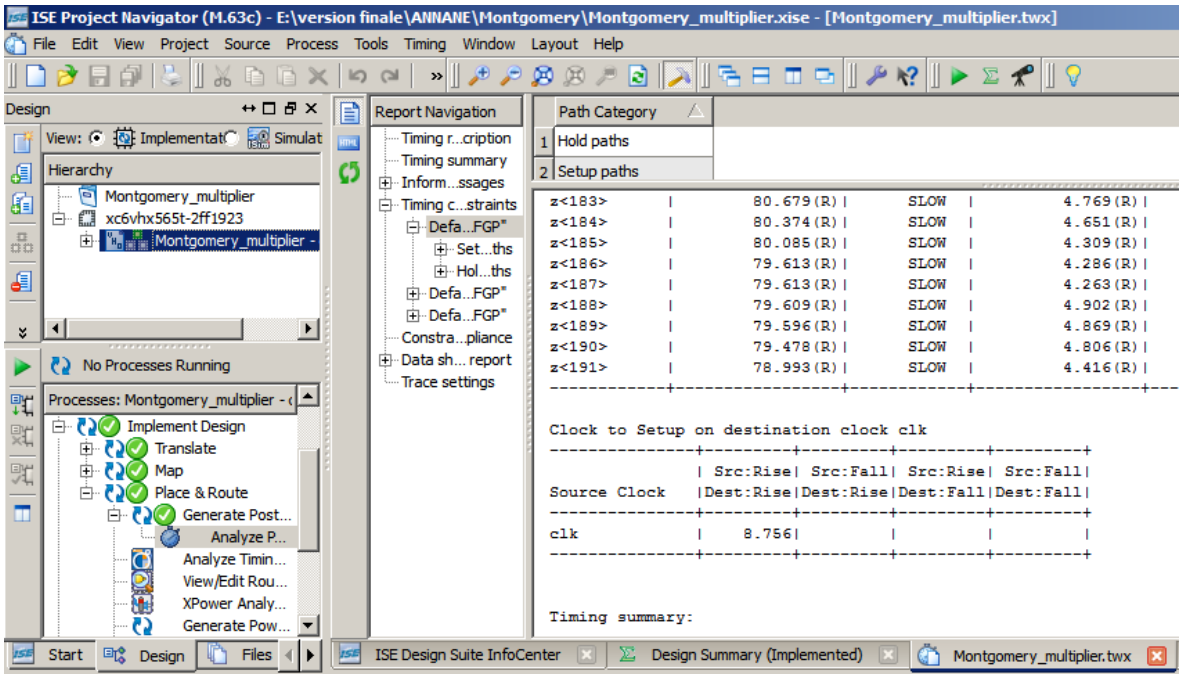


Figure 5.8 : Le temps d'un cycle globale du multiplieur de Montgomery.

5.7.2- Addition/soustraction modulaire à 192 bits

Concernant l'addition/soustraction on a utilisé les valeurs de *x* et *y* suivantes:

- *x* = 1226758474491164491549484325355196824873939855883911122940,
- *y* = 5649912593386705314525472821800340694766875234958256382827,
- *p* = 627710173538668076383578942320766641608390870039032496127 9.

A)- Résultats de simulation avec (Isim)

La figure suivante montre le résultat d'addition modulaire de *x* et *y*.

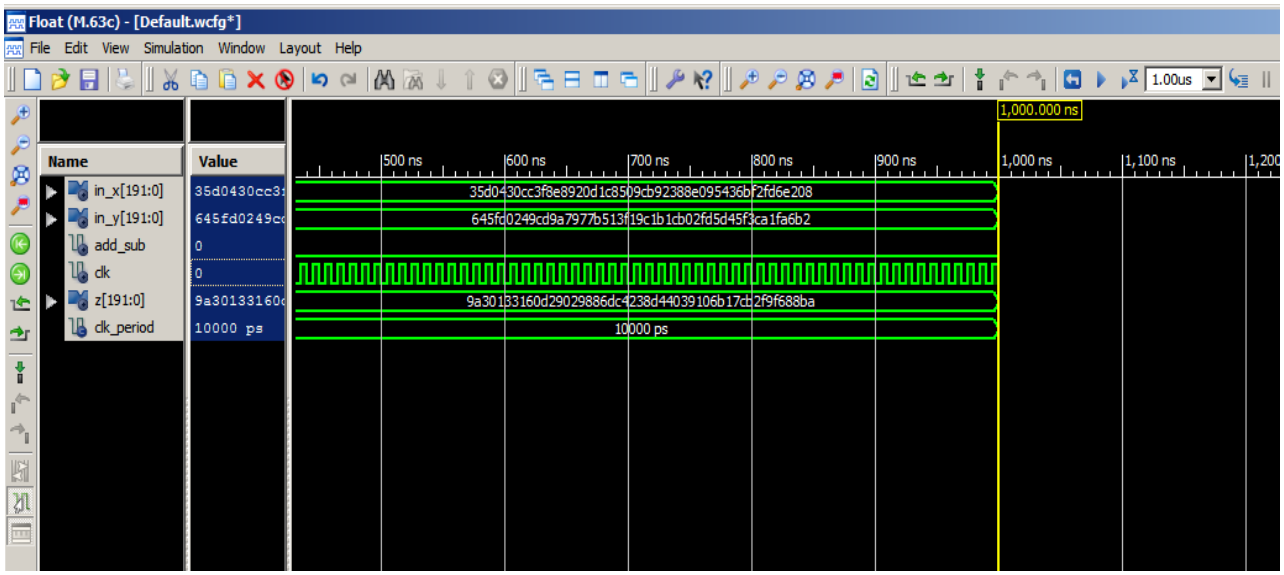


Figure 5.9 : Résultat de simulation de l'addition modulaire.

B)- Résultats de Maple

L'outil Maple est utilisé vérifié les résultats obtenue avec le simulateur Isim. Au début nous avons entrées les valeurs de *x* et *y* dans le domaine de Montgomery puis nous avons fait l'addition et la soustraction modulaires et enfin, nous avons converti le résultat en hexadécimal.

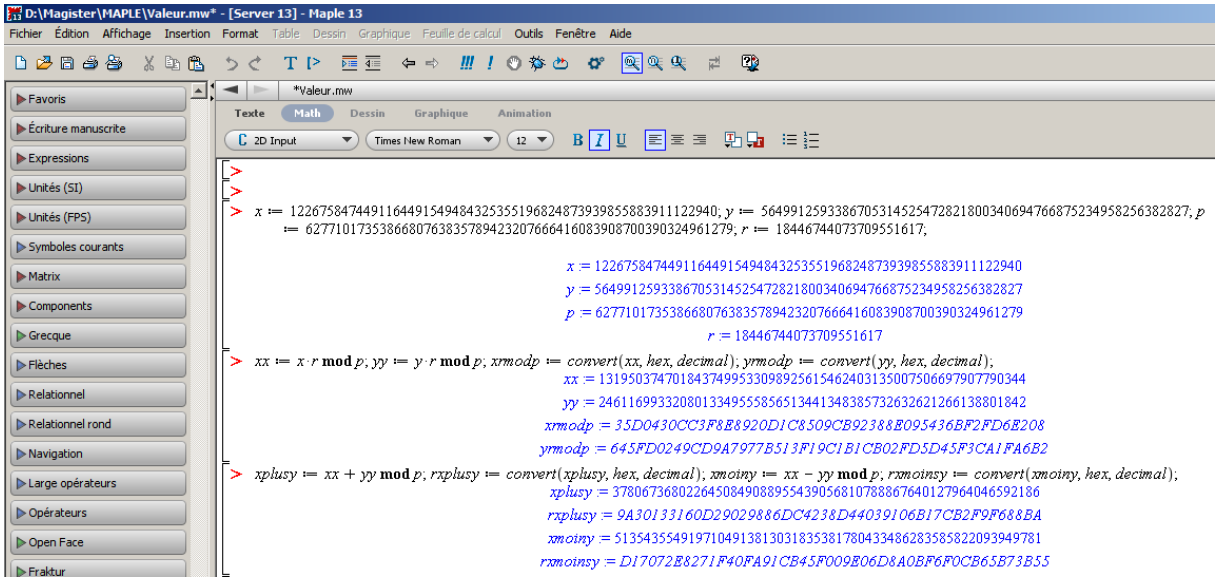


Figure 5.10 : Résultat d'addition/soustraction modulaire de *x* et *y*.

C)- Résultats de synthèse

La figure suivante montre le schéma RTL pour le module Addition/soustraction :

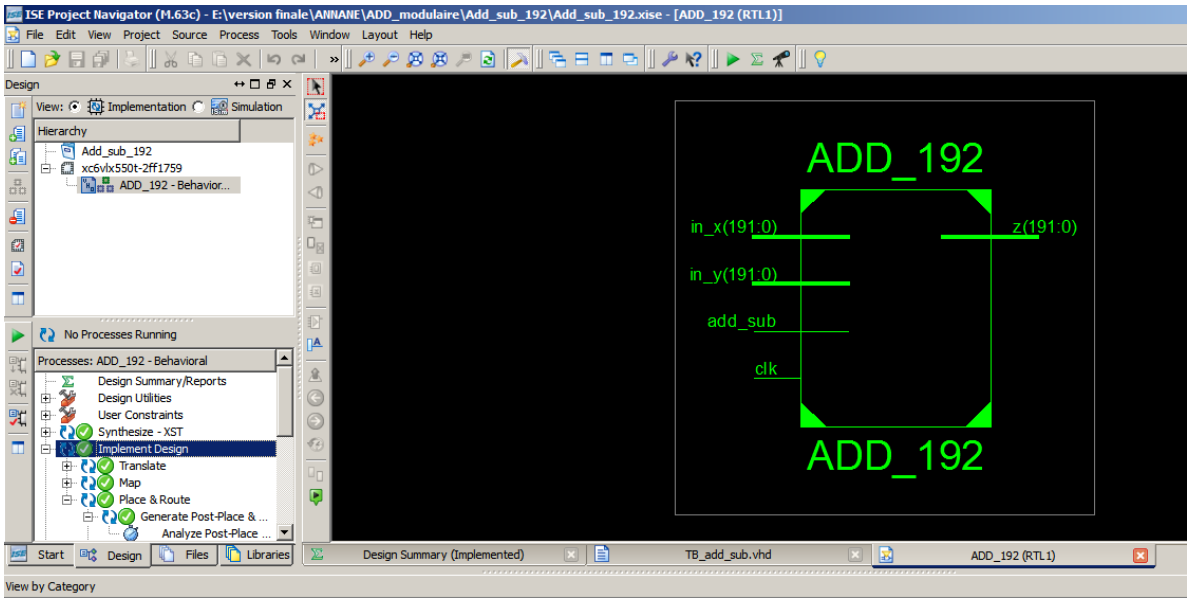


Figure 5.11 : Le schéma RTL du module d'addition/soustraction modulaire.

5.7.3- Bloc mul_add

A)- Résultats de synthèse

Le bloc mul_add effectué l'inversion modulaire, additionnement de point et le doublement d'un point. La figure suivante montre le schéma RTL pour ce bloc :

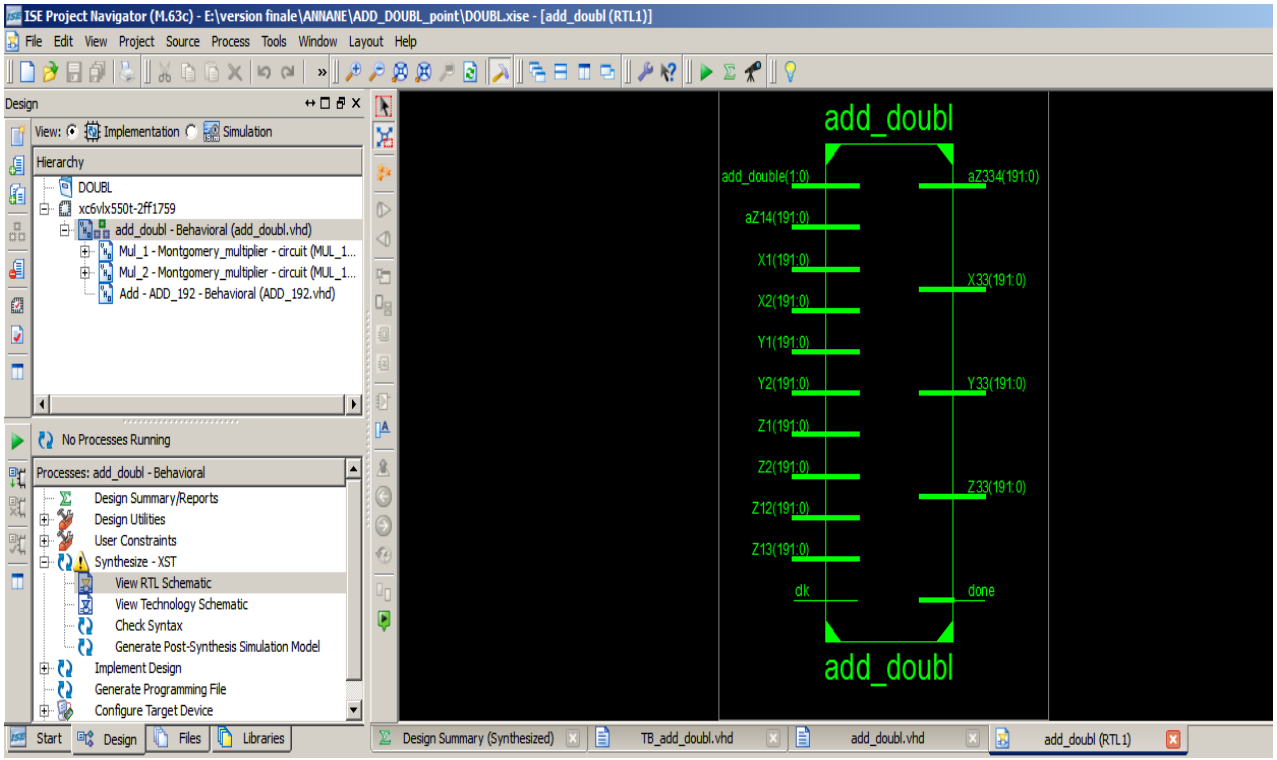


Figure 5.12 : Le schéma RTL du bloc mul_add.

B.1)- Les résultats de l'inversion modulaire

La valeur à inverser

- x = 231093120654728217851775499891519489754398939640133918049,
- r = 18446744073709551617,

Les résultats :

- x^{-1} = $(136995703159005844619970555572458408207381858508466623933 8)_{10}$,
- $x^{-1} * r \bmod p$ = $(149502340406104215217375626494964416313926066492216961473 9)_{10}$,
- $x^{-1} * r \bmod p$ = $(3CF8C5CB8E0CC367AE53275DE6C79CD1A93966614A7F2193)_{16}$,

B.1.1)- Résultats de simulation de l'inversion modulaire (Isim)

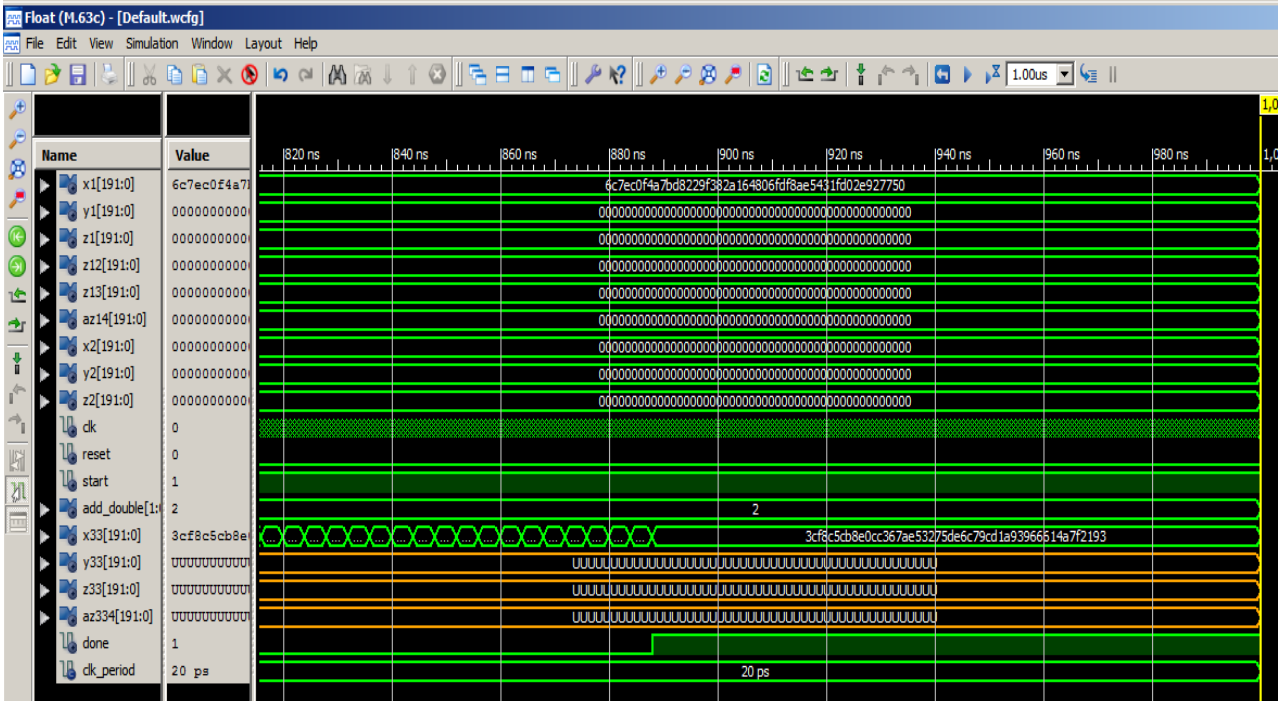


Figure 5.13 : Résultat de simulation de l'inversion modulaire.

B.1.2)- Résultats de Maple

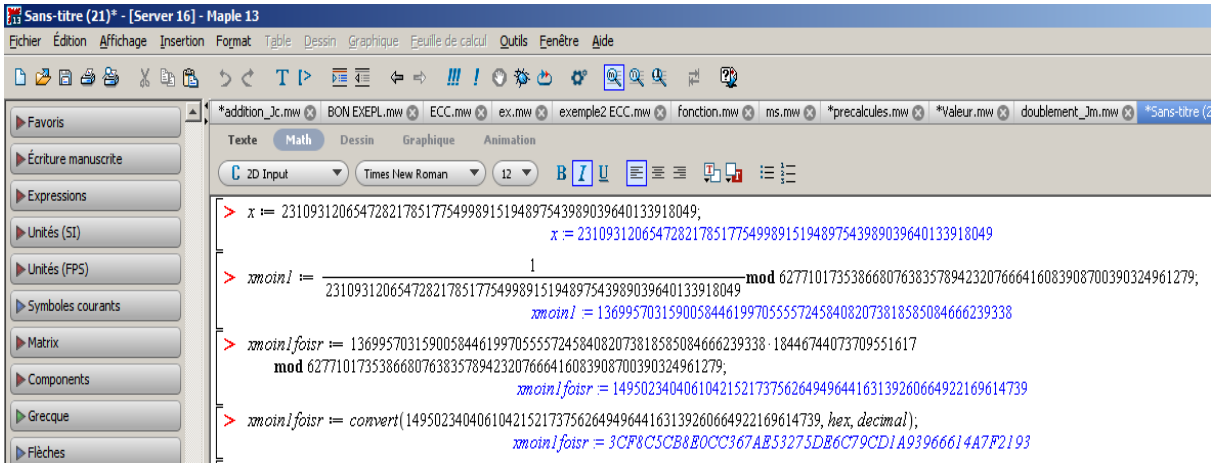


Figure 5.14 : Résultat d'inversion modulaire de *x*.

B.2)- Les résultats du doublement

On montre le doublement de point *P*, ou le point *P* est en coordonnées Affine :

- *Px* = 6020462823756886567582134805875261119166989766368846848 18,
- *Py* =1740503322936220314048575522802194103640234889273866506 41.

Les résultats en coordonnées J^m :

- X_3 = 39862333993368556624051811639032220320772072321156400682902,
- Y_3 = 29997382362783331756446554274980050370015251246033354603 69,
- Z_3 = 52061249440175047067679582775686617461222022196317255283 38,
- Z_{34}= 40274479358076817648309423306269311433186007259587790 98251.

B.2.1)- Résultat de simulation du doublement avec (Isim)

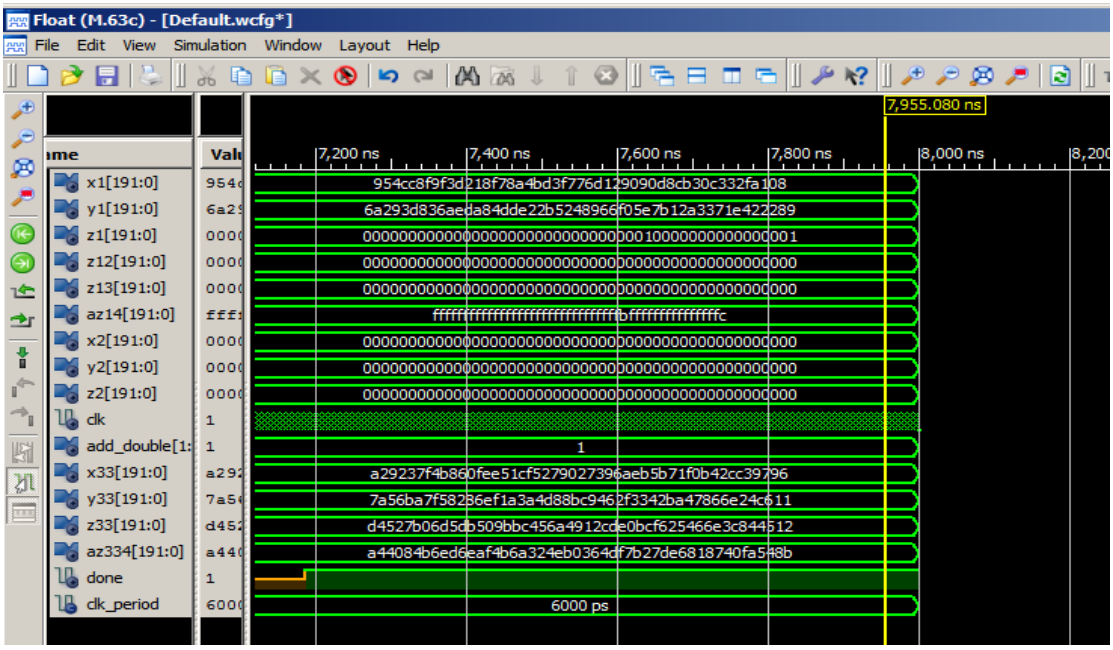


Figure 5.15 : Résultat de simulation de doublement du point *P*.

B.2.2)-Résultats de Maple

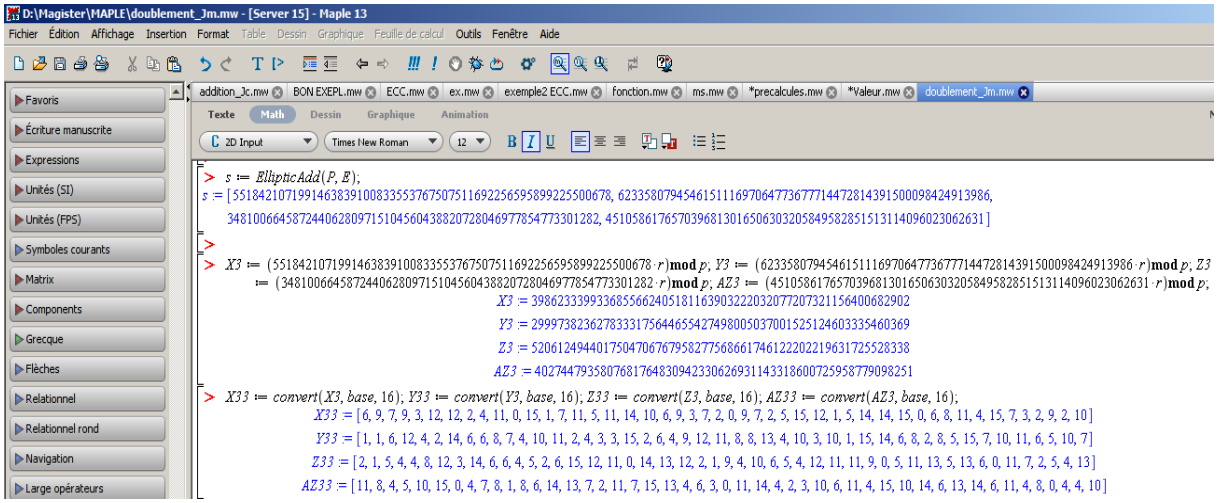


Figure 5.16 : Résultat de doublement de point *P* en coordonnée J^m.

B.3)- Résultats d'additionnement

On montre l'additionnement de deux points P et Q, où les points P et Q sont en coordonnées Affine comme suit :

- Px =2506181613215082207211844472230361433516887700046501773450,
- Py =154268489670710067923594416363118769061247460449331941213 7,
- Qx = 225551605063565352693168512114845176430125366686711569116 4,
- Qy =147353874581637132233722037998253315515728142927359229198 5.

Les résultats d'additionnement en coordonnées J^c :

- X_3 = 844559359007467347795191752219284382746623826385079798 93,
- Y_3 = 535367395831134151988541741454242436989945708784152528 2928,
- Z_3 = 156493083849970909032187825343833087652998819212401784 3596,
- Z_{34}= 302657871184376004527092557666344456145110939794064272 3427.

B.3.1)- Résultat de simulation d'additionnement (Isim)

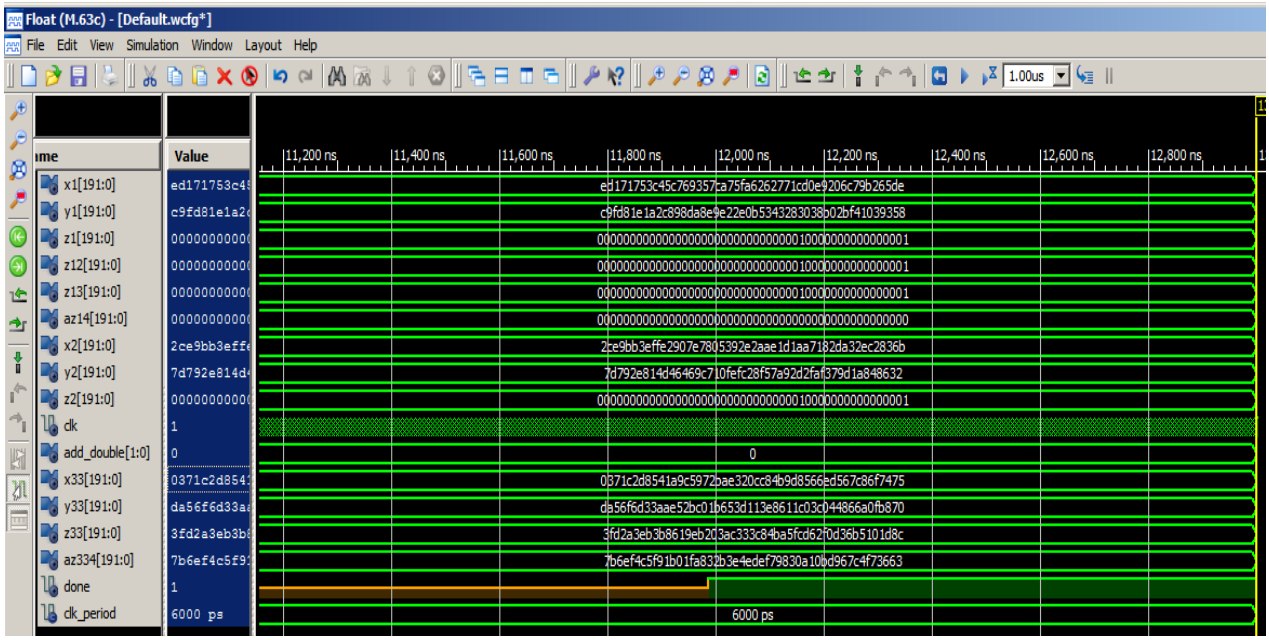


Figure 5.17 : Résultat de simulation d'additionnement de deux points P et Q.

B.3.2)-Résultats de Maple

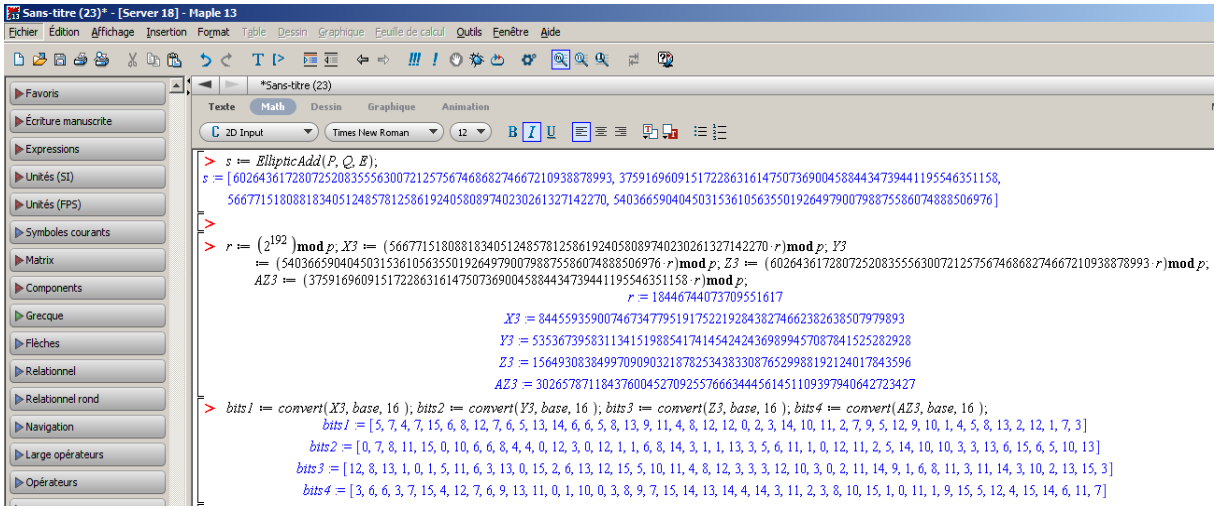


Figure 5.18 : Résultat d'additionnement de deux points *P* et *Q* en J^c.

5.7.4- Multiplication scalaire

On montre la multiplication scalaire de *k* par le point *P* où :

- *Px* = 602046282375688656758213480587526111916698976636884684818,
- *Py* = 174050332293622031404857552280219410364023488927386650641,
- *k* = 313855086769334038157761234466709104342022279161366371298.

Les résultats de la multiplication scalaire :

- *Qx* = 309217252908005351560446949338599508571189343319295840353 6,
- *Qy* = 395306231916643341390066992192807235020658491112187788382 1.

A)- Résultats de simulation

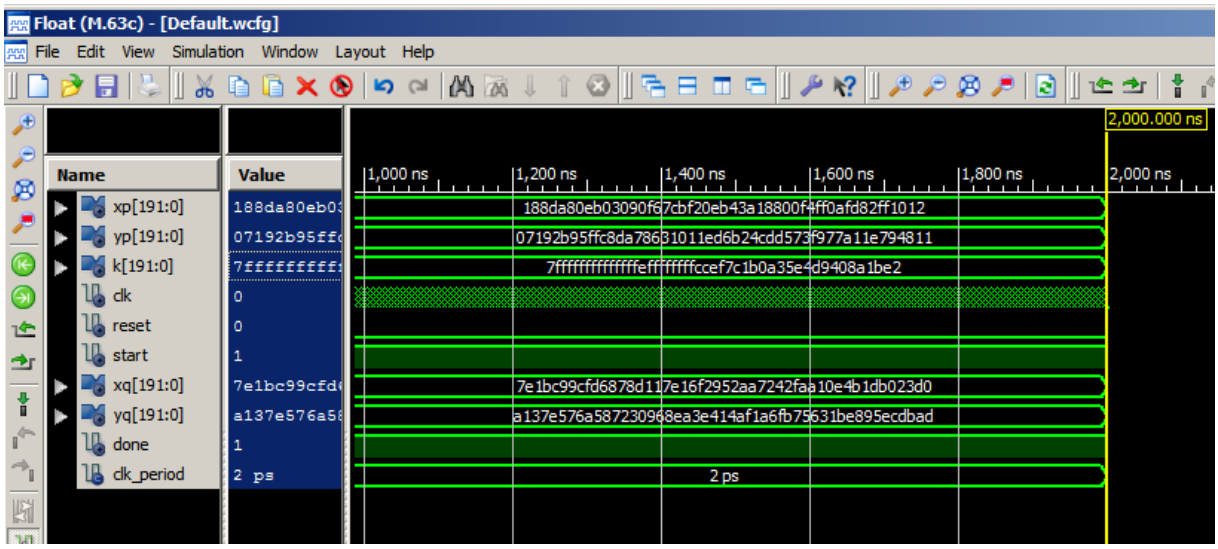


Figure 5.18 : Résultat de simulation de la multiplication scalaire de *k* par *P*.

B)- Résultat de Maple

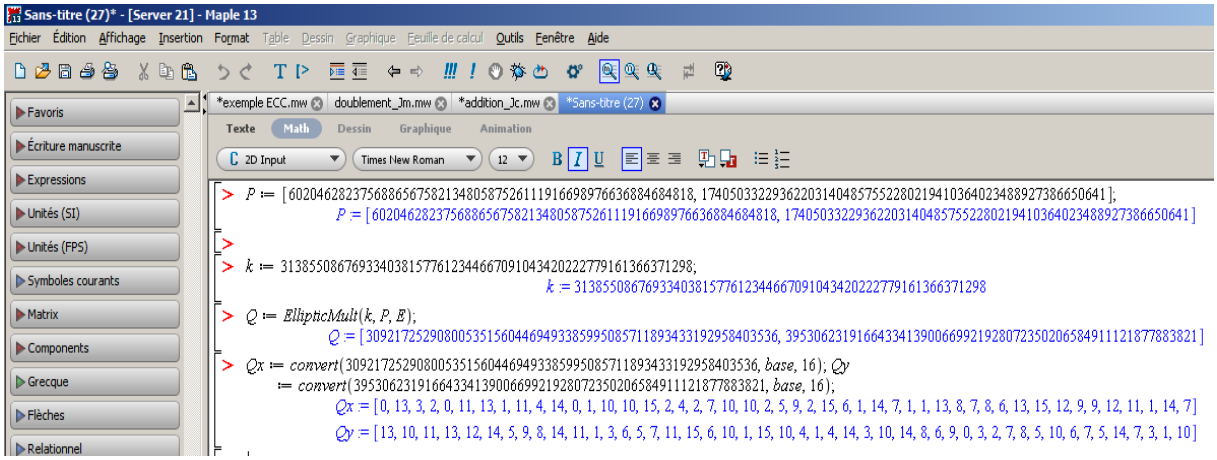


Figure 5.19 : Résultat de la multiplication scalaire de *k* par *P*.

C) Résultats de synthèse

La figure suivante montre le schéma RTL (Register Transfer Level) pour le module de la multiplication scalaire :

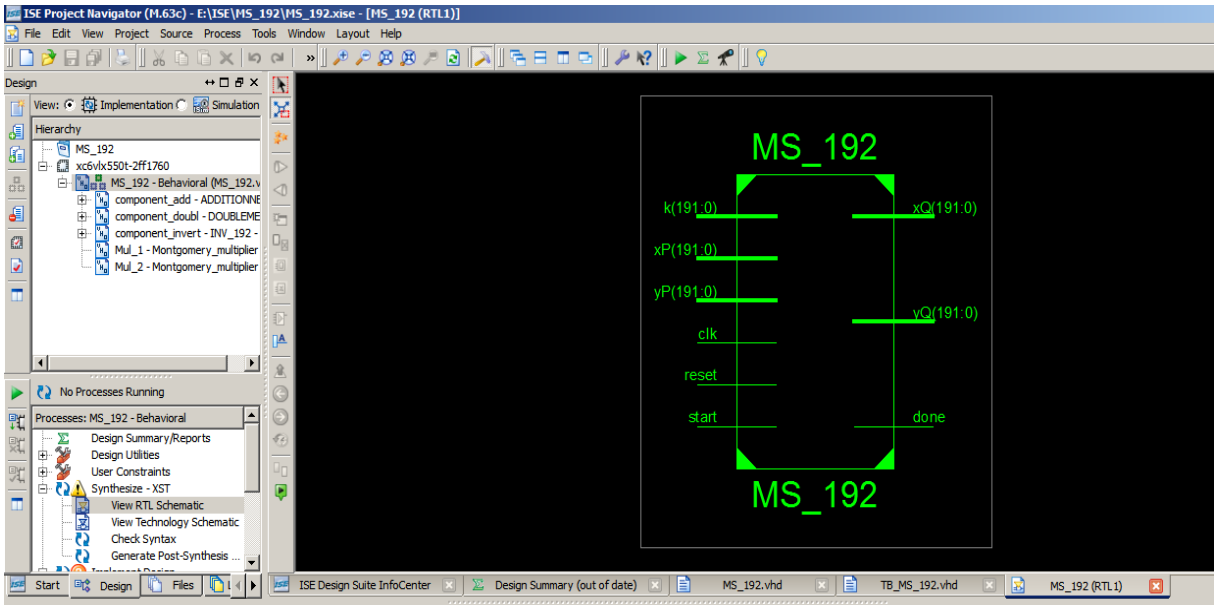


Figure 5.20 : Le schéma RTL de la multiplication scalaire.

5.7.5- Les temps globaux pour les différents modules implémentés

Le module d'inversion modulaire exécute la multiplication modulaire 192 fois donc le temps d'une inversion modulaire est 1135 multiplié par 192, le temps d'un additionnement est

le temps d'une multiplication de Montgomery multiplié par dix, et le temps d'un doublement est le temps d'une multiplication de Montgomery multiplié par six.

Le temps d'une multiplication scalaire en moyenne égale à 192 fois le temps d'un doublement plus 96 fois le temps d'un additionnement plus le temps d'une inversion modulaire plus le temps de deux multiplications modulaires pour entrer et sortir les valeurs du domaine de Montgomery.

Le tableau suivant résume les résultats des différents modules en nano seconde (ns) :

L'opération	Le temps
Multiplication de Montgomery	1135
Inversion modulaire	217920
Additionnement	11350
Doublement	6810
Multiplication scalaire	2440250

Tableau 5.1 : Les temps moyens des différents modules.

5.8- Comparaison à d'autres travaux

Il est à noter qu'il est difficile de faire des comparaisons adéquates de notre travail avec d'autres travaux, car les supports d'implémentation et les outils de synthèse diffèrent, mais jouent un rôle important dans les performances des circuits résultants.

Les performances temporelles obtenues en comparaison avec d'autres travaux sont jugé satisfaisant.

	taille (bits)	Délai	Circuit
Notre	192	2.44 ms	XC6VHX565T
[35]	160	14.414 ms	V1000E-BG580-8
[36]	192	3 ms	XCV1000E-BG680
[37]	256	3,86 ms	XC2VP125-7-ff1696
[38]	256	0.624 ms	Virtex-4

Tableau 5.2 : Comparaison avec d'autres travaux.

5.9- Conclusion

Dans ce dernier chapitre nous avons présenté les caractéristiques du circuit FPGA Virtex-6 qu'on a utilisé pour implémenter les différents modules de la multiplication scalaire.

Puis, nous avons présenté l'outil ISE de Xilinx qu'on a utilisé pour l'implémentation ; en citons les différentes étapes : la simulation, la synthèse et l'implémentation.

Nous avons montré les résultats de simulation à l'aide de simulateur Isim en validant ces résultats avec l'outil Maple et les résultats de synthèse avec XST en montrant les schémas RTL pour les différents modules conçoivent.

Enfin, nous avons comparé le temps obtenue pour la multiplication scalaire à l'aide de timing analyser avec le temps des d'autres travaux.

Conclusion générale

L'objectif de notre projet est la proposition d'une architecture hardware optimisé de l'opération de la multiplication scalaire en terme complexité calculatoire et en termes de temps d'exécution ; ensuite la conception et l'implémentation de cette opération sur un circuit FPGA.

L'opération de la multiplication scalaire représente environ de 80% du temps de calcul pour les crypto systèmes basé sur les courbes elliptiques.

De ce fait, nous avons entamé notre projet par une étude des protocoles cryptographiques, en particulier le crypto système à clé publique ECC, ensuite nous avons étudié la multiplication scalaire n'est autre qu'une suite de doublements et additionnements, chacune de ces opérations nécessite un ensemble d'opérations arithmétiques modulaires : addition/soustraction modulaires, multiplication modulaire et l'inversion modulaire.

Au niveau algorithmique, nous avons conclu que l'utilisation des coordonnées mixtes : les coordonnées Jacobienne modifié pour le doublement et les coordonnées Jacobienne Chodnovsky pour l'additionnement donne un meilleur coût en terme de complexité calculatoire, ces coordonnées permet aussi d'éviter l'inversion modulaire qui très couteuse.

Au niveau d'architecture, nous avons vu que l'utilisation de deux multiplieurs en parallèle donne un bon compromis entre le temps d'exécution et la surface occupé sur le circuit FPGA ; et nous avons utilisé un compresseur 4 : 2 dans le multiplieur de Montgomery qui permet d'éliminer la propagation des carrés et par conséquent un bon performance pour le multiplieur de Montgomery.

Au niveau hardware, notre architecture est implémentée dans un circuit FPGA de Xilinx pour des raisons de performance qu'offrent ces circuits. La conception des différents modules sont fait en langage VHDL à l'aide de l'outil ISE.

Nous avons montré les résultats de simulation, synthèse et d'implémentation de notre architecture pour la multiplication scalaire.

Enfin, une comparaison avec d'autres travaux est faite concernant le temps obtenue pour la multiplication scalaire.

Bibliographie

[1] J. Daemen, V. Rijmen « *AES Proposal: Rijmen* », Document version 2, (1999).

[2] M. Matsui, « *Linear cryptanalysis method for DES cipher* », Advances in Cryptology, Proceedings Eurocrypt'93, Ed : Springer-Verlag, 1994, pp. 386-397.

[3] Renaud Dumont, « *Cryptographie et Sécurité informatique* », Université de Liège Faculté des Sciences Appliquées, pp.1 - 254, 2009 – 2010.

[4] Jingyang Gao, Hai Cheng, Ziheng Yang, Qun Ding , « *The Research and Design of Embed RSA Encryption Algorithm Network Encryption Card Driver* », in 2013 International Conference on Sensor Network Security Technology and Privacy Communication System (SNS & PCS), pp.83-87, May 18-19, 2013.

[5]V.S.Miller, « *Use of elliptic curves in cryptography* », Advances in CryptologyCRYPTO'85, Lecture Notes in Computer Science, vol 218, pp. 417-426.

[6]N.Koblitz, « *Elliptic curve cryptosystems* », Mathematics of Computation, vol 48 n°177, pp. 203-209, 1987.

[7] M. Joye, «*Introduction élémentaire à la théorie des courbes elliptiques* », Technical report, CG-1995/1, UCL Crypto Group, Louvain-la-Neuve, 1995.

[8]D. Hankerson, A. Menezes, and S. Vanstone, « *Guide to Elliptic Curve Cryptography* », Springer, 2004.

[9] Ion TUTANESCU, Constantin ANTON, Laurentiu IONESCU, Daniel CARAGATA, «*Elliptic Curves Cryptosystems Approaches*», in International Conference on Information Society (i-Society 2012), pp.357-362, 2012.

[10] Rakotondraina T.E, Randimbindrainibe F, Randriamitantsoa P.A, «*Optimisation des algorithmes de calcul cryptographique basés sur les Courbes Elliptiques* », MADA-ETI, Vol.1, 2013.

[11] IEEE Std 1363-2000, « IEEE Standard Specifications for Public-Key Cryptography », 30 January 2000.

[12] R. Crandall, « Method and apparatus for public key exchange in a cryptographic system », U.S. Patent number 5159632, 27 oct 1992.

[13] J. Solinas, « Generalized Mersenne numbers », Research Report CORR-99-39, Center for Applied Cryptographic Research, University of Waterloo, Waterloo, Canada, 1999.

[14] Don Johnson, Alfred Menezes, Scott Vanstone, « *The Elliptic Curve Digital Signature Algorithm (ECDSA)*», Certicom Research, Canada.

[15] « Recommended Elliptic Curves For Federal Government USE », July 1999.

[16] Digital Signature Standard (DSS), « Federal Information Processing Standards Publication », National Institute of Standards and Technology, 2000.

[17]Xiaoqiang Zhang, Guiliang Zhu, Weiping Wang, Mengmeng Wang, « *Design and Realization of Elliptic Curve Cryptosystem* », international Symposium on instrumentation et Measurement, IEEE, pp.302-305, 2012.

[18] Antonio Cortina Reyes, Ana Karina Vega Castillo, Miguel Morales-Sandoval, Arturo Diaz-Pérez, « *A Performance Comparison of Elliptic Curve Scalar Multiplication Algorithms on Smartphones* », IEEE, pp. 114-119, 2013.

[19] H. Brar, R. Kaur, « *Design and implementation of block method for computing NAF* », International Journal of Computer Applications, vol. 20, pp. 37–41, April 2011.

[20] K. Okeya and T. Takagi, « *The width-w NAF method provides small memory and fast elliptic scalar multiplications secure against side channel attacks* », The cryptographers' track, CT-RSA'03, (Berlin, Heidelberg), pp. 328–343, Springer-Verlag, 2003.

[21] Ravi Kishore Kodali, Harpreet Singh Budwal, « *High Performance Scalar Multiplication for ECC* », Coimbatore, INDIA, Jan. 04 – 06, 2013.

[22] Ravi Kishore Kodali, Srikrishna Karanam, Kashyapkumar Patel and Harpreet Singh Budwal, « *Fast Elliptic Curve Point Multiplication for WSNs* », IEEE Tencon – Spring, pp.194-198, 2013.

[23] Nicolas Sklavos, Xinmiao Zhang, *« Wireless Security and Cryptography»*, Specifications and Implementations, chapitre 5 : pp. 155–160, 2006.

[24] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono, « *Efficient Elliptic Curve Exponentiation Using Mixed Coordinates* », Springer-Verlag Berlin Heidelberg, pp. 51-65, 1998.

[25] Pritam Gajkumar Shah, « *Investigating Effects of Co-ordinate System on Execution Time of Elliptical Curve Protocol in Wireless Sensor Networks* »,IEEE International Conference on Future Communication Networks, *2012.*

[26] Jorge Guajardo, Tim Güneysu, Sandeep S. Kumar ·Christof Paar, Jan Pelzl, «*Efficient Hardware Implementation of Finite Fields with Applications to Cryptography* », Spring , 26 September 2006.

[27] P.Montgomery, « *Modular Multiplication without Trial Division*», Mathematics of Computation, Vol. 44, pp.519-521, 1985.

[28] N. Koblitz, « *A Course in Number Theory and Cryptography* », vol. 114 of *Graduate text in mathematics.* Springer-Verlag, Berlin, Germany, second edition, 1994.

[29] Colin D. Walter, « *Right-to-Left or Left-to-Right Exponentiation?* », First International Workshop on Constructive Side-Channel Analysis and Secure Design, pp. 40-46, 2010.

[30] www.xilinx.com, « *Virtex-6 Family Overview* », 20 Août 2015.

[31] www.xilinx.com, « *Virtex-6 FPGA Configurable Logic Block User Guide*», 3 février, 2012.

[32] www.xilinx.com, « *Virtex-6 FPGA SelectIO Resources* », 7 Novembre, 2014.

[33] www.xilinx.com, « *Synthesis and Simulation Design Guide* », 18 décembre, 2012.

[34] SHNEIDER, THIESSY, « *VHDL : Méthodologie de design et techniques avancées ; guide pratique du concepteur* », Editions DUNOD, 2001.

[35] S. B. Ors, L. Batina, and B. Preneel, « *Hardware implementation of elliptic curve processor over GF(p)* », 14th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'03), pp. 433–443, Jun. 2003.

[36] G. Orlando and C. Paar, « *A Scalable GF(p) Elliptic Curve Processor Architecture for Programmable Hardware* », LNCS 2162, pp. 356–371, 2001.

[37] Ciaran J. McIvor, Máire McLoone, John V. McCanny, « *Hardware Elliptic Curve Cryptographic Processor Over GF(p)*», IEEE transactions on circuits and systems—i: regular papers, vol. 53, no. 9, September 2006.

[38] Osama Al-Khaleel, Chris Papachristou, Francis Wolff, Kiamal Pekmestzi, « *An Elliptic Curve Cryptosystem Design Based on FPGA Pipeline Folding* », 13th IEEE Intenationl On-Line Testing Symposium, 2007.

Dans cette annexe nous avons présenté l'ensemble des procédures Maple utilisées pour le chiffrement et le déchiffrement basant sur les courbes elliptiques.

1- Cette procédure permet le calcul de la multiplication scalaire qui l'opération plus coûteuse dans le chiffrement /déchiffrement avec ECC.

On appelle cette procédure **Doublement-et-Addition de droite à gauche**, son principe de fonctionnement est de convertir le k en binaire et de le parcourir de droite à gauche, si le bit en cours = 1 alors on une addition et un doublement sinon on a seulement un doublement. Cette procédure à comme entrer un entier k , un point générateur P et paramètres de la courbe (a, b, p) et comme sortie un $Q = k \cdot P$.

```
---------------------------------------------------------------------------------------------
EllipticMult := proc (k::integer, P::{list(integer), identical(0)}, E::(list(integer)))
// Déclaration des variables locales.
local t, R, S, bits, l, i;
//vérifié si le point P est dans la courbe.
if IsEllipticPoint(P, E) = false then
        error "the point is not on the curve"
end if;
// Affectation de k au variable t et P au R
if k = 0 or P = 0 then return 0 end if;
if  k>0 then t := k; R := P
        else t := -k; R := EllipticOppositePoint(P, E)
end if;
S := 0;
bits := convert (t, base, 2); // convertir t en binaire
l := nops (bits); // compter le nombre de bits de t
for i to l do if bits [i] = 1 //parcourir les bit de t
then S := EllipticAdd(S, R, E) end if; //appel de procédure d'additionnement
R := EllipticAdd(R, R, E) end do; // appel de procédure doublement
S end proc;
---------------------------------------------------------------------------------------------
```

2- La procédure « *EllipticAdd* » permet le calcul l'additionnement deux point P et Q et de calculer le doublement d'un point P.

```
---------------------------------------------------------------------------------------------

// Les entrées de la procédure deux points P, Q et la courbe E.
 EllipticAdd := proc (P::{list, identical(0)}, Q::{list, identical(0)}, E::(list(integer)))
// déclaration des variables locales.
local p, x1, y1, x2, y2, a, b, m, x3, y3;
//vérifié si les deux points P, Q sont dans la courbe.
if not andmap(proc (x) options operator, arrow; IsEllipticPoint(x, E) end proc, [P, Q]) then
error "both points must be on the curve" end if;
```

```
if P = 0 then return Q // si P est le point à l'infini alors P+Q = Q
elif Q = 0 then return P end if; // si P est le point à l'infini alors P+Q = Q
// affectation des trois paramètres de la courbe au variables a, b, p respectivement.
p := E[3];
a := E[1];
b := E[2];
x1 := `mod`(P[1], p); // associé x1 au premier coordonnées de P
y1 := `mod`(P[2], p); // associé y1 au 2eme coordonnées de P
x2 := `mod`(Q[1], p); // associé x2 au premier coordonnées de Q
y2 := `mod`(Q[2], p); // associé y2 au 2eme coordonnées de Q
if x1 = x2 and y1 = `mod`(-y2, p)  then return 0  end if;
if P = Q  then m := `mod`((1/2)*(3*x1^2+a)/y1, p) // si P = Q alors on un doublement de point P.
else m := `mod`((y2-y1)/(x2-x1), p) // P ≠ Q alors on additionnement de P et Q.
end if;
x3 := `mod`(m^2-x1-x2, p);
y3 := `mod`(m*(x1-x3)-y1, p);
[x3, y3]
end proc;
```

3- Cette procédure permet de vérifié si un tel triplet (a, b, p) est une courbe elliptique pour cela on calcule le déterminant qui égale $(4a^3 + 27b^2) \bmod p$ qui être différent de zéro, sinon la procédure renvoie erreur.

```
> Ellipticcurve := proc (a::{integer, string}, b::{integer, string}, p::{posint})
local q, A, B;
q := p;
if isprime (q) = false or q < 5 then error "%1 is not a prime>3", q
end if;
A := `mod`(a, q);
B := `mod`(b, q);
if `mod`(4*A^3+27*B^2, q) = 0 then error "singular curve"
                                  else [A, B, q]
end if
end proc;
```

4- La procédure « *IsEllipticPoint*» tester un point P s'il est dans la courbe où pas, cette procédure renvoie *true* si un point appartient à la courbe sinon renvoie *false*.

```
> IsEllipticPoint := proc (P, E)
evalb (P = 0 or `mod`(P[2]^2-P[1]^3-E[1]*P[1]-E[2], E[3]) = 0)
end proc;
```

5- La procédure « *EllipticOppositePoint* » permet le calcule l'opposer d'un point *P* (x, y), on a l'opposer d'un point *P* (x, y) est –*P* (x, -y).

```
> EllipticOppositePoint := proc (P::{list(integer), identical(0)}, E::(list(integer)))
local p; p := E[3];
if not IsEllipticPoint(P, E) then
                                    error "the point is not on the curve"
                                elif P = 0
                                    then 0 else [P[1], `mod`(-P[2], p)]
end if
end proc;
```

6- La procédure suivante calcule l'ordre d'un point *P* de la courbe.

```
EllipticPointOrder := proc (P::{list(integer), identical(0)}, E::(list(integer)), ord::posint, { orddivs :=
sort(convert(numtheory:-divisors(ord), list)) })
local d; option remember;
if IsEllipticPoint(P, E) = false then error "the point is not on the curve" end if;
for d in orddivs while EllipticMult(d, P, E) <> 0 do NULL end do; d
end proc;
```

Il reste de fixer la courbe *E* avec ces paramètres : *a*, *b*, *p*, *G*, *n*.

Exemple

On est choisi la courbe elliptique *P-192* recommandée par le NIST (National Institute of Standards and Technology) où ces paramètres sont :

- a = -3,
- b = 2455155546008943817740293915197451784769108058161191238065,
- p = 6277101735386680763835789423207666416083908700390324961279.
- G = [205276475889337534449667902599269120263560324975669085786 2, 3235495846547773372963571948817489147168146215941690951212].
- n = 6277101735386680763835789423176059013767194773182842284081.

Génération des clés

n_B = 313855086769334038157761234466709104342022279161366371298 est généré au hasard.

$P_B = n_B \times G =$ [22831486482987723208776368674205736380420390970903497852 90, 24021183579145250852693002433975704085470378127692538166 60]

Chiffrement

Soit M = 12345678902345678901345678901278 65 le message à crypter.

- n_A = 25107288743164549712664898108403761549678639711577973519 est généré au hasard.
- $P_A = n_A \times G$ = [38425475054487634945670838086115086067625540186376849495 41, 38536239568594056765843200034696862970299730862230627669 33]
- P_s (x_s, y_s) = $n_A \times P_B$ = [26602928902930800577258755708796220008070171561490919237 92, 53398160475159507933015713824012913706037027834325567215 91]
- c_1= (M $*$ x_s) mod p = 34095657405976040646556294980377985125907704320574623476 81 (le chiffrement).

Déchiffrement

Pour chaque bloc l'entité **B** devra calculer :

- P_s (x_s, y_s) = ($n_B \times P_A$) = [26602928902930800577258755708796220008070171561490919237 92, 53398160475159507933015713824012913706037027834325567215 91]
- Le multiplicatif inverse x_s^{-1} = 40242993903164277382085710443776835405175109526561869135 57
- M = ($c_1 * x_s^{-1}$) mod p = 12345678902345678901345678901278 65