École Normale Supérieure de Lyon

Master 2 d'Informatique Fondamentale
Rapport de stage

Génération automatique de circuits pour le calcul de couplages cryptographiques en matériel

Nicolas Estibals

Encadrant : Jérémie Detrey
Co-encadrant : Jean-Luc Beuchat

février – juin 2009

Abstract

This document report the work that I realize during my Master 2 internship in the project-team CACAO at LORIA under the supervision of Jérémie Detrey and the co-supervision of Jean-Luc Beuchat (LCIS, Japan). In order to compute cryptographic pairings efficiently on some dedicated circuits, we design a family of specific coprocessors and an automatic tools for programming them. Then we have explored a large range of trade-off in the choice of algorithms and the architecture of the coprocessor. Thanks to a finely tuned co-design we present good performances in pairing computation.

Keywords : list scheduling, automatic parallelization, finite field arithmetic,elliptic curves, pairing based cryptography, hardware implementation, coprocessor design.

Résumé

Ce rapport rend compte des travaux que j'ai pu effectuer durant mon stage de Master 2 dans l'équipe-projet CACAO au LORIA sous la direction de Jérémie Detrey et la codirection de Jean-Luc Beuchat(LCIS, Japon). Dans le but de calculer efficacement des couplages cryptographiques sur des circuits spécifiques, nous avons conçu une famille de coprocesseurs adaptés et un outil automatique pour les programmer. Grâce à cette approche, nous avons pu explorer une large gamme de compromis algorithmiques et architecturaux. Ainsi nous avons obtenu de bonnes performances pour le calcul de couplages grâce à une adéquation algorithme–architecture fine.

Mots-clefs : ordonnancement de liste, parallélisation automatique, arithmétique des corps finis, courbes elliptiques, cryptographie fondée sur les couplages, implémentation matériel, conception de coprocesseur.

Table des matières

Chapitre 1

Introduction

La cryptographie à base de courbes elliptiques suscite beaucoup d'intérêt depuis qu'elles ont été introduite par Koblitz [19] et Miller [24] en 1985. Leur utilisation est maintenant standardisée et recommandée par diverses organisations gouvernementales (NIST [14], NSA [25], DCSSI, BSI, etc.). Les couplages sont d'abord apparus dans le monde de la cryptographie comme une attaque contre certaines courbes elliptiques [22, 15]. Les couplages sont désormais également utilisés pour construire des protocoles, parmi ceux-ci citons par exemple : l'échange de clef tripartite [16], la signature numérique courte [10] ou encore le chiffrement basé sur l'identité [9], posé par Shamir en 1984 [26], ce problème du chiffrement basé sur l'identité est resté un problème ouvert pendant plus de 15 ans. Les efforts actuels de standardisation de la cryptographie à base de couplage (ISO/IEC 14888-3 et IEEE P1363.3) témoignent de l'intérêt grandissant que la communauté internationale porte aux couplages.

Les couplages utilisés en cryptographie sont des applications bilinéaires qui prennnent deux points d'une courbe elliptique en arguments. Les courbes elliptiques considérées sont définies sur des corps finis. Calculer le couplage de deux points d'une telle courbe peut se ramener à des calculs sur le corps fini de définition de la courbe. Nous étudions ici ces couplages pour des corps finis de petite caractéristique (2 ou 3). Les jeux d'instructions des processeurs généralistes n'étant pas adaptés à l'implémentation de l'arithmétique sur ces corps, nous nous intéressons à la conception d'architectures matérielles pour le calcul des couplages. Le calcul du couplage de deux points d'une courbe comporte deux étapes principales : les itérations de Miller [23] et l'exponentiation finale. Durant ce stage nous nous sommes particulièrement intéressés à cette seconde étape.

Nous nous sommes donc appliqués dans ce stage à la création d'un coprocesseur pour le calcul de l'exponentiation finale. Nous nous sommes basés sur les travaux de Beuchat *et al.* qui ont proposé un tel coprocesseur dans [7]. Les auteurs obtenaient de bonnes performances avec ce coprocesseur, cependant la difficulté résidait dans la programmation de celui-ci. En effet programmer l'exponentiation finale d'un couplage sur une courbe particulière nécessitait une semaine de travail. Ce coût de développement prohibitif ne permettait ainsi pas d'obtenir les exponentiations finales de couplages pour différents niveaux de sécurité. De plus nous ne pouvions pas non plus explorer différents compromis algorithmiques et architecturaux qui permettraient d'améliorer les performances obtenues pour le calcul de l'exponentiation finale.

Cette constatation a été le point de départ des travaux réalisés durant ce stage. L'objectif premier a été de créer un compilateur permettant de programmer automatiquement le coprocesseur proposé dans [7]. Nous avons étendue cet objectif à la création d'un modèle d'architecture de coprocesseur arithmétique et d'un compilateur associé.

1.1 Contribution

Ce stage a été l'occasion de mener des études dans plusieurs domaines. Tout d'abord, il nous a fallu nous intéresser à l'algorithme de calcul de l'exponentiation finale ; nous avons pu en proposer une amélioration fonctionnant sur certaines courbes (Section 2.4.3), amélioration qui s'est révélée effective pour l'un des couplages étudié dans [7] (Section 5.2.3). Les principales contributions de ce stage ont été de proposer un modèle de coprocesseur (Chapitre 3.2) et un compilateur (Chapitre 4) pour celui-ci. Ce compilateur a été développé en Python et nous a permis d'obtenir différents ordonnancements, nous permettant ainsi de comparer différents compromis algorithme/architecture.

1.2 Contenu

Nous développerons dans ce rapport les trois axes qui ont permis la conception d'une famille de coprocesseurs pour le calcul de l'exponentiation finale d'un couplage en caractéristique 2 et 3 : les aspects mathématiques et algorithmiques (Chapitre 2), la description d'un modèle architectural de coprocesseurs (Chapitre 3) et la compilation de ces algorithmes pour l'exécution sur ces coprocesseurs (Chapitre 4). Puis nous décrivons les résultats obtenus dans le Chapitre 5.

Chapitre 2

Courbes elliptiques et couplages

Dans ce Chapitre, nous introduisons un certain nombre de notations classiques en donnant quelques points-clefs de la construction des courbes elliptiques et du couplage de Tate avant de détailler plus précisément l'algorithme de calcul de la phase dite d'*exponentiation finale* de ce couplage. Nous n'avons malheureusement pas la place dans ce rapport de rappeler l'arithmétique des corps finis, mais le lecteur pourra se reporter à l'ouvrage de Lang [21] pour leur construction et au livre de von zur Gathen et Gerhard [29] pour un point de vue plus algorithmique.

2.1 Courbes elliptiques

Soit K un corps. Nous définissons la courbe elliptique E sur K comme l'ensemble des points $(x, y) \in K^2$ solutions d'une équation polynomiale de degré 3 en x et quadratique en y. Cette équation peut s'écrire classiquement sous la forme de Weierstraß :

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$

où a_1, a_2, a_3, a_4, a_6 sont dans K.

Nous cherchons alors à munir cet ensemble de points d'une structure de groupe additif. Cette addition peut être définie géométriquement. Une droite passant par deux points de la courbe va l'intersecter en un troisième point. Dans le cas où la droite est verticale, nous considérons alors qu'elle intersecte aussi la courbe en un troisième point : le point *à l'infini*, noté $\mathcal{O}$. Ce point apparaît de façon naturelle en considérant la courbe dans le plan projectif (voir [30, sections 2.2 et 2.3] pour une construction rigoureuse). Il nous fournit le neutre pour l'addition que nous définissons de la manière suivante : la somme de trois points alignés est toujours nulle. Ainsi deux points verticalement alignés sont opposés l'un de l'autre et la somme de deux points de la courbe non verticalement alignés P et Q est le symétrique par rapport à l'axe des abscisses du troisième point d'intersection de la courbe avec la droite (PQ). Ces constructions sont résumées dans la Figure 2.1. Nous notons $(E(K), +)$ le groupe ainsi obtenu.

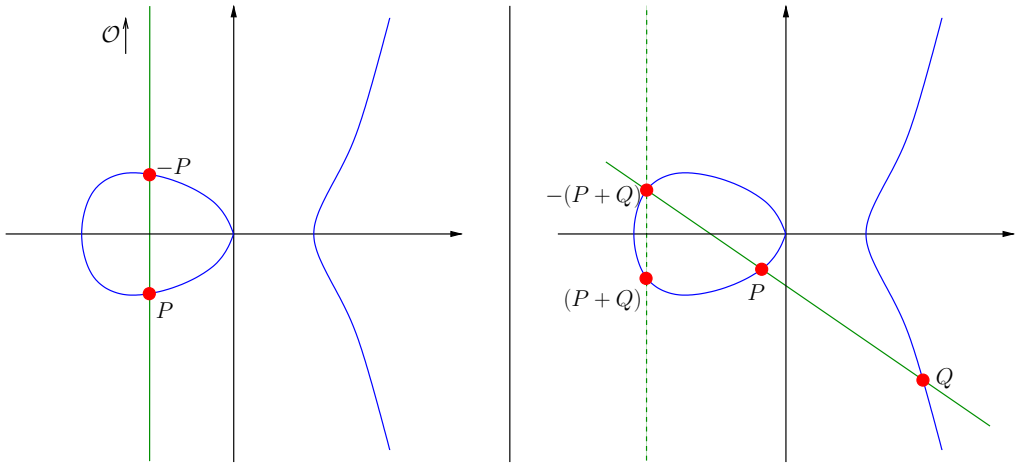

FIGURE 2.1 – Opposé d'un point et somme de deux points d'une courbe elliptique sur le corps $K = \mathbb{R}$

Dans notre cas nous nous restreignons aux courbes elliptiques sur les corps finis. Un corps fini à q éléments se note $\mathbb{F}_q$. Remarquons en guise de rappel qu'un corps fini est nécessairement : (i) commutatif, (ii) de cardinal la puissance d'un nombre premier ($q = p^m$, ce p s'appelle la caractéristique du corps). Le groupe $E(\mathbb{F}_q)$ est alors fini et le théorème de Hasse nous fournit une borne sur le cardinal de celui-ci.

Théorème 1 (Hasse)**.** *Le cardinal d'une courbe elliptique sur un corps fini est donné par la formule suivante :*

$$\#E(\mathbb{F}_q) = q - t + 1 \text{ avec } |t| \leqslant 2\sqrt{q}$$

Nous posons ℓ le plus grand facteur premier du cardinal de $E(\mathbb{F}_q)$. Nous définissons alors la ℓ-torsion de la courbe E sur $\mathbb{F}_q$ (notée $E(\mathbb{F}_q)[\ell]$) comme le groupe des points d'ordre ℓ — c'est-à-dire les points P de $E(\mathbb{F}_q)$ tels que :

$$\underbrace{P + P + \cdots + P}_{\ell \text{ fois}} = [\ell]P = \mathcal{O}.$$

2.2 Couplage de Tate

Soit $\mu_\ell = \left\{x \in \overline{\mathbb{F}}_q \mid x^\ell = 1\right\}$ le groupe des racines ℓ-ièmes de l'unité. Sur certaines courbes elliptiques (notamment les courbes supersingulières [8, section IX.10], [30, section 4.6]), on peut définir le couplage de Tate réduit comme une application qui associe à deux points de ℓ-torsion une racine ℓ-ième de l'unité.

Soit k le plus petit entier tel que toutes les racines ℓ-ième de l'unité soient dans $\mathbb{F}_{q^k}$. Ce k s'appelle le degré de plongement (*embedding degree*). μ_ℓ est alors un sous-groupe de $\mathbb{F}_{q^k}^*$ et nous avons $\ell \mid q^k - 1$.

Définition 2. Le couplage de Tate réduit est défini ainsi :

$$\begin{array}{rccl} \hat{e}: & E(\mathbb{F}_q)[\ell] \times E(\mathbb{F}_q)[\ell] & \to & \mu_\ell \subseteq \mathbb{F}_{q^k}^* \\ & (P, Q) & \mapsto & \langle P, Q\rangle_N^{\frac{q^k-1}{N}}, \end{array}$$

où $\langle P, Q\rangle_N \in \mathbb{F}_{q^k}^*$ est le couplage de Tate (non réduit) d'ordre N [30, section 11.3] et où $\ell \mid N \mid q^k - 1$ (généralement nous prenons $N = \#E(\mathbb{F}_q)$).

Le couplage non réduit $\langle \cdot, \cdot \rangle_N$ peut se calculer grâce à l'algorithme de Miller [23] en $\lceil \log_2 N \rceil$ itérations, à la manière d'un algorithme de multiplication par doublement/addition : chaque itération se compose du doublement d'un point de la courbe et d'un carré sur $\mathbb{F}_{q^k}^*$, ainsi qu'éventuellement une somme de deux points et d'une multiplication sur l'extension $\mathbb{F}_{q^k}^*$ [8, section IX.8]. En prenant $N = \#E(\mathbb{F}_q)$, nous déduisons du théorème de Hasse qu'il faut environ $\lceil \log_2 q \rceil = \lceil m \log_2 p \rceil$ itérations de Miller pour calculer un tel couplage.

La dernière étape du calcul du couplage de Tate réduit consiste à élever la valeur du couplage non réduit, qui appartient à $\mathbb{F}_{q^k}^*$, à la puissance $M = \frac{q^k-1}{N}$ afin de l'emmener dans μ_ℓ ; nous détaillons ce calcul dans la section 2.4.

2.3 Courbes en petite caractéristique

Nous décrivons dans cette Section les courbes que nous utilisons, telles qu'initialement décrites par Barreto *et al.* dans [2].

2.3.1 Caractéristique 2

En caractéristique 2 nous définissons la courbe suivante :

$$E: \quad y^2 + y = x^3 + x + b,\ b \in \mathbb{F}_2.$$

Nous nous restreignons au cas où m est premier et nous avons alors $N = \#E(\mathbb{F}_{2^m}) = 2^m \pm 2^{\frac{m+1}{2}} + 1$.

Le degré de plongement est $k = 4$ et nous construisons l'extension $\mathbb{F}_{2^{4m}}$ par la tour d'extensions suivante : soit $s \in \mathbb{F}_{2^2}$ tel que $s^2 = s + 1$ et $t \in \mathbb{F}_{2^4}$ tel que $t^2 + t = s$, un élément de $\mathbb{F}_{2^{4m}}$ peut alors s'écrire sous la forme $u_0 + u_1 s + u_2 t + u_3 st$ avec les u_i dans $\mathbb{F}_{2^m}$, car on a :

$$\mathbb{F}_{2^{4m}} \cong \mathbb{F}_{2^m}[s, t] \cong \mathbb{F}_{2^m}[X, Y]/(X^2 + X + 1, Y^2 + Y + X).$$

Ainsi, toutes les opérations sur $\mathbb{F}_{2^{4m}}$ peuvent se ramener à des calculs sur $\mathbb{F}_{2^m}$.

L'exposant M est alors :

$$\frac{2^{4m} - 1}{N} = (2^{2m} - 1)(2^m \mp 2^{\frac{m+1}{2}} + 1).$$

2.3.2 Caractéristique 3

En caractéristique 3 la courbe que nous utilisons est définie par l'équation suivante :

$$E: \quad y^2 = x^3 - x + b,\ b \in \{-1, 1\}.$$

Nous considérons cette courbe sur $\mathbb{F}_{3^m}$ pour m premier avec 6 ; son cardinal est alors $N = 3^m \pm 3^{\frac{m+1}{2}} + 1$.

L'extension dans laquelle nous nous plongeons est de degré $k = 6$ et nous la construisons en posant $\rho \in \mathbb{F}_{3^3}$ tel que $\rho^3 = \rho + b$ et $\sigma \in \mathbb{F}_{3^2}$ tel que $\sigma^2 = -1$. Tous les éléments de $\mathbb{F}_{3^{6m}}$ peuvent alors s'écrire $u_0 + u_1\sigma + u_2\rho + u_3\sigma\rho + u_4\rho + u_5\sigma\rho^2$ avec les u_i dans $\mathbb{F}_{3^m}$ et les opérations sur $\mathbb{F}_{3^{6m}}$ se réduisent à des calculs sur le corps de base, en effet :

$$\mathbb{F}_{3^{6m}} \cong \mathbb{F}_{3^m}[\rho, \sigma] \cong \mathbb{F}_{3^m}[X, Y]/(X^3 - X - b, Y^2 + 1).$$

De même qu'en caractéristique 2, l'exponentiation finale a une forme factorisée particulière avec :

$$M = \frac{3^{6m} - 1}{N} = (3^{3m} - 1)(3^m + 1)(3^m \mp 3^{\frac{m+1}{2}} + 1).$$

2.4 Exponentiation finale

Durant ce stage nous nous sommes particulièrement intéressés au calcul de l'exponentiation finale du couplage de Tate réduit, c'est-à-dire à calculer U^M une fois $U = \langle P, Q\rangle_N \in \mathbb{F}^*_{p^{km}}$ obtenu grâce à l'algorithme de Miller.

Remarquons que sur un corps fini de caractéristique p, l'élévation à la puissance p est un automorphisme : elle est linéaire ($(a+b)^p = a^p + b^p$) et bijective ; on parle de l'automorphisme de Frobenius. Remarquons que sa réciproque (l'extraction de racine p-ième) est elle aussi linéaire. Nous détaillerons leur calcul dans la Section 3.1.3.

Calculer l'exponentiation finale en utilisant l'algorithme d'exponentiation rapide en base p nécessiterait au plus $\lceil \log_p M \rceil \approx (k-1)m$ Frobenius et multiplications sur $\mathbb{F}^*_{p^{km}}$. Toutefois les multiplications sur l'extension $(\mathbb{F}^*_{p^{km}})$ sont coûteuses : il faut effectuer 9 multiplications et 20 additions sur le corps de base en caractéristique 2, ou 15 multiplications et 67 additions en caractéristique 3. Nous allons voir que la forme particulière de notre exposant M va nous permettre de réduire le coût global de cette exponentiation finale.

2.4.1 Caractéristique 2

Pour les deux caractéristiques nous commençons par élever à la puissance $p^{\frac{k}{2}m} - 1$. En caractéristique 2, il suffit pour ce calcul de faire 12 multiplications, 22 additions, 5 Frobenius et une inversion sur $\mathbb{F}_{2^m}$ (voir l'algorithme 3 de [4]).

Comment réaliser cette inversion ? Soit $u \in \mathbb{F}^*_{2^m}$; grâce au petit théorème de Fermat, nous avons :

$$u^{-1} = u^{2^m - 2} = \left(u^{2^{m-1}-1}\right)^2.$$

Nous remarquons alors qu'en base 2, l'entier $2^{m-1} - 1$ s'écrit $\overline{11\ldots1}^{(2)}$, nous appelons ce nombre le *repunit* (pour *repeated unit*) en base 2 de taille $m-1$. Beuchat *et al.* donnent une méthode de calcul pour l'élévation à cette puissance dans [5, section 3.2]. Une chaîne d'additions [18, section 4.6.3] pour n est une suite d'entiers commençant par 1 telle que chaque élément soit la somme de deux éléments qui le précèdent et qui termine par n. Si cette chaîne est de longueur minimale, elle comprend $\Theta(\log_2 n)$ entiers. Soit (n_i) une telle chaîne pour $m-1$. Nous calculons alors les $u_{n_i} = u^{2^{n_i}-1}$ séquentiellement ; en effet si $n_i = n_{i_1} + n_{i_2}$, nous avons :

$$u_{n_i} = u_{n_{i_1}} \cdot \left(u_{n_{i_2}}\right)^{2^{n_{i_1}}}.$$

Il nous faut alors $\Theta(\log_2 m)$ multiplications et $m-2$ Frobenius sur le corps de base pour calculer $u^{2^{m-1}-1}$.

Exemple 1. Soit $u \in \mathbb{F}_{2^{11}}$, calculons $u^{2^{10}-1}$. Choisissons la chaîne d'addition suivante 1, 2, 3, 5, 10. Nous pouvons alors calculer les $u^{2^n-1} = u_n$ ainsi :

$$\begin{array}{lllll}
u_1 & \leftarrow & u & = & u^{\overline{1}^{(2)}} \\
u_2 & \leftarrow & {u_1}^2 \cdot u_1 & = & u^{\overline{10}^{(2)} + \overline{1}^{(2)}} \\
u_3 & \leftarrow & {u_2}^2 \cdot u_1 & = & u^{\overline{110}^{(2)} + \overline{1}^{(2)}} \\
u_5 & \leftarrow & {u_3}^{2^2} \cdot u_2 & = & u^{\overline{11100}^{(2)} + \overline{11}^{(2)}} \\
u_{10} & \leftarrow & {u_5}^{2^5} \cdot u_5 & = & u^{\overline{1111100000}^{(2)} + \overline{11111}^{(2)}}
\end{array}$$

Comme $U \in \mathbb{F}_{2^{4m}}^*$, U est d'ordre divisant $2^{4m}-1$. Une fois calculé $V = U^{2^{2m}-1}$, V est d'ordre divisant $2^{2m}+1$ et, comme Beuchat *et al.* l'ont montré dans [4], nous pouvons alors obtenir V^{2^m+1} en seulement 5 multiplications, 2 Frobenius et 9 additions sur le corps de base. Nous devons aussi calculer $V^{2^{\frac{m+1}{2}}}$: il s'agit d'appliquer $\frac{m+1}{2}$ fois le Frobenius sur chacun des coefficients de V dans $\mathbb{F}_{2^m}$ et d'effectuer 4 additions sur les résultats [7, section 2.2].

Selon le signe dans M, il nous faut possiblement inverser $V^{2^{\frac{m+1}{2}}}$. Cependant cette opération ne coûte que 2 additions ; en effet pour inverser un élément d'ordre $2^{2m}+1$ sur $\mathbb{F}_{2^{4m}}^*$, il suffit d'élever à la puissance 2^{2m} et :

$$\text{pour } V = V_0 + tV_1 \in \mathbb{F}_{2^{4m}} \text{ avec } V_0, V_1 \in \mathbb{F}_{2^{2m}}, V^{2^{2m}} = V_0 + t^{2^{2m}} V_1 = V_0 + V_1 + tV_1.$$

Enfin il reste à multiplier V^{2^m+1} et $V^{\pm 2^{\frac{m+1}{2}}}$ (9 multiplications et 20 additions sur $\mathbb{F}_{2^m}$).

2.4.2 Caractéristique 3

La première étape du calcul est l'élévation à la puissance $3^{\frac{k}{2}m} - 1$; comme en caractéristique 2 nous disposons d'un algorithme au coût réduit pour ce calcul : 40 multiplications, 67 additions et une inversion sur $\mathbb{F}_{3^m}$ [5, algorithme 6].

À l'instar de la caractéristique 2, l'inversion sur $\mathbb{F}_{3^m}$ se ramène aussi au calcul de l'élévation à la puissance le *repunit* de taille $(m-1)$ en base 3 (c'est-à-dire $\frac{3^{m-1}-1}{2}$) au prix de 2 multiplications et un Frobenius :

$$\text{Soit } u \in \mathbb{F}_{3^m}^*, \quad u^{-1} = u^{3^m-2} = \left(\left(u^{\frac{3^{m-1}-1}{2}}\right)^3\right)^2 \cdot u.$$

Cette fois il faut élever deux fois $V = U^{3^{\frac{k}{2}m}-1}$ à la puissance 3^m+1, chacune de ces élévations coûtant 9 multiplications et 18 ou 19 additions (selon la valeur du b définissant la courbe)[6]. Il faut aussi élever à la puissance $p^{\frac{m+1}{2}}$ ce qui demande $3m+3$ Frobenius et 6 additions sur $\mathbb{F}_{3^m}$ et éventuellement inverser le résultat : cette inversion d'un élément d'ordre divisant $3^{3m}+1$ est gratuite : il suffit de l'élever à la puissance 3^{3m}, ce qui correspond à changer quelques signes [6]. Il ne reste alors que le produit sur $\mathbb{F}_{3^{6m}}^*$ (15 multiplications et 67 additions sur $\mathbb{F}_{3^m}$).

2.4.3 Calculer avec le Frobenius inverse

Comme nous le verrons en Section 3.1.3, calculer la réciproque du Frobenius peut être moins coûteux que de calculer le Frobenius sur certains corps finis. Pouvons-nous adapter nos algorithmes pour qu'ils n'utilisent que des Frobenius inverses ? Remarquons dans un premier temps que, si $u \in \mathbb{F}_{p^m}$, $u^{p^m} = u$ et donc :

$$u^{p^i} = u^{p^{i \bmod m}}, \text{ en particulier, } \sqrt[p]{u} = u^{p^{m-1}}.$$

L'élévation à la puissance le *repunit* de taille $(m-1)$ en base p (noté $\mathrm{rep}(m-1)$), nécessaire pour l'inversion, peut se calculer de façon très similaire : il suffit de partir de $\sqrt[p]{u}$ et de construire successivement les $u'_{n_i} = u^{\mathrm{rep}(m-1)-\mathrm{rep}(m-n_i-1)}$ avec (n_i) une chaîne d'addition pour $(m-1)$. Il suffit pour cela de remarquer que si $n_i = n_{i_1} + n_{i_2}$, nous avons $u'_{n_i} = \sqrt[p^{n_{i_1}}]{u'_{n_{i_2}}} \cdot u'_{n_{i_1}}$. Il faut ainsi $(m-1)$ Frobenius inverses et $\Theta(\log_2 m)$ multiplications.

La seconde étape de calcul nécessitant des Frobenius est l'élévation à la puissance $p^{\frac{m+1}{2}}$ sur $\mathbb{F}_{p^{km}}^*$, elle consiste en quelques additions et k élévations à cette même puissance sur $\mathbb{F}_{p^m}^*$, or :

$$\text{Soit } u \in \mathbb{F}_{p^m}^*, \quad u^{p^{\frac{m+1}{2}}} = u^{p^{\frac{m+1}{2}-m}} = \sqrt[p^{\frac{m-1}{2}}]{u}.$$

Nous transformons ainsi le $k\frac{m+1}{2}$ Frobenius en $k\frac{m-1}{2}$ Frobenius inverses.

2.4.4 Récapitulatif des coûts

Nous récapitulons dans le tableau ci-dessous le total des coûts de calcul pour l'exponentiation finale en nombre d'opération sur le corps de base ($\mathbb{F}_{p^m}$). Cependant il faut noter l'hétérogénéité de ce calcul : certaines parties peuvent être parallélisées (comme le produit sur $\mathbb{F}_{p^{km}}$), d'autre sont purement séquentielles (inversion, chaînes de Frobenius).

	Multiplications	Additions	Frobenius
Caractéristique 2	$26 + \Theta(\log_2 m)$	51 ou 53	$3m+3$
Caractéristique 3	$73 + \Theta(\log_2 m)$	176 ou 179	$4m+1$

Chapitre 3

Un modèle de coprocesseur arithmétique

Nous avons vu que nous pouvions ramener nos calculs à des opérations sur $\mathbb{F}_{p^m}$, nous cherchons donc à créer un coprocesseur arithmétique capable de manipuler ces éléments. Après avoir montré comment réaliser les différentes opérations en matériel, nous dégagerons un modèle de coprocesseur avant de l'appliquer au calcul de l'exponentiation finale.

3.1 Opérateurs arithmétiques sur $\mathbb{F}_{p^m}$

3.1.1 Représentation des éléments

Le corps fini $\mathbb{F}_{p^m}$ est construit comme $\mathbb{F}_p[X]/(f)$ avec f un polynôme irréductible sur $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ de degré m, ainsi nous représentons les éléments de $\mathbb{F}_{p^m}$ comme les m coefficients $a_{m-1}, \ldots, a_1, a_0$ dans $\mathbb{F}_p$ de l'unique polynôme de degré au plus $(m-1)$ le représentant.

En caractéristique 2 chaque coefficient peut être représenté par un bit, l'addition est alors le `XOR` et la multiplication sur $\mathbb{F}_2$ le `AND`. Il faut ainsi m bits pour écrire un élément de $\mathbb{F}_{2^m}$.

Un élément de $\mathbb{F}_3$ peut prendre trois valeurs, nous choisissons alors de les représenter par deux bits (a^+ et a^-) à la manière de la représentation *borrow-save* [13] : la valeur est alors $a^+ - a^-$. Ainsi il faut $2m$ bits pour représenter un élément de $\mathbb{F}_{3^m}$. Notons que cette représentation l'avantage de diviser par 2 le *fanout*[1] dans le multiplieur, nous limitons ainsi la chuter de la fréquence d'horloge de notre coprocesseur dû à ce *fanout*. De plus il suffit de permuter a^+ et a^- pour obtenir l'opposé d'un coefficient, ce qui est gratuit en matériel.

3.1.2 Addition, multiplication

L'addition sur $\mathbb{F}_{p^m}$ consiste à sommer les opérandes coefficient à coefficient.

En ce qui concerne la multiplication nous cherchons à obtenir un opérateur relativement compact. Nous utilisons un multiplieur suivant le paradigme parallèle-série. Cela consiste à traiter un opérande en parallèle et traiter l'autre séquentiellement. Song et Parhi ont proposé un multiplieur qui traite D coefficients du second opérande à chaque cycle, il faut ainsi $\lceil m/D \rceil$ cycles par multiplication [27]. C'est ce type de multiplieur que nous utilisons. Nous pouvons choisir D ; il y a donc un compromis à trouver entre :

- un multiplieur volumineux qui traite une multiplication en peu de cycles ; et
- un multiplieur compact qui sera plus lent (diminuer D).

Notons que ce compromis est limité par le fait que si l'on augmente trop D la fréquence d'horloge du multiplieur baisse.

3.1.3 Frobenius et Frobenius inverse

Élever à la puissance p — c'est-à-dire appliquer le Frobenius — fixe les éléments de $\mathbb{F}_p$. Soit $u = u_{m-1}X^{m-1} + \cdots + u_1X + u_0 \in \mathbb{F}_{p^m}$. Par linéarité du Frobenius $u^p = u_{m-1}X^{pm-p} + \cdots + u_1X + u_0$, il suffit donc de calculer les $X^{pm-p}, \ldots, X^p$ modulo le polynôme irréductible f pour obtenir chaque coefficient de u^p sous la forme d'une combinaison linéaire des u_i. Il en est de même pour l'extraction de racine p-ième. Selon le choix du polynôme irréductible f, les coefficients de u^p sont des sommes d'un nombre plus ou moins grand de coefficients de u. La taille des ces sommes est l'élément déterminant la longueur du chemin critique

1. nombre maximal de portes logiques qui utilisent la sortie d'une autre porte.

d'un opérateur réalisant le Frobenius (ou le Frobenius inverse). Nous choisissons f de manière à maximiser cette fréquence (f est généralement un trinôme ou un pentanôme).

Exemple 2. Sur le corps $\mathbb{F}_{3^{11}}$ avec $X^{11} - X^2 + 1$ comme polynôme irréductible, le Frobenius est plus coûteux que sa réciproque. Comme chaque coefficient peut être traité en parallèle, le coût est donné par le nombre maximal de termes à sommer pour obtenir chaque coefficient du résultat , ici : 4 pour le Frobenius et 3 pour le Frobenius inverse.

$$\begin{aligned}(a_{10}X^{10} + \cdots + a_1X + a_0)^3 &= a_{10}X^{30} + \cdots + a_1X^3 + a_0 \\ = \quad & (a_{10} - a_7)X^{10} + (a_3 + a_6 + a_9)X^9 + a_{10}\ X^8 \\ + \quad & (a_9 - a_6)X^7 + (a_2 + a_5 + a_8)X^6 + a_9\ X^5 \\ + \quad & (a_8 - a_5)X^4 + (a_1 + a_4 + a_7 + a_{10})X^3 + a_8\ X^2 \\ - \quad & (a_4 + a_7 + a_{10})X + a_0\end{aligned}$$

$$\begin{aligned}\sqrt[3]{a_{10}X^{10} + \cdots + a_1X + a_0} &= a_{10}\sqrt[3]{X}^{10} + \cdots + a_1\sqrt[3]{X} + a_0 \\ = \quad & a_8\ X^{10} + a_5\ X^9 + a_2\ X^8 + (a_8 - a_{10})X^7 \\ + \quad & (a_5 - a_7)X^6 + (a_2 - a_4)X^5 - (a_1 - a_8 - a_{10})X^4 + (a_5 + a_7 + a_9\ )X^3 \\ + \quad & (a_2 + a_4 + a_6\ )X^2 + (a_1 + a_3)X + a_0\end{aligned}$$

3.2 Un modèle de coprocesseur arithmétique

3.2.1 Modèle général

Nous avons vu comment réaliser les opérateurs arithmétiques pour le corps de base, nous devons maintenant les assembler pour former un coprocesseur capable d'exécuter, entre autres, les algorithmes que nous avons décrits au Chapitre 2. Ces algorithmes contiennent des calculs que nous pouvons paralléliser, il nous faut donc proposer une architecture permettant d'effectuer plusieurs opérations à chaque cycle d'horloge.

De plus les opérations arithmétiques que nous réalisons n'ont pas toutes le même temps de calcul : la multiplication est beaucoup plus coûteuse que le reste des opérations qui peuvent être réalisées en un cycle à une fréquence raisonnable — pour les valeurs de m et D qui nous concernent, il faut entre 5 et 50 cycles pour une multiplication. Les multiplications se retrouvent ainsi sur les chemins critiques de nos calculs, il faut donc pouvoir recouvrir les multiplications avec les autres opérations.

Toutefois, les algorithmes que nous manipulons ne sont pas réguliers. Nous ne pouvons donc pas dessiner une architecture spécifique à chaque algorithme : trop d'opérateurs seraient inutilisés à chaque étape du calcul.

Enfin, nous avons besoin d'utiliser un grand nombre de variables intermédiaires au cours du calcul, nous avons donc besoin d'un banc de registres à la manière des processeurs généralistes pour les stocker. Cependant un passage systématique par ces registres serait trop coûteux, en effet il faut deux cycles pour écrire puis lire dans la mémoire.

Nous proposons donc un modèle de coprocesseur (voir Figure 3.1 p.9) comportant des unités de calcul capables de réaliser une ou plusieurs opérations arithmétiques, une unité de mémoire accessible par un ou deux ports en lecture et en écriture ainsi qu'un réseau creux reliant les sorties des unités aux entrées des autres.

Notons qu'à chaque cycle les unités et le réseau attendent un certain nombre de bits de contrôle pour sélectionner les opérations à réaliser, les routes à emprunter et les adresses à utiliser en mémoire. Ces bits de contrôle sont fournis par un petit automate qui les lit dans une mémoire morte.

3.2.2 Réseau creux

Le rôle du réseau est de transporter les valeurs depuis/vers les unités de calcul et la mémoire. Les valeurs transportées sont des éléments de $\mathbb{F}_{p^m}$, chaque route du réseau a une largeur comprise entre environ 200 et 500 bits en considérant les corps que nous utilisons. Étant donnée cette largeur, dessiner un réseau contenant tous les liens est trop coûteux.

Cependant il peut être utile pour une unité de calcul de sélectionner les sources des ses entrées. Cela permet par exemple d'éviter des passages par la mémoire entre les applications successives du Frobenius que l'on trouve dans les algorithmes considérés. Cette sélection demande alors d'utiliser des multiplexeurs. Ces multiplexeurs occupent une surface non négligeable et ont une influence sur la fréquence d'horloge atteignable,

il est donc indispensable de bien dimensionner le réseau en adéquation avec l'algorithme utilisé ; la conception de notre compilateur a été grandement motivée par la volonté d'explorer rapidement différentes solutions architecturales au niveau de ce réseau.

Des registres peuvent être placés dans ce réseau pour couper les différents chemins critiques et ne pas perdre en fréquence d'horloge. Comme les différentes routes ne traversent pas le même nombre de multiplexeurs et proviennent d'unités dont les latences peuvent varier, des registres peuvent être placés à différents endroits du réseau ; nous obtenons ainsi un réseau dont les délais en nombre de cycles d'horloge varient en fonction des routes.

Nous donnons un exemple de réseau creux dans la Figure 3.2.

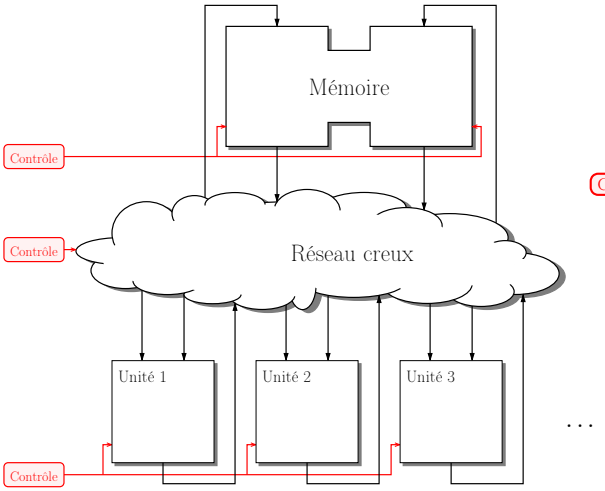

FIGURE 3.1 – Modèle de coprocesseur arithmétique

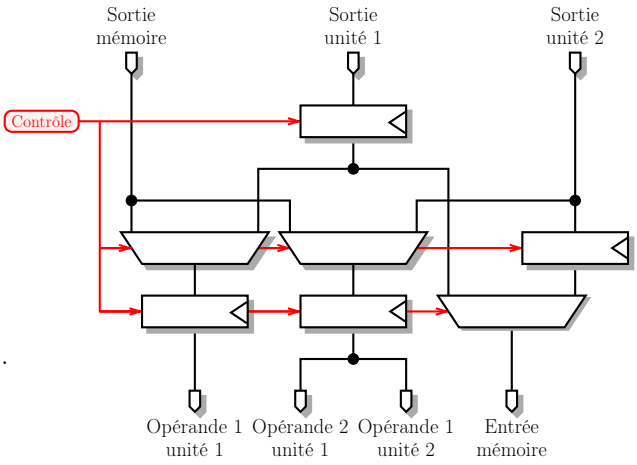

FIGURE 3.2 – Un exemple de réseaux creux

Nous montrons dans la Section suivante comment combiner plusieurs opérateurs en une seule unité de calcul à travers la description d'un des coprocesseurs que nous avons utilisés [7]. Encore une fois différentes combinaisons d'opérateurs peuvent être envisagées et notre compilateur est là pour nous permettre d'en explorer plusieurs.

3.3 Un coprocesseur pour l'exponentiation finale en caractéristique 3

La multiplication étant lente par rapport aux autres opérations, nous avons choisi de réaliser une architecture avec deux unités de calcul : une unité `M` pour le multiplieur et une unité `A` pour les autres opérations : les additions, soustractions et Frobenius.

Nous allons détailler les motivations qui ont conduit à la conception du coprocesseur présenté en Figure 3.3. Comme nos algorithmes comportent une proportion non négligeable d'accumulations et de séquences de Frobenius, nous mettons une route de l'unité `A` vers l'un de ces opérandes. Nous remarquons aussi que les multiplications sont souvent précédées d'additions préparant les opérandes et suivies d'additions utilisant le résultat mais quasiment jamais de multiplications, c'est pourquoi nous mettons des routes de l'unité `A` à l'unité `M` et inversement mais pas de `M` vers elle-même.

L'algorithme d'exponentiation finale comprend de longues séquences de Frobenius, nous donnons donc à notre unité `A` la capacité de calculer deux Frobenius en un cycle. L'architecture de cette unité est donnée dans la Figure 3.4. Constatons que notre unité `A` est capable de calculer : une somme, l'opposé d'une somme, une différence, l'application d'un ou deux Frobenius ou leur opposé.

Étant donné que nous avons placé un certain nombre de routes directes entre les unités de calcul, nous ne devrions pas avoir une pression trop forte en écriture dans les registres. C'est pourquoi nous ne gardons qu'un port d'écriture dans la mémoire. Par contre les deux ports en lecture seront utiles pour alimenter les unités de calcul en opérandes.

Pour contrôler ce coprocesseur il faut donner une valeur, pour chaque cycle, à chacun des 19 bits contrôlant les registres, les multiplexeurs et les unités de calcul, ainsi qu'aux deux adresses correspondant au deux ports de la mémoire.

Cette architecture de coprocesseur est celle que nous avons présentée dans l'article [7] dans la Section 4.2. La Figure 3.5 montre l'architecture complète du coprocesseur.

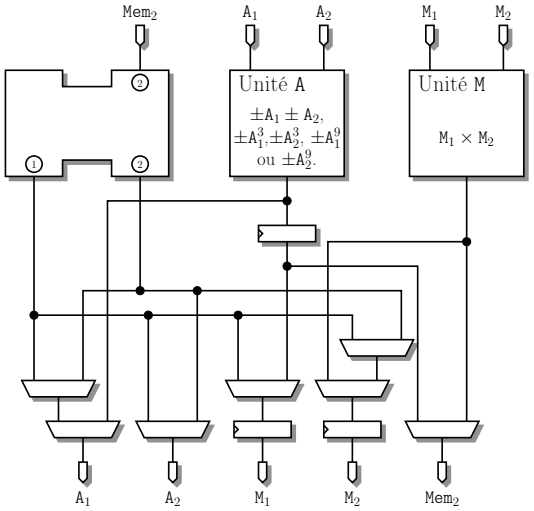

FIGURE 3.3 – Un coprocesseur pour l'exponentiation finale

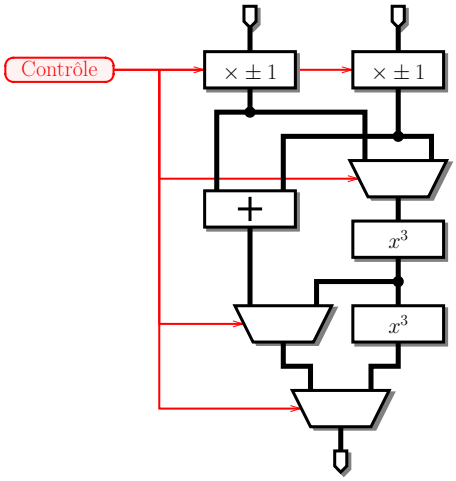

FIGURE 3.4 – L'unité A

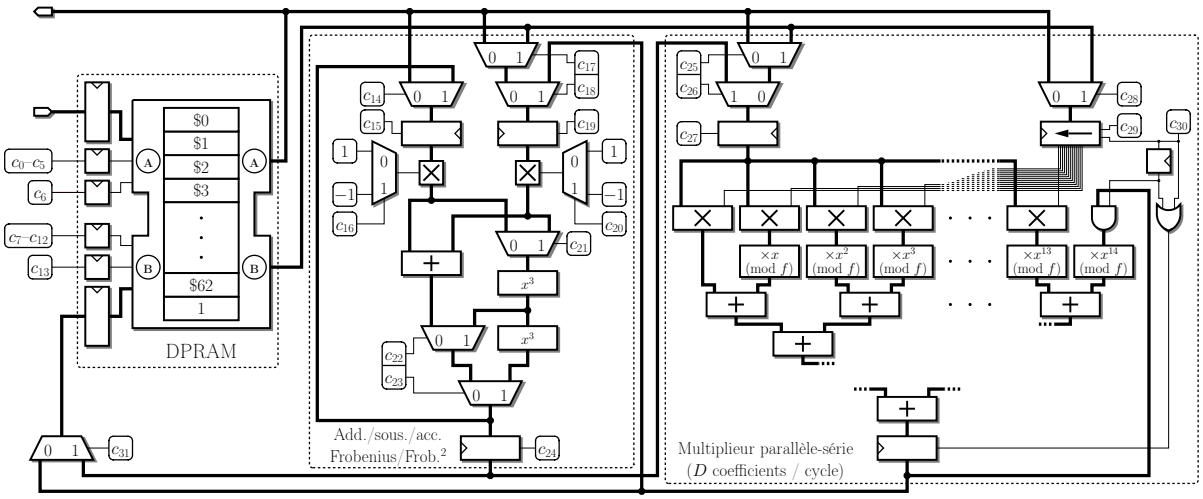

FIGURE 3.5 – Architecture complète du coprocesseur

Chapitre 4

Ordonnancement et compilation

Nous avons décrit un ensemble d'algorithmes arithmétiques (Chapitre 2) que nous voulons pouvoir exécuter sur un coprocesseur suivant un certain modèle (Chapitre 3). Il nous faut maintenant décrire un algorithme de compilation permettant de passer automatiquement de la description du calcul au contrôle du coprocesseur. Cette compilation passe par l'ordonnancement des différentes opérations composant le calcul. La conception d'un tel outil nous permettra d'explorer un grand nombre de compromis mathématiques et architecturaux pour une meilleur adéquation algorithme/architecture.

4.1 État de l'art

Les problèmes d'ordonnancement ont été très largement étudiés dans la littérature. Nous plaçons ici notre problème d'ordonnancement par rapport aux problèmes connus.

Tout d'abord, il faut représenter notre calcul : il s'agit d'un certain nombre de tâches (ici des opérations sur $\mathbb{F}_{p^m}$) qui dépendent les unes des autres, ces tâches forment les nœuds d'un graphe orienté où les arêtes représentent les dépendances. Ce graphe est nécessairement acyclique : une tâche ne peut pas dépendre de sa production. L'ordonnancement d'un tel graphe est NP-complet pour un ensemble d'éléments de calcul) interconnectés (nœuds d'un cluster, unités arithmétiques d'un processeur, ... [20, section 3]).

Commençons par différencier l'ordonnancement statique du dynamique : dans le cas statique l'ensemble des tâches à exécuter et des dépendances entre elles est connu à l'avance ; contrairement au cas dynamique où les tâches arrivent au fur et à mesure et les contraintes de performances ne permettent pas de mettre en place des algorithmes coûteux. Notre ordonnancement est statique, en effet nous devons générer tout le contrôle de notre coprocesseur avant de pouvoir lancer le calcul.

La plupart des modèles étudiés dans la littérature prend en compte des coûts de communications pour le transfert de données entre les différents éléments de calcul. Ces coûts de communication peuvent devenir prohibitifs et il a été proposé de dupliquer certaines tâches sur plusieurs éléments de calcul plutôt que d'utiliser les connexions. Cette approche permet de réduire le temps de calcul total dans certains cas. Cependant cela n'est pas applicable dans notre cas, en effet nos unités ne sont pas polyvalentes et un calcul ne peut généralement être effectué que par une seule unité.

Différentes heuristiques ont été proposées pour résoudre les problèmes d'ordonnancement. Il y a deux grandes familles d'algorithmes qui peuvent s'appliquer dans notre cas : les ordonnancements de listes [1] et l'exploration d'un espace d'ordonnancements valides. Étant donnée l'explosion combinatoire qu'engendre l'ensemble des ordonnancements valides, cet espace de solutions est parcouru de différentes manières : méthodes séparation-évaluation (*branch-and-bound*), parcours A^*, recherche locale, recuit simulé, algorithme génétique, etc. (voir [11] pour une comparaison de celles-ci). Nous avons choisi de concevoir des heuristiques appartenant à la première famille pour leur simplicité, cependant nous envisageons d'étudier les autres dans des travaux futurs (Section 4.4).

4.2 Pipeline de compilation

Nous décrivons dans cette section les différentes étapes de compilation.

4.2.1 Description du contrôle du coprocesseur

Examinons plus en détail le contrôle de notre modèle d'architecture. Nous avons deux types de signaux de contrôle :

– les bits de contrôle des multiplexeurs, des registres et des unités de calcul, et
– les adresses du banc de registre.

Un bit de contrôle peut être soit forcé à 1, forcé à 0 ou ne pas avoir de valeur imposée. De même une adresse peut voir sa valeur imposée ou non ; cependant un registre de la mémoire ne peut pas être lu et écrit durant le même cycle d'horloge dans la mémoire, nous maintenons donc dans la structure de données représentant un signal d'adresse une liste des valeurs qu'il ne peut pas prendre. Ceci nous permet de nous prémunir des collisions dans le banc de registres.

Sélectionner une route dans le réseau ou une fonction à réaliser sur une unité revient à forcer la valeur de certains de ces signaux. La description du coprocesseur consiste ainsi à donner : d'une part, la liste des fonctions que peuvent réaliser les unités, chacune munie de l'ensemble des signaux qu'elle force et les ports d'entrée utilisés ; et d'autre part, la liste des routes du réseau et les signaux qu'il faut forcer pour les emprunter. Nous appelons réalisation d'une fonction ou d'une route l'ensemble des signaux qu'il faut forcer. Notons que chaque fonction peut avoir plusieurs réalisations.

Ainsi, construire le programme de contrôle de notre coprocesseur consiste à combiner différentes réalisations en s'assurant que si un signal est forcé à la même date par plusieurs réalisations il le soit à la même valeur.

4.2.2 Description de l'algorithme à exécuter

Nous représentons les calculs à compiler vers le programme de contrôle du coprocesseur comme un graphe acyclique orienté (DAG pour *Directed Acyclic Graph*). Les nœuds du graphe représentent une opération à calculer sur ses parents. Ces opérations sont à choisir parmi les fonctions que le coprocesseur considéré est capable de réaliser. Un exemple d'un tel graphe est donné à la Figure 4.1. Les nœuds en forme de triangle vers le bas représentent les lectures en mémoire, les autres représentent des opérations ; si une opération a une forme de triangle vers le haut, cela signifie que son résultat doit être écrit en mémoire.

```
exemple  = funcgraph.FuncGraph("exemple")

u0 = Ld('u0', exemple)   // Lecture en mémoire de la variable u0
u1 = Ld('u1', exemple)
v0 = Ld('v0', exemple)
v1 = Ld('v1', exemple)
a0 = FrobFrob(u0)        // a0 <- u0^(p^2)
a1 = NegFrobFrob(u1)     // a1 <- - u1^(p^2)
b0 = Add(u0, v0)         // b0 <- u0 + v0
b1 = Sub(u1, v1)         // ...
t0 = Mul(a0, b0)
t1 = Mul(a1, b1)
t2 = Mul(Add(a0, a1),
         Add(b0, b1))
Str(Add(t0, t2), 'r0')
Str(t1, 'r1')            // Ecriture en mémoire du contenu
                         // de t1 dans la variable r1

exemple.getDotGraph('graphe.dot')
```

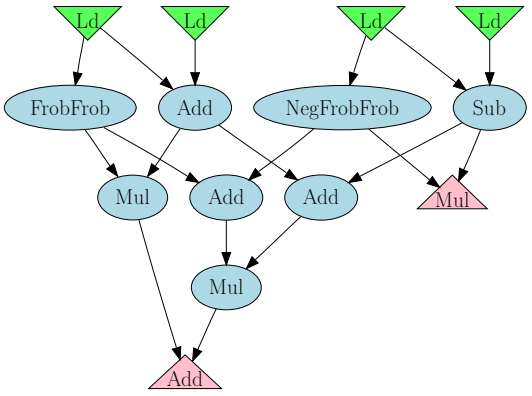

FIGURE 4.1 – Un exemple de saisie d'un algorithme et le graphe correspondant

4.2.3 Transformations sur le graphe

Avant de commencer l'ordonnancement des nœuds du DAG, nous effectuons un ensemble de transformations sur le graphe. Ces transformations vont nous permettre de créer un graphe de sous-arbres du DAG en entrée, nous ordonnancerons alors ces sous-arbres. Ces sous-arbres sont des parties du calcul que nous pouvons exécuter sans avoir à passer par la mémoire. Nos transformations vont donc consister à insérer des *spills* — c'est-à-dire des passages par la mémoire : une écriture, puis une lecture — sur certaines arêtes du DAG. Nous cherchons à n'insérer qu'un minimum de *spills*, en effet nous avons déjà vu qu'un passage par la mémoire est coûteux.

La première transformation consiste à couper les arêtes qui correspondent à des routes indisponibles dans le réseau. Par exemple dans l'architecture que nous proposons en Section 3.3, il n'y a qu'un multiplieur et il n'y pas de route partant de sa sortie et arrivant sur l'une de ses deux entrées, ainsi il faut insérer un *spill* sur les arêtes multiplication vers multiplication du DAG.

Il peut aussi y avoir une collision dans les ressources nécessaires au calcul des deux parents d'un nœud binaire : par exemple, et toujours dans le cas où nous ne disposons que d'un multiplieur, une des deux

multiplications impliquées dans la somme de deux produits doit être *spillée*. De même il peut y avoir une collision dans l'utilisation du résultat d'un calcul : par exemple lorsqu'un résultat est utilisé pour deux multiplications différentes alors que nous ne disposons que d'un multiplieur dans le coprocesseur concerné. Dans ce cas nous insérons des *spills* pour tous les nœuds en collision sauf celui qui a la plus grande priorité. Il y a plusieurs définitions possible pour cette priorité, nous les discutons à la Section 4.3.1.

Un autre sous-graphe que notre modèle d'architecture ne peut traiter correspond au cas où l'un des deux parents d'un nœud binaire est nécessaire au calcul du deuxième parent (voir Figure 4.2, la flèche en pointillé indique une dépendance de données en mémoire). En effet cette dépendance triangulaire des opérations empêche les opérandes du nœud le plus profond d'arriver de façon synchrone.

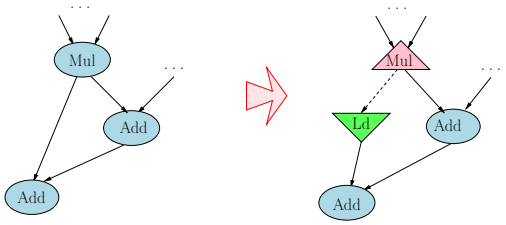

FIGURE 4.2 – Un exemple de dépendance triangulaire dans le DAG qu'il faut éliminer.

Nous insérons enfin des *spills* afin que tous les nœuds n'aient au plus qu'un seul fils qui ne soit pas une lecture en mémoire. Ainsi, si nous ne tenons pas compte des liens d'une écriture vers une lecture, nous obtenons un ensemble de sous-arbres du DAG initial. Ces sous-arbres forment les nœuds du graphe que nous fournissons à notre algorithme d'ordonnancement ; les arêtes de ce graphe sont fournies par les arêtes écriture vers lecture du DAG précédent, elles représentent les dépendances de données sur le banc de registre. De cette façon, chaque sous-arbre n'utilise que les routes directes entre les unités de calcul pour ses communications. Ainsi cette structure de graphe des sous-arbres nous permet d'exhiber les dépendances entres opérations qui n'utiliseront le banc de registres.

4.2.4 Ordonnancement du graphe

Nous avons choisi une heuristique d'ordonnancement de listes pour traduire notre graphe de sous-arbres en un programme de contrôle linéaire. Un ordonnancement de listes peut être ascendant ou descendant. Nous avons choisi d'ordonnancer les sous-arbres à partir des puits du graphe des sous-arbres, c'est-à-dire à partir de la fin, ainsi nous fixons la date d'utilisation d'une donnée avant celle de production.

L'ordonnancement de listes que nous avons conçu est présenté dans l'Algorithme 1. Il consiste à maintenir une liste des sous-arbres prêts à être ordonnancés, c'est-à-dire les sous-arbres dont les lectures associées aux écritures qu'ils contiennent ont toutes été ordonnancées. Ainsi, pour chacun de ces sous-arbres, la date la plus tardive à laquelle ils peuvent être ordonnancés est connue. Puis il faut choisir l'un de ces sous-arbres prêts à être ordonnancés et tenter de l'exécuter à la date associée : en cas d'échec (conflit d'utilisation des unités ou congestion mémoire) il faudra avancer la date où il peut être ordonnancé. L'algorithme boucle jusqu'à ce que tous les sous-arbres aient été ordonnancés. La fonction de choix du sous-arbre (ligne 5) à ordonnancer est déterminante dans la performance de l'algorithme. Nous en détaillons plusieurs dans la Section 4.3.

4.2.5 Allocation de registres

Une fois que nous avons obtenu la suite des instructions de contrôle correspondant au programme compilé, il nous reste à donner des adresses physiques aux variables que l'on a stockées sur le banc de registre. En effet, à cette étape de la compilation nous avons ordonnancé toutes les écritures et toutes les lectures en mémoire en considérant que nous avions une mémoire infinie à la manière des représentations SSA [1].

Pour attribuer une adresse physique à chacune des adresses virtuelles, nous commençons par déterminer la vivacité des adresses virtuelles : une variable est vivante depuis son écriture jusqu'à sa dernière lecture. Puis nous déterminons le nombre maximal d'adresses utilisées et enfin nous attribuons une adresse physique à chaque adresse virtuelle de manière gloutonne.

1. *Single Static Assignment* : chaque variable n'est écrite qu'une seule fois en mémoire.

Algorithme 1 Ordonnancement de listes remontant

Entrée: Description de l'architecture, graphe des sous-arbres.
Sortie: Suite des mots de contrôle.

```
1.  Pour tous les puits du graphe des sous-arbres faire :
2.      Ajouter le puits à l'ensemble PrêtÀOrdonnancer
3.      Marquer ce puits comme à réaliser pour la date 0
4.  Tant que PrêtÀOrdonnancer n'est pas vide faire :
5.      Choisir un sous-arbre A dans PrêtÀOrdonnancer
6.      t ← la date avant laquelle A doit être réalisé
7.      Si A peut être réalisé pour la date t alors :
8.          Mettre à jour la suite des mots de contrôle
9.          Mettre à jour le diagramme de Gantt d'utilisation des unités.
10.         Pour tous les nœuds de A qui sont une lecture en mémoire faire :
11.             Marquer l'écriture correspondante comme à réaliser avant la date de lecture
12.             Si toutes les lectures correspondants à cette écriture ont été ordonnancées alors
13.                 Ajouter l'arbre correspondant à cette écriture dans dans PrêtÀOrdonnancer
14.         Enlever A de PrêtÀOrdonnancer
15.     Sinon :
16.         Trouver la date t' antérieure à t où il existe une unité de calcul libre pouvant réaliser
                la racine du sous-arbre A dans le diagramme d'utilisation des unités
17.         Marquer A comme à réaliser pour la date t'
```

4.3 Heuristiques d'ordonnancement de listes

Nous détaillons dans cette Section les fonctions de choix d'un sous-arbre dans la liste de ceux qui sont prêts à être ordonnancés que nous avons utilisées et développées.

4.3.1 Sélection par priorité

L'approche la plus courante des algorithmes d'ordonnancement de listes est d'attribuer une priorité à chacun des nœuds du graphe et de choisir celui qui a la plus grande priorité [28]. Cette priorité peut être définie de différentes façons, nous en avons essayé plusieurs. La plus naturelle est certainement de considérer la distance aux sources du DAG : plus un nœud est situé loin des sources, plus il faut l'ordonnancer tard, d'autres nœuds pourront être ordonnancés avant. Pour tenir compte de l'hétérogénéité du temps d'exécution des différentes fonctions (les multiplications sont plus lentes que les additions par exemple) nous avons aussi calculé cette distance en tenant compte du temps d'exécution pour chaque nœud. De même nous avons essayé d'utiliser la fonction qui associe à un nœud la somme des temps d'exécution de chacun de ces prédécesseurs comme fonction de priorité. De façon duale, nous avons aussi développé la distance au puits du graphe, cette même distance pondérée par les temps d'exécution et la somme des temps d'exécution de tous les successeurs comme fonctions de priorité, dans ce cas il faut sélectionner le nœud avec la priorité minimale et non maximale.

4.3.2 Rapprochement maximum aux nœuds déjà ordonnancés

Cependant les fonctions de priorité précédentes ne nous donnaient pas satisfaction. En effet choisir le sous-arbre à ordonnancer grâce à une priorité tend à favoriser un parcours en largeur du graphe et la forme particulière du graphe obtenu avec le calcul de l'exponentiation finale est un pire cas pour ce type d'heuristique : l'exponentiation finale se termine par un certain nombre de multiplications (lentes) précédées par de longues chaînes de Frobenius ; ces opérations étant réalisées par deux unités distinctes, nous avons tout intérêt à les exécuter en parallèle. Pour ce faire nous devons concevoir une fonction de choix qui favorise un parcours plus en profondeur : un choix qui permet de sélectionner d'abord un petit groupe de multiplications permettant de calculer au plus tard une des chaînes de Frobenius et ainsi la recouvrir avec d'autres multiplications.

Dans la Figure 4.3 nous montrons un exemple de graphe qui met la sélection par priorité en échec. Supposons que les nœuds carrés ne puissent être réalisés que par une unité et les ronds par une autre. Les nœuds ronds 1, 2 et 3 ne sont pas distinguables par une priorité. Pour autant si le nœud 1 est déjà ordonnancé, nous devons choisir d'ordonnancer au plus tard le nœud 2. Rappelons que nous utilisons un ordonnancement de listes par le bas, ordonnancer un nœud au plus tard revient donc à le traiter de façon prioritaire. En effet

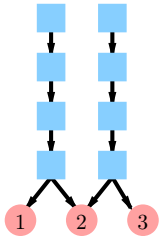

FIGURE 4.3 – Un pire cas pour la sélection par priorité.

choisir le nœud 2 plutôt que le nœud 3 nous permet d'exécuter au plus tard les nœuds carrés qui les précèdent et réduire ainsi le temps total d'exécution.

Fort de ces constatations, nous cherchons une nouvelle heuristique pour le choix du sous-arbre à ordonnancer. Il nous faut favoriser un parcours plus en profondeur qui permettrait d'ordonnancer à la suite les groupes de sous-arbres menant à rapprocher un maximum de nœuds de ce qui ont déjà été ordonnancés. Comment calculer ce rapprochement ? Soit X l'ensemble des nœuds des sous-arbres déjà ordonnancés, Y celui de ceux qui sont prêts à l'être et Z l'ensemble des autres sous-arbre. Nous calculons dans un premier temps la somme des distances minimales de chaque nœud de $Y \cup Z$ aux nœuds déjà ordonnancés. Puis nous calculons pour chaque sous-arbre A prêt à être ordonnancé, cette même somme en enlevant les nœuds de A de l'ensemble Y pour les rajouter à X. Le sous-arbre prêt à ordonnancer qui maximise la différence entre ces deux sommes est alors sélectionné. Ainsi nous espérons nous rapprocher au plus vite des longues chaînes de Frobenius et pouvoir les recouvrir par d'autres opérations.

4.3.3 Sélection par estimation optimiste

L'approche précédente a été conçue grâce à l'analyse des performances obtenues avec des heuristiques de sélection par priorité. Nous avons donc aussi cherché une heuristique qui demande moins de connaissance *a priori* des calculs à ordonnancer.

Nous avons déjà vu que nous avons besoin d'une heuristique prenant en compte l'ordonnancement déjà réalisé. Ainsi nous proposons de choisir parmi tous les sous-arbres prêts à être ordonnancés celui qui donnera le plus court temps total d'exécution pour l'ensemble du programme. Nous ne pouvons pas calculer ce temps d'exécution sans l'ordonnancement complet du programme, c'est pourquoi nous en faisons une estimation optimiste. Cette approximation est calculée en considérant que les différentes unités sont toujours libres : un calcul peut toujours être fait le plus tard possible (ALAP : *As Late As Possible* [20]). Ainsi pour chaque sous-arbre prêt à être ordonnancé, nous supposons que nous l'exécutons à la date qui lui est associée et nous estimons alors le temps d'exécution du programme complet, le sous-arbre qui donne la meilleure estimation est alors sélectionné.

4.4 Propositions pour des travaux futurs

Nous avons proposé un algorithme d'ordonnancement de listes pour la compilation de calculs arithmétiques vers notre modèle de coprocesseur. Cependant les particularités de notre architecture ne nous permettent pas d'exploiter simplement cet algorithme qui est mieux adapté à l'ordonnancement d'instructions pour un processeur généraliste ou l'ordonnancement de tâches sur un *cluster*. En effet ce coprocesseur est très hétérogène : chaque fonction que réalise notre coprocesseur ne peut être réalisée sur toutes ses unités. De plus le modèle de coût des communications est assez rigide : les données doivent arriver à la date exacte où elle seront utilisées, il n'est pas possible de conserver dans une file les opérandes qu'une unité de calcul utilisera plus tard.

Ce qui limite le plus les performances de notre algorithme d'ordonnancement est certainement que, une fois qu'il a ordonnancé un sous-arbre, il ne s'autorise pas à remettre ce choix en cause. En effet nous avons constaté que des informations strictement locales ne permettent pas nécessairement d'ordonnancer à la suite des calculs qui mènent à une même partie du graphe (Section 4.3.2, contre-exemple de la Figure 4.3). Or les graphes que nous traitons ne sont pas aussi simples que celui exhibé en contre-exemple : les multiplications sont généralement entourées par des additions qui *cachent* la forme globale du graphe à l'heuristique de sélection du sous-arbre. Ainsi nous pensons qu'il pourrait être très intéressant de pouvoir revenir sur cette sélection après avoir ordonnancé les éventuels nœuds cachant la topologie du graphe. Pour cela nous proposons d'utiliser une heuristique du type A^*. Les heuristiques A^* ont été utilisées avec succès pour calculer des ordonnancements particulièrement dans les cas de compilation vers une architecture [12] ou un *cluster* [17] très hétérogène.

Chapitre 5

Résultats et expérimentations

Nous présentons dans ce Chapitre les performances que notre couple architecture/compilateur permettent d'atteindre après avoir comparé expérimentalement les différentes heuristiques d'ordonnancement de listes présentées dans la Section 4.3.

5.1 Comparaison des heuristiques de compilation

Nous avons comparé nos différentes approches de compilation sur le calcul d'exponentiation finale en caractéristique 3 pour $m = 97$ et $m = 193$. L'architecture que nous avons utilisée est celle que nous avons décrite dans la Section 3.3, précisons que le multiplieur utilisé traite une multiplication tous les 7 cycles d'horloge sur $\mathbb{F}_{3^{97}}$ ($D = 14$) et une tous les 15 cycles sur $\mathbb{F}_{3^{193}}$ ($D = 13$). Nous disposons pour cette architecture et ces exponentiations finales d'ordonnancements programmés *à la main* [7], ce qui nous a permis de comparer de façon plus objective nos heuristiques de compilation.

La performance du compilateur se traduit par sa capacité à recouvrir au maximum les multiplications par les autres opérations (additions, soustractions, Frobenius, etc.) ; en effet, le temps de calcul associé aux multiplications (574 cycles d'horloge pour $\mathbb{F}_{3^{97}}$, 1245 cycles pour $\mathbb{F}_{3^{193}}$) est plus grand que pour les autres opérations (377 cycles d'horloge pour $\mathbb{F}_{3^{97}}$, 569 cycles pour $\mathbb{F}_{3^{193}}$). Cependant dans le cas de $\mathbb{F}_{3^{97}}$, les multiplications sont relativement rapides et l'ordonnancement des autres opérations devient crucial : dans ce cas aucune de nos heuristiques ne nous permet d'atteindre les performances de l'ordonnancement que nous avons réalisé sans l'aide du compilateur. Par contre sur $\mathbb{F}_{3^{193}}$, les multiplications sont plus longues et la pression est moins grande sur le compilateur qui atteint des performances comparables à celles de l'ordonnancement manuel.

C'est l'heuristique de rapprochement maximum aux nœuds déjà ordonnancés (Section 4.3.2) qui nous fournit le meilleur ordonnancement sur $\mathbb{F}_{3^{193}}$. Nous expliquons ceci par le fait que cette heuristique semble la plus adaptée à recouvrir les chaînes de Frobenius associées à l'élévation à la puissance $3^{\frac{m+1}{2}}$ qui semble être l'un des goulots d'étranglement du calcul d'exponentiation finale pour ce coprocesseur.

Corps de base	Heuristique de compilation	Temps de calcul pour l'exponentiation finale (cycles)	Taux d'utilisation de l'unité `A`	Taux d'utilisation de l'unité `M`
$\mathbb{F}_{3^{97}}$	Manuel	654	58%	88%
	Priorité (distance aux sources)	819	46%	70%
	Priorité (poids des prédécesseurs)	815	46%	70%
	Rapprochement maximum	816	46%	70%
	Estimation optimiste	827	45%	69%
$\mathbb{F}_{3^{193}}$	Manuel	1427	40%	86%
	Priorité (distance aux sources)	1578	36%	79%
	Priorité (poids des prédécesseurs)	1579	36%	79%
	Rapprochement maximum	1472	39%	85%
	Estimation optimiste	1549	37%	80%

TABLE 5.1 – Comparaison des différentes heuristiques d'ordonnancement de listes

5.2 Expérimentations

5.2.1 Coprocesseur d'exponentiation finale

Nous avons implémenté le coprocesseur pour l'exponentiation finale en caractéristique 3 décrit à la Section 3.3 [7].

Nous avons également expérimenté le cas de l'exponentiation finale en caractéristique 2 sur les corps suivants : $\mathbb{F}_{2^{239}}$ et $\mathbb{F}_{2^{313}}$. Comme le Frobenius est linéaire, la composition de deux Frobenius l'est aussi ; si $u \in \mathbb{F}_{2^m}$, les coefficients de u^{2^2} sont des combinaisons linéaires des coefficients de u. Or nous avons trouvé des trinômes irréductibles pour construire $\mathbb{F}_{2^{239}}$ et $\mathbb{F}_{2^{313}}$ tels que cette combinaison linéaire soit peu coûteuse. Ainsi nous constatons qu'un opérateur calculant le double Frobenius fonctionne à une fréquence raisonnable. Nous dessinons donc un processeur identique à celui que nous utilisons pour la caractéristique 3 mais dont l'un des opérateurs pour le Frobenius de l'unité `A` a été remplacé par un opérateur de double Frobenius, l'unité est ainsi capable de calculer des Frobenius et des triples Frobenius.

Nous avons implémenté nos coprocesseurs en VHDL et nous les avons placé et routé sur des circuits reconfigurable FPGA de la gamme Virtex II Pro, nous donnons dans la Table 5.2 les performances de nos coprocesseurs pour le calcul de l'exponentiation finale. Les ordonnancements utilisés ici ont été calculés par notre compilateur avec l'heuristique de rapprochement maximum.

Corps de base	Fréquence (MHz)	Surface (slices[a])	Temps de calcul total (μs)	Produit temps-surface (ms.slices)
$\mathbb{F}_{2^{239}}$	192	3026	3,94	11,9
$\mathbb{F}_{2^{313}}$	175	4001	5,53	22,1
$\mathbb{F}_{3^{97}}$	136	4720	5,99	28,3
$\mathbb{F}_{3^{193}}$	105	8926	13,9	124

TABLE 5.2 – Coprocesseur pour le calcul de l'exponentiation finale

[a] : La *slice* est la brique de base d'un circuit sur un FPGA de la gamme Virtex, elle contient deux portes capables de réaliser n'importe quelle fonction booléenne à 4 entrées ainsi que deux élément de mémorisation.

5.2.2 Variations architecturales

Notre couple architecture/compilateur a été conçu pour nous permettre d'explorer un certain nombre de variations architecturales par rapport à la version proposée dans l'article [7].

Une part importante du circuit est occupée par le multiplieur. Nous pouvons faire varier la taille de celui-ci et le rendre ainsi plus ou moins rapide : il y a donc un compromis à trouver sur D le nombre de coefficients qu'il est capable de traiter en un cycle d'horloge. La table 5.3 présente les performances obtenues pour différents multiplieurs. Afin de comparer ces circuits, nous avons utilisé le produit temps-surface ; celui-ci permet de comparer objective des circuits de tailles différentes : une architecture deux fois plus rapide qu'une autre n'est utile que si l'on n'obtient pas de meilleures performances en utilisant deux instances de la première architecture. Cette comparaison nous a permis de vérifier que les choix que nous avions faits initialement ($D = 15$ pour $\mathbb{F}_{2^{313}}$ et $D = 13$ pour $\mathbb{F}_{3^{193}}$) donnent le meilleur compromis. Il faut cependant nuancer cette conclusion : en effet nous avons vu que notre compilateur ne fournissait pas de bon ordonnancement lorsque la multiplication est rapide, ainsi il est possible que de bons ordonnancements permettent d'améliorer le compromis temps-surface quand D est supérieur à 20.

Corps de base	D	Fréquence (MHz)	Surface (slices)	Temps de calcul total (μs)	Produit temps-surface
$\mathbb{F}_{2^{313}}$	9	181	3367	7,55	25,4
	15	**175**	**4001**	**5,53**	**21,2**
	32	154	6087	4,19	25,5
$\mathbb{F}_{3^{193}}$	10	108	7703	17,7	129
	13	**105**	**8926**	**13,9**	**124**
	20	103	11882	11,2	133
	28	95	15375	10,0	15,4

TABLE 5.3 – Performance du coprocesseur pour différentes tailles de multiplieur

De même nous avons aussi cherché à valider la topologie du réseau creux que nous utilisons. Nous avons donc développé deux autres réseaux pour notre coprocesseur :

- l'un est le réseau qui lie seulement les sorties du banc de registres aux entrées des unités de calcul et les sorties des ces unités au port d'écriture de la mémoire ;
- l'autre est celui proposé initialement où l'on a retiré la route reliant la sortie de l'unité `A` à l'une de ses opérandes.

Cet étude nous a permis de valider l'utilité d'une boucle autour de l'unité `A`. Cependant elle nous a aussi montré que d'utiliser un réseau simple, où toutes les communications passent par la mémoire, donne des performances comparables, le surcoût lié au passage systématique par le banc de registre est compensé par le gain en fréquence et surface obtenu sur le coprocesseur. Nuançons ceci par le fait que l'une des difficultés dans la création de notre compilateur était d'utiliser au mieux les routes entre les unités, un meilleur ordonnancement pourrait permettre de redonner l'avantage à l'architecture proposé dans [7].

Corps de base	Réseau creux	Fréquence (MHz)	Surface (slices)	Temps de calcul total (μs)	Produit temps-surface
$\mathbb{F}_{2^{313}}$	Version initiale	175	4001	5,53	21,2
	Accès à la mémoire seulement	200	3411	6,21	21,2
	Sans la boucle sur `A`	175	3868	6,40	24,8
$\mathbb{F}_{3^{193}}$	Version initiale	105	8926	13,9	124
	Accès à la mémoire seulement	122	8177	14,0	114
	Sans la boucle sur `A`	118	8678	14,4	125

TABLE 5.4 – Performance du coprocesseur pour différents types de réseaux

5.2.3 Utilisation du Frobenius inverse

Nous avons proposé à la Section 2.4.3 de ne calculer l'exponentiation finale en n'utilisant que des Frobenius inverses à la place des Frobenius dans le cas où il serait plus facile à calculer. Or il se trouve que sur $\mathbb{F}_{3^{193}}$, l'opérateur de calcul du Frobenius inverse est plus rapide et plus petit que l'opérateur pour le Frobenius. Cela nous permet d'améliorer les performances de notre coprocesseur (Table 5.5)

Corps de base	Frobenius inverse	Fréquence (MHz)	Surface (slices)	Temps de calcul total (μs)	Produit temps-surface
$\mathbb{F}_{3^{193}}$	non	105	8926	13,9	124
	oui	**115**	**8421**	**10,7**	**107**

TABLE 5.5 – Utilisation du Frobenius inverse pour le calcul d'exponentiation finale

Chapitre 6

Conclusion

Nous avons étudié dans ce rapport différentes perspectives liées au calcul de l'exponentiation finale du couplage de Tate. Dans un premier temps nous avons approfondi les aspects mathématiques qui conduisent à un algorithme arithmétique efficace pour ce calcul. Puis nous avons présenté un modèle de coprocesseur permettant d'exécuter cet algorithme. Enfin nous avons décrit le squelette d'un compilateur permettant de générer automatiquement le programme de contrôle d'un coprocesseur suivant ce modèle.

Ce compilateur a été programmé et nous a permis d'obtenir, dans certains cas, des performances comparables à celles que nous obtenons en faisant l'ordonnancement *à la main*. Cependant notre compilateur a montré ses limites dans d'autres cas : quand nous mettons un multiplieur plus rapide, l'ordonnancement des opérations doit être plus fin et les performances déclinent.

Par conséquent nous souhaiterions donc continuer cette étude et proposer de meilleures heuristiques de compilation. Nous espérons ainsi atteindre de bonnes performances avec des coprocesseurs à multiplieur rapide. De plus notre modèle de coprocesseur permet d'imaginer des architectures possédant plusieurs multiplieurs. Nous avons donc l'intention d'explorer plus largement les choix architecturaux qui ont conduit aux coprocesseurs proposés.

Enfin les coprocesseurs que nous décrivons sont des coprocesseurs arithmétiques sur corps finis. Nous rechercherons donc à exécuter d'autres algorithmes sur ce coprocesseur. Parmi ceux-ci nous avons d'ores et déjà réalisé des études pour le calcul complet de couplage en petite caractéristique : nos résultats préliminaires montrent qu'il est possible de calculer l'intégralité du couplage de Tate (itérations de Miller et exponentiation finale) sur $\mathbb{F}_{3^{193}}$ en 184 μs avec le coprocesseur que nous avons décrit dans la Section 3.3 pour un compromis temps-surface de 1,84 s.slices. Ces résultats sont très encourageants, en effet ce coprocesseur donnent un compromis temps-surface compris entre celui de l'accelérateur parallèle de [7] (0,47 s.slices) et celui de l'opérateur unifié présenté dans [3] (2,46 s.slices). De même nous aimerions expérimenter ce modèle de coprocesseur pour calculer d'autres fonctions cryptographiques comme : le calcul de couplage sur courbes hyperelliptiques, ou encore la multiplication scalaire sur courbes elliptiques et hyperelliptiques.

Bibliographie

[1] T.L. Adam, K.M. Chandy, and JR Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, 1974.

[2] P. Barreto, H.Y. Kim, B. Lynn, and M. Scott. Efficient algorithms for pairing-based cryptosystems. In M. Yung, editor, *Advances in Cryptology – CRYPTO 2002*, number 2442 in Lecture Notes in Computer Science, pages 354–368. Springer, 2002.

[3] J.-L. Beuchat, N. Brisebarre, J. Detrey, and E. Okamoto. Arithmetic operators for pairing-based cryptography. In P. Paillier and I. Verbauwhede, editors, *Proceedings of CHES 2007*, number 4727 in Lecture Notes in Computer Science, pages 239–255. Springer, 2007.

[4] J.-L. Beuchat, N. Brisebarre, J. Detrey, E. Okamoto, and F. Rodríguez-Henríquez. A comparison between hardware accelerators for the modified Tate pairing over $\mathbb{F}_{2^m}$ and $\mathbb{F}_{3^m}$. In S.D. Galbraith and K.G. Paterson, editors, *Pairing 2008*, number 5209 in Lecture Notes in Computer Science, pages 297–315. Springer, 2008.

[5] J.-L. Beuchat, N. Brisebarre, J. Detrey, E. Okamoto, M. Shirase, and T. Takagi. Algorithms and arithmetic operators for computing the η_T pairing in characteristic three. *IEEE Transactions on Computers*, 57(11):1454–1468, November 2008.

[6] J.-L. Beuchat, N. Brisebarre, M. Shirase, T. Takagi, and E. Okamoto. A coprocessor for the final exponentiation of the η_T pairing in characteristic three. In C. Carlet and B. Sunar, editors, *Proceedings of Waifi 2007*, number 4547 in Lecture Notes in Computer Science, pages 25–39. Springer, 2007.

[7] J.-L. Beuchat, J. Detrey, N. Estibals, E. Okamoto, and F. Rodríguez-Henríquez. Hardware accelerator for the Tate pairing in characteristic three based on karatsuba-ofman multipliers. In *Proceedings of CHES 2009*, 2009. To appear.

[8] I.F. Blake, G. Seroussi, and N.P. Smart. *Advances in elliptic curve cryptography*. Cambridge University Press, 2005.

[9] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. In *Advances in Cryptology – CRYPTO 2001*, number 2139 in Lecture Notes in Computer Science, pages 213–229. Springer-Verlag London, UK, 2001.

[10] Dan Boneh, Ben Lynn, and Hovav Shacham. Short Signatures from the Weil Pairing. *Journal of Cryptology*, 17(4):297–319, 2004.

[11] T.D. Braun, H.J. Siegal, N. Beck, L.L. Boloni, M. Maheswaran, A.I. Reuther, J.P. Robertson, M.D. Theys, B. Yao, D. Hensgen, et al. A comparison study of static mapping heuristics for a class ofmetatasks on heterogeneous computing systems. In *Heterogeneous Computing Workshop, 1999.(HCW'99) Proceedings. Eighth*, pages 15–29, 1999.

[12] K.W. Chow and B. Liu. On mapping signal processing algorithms to a heterogeneousmultiprocessor system. In *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*, pages 1585–1588, 1991.

[13] J. Duprat and J.-M. Muller. Écrire les nombres autrement pour calculer plus vite. *Technique et Science Informatique*, 10(3), 1991.

[14] PUB FIPS. 186-2, Digital Signature Standard (DSS). *Gaithersburg, MD, January*, 2000.

[15] G. Frey and H.-G. Rück. A remark concerning m-divisibility and the discrete logarithm in the divisor class group of curves. *Mathematics of Computation*, 62(206):865–874, 1994.

[16] A. Joux. A one round protocol for tripartite Diffie–Hellman. *Journal of Cryptology*, 17(4):263–276, 2004.

[17] M. Kafil and I. Ahmad. Optimal task assignment in heterogeneous computing systems. In *Heterogeneous Computing Workshop, 1997.(HCW'97) Proceedings., Sixth*, pages 135–146, 1997.

[18] D. E. Knuth. *The art of computer programming. Vol. 2: Seminumerical algorithms 3rd ed.* Addison-Wesley Series in Computer Science and Information Processing, 1997.

[19] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, pages 203–209, 1987.

[20] Y.K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[21] S. Lang. *Algebra*. Addison-Wesley, 2nd edition, 1984.

[22] A. J. Menezes, T. Okamoto, and S. A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *Information Theory, IEEE Transactions on*, 39(5):1639–1646, 1993.

[23] V. S. Miller. Short programs for functions on curves. IBM, Thomas J. Watson Research Center, 1986.

[24] V.S. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology CRYPTO'85 Proceedings*, pages 417–426, 1985.

[25] NSA. The case for elliptic curve cryptography. `http://www.nsa.gov/business/programs/elliptic_curve.shtml`.

[26] A. Shamir. Identity-based cryptosystems and signature schemes. In *Proceedings of CRYPTO*, volume 84, pages 47–53. Springer, 1984.

[27] L. Song and K.K. Parhi. Low-energy digit-serial/parallel finite field multipliers. *The Journal of VLSI Signal Processing*, 19(2):149–166, 1998.

[28] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, 2002.

[29] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 2003.

[30] L. C. Washington. *Elliptic Curves: Number Theory and Cryptography*. Chapman & Hall/CRC Press, 2008.