RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

UNIVERSITÉ DE TIZI-OUZOU
DEPARTEMENT D'INFORMATIQUE

Thème :

Implémentation d'un crypto système ECC sur circuit FPGA à base du processeur ARM.

Projet mémoire de fin du cycle mastère.

Réalisé & soutenu publiquement par :

M^lle ISSAD Lamia.

Le : ….. / ….. / …..…..

En vue de l'obtention du

« **Grade de Master 2 en informatique, option ''Réseaux, Mobilité & Systèmes Embarqués''** ».

Sous la direction de :

M^me BENSAID Samia née BELLATAF
&
M^r ISSAD Mohammed
&
M^r BENLEMOU Mohammed.

Devant le jury composé de :

- **Président :** .....
- **Encadreur :**.....
- **Promotrice :** .....
- **Examinateur(trice) :**.....

2016/2017

À la mémoire de ma grand-mère Fadhma,

paix à son âme.

Remerciements

De prime abord, je tiens à remercier M^{me} S. BENSAID ainsi que M^{r} M.ISSAD, mes encadreurs, de m'avoir dirigée tout au long de ce projet.

Mes profonds remerciements à toute l'équipe de la division ASM du CDTA, spécialement M^{r} BENLEMOU Mohammed.

Que les membres du jury trouvent ici mes remerciements les plus vifs pour avoir accepté d'honorer par leurs jugements ce modeste travail.

À tous les enseignants qui ont assuré ma formation durant mon parcours universitaire, pour l'ensemble des connaissances qu'ils m'ont transmis ; je souhaite qu'en ces quelques mots ils trouvent l'expression de mon infinie reconnaissance.

Je souhaite également exprimer ma profonde gratitude à tous ceux et celles qui de près ou de loin ont apporté leur précieuse aide dans la réalisation du présent travail.

Table des matières

Chapitre 2 - Systèmes Embarqués

Chapitre 3 - Implémentation de la plateforme de chiffrement/Déchiffrement

Chapitre 4 - Résultats d'implémentation et présentation de l'IHM

Liste des acronymes

AMBA	Advanced Microcontroller Bus Architecture
APU	Application Processing Unit
AXI	Advanced eXtensible Interface
CAN	Controller Area Network
CLB	Configurable Logic Block
DDR	Double Data Rate
DSP	Digital Signal Processing
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete Logarithm Problem
EPP	Extensible Processing Platform
FF	Flip-flop
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GPIO	General Purpose Input /Output
IM	Inverse Modulaire
IOB	Input / Output Block
IOP	Input/output Peripherals
LPDDR	Low Power Double Data Rate
LUT	Lookup Table
MAC	Media Access Control
MIO	Multiplexed Input / Output
MM	Multiplication Modulaire
MMM	Multiplication Modulaire de Montgomery
MMU	Memory Management Unit
OCM	On Chip Memory
PL	Programmable Logic
PS	Processing System
PSoC	Programmable System on Chip
QSPI	Quad Serial Peripheral Interface
RAM	Random Access Memory

ROM	Read Only Memory
RSA	Rivest-Shamir-Adleman
SCU	Snoop Control Unit
SD	Secure Digital
SDIO	Secure Digital Input Output
SDK	Software Design Kit
SoC	System on Chip
SPI	Shared Peripherals Interrupts
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus

Liste des figures

Liste des tableaux

Introduction Générale

Autrefois réservée au domaine militaire ou politique, la cryptographie est devenue depuis quelques décennies si indispensable, cachée derrière tous nos gestes quotidiens tels que l'e-paiement, le courrier électronique, le téléphone mobile, internet…etc. Cependant, ce que l'on nomme cryptographie classique ou encore cryptographie à clé secrète atteint ses limites. Le paradoxe étant que pour pouvoir échanger des informations secrètement, les deux parties doivent déjà partager un secret en l'occurrence la clé de chiffrement. S'il parait simple à résoudre dans le cadre de communications entre deux personnes, le problème d'échange de clés prend une toute autre dimension à l'heure d'internet et de son milliard d'utilisateurs.

Comme issue à ce problème, le principe de la cryptographie à clé publique fut proposé en 1975 par Diffie et Hellman [1]. Le premier protocole effectif fut le RSA (Rivest Shamir Adleman) présenté en 1977, principalement basé sur le problème de la factorisation des grands entiers. RSA est encore aujourd'hui la primitive la plus utilisée en cryptographie. Cependant, les nombreux progrès effectués dans le domaine de la factorisation font que la taille des clés RSA augmente plus vite que ne le requiert l'augmentation de la puissance des ordinateurs. C'est l'une des raisons pour lesquelles la cryptographie basée sur les courbes elliptiques (ECC) connaît un tel intérêt depuis son introduction par Miller et Koblitz en 1987. Reposant sur le problème du logarithme discret, ECC requiert, à un niveau de sécurité équivalent, 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 protocole ECC repose essentiellement sur l'exécution d'une multiplication d'un scalaire k par un point P. A un bas niveau d'abstraction, le calcul de $k \times P$ est effectué par une combinaison des quatre opérations de base à savoir, l'addition modulaire, la soustraction modulaire, la multiplication modulaire et l'inverse modulaire.

Dans ce projet de fin d'étude, il est question d'étudier la nature et la complexité calculatoire d'un crypto système ECC. On s'intéresse plus particulièrement à ses

opérations arithmétiques de base et à son implémentation sur circuit FPGA de la famille Zynq. La plateforme ciblée est du type PSoC (Programmable System on Chip) à base du processeur embarqué ARM.

Nous présenterons ce travail en quatre chapitres. Le premier chapitre mettra l'accent sur les concepts fondamentaux de la cryptographie à base de courbes elliptiques et de ses besoins arithmétiques. Le deuxième chapitre sera consacré à la présentation de l'architecture des systèmes embarqués. Le troisième chapitre sera dédié à la présentation de la plateforme proposée pour la réalisation du crypto système ECC. Le quatrième chapitre nous permettra de présenter les résultats d'implémentation et l'IHM développée. Une conclusion vient en dernier lieu pour clôturer le travail ainsi accompli en proposant quelques perspectives pour une éventuelle continuité de ce travail dans le futur.

Chapitre 1

Généralités sur la Cryptographie ECC

1.1 Introduction

De nos jours, face au développement des réseaux informatiques et à la quantité des données confidentielles qui ne cessent de croitre, la sécurité de l'information transmise devient une exigence incontournable. Pour ce faire, la cryptographie se trouve être un outil efficace permettant de réponde à cette contrainte.

Ce chapitre est consacré à la présentation des généralités sur la cryptographie. Une grande partie est réservée au protocole de cryptographie que nous avons utilisé, en l'occurrence la cryptographie à base de courbes elliptiques. On s'intéressera à ses notions de base et ses besoins en arithmétique.

1.2 La cryptographie

La cryptographie est l'art de rendre inintelligible, de crypter, de coder, un message pour ceux qui ne sont pas habilités à en prendre connaissance [1].

La cryptographie est l'art et la science de garder le secret de messages. Cette dernière est une des disciplines de la cryptologie s'attachant à protéger des messages et assurant confidentialité, intégrité, authentification et non répudiation en s'aidant de secrets ou clés [2].

- **Confidentialité**

La confidentialité est la propriété qui assure que l'information est rendu inintelligible aux individus, entités, et processus non autorisés.

- **Intégrité**

L'intégrité permet de vérifier qu'une donnée n'a pas été modifiée par une entité tierce (accidentellement ou intentionnellement).

- **Authentification**

C'est la propriété qui permet de vérifier que la source de données est bien l'identité prétendue.

- **Non répudiation**

Elle assure que l'émetteur du message ne pourra pas nier avoir émis le message dans le futur.

1.3 Notions de chiffrement et de déchiffrement [2]

Le processus de transformation d'un message de telle manière à le rendre incompréhensible est appelé chiffrement (encryption). Le résultat de ce processus de chiffrement correspond au message chiffré (cryptogramme). Le processus de reconstruction du message en clair à partir du message chiffré est appelé déchiffrement (décryptage).Ces différents processus sont illustrés par la figure 1.1.

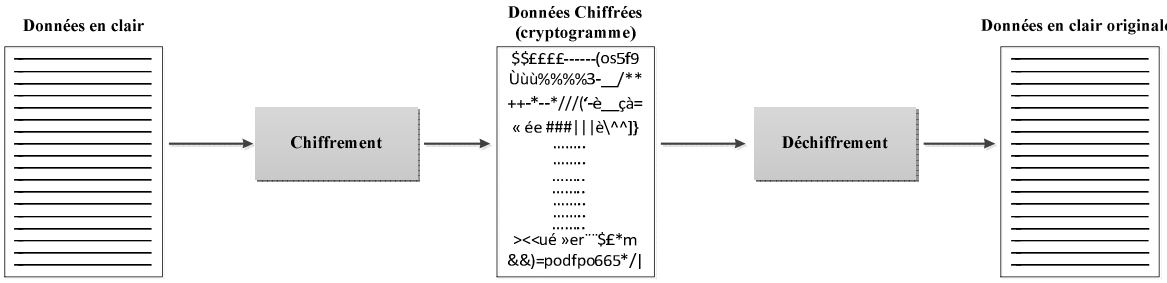


Figure 1.1 - Processus de chiffrement et de déchiffrement

Notons M le message en clair. M peut-être une suite de bits, un fichier de texte, un enregistrement de voix numérisé, ou une image vidéo numérique,…etc. Du point de vue de l'ordinateur, M n'est rien d'autre que de l'information binaire. Le message en clair peut être transmis ou stocké.

Le message chiffré est noté C. La fonction de chiffrement est notée E, cette fonction transforme M en C. En notation mathématique l'opération de chiffrement E est représentée par l'équation (1.1) :

$$E(M) = C \tag{1.1}$$

La fonction inverse, notée D, de déchiffrement transforme C en M. D est définie par l'équation (1.2):

$$D(C) = M \tag{1.2}$$

Comme le but de toutes ces opérations n'est rien d'autre que de retrouver le message en clair à partir du message chiffré de ce même message, l'identité suivante doit être vérifiée :

$$D\big(E(M)\big) = M \tag{1.3}$$

1.4 Types de Cryptographie

D'une manière générale, il existe deux principaux types d'algorithmes de cryptographie qui permettent de sécuriser l'information en clair :

1. Cryptographie symétrique ;
2. Cryptographe Asymétrique.

Ces deux types sont appelés respectivement cryptographie à clé secrète et cryptographie à clé publique.

1.4.1 Cryptographie Symétrique

La cryptographie symétrique est la cryptographie où une seule clé k est utilisée pour le chiffrement et le déchiffrement. Pour cela l'émetteur et le destinataire doivent se mettre d'accord sur la clé à utiliser avant d'échanger des messages. Cette clé doit être gardée secrète [2].

Les fonctions de chiffrement et de déchiffrement s'écrivent de la manière suivante:

$$E_k(M) = C \tag{1.4}$$

$$D_k(C) = M \tag{1.5}$$

La figure 1.2 représente le principe de la cryptographie symétrique.

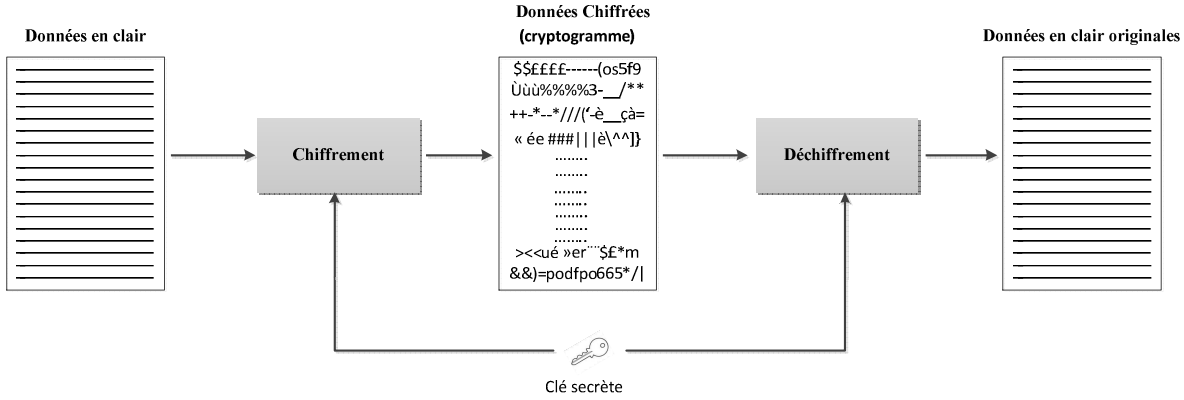


Figure 1.2 - Processus de base de la cryptographie symétrique

1.4.2 Cryptographie Asymétrique

Cette cryptographie est conçue de telle manière que la clé de chiffrement soit différente de la clé de déchiffrement. Ce type de cryptographie est appelée cryptographie à clé publique, car la clé de chiffrement est rendue publique. Autrement dit, n'importe qui peut utiliser la clé de chiffrement pour chiffrer un message mais seul celui qui possède la clé de déchiffrement peut déchiffrer le message chiffré. Dans de tels systèmes, la clé de chiffrement est appelée ***clé publique*** $Kpub$ et la clé de déchiffrement est appelée ***clé privée*** Kpr.

Le chiffrement en utilisant la clé publique est noté :

$$E_{Kpub}(M) = C \tag{1.6}$$

L'opération de déchiffrement est définie par l'équation suivante :

$$D_{Kpr}(C) = M \tag{1.7}$$

Le principe de base de la cryptographie à clé publique est représenté sur la figure 1.3.

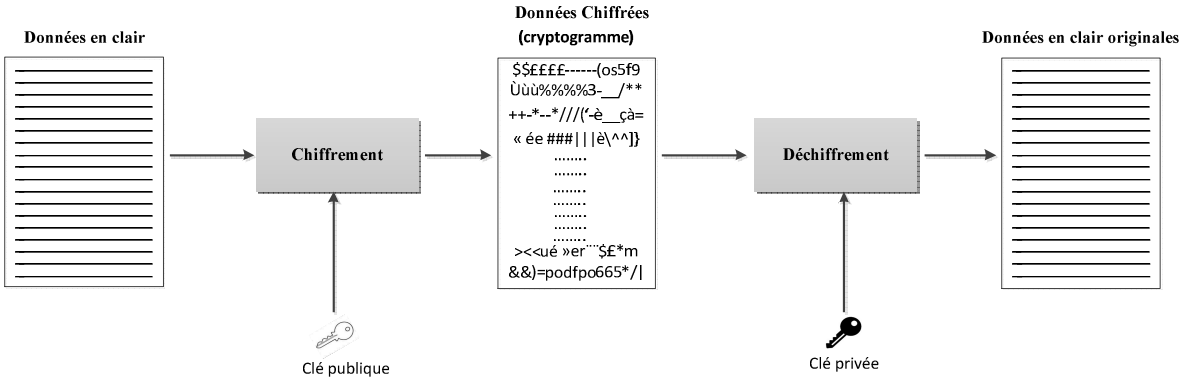


Figure 1.3 - Processus de base de la cryptographie asymétrique

Les algorithmes de cryptographie à clés publiques couramment utilisés, sont:

- Le RSA (Rivest-Shamir-Adleman).
- L'ECC (Elliptic Curve Cryptography).

La sécurité de ces algorithmes est basée sur :

- Le problème de factorisation des nombres entiers, dont la difficulté est essentielle pour la sécurité des crypto systèmes RSA.
- Le problème logarithmique discret de courbe elliptique, dont la difficulté est essentielle pour la sécurité de tous les schémas cryptographiques à courbes elliptiques.

Dans ce travail, nous nous sommes focalisés sur la cryptographie asymétrique à base de courbes elliptiques.

1.5 Cryptographie à base de courbes elliptiques

Dans le but de mieux comprendre la cryptographie à base de courbes elliptiques, nous devons dans un premier temps définir quelques notions algébriques.

1.5.1 Notions algébriques

- **Théorie des Groupes**

On appelle groupe tout ensemble non-vide G muni d'une loi de composition interne « * » qu'on note (G, *), vérifiant les trois propriétés suivantes (axiomes de la structure de groupe): [3].

(i) *Associativité* : $a * (b * c) = (a * b) * c$ pour tout $a, b, c \in G$.

(ii) *Elément neutre* : il existe un élément $e \in G$ tel que $a * e = e * a = a$ pour tout $a \in G$.

(iii) *Elément symétrique :* pour tout $a \epsilon G$, il existe un élément $b \epsilon G$, appelé inverse de a, tel que $a * b = b * a = e$.

Si de plus la loi « * » du groupe admet la propriété suivante :

(iv) *Commutativité :* $a * b = b * a$ pour tout $a, b \in G$.

On dit alors que (G,*) est un groupe Abélien(Commutatif).

- **Groupe fini Cyclique**

Un groupe fini G est dit cyclique, lorsqu'il est engendré par un de ses éléments. Autrement dit, n'importe quel élément x de G peut s'écrire sous forme d'une puissance g ou d'un multiple de g. g est appelé le générateur du groupe G [4]. L'ordre d'un élément e de G est le nombre entier n tel que :

- ✓ $n * e = 0$ si la loi « * » est une opération d'addition.
- ✓ $e^n = 1$ si la loi « * » est une opération de multiplication.

A titre d'exemple, si on considère que $G = \{g^0, g^1, g^2, g^3, g^4, g^5, g^6\}$ et $g^6 = g^0$=1 est l'élément neutre, alors G est un groupe cyclique. L'ordre de l'élément g^2 est 3 car l'élément neutre du groupe est $g^0 = 1$ et $(g^2)^3 = g^6$.

- **Les Corps finis**

Un corps fini F est constitué d'un ensemble fini d'éléments et de deux opérations : addition dénotée par $(+)$ et de multiplication désignée par $(\times)$ [3]. Ces dernières satisfont les propriétés arithmétiques usuelles. Autrement dit, $(F, +)$ et $(F \setminus \{0\}, \times)$ constituent deux groupes abéliens. Leurs éléments neutres sont respectivement 0 et 1.

De plus, l'opération de multiplication est distributive sur l'addition, c'est-à-dire :

$(a + b) \times c = (a \times c) + (b \times c)$ pour tout $a, b \, et \, c \in F$.

Dans la cryptographie basée sur les courbes elliptiques deux types de corps finis sont utilisés [10]:

- ✓ ***Corps premier***

Soit p un nombre premier. Les entiers modulos p, définis dans l'ensemble $\{0, 1, 2, \ldots \ldots p - 1\}$ est un corps fini d'ordre p. Nous désignerons ce corps par F_p. Pour tout entier a, $a \, mod \, p$ correspond au reste unique r, $0 \leq r \leq p - 1$, obtenu en divisant a par p. Cette opération s'appelle réduction modulo p [3] [10].

✓ ***Corps binaire***

Les corps finis d'ordre 2^m sont appelés corps binaires notés par F_2^m.Une façon de construire F_2^m est d'utiliser une représentation de base polynomiale. Ici, les éléments de F_2^m sont des polynômes binaires de degré égal au plus $m-1$ et dont les coefficients $\boldsymbol{a_i}$ sont dans le champ $F_2 = \{0,1\}$ [3] [10] :

$$F_2^m = \left\{a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + \cdots + a_2z^2 + a_1z + a_0 \ : a_i \epsilon \{0,1\}\right\}$$

A noter que la cryptographie manipulant les courbes elliptiques est basée sur une arithmétique adaptée à savoir l'arithmétique modulaire. Cette dernière manipule des opérations non pas sur des entiers de grande taille, mais sur les restes obtenus par la division de ces entiers sur un q avec $q = p$ ou $q = p^m$. p est la caractéristique du corps.

1.5.2 Crypto système à base de courbes elliptiques

Une courbe elliptique E sur un corps fini F_q possède un nombre fini de points avec ses coordonnées (x, y) définies dans F_q qui satisfont l'équation de Weierstrass suivante [8] :

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \qquad (1.8)$$

Avec, $a_1, a_2, a_3, a_4, a_6 \epsilon F_q$ et $\Delta \neq 0$. Δ est le discriminant de E, défini tel que :

- $\Delta = -{d_2}^2 d_8 - 8{d_4}^3 - 27{d_6}^2 + 9d_2d_4d_6$
- $d_2 = {a_1}^2 + 4a_2$
- $d_4 = 2a_4 + a_1a_3$
- $d_6 = {a_3}^2 + 4a_6$
- $d_8 = {a_1}^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2{a_3}^2 - {a_4}^2$

Si la caractéristique de F_q est égale à $2 (q = 2)$, l'équation d'une courbe elliptique dans F_2^m est simplifiée à l'équation suivante [3] :

$$E: y^2 + xy = x^3 + ax^2 + b \mod 2^m \qquad a, b \epsilon F_2^m \qquad (1.9)$$

Sinon, si la caractéristique de corps est p tel que $p \neq 2$, l'équation peut s'écrire dans F_p, comme ci-dessous [3] :

$$E: y^2 = x^3 + ax + b \mod p \qquad a, b \epsilon F_p \qquad (1.10)$$

Dans notre travail, nous avons utilisé ce type de courbe. Cette dernière est définie par :

Son discriminant Δ est déterminé tel que:

$$\Delta = -16(4a^3 + 27b^2) \mod p \neq 0 \qquad (1.11)$$

Les paramètres d'une courbe elliptique sont *G, p, a, b, n* tels que [8]:

- *p* : la taille du corps premier fini.
- *a, b* : les coefficients de l'équation de la courbe E.
- *G* : point générateur de la courbe.
- *n* : ordre du point générateur.

1.5.2.1 Loi de groupe sur les courbes elliptiques

Afin de définir un crypto système sur l'ensemble des points d'une courbe elliptique, nous devons définir une structure algébrique sur les points. La structure algébrique la plus simple qui nous fournit tous les outils nécessaires est le groupe. Il faut donc définir un élément neutre, un élément inverse, le double d'un point et l'addition de deux points de la courbe elliptique qui doivent être associatifs [3].

Soit E_p une courbe elliptique définie sur F_p , et soit P et Q deux points sur E_p:

1. *L'élément neutre* : $P + \infty = \infty + P = P$ pour tout $P \in E_p$. Le point à l'infini ∞ est l'élément neutre.
2. *Symétrique* : Si $P = (x, y) \in E_p$alors $(x, y) + (x, -y) = \infty$. Le point $(x, -y)$ est noté – P. Il représente le point symétrique de P.
3. *Addition de points* : Soit $P = (x_1, y_1) \in E_p$ et $Q = (x_2, y_2) \in E_p$où $P \neq \pm Q$. Alors $P + Q = (x_3, y_3) \in E_p$
4. *Point double* : Soit $P = (x_1, y_1) \in E_p$ où $P \neq -P$. Alors $2P = (x_3, y_3) \in E_p$.

1.5.2.2 Les règles d'addition et de dédoublement de points

Nous avons définis précédemment les notions d'addition de deux points et de dédoublement d'un point. Nous allons à présent donner les équations qui permettent de calculer des coordonnées de ces deux points. En effet dans la littérature, deux systèmes de coordonnées sont utilisés pour manipuler l'addition de deux points et le dédoublement d'un point, à savoir : système de coordonnées affines et système de coordonnées projectives.

1. Système de Coordonnées Affines

Dans ce système chaque point P est représenté par les coordonnées (x, y). Quand l'équation de Weierstrass se trouve sous sa forme affine, les coordonnées dans le plan affine des points obtenus de l'addition de deux points et de dédoublement d'un point sont définis comme suit [11] :

✓ **Addition de points**

On considère que $P = (x_1, y_1) \in E_p$, $Q = (x_2, y_2) \in E_p$ et $P \neq Q$. Le point $R = (x_3, y_3)$issu de l'addition des deux points $P + Q$ est définit comme suit :

$$x_3 = \sigma_a^2 - x_1 - x_2 \; mod \; p \tag{1.12}$$

$$y_3 = \sigma_a(x_1 - x_3) - y_1 \; mod \; p \tag{1.13}$$

Avec :

$$\sigma_a = \left(\frac{y_2 - y_1}{x_2 - x_1}\right) \; mod \; p \tag{1.14}$$

✓ **Dédoublement d'un point**

Soit $P = (x_1, y_1) \in E_p$ et $P \neq -P$ .Le point $R = (x_3, y_3)$ obtenu de dédoublement du point P est défini comme suit :

$$x_3 = \sigma_d^2 - 2x_1 \; mod \; p \tag{1.15}$$

$$y_3 = \sigma_d(x_1 - x_3) - y_1 \; mod \; p \tag{1.16}$$

Avec :

$$\sigma_d = \left(\frac{3x_1^2 + a}{2y_1}\right) \; mod \; p \tag{1.17}$$

2. Système de Coordonnées Projectives

À l'origine ce système de coordonnées a été développé pour contourner le calcul de l'inverse modulaire. Car cette opération est souvent considérée comme complexe à implémenter. Le système de coordonnées projectives est définit comme suit [12] :

Soit F_p un corps premier fini. Soit c et d des entiers positifs. On peut définir une relation d'équivalence sur l'ensemble ${F_p}^3 \setminus \{(0,0,0)\}$ de triples non nuls sur F_p, ayant $Z \neq 0$.

Les coordonnées dans un plan projectif $(X: Y: Z)$ d'un point représenté dans un système de coordonnées affines par (x, y)sont définies telles que :

$$\begin{cases} X = x.Z^c \\ Y = y.Z^d \\ Z = 1 \end{cases}$$

Dans la littérature, selon les valeurs de c et d, trois types de systèmes de coordonnées projectives sont proposés, standard, Chudnovsky et Jacobien**.** Dans notre travail, nous avons étudié et utilisé le système Jacobien**.**

2.1. Système de Coordonnées Jacobiennes

Pour avoir la forme projective de l'équation de Weierstrass sur F_p, nous devons effectuer un changement de variables, tel que $c = 2\ et\ d = 3$ [6]:

$$\begin{cases} x = \frac{X}{Z^2} \\ y = \frac{Y}{Z^3} \end{cases}$$

$$E: Y^2 = X^3 + aXZ^4 + bZ^6 \quad a, b \in F_p \tag{1.19}$$

Avec :

- Le point à l'infini : $\infty = (1:1:0)$
- Symétrique : Si $P = (X_1:Y_1:Z_1) \in E_p$ alors $-P = (X_1:-Y_1:Z_1) \in E_p$

Les coordonnées dans le plan Jacobien des points obtenus de l'addition de deux points et de dédoublement de point sont définies comme suit :

✓ **Addition de points**

Soit $P = (X_1:Y_1:Z_1) et\ Q = (X_2:Y_2:Z_2) \in E(F_p)\backslash\{\infty\}$et $P \neq Q$, on aura

$P + Q$=$\left(\frac{X_3}{Z_3^2}, \frac{Y_3}{Z_3^3}, 1\right)$, avec :

$$\frac{X_3}{Z_3^2} = \frac{X_{add}}{Z_{add}^2}$$

$$\frac{Y_3}{Z_3^3} = \frac{Y_{add}}{Z_{add}^3}$$

$X_{add} = (Y_1Z_2^3 - Y_2Z_1^3)^2 - (X_1Z_2^2 + X_2Z_1^2)(X_1Z_2^2 - X_2Z_1^2)^2$

$Y_{add} = (Y_1Z_2^3 - Y_2Z_1^3)[X_1Z_2^2(X_1Z_2^2 - X_2Z_1^2) - X_3] - Y_1Z_2^3(X_1Z_2^2 - X_2Z_1^2)^3$

$Z_{add} = Z_1Z_2(X_1Z_2^2 - X_2Z_1^2)$

✓ **Dédoublement d'un point**

Soit $P = (X_1:Y_1:Z_1) \in E(F_p)\backslash\{\infty\}$ et $P \neq -P$, on aura $2 \times P$=$\left(\frac{X_3}{Z_3^2}, \frac{Y_3}{Z_3^3}, 1\right)$, avec :

$$\frac{X_3}{Z_3^2} = \frac{X_{ded}}{Z_{ded}^2}$$

$$\frac{Y_3}{Z_3^3} = \frac{Y_{ded}}{Z_{ded}^3}$$

$X_{ded} = \left(3{X_1}^2 + aZ_1^4\right)^2 - 8X_1Y_1^2$

$Y_{ded} = (4X_1Y_1^2 - X_3)(3X_1^2 + aZ_1^4) - 8Y_1^4$

$Z_{ded} = 2Y_1Z_1$

3. Etude comparative entre les coordonnées affines et projectives

Les formules pour l'addition et le dédoublement de points ont été présentées pour les courbes elliptiques définies par $y^2 = x^3 + ax + b$ sur un corps F_p de caractéristique différente de 2. Les formules d'addition de points et le doublement de points en coordonnées affines nécessitent une inversion et plusieurs multiplications modulaires. La comparaison des performances des deux systèmes affine et Jacobien pour le calcul des coordonnés des points issus de l'addition et du dédoublement est présentés sur le tableau 1.1 [6] [12]. Cette comparaison est montrée en termes de nombre d'inverse modulaire (IM) et de multiplication modulaire (MM).

Tableau 1.1 - Comparaison entre les systèmes de coordonnées affines et Jacobiennes

Systèmes de coordonnées (c, d)	**Addition de deux points**	**Dédoublement de point**
Affine	$3 \times MM + 1 \times MI$	$4 \times MM + 1 \times MI$
Jacobien (2,3)	$16 \times MM$	$10 \times MM$

On retiendra qu'avec le système de coordonnées projectives, aucune inversion n'est requise pour calculer les coordonnés des éléments d'une courbe. Les algorithmes de multiplication par un scalaire ne nécessitent alors qu'une seule inversion finale pour obtenir un résultat en coordonnées affines.

Sachant que l'inversion modulaire est significativement plus coûteuse que la multiplication modulaire, alors il peut être avantageux de représenter des points en utilisant des coordonnées projectives.

1.6 Problème du Logarithme Discret pour les courbes elliptiques (ECDLP)

Le problème peut être difficile suivant le groupe dans lequel il est posé. Ainsi le niveau de sécurité d'un crypto système basé sur les courbes elliptique dépend de la courbe elliptique utilisée. Il est définit comme suit [9] :

Soit :

- une courbe elliptique E définie sur un corps fini Fq.
- $(E(Fq), +)$ le groupe des points de la courbe elliptique Eq.
- P un point $\in E\ (Fq)$ d'ordre n.
- Q un point $\in E\ (Fq)$ tel que :

$$\begin{cases} \mathrm{Q} = \mathrm{k} \cdot \mathrm{P} \\ \mathrm{k} \leq n \end{cases}$$

On appelle alors problème du logarithme discret le problème qui consiste à déterminer l'entier $k \in [0, n-1]$ connaissant la courbe Eq et les points P et Q.
L'entier k est appelé le logarithme discret de Q à la base P.

Les paramètres de la courbe elliptique pour les schémas cryptographiques doivent être soigneusement choisis afin de résister à toutes les attaques connues sur l'ECDLP. L'algorithme le plus naïf pour résoudre l'ECDLP est la recherche exhaustive par laquelle on calcule la séquence des points $P, 2P, 3P, 4P, \ldots$ Jusqu'à ce que Q soit rencontré. Le temps de fonctionnement est d'environ n étapes dans le pire des cas et $n / 2$ étapes en moyenne. Par conséquent, la recherche exhaustive peut être contournée en sélectionnant des paramètres de courbe elliptique avec n suffisamment grand pour représenter une quantité infaillible de calcul.

1.7 Protocoles de Chiffrement et de Déchiffrement de données basés sur les courbes elliptiques

Soit la courbe elliptique E_p avec les paramètres $E_p = (p, a, b, G, n)$ et soit M le message à envoyer. Nous devons d'abord représenter ce message sur la courbe elliptique. Soit Pm le point correspondant à la représentation du message M sur la courbe E_p [5].

1.7.1 Protocole de chiffrement

Pour chiffrer le point Pm, nous avons besoin d'une clé publique et d'une clé privée. Considérons qu'Alice et Bob sont les deux parties communicantes. Ils conviennent d'une équation de courbe elliptique E_p commune et d'un générateur G.
Soient les clés privées d'Alice et de Bob respectivement nA et nB tel que $0 < nA, nB < n$.
Les clés publiques d'Alice et de Bob sont données respectivement par:
$Pa = nA \times G$ et $Pb = nB \times G$.

Si Alice veut envoyer un message Pm à Bob, Alice précède par une génération d'un entier k dans l'intervalle $[1, (n-1)]$ puis utilise la clé publique de Bob pour chiffrer le point Pm.
La paire de points Pc qui représente le résultat du chiffrement est calculé par l'équation suivante :

$$Pc = \{(kG), (Pm + kPb)\} \quad (1.20)$$

1.7.2 Protocole de déchiffrement

Pour retrouver le point Pm correspondant au message M en clair, Bob calcule la multiplication $nB \times k.G$, puis exécute la soustraction suivante :

$$Pm = \{(Pm + kPb) - (nBkG)\} \quad (1.21)$$

Sachant que : $k \times Pb = nB \times k.G$.

Comme le multiplicateur nB est la clé secrète de Bob, seul Bob peut déchiffrer le message envoyé par Alice. La figure 1.4 illustre les étapes de chiffrement et de déchiffrement du message M.

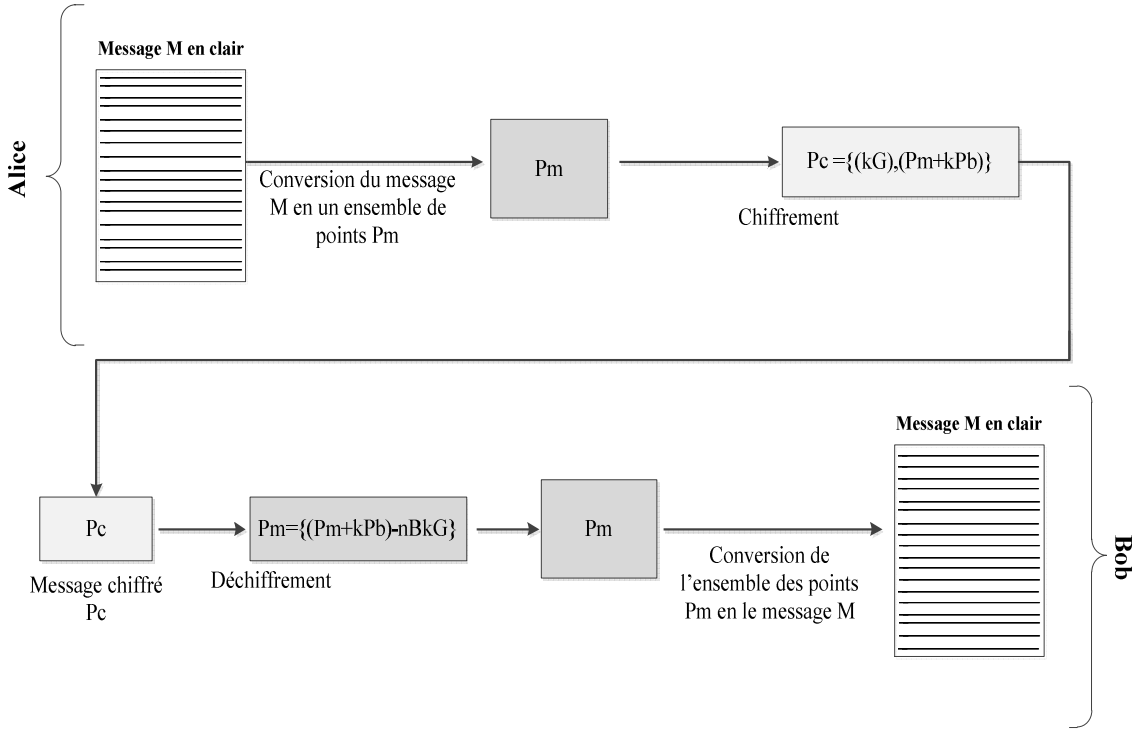


Figure 1.4 - Protocole de chiffrement et de déchiffrement à base de courbes elliptiques

1.8 Génération des clés

Une paire de clés est associée à un ensemble particulier de paramètres de courbe $E_p = (N, a, b, G, n)$ [3] [11].

- La clé privée est un nombre entier d sélectionné aléatoirement dans l'intervalle $[1, n-1]$
- La clé publique correspondante est un point Q tel que : $Q = d \times G$

Algorithme 1.1- Génération d'une paire de clés

Entrée : Paramètres de la courbe $E_p = (N, a, b, G, n)$

Sortie : Clé publique Q, clé privée d.

Début

1) $\boldsymbol{d = Random[1, n-1]}$
2) Calculer $\boldsymbol{Q = d \times G}$
3) Retourner $(\boldsymbol{d, Q})$

Fin

Notons que dans ce qui a précédé, multiplier ne signifie pas une simple multiplication que nous faisons en algèbre, mais plutôt une multiplication d'un point par un scalaire. Cette opération est nommée la multiplication scalaire. Elle est considérée comme étant l'opération cœur d'un crypto système ECC.

1.9 La multiplication scalaire

On remarque que toute la sécurité des courbes elliptiques repose sur la complexité de calcul de la multiplication d'un point P par un scalaire k, Noté: $Q = k \times P$ avec P et Q des points de la courbe E_p et $k \in F_p$. Cette opération constitue non seulement la base du calcul du chiffrement et du déchiffrement, mais aussi elle est présente dans le processus de génération des clés.

La multiplication scalaire peut être exécutée par une suite d'addition consécutive de points :

$$Q = k \times P = \underbrace{P + P + P + \cdots + P}_{\boldsymbol{k\ fois}} \quad (1.22)$$

Dans la littérature, plusieurs algorithmes sont proposés pour accélérer l'exécution de cette opération , la méthode de *Montgomery power ladde*r a été proposée pour cela [7].

- ✓ **Montgomery Power Ladder**

Cette méthode consiste à représenter l'entier k sous sa forme binaire $\sum_{i=0}^{l-1} k_i\, 2^i$ et de parcourir ses l bits du bit de poids fort au bit de poids faible, en suivant l'enchainement des opérations illustrées dans l'algorithme ci-dessous :

Algorithme 1.2- Montgomery power ladder

Entrées :$\boldsymbol{P}$, $\boldsymbol{k} = (\mathbf{1}, \boldsymbol{k_{l-2}}, \dots, \boldsymbol{k_0})_{\mathbf{2}}$

Sorties :$\boldsymbol{Q} = \boldsymbol{k} \times \boldsymbol{P}$

Début

$\boldsymbol{R_0} \leftarrow \boldsymbol{P}$;//initialisation

$\boldsymbol{R_1} \leftarrow \mathbf{2}\boldsymbol{P}$;

Pour $\boldsymbol{i} = \boldsymbol{l} - \mathbf{2}$ jusqu'à $\mathbf{0}$ faire

Si ($\boldsymbol{k_i} = \mathbf{0}$) alors

$\boldsymbol{R_1} \leftarrow \boldsymbol{R_0} + \boldsymbol{R_1}$; //addition de deux points

$\boldsymbol{R_0} \leftarrow \mathbf{2}\boldsymbol{R_0}$;//dédoublement de point

Sinon

Si ($\boldsymbol{k_i} = \mathbf{1}$) alors

$\boldsymbol{R_0} \leftarrow \boldsymbol{R_0} + \boldsymbol{R_1}$;//addition de deux points

$\boldsymbol{R_1} \leftarrow \mathbf{2}\boldsymbol{R_1}$;//dédoublement de point

Finsi ;

Finsi ;

Fait ;

$\boldsymbol{Q} \leftarrow \boldsymbol{R_0}$;

Retourner Q ;

Fin

La multiplication scalaire $k \times P$ est l'opération principale du protocole ECC. Ainsi, la complexité de la conception d'un crypto système ECC est liée à l'implémentation de ce type de multiplication et de ses opérations arithmétiques de base, à savoir l'arithmétique modulaire.

1.10 Arithmétique modulaire

Soit A, p deux entiers tels que $0 \leq A < p$. L'arithmétique modulaire est basée sur le calcul du reste T qui correspond à la division de A sur p. Elle utilise une notation particulière, définie par l'équation suivante :

$$T = A \bmod p \tag{1.23}$$

Le concept principal de cette arithmétique est de manipuler non pas sur les nombres eux-mêmes, mais sur les restes de leur division par entier.

1.10.1 Addition modulaire

L'addition modulaire est effectuée en exécutant une addition de deux nombres Aet B. Puis déterminé le reste T de la division de$(A + B)$ sur l'entier p. Elle est définie par l'expression suivante :

$$T = (A + B)\ mod\ p \tag{1.24}$$

Son algorithme est présenté comme suit [13] :

Algorithme 1.3- Algorithme de l'addition modulaire

Entrées: A, B, p

Sorties: $T = (A + B)\ mod\ p$

Variable intermédiaire: C

Début

1. $C = A + B$;
2. ***Si*** $C \geq p$ ***alors***
3. $C = C - p$;
4. ***Sinon***
5. $T = C$;
6. ***Fin si***
7. $Retourner\ T$;

Fin

1.10.2 Soustraction modulaire

La soustraction modulaire consiste en une soustraction suivie d'une division. Elle soustrait un nombre B de A puis elle prend le reste T de la division de $(A - B)$ sur l'entier p. Cette opération est définie par l'expression suivante :

$$T = (A - B) mod\ p \tag{1.25}$$

L'algorithme de calcul de cette opération est défini comme suit [13]:

Algorithme 1.4- Algorithme de la soustraction modulaire

Entrée: A, B, p

Sortie: $T = (A - B)\ mod\ p$

Variable intermédiaire: C

Début

1. $C = A - B$;
2. ***Si*** $C < 0$ ***alors***
3. $C = C + p$;
4. ***Sinon***
5. $T = C$;
6. ***Fin si***
7. $Retourner\ T$;

Fin

1.10.3 Multiplication modulaire

La multiplication modulaire consiste à calculer l'opération$(A \times B)$ et déterminer le reste T de sa division par l'entier p. Cette opération est définie par l'équation suivante :

$$T = (A \times B) mod p \tag{1.26}$$

Son algorithme d'exécution est défini comme suit [13] :

Algorithme 1.5- Algorithme de la multiplication modulaire

Entrée: A, B, p

Sortie: $T = (A \times B)\ mod\ p$

Variable intermédiaire: C

Début

1. $C = A \times B$;
2. $T = C\ mod\ p$;
3. $Retourner\ T$;

Fin

La multiplication modulaire est une opération importante dans les protocoles de cryptographie à clé publique. Elle est souvent considérée comme étant une opération critique, car elle exige le calcul de deux opérations complexe, à savoir, la multiplication et la division. De ce fait, pour contourner cette problématique, plusieurs méthodes ont été développées. Parmi ces dernières la méthode proposée par Montgomery est considérée comme étant la

méthode la plus adaptée pour l'implémentation de cette opération. Cette méthode repose sur des additions et des décalages.

1.10.3.1 Multiplication Modulaire de Montgomery (MMM)

En 1985, Peter Montgomery propose un algorithme de multiplication modulaire qui transforme la division par p en une division par une puissance n est le nombre de chiffres qui compose le modulo *p.* β est la base de numérotation [14]. La MMM est définie par l'équation suivante :

$$T = (A \times B \times R^{-1})\ mod p \tag{1.27}$$

Avec :

- $0 \leq A, B < p$
- R est la constante de Montgomery, tel que $R \geq \beta - n$ et $PGCD(R, p) = 1$.
- β est la base de représentation des données.
- n est le nombre de chiffres qui correspond à la représentation de p en base β.

L'algorithme de Montgomery est défini comme suit [15]:

Algorithme 1.6- Algorithme de la multiplication de Montgomery

Entrées: $A, B, p\ avec\ 0 \leq A, B < p$

Précalculés: $p'\ et\ R\ avec\ R \geq \beta^{-n}, (-p) \times p' = 1\ mod\ R\ et\ PGCD(R, N) = 1$

Sortie: $T = (A \times B \times R^{-1}) mod\ p$

Variable intermédiaire: C

Début

1. $C = A \times B$;
2. $q = (C \times p') mod\ R$;
3. $T = (C + q \times B)/R$;
4. ***Si*** $T \geq p$ ***alors***
5. $T \leftarrow T - p$;
6. ***Si non***
7. $T \leftarrow T$;
8. ***Fin si***
9. $Retourner\ T$;

Fin

1.10.3.2 Représentation de Montgomery et ses propriétés

La MMM calcul non pas comporte $T = (A \times B)\ mod\ p$, mais le résultat est obtenu avec un facteur supplémentaire R^{-1}. Ce qui pose une difficulté lors d'un calcul intensif de la multiplication modulaire. Pour ce faire, Montgomery utilise une conversion des données d'entrées vers un autre domaine de représentation, en utilisant la notation $mon(X)$. Le principe est de calculer $mon(X) = (X \times R)\ mod\ p$ où $X < p$. Les propriétés de la MMM sont résultat dans les points suivants [16]:

a. Conversion :

- De la représentation classique vers la représentation de Montgomery :

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

- De la représentation de Montgomery vers la représentation classique:

$$mon(A) \rightarrow A: Montgomery(mon(A), 1) = (mon(A) \times 1) mod\ p$$
$$= (A \times R \times R^{-1}) mod\ p = Amodp$$

b. Stabilité :

$$Mongomery\big(mon(A), mon(B)\big) = (mon(A) \times mon(B) \times R^{-1} modp$$
$$= (A \times R \times B \times R \times R^{-1}) modp = (A \times B \times R) modp = mon(A \times B)$$

Cela signifie que le résultat de la MMM de deux opérandes représentés dans le domaine de Montgomery, est obtenu dans le même domaine.

1.10.4 Inverse modulaire

L'inverse modulaire est défini par l'équation suivante :

$$T = x^{-1} mod\ N \tag{1.28}$$

Parmi les quatre opérations de base, l'inverse modulaire est considéré comme étant l'opération la plus difficile à réaliser. Dans notre travail, l'opération en question est utilisée lors de la conversion des coordonnées d'un point du plan projectif vers le plan affine. Le calcul de cette opération peut être remplacé par le calcul d'une exponentiation modulaire, en utilisant le petit théorème de Fermat [17]. Ce théorème est défini comme suit:

Si p est un nombre premier, nous avons :

$$1 = x^{p-1}\ \ mod\ p$$

Ainsi :

$$x^{-1} = x^{N-2}\ \ mod\ N \tag{1.29}$$

1.10.5 Exponentiation modulaire

L'exponentiation modulaire est définie par l'équation suivante :

$$Y = X^Z \ mod \ p \tag{1.30}$$

La première règle de l'exponentiation modulaire est qu'on ne peut pas calculer Y par le calcul direct de X^Z puis calculer ensuite le reste de la division du résultat obtenu par p pour trouver le reste Y. La méthode naïve serait d'initialiser avec la valeur $Y = X \ mod \ p$ et de faire la multiplication modulaire Z *fois* jusqu'à obtenir $Y = X^Z \ mod \ p$. Cette méthode est pratiquement infaisable pour des exposants de de grand taille. De ce fait, plusieurs algorithmes sont proposés pour calculer cette opération dans la littérature. Dans notre travail, nous avons utilisé l'algorithme binaire R2L [18]. Ce dernier est basé sur des décalages à droite *bit-par-bit* sur l'exposant *Z.* Cet algorithme est présenté comme suit :

Algorithme 1.7 –Algorithme R2L

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

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

Sortie *:* $Y := X^z \ mod \ N$

Début

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

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

1.11 Application de la MMM à la Multiplication scalaire

En vertu des étapes d'exécution de la MMM (conversion des données d'entrée dans le domaine de Montgomery, calcul de la MMM dans le domaine de Montgomery, représentation du résultat dans le domaine classique), son application à la multiplication scalaire $Q = k \times P$ nécessite des modifications sur l'exécution de cette dernière. L'algorithme obtenu est définie comme suit :

Algorithme 1.8- Algorithme de la multiplication scalaire de Montgomery

Entrées: $k = (1, k_{t-2}, k_{t-3}, \dots k_0)_2$, $P(X_P, Y_P, Z_P) \in E_p\big(F(p)\big), a, p, R$

Sortie: $Q\big(X_q, Y_q, Z_q\big) = k \times G$

Début

1 .$X_P = mon(X_P) = Montgomery(X_P, R^2 mod\ p)$;

2. $Y_P = mon(Y_P) = Montgomery(Y_P, R^2 mod\ p)$;

3. $Z_P = mon(Z_P) = Montgomery(Z_P, R^2 mod\ p)$;

4. $a = mon(a) = Montgomery(a, R^2 mod\ p)$;

5. $R_{\mathbf{0}} \leftarrow P$;

6. $R_1 \leftarrow Dédoublement\ (P, a, p)$;

7. ***Pour*** i ***allant de*** $t-2$ ***jusqu'***à 0 ***faire***

8. ***Si*** $k_i = 0$ ***alors***

9. $R_1 = Addition(R_{\mathbf{0}}, R_{\mathbf{1}}, p)$;

10. $R_0 = Dédoublement(R_{\mathbf{0}}, a, p)$;

11. ***Si non***

12. $R_0 = Addition(R_{\mathbf{0}}, R_{\mathbf{1}}, N)$;

13. $R_1 = Dédoublement(R_{\mathbf{1}}, a, N)$;

14. ***Fin si***

15. ***Fin pour***

16. $Q\ (X_q, Y_q, Z_q) = R_0$;

17. $X_q = Montgomery\big(X_q \times R\ mod\ N, 1\big)$;

18. $Y_q = Montgomery\big(Y_q \times R\ mod\ p, 1\big)$;

19. $Z_q = Montgomery\big(Z_q \times R\ mod\ p, 1\big)$;

20. $Retourner\ Q$;

21. ***Fin***

Dans cet algorithme, les fonctions $Montgomery(.)$, $Addition(.)$ et $Dédoublement(.)$ correspondent respectivement, aux fonctions de calcul de la MMM, d'addition de deux points et de dédoublement de point. Son exécution est effectuée en trois étapes :

- La première étape consiste en la conversion des coordonnées (X_P, Y_P, Z_P) du point P et du paramètre a de la courbe elliptique, du domaine classique vers le domaine de Montgomery La valeur de $R^2\ mod\ p$ est une constante qui dépend de p et de la base β. Lors de cette étape, un premier dédoublement est effectué aussi sur le point P.

- La deuxième étape correspond au processus itératif du calcul de la multiplication scalaire, où cette dernière est exécutée dans le domaine de Montgomery.
- La troisième étape est associée à la conversion des coordonnées du point résultant $Q(X_q, Y_q, Z_q)$ vers le domaine classique de présentation des nombres.

La complexité de calcul de l'algorithme 1.8 en termes du nombre de MMM est montrée sur le tableau 1.2.

Tableau 1.2 - Complexité algorithmique de l'algorithme 1.8

Opération	Complexité
Addition de deux points	$16 \times MMM$
Dédoublement de point	$10 \times MM$M
Multiplication scalaire	$3 \times MMM + 10 \times MMM + (t-1) \times ((16 \times MMM) + (10 \times MMM))$

1.12 Conclusion

Dans ce chapitre nous avons présenté des généralités sur la cryptographie. Nous nous sommes intéressés en particulier à l'étude de l'un des algorithmes asymétriques les plus utilisés, en l'occurrence l'ECC. Pour mieux comprendre les mécanismes de ce dernier, nous avons présenté également le type d'arithmétique utilisée, à savoir l'arithmétique modulaire.

Le chapitre suivant sera consacré à la présentation des systèmes embarqués sur circuit FPGA, ainsi que, à la plateforme utilisée pour l'implémentation du crypto système ECC.

Chapitre 2

Systèmes Embarqués

2.1 Introduction

Le monde de l'électronique a connu une croissance fulgurante durant les dernières décades. Des Télécommunications aux jeux, de l'automatisme aux banques. Presque tous les aspects de notre quotidien de vie emploient des composants électroniques. Ces composants sont dits systèmes embarqués. Ces derniers sont souvent perçus comme ordinateurs. Par exemple, ils n'ont pas de clavier ou d'écran pour interagir avec l'utilisateur. De plus, ils ne font pas tourner de manière classique les systèmes d'exploitation et les applications. Quelques fois, ces systèmes constituent leurs propres produits (téléphone mobile) mais généralement ils sont embarqués dans un autre système pour qu'ils fournissent de meilleures fonctionnalités et de performance (ex appareil de contrôle d'un moteur de véhicule).

Dans le passé, les systèmes embarqués font référence aux applications qui nécessitent l'intégration de plusieurs composants sur une même carte électronique. Cependant, grâce à l'évolution de la technologie de fabrication des circuits intégrés, des systèmes entiers peuvent être implémenté sur une même puce. Ainsi, on parle souvent de SoC (System on Chip) et de système sur circuits programmables PSoC (Programmable System on Chip). La solution SoC est généralement à faible coût. Elle permet d'implémenter des applications performantes, en termes de consommation de puissance et de temps d'exécution. Les plateformes de types PSoC sont recommandées pour le prototypage et la production à faible série. Les circuits intégrés sont de type FPGAs (Field Programmable Gate Array) [20] [22].

Ce chapitre est consacré à la présentation des systèmes embarqués et leur méthodologie de conception sur circuit FPGA de Xilinx.

2.2 Systèmes embarqués sur puce (SoC)

Un système embarqué sur puce ou SoC est un système autonome composé par un ou plusieurs processeurs, de la mémoire, des périphériques et un bus d'interconnexion [21]. L'ensemble de ces composants constituent la partie matérielle du SoC. L'architecture de base d'un SoC est montrée sur la figure 2.1.

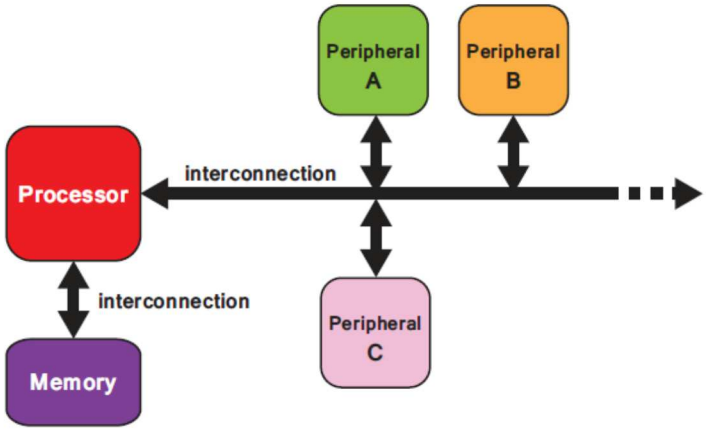


Figure 2.1- L'architecture de la partie matérielle d'un SoC

Le processeur est considéré comme étant l'élément central de la partie matérielle. La partie logicielle est exécutée sur le processeur, comprenant des applications, généralement basées sur un système d'exploitation et une hiérarchie de couches de fonctionnalités logicielles. Ces dernières constituent la partie logicielle de l'application.

La communication entre les éléments du système s'effectue via des interconnexions. La communication entre les périphériques et le processeur est effectuée en déclarant sa basse adresse mémoire dans la partie logicielle, stockée dans la mémoire. Ce type de communication est appelé *Memory mapped interface* [23].

2.3 Systèmes Embarqués sur la carte ZedBoard

Dans notre travail, nous avons utilisé la carte de prototypage ZedBoard. Celle-ci contient le circuit FGPA XC7Z020-1CSG484CES EPP de la famille Zynq-7000 de Xilinx [26]. Ce type de circuit est constitué d'un processeur core-duo ARM et d'une partie dédiée à l'implémentation des circuits logiques. La structure de cette dernière partie est basée sur l'architecture traditionnelle d'un circuit FPGA.

Dans Zynq, ARM est un processeur de qualité d'application, capable d'exécuter des systèmes d'exploitation complets tels que Linux [26], [24]. Cela fait de Zynq une plateforme idéale pour la mise en œuvre de SoCs flexibles.

2.3.1 Architecture d'un circuit FPGA Zynq-7000

L'architecture globale d'un circuit FPGA de la famille Zynq-7000 est montrée sur la figure 2.2 [23].

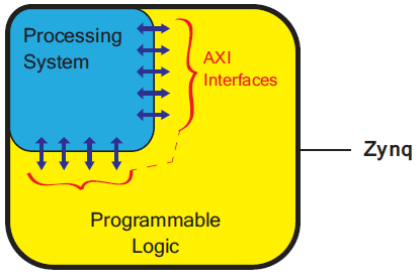


Figure 2.2 - Architecture élémentaire d'un circuit Zynq-7000

Ce type de circuit FPGA comprend deux parties principales :

- Un système de traitement (PS) formé autour d'un processeur ARM Cortex-A9 à double cœur.
- Une logique programmable (PL) équivalente à celle d'un circuit FPGA.

Il dispose également :

- Une mémoire intégrée.
- Une variété de périphériques tel que : un timer, un contrôleur d'interruption,…etc.
- Des interfaces de communication, tel que l'UART, un contrôleur USB ...

La figure 2.3 fournit une vue du système matériel représenté sur la figure 2.1, mappé sur le circuit Zynq-7000. Les architectures ont été simplifiées, mais l'objectif est de fournir une clarification de haut niveau sur la façon dont les SoCs intégrés se mappent vers les périphériques Zynq. Le PS dispose d'une architecture fixe et héberge le processeur et la mémoire système, tandis que la PL est complètement flexible, ce qui donne au concepteur une «toile vierge» pour créer des périphériques personnalisés ou pour réutiliser les standards. Les interconnexions sont implémentées via des interfaces AXI reliant le PS et le PL.

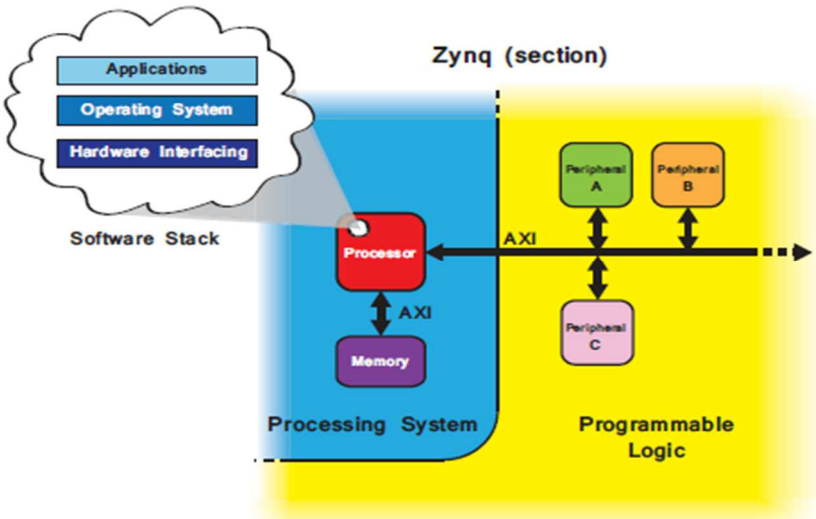


Figure 2.3 - Relation entre le système matériel d'un SoC, le système logiciel et l'architecture Zynq

Nous allons à présent détailler les deux parties de la plateforme Zynq de la famille Zynq-7000 All Programmable SoC, à savoir : le système de traitement (PS) et la logique programmable (PL).

2.3.1.1 Definition de la partie PS (Processing System)

La partie PS du circuit Zynq est une partie fixe de silicium qui ne varie pas entre les différents composants de la famille Zynq-7000, ni en capacité ni en taille [23] [25]. Un schéma fonctionnel montrant l'architecture de la partie PS est illustré à la Figure 2.5.

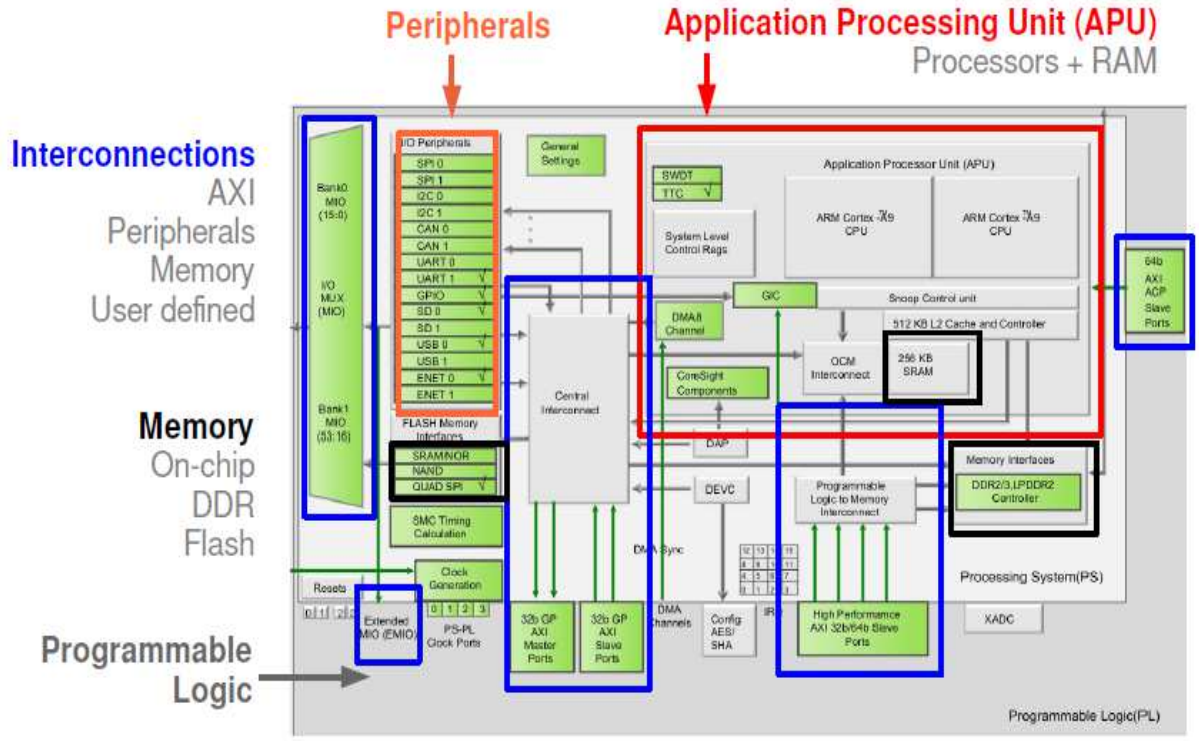


Figure 2.4 - Vue détaillée de la partie PS de Zynq-7000

On peut voir que la PS comprend quatre blocs majeurs:

- ✓ L'Application Processor Unit (APU) qui contient le processeur ARM Cortex-A9.
- ✓ Interfaces mémoires(Memory).
- ✓ Périphériques Entrées/Sorties (I/O Peripherals).
- ✓ Interconnections (Interconnections).

a) Unité de traitement d'applications (APU)

Un schéma synoptique simplifié de l'APU est illustré à la figure 3.6.

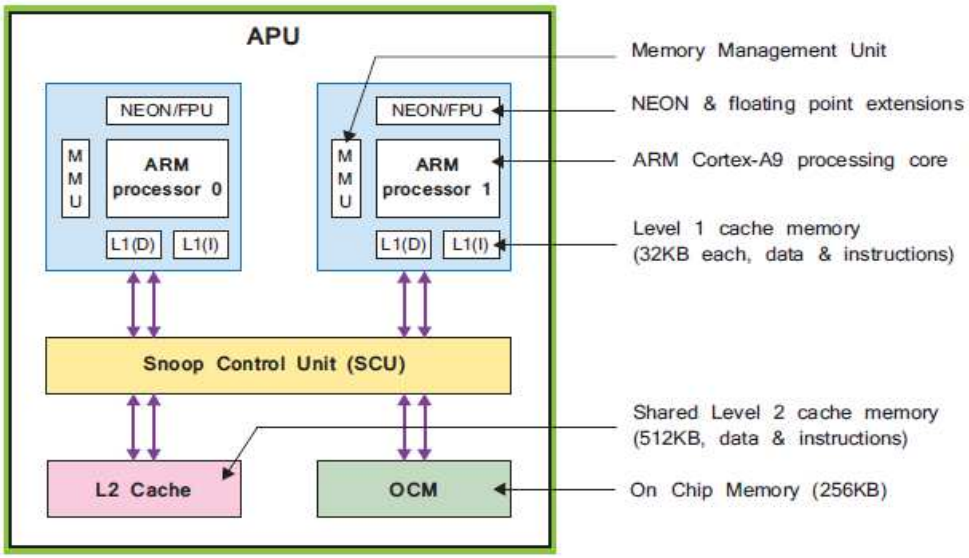


Figure 2.5 - Schéma de l'APU

L'APU est principalement composé de :

- Deux noyaux de traitement ARM, chacun avec des unités de calcul associées à un moteur de traitement des médias NEON et une unité à point flottant FPU, une unité de gestion de la mémoire MMU et une mémoire cache de niveau 1 (en deux sections pour les instructions et les données).
- Une mémoire cache de niveau 2 et une autre mémoire sur puce OCM.
- Une unité de contrôle SCU forme un pont entre les noyaux ARM et le cache de niveau 2 et les mémoires OCM. Cette unité a également la responsabilité d'interagir avec la PL.

b) Interfaces Mémoires (Memory)

L'unité interface mémoire contient un contrôleur mémoire dynamique et des modules d'interface mémoire statique. Le contrôleur mémoire dynamique supporte les mémoires DDR3, DDR3L, DDR2 et LPDDR2. Le contrôleur mémoire statique supporte les interfaces NAND et QSPI flash, un bus de données parallèle et une interface NOR flash parallèle.

c) Périphériques Entrées/Sorties (I/O Peripherals)

L'unité IOP contient les périphériques de communication (deux périphériques Ethernet MAC, deux périphériques USB 2.0, deux contrôleurs CAN, deux contrôleurs SD/SDIO, deux ports SPI full-duplex, deux UART, deux interface I2C maitre et esclaves I2C et jusqu'à 118 bits de GPIO.

Les périphériques IOP communiquent aux composants extérieurs via les pins E/S multi-usage (MIO).

d) Interconnections

L'APU, l'unité interface mémoire et les IOPs sont tous connectés les uns aux autres et à la partie PL par une interconnexion ARM AMBA AXI. L'interconnexion est non-bloquante et supporte des multiples transactions maitre-esclave simultanées.

2.3.1.2 Definition de la partie PL (Programmable Logic)

La partie PL du périphérique Zynq est représentée dans la Figure 2.6, avec diverses fonctionnalités mises en surbrillance. La PL se compose principalement d'un tissu logique FPGA à usage général. Elle est composée d'une matrice de blocs logiques configurables CLB entourés de blocs d'entrée sortie programmable IOB pour l'interfaçage. L'ensemble est relié par un réseau d'interconnexions programmable [24] [25].

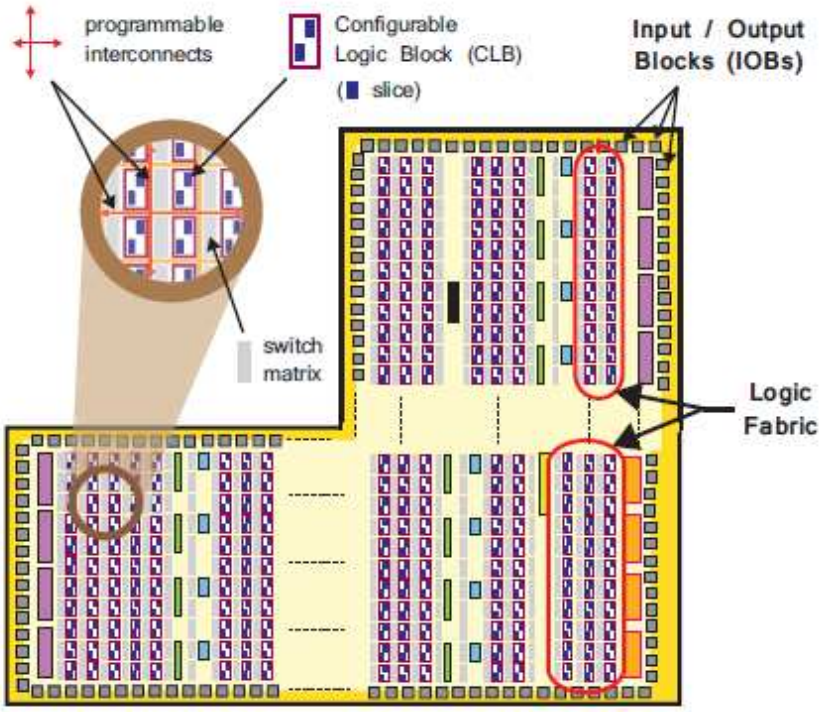


Figure 2.6 - Vue détaillée de la partie PL de Zynq-7000

Les caractéristiques de la PL (illustrées à la figure 2.6) peuvent être résumées comme suit:

a) CLB (Bloc logique configurable)

Les CLB sont des groupements légers et réguliers d'éléments logiques disposés dans un réseau bidimensionnel sur la PL et connectés à d'autres ressources similaires via des interconnexions programmables. Chaque CLB est positionné à côté d'une matrice de

commutation et contient deux tranches logiques (slice). La structure d'un CLB est montrée sur la figure 2.7.

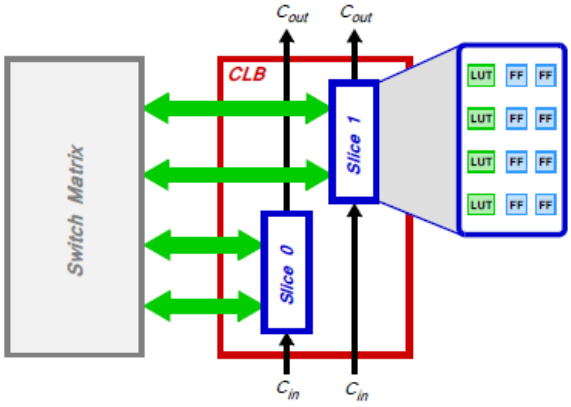


Figure 2.7 - Composition d'un bloc logique configurable (CLB)

- **Slice**

Une sous-unité dans le CLB, qui contient des ressources pour la mise en œuvre de circuits logiques combinatoires et séquentiels. Les tranches Zynq sont composées de 4 tables de recherche, de 8 flip-flops et d'autres logiques.

- **Table de recherche (LUT)**

Une ressource flexible capable de mettre en œuvre soit :

(i) une fonction logique allant jusqu'à six entrées.

(ii) une petite mémoire ROM.

(iii) une petite mémoire d'accès aléatoire (RAM).

(iv) un registre à décalage horaire.

Les LUT peuvent être combinés pour former des fonctions logiques, des mémoires ou des registres à décalage plus importants, au besoin.

- **Flip-flop (FF)**

Un élément de circuit séquentiel implémentant un registre à 1 bit, avec fonctionnalité de réinitialisation. L'un des FF peut éventuellement être utilisé pour implémenter un loquet.

- **Switch Matrix**

Une matrice de commutation se trouve à côté de chaque CLB, et fournit une installation de routage flexible pour établir des connexions :

(i) entre les éléments d'un CLB.

(ii) d'un CLB à d'autres ressources sur le PL.

b) Blocs d'entrée / sortie (IOB)

Les IOB sont des ressources qui fournissent une interface entre les ressources logiques PL et les «pads» du périphérique physique utilisés pour se connecter aux circuits externes. Chaque IOB peut gérer un signal d'entrée ou de sortie 1 bit. Les IOB sont généralement situés autour du périmètre de l'appareil.

En plus du tissu général, il existe deux composants spéciaux: les blocs RAM pour les besoins en mémoire dense et les tranches DSP48E1 pour l'arithmétique à haute vitesse.

c) Blocs RAMs

Ces blocs RAMs peuvent être configurés en mémoire RAM, en mémoire ROM ou en mémoire FIFO. Chaque bloc mémoire est capable de stocker jusqu'à 36Kb.Ils peuvent être configurés comme une mémoire d'une taille de 36 Kb ou comme deux mémoires indépendantes de taille 18 Kb.

d) Les Blocs DSPs

Les blocs DSP48E sont des ressources matérielles à haute performance, capables d'accélérer les opérations arithmétiques :

- Additions/soustractions.
- Multiplications.
- Additions/soustractions suivies par des multiplications, ou multiplications suivies par des additions/soustractions.

2.3.2 la carte de prototypage ZedBoard

La ZedBoard est une carte d'évaluation et de développement basée sur la plate-forme de traitement extensible Xilinx Zynq-7000. En combinant un système de traitement (PS) Cortex-A9 avec 85 000 cellules logiques programmables (PL) série-7, l'EPP Zynq-7000 peut être ciblé pour une large utilisation dans de nombreuses applications. Le solide mélange de périphériques embarqués et de capacités d'extension ont fait de la ZedBoard une plate-forme idéale pour les novices et les designers expérimentés [26]. Quelques fonctions fournies par le ZedBoard:

- Deux boutons poussoirs.
- Un port Ethernet.
- Un port USB.
- Un port USB-to UART qui est un adaptateur avec le protocole RS232.

- Un port HDMI.
- De la mémoire (512 Mo de DDR3 (128M x 32), 256 Mb QSPI Flash).
- Huit Leds et huit commutateurs.
- Un connecteur VGA.
- Un port Audio.

La carte de prototypage ZedBoard est illustrée sur la figure2.8.

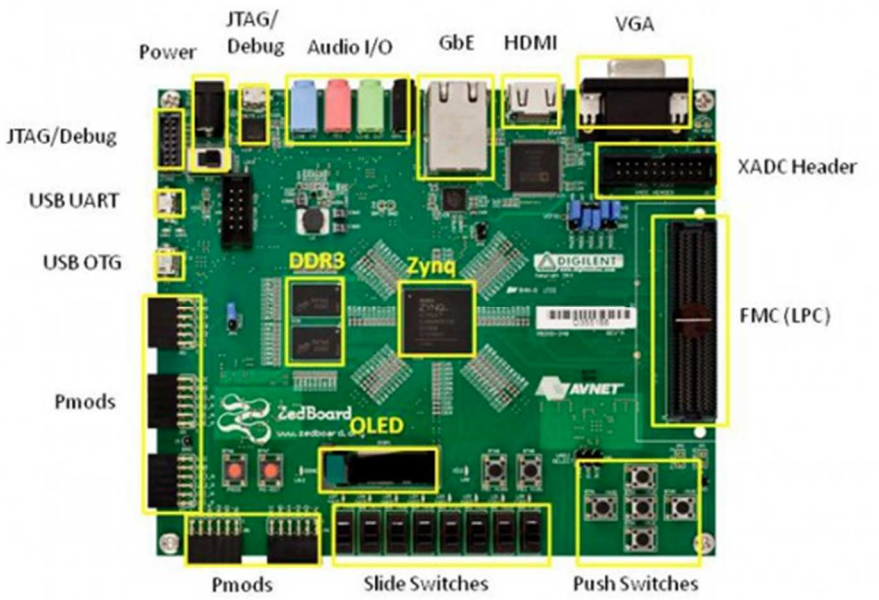


Figure 2.8 - Carte ZedBoard

2.4 Conclusion

Dans ce chapitre, nous avons défini d'une manière succincte les systèmes embarqués sur puce. Puis nous avons présenté la famille du FPGA ciblée, en l'occurrence la famille Zynq-7000, accompagné d'une étude de l'architecture de ce type de circuit dans le but de fournir une clarification sur la façon dont les SoCs intégrés se mappent vers les périphériques Zynq à base du processeur ARM. En dernier lieu, nous avons présenté la carte de prototypage utilisée pour l'implémentation de notre crypto système ECC, à savoir la carte ZedBoard.

Dans le chapitre suivant nous allons présenter la méthodologie développée pour la réalisation de notre crypto système embarqué.

Chapitre 3

Implémentation de la plateforme de chiffrement / Déchiffrement

---

3.1. Introduction

L'objectif principal de notre travail consiste en la réalisation d'un crypto système ECC basé sur la carte de prototypage ZedBoard. Pour cela, nous avons proposé une plateforme de chiffrement et de déchiffrement, composée de deux parties. La première est une Interface Homme/Machine (IHM), développée en langage Java. La seconde est constituée par des fonctions implémentées en langage C. Ces fonctions sont exécutées par la partie PS, en l'occurrence le processeur ARM Cortex A9 (0) du circuit FPGA Zynq.

Ce chapitre est consacré à la présentation de la plateforme proposée, des rôles attribués aux deux parties qui constituent cette plateforme, de leur implémentation et de leur fonctionnement.

3.2. Description de la plateforme de chiffrement/déchiffrement ECC

La plateforme réalisée est composée de deux parties comme cité précédemment. Ces dernières sont liées par un canal de communication. Son architecture est montrée sur la figure 3.1.

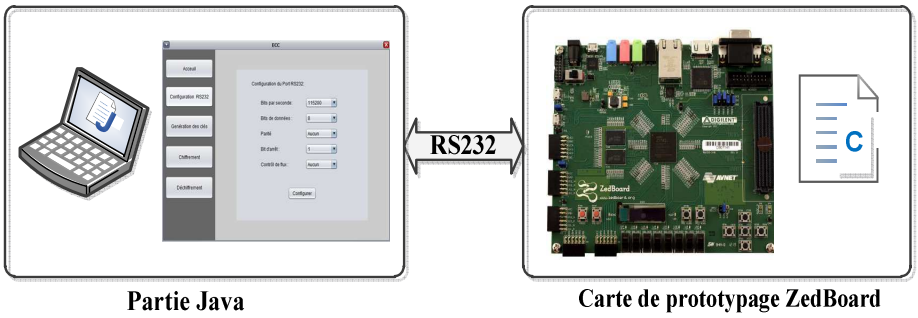


Figure 3.1 - Plateforme de chiffrement et de déchiffrement à base de crypto système ECC

✓ **Le canal de communication**

Le canal de communication entre les deux parties est basé sur le protocole RS232. Ce denier est chargé de transférer toutes les données de l'ordinateur vers la carte FPGA ou la réception des résultats calculés par cette carte.

✓ **La carte de prototypage**

La carte de prototypage utilisée est la carte ZedBoard. Elle est constituée d'un circuit FPGA de la famille Zynq-7000, en l'occurrence le circuit XC7Z020-1CSG484CES EPP [26]. Ce circuit a pour rôle d'exécuter les opérations du chiffrement, du déchiffrement et de la multiplication scalaire.

✓ **La partie Java**

Cette partie est une interface réalisée en langage java, exécutée sur un ordinateur. Celle-ci permet une utilisation flexible de la partie embarquée sur circuit FPGA et sa configuration. Les principales tâches exécutées sont définies ci-dessous:

- Configuration des paramètres du protocole RS232.
- Génération des clés publique et privée utilisées respectivement pour le chiffrement et le déchiffrement.
- Décomposition des données sous forme de suite d'octet.
- Transmission des données vers le circuit FPGA.
- Réception et affichage des résultats du chiffrement ou du déchiffrement.

Le fonctionnement de la plateforme réalisée est résumé comme suit :

Après la configuration du protocole RS232 et la génération des clés, la partie Java intervient par la lecture de la clé publique ou privée du destinataire. Une fois celle-ci est obtenue, les données nécessaires pour le chiffrement/déchiffrement sont décomposées sur des données de taille 8-bits et transmises vers la carte FPGA. Dans le cas du chiffrement, ces données sont constituées par la valeur du scalaire k, des coordonnées du point générateur G, de la clé publique Pb et du message M en clair. Si l'opération considérée est un déchiffrement, ces données sont constituées par la clé privée nb et les coordonnées de la multiplication scalaire $k \times G$ et du message chiffré. La configuration du système embarqué pour le chiffrement ou le déchiffrement est réalisée grâce à une entrée supplémentaire transmise à partir de la partie Java.

Les paramètres de la courbe utilisée correspondent au standard secp256r1 [19]. Ce dernier assure un niveau de sécurité de 256 bits. Les valeurs des paramètres de la courbe a, b, n, N et G sont définis dans le chapitre suivant.

Après la réception des données par le système embarqué, l'algorithme de chiffrement ou de déchiffrement sera exécuté. Une fois l'exécution terminée sur le circuit FPGA, le résultat sera transmis vers l'ordinateur.

La complexité de l'exécution des opérations de chiffrement et de déchiffrement du crypto système ECC est liée principalement à la partie embarquée sur circuit FPGA. Celle-ci repose sur le calcul de l'opération de base, à savoir, la multiplication scalaire. Dans ce qui suit, nous allons présenter l'implémentation de la partie embarquée. Puis nous décrivons la partie Java.

3.3. Architecture matérielle du crypto système embarqué

L'architecture matérielle de la partie embarquée sur circuit FPGA est montrée sur la figure 3.2. Celle-ci a été configurée et implémenté en utilisant l'outil Vivado 2015.4 de Xilinx.

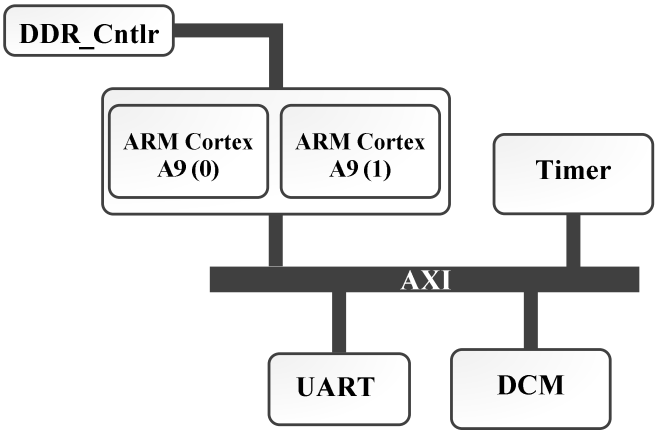


Figure 3.2 - Architecture matérielle du système embarqué.

Cette partie est composée de :

- Un processeur à double cœur ARM Cortex A9 (0) et ARM Cortex A9 (1).
- Une DCM (Digital Clock Manager).
- Un contrôleur de la mémoire DDR.
- Un bus système AXI (Advanced eXtensible Interface (AXI).
- Une UART.
- Un timer.

L'UART permet la communication avec le port RS232 de la carte FPGA. Le rôle du Timer est d'évaluer les performances d'exécution des opérations du chiffrement et du déchiffrement.

3.4. Implémentation de la partie logicielle du système embarqué

La partie logicielle du système embarqué réalisée a été implémentée en utilisant l'outil SDK 2015.4 (Software Design Kit) de Xilinx. Cette partie est constituée par des fonctions décrites en langage C. Elles sont stockées dans la mémoire DDR de la carte de prototypage ZedBoard, puis exécutées par le processeur ARM Cortex A9 (0).
Les opérations attribuées au processeur sont définies comme suit :

- Réception des données d'entrées.
- Exécution de l'opération du chiffrement/déchiffrement.
- Exécution des algorithmes des opérations de base à savoir : la multiplication scalaire (incluant l'addition de deux points et le dédoublement d'un point), l'addition, la soustraction, la multiplication et l'exponentiation modulaires.
- Transmission du résultat vers l'ordinateur pour le stockage et leur affichage.

Les fonctions développées pour réaliser le chiffrement ou le déchiffrement sont hiérarchisées sur cinq niveaux d'abstractions. L'organisation de ces fonctions est montrée dans la figure 3.3.

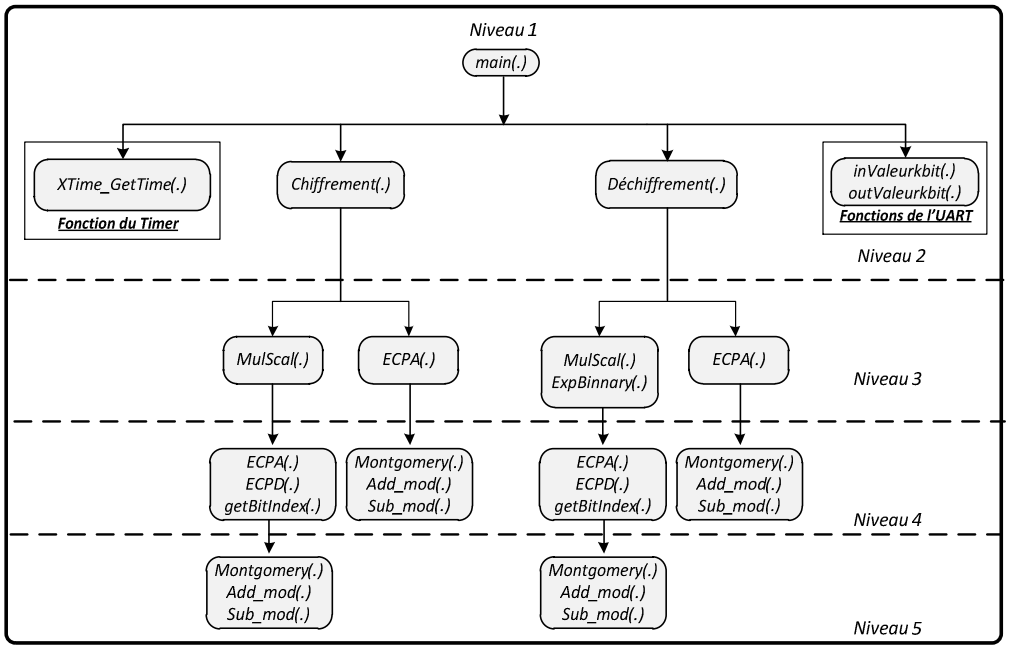


Figure 3.3 - Organisation des fonctions des parties logicielles de la partie embarquée sur circuit FPGA.

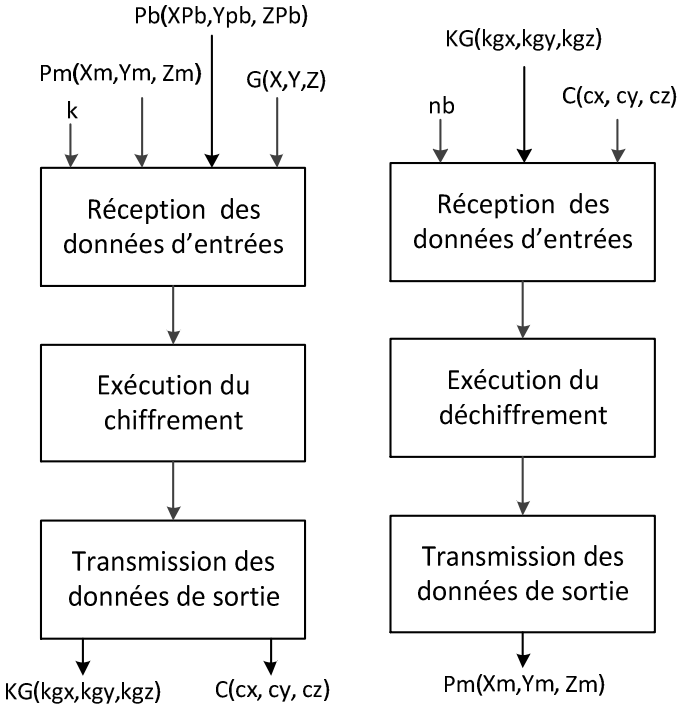
Le programme principale Main(.) est décrit en langage C. Il contient les fonctions de communication avec l'UART, les fonctions du Timer et les fonctions du chiffrement et du déchiffrement, définies respectivement par $Chiffrement\,(.)$ et $Déchiffrement\,(.)$. Le code C du programme principal $Main(.)$ est représenté en annexe sur la figure 1.

3.4.1. Exécution d'un chiffrement/déchiffrement

Le fonctionnement de la partie embarquée sur circuit FPGA est exécuté en trois étapes :

- Réception des données d'entrées.
- Exécution des fonctions du chiffrement/déchiffrement.
- Transmission du résultat.

La figure 3.4 illustre ces trois étapes .



(a) Etapes d'exécution du chiffrement.

(b) Etapes d'exécution du déchiffrement.

Figure 3.4 - Etapes d'exécution du chiffrement et de déchiffrement.

3.4.1.1. Réception des données d'entrées

Après l'initialisation de toutes les variables intermédiaires, les données transmises par la partie Java sont stockées dans la mémoire DDR de la carte ZedBoard sous forme

de chiffres de taille 32 bits. La réception des données est basée sur la fonction $inValeurkBit\ (.)$. Son code C est représenté dans la figure 3.5.

```
inline unsigned long inValeurkBit(){
       unsigned long a=0,tmp;int i;
       for(i=0;i<=3;i++){
                     tmp=XUartPs_RecvByte(0xe0001000)<<(8*i);
                     a|=tmp;}
       return a;}
```

Figure 3.5 - Code C de la fonction $inValeurkBit()$

A l'origine, la liaison RS232 entre la carte et l'IHM impose qu'à chaque trame de donnée transmise par l'ordinateur seulement 8 bits sont actifs. De ce fait, la fonction $inValeurkBit\ (.)$ permet de reconstruire des données codées sur 32 bits, à partir des données d'entrées codées sur 8 bits, son implémentation est basée sur :

- Une fonction assurant la transmission vers le processeur ARM des 8 bits reçus à travers l'UART. La fonction en question est fournie par Xilinx. Cette dernière est définie par :$XUartPs\_RecvByte(.)$.
- Des décalages à gauche de 8 bits, suivant un indice i.

3.4.1.2. Exécution des fonctions du chiffrement/déchiffrement

Ces fonctions sont basées sur la multiplication scalaire et l'addition de deux points. Selon une entrée de sélection, le processeur embarqué exécute l'une des opérations de chiffrement ou de déchiffrement. Cette entrée est notée par sel dans le programme principal $main(.)$.

a. Chiffrement :

La fonction $Chiffrement(.)$ permet d'exécuter un chiffrement sur les cordonnées (Xm,Ym, Zm) du message Pm. L'exécution de cette opération correspond à l'exécution de l'équation (1.20). Le résultat de cette dernière est un couple de point $P_c = [(k \times G, (P_m + k \times P_b)]$ tel que $kG = (kgx, kgy, kgy)$ et $(P_m + kP_b) = (cx, cy, cz)$. La fonction $Chiffrement(.)$ comprend les fonctions $mul\_scal(.)$ et $ECPA(.)$. La fonction $mul\_scal(.)$ permet de calculer les deux multiplications scalaires $k \times G$ et $k \times P_b$. Elle prend en entrée les coordonnées(X, Y, Z) du point G, les cordonnées(Xpb, Yby, ZPb) du point P_b qui représente la clé publique, le scalaire$k$ et les cordonnées (Xm, Ym, Zm) du message Pm. La figure 3.6 représente l'organigramme de l'opération du chiffrement.

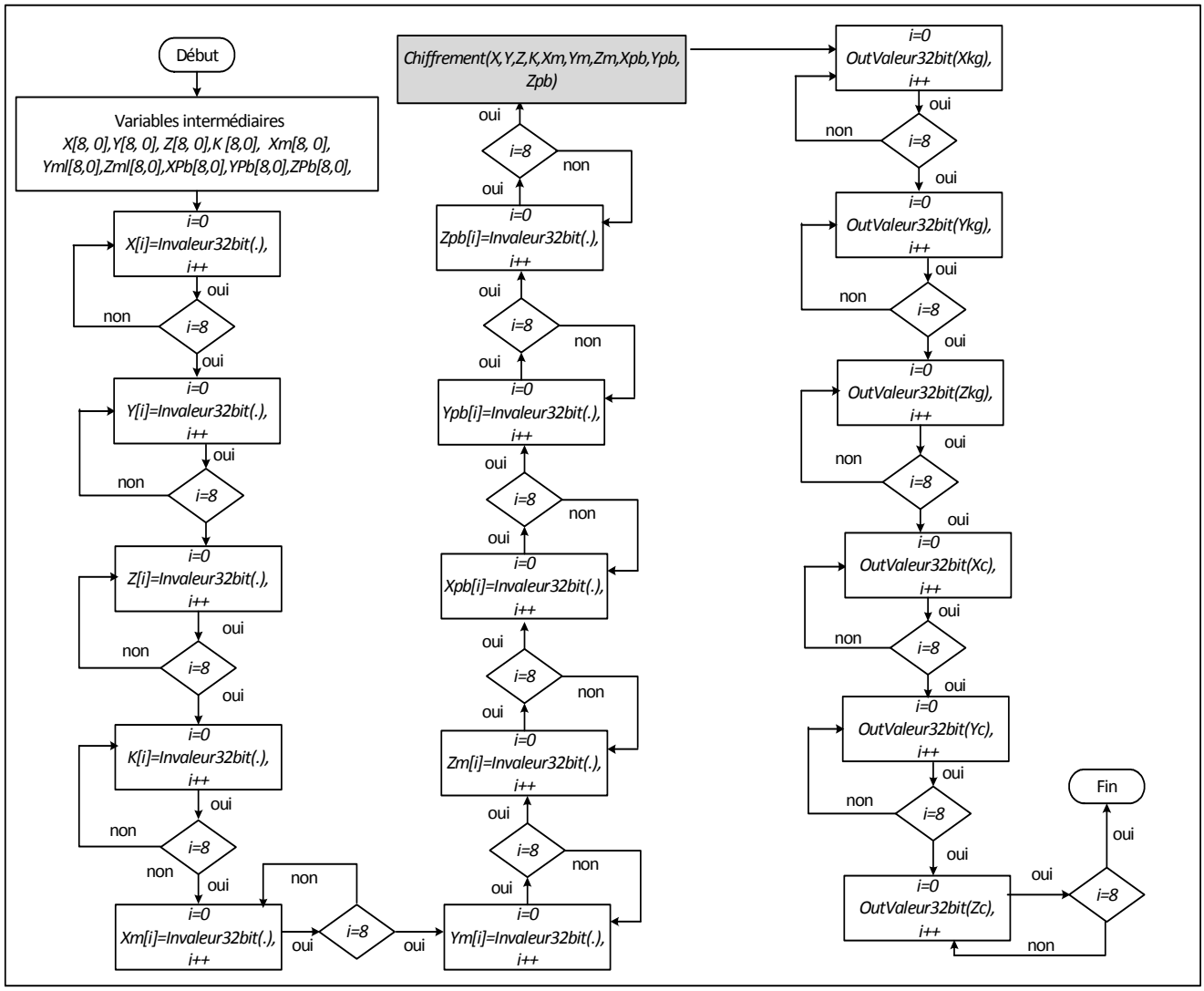


Figure 3.6 - L'organigramme du chiffrement.

b. Déchiffrement :

La fonction $Dechiffrement(.)$ permet de restituer les cordonnées (Xm, Ym, Zm) du message M en clair à partir des coordonnées (cx, cy, cz) du message chiffré. Elle reçoit en entrée (cx, cy, cz) ,(kgx, kgy, kgz) et la clé privée nb. La fonction $Dechiffrement(.)$ comprend les fonctions $mul\_scal(.)$, $ECPA(.)$ et $ExpBinarrybase32(.)$. La fonction $mul\_scal(.)$ permet de calculer la multiplication scalaire $nb \times (k \times G)$ de l'équation (1.21). La fonction $ECPA(.)$ permet de calculer l'addition $(P_m + k.P_b) - n_b(k.G)$ de l'équation (1.21). $ExpBinarry(.)$ assure la conversion vers le plan affine des coordonnées du résultat de déchiffrement, obtenu dans le plan projectif. La figure 3.7 représente l'organigramme de l'opération du déchiffrement.

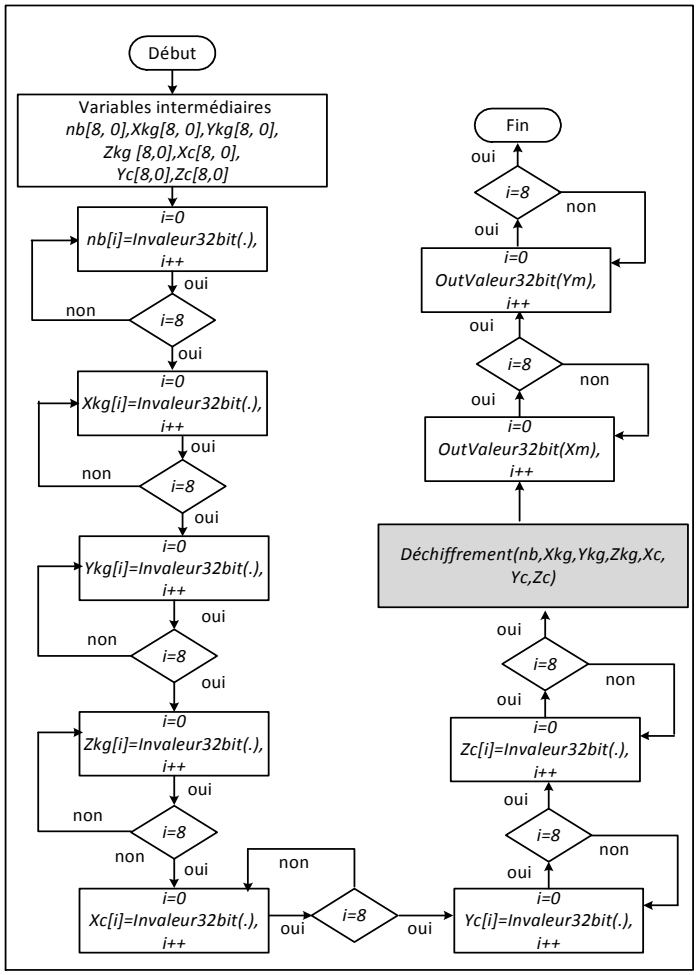


Figure 3.7 - Organigramme du déchiffrement

3.4.1.3. Transmission du résultat

Durant cette étape, le résultat du chiffrement ou du déchiffrement est transmis vers la partie Java. Elle est basée sur la fonction $OutValeur32bits(.)$. Celle-ci assure la décomposision du résultat sur des mots de taille 8 bits. Le code C de cette fonction est montré sur la figure 3.8.

```
inline void outValeur32Bit(unsigned long valeur)
{
int i;
for(i=0;i<=3;i++)
XUartPs_SendByte (0xe0001000,valeur>>(8*i));
}
```

Figure 3.8 - Code C de la fonction $outValeurkBit(.)$

L'implémentation de cette dernière est basée sur :

- Des décalages à droite, suivant l'indice *i*.
- La fonction $XUartPs\_SendByte(.)$, fournie par Xilinx.

3.4.2. Fonction de la multiplication scalaire *mul_scal(.)*

La fonction $mul\_scal(.)$ est utilisée dans le chiffrement ainsi que dans le déchiffrement. Elle permet de calculer l'opération de base $Q = k \times P$, tel que P est un point et k est un scalaire. Cette fonction est exécutée sur le processeur ARM Cortex A9 (0) selon les étapes de l'algorithme 1.8, à savoir :

- Décalage à droite du i^{éme} chiffre du scalaire k.
- Tester le i^{éme} bit de $k[i]$ selon la ligne 8 de cet algorithme.
- Effectuer l'opération d'addition et de dédoublement de points.
- Stocker le résultat dans la variable de sortie Q.

Le code C de la fonction $mul\_scal(.)$ est montré en annexe sur la figure 2. Elle est constituée par les fonctions: $ECPA(.)$ et $ECPD(.)$. Ces dernières assurent le calcul d'une addition et d'un dédoublement de points. En plus de ces fonctions, le code C de $mul\_scal(.)$ comprend aussi une fonction qui permet d'effectuer les décalages du scalaire k. Celle-ci est nommée $getBitIndex1(.)$. Son organigramme est montré sur la figure 3.9.

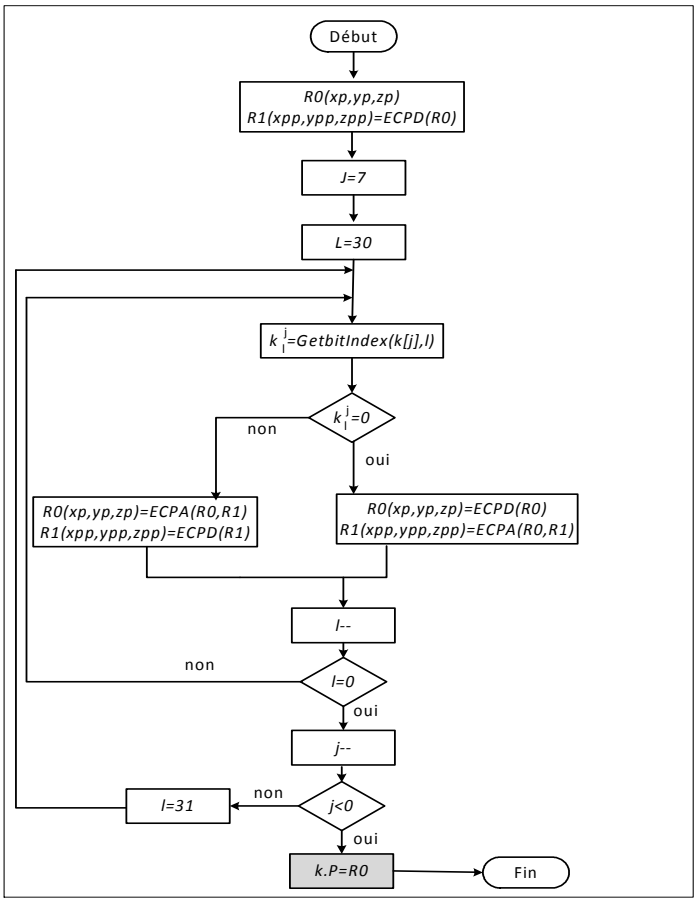


Figure 3.9 - Organigramme de la fonction $mul\_scal(.)$

- $\boldsymbol{ECPA}(.)$ permet l'exécution de l'addition deux points. Elle est constituée par les sous fonctions $Montgomery(.)$, $Add\_Mod(.)$ et $Sub\_Mod(.)$. Ceci est montré sur la figure 3.10.

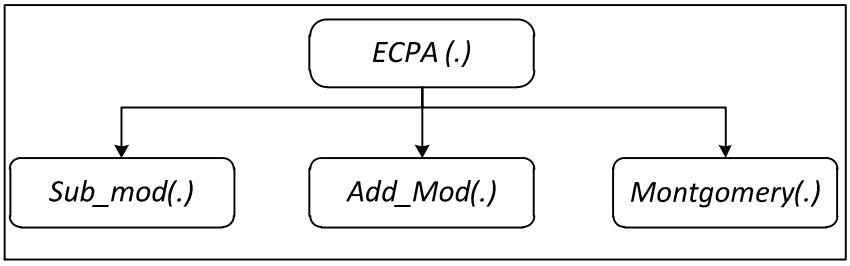


Figure 3.10 - Organisation de la fonction $ECPA(.)$

- $\boldsymbol{ECPD}(.)$ permet le calcul d'un dédoublement de point. Elle est constituée des sous fonctions $add\_mod(.)$, $Sub\_Mod(.)$ et $Montgomery(.)$ comme montré sur la figure 3.11.

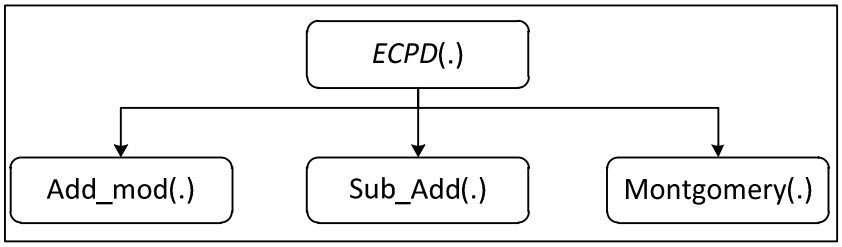


Figure 3.11 - Organisation de la fonction $ECPD(.)$

- $\boldsymbol{GetBitIndex1}(.)$ assure les décalages des i^{eme} chiffres du scalaire k. Le code C de cette fonction est montré sur la figure 3.12.

```
Xuint32 getBitIndex1(Xuint32 num,int index){
Xuint32 bit=(num>>index)&1;
return bit;
}
```

Figure 3.12 - Le Code C de la fonction GetBitIndex1(.)

3.4.3. Fonctions d'exécution des opérations arithmétiques modulaires

Dans la partie précédente, nous avons présenté l'implémentation en langage C de l'opération cœur du chiffrement et du déchiffrement, à savoir la multiplication scalaire. Dans ce qui suit, nous allons présenter les fonctions permettant l'exécution des opérations arithmétiques du crypto système ECC , en l'occurrence $expBinary(.)$, $montgomery(.)$, $add\_mod(.)$ et $sub\_mod(.)$.

- ***expBinary***(.) correspond à l'implémentation en langage C de l'algorithme 1.7. Elle permet le calcul de l'exponentiation modulaire $C = X^y mod\ N$. Cette fonction est constituée par les deux fonctions, $getBitIndex1(.)$ permettant d'effectuer des décalages à droite sur l'exposant et la fonction $montgomery(.)$ assurant l'exécution itérative de la MMM de manière logicielle. Son organisation est montrée sur la figure 3.13.

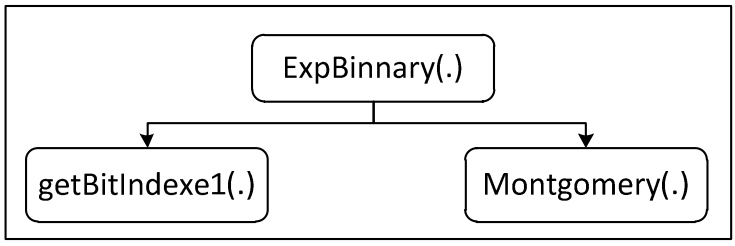


Figure 3.13 - Organisation de la fonction $expBinnary(.)$

- ***montgomery***(.) permet d'effectuer la multiplication modulaire de deux entiers A et B. Elle correspond à la description en langage C de l'algorithme 1.6. Cette dernière comprend les fonctions $multiply(.)$ et $add3Op(.)$. Le code C de la fonction $montgomery(.)$ est définit sur la figure 3.14.

```
void montgomery(unsigned long A[9],unsigned long B[9],unsigned long
N[9],unsigned longm,unsigned long Resultat[9]) {
int Z=0,P=0,C1=0,C2=0,L=0,i,j,ii;
       int q=0,retenue1=0,retenue2=0,tmp=0;
       for(i=0;i<=8;i++)S[i]=0;
for(i=0;i<=8;i++){// debut boucle i
C1=0;C2=0;retenue1=0;retenue2=0;tmp=0;//initialisation des retenu
dans chaque iterationP=A[i]*B[0];
Z=P+S[0];
  q=(Z*m);
    //*****calcul de S=A[i]*B +q*N /R
              for(j=0;j<=8;j++)
       {// debut boucle j
              tmp=C1; // save la valeur de C1
       multiply(A[i],B[j],&C1,&P); //calcul P=A[i]*B[j]de 32 bits et
C1=retenu de 32bit
       add3Op(P,S[j],tmp,&retenue1,&Z);
       tmp=C2;// save la valeur de C2
              multiply(q,N[j],&C2,&P);
       add3Op(Z,P,tmp,&retenue2,&P);
              if(j!=0)S[j-1]=P;}// fin boucle j
       S[j-1]=C1+C2+retenue1+retenue2;}
for(i=0;i<=8;i++)Resultat[i]=S[i];}
```

Figure 3.14 - Code C de la fonction $montgomery(.)$

- ***add_mod***(.) permet de calculer l'addition modulaire des deux entiers A et B. Elle correspond à l'implémentation en langage C de l'algorithme 1.3. Cette dernière comprend les sous fonctions élémentaires $add2Op(.)$, $sous2Op(.)$ et

$getBitIndex1(.)$. Le code source de la fonction $add\_mod(.)$ est présenté sur la figure 3.15.

```
voidadd_mod(Xuint32 A[9],Xuint32 B[9],Xuint32
N[9],Xuint32 *SZ[9]){
       Xuint32 SZ1[9],SZ2[9];Xuint32 i,C1=0,C2=1,sign;
       for(i=0;i<9;i++){
              add2Op(A[i],B[i],&C1,&SZ1[i]);
              sous2Op(SZ1[i],N[i],&C2,&SZ2[i]);}
       sign=getBitIndex1((C1+C2+3),1);
       if(sign==1)setIN(SZ1,SZ);
       elsesetIN(SZ2,SZ);}
```

Figure 3.15 - Code C de la fonction add_mod(.)

- $\boldsymbol{sub\_mod}(.)$ permet d'effectuer la soustraction modulaire des deux entiers A et B.

Cette dernière comprend les sous fonctions élémentaires $sous2Op(.)$, $add2Op(.)$ et $getBitIndex1()$.Le code source de la fonction $sub\_mod(.)$ est montré sur la figure 3.16.

```
voidsub_mod(Xuint32 A[9],Xuint32 B[9],Xuint32
N[9],Xuint32 *SZ[9]){
       Xuint32 SZ1[9],SZ2[9];
       Xuint32 i,C1=1,C2=0,sign;
       for(i=0;i<9;i++){
              sous2Op(A[i],B[i],&C1,&SZ1[i]);
              add2Op(SZ1[i],N[i],&C2,&SZ2[i]);
              }
       sign =getBitIndex1((C1+3),0);
       if(sign==1)setIN(SZ2,SZ);
       elsesetIN(SZ1,SZ); }
```

Figure 3.16 - Code C de la fonction $sub\_mod(.)$

3.5. Partie Java du crypto système ECC

Cette partie a été développée dans le but de :

- Vérifier le fonctionnement de la partie embarquée sur circuit FPGA.
- Utiliser le crypto système ECC que nous avons développé pour le chiffrement et le déchiffrement des données.

L'implémentation de la partie en question a été réalisée sous l'environnement NetBeans. Son rôle est résumé dans les points suivants :

1. Configuration du protocole de communication RS232.
2. Génération des clés du chiffrement et du déchiffrement.
3. La décomposition et conversion du message clair en un point.
4. Transmission des données pour leur chiffrement ou leur déchiffrement vers la carte FPGA.

5. Réception des résultats issus de la carte FPGA.
6. Conversion du point résultant du déchiffrement en un message clair.

L'organigramme du programme principal de la partie Java est montré sur la figure 3.17. Il est constitué par des fonctions ou chacune lui attribué un des rôles présentés ci-dessus.

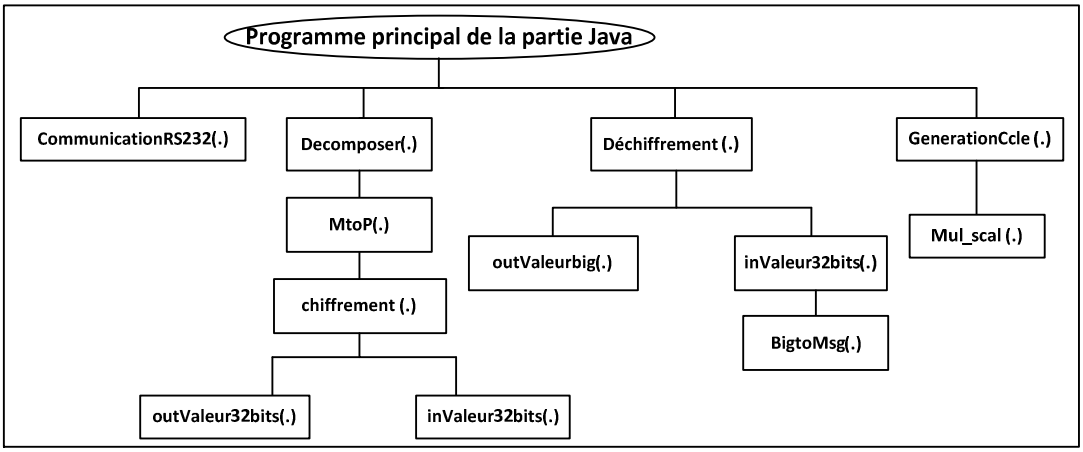


Figure 3.17 - Organisation du programme principal de la partie Java

3.5.1. Configuration du protocole de communication RS232

La configuration du protocole RS232 permet d'ouvrir la liaison de communication entre l'ordinateur et la carte de prototypage utilisée, à savoir la carte ZedBoard. Cette opération est basée sur la définition des paramètres suivants :

- Débit de transmission des données.
- La taille des données à transmettre.
- Le bit de parité.
- Le bit d'arrêt.
- Le control de flux.

Ces paramètres doivent être configurés de la même manière que ceux de l'UART du système embarqué. Le code Java de la configuration du port RS232 est défini dans la fonction CommunicationRS232(.). Le code en question est présenté sur la figure 3.18.

```
public CommuntionRS232(String nomPort,int debit,int donnee,String parite,int arret,String flux){
        Win32Driver w32Driver= new Win32Driver();
        w32Driver.initialize();
Enumeration portList=CommPortIdentifier.getPortIdentifiers();
        while (portList.hasMoreElements()){
        portId=(CommPortIdentifier)portList.nextElement();
        System.out.println(portId.getName());}
portId = null;
try{                            portId=CommPortIdentifier.getPortIdentifier("COM3");
                        SerialPort port;
                                port=(SerialPort)portId.open("Mon_Appli", 10000);
                                int p1;
                                if(parite.equals("Aucun"))p1=SerialPort.PARITY_NONE;
                                else if(parite.equals("Pair"))p1=SerialPort.PARITY_EVEN;
                                else p1=SerialPort.PARITY_MARK;
                                int p2 = 0;
                                if(flux.equals("Aucun"))p2=SerialPort.FLOWCONTROL_NONE;
                                else p2=SerialPort.FLOWCONTROL_RTSCTS_IN;
                                port.setFlowControlMode(p2);
                                port.setSerialPortParams(debit, donnee,arret, p1);
                                in=new BufferedInputStream(port.getInputStream());
                         out=new BufferedOutputStream(port.getOutputStream());
                        } catch (PortInUseException ex) {
                                //traitement de l'exception
                        } catch (NoSuchPortException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        } catch (UnsupportedCommOperationException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        } catch (IOException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }}
```

Figure 3.19 - Code Java de la fonction CommunicationRS232(.)

3.5.2. Génération des clés du chiffrement et du déchiffrement

La génération des clés est exécutée suivant l'algorithme 1.1. La taille des clés qui définit le niveau de sécurité, est choisi suivant le standard sec256 sur 256 bits [19]. Le programme Java de cette partie est basé sur la fonction $KeyGeneration(.)$. Celle-ci permet de générer la clé privée nb, telle que $nb < n$, où n est l'ordre du point générateur G. La clé publique Pb est calculée par la multiplication scalaire : $P_b = nb \times G$. La génération de la clé privée est basée sur la class Random. Celle-ci permet de générer un nombre aléatoire. Le résultat obtenu est de type BigInteger muni d'un paramètre indiquant la taille exacte du résultat. Le programme java de cette partie est montré sur la figure 3.20.

```
public  BigIntegergetRandomBigInteger(){
        Random rand=new Random();
BigInteger result= new BigInteger(256, rand);
return result; }
    Cle Generation(){
BigIntegernb=getRandomBigInteger();
while(nb.max(n)==nb){
nb=getRandomBigInteger();}
ReglesAdditionrr=new ReglesAddition();
          Point p;
          p=rr.multiScalairee(G, a, nb, modulo);
         Cle c=new Cle(nb,p);
return c;}
```

Figure 3.20 - Code java de la génération des clés

3.5.3. Décomposition et conversion du message clair en points

Cette étape peut être considérée comme une étape de prétraitement où un message en clair M de taille variable est décomposé et converti en points $Pm(i)$ avant qu'il soit transmis vers le circuit FPGA. Elle est effectuée en trois phases comme montré sur la figure 3.21.

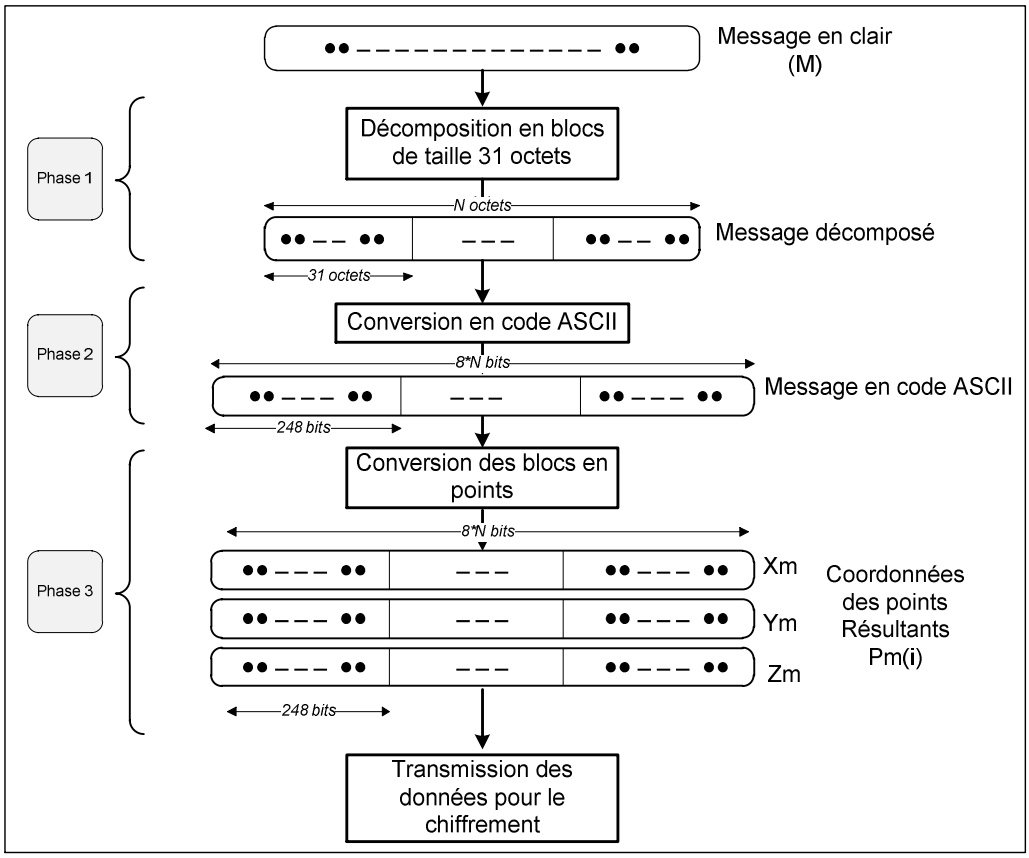


Figure 3.21 - Etapes de décomposition et de conversion d'un message en clair

Les trois phases de cette étape sont définies comme suit :

- La première phase correspond à la décomposition du message en blocs de taille 31 octets. Cette opération est nécessaire car le protocole ECC exige que la valeur numérique du message à chiffrer soit inférieure à la valeur du modulo p. Cette phase de décomposition est réalisée par la fonction $decomposer(.)$.
- La deuxième phase consiste à convertir chacun des blocs obtenus en code ASCII, puis stocker le résultat dans une variable définie comme un BigInteger de taille égale à celle du message d'origine. La taille des blocs obtenus en termes de nombre de bits est de 248 bits. Cela se fait par l'utilisation de la fonction $MsgToBig(.)$.
- La troisième phase consiste à générer les coordonnées (Xm, Ym, Zm) des points $Pm(i)$ qui correspondent à chaque bloc. Cette étape est réalisée par la fonction $MtoP(.)$.

Les codes Java de la décomposition et de la conversion du message sont montrés respectivement sur les figures 3.22 et 3.23.

```
public class Decomposition {
    public static ArrayList<String> Decomposer (String message){
     ArrayList<String> Mi = new ArrayList<>() ;
         int w=0;
         do {
        if((w+31)<message.length())
        Mi.add(message.substring(w, w+31));
        else
        Mi.add(message.substring(w, message.length()));
        w=w+31;
    }while(w<=message.length());
    return M; }
}
```

Figure 3.22 - Code java de la décomposition du message en clair

```
// function de conversion en code ASCII
public BigInteger MsgToBig(String message) throws
UnsupportedEncodingException {
               return new BigInteger(message.getBytes("us-ascii"));
       }
              public BigInteger GetPoint(BigInteger x{
              BigInteger
y2=(x.pow(3).add(a.multiply(x)).add(b)).mod(p);
                              return y2; }
// function de conversion en point
public Point MtoP(BigInteger msg,BigInteger k){
              BigInteger j=BigInteger.ZERO;
              BigInteger msgx=BigInteger.ZERO,
                 Zi=BigInteger.ZERO;
              while(j.min(k)!=k) {
                     msgx=(msg.multiply(k)).add(j);
                     Zi=GetPoint(msgx);

                     if(IsSqrtRoot(Zi))
                            j=k;
                     else{
                            j=j.add(BigInteger.ONE); }}
                             msgy= sqrt(Zi)
              return new Point(msgx,msgy); }
```

Figure 3.23 - Code java de conversion d'un message en code ASCII et en point

3.5.4. Transmission vers la carte FPGA des données nécessaires pour le chiffrement ou déchiffrement

Avant d'entamer un chiffrement ou un déchiffrement de données, la partie Java transmet vers la carte FPGA une donnée de sélection. Celle-ci permet de configurer la partie embarquée pour réaliser l'une des deux opérations.

Une fois que les étapes de prétraitement sur le message à chiffrer sont effectuées, toutes les données nécessaires au chiffrement sont transmises vers la carte FPGA. Ces données sont constituées de :

La clé publique Pb, le point du message $Pm(i)$ (Xm, Ym, Zm) associé à chaque bloc de donnée, le point générateur de la courbe $G(X, Y, Z)$ et la valeur aléatoire du paramètre k. La transmission de ces données pour le chiffrement est basée sur les deux fonctions $chiffrement(.)$ et $outValeurbig(.)$. Leurs codes Java sont présentés respectivement sur les figures 3.24 et 3.25.

```
public class chiffrement {
    String ch1x,ch1y,ch1z,chx,chy,chz;
    public void chiffrement(byte action,CommuntionRS232 CR ,
    BigInteger Xg,BigInteger Yg,BigInteger Zg, BigInteger  k,BigInteger
    xpb,BigInteger ypb, BigInteger zpb,
    BigInteger xpm,BigInteger ypm, BigInteger zpm,BigInteger res1x,
    BigInteger res1y,
    BigInteger res1z,
    BigInteger resx, BigInteger resy, BigInteger resz){
        CR.outValeur8Bit(action);
        CR.outValeurbig(Xg, 256);
        CR.outValeurbig(Yg, 256);
        CR.outValeurbig(Zg, 256);
        CR.outValeurbig(k, 256);
        CR.outValeurbig(xpm, 256);
        CR.outValeurbig(ypm, 256);
        CR.outValeurbig(zpm, 256);
        CR.outValeurbig(xpb, 256);
        CR.outValeurbig(ypb, 256);
        CR.outValeurbig(zpb, 256);
 ch1x="";
 ch1y="";
 ch1z="";
 chx="";
 chy="";
 chz="";
for(int i=0;i<8;i++){
res1x=new BigInteger(Integer.toHexString(CR.inValeur32Bit()),16);
ch1x= Chif_dechi_ecc.test_paquet_complet(res1x.toString(16))+ch1x;
}
System.out.println("x de k*G est " +ch1x);
for(int i=0;i<8;i++){
res1y=new BigInteger(Integer.toHexString(CR.inValeur32Bit()),16);
ch1y= Chif_dechi_ecc.test_paquet_complet(res1y.toString(16))+ch1y;
}
System.out.println("y de k*G est " +ch1y);
for(int i=0;i<8;i++){
res1z=new BigInteger(Integer.toHexString(CR.inValeur32Bit()),16);
ch1z=Chif_dechi_ecc.test_paquet_complet(res1z.toString(16))+ch1z;
}
System.out.println("z de k*G estt " +ch1z);
for(int i=0;i<8;i++){
resx=new BigInteger(Integer.toHexString(CR.inValeur32Bit()),16);
chx= Chif_dechi_ecc.test_paquet_complet(resx.toString(16))+chx;
}
System.out.println("x de k*pb+pm est " +chx);
for(int i=0;i<8;i++){
resy=new BigInteger(Integer.toHexString(CR.inValeur32Bit()),16);
chy= Chif_dechi_ecc.test_paquet_complet(resy.toString(16))+chy;
}
System.out.println("y de k*pb+pm est "  +chy);
for(int i=0;i<8;i++){
resz=new BigInteger(Integer.toHexString(CR.inValeur32Bit()),16);
chz=Chif_dechi_ecc.test_paquet_complet(resz.toString(16))+chz;
}
//resz=new BigInteger (chz,16);
System.out.println("z de k*pb+pm est "  +chz);
    }
}
```

Figure 3.24 - Code java de la fonction de chiffrement

```
public void outValeurbig(BigInteger valeur,int taille) {
                BigInteger big=(new
BigInteger("2",10)).pow(taille+1);
                BigInteger v=valeur.add(big);
                byte[] valeurS=v.toByteArray();
                byte b=0;
                int t=taille/8;
                int a,c;
                if(valeurS.length==t+1){a=t;c=1;}
                else {a=t-1;c=0;}
                try {
                    (valeur>>(8*i));
                       for(int i=a;i>=c;i--){
                           if(i<valeurS.length)
                           out.write(valeurS[i]);
                           else
                           out.write(b); }
                      out.flush();
                        } catch (IOException e) {
                    e.printStackTrace();}}
```

Figure 3.25 - Code java de la fonction $outValeurbig(.)$

Les résultats de l'opération du chiffrement sont dans le disque dur de l'ordinateur dans des fichiers textes, ils seront transmis par la suite vers la carte pour le déchiffrement. Les données nécessaires pour restituer le message en clair sont constituées de :

- Les points C (cx, yx, zx) où les blocs de messages chiffrés sont embarqués.
- Le point KG (kgx, kgy, kgz) résultant de la multiplication scalaire $k \times G$.
- La clé privée nb.

La transmission de toutes ces données pour le déchiffrement est basée sur les fonctions $dechiffrement(.)$ et $outValeurbig(.)$. Leurs codes Java sont présentés respectivement sur les figures 3.26 et 3.25.

```
import java.math.BigInteger;
public class dechiffrement {
    String  chxf , chyf;
    public void dechiffrement(byte action,CommuntionRS232 CR ,
    BigInteger nb,BigInteger res1x, BigInteger res1y, BigInteger
    res1z, BigInteger resx, BigInteger resy, BigInteger resz,
    BigInteger xf, BigInteger yf)
    {
    CR.outValeur8Bit(action); CR.outValeurbig(nb, 256);
    CR.outValeurbig(res1x, 256); CR.outValeurbig(res1y, 256);
    CR.outValeurbig(res1z, 256);    CR.outValeurbig(resx, 256);
    CR.outValeurbig(resy, 256);  CR.outValeurbig(resz, 256);
    chxf="";
    chyf="";
    for(int i=0 ; i<8 ; i++){
    xf=new BigInteger(Integer.toHexString(CR.inValeur32Bit()),16);
    chxf=Chif_dechi_ecc.test_paquet_complet(xf.toString(16))+chxf;
    }
    System.out.println("x final est: " +chxf);
      for(int i=0 ; i<8 ; i++){
    yf=new BigInteger(Integer.toHexString(CR.inValeur32Bit()),16);
    chyf=Chif_dechi_ecc.test_paquet_complet(yf.toString(16))+chyf;
      }
      System.out.println("y final est: " +chyf);
    }
    }
```

Figure 3.26 - Code java de la fonction de déchiffrement

3.5.5. Réception des résultats issus de la carte FPGA

La réception des résultats de chiffrement et de déchiffrement est basée sur la fonction $inValeur32bits()$. Son code java est montré sur la figure 3.27.

```
public int inValeur32Bit(){
     int valeur=0;//byte b;
      for(int i=0;i<=3;i++)
        try {valeur |=in.read()<<(i*8);}
         catch (IOException e) {
          e.printStackTrace();}
       return valeur;}
```

Figure 3.27 - Code java de la fonction de réception $inValeur32bits()$.

3.5.6. Conversion du point résultant du déchiffrement en un message en clair

Le résultat du déchiffrement obtenu de la carte est représenté dans le plan affine par : $Pm(xm, ym)$. Sa conversion en un message en clair est effectuée en utilisant la fonction $BigToMsg(.)$. Le code java de celle-ci est montré sur la figure 3.28.

```
public static String BigToMsg(BigInteger big){
return new String(big.toByteArray());}
```

Figure 3.28 - Code java de la conversion d'un point en message en alpha numérique.

3.6. Conclusion

Dans ce chapitre, nous avons présenté une approche d'implémentation purement logicielle du crypto système ECC. La plateforme proposée est basée sur la conception de deux parties. La première est associée à l'étape de génération des clés. Celle-ci est implémentée en langage Java. La seconde correspond à la mise en œuvre de la multiplication scalaire sur la carte de prototypage ZedBoard. Le chapitre suivant sera consacré à la présentation des résultats d'implémentation sur circuit FPGA et de l'IHM développée pour la vérification fonctionnelle de notre plateforme.

Chapitre 4

Résultats d'implémentation et présentation de l'IHM

4.1 Introduction

Dans le chapitre qui a précédé, nous avons présenté l'implémentation d'une plateforme de chiffrement et de déchiffrement ECC, dans un environnement PSoC. Notre contribution dans ce travail consiste en la mise en œuvre de l'opération cœur de ce crypto système sur la carte de prototypage ZedBoard. L'opération en question est la multiplication scalaire.

Ce chapitre est consacré à la présentation des résultats d'implémentation. Nous allons entamer par décrire la méthodologie élaborée pour le développement de la plateforme réalisée. Ensuite, nous présenterons les performances issues de l'implémentation sur circuit FPGA. Nous terminerons par la description de l'IHM.

4.2 Méthodologie de conception

La méthodologie élaborée pour la réalisation du crypto système ECC, est répartie en trois étapes :

1. Configuration de la partie matérielle sur circuit FPGA à base du processeur ARM Cortex A9.
2. Développement des programmes en langage C pour l'exécution de la multiplication scalaire.
3. Développement de la partie Java et de l'IHM.

Les deux premières étapes ont été réalisées, en utilisant les outils de Xilinx :

- Vivado 2015.4 pour la spécification du système embarqué [27].
- SDK (Software Development Kit) pour le développement de sa partie logicielle en langage C [28].

La troisième étape a été réalisée en langage Java, sous l'environnement NetBeans.

Le standard de la courbe elliptique utilisé dans ce travail est défini dans secp256r1 [19]. Ce dernier fournit un niveau de sécurité d'une taille de 256 bits. Les valeurs numériques en hexadécimal des paramètres p, a, b, G(X, Y) et n de la courbe sont définies comme suit :

$$p = 0xFFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF$$
$$a = 0xFFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFC$$
$$b = 0x5AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3C3E27D2604B$$
$$X = 0x6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C296$$
$$Y = 0x4FE342E2FE1A7F9B8EE7EB4A7C0F9E162BCE33576B315ECECBB6406837BF51F5$$
$$n = 0xFFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551$$

4.3 Performances temporelles et ressources matérielles occupées

Les performances du crypto système ECC sont présentées en termes de temps d'exécution et des ressources matérielles occupées. L'évaluation de ces performances concerne principalement la partie embarquée sur circuit FPGA.

4.3.1 Performances temporelles

Notre étude pour déterminer les performances temporelles de la partie implémentée sur circuit FPGA XC7Z020-1CLG484CES EPP, est basée sur l'évaluation des délais d'exécution de la multiplication scalaire, du chiffrement et du déchiffrement. Ces délais sont calculés par la multiplication de la période par le nombre top d'horloge. Ce dernier est déterminé en activant et en désactivant respectivement le Timer du système embarqué au début et à la fin de chacune des fonctions ciblées. La fonction du Timer permettant d'effectuer ces deux tâches est : *XTime_GetTime(.)*. Les performances temporelles de ces opérations sont illustrées dans le tableau 4.1.

Tableau 4.1 - Performances temporelles.

Fréquence (Mhz)	**Multiplication scalaire (us)**	**Chiffrement (us)**	**Déchiffrement (us)**
666,66	$2{,}78\ 10^4$	$5{,}58\ 10^4$	$2{,}94\ 10^4$

A partir de ces résultats, on constate que le délai du chiffrement représente approximativement le double du délai de la multiplication scalaire. Ceci est justifié par le fait que l'opération de chiffrement utilise deux multiplications scalaires. Alors que le temps du déchiffrement est légèrement supérieur au délai de la multiplication scalaire. Ceci est justifié, car d'après l'équation du déchiffrement, cette dernière nécessite une multiplication scalaire et une addition de deux points.

4.3.2 Ressources matérielles requises

Dans ce travail, aucune des ressources matérielles de la partie PL du circuit FPGA n'est utilisée pour l'implémentation du système embarqué proposé. De ce fait, les ressources requises pour la réalisation de notre système se résument uniquement dans l'utilisation de la partie PS.

4.4 Description de l'IHM

Nous avons réalisé cette interface pour une meilleure flexibilité de la plateforme réalisée. Les tâches assurées par celle-ci sont les suivantes :

- Configuration du port RS232 ;
- Génération des clés publique et privée ;
- Transmission des données pour leur chiffrement ou leur déchiffrement vers la carte FPGA.

La page d'accueil de l'IHM est présentée sur la figure 4.1.

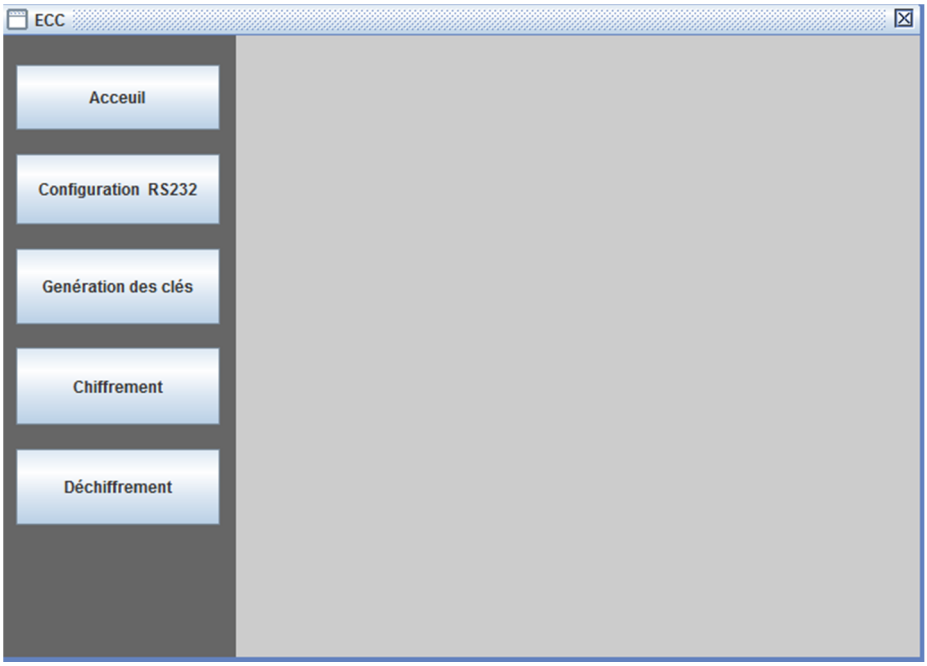


Figure 4.1 - Page d'accueil de l'IHM.

4.4.1 Configuration du port RS232

L'utilisation de la plateforme réalisée nécessite l'ouverture de la liaison de communication entre l'ordinateur et la partie embarquée sur circuit FPGA. Pour ce faire, cette partie de l'IHM nous permet de configurer les paramètres du protocole RS232. Ces derniers sont :

- Débit de transmission des données.

- La taille des données à transmettre.
- Le bit de parité.
- Le bit d'arrêt.
- Le control de flux.

Dans notre travail, la configuration utilisée est montrée sur la figure 4.2. A noter que la définition de ces paramètres doit être identique à la configuration des paramètres de l'UART embraquée sur circuit FPGA.

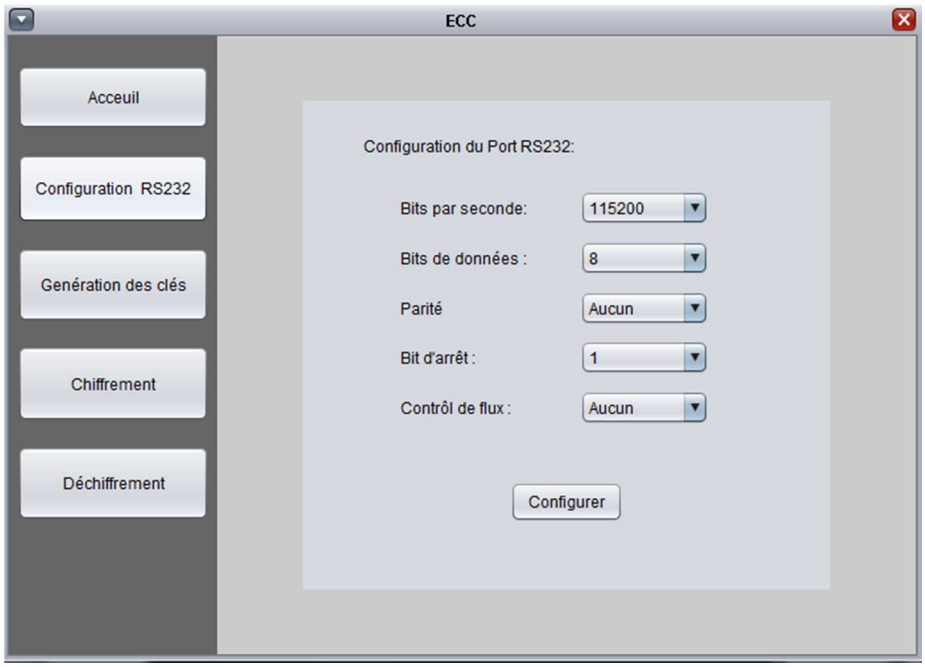


Figure 4.2 - Configuration du port RS232.

4.4.2 Génération des clés publique et privée

Cette partie de l'IHM permet à l'utilisateur de générer les deux clés publique $Pb(Xpb, Ypb, Zpb)$ et privée nb et de les enregistrer dans un fichier texte. La fenêtre permettant d'effectuer ces opérations est montrée sur la figure 4.3.

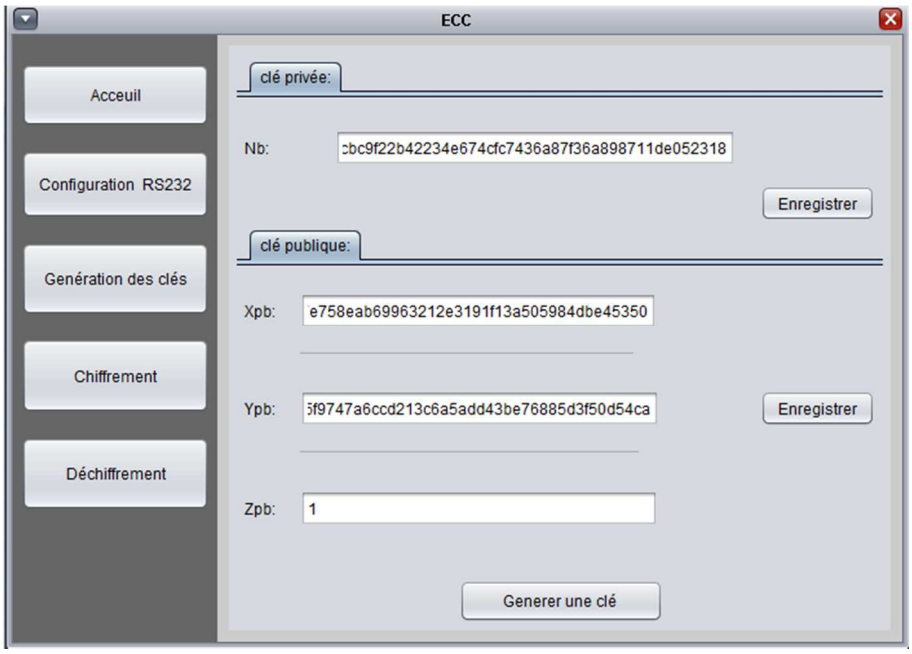


Figure 4.3 - Génération des clés.

Un exemple de génération de clés est illustré ci-dessous :

$nb = bd16815c5a1febe5f6e70f8ba4daed0a0462d481d556410d4ef58755a4059e47$

$Xpb = b3502afcc6116a4910ace0ef593d89c257d1e0a02e0d5d40af6572bf78f9b943$

$Ypb = 3fd968105ec08003f03f2c89de57f6a2a8b49031701000aaf6f6619029e02293$

$Zpb = 1$

4.4.3 Transmission des données pour leur chiffrement ou leur déchiffrement vers la carte FPGA

4.4.3.1 Transmission des données pour leur chiffrement

Lors de cette étape, l'opération du chiffrement des données est exécutée sur circuit FPGA. L'interface permettant d'effectuer cette tâche est constituée de quatre parties :

- Un champ pour saisir le message à chiffrer.
- Un bouton pour indiquer le chemin du fichier où la clé publique est enregistrée.
- Un bouton pour la transmission des données saisies vers la carte FPGA.
- Un bouton pour spécifier le chemin où le résultat obtenu de la carte sera stocké.

Cette partie de l'IHM est montrée sur la figure 4.4.

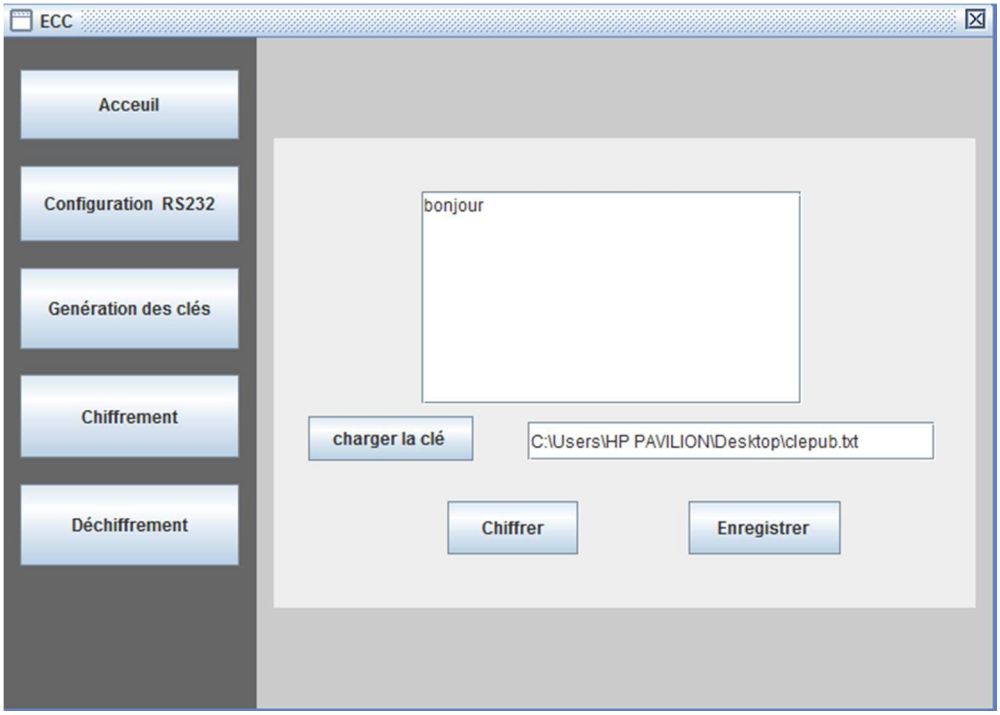


Figure 4.4 - chiffrement des données.

Un exemple de chiffrement d'un message est présenté ci-dessous:

Message en clair : *bonjour*

Le point $Pm(Xm, Ym, Zm)$ représentant le message :

$Xm = 1ec2d28142d4b3e$

$Ym = 9870db06f9e0d71e6ac94da93ac3a35c$

$Zm = 1$

Les points $kG(Xkg, Ykg, Zkg)$ et $Pc(Xc, Yc, Zc)$ résultants du chiffrement sont :

$Xkg = bdf54cec88dd67fcb472e8d68523d5eee8f22124dba9a32547f84e1462507fae$

$Ykg = 7c4ec485c6c8670e5c3b15676dc44bdbd149be63b9099983b413f885d5d535f1$

$Zkg = 5449107124a4d3008dc1f6c94fc5fe4752019e24c1e3843c0cd9928c18d94c53$

$Xc = 520cd94b54f125c37fc6f01cf354e953b6809db201a40316b8f58a70ed81cc9a$

$Yc = 88609619adc04317c609b03b8310b6f28b8df8e5e622bf32f6c852778cc97c06$

$Zc = 5e059b8ebe7216e303252e5beb360b47090f59353c0c62f79cd9bfe31c4a3d46$

Ces résultats de chiffrement seront enregistrés dans un fichier texte (**.txt).*

4.4.3.2 Transmission des données pour leur déchiffrement

Dans cette étape, l'interface assure la transmission des données à déchiffrer vers la carte FPGA et la réception des données résultantes du déchiffrement. Pour cela, quatre parties sont définies comme suit :

- Un bouton pour spécifier le chemin du fichier à déchiffrer.
- Un bouton pour le chargement du fichier de la clé privée.
- Un bouton pour la transmission des données à déchiffrer vers la carte FPGA.
- Un champ pour récupérer le message en clair obtenu après déchiffrement.

Cette partie de l'IHM est montrée sur la figure 4.5.

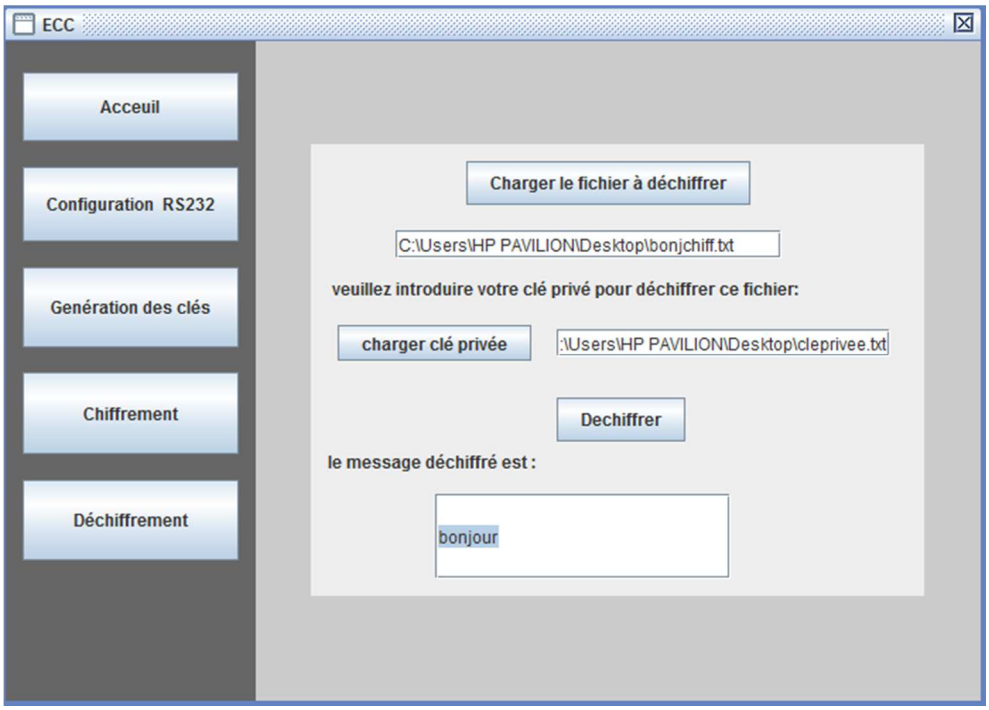


Figure 4.5 - Déchiffrement des données.

4.5 Conclusion

Dans ce chapitre, nous avons présenté la méthodologie élaborée pour la conception d'un crypto système ECC, à base d'une plateforme PSoC. Nous avons montré les performances atteintes en termes de temps d'exécution de la partie embarquée sur circuit FPGA. Nous avons terminé par la description de l'IHM développée pour la vérification fonctionnelle et l'utilisation de notre crypto système.

Conclusion Générale

L'objectif du travail réalisé dans le cadre de ce projet, est d'implémenter d'une manière purement logicielle un crypto système à clé publique ECC dans un environnement PSoC, à base du processeur ARM de Xilinx. Pour ce faire, nous avons étudié dans un premier temps la théorie sur la cryptographie ECC, puis nous avons présenté les systèmes embarqués sur puce mappée sur les périphériques Zynq à base du processeur ARM.

Une fonction cœur ressort de la cryptographie à base de courbes elliptiques, c'est la multiplication scalaire, qu'on retrouve dans le chiffrement, le déchiffrement et la génération des clés. Les performances du crypto système embarqué sont liées à la rapidité d'exécution de cette opération et à celle de ses opérations arithmétiques. Plus précisément la multiplication modulaire et l'exponentiation modulaire qui sont considérées comme étant des opérations critiques en termes de ressource et de temps d'exécution.

Pour cela, il est judicieux de bien choisir les algorithmes à implémenter. Après une étude de certains des algorithmes proposés à ces fins-là, nous avons fini par choisir l'algorithme de multiplication modulaire de Montgomery pour accélérer la multiplication modulaire et l'algorithme R2L pour l'exponentiation modulaire.

Après avoir implémenté toutes les opérations nécessaires pour la réalisation de notre crypto système ECC embarqué sur FPGA. Nous avons développé une interface Homme/Machine, afin de pouvoir utiliser le système d'une façon plus flexible et de vérifier son bon fonctionnement.

En guise de perspectives, nous proposons d'exploiter toutes les ressources de la carte utilisée, à savoir la carte ZedBoard muni d'un processeur ARM à double cœur. Sur lequel le parallélisme se trouvant dans l'algorithme de la multiplication scalaire, implémenté sur les deux cœurs du processeur peut augmenter d'une manière très considérable les performances du système en question de temps d'exécution.

```
while(1){
      //Reception
      Xuint8 temp=XUartPs_RecvByte(0xe0001000);
if (temp==0){
      for(i=0;i<8;i++)  {    X[i]=inValeur32Bit();  }
      for(i=0;i<8;i++)  {    Y[i]=inValeur32Bit();  }
      for(i=0;i<8;i++)  {     Z[i]=inValeur32Bit(); }
      for(i=0;i<8;i++)  {     k[i]=inValeur32Bit();  }
      for(i=0;i<8;i++)  {     Xm[i]=inValeur32Bit(); }
      for(i=0;i<8;i++)  {     Ym[i]=inValeur32Bit(); }
      for(i=0;i<8;i++)  {     Zm[i]=inValeur32Bit(); }
      for(i=0;i<8;i++)  {     Xpb[i]=inValeur32Bit();}
      for(i=0;i<8;i++)  {     Ypb[i]=inValeur32Bit();}
      for(i=0;i<8;i++)  {     Zpb[i]=inValeur32Bit();}
                              montgomery(X,R2,M,m,X);
                              montgomery(Y,R2,M,m,Y);
                              montgomery(Z,R2,M,m,Z);

                              montgomery(Xm,R2,M,m,Xm);
                              montgomery(Ym,R2,M,m,Ym);
                              montgomery(Zm,R2,M,m,Zm);

                               montgomery(Xpb,R2,M,m,Xpb);
                              montgomery(Ypb,R2,M,m,Ypb);
                              montgomery(Zpb,R2,M,m,Zpb);
chiffrement(X,Y,Z,a,M,m,R2,k,u,Xm,Ym,Zm,Xpb,Ypb,Zpb,Xkg,Ykg,Zkg,Xc,Yc,Zc);
//envoi
// XUartPs_SendByte (0xe0001000, temp);
  for(i=0;i<8;i++) {   outValeur32Bit(Xkg[i]);  }
  for(i=0;i<8;i++) {   outValeur32Bit(Ykg[i]);  }
  for(i=0;i<8;i++) {   outValeur32Bit(Zkg[i]);  }
  for(i=0;i<8;i++) {   outValeur32Bit(Xc[i]); }
  for(i=0;i<8;i++) {   outValeur32Bit(Yc[i]); }
  for(i=0;i<8;i++) {   outValeur32Bit(Zc[i]); }
}
else {
      if (temp==1)
for(i=0;i<8;i++){    nb[i]=inValeur32Bit();  }
for(i=0;i<8;i++) {   Xkg[i]=inValeur32Bit(); }
for(i=0;i<8;i++) {   Ykg[i]=inValeur32Bit(); }
for(i=0;i<8;i++){    Zkg[i]=inValeur32Bit(); }
for(i=0;i<8;i++) {   Xc[i]=inValeur32Bit(); }
for(i=0;i<8;i++) {   Yc[i]=inValeur32Bit(); }
for(i=0;i<8;i++){    Zc[i]=inValeur32Bit(); }
Dechiffrement(Xkg,Ykg,Zkg,resf1,resf2,resf3,nb,E,R,a,M,m, R2,k,u,Xm, Ym);
for(i=0;i<8;i++) {   outValeur32Bit(Xm[i]);}
for(i=0;i<8;i++) {  outValeur32Bit(Yf[i]); }
}
}
```

Figure 1 - Code C du programme main.

```
void chiffrement(Xuint32 X[9],Xuint32 Y[9],Xuint32 Z[9],Xuint32 a[9],Xuint32 M[9],Xuint32 m,
    Xuint32 R2[9],Xuint32 k[9],Xuint32 u[9],Xuint32 Xm[9],Xuint32 Ym[9],Xuint32 Zm[9],Xuint32 Xpb[9],Xuint32
      Ypb[9], Xuint32 Zpb[9],Xuint32 Xkg[9],Xuint32 Ykg[9],Xuint32 Zkg[9],
    Xuint32 Xc[9],Xuint32 Yc[9],Xuint32 Zc[9])
    {

      Xuint32 Xkpb[9], Ykpb[9],Zkpb[9];
int i ;

mul_scal1(X,Y,Z,a,M,m,R2,k,u, Xkg, Ykg, Zkg);

mul_scal1(Xpb,Ypb,Zpb,a,M,m,R2,k,u, Xkpb, Ykpb, Zkpb);

ECPA(Xm,Ym,Zm, Xkpb, Ykpb, Zkpb,M,m,R2,Xc,Yc,Zc);
}
```

Figure 2 - Code C de la fonction chiffrement.

```
void Dechiffrement(Xuint32 Xkg[9],Xuint32 Ykg[9],Xuint32 Zkg[9],
           Xuint32 Xc[9],Xuint32 Yc[9],Xuint32 Zc[9],
             Xuint32 nb[9],Xuint32 E[9],Xuint32 R[9],Xuint32 a[9],Xuint32 M[9],Xuint32 m,Xuint32 R2[9],Xuint32
      k[9],Xuint32 u[9], Xuint32 Xm[9],Xuint32 Ym[9])
{
 int i;
int nbrPaquet=8;
Xuint32 deux[9],trois[9];

Xuint32 xkGnb[9],ykGnb[9],zkGnb[9],SykGnb[9];
Xuint32 zero[9]={0};
Xuint32 xpmf[9],ypmf[9],zpmf[9];

mul_scal1(Xkg,Ykg,Zkg,a,M,m,R2,nb,u,xkGnb,ykGnb,zkGnb);

sub(zero,ykGnb,M,SykGnb);

ECPA(Xc,Yc,Zc,xkGnb,SykGnb,zkGnb,M,m,R2,xpmf,ypmf,zpmf);

 expBinnaryBase32(zpmf,E,M,R,m,zpminv,nbrPaquet);

      montgomery(zpminv,zpminv,M,m,zpminv2);

      montgomery(xpmf,zpminv2,M,m,xpm);

      montgomery(xpm,u,M,m,Xm);

      montgomery(zpminv2,zpminv,M,m,zpminv3);

      montgomery(ypmf,zpminv3,M,m,ypm);

      montgomery(ypm,u,M,m,Ym);

      }
```

Figure 3 - Code C de la fonction de déchiffrement.

```
void mul_scal1(Xuint32 xp[9],Xuint32 yp[9],Xuint32 zp[9],Xuint32 a[9],Xuint32 M[9],Xuint32 m,
    Xuint32 R2[9],Xuint32 k[9],Xuint32 u[9],Xuint32 Resultat1[9],Xuint32 Resultat2[9],Xuint32 Resultat3[9])
{
      int i ,j, l;
Xuint32 xpp[9],ypp[9],zpp[9];
Xuint32 xpd[9],ypd[9],zpd[9];
Xuint32 xpa[9],ypa[9],zpa[9];
Xuint32 xfin[9],yfin[9],zfin[9];
ECPD(xp,yp,zp,a,M,m,xpp,ypp,zpp);
int nbpaquet=8;
int nbbit=30;
for(j=nbpaquet-1 ; j>=0 ;j--)
{for (l=nbbit; l>=0; l--)
{    if(getBitIndex1(k[j],l)==0){
            ECPA(xp,yp,zp,xpp,ypp,zpp,M,m,R2,xpa,ypa,zpa);
            for (i=0;i<9;i++){              xpp[i]=xpa[i];
                                            ypp[i]=ypa[i];
                                            zpp[i]=zpa[i];
                                }
            ECPD(xp,yp,zp,a,M,m,xpd,ypd,zpd);
            for (i=0;i<9;i++)   {           xp[i]=xpd[i];
                                            yp[i]=ypd[i];
                                            zp[i]=zpd[i];
                                }
            for (i=0;i<9;i++)   {
                                            xfin[i]=xpd[i];
                                            yfin[i]=ypd[i];
                                            zfin[i]=zpd[i];
                                }
              }
else {
           ECPA(xp,yp,zp,xpp,ypp,zpp,M,m,R2,xpa,ypa,zpa);
           for (i=0;i<9;i++)
             {
                        xp[i]=xpa[i];
                        yp[i]=ypa[i];
                        zp[i]=zpa[i];
             }
           ECPD(xpp,ypp,zpp,a,M,m,xpd,ypd,zpd);
           for (i=0;i<9;i++)
      {                 xpp[i]=xpd[i];
                        ypp[i]=ypd[i];
                        zpp[i]=zpd[i];
}
             for (i=0;i<9;i++)
{                       xfin[i]=xpa[i];
                        yfin[i]=ypa[i];
                        zfin[i]=zpa[i];
      }}
}
nbbit=31;
}
for (i=0;i<=8;i++)
{
       Resultat1[i]=xfin[i];
       Resultat2[i]=yfin[i];
       Resultat3[i]=zfin[i];
}
}
```

Figure 4 - Code C de la fonction de multiplication scalaire.

```
void ECPA(Xuint32 x1[9],Xuint32 y1[9],Xuint32 z1[9],Xuint32 Xo[9],Xuint32 Yo[9],
          Xuint32 Zo[9],Xuint32 M[9],Xuint32 m,Xuint32 R2[9],
          Xuint32 Resultat1[9],Xuint32 Resultat2[9],Xuint32 Resultat3[9])
{
    Xuint32 U12[9],U1[9],U22[9],U2[9],S12[9],S1[9],S22[9],S2[9],H[9],H3[9],H2[9],F[9] , Xad[9], V[9], V2[9],
    W [9],Yad[9],Zad[9];
int i ;
montgomery(Zo,Zo,M,m,U12);
montgomery(x1,U12,M,m,U1);
montgomery(z1,z1,M,m,U22);
montgomery(U22,Xo,M,m,U2);

montgomery(U12,Zo,M,m,S12);
montgomery(S12,y1,M,m,S1);

montgomery(U22,z1,M,m,S22);
montgomery(S22,Yo,M,m,S2);

sub(U1,U2,M,H);
montgomery(H,H,M,m,H2);
montgomery(H2,H,M,m,H3);

sub(S1,S2,M,F);

montgomery(F,F,M,m,Xad);
add(Xad,H3,M,Xad);
montgomery(H2,U1,M,m,V);
add(V,V,M,V2);
sub(Xad,V2,M,Xad); //X addition
for(i=0;i<=8;i++)
    Resultat1[i]=Xad[i];

sub(V,Xad,M,Yad);
montgomery(Yad,F,M,m,Yad);
montgomery(H3,S1,M,m,W);
sub(Yad,W,M,Yad); //Y addition
for(i=0;i<=8;i++)
    Resultat2[i]=Yad[i];

montgomery(Zo,z1,M,m,Zad);
montgomery(Zad,H,M,m,Zad); //Z addition
for(i=0;i<=8;i++)
    Resultat3[i]=Zad[i];
}
```

Figure 5 - Code C de la fonction d'addition de deux points.

```
void ECPD(Xuint32 x1[9],Xuint32 y1[9],Xuint32 z1[9],Xuint32 a[9],Xuint32 M[9],
Xuint32 m,Xuint32 Resultat1[9],Xuint32 Resultat2[9],Xuint32 Resultat3[9])
{
Xuint32  K[9],Xo[9] , Stmp[9],Yo[9] ,Q[9],y3[9] , z[9] , Zo[9],M1[9], M12[9] ;
int i ;
montgomery(y1,y1,M,m,Q);
montgomery(Q,x1,M,m,Q);
add(Q,Q,M,Q);
add(Q,Q,M,Q);

montgomery(z1,z1,M,m,K);
montgomery(K,K,M,m,K);
montgomery(K,a,M,m,K);
montgomery(x1,x1,M,m,M1);
add(M1,M1,M,M12);
add(M12,M1,M,M12);
add(K,M12,M,K);
montgomery(K,K,M,m,Xo);
add(Q,Q,M,Stmp);
sub(Xo,Stmp,M,Xo);

for(i=0;i<=8;i++)
Resultat1[i]=Xo[i];
// X dedoublement

sub(Q,Xo,M,Yo);
montgomery(Yo,K,M,m,Yo);
montgomery(y1,y1,M,m,y3);
montgomery(y3,y3,M,m,y3);
add(y3,y3,M,y3);
add(y3,y3,M,y3);
add(y3,y3,M,y3);
sub(Yo,y3,M,Yo);
for(i=0;i<=8;i++)
Resultat2[i]=Yo[i];
// Y dedoublement

montgomery(y1,z1,M,m,z);
add(z,z,M,Zo);
for(i=0;i<=8;i++)
Resultat3[i]=Zo[i];
// Z dedoublement
}
```

Figure 6 - Code C de la fonction de dédoublement de point.

```
void expBinnaryBase32(unsigned long Y2[],unsigned long E[ ],unsigned long N[ ],unsigned long R[ ],unsigned
      long m,unsigned long Resultat[ ],int nbrPaquet)
{
int i,j;
for (i=0;i<9;i++)
      {
            S1[i]=Y2[i];
            C[i]=R[i];
      }
for(i=0;i<nbrPaquet;i++)
{
      for(j=0;j<32;j++)
      {    if(getBitIndex(E[i],j)==1
            {
            montgomery(C,S1,N,m,C);
      }
            montgomery(S1,S1,N,m,S1);
      }
}
for(i=0;i<nbrPaquet;i++)
      Resultat[i]=C[i];
}
```

Figure 7 - Code c de la fonction de l'exponentiation binaire.

Références Bibliographiques

[1]: Bruce Schneier « Cryptographie appliqué» Vuibert Informatique.

[2]: H.FERRADI « Initiation à la cryptographie : théorie et pratique », janvier 2016.

[3]: *Darrel Hankerson/Alfred Menezes/Scott Vanstone* "Guide to Elliptic Curve Cryptography "Springer ,2004.

[4]: J.J.Risel /P. Boyer, "Algèbre pour la licence 3", 27 février 2006.

[5]: Laiphrakpam Dolendro Singh « Implementation of Text Encryption using Elliptic Curve Cryptography 1-s2.0-S1877050915013332» IMCIP-2015.

[6]: Marc Joye « Scalar Multiplication on Weierstraÿ Elliptic Curves From Co-*Z* Arithmetic» RAIM 2013.

[7]: Marc Joye1 and Sung-Ming Yen2 «The Montgomery Powering Ladder », Ches, 2002.

[8]: J.S.Miline, « elliptic curves », BookSurge Publishers, pp 238-viii, 2006.

[9] : Stéphane Ballet et Alexis Bonecaze « Courbes Elliptiques Application à la cryptographie » Ecole Polytech de Marseille,2008.

[10] : Certicom Research," SEC 1: Elliptic Curve Cryptography", September 20, 2000, Version 1.0.

[11]: J.Y.Chouinard, "design of secure computer Systems CSI4138/CEG4394 :Notes on Elliptic Curve Cryptography, September 24,2002.

[12]: Björn Fay, « Double-and-Add with Relative Jacobian Coordinates", December 20, 2014.

[13]:M.H.Ahmed, S.W.Alam and I.Baig, "Architecture and Implementation of Modular Arithmetic Unit for SOC", IEEE computer society, Frontiers of Information Technology, 2011.

[14]:P.L.Montgomery, "Modular Multiplication without Trial Division", Mathematics of Computation, vol 44 (170).

[15]:T.Plantard , "Arithmétique Modulaire pour la Cryptographie" , Thèse PhD, Université Montpelier II, Laboratoire d'informatique, 2005.

[16] :M.Issad, « Crypto système Embarqué sur FPGA », Thèse de Doctorat , Département Télécommunications, Faculté d'Electronique et d'Informatique, usthb , 2014.

[17] : M.Gouy,G.Huvent and A.Ladureau, « Fermat revisité », mars 2003.

[18] : M.Issad,B.Boudraa, M.Anane and S.Sddiki, « FPGA Implementation of Modular Exponentiation Using Modular Multiplier » , International Conference on Circuits, Systems, Signal Processing, Communications and Computers, Venice, Italy, 2014.

[19]: Certicom Research, « SEC 2: Recommended Elliptic Curve Domain Parameters", September 20, 2000, Version 1.0.
[20]: Pr. Mustapha LALAM, « Systèmes Embarqués », cours RMSE 2016.
[21]: R.Saleh, S.Wilton, S.Mirabbasi, A.Hu, M.Greenstreet, G.Lemieux, O.P.Pand, C.Grecu and A.Ivanoc, « System-on-chip : Reuse and Integration », Proceedings of the IEEE, vol 94, No.6, June 2006.
[22]:E.A.Lee and S.A.Seshia, « Introduction Of Embedded Systems A Cyber-Physical Systems Approch", LeeSeshia.org,2011.
[23]: Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, Robert W. Stewart "Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC", July 2014 by Strathclyde Academic Media.
[24]: XILINX, "7 Series FPGAs Configurable Logic Block", UG474 (v1.8) September 27, 2016.
[25]: Fernando Rincón, « Zynq Architecture, PS (ARM) and PL", 5 June 2015.
[26]: AVNET electronics marketing, "ZedBoard (Zynq Evaluation and Development) Hardware User's Guide", 1-aug-2012.
[27]:XILINX , "Vivado Design Suite User Guide *Embedded Processor Hardware Design",* UG898 ,May 9, 2014.
[28]: Campbell scientific, Inc., « CSI Software Development Kit Beginner's Guide", 2002-2006.

Résumé

Le but du travail réalisé est d'implémenter d'une manière purement logicielle un crypto système à clé publique ECC (Elliptic Curve Cryptography) dans un environnement PSoC, sur la carte ZedBoard à base du processeur ARM de Xilinx. Dans un premier temps, nous avons effectué une étude théorique sur la cryptographie ECC, puis nous avons étudié les systèmes embarqués sur puce mappée sur les périphériques Zynq à base du processeur ARM. Au cours du projet, nous avons fait le choix d'utiliser certains algorithmes pour accélérer quelques opérations arithmétiques modulaires tel que la multiplication modulaire de Montgomery ou encore R2L pour l'exponentiation modulaire. Ce qui implique l'accélération de la multiplication scalaire qui se trouve être l'opération cœur des protocoles ECC. Une fois toutes ces opérations implémentées d'une façon purement logicielle sur le circuit FPGA, nous avons développé une IHM (Interface Homme/Machine) pour vérifier le bon fonctionnement du système.

Mot clés : cryptographie, ECC, FPGA, Montgomery, multiplication scalaire.

Abstract

The goal of the work is to implement a crypto public key system ECC (Elliptic Curve Cryptography) in a PSoC environment, on a ZedBoard based on the Xilinx ARM processor. Initially, we conducted a theoretical study on ECC cryptography, and then we studied embedded systems on a chip mapped to Zynq devices based on the ARM processor. During the project, we chose to use some algorithms to accelerate some modular arithmetic operations such as Montgomery's modular multiplication or R2L for modular exponentiation. This implies the acceleration of the scalar multiplication which happens to be the core operation of the ECC protocols. Once all these operations were implemented in a purely software way on the FPGA circuit, we developed an MMI (Man / Machine Interface) to check the correct functioning of the system.

Keywords: cryptography, ECC, FPGA, Montgomery, scalar multiplication.