Unités arithmétiques et cryptoprocesseurs matériels pour la cryptographie sur courbe hyperelliptique

Gabriel Gallin

UNIVERSITE BRETAGNE LOIRE MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par

Gabriel GALLIN

"Unités arithmétiques et cryptoprocesseurs matériels pour la cryptographie sur courbe hyperelliptique"

Thèse présentée et soutenue à Rennes, le 29 novembre 2018
Unité de recherche : IRISA UMR 6074

Rapporteurs avant soutenance :

Roselyne CHOTIN-AVOT	Maître de Conférences HDR, Sorbonne Université, LIP6
Laurent-Stéphane DIDIER	Professeur, Université de Toulon, IMATH

Composition du Jury :

Président :	Emmanuel CASSEAU	Professeur, Université Rennes 1 - ENSSAT, IRISA
Examinateurs :	Roselyne CHOTIN-AVOT	Maître de Conférences HDR, Sorbonne Université, LIP6
	Laurent-Stéphane DIDIER	Professeur, Université de Toulon, IMATH
	William MARNANE	Senior Lecturer, University College Cork
	Florent BERNARD	Maître de Conférences, Université Jean Monnet, Lab. Hubert Curien
Dir. de thèse :	Arnaud TISSERAND	Directeur de Recherche CNRS, Lab-STICC

Table des matières

Table des figures

Liste des tableaux

Liste des Algorithmes

1 Introduction et contexte de la thèse

Depuis quelques années, la sécurité numérique est devenue une préoccupation majeure dans de nombreux domaines allant du militaire au bancaire en passant par la protection de la vie privée ou la protection des entreprises. Ce besoin de sécurité s'est particulièrement intensifié avec l'augmentation du nombre d'applications pouvant être amenées à échanger des informations potentiellement sensibles : *e*-commerce, téléphonie mobile, *set-top box* TV, *body area networks* (BAN), communications, *cloud computing*, contrôle d'accès, etc. C'est encore plus vrai avec les dernières avancées dans le domaine de l'*Internet of Things* (IoT) visant à connecter via Internet une majorité d'objets physiques utilisés dans notre vie quotidienne (voitures, maisons, lunettes, dispositifs médicaux ou dispositifs de surveillances par exemple) ou professionnelle.

L'émergence et la multiplication de telles applications a motivé de nombreux états, organismes de recherche et entreprises à augmenter leur investissement dans les domaines de la cybersécurité et de la *cryptographie*.

La cryptographie correspond, entre autres, à l'étude des techniques de *chiffrement* permettant de transformer à l'aide d'une *clé de chiffrement* un message *en clair* en un message *chiffré*. Le message chiffré, ou *cryptogramme*, est construit de façon à être incompréhensible par toute personne ne possédant pas la *clé de déchiffrement*. Le pendant de la cryptographie est la *cryptanalyse*, qui correspond à l'étude des techniques de *décryptage* permettant de deviner le message en clair à partir d'un cryptogramme sans posséder la clé de déchiffrement. La cryptographie et la cryptanalyse forment avec la stéganographie le domaine de la *cryptologie*, dont l'origine étymologique désigne la « science du secret ».

La cryptographie permet d'assurer la *confidentialité*, l'*intégrité*, l'*authenticité* et la *non-répudiation* lors de l'échange de messages. Ces différentes propriétés sont définies par l'organisation internationale de standardisation (ISO) dans la norme ISO/IEC 27000:2018(fr) [1] :

- confidentialité : « propriété selon laquelle l'information n'est pas diffusée ni divulguée à des personnes, des entités ou des processus non autorisés » ;
- intégrité : « propriété d'exactitude et de complétude » ;
- authenticité : « propriété selon laquelle une entité est ce qu'elle revendique être » ;
- non-répudiation : « capacité à prouver l'occurrence d'un événement ou d'une action donnée(e) et des entités qui en sont à l'origine ».

Différents domaines d'étude sont regroupés sous le terme cryptographie, parmi lesquels les plus connus sont la *cryptographie symétrique* ou *cryptographie à clé secrète* et la *cryptographie asymétrique* ou *cryptographie à clé publique*. On notera que la cryptographie englobe aussi l'étude des *fonctions de hachage* utilisées dans certains protocoles cryptographiques mais que nous ne discuterons pas ici.

La cryptographie à clé secrète, ou cryptographie symétrique, permet d'assurer la *confidentialité* lors de l'échange de messages. Son nom vient du fait qu'une *unique clé* est utilisée à la fois pour le chiffrement et pour le déchiffrement. Cette clé unique de chiffrement et de déchiffrement est souvent désignée sous

1. Voir le lien `https://www.iso.org/obp/ui/#iso:std:iso-iec:27000:ed-5:v1:fr` (consulté en août 2018).

le terme de *clé secrète* car elle ne doit être connue que par l'émetteur du cryptogramme et par son destinataire. Les premières utilisations connues de la cryptographie symétrique sont datées d'environ 2000 ans avant J.-C. en Égypte antique. Elle a depuis été couramment utilisée, en particulier à partir du début du 20^{e} siècle durant les deux guerres mondiales et la guerre froide, pour garantir la *confidentialité* des communications.

Parmi les algorithmes de chiffrement symétrique récents les plus connus, on peut citer le DES (pour *data encryption standard*) standardisé en janvier 1977 et déclaré obsolète en octobre 1999, l'algorithme Blowfish conçu en 1993 et placé dans le domaine publique par son créateur, ou encore l'AES (pour *advanced encryption standard*) standardisé par l'institut de normalisation américain NIST en 2001. Ce dernier est encore à jour le standard pour l'implantation logicielle ou matérielle de cryptographie symétrique.

Les algorithmes de cryptographie symétrique comme Blowfish et AES permettent le chiffrement et le déchiffrement rapide et à faible coût. Ils ont toutefois pour inconvénient majeur la nécessité de *partager* la clé secrète de façon *absolument sûre*. En effet, si une tierce entité non autorisée a accès à une clé secrète durant son transfert, elle sera capable de déchiffrer les cryptogrammes échangés et la confidentialité des échanges ne pourra alors plus être garantie. Historiquement, de nombreuses méthodes d'échange de clés secrètes ont été mises en place, reposant sur l'utilisation de canaux sécurisés de transmission : échanges physiques via mallettes blindées sécurisées, échanges téléphoniques via lignes protégées ou encore transmission par tiers de confiance par exemple. Aucune de ces méthodes n'est cependant à la fois absolument fiable, les canaux de communication utilisés pouvant être forcés, écoutés ou corrompus, ni efficaces en pratique.

L'une des évolutions majeures de la cryptographie a eu lieu en 1976 lorsque Diffie et Hellman ont proposé dans [DH76] une méthode pour l'*échange de clé* basée sur l'utilisation de *deux clés différentes* au lieu d'une unique clé comme c'est le cas dans les cryptosystèmes symétriques. La première clé d'une paire est la *clé privée* $k_{\texttt{prv}}$, connue par une unique entité, et la deuxième est la *clé publique* $k_{\texttt{pub}}$, dérivée de la clé privée $k_{\texttt{prv}}$ et diffusée publiquement. L'article [DH76] de Diffie et Hellman est le premier à avoir introduit le concept révolutionnaire de cryptographie à clé publique, ou cryptographie asymétrique. On considère toutefois que ce concept été découvert simultanément par Merkle bien que les travaux de ce dernier n'aient été publiés dans [Mer78] qu'en 1978.

Le protocole d'échange de clé de Diffie-Hellman permet de répondre au problème du partage de clé secrète pour le chiffrement symétrique de messages entre deux entités. Pour échanger une clé secrète, Alice et Bob commencent par choisir ensemble une valeur G puis par tirer chacun de leur côté un nombre aléatoire : $k^A_{\texttt{prv}}$ pour Alice et $k^B_{\texttt{prv}}$ pour Bob. Ces 2 nombres $k^A_{\texttt{prv}}$ et $k^B_{\texttt{prv}}$ correspondent aux clés secrètes respectives d'Alice et de Bob. Ainsi, seule Alice connaît la valeur de $k^A_{\texttt{prv}}$ et Bob celle de $k^B_{\texttt{prv}}$. Alice et Bob utilisent une fonction f spécifique pour calculer respectivement $k^A_{\texttt{pub}} = f(G, k^A_{\texttt{prv}})$ et $k^B_{\texttt{pub}} = f(G, k^B_{\texttt{prv}})$, correspondant à leurs clés publiques respectives. Ils échangent ensuite leurs clés publiques, puis Alice calcule $K_s = f(k^B_{\texttt{pub}}, k^A_{\texttt{prv}})$ et Bob calcule $K_s = f(k^A_{\texttt{pub}}, k^B_{\texttt{prv}})$. La valeur K_s obtenu respectivement par Alice et par Bob est identique et correspond à la clé secrète partagée.

La fonction f utilisée doit être *associative* et *commutative*, c'est-à-dire qu'elle doit vérifier

$$f\left(f(G, k^A_{\texttt{prv}}), k^B_{\texttt{prv}}\right) = f\left(f(G, k^B_{\texttt{prv}}), k^A_{\texttt{prv}}\right).$$

Il doit aussi s'agir d'une *fonction à sens unique à trappe* (*trap-door one-way function* en anglais), c'est à dire une fonction pour laquelle il est impossible de calculer $k^A_{\texttt{prv}}$ (resp. $k^B_{\texttt{prv}}$) à partir de G et de $k^A_{\texttt{pub}}$ (resp. $k^B_{\texttt{pub}}$) à un coût raisonnable mais grâce à laquelle on peut calculer K_s à partir de $k^A_{\texttt{prv}}$ et de $k^B_{\texttt{pub}}$

ou de $k^B_{\texttt{prv}}$ et de $k^A_{\texttt{pub}}$. L'utilisation d'une telle fonction à sens unique à trappe permet de garantir qu'une tierce personne ne pourra pas calculer K_s à partir de $k^A_{\texttt{pub}}$ et de $k^B_{\texttt{pub}}$ sans connaître au moins l'une des 2 clés secrètes. Ainsi, Alice et Bob peuvent générer chacun de leur côté un couple ($k_{\texttt{prv}}$, $k_{\texttt{pub}}$) et *diffuser publiquement* leurs clés publiques $k_{\texttt{pub}}$ respectives.

Dans [DH76], Diffie et Hellman ont aussi montré qu'il était possible d'utiliser le concept de la cryptographie asymétrique au sein de protocoles de *signature numérique* permettant d'assurer l'*authenticité*, l'*intégrité* et la *non-répudiation* lors du transfert de messages. Cependant, ni Diffie ni Hellman dans [DH76], ni Merkle dans [Mer78] n'ont proposé de cryptosystème asymétrique complet.

La première description d'un cryptosystème asymétrique complet a été proposée par Rivest, Shamir et Adleman en 1977 et 1978 dans [RSA77] et [RSA78]. Dans ce cryptosystème, nommé RSA d'après les initiales de ses créateurs, la fonction à sens unique à trappe est construite à partir du problème mathématique difficile de la *factorisation des grands entiers*. RSA a longtemps été le cryptosystème recommandé par les instances de standardisation pour la mise en place de protocoles d'échanges de clés, de protocoles de signatures numériques et de quelques autres protocoles spécifiques. Toutefois, les récents travaux en cryptanalyse et l'augmentation de la puissance disponible dans nos supports de calcul ont entrainé l'augmentation des tailles des paramètres et des clés dans RSA. En particulier, on notera l'existence de l'algorithme du crible algébrique (*number field sieve* en anglais) proposé par Pollard en 1988 et permettant la factorisation de grands entiers avec une complexité sous exponentielle (cf. [LLMP93]).

Par exemple, les implantations contemporaines de RSA nécessitent des clés d'au moins 2048 bits pour garantir les niveaux de sécurité théorique [2] recommandés de nos jours (soit environ 128 bits de sécurité). La gestion d'un chemin de données aussi large dans les implantations de RSA, que ce soit en logiciel ou en matériel, devient alors un véritable challenge pour les concepteurs cherchant à garantir de bonnes performances tout en limitant l'utilisation de ressources.

Près de 10 ans après la proposition de RSA, Miller et Koblitz ont indépendamment proposé en 1985 dans [Mil85] et en 1987 dans [Kob87] l'utilisation des *courbes elliptiques* pour définir une fonction à sens unique à trappe utilisable dans des cryptosystèmes asymétriques, donnant ainsi naissance à la *cryptographie sur courbe elliptique* (ECC). La fonction à sens unique à trappe utilisée est basée sur l'opération de *multiplication scalaire* d'un point d'une courbe elliptique par un scalaire k.

On notera que ECC n'est pas le premier cas d'utilisation des courbes elliptiques en cryptologie. En effet, il est amusant de constater que ces dernières ont initialement été utilisées en cryptanalyse comme support pour l'accélération d'algorithmes de factorisation de grands entiers visant à casser RSA (voir [Len87]).

La robustesse des cryptosystèmes ECC repose sur le problème mathématique difficile du *logarithme discret sur les courbes elliptiques* (ECDLP). Il n'existe en effet à ce jour aucun algorithme permettant le calcul du logarithme discret sur courbe elliptique avec un complexité sous exponentielle. Pour cette raison, les tailles de clés et de paramètres dans ECC sont très inférieures à celles nécessaires dans RSA. Par exemple, l'utilisation de clés de 256 bits dans ECC permet d'atteindre le même niveau de sécurité que l'utilisation de clés d'au moins 2048 bits dans RSA. ECC permet donc des implantations logicielles et matérielles plus efficaces que RSA quant à la consommation d'énergie et aux performances de calcul pour un niveau de sécurité théorique équivalent.

ECC a été standardisée par le NIST en janvier 2000 dans le standard FIPS 186-2 pour les signatures numériques (DSS) dont la dernière révision est disponible dans [KG13]. En mars 2006, le NIST a aussi

2. Le niveau de sécurité théorique de x bits d'un cryptosystème indique qu'attaquer ce dernier avec le meilleur algorithme connu revient, du point de vue de la complexité, à tester une à une les $2^x - 1$ clés de x bits possibles.

proposé dans SP 800-56A un ensemble de cryptosystèmes ECC pour l'échange de clé. La dernière version de ce document, datant du mois d'avril 2018, est disponible dans [BCR+18].

En 2005, l'agence de sécurité nationale américaine (NSA) a annoncé la publication de la « Suite B », regroupant un ensemble d'algorithmes recommandés pour la mise en place de protocoles cryptographiques. Dans ce document, la NSA recommande *l'utilisation exclusive de ECC* en remplacement de RSA pour les protocoles de signature numérique et d'échange de clé.

On notera enfin que ECC est aussi utilisée depuis 2009 dans les passeports biométriques européens, toujours pour la mise en place de protocoles d'échange de clé ou de signature numérique (cf. [Lac13]).

En 1988, Koblitz a étendu dans [Kob88] l'utilisation de ECDLP au cas des courbes hyperelliptiques, donnant ainsi naissance à la *cryptographie sur courbe hyperelliptique* (HECC). Les fonctions à trappe dans HECC sont basées sur l'opération de multiplication scalaire, comme pour ECC, mais calculée sur les points d'une courbe hyperelliptique.

Les courbes hyperelliptiques peuvent être vues comme une généralisation des courbes elliptiques pour lesquelles le calcul de la multiplication scalaire nécessite l'utilisation d'une représentation particulière des points. Cette représentation rend le calcul de la multiplication scalaire plus compliqué pour HECC que pour ECC. Elle permet cependant en contrepartie de manipuler des paramètres de courbe et des coordonnées de points de tailles réduites de moitié dans HECC. Par exemple, pour atteindre 128 bits de sécurité, un cryptosystème ECC utilisera des clés et des paramètres de 256 bits. Pour le même niveau de sécurité, un cryptosystème HECC utilisera quant à lui des clés de 256 bits mais des paramètres de seulement 128 bits.

L'algorithme de multiplication scalaire utilisé par Koblitz dans [Kob88] est basé sur les formules publiées par Cantor en 1987 dans [Can87]. En raison de la complexité de ces dernières, il a été estimé dans un premier temps que HECC n'offrait aucun gain de performances comparé à ECC, et ce malgré la réduction de la taille des paramètres et des coordonnées manipulés (cf. [SSI98, Sma99]). Il a fallu attendre le début des années 2000, avec en particulier la publication de nouvelles formules par Lange dans [Lan02] et [Lan05], pour que HECC soit reconnue comme une alternative viable à ECC. Par exemple, les performances des implantations logicielles de HECC publiées par Avanzi en 2004 dans [Ava04] s'approchent de celles obtenues pour ECC : environ 15% d'écart en temps de calcul pour les mêmes tailles de clés.

Depuis cette période, HECC a connu de nombreuses évolutions et améliorations, avec notamment les travaux de Gaudry publiés dans [Gau07] en 2007 et ceux de Bos et coll. publiés dans [BCHL16] en 2016. Ces travaux ont servi de base pour la mise en place d'implantations de cryptosystèmes HECC performants rivalisant avec les meilleures implantations ECC de l'état de l'art. En particulier, l'implantation logicielle μKummer de Renes et coll. proposée en 2016 dans [RSSB16] et utilisant les travaux de Gaudry sur les *surfaces de Kummer* permet des gains de performances en temps de 75% pour le protocole de signature numérique implanté sur un microcontrôleur ARM Cortex M0 par rapport à la meilleure solution ECC sur le même microcontrôleur. μKummer a aussi permis à Renes et coll. de réduire de 32% le temps de calcul pour un échange de clé de type Diffie-Hellman sur le petit microcontrôleur AVR ATmega comparé à la meilleure solution ECC implantée sur ce microcontrôleur.

Les constantes évolutions de HECC rendent difficile sa standardisation. De ce fait, il faudra peut-être attendre quelques années supplémentaires avant que les choix de courbes hyperelliptiques et de paramètres soient suffisamment stables pour être utilisés dans des standards tels que celui du NIST.

Il est cependant important de remarquer que la nécessité d'investir des efforts supplémentaires dans le développement de HECC a été récemment mise en question suite aux dernières avancées technologiques

rendant possible la future conception de l'ordinateur quantique. L'apparition de l'ordinateur quantique permettrait en effet de calculer le logarithme discret sur courbe (hyper)elliptique avec une *complexité polynômiale*. Il serait alors envisageable d'inverser « facilement » la fonction à sens unique à trappe utilisée dans ECC et HECC, à savoir l'opération de multiplication scalaire, ce qui remettrait en cause leur robustesse face à la cryptanalyse.

En avril 2016, le NIST a publié le rapport [NIS16] conseillant aux organismes de recherche et entreprises impliqués dans la cryptographie de consacrer leurs efforts à la mise en place de cryptosystèmes asymétriques basés sur la *cryptographie post-quantique* (PQC), c'est-à-dire résistants à l'ordinateur quantique. Ce rapport estime ainsi qu'il est moins important d'améliorer la sécurité des cryptosystèmes asymétriques actuels tels que ceux basés sur les courbes elliptiques ou hyperelliptiques : « *transitioning from 112 to 128 (or higher) bits of security is perhaps less urgent than transitioning from existing cryptosystems to post-quantum cryptosystems.* » ([NIS16] page 6).

Malgré les inquiétudes du NIST quant à l'arrivée de l'ordinateur quantique, il est cependant considéré que la nécessité d'investir dans la recherche sur ECC et HECC est plus que jamais d'actualité. Le premier argument justifiant ce point de vue est que la mise en place de nouveaux standards pour la PQC demandera un temps non négligeable durant lequel les cryptosystèmes à base de courbe seront encore largement utilisés, comme l'exposent Koblitz et Menezes dans [KM15] : « *If practical quantum computers are at least 15 years away, and possibly much longer, and if it will take many years to develop and test the proposed PQC systems and reach a consensus on standards, then a long time remains when people will be relying on ECC* ». L'autre argument en faveur de la cryptographie sur courbes est que de nombreuses années seront nécessaires avant que l'ordinateur quantique ne soit capable de casser des instances de ECC et de HECC à un coût raisonnable. En effet, d'après les *estimations* du NIST [NIS16], les premiers ordinateurs quantiques pourraient être accessibles à l'horizon 2030 pour la modique somme d'environ un milliard de dollars, le coût de l'infrastructure nécessaire pour casser une unique clé ECC de 256 bits étant quand à lui estimé à quelques millions de dollars au bas mot.

Pour cette raison, si l'avènement de l'ordinateur quantique représente une réelle menace pour la protection des données au niveau militaire ou au niveau de certaines instances particulières (p. ex. gouvernementales ou bancaires), il reste en pratique peu probable qu'il soit utilisé dans un futur proche pour attaquer des applications de la vie courante. La conception et l'implantation de cryptosystèmes ECC et HECC performants reste donc encore à ce jour un axe de recherche essentiel pour la protection de nombreuses applications.

Contexte et objectifs de la thèse : le projet HAH

C'est dans cette optique qu'à été initié en 2014 le projet [HAH] (pour *Hardware and Arithmetic for HECC*) qui finançait la thèse présentée dans ce document. Le but de ce projet était de proposer de nouvelles implantations d'*accélérateurs matériels performants*, *flexibles* et *robustes* pour HECC, basés sur l'étude de nouveaux *algorithmes et architectures efficaces* pour le calcul des *opérations arithmétiques* intervenant dans le *calcul de la multiplication scalaire*.

Démarré à l'automne 2014 pour une durée de trois ans, ce projet a réuni des équipes de recherche en électronique et informatique (laboratoires IRISA UMR 6074 et Lab-STICC UMR 6285) et en mathématiques (laboratoire IRMAR UMR 6625).

En pratique, le calcul de la multiplication scalaire n'est effectué que ponctuellement dans la majorité des applications faisant appel à des cryptosystèmes asymétriques. Par exemple, l'échange de clé Diffie-

Hellman nécessitant 2 multiplications scalaire n'est en général effectué que rarement, pour la mise en place de communications sécurisées par chiffrement symétrique. De façon similaire, la multiplication scalaire ne représente qu'une petite partie des calculs à effectuer lors de l'établissement et la vérification de signatures numériques. De ce fait, l'implantation d'accélérateurs matériels pour le calcul de la multiplication scalaire nécessitant une grande surface de circuit semble déraisonnable [3], en particulier pour des applications embarquées dans lesquelles la surface du circuit et la consommation d'énergie sont contraintes. Pour évaluer les performances d'un cryptoprocesseur, il est dès lors important de prendre en compte non seulement ses performances en matière de temps de calcul mais aussi la *surface de circuit* utilisée.

Durant la thèse, la capacité à pouvoir modifier les paramètres des courbes et du corps fini utilisés pour une taille d'éléments fixée a été une contrainte forte pour la conception de nos cryptoprocesseurs. En effet, HECC évolue encore beaucoup de nos jours et il est de ce fait difficile de prévoir quels cryptosystèmes seront utilisés, ou standardisés, dans les années à venir.

Afin d'augmenter la durée de vie des circuits proposés dans le cadre du projet, nous avons dû nous assurer que ces derniers puissent s'adapter aux évolutions de HECC. L'application de correctifs pour la modification des cryptosystèmes implantés sur FPGA est difficile à mettre en œuvre : un nouveau *bitstream* doit être généré et validé pour chaque cible FPGA. De plus, et pour des raisons de sécurité, l'intégrité et l'authenticité de chaque *bitstream* doit être vérifié au niveau du chaque FPGA avant configuration de ce dernier. Cela implique l'utilisation sur chaque FPGA de primitives de cryptographie asymétriques (ECC ou HECC par exemple) pour la vérification de signatures numériques, ce qui pose un problème de poules et d'œufs. Nous avons alors choisi de proposer des cryptoprocesseurs pour lesquels les paramètres des courbes et des corps finis peuvent être *modifiés à l'exécution*.

L'implantation d'accélérateurs flexibles empêche en contrepartie l'utilisation d'algorithmes optimisés pour certains premiers P, permettant d'accélérer certains calculs au niveau du corps. De tels algorithmes sont par exemple utilisés dans la récente implantation FPGA de μKummer publiée par Koppermann et coll. dans [KSHS18] en 2018. Ils permettent à ces derniers de proposer une implantation de cryptoprocesseur très rapide, mais *limitée* à l'utilisation d'*une unique courbe hyperelliptique*. Ce n'est pas le cas de nos accélérateurs qui pourront être utilisés pour différents cryptosystèmes HECC proposés dans le futur, à condition bien sûr que les tailles des premiers P et des scalaires soient inférieures ou égales aux tailles supportées dans nos cryptoprocesseurs.

Les cryptosystèmes implantés se doivent aussi d'être robustes face à de nombreux types d'attaques visant à retrouver la valeur des clés privées utilisées dans les protocoles cryptographiques. Dans le contexte de la thèse, nous avons évoqué les *attaques théoriques* visant principalement à résoudre ponctuellement le problème ECDLP. Les progrès récents des attaques théoriques sur ECDLP sont détaillés dans l'étude effectuée par Galbraith et Gaudry et publiée en 2016 dans [GG16]. Il existe cependant d'autres types d'attaques visant à récupérer les clés privées à partir de faiblesses et de failles apparaissant au niveau des *implantations* de cryptosystèmes théoriquement sûrs.

Les *attaques logiques* se basent sur l'exploitation de *failles logicielles* dans les implantations de cryptosystèmes pour révéler des informations sensibles, comme la valeur de la clé privée par exemple. Un exemple d'attaque logique sur microcontrôleur contre le protocole d'échange de clé ECDH implanté dans la bibliothèque Libcrypt et utilisant la courbe elliptique Curve25519 a été publié en 2017 par Genkin et coll. dans [GVY17]. Cette attaque consiste à utiliser des valeurs particulières en entrée de l'algorithme de multiplication scalaire afin de déclencher l'exécution de codes dépendant directement de la valeur

3. cf. l'implantation ECC de [AR14] utilisant 289 *slices* DSP, soit plus d'un tiers des *slices* DSP du FPGA, par exemple.

des bits de clé privée. En utilisant les fuites d'informations au niveau des caches du microcontrôleur, les auteurs ont montré qu'il était possible de récupérer la valeur de la clé privée en exécutant en utilisant au plus 11 traces d'exécution. Nos implantations FPGA de HECC ne sont pas concernées par ce type d'attaque, utilisé principalement pour casser des cryptoprocesseurs implantés en logiciel sur processeurs généralistes ou microcontrôleurs.

Les *attaques physiques* nécessitent d'avoir accès au circuit et sont séparées en deux catégories distinctes : les attaques par *observation* et les attaques par *perturbation*.

Les attaques physiques par *observation* se basent sur l'analyse des fuites d'information pouvant avoir lieu au niveau de mesures de grandeurs physiques telles que la consommation d'énergie, le rayonnement électromagnétique ou le temps d'exécution. On parle alors d'*analyse de canaux auxiliaires* ou cachés (SCA pour *side-channel analysis* en anglais). Les premières attaques physiques par observation ont été publiées par Kocher en 1996 dans [Koc96] et par Kocher et coll. en 1999 dans [KJB99]. Ces publications ont marqué un tournant majeur en cryptographie, montrant qu'assurer la sécurité d'un cryptosystème au niveau théorique n'était pas suffisant et que les implantations de ces cryptosystèmes devaient elles aussi être protégées.

Les attaques physiques par *perturbation* consistent à *injecter des fautes* dans le circuit du cryptosystème implanté afin de créer des fuites pouvant révéler des informations secrètes. Par exemple, Boneh et coll. ont montré en 1997 dans [BDL97] qu'il était possible de casser des implantations de signatures numériques RSA et différents autres protocoles d'authentification en utilisant les *fautes matérielles* pouvant être injectées dans les circuits. Un autre exemple d'attaque par injection de fautes est l'attaque *safe-error* proposée par Yen et Joye dans [YJ00] en 2000. Cette attaque vise à retrouver les bits de clé secrète en analysant la validité d'un résultat après injection de fautes dans un circuit. Elle permet entre autres d'attaquer des cryptosystèmes dans lesquels des opérations inutiles sont insérées afin d'uniformiser le temps d'exécution et la consommation électrique, et ainsi d'empêcher les attaques par SPA.

Contributions de la thèse et organisation du manuscrit

Dans le cadre du projet HAH, la thèse présentée dans ce document a été consacrée à la conception à l'évaluation et au prototypage sur FPGA d'architectures d'accélérateurs matériels (ou *cryptoprocesseurs*) performants et robustes pour le calcul de la multiplication scalaire dans HECC.

Pour cela, nous nous sommes dans un premier temps intéressé à l'étude des divers cryptosystèmes HECC proposés dans l'état de l'art ainsi qu'aux implantations matérielles de ces cryptosystèmes sur FPGA. Suite à cette étude, nous avons décidé d'utiliser des courbes hyperelliptiques *premières*, recommandées pour des raisons de sécurité (cf. [Ber06] ou [PQ12]), pour nos implantations sur FPGA. Ces courbes sont définies sur des corps finis premiers, notés $\mathrm{GF}(P)$, correspondant à l'ensemble des nombres entiers modulo un grand nombre premier P, c'est-à-dire à l'ensemble des nombres entiers compris entre 0 et $P-1$ inclus.

Dans le chapitre 2, nous rappelons les notions principales sur les corps finis et les courbes nécessaires pour l'implantation de cryptosystèmes HECC. Nous discuterons aussi les divers paramètres et courbes hyperelliptiques sélectionnés pour nos implantations, provenant essentiellement des travaux de [RSSB16]. Nous illustrerons en fin de chapitre l'utilisation de HECC dans deux exemples de protocoles cryptographique : l'échange de clé de Diffie-Hellman sur courbe (hyper)elliptique (ECDH) et la signature numérique ECDSA.

L'utilisation de courbes hyperelliptiques sur $\mathrm{GF}(P)$ implique de manipuler des paramètres de courbe

et des coordonnées de points dans GF(P). En particulier, le calcul de la multiplication scalaire requière de calculer de nombreuses opérations arithmétiques *modulaires* : additions, soustractions, multiplications et inversions modulo P par exemple.

Dans un second temps, nous nous sommes donc intéressés à la conception d'algorithmes et d'unités matérielles performants pour le calcul des opérations arithmétiques nécessaires à HECC. En particulier, nous nous sommes consacrés à la mise en place d'unités arithmétiques performants pour la *multiplication modulaire* $(A \times B) \bmod P$, qui est l'opération modulaire la plus courante et coûteuse dans HECC.

Nos contributions en matière d'unités arithmétiques pour le calcul de la multiplication modulaire seront détaillées dans le chapitre 3. Nous étudierons et comparerons dans un premier temps les algorithmes classiques de l'état de l'art pour le calcul de la multiplication modulo des premiers P *génériques* (c.-à-d. sans structure binaire particulière). Nous en profiterons pour présenter les meilleures implantations matérielles de multiplieurs modulaires utilisant ces algorithmes.

Nous avons choisi d'implanter dans nos multiplieurs la variante itérative CIOS (*coarsely integrated operand scanning*) [KAK96] de l'algorithme classique de *multiplication modulaire de Montgomery* [Mon85]. Cette variante est l'une des plus connues et l'une des plus utilisées pour les implantations de multiplieurs modulaires sur FPGA. Elle provoque cependant de fortes dépendances de données entraînant l'apparition de « bulles » dans le pipeline interne des opérateurs, c'est-à-dire des cycles durant lesquels aucune opération utile n'est calculée dans certains étages du pipeline interne des opérateurs. Ces « bulles » diminuent l'efficacité des unités implantées. Pour réduire fortement leur nombre, nous avons utilisé des méthodes d'*hyper-threading* [KM03] consistant à partager *une unité physique* entre *plusieurs unités logiques*. Dans nos *multiplieurs modulaires hyperthreadés* (HTMM), plusieurs multiplications modulaires indépendantes sont ainsi calculées en même temps ce qui nous permet de recouvrir les délais internes causés par les dépendances de données. Les HTMM proposés dans la thèse sont optimisés pour utiliser « efficacement » les ressources des FPGA et pour atteindre des fréquences proches des fréquences maximales de ces derniers.

Pour illustrer le fonctionnement de nos unités, nous proposons deux versions de nos HTMM pour des tailles de premiers P de 128 et de 256 bits, utilisables dans des cryptoprocesseurs HECC pour des tailles de clés respectives de 256 ou 512 bits. Les versions 256 bits de nos HTMM peuvent aussi être intégrées dans des cryptoprocesseurs ECC utilisant des clés de 256 bits.

Nos premiers HTMM, implantés à la main pour des premiers de 128 bits figés, ont été proposés dans [GT17d] en 2017 à la conférence internationale *IEEE Asilomar Conference on Signals, Systems and Computers*. Les résultats d'implantations obtenus ont montré que nos HTMM étaient à la fois *plus petits* et *plus rapides* que les meilleurs multiplieurs de l'état de l'art implantés sur les mêmes FPGA pour des opérandes de 128 bits. Par exemple, le HTMM implanté sur un FPGA Spartan-6 de Xilinx utilise 1.9 fois moins de *slices* DSP et 3 fois moins de BRAM et de *slices* logiques pour un temps de calcul réduit de 15%.

Plus tard durant la thèse, nous avons complété ces résultats en apportant de nombreuses améliorations à nos HTMM : augmentation de la fréquence de fonctionnement, réduction du nombre de *slices* DSP, réduction de la latence de l'unité et possibilité de modifier le premier P à l'exécution par exemple. Ces résultats ont été obtenus grâce à l'aide d'un *générateur* de HTMM, implanté par nos soins et distribué en *open source*, permettant de générer les sources VHDL de différents HTMM à partir de différentes *spécifications de paramètres d'architecture*.

Avec l'aide de notre générateur, nous avons pu facilement *explorer* et implanter différentes spécifications de nouveaux HTMM 128 bits et 256 bits. Ces nouveaux HTMM ont été soumis dans [GT18a] au journal *IEEE Transactions on Computers* en juin 2018. Grâce à l'exploration de nombreuses spéci-

fications d'architecture, nous sommes en mesure de proposer des multiplieurs hyperthreadés optimisés présentant de meilleurs compromis temps – surface que les meilleurs multiplieurs de l'état de l'art. Par exemple, notre HTMM le plus rapide implanté sur le FPGA Virtex-4 utilise 2.4 fois moins de *slices* logiques, 4 fois moins de *slices* DSP et 5 fois moins de BRAM que le meilleur multiplieur 256 bits de l'état de l'art, pour un temps de calcul seulement 1.5 fois plus grand.

Pour faciliter les futures comparaisons des multiplieurs modulaires de la littérature avec nos multiplieurs modulaires hyperthreadés, l'ensemble des codes sources et des résultats d'implantation des HTMM proposés dans [GT18a] ainsi que l'ensemble des sources de notre générateur sont disponibles en *open-source* dans [GT18b].

Après avoir conçu nos premiers HTMM 128 bits de [GT17d], nous nous sommes concentré sur la mise en place d'architectures de cryptoprocesseurs pour le calcul de la multiplication scalaire dans HECC. La conception et la validation de ces architectures ainsi que leurs résultats d'implantation sont décris dans le chapitre 4.

Dans ce chapitre, nous étudierons tout d'abord les implantations matérielles de cryptoprocesseurs HECC dans l'état de l'art, dont la très grande majorité date du début des années 2000 et propose des niveaux de sécurité théorique trop faibles par rapport aux standards actuels (environ 80 à 90 bits de sécurité). Nous reviendrons alors sur l'implantation logicielle μKummer de [RSSB16] utilisant les dernières formules et algorithmes de l'état de l'art pour le calcul de la multiplication scalaire sur courbe hyperelliptique. Ces formules s'appuient sur l'utilisation d'une projection particulière des points d'une courbe hyperelliptique sur la *surface de Kummer*, qui permet de réduire le nombre d'opérations arithmétiques dans la multiplication scalaire et d'accélérer le calcul de cette dernière. En raisons des excellents résultats obtenus par μKummer, nous avons décidé de nous baser sur le cryptosystème de [RSSB16] pour nos implantations matérielles de HECC.

Nos cryptoprocesseurs pour Kummer-HECC (KHECC) sont construits sur un modèle d'architecture classique de Harvard. Afin de concevoir la meilleure architecture possible, c'est-à-dire celle permettant d'obtenir le meilleur compromis temps de calcul – surface de circuit, nous avons décidé d'évaluer l'impact de différents paramètres d'architecture sur les performances de nos implantations. Nous nous sommes toutefois rapidement rendu compte que la diversité de ces paramètres impliquait d'explorer et d'implanter de très nombreuses spécifications d'accélérateurs. En raison de la difficulté et du temps requis pour l'implantation, la validation et l'évaluation d'un cryptoprocesseur complet sur différents FPGA, cette exploration n'est pas réalisable manuellement en pratique.

Nous avons donc mis en place un *modèle haut niveau* basé sur le TLM (pour *transaction-level modeling*) nous permettant de *simuler* et de *valider* nos accélérateurs *au niveau architectural*. Dans ce modèle, que nous avons nommé CCABA pour *critical cycle accurate bit accurate*, seuls les cycles significatifs au niveau de l'architecture sont modélisés en CABA. Ces cycles correspondent par exemple aux transferts des opérandes et résultats vers et depuis les unités ou à la modification des signaux de contrôle. Les diverses unités ont quand à elle été complètement implantées et validées sur FPGA et leur comportement est donc parfaitement connu. Les calculs effectués au sein de nos unités sont donc *abstraits*, ce qui nous permet de simplifier notre modèle et d'accélérer la simulation et la validation de nos architectures de cryptoprocesseurs.

Nous avons développé un ensemble d'outils logiciels pour la validation et la simulation rapide de nos modèles d'architecture. Ces outils ont aussi été instrumentés pour nous permettre d'*estimer les performances* respectives de nos architectures. Ils nous ont permis d'explorer de nombreux cryptoprocesseurs

pour différentes spécifications de paramètres : type et nombre des unités arithmétiques, tailles des communications internes, topologie de l'architecture, etc.

Suite à cette exploration, nous avons proposé dans [GCT17], publié en 2017 à la conférence internationale *Indocrypt*, quatre architectures d'accélérateurs matériels pour le calcul de la multiplication scalaire dans KHECC pour des clés de 256 bits. À notre connaissance, nos architectures matérielles implantées sur FPGA sont les premières à utiliser les surfaces de Kummer de courbes hyperelliptiques pour calculer la multiplication scalaire dans HECC. Toutes les architectures proposées dans [GCT17] ont été complètement implantées, validées et évaluées sur 3 FPGA différents pour 3 tailles différentes des communications internes. Nous avons donc proposé un total de 36 implantations de cryptoprocesseurs, parmi lesquels nous avons pu sélectionner les meilleurs compromis temps de calcul – surface de circuit pour chaque FPGA. La comparaison de nos cryptoprocesseurs KHECC avec les meilleurs cryptoprocesseurs ECC pour des *courbes quelconques* et des clés de 256 bits montre que nos accélérateurs sont 40% plus petits en nombre *slices* DSP et 50% plus petits en nombre de *slices* logiques pour des temps de calcul similaires. Nos accélérateurs intègrent aussi certaines protections contre les attaques par observation et par analyse simple de consommation de puissance électrique dans le circuit (SPA).

Nous conclurons finalement ce document dans le chapitre 5 qui nous permettra aussi de donner quelques perspectives d'évolutions et d'améliorations du travail effectué durant la thèse.

Notations

$[k]\mathcal{P}$	*multiplication scalaire* du *point* $\mathcal{P}$ d'une courbe (hyper)elliptique par le *scalaire* $k \in \mathbb{N}$
$\pm\mathcal{P}$	*projection* d'un *point* $\mathcal{P}$ de la Jacobienne $\mathcal{J}_\mathcal{C}$ sur la surface de Kummer $\mathcal{K}_\mathcal{C}$ associée
$(x : y : z : t)$	*coordonnées* d'un point sur $\mathcal{K}_\mathcal{C}$
θ	nombre minimum de cycles entre deux MMM indépendantes *consécutives* calculées dans deux LM *différents*
λ	*latence* en cycles pour le calcul d'une MMM dans un multiplieur modulaire
σ	*nombre de multiplieurs logiques* (LM) dans un HTMM physique
τ	*intervalle* de temps en cycles entre deux MMM indépendantes calculées dans *un même* LM
`ADD`	opération d'addition de points sur la courbe elliptique $\mathcal{E}$ ou la Jacobienne $\mathcal{J}_\mathcal{C}$
`AddSub`	additionneur-soustracteur modulaire dans nos cryptoprocesseurs KHECC
BAN	réseau sans fil de capteurs localisés autour du corps humain (*body area network*)
BRAM	bloc RAM câblé des FPGA
$\mathcal{C}$	une courbe hyperelliptique
CABA	*modélisation* au *cycle près et au bit près* des signaux dans un circuit (*cycle accurate bit accurate*)
CAO	conception assistée par ordinateur
cc, CC	cycle d'horloge (*clock cycle*)
CCABA	notation que nous avons introduite pour désigner la modélisation CABA des signaux aux *cycles critiques* dans nos architectures (*critical cycle accurate bit accurate*)
CIOS	variante de la MMM présentée dans [KAK96] (*coarsely integrated operand scanning*)
CIHS	variante de la MMM présentée dans [KAK96] (*coarsely integrated hybrid scanning*)
`CSWAP`	opération de permutation de points (*conditional swapping operation*)
`DBL`	opération de doublement de point sur la courbe elliptique $\mathcal{E}$ ou la Jacobienne $\mathcal{J}_\mathcal{C}$
`DBLADD`	opération unifiée d'addition-doublement de points sur la courbe elliptique $\mathcal{E}$ ou la Jacobienne $\mathcal{J}_\mathcal{C}$
DPA	analyse différentielle de la puissance consommée dans un circuit (*differential power analysis*)
DRAM	mémoire de type RAM utilisant les ressources logiques des FPGA (*distributed* RAM), à ne pas confondre avec (*dynamic RAM*)
slice DSP	bloc matériel câblé des FPGA embarquant un multiplieur et un accumulateur (DSP signifie *digital signal processing*)
$\mathcal{E}$	une courbe elliptique
ECC	cryptographie sur courbe elliptique (*elliptic curve cryptography*)
FF	bascule synchrone (*flip-flop*)
FIFO	file d'attente (*first in, first out*)
FIOS	variante de la MMM présentée dans [KAK96] (*finely integrated operand scanning*)

FIPS	variante de la MMM présentée dans [Kal93] (*finely integrated product scanning*)
FPGA	circuit logique programmable (*field-programmable gate array*)
FSM	automate à nombre fini d'états (*finite state machine*)
GF	corps fini (*finite field*)
GF(P)	corps fini de *caractéristique première* P
GF(2^m)	*extension* du *corps binaire* GF(2) : $\{0, 1\}$ (m est un nombre premier)
HECC	cryptographie sur courbe hyperelliptique (*hyperelliptic curve cryptography*)
HTMM	multiplieur modulaire hyperthreadé (*hyper-threaded modular multiplier*)
IoT	internet des objets (*Internet of Things*)
ISA	architecture à jeu d'instructions (*instruction set architecture*)
$\mathcal{J}_\mathcal{C}$	la *Jacobienne* d'une *courbe hyperelliptique* $\mathcal{C}$
k	*scalaire* ou *clé privée* sur n_k bits dans nos implantations (H)ECC, $k \in \mathbb{N}$
$\mathcal{K}_\mathcal{C}$	la *surface de Kummer* associée à la *courbe hyperelliptique* $\mathcal{C}$
KHECC	*cryptosystème* HECC utilisant les *surfaces de Kummer* pour accélérer le calcul de la multiplication scalaire (*Kummer-based HECC*)
LM	*multiplieur logique* dans un HTMM physique (*logical multiplier*)
LSB	bit de *poids faible* (*least significant bit*)
LUT	table de correspondance (*lookup table*)
m	taille en bits des nombres modulo P *projetés* dans le domaine de Montgomery (MD)
n.r.	non renseigné
MMM	multiplication modulaire de Montgomery
MSB	bit de *poids fort* (*most significant bit*)
n	taille du *premier* P (en bits)
n_k	taille du *scalaire* k (en bits)
NAF	forme non adjacente d'un nombre (*non adjacent form*)
NIST	institut de standardisation américain (*national institute of standards and technology*)
pgcd	plus grand commun diviseur
PQC	cryptographie post-quantique (*post-quantum cryptography*)
P	un *nombre premier*
$\mathcal{P}$, $\mathcal{Q}$, $\mathcal{R}$, $\mathcal{V}$	*points* de la *courbe elliptique* $\mathcal{E}$ ou de la *Jacobienne* $\mathcal{J}_\mathcal{C}$
RAM	mémoire vive (*random-access memory*)
RSA	cryptosystème proposé par Rivest, Shamir et Adleman dans [RSA78]
s, w	*décomposition des éléments* de GF(P) sur m bits en s mots de w bits dans nos *unités arithmétiques*
$\tilde{s}$, $\tilde{w}$	*décomposition des éléments* de GF(P) sur m bits en $\tilde{s}$ mots de $\tilde{w}$ bits dans nos *cryptoprocesseurs* (mémoires et réseaux d'interconnexion)
SCA	analyse des canaux auxiliaires (*side channel analysis*)
SOS	variante de la MMM présentée dans [DK90] (*separated operand scanning*)
SPA	analyse simple de la puissance consommée dans un circuit (*simple power analysis*)
TLM	modélisation au niveau des transferts de données (*transaction-level modeling*)
VHDL	langage de description de matériel (VHSIC *hardware description language*)
VHSIC	circuits intégrés à très haute vitesse (*very high speed integrated circuits*)
`xADD`	opération `ADD` différentielle de points sur $\mathcal{K}_\mathcal{C}$ (*differential addition*)
`xDBLADD`	opération `DBLADD` différentielle de points sur $\mathcal{K}_\mathcal{C}$ (*differential double-and-add*)

2 Rappels sur les cryptosystèmes ECC et HECC

Sommaire

2.1 Introduction

Dans ce chapitre, nous présenterons quelques notions nécessaires pour comprendre le fonctionnement des cryptosystèmes à base de courbes elliptiques et hyperelliptiques. Nous ne détaillerons pas les mathématiques des cryptosystèmes ECC et HECC. Pour plus de détails, nous renvoyons le lecteur vers les livres [HMV04] pour ECC et [CFA+05] pour HECC.

L'utilisation des courbes hyperelliptiques pour la cryptographie asymétrique a été proposée par Koblitz en 1988 dans [Kob88] pour trouver de nouvelles *fonctions à sens unique à trappe* reposant sur le *problème du logarithme discret dans les courbes elliptiques* (ECDLP). Le problème du logarithme discret (DLP) dans un *groupe fini* G d'éléments muni d'une *opération de multiplication* consiste à retrouver la valeur de l'entier m utilisé pour calculer $a = b^m$ en connaissant les valeurs $a, b \in G$. Quand le groupe G permet au problème du logarithme discret d'être un problème difficile, il est possible de s'en servir pour construire une cryptosystème asymétrique dans lequel le calcul de b^m pour des grandes valeurs de m est la fonction à sens unique à trappe.

En 1985 et 1987, Miller et Koblitz ont respectivement proposé de façon indépendante dans [Mil85] et [Kob87] d'utiliser comme groupe G l'ensemble des points de courbes elliptiques définies sur des corps finis. La fonction à sens unique à trappe utilisée dans les cryptosystèmes ECC de Koblitz et Miller est la fonction de *multiplication scalaire* $[k]\mathcal{P}$ d'un point $\mathcal{P}$ d'une courbe elliptique par un nombre entier positif k. Cette même fonction de multiplication scalaire est utilisée par Koblitz dans [Kob88] pour la construction de cryptosystèmes HECC utilisant des courbes hyperelliptiques.

Les cryptosystèmes ECC et HECC sont construits à partir d'une hiérarchie d'opérations définies sur des courbes elliptiques ou hyperelliptiques et des corps finis premiers ou des extensions du corps fini binaire. Au niveau le plus haut, les protocoles cryptographiques utilisent l'opération de *multiplication scalaire* sur les points de courbes (hyper)elliptiques comme *fonction à sens unique à trappe*. L'opération de multiplication scalaire est elle construite sur un ensemble d'opérations sur les points de courbes elliptiques et hyperelliptiques, par exemple les opérations d'addition de points `ADD` et de doublement de point `DBL`. Enfin, au niveau le plus bas, les opérations au niveau courbe sont elles-mêmes construites grâce à un ensemble d'opérations sur les coordonnées des points des courbes et sur leurs paramètres, définies dans des corps finis premiers $\mathrm{GF}(P)$ pour les courbes dites *premières* ou sur des extensions $\mathrm{GF}(2^m)$ du corps fini binaire pour les courbes dites *binaires*.

Le chapitre courant est organisé suivant cette hiérarchie. Nous partirons du niveau le plus bas en rappelant dans la section 2.2 les opérations arithmétiques dans les corps finis $\mathrm{GF}(P)$ et $\mathrm{GF}(2^m)$ utilisées dans ECC et HECC. Dans la section 2.3 suivante, nous nous intéresserons aux courbes elliptiques et à la définition des opérations dans le groupe des points de ces courbes : addition de points `ADD` et doublement de point `DBL`. Nous listerons entre autres dans cette section différents algorithmes et méthodes pour le calcul de la multiplication scalaire. Nous étudierons les courbes hyperelliptiques dans la section 2.4 et décrirons en particulier les *surfaces de Kummer* utilisées dans les implantations récentes les plus performantes de HECC de l'état de l'art. Finalement, nous donnerons en section 2.5 deux exemples de protocoles cryptographiques reposant sur l'opération de multiplication scalaire sur les courbes elliptiques ou hyperelliptiques.

2.2 Arithmétique dans les corps finis

Deux types de corps finis sont classiquement utilisés pour les implantations matérielles et logicielles de ECC et HECC : les *corps finis premiers* et les *extensions du corps fini binaire* GF(2), parfois appelées simplement *corps binaires* par abus de langage.

Arithmétique dans les corps premiers

Un *corps fini premier*, noté GF(P) ou $\mathbb{F}_P$, est composé d'un *ensemble fini* de nombres entiers *modulo* P pour lequel P est un nombre premier, aussi appelé *caractéristique du corps*. Les éléments de GF(P) sont les entiers compris entre 0 et $P-1$. Les opérations sur les éléments du corps sont calculées modulo le premier P. On parle alors d'*opérations modulaires*. La *réduction* modulo P lors d'une opération modulaire dont les opérandes sont des entiers dans $[0, P-1]$ permet à son résultat de rester dans $[0, P-1]$. La réduction de l'entier a modulo P consiste à calculer le reste r de la division euclidienne de a par P :

$$a = \underbrace{q}_{\text{quotient}} \times P + \underbrace{r}_{\text{reste}}$$

Il s'agit d'une opération coûteuse, impliquant le calcul d'une division et d'une multiplication dans les entiers $\mathbb{Z}$:

$$r = a - \left\lfloor \frac{a}{P} \right\rfloor \times P.$$

Pour ECC et HECC, les opérations dans GF(P) sont en grande majorité des additions, des soustractions, des multiplications, des carrés et des inversions modulaires. L'implantation matérielle ou logicielle de ECC et de HECC nécessite alors l'utilisation d'algorithmes et d'unités arithmétiques efficaces pour le calcul de ces opérations. Dans de nombreuses implantations ECC et HECC de la littérature pour des courbes premières, les auteurs utilisent des premiers possédant des représentations avec une structure particulière. Cette structure particulière leur permet d'accélérer le calcul de la réduction modulaire. C'est par exemple le cas des premiers dits *de Mersenne* $P = 2^n - 1$ avec n un entier positif tel que P soit premier. [1] Si on considère l'entier $a = \sum_{j=0}^{m-1} a_j \, 2^{j\,n}$, $\forall a_j$ entier dans $[0, 2^n - 1]$ et $m \geq 1$, on a $2^{jn} \bmod (2^n - 1) = 1$ et $a \bmod (2^n - 1)$ peut donc s'écrire

$$a \bmod (2^n - 1) = \sum_{j=0}^{m-1} a_j \bmod (2^n - 1).$$

La réduction modulo un premier de Mersenne peut donc être calculée grâce à des additions et décalages successifs. L'utilisation de ce type de premiers particuliers permet la mise en place d'unités arithmétiques rapides pour le calcul des opérations modulaires au sein des accélérateurs matériels. Ces derniers sont toutefois limités à l'utilisation d'un unique premier P pour une taille d'opérandes fixée, ce qui n'est pas souhaitable pour la conception de circuits flexibles.

Nos cryptoprocesseurs HECC flexibles sont construits autour d'unités arithmétiques utilisant des premiers P ne possédant pas de représentation avec une structure particulière, souvent appelés *premiers quelconques* ou *génériques*. Le calcul de la réduction dans ces unités ne dépend pas de la valeur de P, qui peut donc être modifiée à l'exécution pour une taille de premier fixée, mais nécessite souvent en contrepartie l'utilisation d'algorithmes plus complexes que pour la réduction modulo des premiers spécifiques.

1. Seul un sous-ensemble des valeurs de n conduit à des valeurs de $2^n - 1$ premières.

C'est en particulier le cas de la multiplication modulaire.

La multiplication modulaire et le calcul des carrés modulaires sont les opérations de GF(P) les plus courantes et coûteuses dans (H)ECC. Il existe dans la littérature différents algorithmes dédiés au calcul de la multiplication modulo des premiers génériques, comme les algorithmes proposés dans [Bla83], dans [Bar84] ou dans [Mon85] par exemple. Ils ont en général pour but de remplacer le calcul coûteux de la division euclidienne $\lfloor a/P \rfloor$ dans la réduction modulaire de l'entier a par P par des opérations plus simples à calculer. Par exemple, l'algorithme proposé par Montgomery dans [Mon85] en 1985 permet de remplacer cette division par deux multiplications et quelques additions et décalages logiques. Cet algorithme est l'un des plus répandus dans la littérature pour le calcul de multiplications modulo des premiers génériques. De nombreuses variantes de cet algorithme ont été proposées dans l'état de l'art, sur lesquelles nous reviendrons dans le chapitre 3 du manuscrit consacré à nos unités HTMM pour le calcul des multiplications et carrés modulaires.

L'inversion modulo P, c'est à dire le calcul de $a^{-1} \bmod P$ pour un entier a dans $[0, P-1]$, est certainement l'opération arithmétique la plus coûteuse dans les cryptosystèmes ECC et HECC. Par chance, elle est aussi beaucoup moins fréquente que l'opération de multiplication modulaire. Deux méthodes sont classiquement utilisées dans la littérature pour calculer l'opération d'inversion modulaire : le *petit théorème de Fermat* et *l'algorithme d'Euclide étendu*.

D'après le petit théorème de Fermat, pour P un nombre premier et a un entier non multiple de P, on a l'égalité suivante :

$$\left(a^{P-1} - 1\right) \bmod P = 0.$$

On a alors $\left(a \times a^{P-2}\right) \bmod P = 1$ et donc $a^{-1} \bmod P = a^{P-2} \bmod P$. Le calcul de l'inversion modulaire peut donc être calculée par une exponentiation modulaire correspondant à une grande suite de multiplications et de carrés dans GF(P).

L'algorithme d'Euclide étendu, utilisé à l'origine pour le calcul du *plus grand diviseur commun* (pgcd), permet de trouver les éléments d_1 et d_2 tels que, pour deux entiers a et b, on ait :

$$\text{pgcd}(a, b) = d_1 a + d_2 b.$$

Si a et b sont premiers entre eux, alors $\text{pgcd}(a, b) = 1$. Pour l'entier a différent d'un multiple de P et $b = P$, on a donc $d_1 a + d_2 P = 1$ et donc $(d_1 \times a) \bmod P = 1$. On a donc bien calculé $d_1 = a^{-1} \bmod P$. L'algorithme complet est décrit au chapitre 4 de [Knu98].

Nous verrons dans la suite du chapitre qu'il est possible de se passer en très grande partie du calcul des inversions modulaires dans le calcul de la multiplication scalaire grâce à l'utilisation de systèmes de coordonnées particuliers pour les points de la courbe. Pour cette raison, nous n'avons pas implanté d'unité arithmétique pour l'inversion ou la division modulaire dans nos cryptoprocesseurs HECC.

Finalement, les additions modulaires peuvent être calculées grâce à quelques additions et soustractions d'entiers. Pour l'addition de deux entiers positifs a et b, on calcule la somme $a + b$ et on soustrait P au résultat tant que celui-ci est plus grand ou égal à P. Pour deux opérandes a et b déjà dans $[0, P-1]$, le résultat de $a + b$ est dans $[0, 2P-2]$. Au plus une soustraction de P sera donc nécessaire pour la réduction du résultat vers $[0, P-1]$.

De façon similaire, la soustraction de deux entiers positifs a et b nécessite de calculer la différence $a - b$ et d'ajouter P au résultat tant que celui-ci est inférieur à 0. Pour deux opérandes a et b déjà dans

$[0, P-1]$, le résultat de $a-b$ est dans $[-P+1, P-1]$. Au plus une addition de P sera donc nécessaire pour la réduction du résultat vers $[0, P-1]$.

Arithmétique dans les extensions du corps binaire

Les corps finis de type $\mathrm{GF}(2^m)$, parfois dits *binaires*, correspondent à des *extensions* du corps fini $\mathrm{GF}(2)$.[2] Les éléments de $\mathrm{GF}(2^m)$ peuvent être représentés sous forme de *polynômes binaires* de degré inférieur ou égal à $m-1$ et dont les coefficients sont des éléments du corps $\mathrm{GF}(2) : \{0, 1\}$. Les opérations sur les éléments de $\mathrm{GF}(2^m)$ sont calculées modulo un *polynôme binaire irréductible*, c'est à dire un polynôme ne pouvant pas s'écrire sous forme d'un produit de polynômes. Un polynôme irréductible dans $\mathrm{GF}(2^m)$ est l'équivalent d'un nombre premier dans $\mathrm{GF}(P)$.

L'addition d'éléments de $\mathrm{GF}(2^m)$ s'effectue de façon classique en additionnant un à un les coefficients du polynôme. Ces derniers étant définis dans $\mathrm{GF}(2)$, l'addition des polynômes revient à effectuer un simple `xor` bit à bit des coefficients. Il en est de même pour la soustraction dans $\mathrm{GF}(2^m)$, qui correspond alors à la même opération que l'addition. On notera qu'il n'y a pas de propagation de retenues entre les coefficients des polynômes lors des calculs des additions. L'addition d'éléments de $\mathrm{GF}(2^m)$ ne nécessite pas d'étape de réduction, contrairement aux opérations de multiplication et d'inversion qui sont calculées modulo le polynôme irréductible.

Nous ne détaillerons pas dans ce manuscrit le calcul des multiplications et des divisions dans $\mathrm{GF}(2^m)$. Nous renvoyons le lecteur vers, par exemple, la thèse de Métairie [Mét16] pour de plus amples informations et références sur l'utilisation et l'implantation de l'arithmétique dans $\mathrm{GF}(2^m)$ pour ECC.

2.3 Les courbes elliptiques

2.3.1 Définition

Une courbe elliptique $\mathcal{E}/\mathbb{K}$ est une courbe définie sur le corps $\mathbb{K}$ par une équation à deux variables $(x, y) \in (\mathbb{K}, \mathbb{K})$ de la forme suivante, dite *équation de Weierstrass* :

$$\mathcal{E}/\mathbb{K} : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6; \tag{2.1}$$

dans laquelle $a_1, a_2, a_3, a_4, a_6 \in \mathbb{K}$ sont les paramètres de la courbe $\mathcal{E}/\mathbb{K}$.

L'ensemble des points de la courbe correspond à l'ensemble des couples (x, y) vérifiant l'équation 2.1 auquel on rajoute un *point à l'infini*, noté $\mathcal{P}_\infty$. Ce point à l'infini est nécessaire pour la mise en place d'opérations sur les points de $\mathcal{E}/\mathbb{K}$ dont nous parlerons dans les sections 2.3.2 à 2.3.5. Il sert en particulier d'élément neutre pour certaines de ces opérations. Le point à l'infini peut être vu comme un point abstrait situé à l'intersection de la droite d'équation $y = \infty$ et de l'ensemble des droites verticales d'équations $x = c, \forall c \in \mathbb{K}$.

Dans l'équation 2.1, la courbe elliptique $\mathcal{E}/\mathbb{K}$ est *définie sur* le corps $\mathbb{K}$, ce qui signifie que les coefficients a_1, a_2, a_3, a_4 et a_6 ainsi que les coordonnées affines (x, y) des points de la courbe sont définis dans le corps $\mathbb{K}$. En figure 2.1, nous illustrons trois exemples de courbes elliptiques ayant la même équation $\mathcal{E}/\mathbb{K} : y^2 = x^3 - 3x + 1$ mais définies sur des corps $\mathbb{K}$ différents : la première courbe en fig. (a) est définie

2. NB : dans cette section, m désigne la taille des éléments de l'extension de corps. Il ne doit pas être confondu avec la notation m utilisée dans les chapitres suivants pour désigner la taille des opérandes de nos unités HTMM.

sur $\mathbb{K} = \mathbb{R}$, c.-à-d. sur le corps des réels ; la seconde en fig. (b) est définie sur le très petit corps fini premier GF(1223) ; et la troisième courbe donnée en fig. (c) est définie sur le petit corps fini premier GF(11489). Comme on peut le voir dans ces deux dernières figures, une courbe elliptique dans un corps fini ressemble à un nuage de points très dense. De tels exemples de courbes ne sont jamais utilisés en pratiques. Pour les applications cryptographiques, le corps $\mathbb{K}$ est fini et doit contenir un nombre d'éléments très important. Ainsi, pour des raisons de sécurité cryptographique, les courbes utilisées aujourd'hui sont définies sur des corps finis dont la taille des éléments est supérieure à 200 bits.

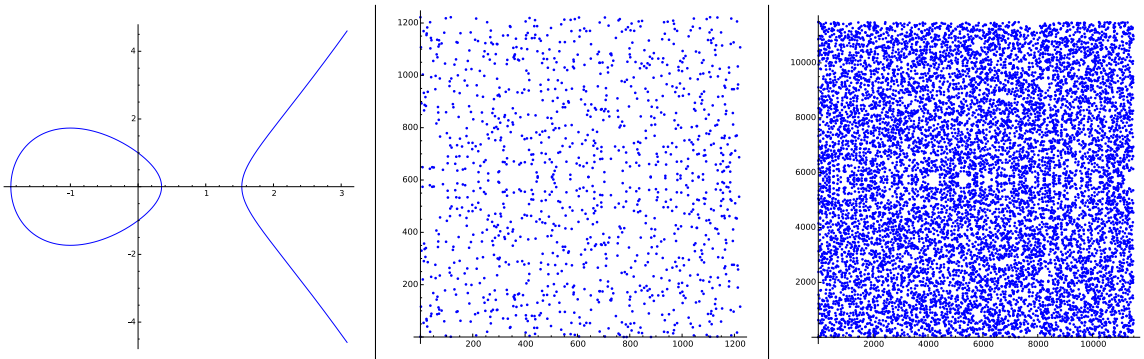


FIGURE 2.1 – Courbes elliptiques d'équation $y^2 = x^3 - 3x + 1$ à coefficients x et y dans les réels $\mathbb{R}$ (fig. a), dans GF(1223) (fig. b) ou dans GF(11489) (fig. c). Le point à l'infini $\mathcal{P}_\infty$ n'est représenté sur aucune des courbes.

La grande majorité des courbes elliptiques utilisées en pratique pour ECC sont définies sur des corps finis binaires GF(2^m) ou des corps finis premiers GF(P). Dans la suite de la section courante, nous listerons quelques exemples de courbes elliptiques définies sur GF(2^m) ou sur GF(P) issus pour la plupart des sites internet *Explicit-Formulas Database* [BL] (EFD) et *SafeCurves* [BL13]. Ces deux sites maintenus par Daniel Bernstein et Tanja Lange ont été une source d'information inestimable pour l'analyse de certaines des implantations matérielles et logicielles de ECC de l'état de l'art que nous avons eu à étudier.

Les courbes elliptiques binaires sont définies sur des corps finis GF(2^m) où m est un nombre premier. Dans la littérature [BL] on peut trouver trois principaux types de courbes elliptiques binaires :

— les courbes de Weierstrass courtes $y^2 + xy = x^3 + a_2x^2 + a_6$;
— les courbes hessiennes $x^3 + y^3 + 1 = 3dxy$;
— les courbes binaires d'Edwards $d_1(x + y) + d_2(x^2 + y^2) = (x + x^2)(y + y^2)$.

Les courbes $\mathcal{E}$/GF(2^m) sont définies pour des paramètres et des coordonnées dans des extensions du corps GF(2). Ils correspondent donc à des polynômes de m coefficients dans GF(2) et calculés modulo un polynôme irréductible. Comme nous l'avons vu en section 2.2, les opérations additions de polynômes à coefficients dans GF(2) se font sans propagation de retenues entre les coefficients. De nombreuses implantations ECC de la littérature utilisent des courbes définies sur des corps finis binaires afin de profiter des opérations arithmétiques rapides inhérentes à ce type de corps. L'utilisation des courbes elliptiques binaires pour ECC est cependant discutée à l'heure actuelle pour des raisons de sécurité (voir [Ber06, BL13] ou [PQ12]). Dans la suite du manuscrit, nous laisserons donc de côté le cas des courbes binaires et nous nous concentrerons sur les courbes recommandées de nos jours et définies sur GF(P).

Il existe de nombreux types et équations de courbes elliptiques $\mathcal{E}/\mathrm{GF}(P)$, dont nous ne listerons qu'un sous-ensemble parmi tous ceux énumérés dans l'EFD [BL] :

— les courbes de Weierstrass courtes $y^2 = x^3 + ax + b$;

— les courbes d'Edwards $x^2 + y^2 = c^2(1 + dx^2y^2)$;

— les courbes « tordues » d'Edwards $ax^2 + y^2 = 1 + dx^2y^2$ (*twisted Edwards curves* en anglais) ;

— les courbes de Montgomery $by^2 = x^3 + ax^2 + x$.

Les courbes de Weierstrass courtes sont les plus répandues dans les standards ECC. On peut citer par exemple les courbes standardisées P-256 (NIST, [KG13]), brainpoolP256t1 (Brainpool, [LM10]), FRP256v1 (ANSSI, [ANS11]) ou secp256k1 (SEC2, [SEC10]). Ces courbes standardisées peuvent être utilisées dans des cryptosystèmes ECC nécessitant un niveau de sécurité théorique de 128 bits.

Pour des implantations de ECC plus performantes, en logiciel ou en matériel, les travaux récents de l'état de l'art s'appuient sur des courbes d'Edwards ou de Montgomery. C'est par exemple le cas de l'implantation matérielle FPGA de Sasdrich et coll. publiée dans [SG15] et basée sur la courbe de Montgomery Curve25519 de Berstein (cf. [Ber06]) pour environ 128 bits de sécurité. Les mêmes auteurs ont d'ailleurs proposé dans [SG17] une nouvelle implantation ECC sur FPGA utilisant la courbe d'Edwards Ed448-Goldilocks proposée par Hamburg dans [Ham15] comme alternative aux courbes du NIST pour 224 bits de sécurité.

Enfin, toujours pour environ 128 bits de sécurité, on peut aussi citer la courbe Four$\mathbb{Q}$, proposée par Costello et coll. en 2015 dans [CL15] et construite à partir d'une courbe tordue d'Edwards définie sur l'extension $\mathrm{GF}\left((2^{127}-1)^2\right)$. Les paramètres et coordonnées de cette courbe sont des polynômes à 2 coefficients dans $\mathrm{GF}(2^{127}-1)$ permettant d'accélérer les opérations arithmétiques modulaires. Les implantations de Four$\mathbb{Q}$ sur FPGA de Jarvinen et coll. présentées en 2016 dans [JMAL16] ont d'ailleurs établi un nouveau record de vitesse parmi les implantations matérielles de ECC pour 128 bits de sécurité : 0.16 ms pour le calcul d'une multiplication scalaire contre 0.40 ms pour l'implantation FPGA de Curve25519 de [SG15] par exemple.

Nous reviendrons sur diverses implantations matérielles sur FPGA des courbes elliptiques présentées ici dans le chapitre 4 où elles nous serviront pour les comparaisons de nos accélérateurs HECC avec l'état de l'art pour ECC avec un même niveau de sécurité théorique de 128 bits.

Nous allons maintenant nous intéresser à la construction des lois de composition définies sur le groupe des points d'une courbe elliptique $\mathcal{E}/\mathbb{K}$. Pour éviter de surcharger les notations utilisées, nous considérerons dans la suite du document que la notation $\mathcal{E}$, sans précision de corps, désigne une courbe elliptique $\mathcal{E}/\mathbb{F}_P$ définie sur $\mathrm{GF}(P)$.

Comme nous l'avons vu précédemment, la courbe $\mathcal{E}$ définit un ensemble de points dont les coordonnées affines (x, y) vérifient l'équation de $\mathcal{E}$, auquel est ajouté le point à l'infini $\mathcal{P}_\infty$. Cet ensemble, que nous noterons aussi $\mathcal{E}$, peut être utilisé pour construire une structure de *groupe* dans lequel on peut définir une *opération d'addition* $(\mathcal{E}, +)$ vérifiant les propriétés suivantes :

— $(\mathcal{P} + \mathcal{Q}) \in \mathcal{E}$ si $\mathcal{P}, \mathcal{Q} \in \mathcal{E}$ et le point résultat $(\mathcal{P} + \mathcal{Q})$ est sur $\mathcal{E}$ et unique (*fermeture*) ;

— $(\mathcal{P} + \mathcal{Q}) + \mathcal{R} = \mathcal{P} + (\mathcal{Q} + \mathcal{R})$ si $\mathcal{P}, \mathcal{Q}, \mathcal{R} \in \mathcal{E}$ (*associativité*) ;

— $\mathcal{P} + \mathcal{P}_\infty = \mathcal{P}_\infty + \mathcal{P} = \mathcal{P}$ si $\mathcal{P} \in \mathcal{E}$ (*élément neutre* $\mathcal{P}_\infty$) ;

— $\mathcal{P} + \mathcal{Q} = \mathcal{Q} + \mathcal{P}$ quels que soient $\mathcal{P}, \mathcal{Q} \in \mathcal{E}$ (*commutativité*) ;

— pour tout point $\mathcal{P}$ de $\mathcal{E}$ il existe un unique point $\overline{\mathcal{P}}$ de $\mathcal{E}$, aussi noté $-\mathcal{P}$, tel que $\overline{\mathcal{P}} + \mathcal{P} = \mathcal{P}_\infty$.

L'opération d'addition dans $\mathcal{E}$ est nommée *addition de points*. Quand les points $\mathcal{P}$ et $\mathcal{Q}$ vérifient $\mathcal{P} = \pm\mathcal{Q}$, l'opération d'addition est notée $\texttt{ADD}(\mathcal{P}, \mathcal{Q}) = \mathcal{P} + \mathcal{Q}$. Dans le cas où $\mathcal{P} = \pm\mathcal{Q}$, l'opération d'addition est

un peu particulière et est notée $\texttt{DBL}(\mathcal{P}) = [2]\mathcal{P} = \mathcal{P} + \mathcal{P}$. Par abus de langage, on appelle *doublement* l'opération d'addition de points pour le cas particulier $\mathcal{P} = \pm\mathcal{Q}$ et simplement *addition* l'opération d'addition de points pour le cas général $\mathcal{P} \neq \pm\mathcal{Q}$.

À partir des opérations d'addition et de doublement de points dans $\mathcal{E}$, on peut construire l'opération de *multiplication scalaire* $[k]\mathcal{P}$ (voir section 2.3.5). Cette opération consiste à multiplier un point $\mathcal{P}$ de $\mathcal{E}$ par un entier positif k :

$$[k]\mathcal{P} = \underbrace{\mathcal{P} + \mathcal{P} + ... + \mathcal{P}}_{k \text{ fois}}. \tag{2.2}$$

Le calcul des opérations d'addition ADD et de doublement DBL de points sur une courbe elliptique par la méthode « corde et séquante » est illustré en figure 2.2 par des exemples utilisant des points $\mathcal{P}$ et $\mathcal{Q}$ d'une courbe $\mathcal{E}/\mathbb{R}$. Si l'utilisation de courbes définies sur $\mathbb{R}$ nous permet d'illustrer plus clairement la construction des opérations sur les points d'une courbe, nous rappelons qu'elles ne sont toutefois jamais utilisées en pratique dans (H)ECC. Les expressions arithmétiques relatives à cette construction s'appliquent cependant elles aussi au cas des corps finis.

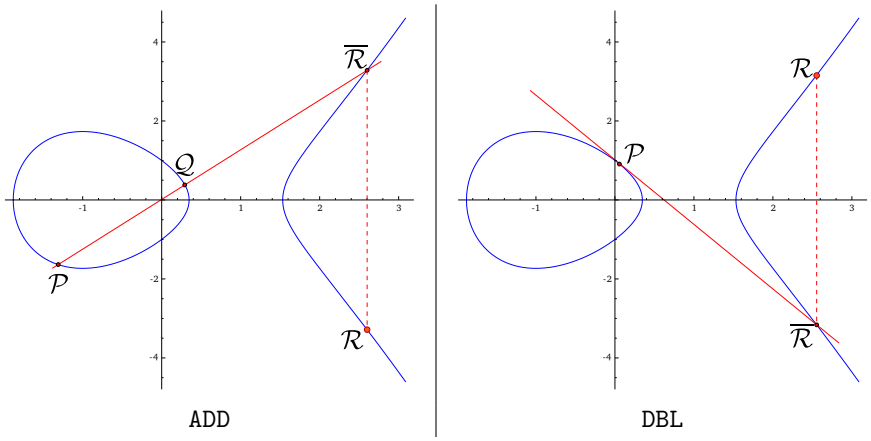


FIGURE 2.2 – Illustration dans $\mathbb{R}$ de l'addition ADD de points $\mathcal{R} = \mathcal{P} + \mathcal{Q}$ et du doublement DBL de point $\mathcal{R} = [2]\mathcal{P}$ sur la courbe elliptique $\mathcal{E}/\mathbb{R} : y^2 = x^3 - 3x + 1$. Le point $\overline{\mathcal{R}}$ est le symétrique de $\mathcal{R}$ par rapport à l'axe des abscisses.

2.3.2 Addition de points dans $\mathcal{E}$ (ADD) pour le cas général $\mathcal{P} \neq \pm\mathcal{Q}$

L'opération d'addition de points ADD dans une courbe elliptique $\mathcal{E}$ est calculée en utilisant la méthode des corde et des sécantes illustrée en figure 2.2 (a). Tout d'abord, on détermine le point $\overline{\mathcal{R}}$ correspondant au point d'intersection entre la droite passant à la fois par $\mathcal{P}$ et $\mathcal{Q}$ et la courbe $\mathcal{E}$. Dans les courbes elliptiques utilisées dans ECC, ce point $\overline{\mathcal{R}}$ est unique (cf. [HMV04]). Le point résultat $\mathcal{R} = \mathcal{P} + \mathcal{Q}$ correspond au symétrique du point $\overline{\mathcal{R}}$ par rapport à l'axe des abscisses.

Les formules utilisées pour déterminer les coordonnées affines du point $\mathcal{R}$ à partir des coordonnées des points $\mathcal{P}$ et $\mathcal{Q}$ et de l'équation de Weierstrass courte de la courbe $\mathcal{E}$ sont données à titre indicatif dans l'équation 2.3 d'après [BL]. L'équation 2.3 permet de mettre en avant le fait que l'addition de points sur une courbe elliptique implique de calculer un certain nombre d'opérations arithmétiques sur les

coordonnées des points de la courbe : multiplications, additions, soustractions ou divisions dans GF(P).

$$\begin{aligned} x_{\mathcal{R}} &= \frac{(y_{\mathcal{Q}} - y_{\mathcal{P}})^2}{(x_{\mathcal{Q}} - x_{\mathcal{P}})^2} - x_{\mathcal{P}} - x_{\mathcal{Q}}; \\ y_{\mathcal{R}} &= \frac{(2\,x_{\mathcal{P}} + x_{\mathcal{Q}}) \times (y_{\mathcal{Q}} - y_{\mathcal{P}})}{(x_{\mathcal{Q}} - x_{\mathcal{P}})} - \frac{(y_{\mathcal{Q}} - y_{\mathcal{P}})^3}{(x_{\mathcal{Q}} - x_{\mathcal{P}})^3} - y_{\mathcal{P}}. \end{aligned} \tag{2.3}$$

2.3.3 Doublement de points dans $\mathcal{E}$ (DBL) pour le cas particulier $\mathcal{P} = \pm\mathcal{Q}$

L'opération de doublement de point DBL est un cas particulier de l'addition pour lequel les abscisses $x_{\mathcal{P}}$ et $x_{\mathcal{Q}}$ des points $\mathcal{P}$ et $\mathcal{Q}$ sont égales, ce qui provoquerait des divisions par 0 dans l'équation 2.3. Comme illustré en figure 2.2 (b), la méthode des cordes et des sécantes est aussi utilisable pour le doublement de points mais en considérant la tangente à la courbe passant par le point $\mathcal{P}$. De même que la sécante utilisée pour l'addition, la tangente à $\mathcal{E}$ passant par $\mathcal{P}$ croise la courbe en un unique point $\overline{\mathcal{R}}$ de $\mathcal{E}$ dont le symétrique $\mathcal{R}$ par rapport à l'axe des abscisses est le point résultat $\mathcal{R} = [2]\mathcal{P}$.

$$\begin{aligned} x_{\mathcal{R}} &= \frac{(3\,x_{\mathcal{P}}^2 + a)^2}{(2\,y_{\mathcal{P}})^2} - 2\,x_{\mathcal{P}}; \\ y_{\mathcal{R}} &= \frac{3\,x_{\mathcal{P}} \times (3\,x_{\mathcal{P}}^2 + a)}{2\,y_{\mathcal{P}}} - \frac{(3\,x_{\mathcal{P}}^2 + a)^3}{(2\,y_{\mathcal{P}})^3} - y_{\mathcal{P}}. \end{aligned} \tag{2.4}$$

Les formules servant au calcul des coordonnées affines $(x_{\mathcal{R}}, y_{\mathcal{R}})$ du point $\mathcal{R}$ à partir du point $\mathcal{P}$ pour une courbe elliptique de Weierstrass courte sont données dans l'équation 2.4 tirée de [BL]. Elles sont en général un peu moins complexes que les formules d'addition mais impliquent néanmoins elles aussi le calcul d'opérations arithmétiques coûteuses dans GF(P) (en particulier les opérations de division).

En comparant les équations 2.3 et 2.4, on peut constater que ces dernières sont très similaires et que les coordonnées $x_{\mathcal{R}}$ et $y_{\mathcal{R}}$ du point $\mathcal{R} = \mathcal{P} + \mathcal{Q}$ peuvent être réécrites sous la forme :

$$\begin{aligned} x_{\mathcal{R}} &= \lambda^2 - x_{\mathcal{P}} - x_{\mathcal{Q}}; \\ y_{\mathcal{R}} &= \lambda(x_{\mathcal{P}} - x_{\mathcal{R}}) - y_{\mathcal{P}}; \end{aligned}$$

dans laquelle λ désigne la pente de la droite passant par $\mathcal{P}$ et $\mathcal{Q}$ et dont la valeur est :

$$\lambda = \begin{cases} \dfrac{(y_{\mathcal{Q}} - y_{\mathcal{P}})}{(x_{\mathcal{Q}} - x_{\mathcal{P}})} & \text{si } \mathcal{P} \neq \pm\mathcal{Q} \\ \dfrac{(3x_{\mathcal{P}}^2 + a)}{2x_{\mathcal{P}}} & \text{si } \mathcal{P} = \pm\mathcal{Q}. \end{cases}$$

2.3.4 Opérations ADD et DBL en coordonnées projectives

Plusieurs optimisations des formules d'addition ADD et de doublement DBL de points ont été proposées dans la littérature pour accélérer le calcul de ces opérations. Elles visent pour la plupart à remplacer l'opération de division nécessaire au calcul de λ dans GF(P) par des opérations moins complexes et moins coûteuses à calculer (multiplications et carrés par exemple). En effet, le coût de cette opération est souvent estimé dans les articles de la littérature comme équivalent au calcul de quelques dizaines, voir de quelques centaines de multiplications modulaires. La méthode la plus courante pour se débarrasser des divisions modulaires dans les formules d'addition et de doublement est l'utilisation de systèmes de

coordonnées	DBL		ADD	
affines	A → A	1I + 2M + 2S	A + A → A	1I + 2M + 1S
projectives standards	P → P	7M + 3S	P + P → P	12M + 2S
jacobiennes	J → J	4M + 4S	J + J → J	12M + 4S
			J + A → J	8M + 3S

TABLE 2.1 – Nombre d'inversions (I), de multiplications (M) et de carrés (S) dans GF(P) pour les opérations ADD et DBL sur la courbe de Weierstrass courte $\mathcal{E}/\mathbb{F}_P : y^2 = x^3 - 3x + b$ pour différents systèmes de coordonnées (A = affines, P = projectives standards, J = jacobiennes) [HMV04].

coordonnées projectives à la place des *coordonnés affines* utilisées dans les équations 2.3 et 2.4.

Par exemple, dans le système de coordonnées projectives standards (*standard projective coordinates*) les points de la courbe elliptique $\mathcal{E}$ sont représentés par des triplets de coordonnées projectives (X, Y, Z) au lieu du couple de coordonnées affines (x, y). Les coordonnées X, Y et Z sont définies de façon à vérifier les égalités suivantes : $x = X/Z$ et $y = Y/Z$ (cf. [BL] ou [HMV04] p.89). L'opposé du point $\mathcal{P} = (X, Y, Z)$ est le point $\overline{\mathcal{P}} = (X, -Y, Z)$ et la projection du point à l'infini $\mathcal{P}_\infty$ correspond au point $(0, 1, 0)$.

Un autre exemple classique de système de coordonnées projectives est le système de coordonnées jacobiennes dans lequel le couple de coordonnées affines (x, y) est remplacé par le triplet de coordonnées (X, Y, Z) vérifiant $x = X/Z^2$ et $y = Y/Z^3$ (cf. [BL] ou [HMV04] p.90). L'opposé du point $\mathcal{P} = (X, Y, Z)$ est le point $\overline{\mathcal{P}} = (X, -Y, Z)$ et la projection du point à l'infini $\mathcal{P}_\infty$ correspond au point $(1, 1, 0)$.

Pour accélérer les calculs de l'opération ADD dans le système de coordonnées jacobiennes, il est possible de représenter l'un des point en coordonnées affines (on parle alors d'addition en coordonnées mixtes). Cette optimisation est principalement utilisée dans le cas où l'un des points utilisé dans l'opération ADD reste le même pendant tout le calcul de la multiplication scalaire, comme c'est par exemple le cas dans l'algorithme 2 de multiplication scalaire présenté en section 2.3.5.

Le tableau 2.1 tiré de [HMV04] présente une comparaison des nombres d'opérations modulaires dans GF(P) nécessaires aux calculs des opérations ADD et DBL sur la courbe elliptique $\mathcal{E}/\mathbb{F}_P : y^2 = x^3 + ax + b$, avec $a = -3$ comme c'est souvent le cas en cryptographie, et pour différents systèmes de coordonnées : coordonnées affines (A), projectives standard (P) ou jacobiennes (J). Dans ce tableau, les additions et soustractions dans GF(P) ne sont pas comptées car elles sont peu coûteuses en comparaison des inversions (I), des multiplications (M) et des carrés (S) modulaires.

De nombreux cryptoprocesseurs matériels pour (H)ECC proposés dans la littérature s'appuient sur l'utilisation de coordonnées projectives pour accélérer le calcul des opérations au niveau courbe ou réduire les coûts des implantations. Il est important de tenir compte de ce fait quand on veut comparer les performances de différentes implantations.

2.3.5 Multiplication scalaire $[k]\mathcal{P}$ dans $\mathcal{E}$

La multiplication scalaire $[k]\mathcal{P}$ permet de multiplier le point $\mathcal{P}$ de $\mathcal{E}$ par l'entier k. Cela revient à additionner k fois le point $\mathcal{P}$ avec lui même en utilisant les opérations ADD et DBL (cf. équation 2.2 p.20). Plusieurs méthodes ont été proposées pour le calcul de la multiplication scalaire, dont nous listerons les plus utilisées dans les sous-sections suivantes. Elles sont pour la plupart dérivées de méthodes utilisées pour le calcul de l'exponentiation : $x^n, \forall n \in \mathbb{N}$.

Méthode binaire *double-and-add* de multiplication scalaire

La méthode binaire de multiplication scalaire rappelée ici, nommée « *double-and-add* » en anglais, est dérivée de la méthode d'exponentiation rapide *square-and-multiply* (cf. [Knu98] chap. 4.6.3, algo. A). Il s'agit d'une méthode itérative dans laquelle les bits k_i du scalaire k sont parcourus successivement, et des opérations au niveau courbe calculées pour chaque k_i : `DBL` quelle que soit la valeur de k_i et `ADD` si $k_i = 1$. La première version dans l'algorithme 1 parcourt les bits de k depuis le bit de poids faible (LSB). La deuxième version dans l'algorithme 2 parcourt les bits de k depuis le bit de poids fort (MSB). Dans ces deux algorithmes, $\mathcal{P}_\infty$ dénote le point à l'infini de la courbe $\mathcal{E}$. Le scalaire k est représenté sur n_k bits.

Opérandes : $k \in \mathbb{N}$ sur n_k bits, $\mathcal{P} \in \mathcal{E}$
Résultat : $\mathcal{Q} = [k]\mathcal{P}, \mathcal{Q} \in \mathcal{E}$
begin
1 $\quad \mathcal{Q} \leftarrow \mathcal{P}_\infty$
2 $\quad$ **for** $i = 0$ **to** $n_k - 1$ **do**
3 $\quad\quad$ **if** $k_i = 1$ **then**
4 $\quad\quad\quad \mathcal{Q} \leftarrow \mathcal{Q} + \mathcal{P}$
5 $\quad\quad \mathcal{P} \leftarrow [2]\mathcal{P}$
6 $\quad$ **return** $\mathcal{Q}$

Algorithme 1 : Multiplication scalaire binaire avec parcours des n_k bits du scalaire k depuis le LSB ; « *right-to-left double-and-add* » en anglais.

Opérandes : $k \in \mathbb{N}$ sur n_k bits, $\mathcal{P} \in \mathcal{E}$
Résultat : $\mathcal{Q} = [k]\mathcal{P}, \mathcal{Q} \in \mathcal{E}$
begin
1 $\quad \mathcal{Q} \leftarrow \mathcal{P}_\infty$
2 $\quad$ **for** $i = n_k - 1$ **downto** 0 **do**
3 $\quad\quad \mathcal{Q} \leftarrow [2]\mathcal{Q}$
4 $\quad\quad$ **if** $k_i = 1$ **then**
5 $\quad\quad\quad \mathcal{Q} \leftarrow \mathcal{Q} + \mathcal{P}$
6 $\quad$ **return** $\mathcal{Q}$

Algorithme 2 : Multiplication scalaire binaire avec parcours des n_k bits du scalaire k depuis le MSB ; « *left-to-right double-and-add* » en anglais.

Pour des raisons de sécurité, les n_k bits des scalaires k utilisés pour (H)ECC sont choisis de façon uniforme et équiprobable. On peut donc considérer qu'en moyenne $n_k/2$ bits du scalaire k valent 1. Le nombre d'opérations sur la courbe $\mathcal{E}$ calculées dans les algorithmes 1 et 2 est alors d'environ

$$\frac{n_k}{2}\,\texttt{ADD} + n_k\,\texttt{DBL}.$$

Le nombre d'opérations arithmétiques calculées dans GF(P) lors de la multiplication scalaire dépend lui de l'équation de la courbe $\mathcal{E}$ et du système de coordonnées utilisé. Pour la courbe elliptique d'équation $y^2 = x^3 - 3x + b$, le tableau 2.1 nous permet d'estimer ce nombre d'opérations à $2.5n_k\,\texttt{S} + 3n_k\,\texttt{M} + 1.5n_k\,\texttt{I}$ pour les coordonnées affines et à $10n_k\,\texttt{M} + 6n_k\,\texttt{S}$ pour les coordonnées jacobiennes par exemple.

Enfin, on peut constater que si le point $\mathcal{Q}$ est modifié dans l'algorithme 2, le point $\mathcal{P}$ reste lui inchangé durant les itérations du *double-and-add*. Il est donc possible dans cette version d'utiliser un système de

coordonnées mixtes dans lequel le point $\mathcal{Q}$ est représenté en coordonnées jacobiennes et le point $\mathcal{P}$ en coordonnées affines. Le nombre d'opérations arithmétiques dans $\mathbb{K}$ moyen pour la multiplication scalaire est alors estimé à $8n_k\,\mathtt{M} + 5.5n_k\,\mathtt{S}$.

Multiplication scalaire avec recodage du scalaire k

Pour accélérer le calcul de la multiplication scalaire, il est possible de modifier la représentation du scalaire k de manière à réduire le nombre d'opérations `ADD` ou `DBL` à effectuer au niveau courbe. Parmi les différents types de recodage que l'on peut trouver dans la littérature, nous nous intéresserons à deux d'entre eux : le *recodage en forme non adjacente* (NAF) et le *recodage* NAF *fenêtré* ($\text{NAF}_\mathtt{w}$). Ceux-ci sont utilisés fréquemment dans les implantations de ECC et HECC de l'état de l'art. En particulier, ils sont implantés dans certains des accélérateurs matériels auxquels nous nous comparerons dans le chapitre 4.

Le recodage en forme non adjacente NAF se base sur une *représentation en chiffres signés* d'un entier positif k :

$$\text{NAF}(k) = \sum_{i=0}^{l-1} \alpha_i\, 2^i, \quad \alpha_i \in \{-1, 0, 1\}. \tag{2.5}$$

Dans l'expression 2.5, la variable l désigne le nombre de chiffres nécessaires à la représentation du scalaire k en NAF, avec $k_{l-1} \neq 0$. La représentation NAF est construite en utilisant l'algorithme 3 et de façon à empêcher que deux chiffres consécutifs soient non nuls : si $\alpha_i = 1$ alors $\alpha_{i-1} = 0$ et $\alpha_{i+1} = 0$).

Opérandes : $k \in \mathbb{N}$ sur n_k bits
Résultat : $\text{NAF}(k) = \sum_{i=0}^{l-1} \alpha_i\, 2^i, \alpha_i \in \{-1, 0, 1\}$
begin

```
1  | i ← 0
2  | while k ≥ 1 do
3  |   if k mod 2 = 1 then
4  |     α_i ← 2 − (k mod 4)
5  |     k ← k − α_i
6  |   else
7  |     α_i ← 0
8  |   k ← k/2
9  |   i ← i + 1
10 | l ← i
11 | return NAF(k) = (α_{l−1}, α_{l−2}, ..., α_1, α_0)_NAF
```

Algorithme 3 : Conversion du scalaire k en représentation NAF, d'après l'algorithme décrit dans [HMV04].

D'après [HMV04], la représentation NAF de l'entier positif k a les propriétés suivantes :

- le scalaire k a une unique représentation NAF, notée $\text{NAF}(k)$;
- $\text{NAF}(k)$ est la représentation de k possédant le moins de chiffres non nuls ;
- la longueur l de $\text{NAF}(k)$ est au plus d'un chiffre supplémentaire par rapport à la représentation binaire de k : $l \leq n_k + 1$;
- pour l la longueur de $\text{NAF}(k)$, $2^{l+1}/3 > k > 2^l/3$;
- la densité moyenne de chiffres non nuls dans $\text{NAF}(k)$ est environ de $1/3$.

L'algorithme 4 présente une version modifiée de méthode binaire de multiplication scalaire depuis le LSB, cf. algorithme 1, utilisant un recodage NAF du scalaire k. Dans cet algorithme, le recodage du

scalaire en NAF est effectué une unique fois en début de multiplication scalaire. L'autre différence avec l'algorithme d'origine est le calcul de la soustraction de points dans le cas où $k_i = -1$.

Opérandes : $k \in \mathbb{N}$ sur n_k bits, $\mathcal{P} \in \mathcal{E}$
Résultat : $\mathcal{Q} = [k]\mathcal{P}, \mathcal{Q} \in \mathcal{E}$
begin
1 Utiliser l'algorithme 3 pour calculer $\alpha = \text{NAF}(k) = \sum_{i=0}^{l-1} \alpha_i \, 2^i, \alpha_i \in \{-1, 0, 1\}$
2 $\mathcal{Q} \leftarrow \mathcal{P}_\infty$
3 **for** $i = l - 1$ **downto** 0 **do**
4 $\mathcal{Q} \leftarrow [2]\mathcal{Q}$
5 **if** $\alpha_i = 1$ **then**
6 $\mathcal{Q} \leftarrow \mathcal{Q} + \mathcal{P}$
7 **if** $\alpha_i = -1$ **then**
8 $\mathcal{Q} \leftarrow \mathcal{Q} - \mathcal{P}$
9 **return** $\mathcal{Q}$

Algorithme 4 : Multiplication scalaire en représentation NAF avec parcours depuis les poids forts des l chiffres $\alpha_i \in \{-1, 0, 1\}$ de la représentation NAF du scalaire k, d'après [HMV04].

Dans les courbes elliptiques définies sur GF(P) la soustraction de points a la même complexité que l'addition de points, grâce au fait que $\mathcal{P} - \mathcal{Q} = \mathcal{P} + \overline{\mathcal{Q}}$ avec $\overline{\mathcal{Q}} = (x_\mathcal{Q}, -y_\mathcal{Q})$. Pour l'évaluation de la complexité de l'algorithme 4, on considèrera donc les soustractions de points comme des additions `ADD`. D'après les propriétés du NAF, la densité de chiffres non nuls dans NAF(k) est d'à peu près 1/3 et il y a au plus 1 chiffre supplémentaire dans cette représentation par rapport au nombre n_k de bits de k. La complexité de la multiplication scalaire utilisant le recodage NAF dans l'algorithme 4 est donc d'environ

$$\frac{n_k}{3}\,\texttt{ADD} + n_k\,\texttt{DBL}.$$

Le recodage NAF_w est une généralisation du recodage NAF qui utilise une méthode fenêtrée pour traiter w bits du scalaire k en même temps. Nous rappelons dans l'algorithme 5 la méthode de conversion du scalaire k en représentation NAF fenêtrée : $\text{NAF}_\mathsf{w}(k) = \sum_{i=0}^{l-1} \beta_i \, 2^i, \beta_i \in [-2^{\mathsf{w}-1}, 2^{\mathsf{w}-1} - 1]$ et β_i entier. Le recodage du scalaire k en représentation NAF fenêtrée présenté dans [HMV04] a les propriétés suivantes :

— le scalaire k a une unique représentation NAF fenêtrée $\text{NAF}_\mathsf{w}(k)$;
— $\text{NAF}_2(k) = \text{NAF}(k)$;
— la longueur l de $\text{NAF}_\mathsf{w}(k)$ est au plus d'un chiffre supplémentaire par rapport à la représentation binaire de k ;
— la densité moyenne de chiffres non nuls dans $\text{NAF}_\mathsf{w}(k)$ est environ de $1/(\mathsf{w}+1)$.

L'algorithme 6 correspond à la méthode binaire de multiplication scalaire depuis les poids forts calculée en utilisant le recodage fenêtré $\text{NAF}_\mathsf{w}(k)$ du scalaire k. Dans cet algorithme, il est nécessaire pour calculer $[k]\mathcal{P}$ de pré-calculer et de mémoriser différents multiples du point $\mathcal{P}$. Pour cette raison, le recodage NAF_w (avec w grand) est surtout utilisable dans les implantations pour lesquelles la mémoire n'est pas une ressource limitée. L'étape de pré-calcul nécessite de calculer environ $\left(\left(2^{\mathsf{w}-1} - 1\right)/2\right)$ multiples de $\mathcal{P}$, ce qui peut être effectué grâce à un doublement de point et $\left(2^{\mathsf{w}-2} - 1\right)$ additions de points.

D'après les propriétés de la représentation fenêtrée NAF_w listées ci-dessus, la complexité de la multi-

Opérandes : $k \in \mathbb{N}$ sur n_k bits
Résultat : $\mathrm{NAF}_{\mathsf{w}}(k) = \sum_{i=0}^{l-1} \beta_i \, 2^i, \beta_i \in [-2^{\mathsf{w}-1}, 2^{\mathsf{w}-1} - 1]$ avec β_i entier
begin

```
1  | i ← 0
2  | while k ≥ 1 do
3  |   if k mod 2 = 1 then
4  |     β_i ← k mod 2^w
5  |     k ← k − β_i
6  |   else
7  |     β_i ← 0
8  |   k ← k/2
9  |   i ← i + 1
10 | l ← i
11 | return NAF_w(k) = (β_{l−1}, β_{l−2}, ..., β_1, β_0)_{NAF_w}
```

Algorithme 5 : Conversion du scalaire k en représentation $\mathrm{NAF}_{\mathsf{w}}$ fenêtrée, d'après l'algorithme décrit dans [HMV04].

Opérandes : $k \in \mathbb{N}$ sur n_k bits, $\mathcal{P} \in \mathcal{E}$
Résultat : $\mathcal{Q} = [k]\mathcal{P}, \mathcal{Q} \in \mathcal{E}$
begin

```
1  | Utiliser l'algorithme 5 pour calculer NAF_w(k) = Σ_{i=0}^{l−1} β_i 2^i, β_i ∈ [−2^{w−1}, 2^{w−1} − 1] avec β_i entier
2  | Pré-calculer P_j = [j]P pour i ∈ {1, 3, ..., 2^{w−1} − 1}
3  | Q ← P_∞
4  | for i = l − 1 downto 0 do
5  |   Q ← [2]Q
6  |   if β_i ≠ 0 then
7  |     if β_i > 0 then
8  |       Q ← Q + P_{β_i}
9  |     else
10 |       Q ← Q − P_{−β_i}
11 | return Q
```

Algorithme 6 : Multiplication scalaire en représentation $\mathrm{NAF}_{\mathsf{w}}$ fenêtrée avec parcours depuis les poids forts des l chiffres entiers $\beta_i \in [-2^{\mathsf{w}-1}, 2^{\mathsf{w}-1} - 1]$ de la représentation $\mathrm{NAF}_{\mathsf{w}}$ du scalaire k, d'après [HMV04].

plication scalaire complète de l'algorithme 6 en terme d'opérations au niveau courbe est alors d'environ

$$\left[\left(2^{\mathsf{w}-2} - 1\right) \, \mathtt{ADD} + 1 \, \mathtt{DBL}\right] + \left[\frac{n_k}{\mathsf{w}+1} \, \mathtt{ADD} + n_k \, \mathtt{DBL}\right].$$

Les deux algorithmes discutés ci-dessus utilisent un recodage spécifique du scalaire k pour réduire le nombre d'additions de points à calculer dans les itérations de la boucle parcourant les bits ou chiffres de k. Cette optimisation permet de réduire le temps de calcul de la multiplication scalaire au prix d'une complexité un peu plus élevée des algorithmes, et d'un besoin de mémoire accru pour le stockage des valeurs précalculées pour la représentation NAF fenêtrée. [3]

3. En pratique on utilise $\mathsf{w} = 2$, 3 ou 4 mais rarement plus.

Robustesse de la multiplication scalaire face aux attaques par observation des canaux auxiliaires

Dans les algorithmes de type *double-and-add*, avec ou sans recodage du scalaire, les opérations effectuées pour chacun des chiffres du scalaire dépendent de la valeur de ces derniers : 1 `DBL` si la valeur du chiffre est à zéro, et 1 `DBL` et 1 `ADD` sinon. Si les opérations de `DBL` et `ADD` utilisent des formules différentes, il est possible de les différentier en utilisant des méthodes d'analyse de canaux auxiliaires (SCA), comme l'a montré Kocher en 1996 dans [Koc96] par analyse du temps de calcul (dans le cas de RSA).

Par exemple, *l'analyse de puissance simple* (SPA) se base sur l'analyse de la consommation instantanée d'énergie au sein d'un circuit électronique lors de l'exécution d'un algorithme cryptographique pour essayer d'extraire des informations sensibles sur les données manipulées dans cet algorithme. Dans le cas des algorithmes de *double-and-add* cités ci-dessus, les opérations de `DBL` et `ADD` se basent sur des formules différentes dans lesquelles des opérations modulaires différentes sont calculées. Les profils de consommation pour ces deux opérations sont donc différentiables (voir [MOP08b] chap. 5 et [KJB99]). En étudiant la consommation d'énergie dans un circuit calculant un *double-and-add* classique, il est alors possible de déterminer si un unique `DBL` a été effectué, auquel cas le bit k_i vaut 0, ou si on a calculé un `DBL` *et* un `ADD`, auquel cas le bit k_i vaut 1. Ces fuites d'informations par l'intermédiaire des canaux physiques auxiliaires, tels que la consommation d'énergie dans le cas de la SPA, posent problème pour la sécurité des cryptosystèmes dans lesquels le scalaire k doit être gardé secret, par exemple quand k est la clé privée. Ces méthodes sont aussi utilisables dans le cas plus général de l'analyse d'une *grandeur physique*[4] dont la mesure durant *une exécution* d'un algorithme cryptographique permet de révéler des informations sur les données manipulées.

Comme nous l'avons rappelé dans l'introduction du manuscrit, il existe aussi d'autres types d'attaques par observation, telles que la DPA [KJB99], et des attaques par perturbations et injection de fautes telles que les attaques *safe-error* [YJ00].

Dans [BJ02] publié en 2002, Brier et Joye ont listé trois méthodes permettant de limiter les fuites d'informations lors du calcul de l'opération de multiplication scalaire en effectuant toujours les mêmes opérations au niveau du corps quelle que soit la valeur du scalaire k.

La première, proposée par Coron en 1999 dans [Cor99] et appelée *double-and-add-always*, consiste à effectuer une addition de point `ADD` *factice* supplémentaire dans chaque itération de l'algorithme pour laquelle $k_i = 0$. Pour chaque bit du scalaire k, on calcule donc bien toujours 1 `DBL` et 1 `ADD`, et les profils de consommation sont les mêmes pour chaque itération de l'algorithme quelle que soit la valeur du scalaire. L'utilisation de l'algorithme *double-and-add-always* a cependant pour inconvénient de rajouter des *calculs inutiles* dans l'algorithme de multiplication scalaire. De plus, et bien que prodiguant une certaine résistance face aux attaques par SPA, cette contre-mesure est faible face à l'utilisation d'attaques de type *safe-error* [YJ00]. Celles-ci permettent de retrouver les bits de clé privée en injectant des fautes matérielles dans le circuit afin de modifier les résultats des opérations `ADD`. Si l'injection de fautes dans un `ADD` *modifie le résultat* de la multiplication scalaire, alors ce `ADD` n'est pas une opération factice et on sait que le bit courant k_i vaut 1. Au contraire, si l'injection de fautes dans un `ADD` *ne modifie pas le résultat* de la multiplication scalaire, alors ce `ADD` est une opération factice et on sait que le bit courant k_i vaut 0.

La seconde méthode pour augmenter la résistance face aux attaques par SPA est l'utilisation de courbes elliptiques spécifiques pour lesquelles les formules d'addition et de doublement de points sont identiques.

4. la SPA s'appuie sur la consommation électrique mais il est aussi possible de mesurer le temps d'exécution (cf. [Koc96]), le rayonnement électromagnétique, les variations de température ou même celles de l'activité acoustique par exemple.

On parle alors de *formules unifiées*. De telles courbes ont été par exemple introduites par Liardet et Smart en 2001 dans [LS01] ou par Joye et Quisquater la même année dans [JQ01], ces derniers utilisant des courbes hessiennes. Cependant, et contrairement aux courbes de Weierstrass, ces courbes elliptiques spécifiques ainsi que les paramètres qu'elles utilisent ne sont pas toujours compatibles avec les standards recommandés, par exemple [KG13] ou [SEC10]. Leur utilisation est donc limitée à des cas d'implantations particuliers.

Enfin, la troisième et dernière méthode rapportée par Brier et Joye dans [BJ02] est l'utilisation de l'algorithme de *l'échelle de Montgomery*, ou Montgomery *ladder* en anglais, décrite en 1987 dans [Mon87]. La méthode de multiplication scalaire décrite dans l'algorithme 7 est une des versions présentées par Joye et Yen dans [JY02]. Elle s'appuie sur l'utilisation de l'échelle de Montgomery grâce à laquelle les opérations au niveau courbe effectuées dans les itérations de la boucle parcourant le scalaire k sont identiques quelle que soit la valeur de k. L'échelle de Montgomery a aussi la propriété de garder fixe la différence entre les points $\mathcal{R}_0$ et $\mathcal{R}_1$ calculés à chaque itération. Dans l'algorithme 7 on a ainsi $\mathcal{R}_0 = \left[k \bmod 2^i\right]\mathcal{P}$ et $\mathcal{R}_1 = \left[(k \bmod 2^i)+1\right]\mathcal{P}$ à l'itération i de la boucle parcourant les bits du scalaire k, et on a bien alors $\mathcal{R}_1 - \mathcal{R}_0 = \mathcal{P}$ quelle que soit l'itération considérée. Grâce au fait que les opérations effectuées dans chaque itération de la boucle externe soient indépendantes du bit k_i courant, la multiplication scalaire de l'algorithme 7 est résistante aux attaques par SPA. Toutes ces opérations sont par ailleurs utilisées dans le calcul de la multiplication scalaire et l'algorithme est donc résistant aux attaques *safe-error*. Toutefois, l'affectation des résultats des opérations `ADD` et `DBL` aux nouveaux points $\mathcal{R}_0$ et $\mathcal{R}_1$ dans les itérations de la boucle externe de l'algorithme 7 dépend de la valeur du bit courant du scalaire. Cette sélection des points $\mathcal{R}_0$ et $\mathcal{R}_1$ en fonction de la valeur du bit k_i courant peut provoquer des fuites d'informations utilisables pour des attaques DPA. Par exemple, si les coordonnées des points $\mathcal{R}_0$ et $\mathcal{R}_1$ sont stockées en mémoire dans des registres distincts, la variation de poids de Hamming dans les adresses de ces registres peut être exploitée par la DPA. Cependant, la mise en œuvre d'attaques par DPA dans le cadre des protocoles tels que l'échange de clé ECDH ou la signature numérique ECDSA (voir section 2.5) est bien plus difficile en pratique. Dans le protocole ECDH, le scalaire k est en effet tiré aléatoirement au début de chaque échange et utilisé dans 2 multiplications scalaires. Dans le protocole ECDSA, le scalaire k est aussi tiré aléatoirement et n'est utilisé que dans une unique multiplication scalaire pour la mise en place de la signature. La mise en place d'une attaque DPA contre ces cryptosystèmes, nécessitant de rejouer de nombreuses fois le calcul de la multiplication scalaire avec le même scalaire, n'a alors que peu de sens. On notera toutefois qu'il est possible d'implanter certaines protections contre la DPA dans les accélérateurs. On citera par exemple la *randomisation du scalaire* k, le *masquage du point de base* $\mathcal{P}$ dans le calcul de la multiplication scalaire $[k]\mathcal{P}$ (cf. [Cor99]) ou la *randomisation des adresses mémoires*. Ces protections ont pour but de compliquer les analyses statistiques utilisées dans la DPA en introduisant de l'aléa dans les signaux de contrôle et les données manipulées dans les cryptoprocesseurs. Nous renvoyons le lecteur intéressé vers [FV12] où sont listées certaines contre-mesures contre la SPA et la DPA.

2.4 Les courbes hyperelliptiques

L'utilisation des courbes hyperelliptiques pour les cryptosystèmes basés sur le problème du logarithme discret a été proposée par Koblitz en 1988 dans [Kob88]. Les courbes hyperelliptiques peuvent être vues comme une généralisation des courbes elliptiques. Leur utilisation est motivée en cryptographie par le fait que la taille des éléments de $\mathrm{GF}(P)$ dans HECC peut être *divisée par 2* par rapport à ECC pour le même niveau de sécurité, ce qui permet de réduire le coût des opérations modulo P.

Opérandes : $k \in \mathbb{N}$ sur n_k bits, $\mathcal{P} \in \mathcal{E}$
Résultat : $\mathcal{R}_0 = [k]\mathcal{P}, \mathcal{R}_0 \in \mathcal{E}$
begin

```
1  | R0 ← P
2  | R1 ← [2]P
3  | for i = n_k − 2 downto 0 do
4  |   if k_i = 0 then
5  |     R1 ← R0 + R1
6  |     R0 ← [2]R0
7  |   else if k_i = 1 then
8  |     R0 ← R0 + R1
9  |     R1 ← [2]R1
10 | return R0
```

Algorithme 7 : Multiplication scalaire $[k]\mathcal{P}$ par l'échelle de Montgomery telle que présentée dans [JY02].

Dans cette section, nous allons présenter les principales différences entre les courbes hyperelliptiques et les courbes elliptiques. Ces différences concerneront principalement la définition des courbes, la structure de groupe pour le calcul de la multiplication scalaire et les formules utilisées pour les opérations d'addition et de doublement des éléments dans ce groupe.

2.4.1 Définition

Une courbe hyperelliptique $\mathcal{C}/\mathbb{K}$ est une courbe définie par une équation à deux variables (x, y) de la forme :

$$\mathcal{C}/\mathbb{K} : y^2 + h(x)\, y = f(x) \tag{2.6}$$

dans laquelle $h(x)$ et $f(x)$ sont des polynômes à coefficients dans $\mathbb{K}$.

En particulier, $f(x)$ est un *polynôme unitaire* (c.-à-d. dont le coefficient de plus haut degré vaut 1) de degré égal à $2g + 1$ et $h(x)$ est un polynôme de degré inférieur ou égal à g. Le paramètre g est nommé *genre* de la courbe hyperelliptique $\mathcal{C}$. On peut d'ailleurs remarquer qu'une courbe elliptique d'équation de Weierstrass classique (cf. équation 2.1) correspond à une courbe hyperelliptique de genre $g = 1$ dans laquelle $h(x) = a_1x + a_3$ et $f(x) = x^3 + a_2x^2 + a_4x + a_6$.

En figure 2.3, nous présentons un exemple de courbe hyperelliptique $\mathcal{C}$ de genre 2 définie sur les réels par l'équation suivante pour laquelle $h(x) = 0$:

$$\mathcal{C}/\mathbb{R} : y^2 = x^5 - 5x^3 + 4x^2 + 1. \tag{2.7}$$

La courbe hyperelliptique $\mathcal{C}/\mathbb{K}$ permet de définir un ensemble de points dont les coordonnées affines correspondent aux couples (x, y) vérifiant l'équation 2.6, auquel on ajoute un point à l'infini $\mathcal{P}_\infty$ défini de la même façon que pour les courbes elliptiques. La principale différence entre les courbes elliptiques et les courbes hyperelliptiques de genre $g > 1$ est que les ensembles des points définis par ces dernières *n'ont pas une structure de groupe*. Cela signifie que l'on ne peut pas définir d'opération d'addition entre les points d'une courbe hyperelliptique de genre g supérieur ou égal à 2.

En 1988, Koblitz a proposé dans [Kob88] d'utiliser *les Jacobiennes* des courbes hyperelliptiques pour construire un groupe de points possédant une loi d'addition. Sans rentrer dans les détails mathématiques,

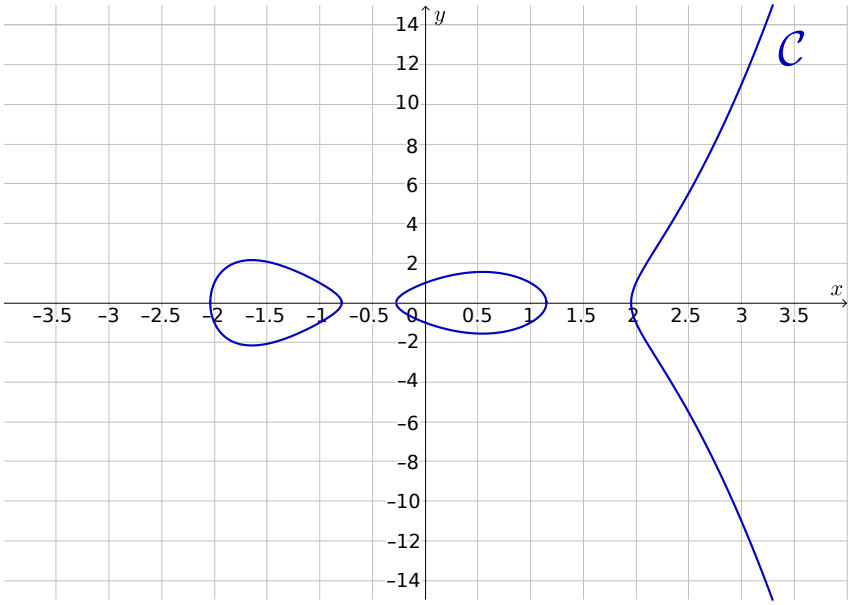


FIGURE 2.3 – Exemple de courbe hyperelliptique de genre 2 définie dans les réels $\mathbb{R}$ par l'équation 2.7.

la Jacobienne $\mathcal{J}_\mathcal{C}/\mathbb{K}$ d'une courbe hyperelliptique $\mathcal{C}/\mathbb{K}$ peut être vue comme un ensemble de paires de points de $\mathcal{C}/\mathbb{K}$ vérifiant certaines propriétés. Les éléments de $\mathcal{J}_\mathcal{C}/\mathbb{K}$ sont nommés *diviseurs*. La loi d'addition définie sur le groupe des points de la Jacobienne $\mathcal{J}_\mathcal{C}/\mathbb{K}$ d'une courbe hyperelliptique $\mathcal{C}/\mathbb{K}$ auquel est ajouté le point à l'infini $\mathcal{P}_\infty$ vérifie les propriétés listées dans la section sur les courbes elliptiques : fermeture, associativité, existence de l'élément neutre $\mathcal{P}_\infty$, existence des points opposés et commutativité.

Dans le cadre de cette thèse, nous nous intéresserons principalement au cas des courbes hyperelliptiques *de genre 2* définies sur *des corps finis premiers* $\mathrm{GF}(P)$ de grande caractéristique et utilisées dans les solutions récentes de l'état de l'art pour HECC. On peut trouver des exemples de telles courbes dans la littérature récente, comme par exemple la courbe $\mathcal{C}_{11,-22,-19,-3}$ publiée par Gaudry et Schost en 2012 dans [GS12] ou encore les courbes définies sur $\mathrm{GF}(2^{127}-1)$ ou $\mathrm{GF}(2^{128}-173)$ proposées en 2016 par Bos et coll. dans [BCHL16]. Pour des questions de lisibilité, nous utiliserons dans la suite du manuscrit la notation $\mathcal{C}$ pour désigner une courbe hyperelliptique définie sur $\mathrm{GF}(P)$ et la notation $\mathcal{J}_\mathcal{C}$ pour la Jacobienne de cette courbe.

Représentation de Mumford des points de $\mathcal{J}_\mathcal{C}$

Avant de nous intéresser au détail de l'opération d'addition de diviseurs dans $\mathcal{J}_\mathcal{C}$, et au cas particulier du doublement de diviseur, nous allons rappeler rapidement la *représentation de Mumford* des éléments de $\mathcal{J}_\mathcal{C}$. La notation de Mumford permet de représenter chaque diviseur $\mathcal{P}$ de $\mathcal{J}_\mathcal{C}$, qui correspond à une paire de points $(\mathcal{P}_1, \mathcal{P}_2)$ de $\mathcal{C}$ en utilisant une paire de polynômes $(u(x), v(x))$ à coefficients dans $\mathrm{GF}(P)$. Le polynôme $u(x) = x^2 + u_1 x + u_0$ est défini de façon à ce que $u(x_1) = u(x_2) = 0$ si x_1 et x_2 sont les coordonnées x respectives de $\mathcal{P}_1$ et de $\mathcal{P}_2$. Le polynôme $v(x) = v_1 x + v_0$ correspond lui à l'équation de la droite définie sur $\mathrm{GF}(P)$ et passant par les points de $\mathcal{P}_1$ et $\mathcal{P}_2$ de $\mathcal{C}$. Les diviseurs sur $\mathcal{J}_\mathcal{C}$ sont alors représentés de façon unique par 4 coordonnées (u_1, u_0, v_1, v_0) correspondant aux coefficients de $u(x)$ et de $v(x)$.

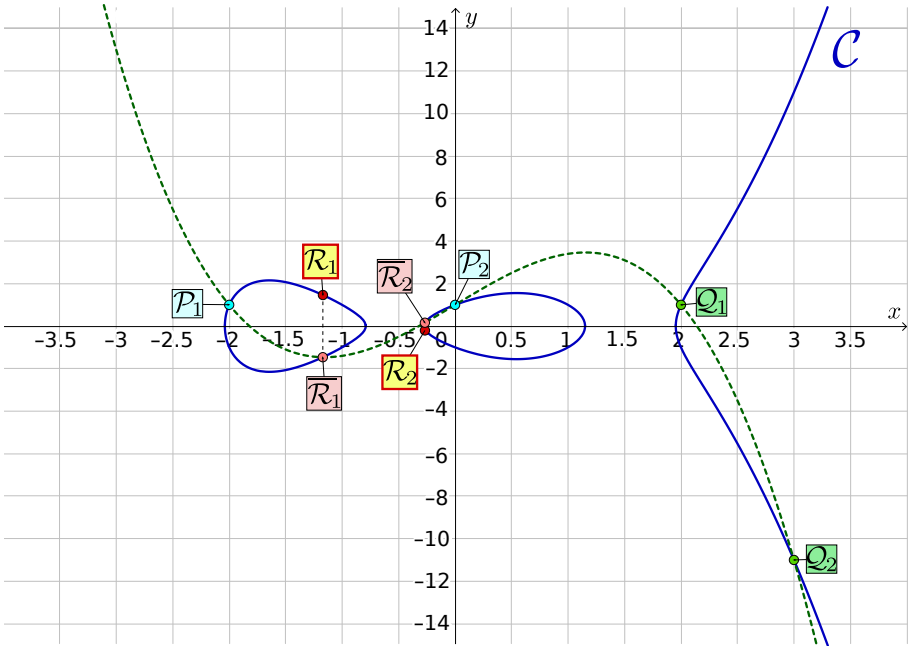


FIGURE 2.4 – Addition des points $\mathcal{P} + \mathcal{Q} = \mathcal{R}$ de la Jacobienne $\mathcal{J}_\mathcal{C}$ de la courbe hyperelliptique $\mathcal{C}$ de genre 2, définie sur les réels par l'équation 2.7. Les diviseurs $\mathcal{P}$, $\mathcal{Q}$ et $\mathcal{R}$ sur $\mathcal{J}_\mathcal{C}$ sont respectivement représentés par les couples de points $(\mathcal{P}_1, \mathcal{P}_2)$, $(\mathcal{Q}_1, \mathcal{Q}_2)$ et $(\mathcal{R}_1, \mathcal{R}_2)$ sur $\mathcal{C}$. Le diviseur $\overline{\mathcal{R}} = (\overline{\mathcal{R}}_1, \overline{\mathcal{R}}_2)$ est l'opposé de $\mathcal{R}$ sur $\mathcal{J}_\mathcal{C}$.

Taille des éléments de corps fini : HECC *versus* ECC

La représentation des diviseurs de la Jacobienne d'une courbe hyperelliptique de genre $g \geq 2$ se fait dans HECC grâce à 4 coordonnées, là où seulement 2 ou 3 coordonnées sont nécessaires dans ECC en fonction du système de coordonnées utilisé. Du fait de la structure complexe des courbes hyperelliptiques, les opérations au niveau courbe sont plus coûteuses à effectuer pour HECC que pour ECC. Il suffit pour s'en rendre compte de comparer les nombres d'opérations dans GF(P) nécessaires au calcul de `ADD` et de `DBL` dans ECC (cf. tableau 2.1) et dans HECC (cf. tableau 2.2).

Toutefois, la taille des éléments dans GF(P) est divisé par 2 dans HECC par rapport à ECC pour le même niveau de sécurité théorique. Ainsi, pour des cryptosystèmes HECC équivalents en terme de sécurité aux standards ECC 256 bits actuels, nous utiliserons des courbes hyperelliptiques définies sur des corps finis de grandes caractéristiques premières d'environ 128 bits. Pour ce niveau de sécurité, la taille des scalaires k utilisés dans HECC reste, elle, de 256 bits comme pour ECC.

2.4.2 Addition, doublement et multiplication scalaire dans $\mathcal{J}_\mathcal{C}$

L'opération d'addition `ADD` de diviseurs sur la Jacobienne $\mathcal{J}_\mathcal{C}$ d'une courbe hyperelliptique $\mathcal{C}$ de genre 2 définie sur les réels par l'équation 2.7 est illustrée en figure 2.4. Elle fonctionne de façon similaire à la méthode des cordes et des sécantes utilisée pour les courbes elliptiques mais en utilisant des équations polynômiales de degré 3 au lieu d'équations de droites. Dans cet exemple, tiré de [Sch15], on souhaite additionner les diviseurs $\mathcal{P}$ et $\mathcal{Q}$ de $\mathcal{J}_\mathcal{C}$, représentés respectivement par les couples de points $(\mathcal{P}_1, \mathcal{P}_2)$ et $(\mathcal{Q}_1, \mathcal{Q}_2)$ de $\mathcal{C}$, de coordonnées affines $\mathcal{P}_1 = (-2, 1)$, $\mathcal{P}_2 = (0, 1)$, $\mathcal{Q}_1 = (2, 1)$ et $\mathcal{Q}_2 = (3, -11)$.

$$y = -\frac{4}{5}x^3 + \frac{16}{5}x + 1. \tag{2.8}$$

L'unique fonction f de degré 3 passant par les points $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{Q}_1$ et $\mathcal{Q}_2$ est définie par l'équation 2.8 et tracée en pointillés verts en figure 2.4. Par propriété des courbes hyperelliptiques de genre 2, la courbe de f coupe $\mathcal{C}$ en deux points $\overline{\mathcal{R}}_1$ et $\overline{\mathcal{R}}_2$ dont les abscisses sont les solutions de l'équation :

$$\left(-\frac{4}{5}x^3 + \frac{16}{5}x + 1\right)^2 = x^5 - 5x^3 + 4x^2 + 1.$$

D'après l'exemple de [Sch15], les coordonnées affines de ces points représentant le diviseur $\overline{\mathcal{R}}$ de $\mathcal{J}_\mathcal{C}$ sont :

$$\overline{\mathcal{R}}_1 = \left(\frac{-23 - \sqrt{209}}{32}, -\frac{1333 + 115\sqrt{209}}{2048}\right) \quad \text{et} \quad \overline{\mathcal{R}}_2 = \left(\frac{-23 + \sqrt{209}}{32}, -\frac{1333 - 115\sqrt{209}}{2048}\right).$$

Le diviseur $\overline{\mathcal{R}}$ est l'opposé sur la Jacobienne $\mathcal{J}_\mathcal{C}$ du diviseur $\mathcal{R}$, qui est le résultat de la somme des points $\mathcal{P}$ et $\mathcal{Q}$ et correspond au couple de points de $\mathcal{C}$:

$$\mathcal{R}_1 = \left(\frac{-23 - \sqrt{209}}{32}, \frac{1333 + 115\sqrt{209}}{2048}\right) \quad \text{et} \quad \mathcal{R}_2 = \left(\frac{-23 + \sqrt{209}}{32}, \frac{1333 - 115\sqrt{209}}{2048}\right).$$

Nous n'avons pas illustré le fonctionnement de l'opération de doublement de point `DBL` dans $\mathcal{J}_\mathcal{C}$, qui correspond simplement au cas où la courbe tracée en pointillés verts est construite de façon à couper la courbe $\mathcal{C}$ en seulement deux couples de points $(\mathcal{P}_1, \mathcal{P}_2)$ et $(\overline{\mathcal{R}}_1, \overline{\mathcal{R}}_2)$.

À partir de maintenant et pour le reste du manuscrit, nous appellerons simplement *points* les *diviseurs* sur la Jacobienne $\mathcal{J}_\mathcal{C}$.

Formules d'additions et de doublement de points dans $\mathcal{J}_\mathcal{C}$

Le premier algorithme d'addition de points dans la Jacobienne d'une courbe hyperelliptique de genre $g \geq 2$ a été proposé par Cantor en 1987 dans [Can87]. L'algorithme original de Cantor ne fonctionnait que pour des courbes définies sur des corps finis de caractéristique impaire. Il a été généralisé pour les corps de caractéristique paire par Koblitz en 1989 dans [Kob89]. On notera qu'un algorithme d'addition plus performant a été proposé par Harley en 2000 dans [Har00] à partir des travaux de Cantor.

En 2000, Gaudry et Harley ont proposé dans [GH00] d'exprimer l'algorithme de Harley sous forme de *formules arithmétiques explicites*. Ces formules explicites permettent de réduire le nombre d'opérations au niveaux corps impliquées dans le calcul de l'addition de points sur $\mathcal{J}_\mathcal{C}$. Elles sont issues d'une analyse poussée de l'algorithme générique de Cantor pour différents choix de paramètres de courbe et de corps.

Différents ensembles de formules explicites pour l'addition `ADD` ou le doublement `DBL` de points ont été proposés dans la littérature pour différents types de courbes hyperelliptiques. En particulier, on pourra citer les travaux de Wollinger et coll. publiés dans [WPP05] en 2005 et qui proposent des formules explicites efficaces pour des courbes hyperelliptiques de genre 2 et 3. Les formules les plus efficaces ont été implantées par les auteurs sur microprocesseurs Pentium et ARM pour des courbes de genre 1, 2 et 3 présentant le même niveau de sécurité théorique. Les temps obtenus dans [WPP05] pour une multiplication scalaire montrent que, pour les courbes et les niveaux de sécurité choisis, HECC peut être aussi performant que ECC.

Toujours en 2005, Lange a aussi publié dans [Lan05] un ensemble de formules explicites issues de l'algorithme de Cantor pour différents paramètres de courbes de genre 2 et de corps. Celles-ci dépendent de la caractéristique du corps (paire ou impaire), des valeurs des coefficients du polynôme $u(x)$, ou encore du degré des polynômes $u(x)$ des points de $\mathcal{J}_\mathcal{C}$ additionnés. Les formules explicites sont aussi adaptées pour utiliser différents systèmes de coordonnées, voir [Lan02], parmi lesquelles les coordonnées :

— *affines* (A) : un point $\mathcal{P}$ de $\mathcal{J}_\mathcal{C}$ est représenté par les coordonnées (u_1, u_0, v_1, v_0) ;
— *projectives* (P) : un point $\mathcal{P}$ de $\mathcal{J}_\mathcal{C}$ est représenté par les coordonnées (U_1, U_0, V_1, V_0, Z) définies de telle sorte que $u_i = U_i/Z$ et $v_i = V_i/Z$ pour $i \in \{0, 1\}$ et $Z \neq 0$.

Les différents systèmes de coordonnées projectives utilisées par Lange sont décrits de façon plus complète dans [Lan02] et [Lan05]. Dans le tableau 2.2 tiré de [Lan05], nous indiquons les nombres d'opérations modulaires dans GF(P) (inversions `I`, multiplications `M` et carrés `S`) nécessaires aux calculs des opérations `ADD` et `DBL` sur une courbe hyperelliptique de genre 2 définie sur un corps fini GF(P) de caractéristique impaire pour les systèmes de coordonnées listés.

coordonnées	DBL		ADD	
affines	A → A	1I + 22M + 5S	A + A → A	1I + 22M + 3S
projectives	P → P	38M + 6S	P + P → P	47M + 4S
mixtes	P → P	38M + 6S	P + A → P	40M + 3S

TABLE 2.2 – Nombre d'inversions (`I`), de multiplications (`M`) et de carrés (`S`) dans GF(P) pour les opérations `ADD` et `DBL` sur la Jacobienne $\mathcal{J}_\mathcal{C}$ d'une courbe hyperelliptique de genre 2 en caractéristique impaire pour différents systèmes de coordonnées (A = affines, P = projectives) [Lan05].

Multiplication scalaire $[k]\mathcal{P}$ dans $\mathcal{J}_\mathcal{C}$

Le calcul de la multiplication scalaire dans HECC repose sur les mêmes principes que pour ECC. Il est donc possible d'utiliser les mêmes algorithmes dans lesquels il suffit de remplacer les opérations `ADD` et `DBL` sur la courbe elliptique $\mathcal{E}$ par celles adaptées à la Jacobienne $\mathcal{J}_\mathcal{C}$ de la courbe hyperelliptique $\mathcal{C}$ considérée.

Pour ce qui est de la robustesse des cryptosystèmes HECC face aux attaques par SPA, les remarques faites pour ECC sont ici aussi valables. Dans [Lan05], Lange fait aussi remarquer que les formules de `ADD` et de `DBL` sont très semblables en terme d'opérations dans GF(P). De ce fait, les formules explicites affines de [Lan05] permettent la mise en place simple de contremesures contre les attaques par SPA ou par analyse du temps de calcul des `ADD` et des `DBL`. L'étude de la résistance aux attaques par SCA des cryptosystèmes HECC utilisant les formules explicites de [Lan05] est détaillée dans [LM05] de Lange et Mishra, publié en 2005.

2.4.3 Multiplication scalaire sur la surface de Kummer $\mathcal{K}_\mathcal{C}$ de la Jacobienne $\mathcal{J}_\mathcal{C}$

Dans [SS99], publié en 1999, Smart et Siksek ont proposé un protocole d'échange de clé de type Diffie-Hellman basé sur des courbes hyperelliptiques de genre 2. Pour accélérer le calcul de la multiplication scalaire, les auteurs utilisent une *projection particulière* des points de la Jacobienne $\mathcal{J}_\mathcal{C}$, définie sur GF(P), sur *la surface de Kummer* $\mathcal{K}_\mathcal{C}$ associée à $\mathcal{J}_\mathcal{C}$. Cette projection leur permet de diminuer la complexité de la multiplication scalaire en évitant notamment le calcul d'inversions modulaires dans GF(P) ou encore

l'utilisation de l'arithmétique sur les polynômes nécessaire dans l'algorithme de Cantor. [5]

Conformément aux notations utilisées dans l'article [RSSB16], sur lequel nous reviendrons plus tard, les points de la surface de Kummer $\mathcal{K}_\mathcal{C}$ sont dénotés avec le signe « $\pm$ ». Par exemple, $\pm\mathcal{P}$ désigne la projection sur $\mathcal{K}_\mathcal{C}$ du point $\mathcal{P}$ de la Jacobienne $\mathcal{J}_\mathcal{C}$. Un point $\pm\mathcal{P}$ sur $\mathcal{K}_\mathcal{C}$ est représenté par 4 coordonnées dans GF(P) notées $(x : y : z : t)$. La projection du point à l'infini $\mathcal{P}_\infty$ de $\mathcal{J}_\mathcal{C}$ sur $\mathcal{K}_\mathcal{C}$ est représenté par les coordonnées de GF(P) $(a : b : c : d)$ *constantes* correspondant à des paramètres de la courbe hyperelliptique $\mathcal{C}$.

Les points de la Jacobienne $\mathcal{J}_\mathcal{C}$ sont projetés sur la surface de Kummer $\mathcal{K}_\mathcal{C}$ de telle sorte que $\pm\mathcal{P} = \pm\overline{\mathcal{P}}$ si $\pm\mathcal{P}$ et $\pm\overline{\mathcal{P}}$ sont les projections respectives des points $\mathcal{P}$ et $\overline{\mathcal{P}} = -\mathcal{P}$ de $\mathcal{J}_\mathcal{C}$ sur $\mathcal{K}_\mathcal{C}$. L'ensemble des points sur $\mathcal{K}_\mathcal{C}$ ne possède *pas une structure de groupe*. Si on considère les points $\mathcal{P}$, $\mathcal{Q}$ et $\overline{\mathcal{Q}} = -\mathcal{Q}$ sur $\mathcal{J}_\mathcal{C}$, il est impossible de calculer $\pm(\mathcal{P}+\mathcal{Q})$ à partir de $\pm\mathcal{P}$ et de $\pm\mathcal{Q} = \pm\overline{\mathcal{Q}}$. En effet, s'il existait une loi d'addition « $\diamond$ » dans $\mathcal{K}_\mathcal{C}$, on aurait les relations suivantes :

$$\begin{aligned} \pm\mathcal{P} \diamond \pm\mathcal{Q} &= \pm(\mathcal{P}+\mathcal{Q}) \\ \pm\mathcal{P} \diamond \pm\overline{\mathcal{Q}} &= \pm(\mathcal{P}-\mathcal{Q}). \end{aligned} \tag{2.9}$$

Or il existe des points $\mathcal{P}$ et $\mathcal{Q}$ de $\mathcal{J}_\mathcal{C}$ pour lesquels $\pm(\mathcal{P}+\mathcal{Q}) \neq \pm(\mathcal{P}-\mathcal{Q})$. D'après l'équation 2.9, on aurait pour ces points la relation

$$\pm\mathcal{P} \diamond \pm\mathcal{Q} \neq \pm\mathcal{P} \diamond \pm\overline{\mathcal{Q}},$$

ce qui est impossible étant donné que $\pm\mathcal{Q} = \pm\overline{\mathcal{Q}}$ sur $\mathcal{K}_\mathcal{C}$.

En d'autres termes, il n'est pas possible de définir une opération « $\diamond$ » *stable* car le calcul de $\pm\mathcal{P} \diamond \pm\mathcal{Q}$ a deux résultats possibles : $\pm(\mathcal{P}+\mathcal{Q})$ et $\pm(\mathcal{P}-\mathcal{Q})$.

S'il est impossible de définir une opération d'addition sur les points de $\mathcal{K}_\mathcal{C}$, il est toutefois possible de définir les opérations de doublement de point $\texttt{DBL}(\pm\mathcal{P}) = [2](\pm\mathcal{P}) = \pm([2]\mathcal{P})$ et de multiplication scalaire $[k](\pm\mathcal{P}) = \pm([k]\mathcal{P})$ sur $\mathcal{K}_\mathcal{C}$.

Il est aussi possible de définir une opération de *pseudo-addition*, notée `xADD`, qui pour les trois opérandes $\pm\mathcal{P}$, $\pm\mathcal{Q}$ et $\pm(\mathcal{P}-\mathcal{Q})$ permet de calculer $\pm(\mathcal{P}+\mathcal{Q})$:

$$\texttt{xADD}(\pm\mathcal{P}, \pm\mathcal{Q}, \pm(\mathcal{P}-\mathcal{Q})) = \pm(\mathcal{P}+\mathcal{Q}).$$

Cette opération semble à première vue inutilisable en pratique pour des points $\pm\mathcal{P}$ et $\pm\mathcal{Q}$ quelconques. Le calcul de $\pm(\mathcal{P}+\mathcal{Q})$ grâce à `xADD` nécessite en effet de connaître le résultat de $\pm(\mathcal{P}-\mathcal{Q})$ qui, lui, ne peut être calculé dans le cas général que si l'on connait $\pm(\mathcal{P}+\mathcal{Q})$.

En 2007 dans [Gau07], Gaudry utilise l'opération `xADD` dans un algorithme de multiplication scalaire basé sur *l'échelle de Montgomery* et décrit dans l'algorithme 8. Comme nous l'avons vu en section 2.3.5, la différence $\mathcal{R}_1 - \mathcal{R}_0 = \mathcal{P}$ dans l'échelle de Montgomery de l'algorithme 8. On a alors $\pm(\mathcal{R}_1 - \mathcal{R}_0) = \pm\mathcal{P}$ et $\pm(\mathcal{R}_1 + \mathcal{R}_0) = \texttt{xADD}(\pm\mathcal{R}_0, \pm\mathcal{R}_1, \pm\mathcal{P})$.

En 2016, Renes et coll. ont proposé dans [RSSB16] des implantations logicielles de cryptosystèmes HECC utilisant les surfaces de Kummer étudiées par Gaudry dans [Gau07].

Ces implantations de Kummer-HECC (KHECC), nommées μKummer, ont été réalisées sur les petits microcontrôleurs ARM Cortex M0 et AVR ATmega. Elles ont permis à leurs auteurs de marquer de

5. On rappellera que l'article [SS99] a été publié en 1999, c'est à dire avant les premières propositions de formules explicites par Gaudry et Harley dans [GH00] pour les calculs sur $\mathcal{J}_\mathcal{C}$.

Opérandes : $k \in \mathbb{N}$ sur n_k bits, $\pm\mathcal{P} \in \mathcal{K}_\mathcal{C}$
Résultat : $\pm\mathcal{R}_0 = [k](\pm\mathcal{P}), \pm\mathcal{R}_0 \in \mathcal{K}_\mathcal{C}$
begin
1 $\pm\mathcal{R}_0 \leftarrow \pm\mathcal{P}$
2 $\pm\mathcal{R}_1 \leftarrow [2] \pm \mathcal{P}$
3 **for** $i = n_k - 2$ **downto** 0 **do**
4 **if** $k_i = 0$ **then**
5 $\pm\mathcal{R}_1 \leftarrow \texttt{xADD}(\pm\mathcal{R}_0, \pm\mathcal{R}_1, \pm\mathcal{P})$
6 $\pm\mathcal{R}_0 \leftarrow [2](\pm\mathcal{R}_0)$
7 **else if** $k_i = 1$ **then**
8 $\pm\mathcal{R}_0 \leftarrow \texttt{xADD}(\pm\mathcal{R}_0, \pm\mathcal{R}_1, \pm\mathcal{P})$
9 $\pm\mathcal{R}_1 \leftarrow [2](\pm\mathcal{R}_1)$
10 **return** $\pm\mathcal{R}_0$

Algorithme 8 : Multiplication scalaire $[k](\pm\mathcal{P})$ par l'échelle de Montgomery sur la surface de Kummer $\mathcal{K}_\mathcal{C}$ tel que présentée dans [Gau07].

nouveaux records de vitesse pour le calcul des primitives cryptographiques pour l'échange de clés Diffie-Hellman et la signature numérique sur processeurs embarqués : nombres de cycles d'horloge réduits de 27% et de 75% sur Cortex M0 pour l'échange de clé et la signature numérique respectivement ; et de 32% et 71% sur ATmega pour les mêmes primitives par rapport aux précédents records de l'état de l'art.

Dans leurs implantations, Renes et coll. utilisent *l'opération unifiée* introduite dans [BCHL16] pour le calcul des pseudo-additions `xADD` et des doublements de points dans l'algorithme de multiplication scalaire. Cette opération, notée `xDBLADD` pour *differential doubling and addition*, permet le calcul simultané des points $\pm\mathcal{R}_0$ et $\pm\mathcal{R}_1$ de $\mathcal{K}_\mathcal{C}$ à partir des points $\pm\mathcal{R}_0^{(\text{prev})}$ et $\pm\mathcal{R}_1^{(\text{prev})}$ calculés à l'itération précédente et du point $\pm\mathcal{P}$ de $\mathcal{K}_\mathcal{C}$: $(\pm\mathcal{R}_0, \pm\mathcal{R}_1) \leftarrow \texttt{xDBLADD}(\pm\mathcal{R}_0^{(\text{prev})}, \pm\mathcal{R}_1^{(\text{prev})}, \pm\mathcal{P})$ avec $\pm\mathcal{R}_0 = \texttt{xADD}(\pm\mathcal{R}_0^{(\text{prev})}, \pm\mathcal{R}_1^{(\text{prev})}, \pm\mathcal{P})$ et $\pm\mathcal{R}_1 = [2](\pm\mathcal{R}_1^{(\text{prev})})$.

Les implantations de μKummer présentées par Renes et coll. dans [RSSB16] nous ont servi de base pour la conception de nos accélérateurs matériels dédiés au calcul de la multiplication scalaire dans KHECC. Nous les étudierons plus en détail dans le chapitre 4.

Opérandes : $k \in \mathbb{N}$ sur n_k bits, $\pm\mathcal{P} \in \mathcal{K}_\mathcal{C}$
Résultat : $\pm\mathcal{R}_0 = [k](\pm\mathcal{P}), \pm\mathcal{R}_0 \in \mathcal{K}_\mathcal{C}$
begin
1 $\pm\mathcal{R}_0 \leftarrow \pm\mathcal{P}$
2 $\pm\mathcal{R}_1 \leftarrow [2] \pm \mathcal{P}$
3 **for** $i = n_k - 2$ **downto** 0 **do**
4 **if** $k_i = 0$ **then**
5 $(\pm\mathcal{R}_1, \pm\mathcal{R}_0) \leftarrow \texttt{xDBLADD}(\pm\mathcal{R}_0, \pm\mathcal{R}_1, \pm\mathcal{P})$
6 **else if** $k_i = 1$ **then**
7 $(\pm\mathcal{R}_0, \pm\mathcal{R}_1) \leftarrow \texttt{xDBLADD}(\pm\mathcal{R}_0, \pm\mathcal{R}_1, \pm\mathcal{P})$
8 **return** $\pm\mathcal{R}_0$

Algorithme 9 : Multiplication scalaire $[k](\pm\mathcal{P})$ par l'échelle de Montgomery sur la surface de Kummer $\mathcal{K}_\mathcal{C}$ utilisant l'opération unifiée `xDBLADD` présentée dans [BCHL16].

2.5 Exemples de protocoles cryptographiques dans (H)ECC

Les protocoles cryptographiques les plus répandus utilisant l'opération de multiplication scalaire de points de courbes elliptiques ou hyperelliptiques sont le protocole *d'échange de clé* de Diffie et Hellman (cf. [DH76]) sur courbes elliptiques (ECDH) et le protocole de *signature numérique* ECDSA standardisé par le NIST dans [KG13].

Le protocole d'échange de clé ECDH sur la surface de Kummer $\mathcal{K}_\mathcal{C}$ d'une courbe hyperelliptique $\mathcal{C}$ est présenté dans l'algorithme 10. Il repose sur le fait que la multiplication scalaire sur $\mathcal{K}_\mathcal{C}$ est une opération *commutative* :

$$[k_A]([k_B](\pm\mathcal{P})) = [k_B]([k_A](\pm\mathcal{P})) = [k_A \times k_B](\pm\mathcal{P})$$

La robustesse de ce protocole est assurée par ECDLP selon lequel il est impossible de retrouver k_A, k_B ou $(k_A \times k_B)$ à un coût réaliste à partir de $[k_A](\pm\mathcal{P})$, de $[k_B](\pm\mathcal{P})$ ou de $[k_A \times k_B](\pm\mathcal{P})$, pour des tailles n_k de k_A et k_B suffisamment grandes et un *générateur* $\pm\mathcal{P}$ choisi avec de bonnes propriétés.

Prérequis : La surface de Kummer $\mathcal{K}_\mathcal{C}$ de la courbe hyperelliptique $\mathcal{C}$ et une taille de scalaire n_k.
Résultat : Un secret $\pm\mathcal{R}$ est partagé entre Alice et Bob, qui sont les seuls à le connaître.
begin

1. Alice et Bob choisissent ensemble un point $\pm\mathcal{P}$ de $\mathcal{K}_\mathcal{C}$, appelé *générateur*.
2. Alice tire au hasard un nombre k_A dans $[2, 2^{n_k} - 1]$.
   Bob tire au hasard un nombre k_B dans $[2, 2^{n_k} - 1]$.
3. Alice calcule $\pm\mathcal{R}_A = [k_A](\pm\mathcal{P})$.
   Bob calcule $\pm\mathcal{R}_B = [k_B](\pm\mathcal{P})$.
4. Alice envoie $\pm\mathcal{R}_A$ à Bob.
   Bob envoie $\pm\mathcal{R}_B$ à Alice.
5. Alice calcule $[k_A](\pm\mathcal{R}_B) = [k_A \times k_B](\pm\mathcal{P}) = \pm\mathcal{R}$.
   Bob calcule $[k_B](\pm\mathcal{R}_A) = [k_B \times k_A](\pm\mathcal{P}) = \pm\mathcal{R}$.

Algorithme 10 : Échange de clé de type Diffie-Hellman [DH76] entre Alice et Bob sur courbe hyperelliptique utilisant la surface de Kummer $\mathcal{K}_\mathcal{C}$.

Le protocole d'échange de clé ECDH est utilisé dans les cas où un secret doit être partagé entre deux interlocuteurs. Par exemple, il permet de générer des clés secrètes pour la mise en place de communications chiffrées par un algorithme de chiffrement symétrique.

On notera que les attaques physiques de type DPA, nécessitant plusieurs exécutions de l'opération de multiplication scalaire pour un *scalaire donné*, n'ont que peu de sens en pratique dans le cadre de ECDH. En effet, les scalaires k_A et k_B utilisés dans ECDH étant tirés aléatoirement lors de chaque échange de clé, ils sont utilisés seulement dans 2 multiplications scalaires (sur des points différents de $\mathcal{K}_\mathcal{C}$).

Le protocole de signature numérique ECDSA de [KG13] permet d'assurer *l'authentification* et la *non répudiation* d'un message. Il permet aussi d'en assurer *l'intégrité*. L'authentification permet de s'assurer que le message transmis a bien été envoyé par le bon individu. La non répudiation permet de prouver l'occurrence d'un évènement (envoie de message par exemple) ainsi que l'identité de l'individu l'ayant provoqué. Enfin, l'intégrité permet de s'assurer que le message reçu n'a pas été modifié en cours de transmission.

Comme tout système de signature numérique (DSA), ECDSA repose sur l'utilisation de 2 algorithmes :

— l'algorithme de *signature* permettant de générer une signature S_M à partir d'un message M et d'une clé privée $k_{\texttt{priv}}$;

— l'algorithme de *vérification de signature* permettant à partir d'un message M, d'une clé publique $k_{\texttt{pub}}$ et d'une signature S_M de prouver ou d'invalider l'authenticité du message M.

Ces algorithmes sont décrits dans [HMV04] dans le cas de ECC et nous ne les rappellerons pas ici. On notera simplement que la génération de la signature dans ECDSA utilise *1 multiplication scalaire* impliquant un scalaire tiré aléatoirement et *1 multiplication modulaire* impliquant la clé privée de l'émetteur. La vérification de signature nécessite *2 multiplications scalaires* et *1 addition de points*, et ne fait intervenir que la clé publique de l'émetteur. L'utilisation dans KHECC des algorithmes de signature et de vérification de [HMV04] est illustrée dans l'algorithme 11.

Prérequis : La surface de Kummer $\mathcal{K}_\mathcal{C}$ et la Jacobienne $\mathcal{J}_\mathcal{C}$ de la courbe hyperelliptique $\mathcal{C}$, une taille de scalaire n_k et un message M.

Résultat : Bob peut authentifier le message M comme étant bien envoyé par Alice et en vérifier l'intégrité.

begin

1 Alice tire au hasard une clé privée $k_{\texttt{priv}}$ de n_k bits.
Elle s'en sert pour générer une clé publique $k_{\texttt{pub}}$ de n_k bits.
Alice diffuse sa clé publique $k_{\texttt{pub}}$ et garde sa clé privée $k_{\texttt{priv}}$ secrète.

2 Pour signer le message M, Alice utilise une fonction de *hachage cryptographique*[5] $\texttt{H}$ pour *hacher* le message M et en calculer *l'empreinte* $H_M = \texttt{H}(M)$.
Elle calcule ensuite la signature S_M à partir de H_M.

3 Alice envoie à Bob le message $(M|S_M)$ correspondant à la concaténation de M et de S_M.

4 Bob utilise la clé publique $k_{\texttt{pub}}$ d'Alice pour déchiffrer S_M et obtenir H_M.
Il calcule ensuite l'empreinte $H'_M = \texttt{H}(M)$ en utilisant la même fonction $\texttt{H}$ qu'Alice.
Si $H'_M = H_M$, alors le message M est authentique, intègre, et a bien été envoyé par Alice.

Algorithme 11 : Mise en place et vérification d'une signature numérique S_M pour l'envoi d'un message M entre Alice et Bob, d'après le protocole de [KG13] adapté pour KHECC.

Si le message $(M|S_M)$ est modifié pendant le transfert, l'empreinte H'_M calculée par Bob à partir de M sera différente de l'empreinte H_M obtenue par déchiffrement de S_M en utilisant la clé publique d'Alice.

Si le message $H_M = H'_M$, cela signifie que la clé privée utilisée pour le chiffrement du message S_M est bien celle associée à la clé publique d'Alice utilisée par Bob. [6] Il s'agit donc bien de la clé secrète d'Alice, que seule cette dernière possède. Cela permet de s'assurer que le message M a bien été envoyé par Alice.

Il existe d'autres protocoles de signature numérique dans la littérature, tels que le protocole utilisé par Renes et coll. dans [RSSB16] et inspiré du EdDSA de Bernstein et coll. [BDL+11]. Nous ne détaillerons pas ce dernier ici car son fonctionnement est similaire à celui de ECDSA présenté dans l'algorithme 11. On notera toutefois que l'algorithme de *signature* implanté dans [RSSB16] fait appel à une opération de multiplication scalaire sur la *surface de Kummer* $\mathcal{K}_\mathcal{C}$ d'une courbe hyperelliptique $\mathcal{C}$. Cette multiplication scalaire fait intervenir un scalaire k résultant de plusieurs appels à une fonction de hachage cryptographique sur la clé privée et le message en clair M (contrairement à la signature ECDSA ne

6. Pour rappel, il est considéré comme impossible en pratique d'obtenir la même empreinte à partir de 2 messages différents si la fonction de hachage est une fonction de hachage cryptographique suffisamment robuste (SHA-3 par exemple).

faisant pas intervenir la clé privée dans la multiplication scalaire). De ce fait, les implantations de cet algorithme sont possiblement attaquable par DPA. La *vérification* de signature fait quant à elle appel à 2 multiplications scalaires sur $\mathcal{K}_\mathcal{C}$ et à une addition de points sur la Jacobienne $\mathcal{J}_\mathcal{C}$ de la courbe $\mathcal{C}$. Comme pour ECDSA, la vérification n'implique pas l'utilisation de la clé privée.

3 Multiplieurs modulaires hyper-threadés (HTMM)

Sommaire

3.1 Introduction

La conception d'opérateurs arithmétiques efficaces pour les calculs dans des corps finis premiers GF(P) est primordiale pour les implantations matérielles de cryptosystèmes asymétriques performants. En particulier, la multiplication modulaire dans GF(P) représente l'opération la plus courante et la plus complexe dans ces cryptosystèmes. Son impact est donc très important, aussi bien sur temps de calcul que sur la consommation de ressources et d'énergie dans les circuits.

Dans ce chapitre, nous proposerons une nouvelle architecture de multiplieurs modulaires pour le calcul de multiplications modulo des premiers P *génériques*, c.-à-d. quelconques. Cette architecture est conçue pour optimiser l'utilisation des multiplieurs–accumulateurs câblés, ou *slices* DSP, présents dans les FPGA contemporains.

L'algorithme de multiplication modulaire de Montgomery (MMM), proposé en 1985 par Montgomery dans [Mon85], est l'un des plus efficaces dans l'état de l'art de la multiplication modulo des premiers génériques.

Dans nos multiplieurs modulaires nous utilisons la variante itérative CIOS (pour *coarsely integrated operands scanning*) de cet algorithme proposée en 1996 dans [KAK96]. En raison de sa régularité et de sa simplicité, l'algorithme CIOS peut être facilement implanté dans les *slices* DSP des FPGA. Toutefois, les fortes dépendances entre les itérations des boucles internes du CIOS provoquent l'apparition de « bulles » dans le pipeline interne des *slices* DSP. Ces « bulles » correspondent à des cycles pendant lesquels certains étages du pipeline ne calculent aucune opération utile. Elles entraînent une utilisation moins efficace des ressources du FPGA et des *slices* DSP en particulier.

La plupart des solutions proposées dans la littérature utilisent des algorithmes complexes et coûteux pour relâcher ces dépendances de données et améliorer les performances de calcul dans les multiplieurs implantés. Du fait de la complexité de ces algorithmes, les multiplieurs modulaires rapides correspondant consomment souvent une quantité importante des ressources matérielles des FPGA. Ainsi, on peut trouver dans la littérature des exemples d'implantations FPGA de multiplieurs modulaires au sein de cryptosystèmes ECC utilisant plusieurs centaines de *slices* DSP (voir par exemple [AR14]). Nous estimons qu'une telle consommation de ressources n'est pas raisonnable pour des implantations matérielles de cryptosystèmes embarqués.

Nous avons donc décidé de proposer une solution différente basée sur des optimisations *architecturales* plutôt que algorithmiques. Celles-ci nous permettent de réduire le nombre de « bulles » dans le pipeline de nos unités grâce au calcul d'autres multiplications modulaires indépendantes. Dans nos multiplieurs, les étages de pipeline inutilisés dans les *slices* DSP servent au calcul d'autres opérations pour d'autres multiplications modulaires. Ce principe est connu sous le nom *d'hyper-threading* et a été proposé entre autres pour améliorer les performances de calcul dans les processeurs généralistes (voir [KM03] par exemple).

Nos *multiplieurs modulaires hyper-threadés* (HTMM) sont à notre connaissance les premiers à utiliser *l'hyper-threading* pour optimiser l'utilisation des ressources matérielles pour le calcul de la MMM. Nous verrons dans la suite de ce chapitre qu'ils permettent d'atteindre des performances supérieures à celles des meilleurs multiplieurs modulaires génériques de l'état de l'art pour des surfaces divisées par 2 ou 3 (pour des tailles de 128 ou 256 bits par exemple).

Dans un premier temps, nous rappellerons brièvement en section 3.2 quelques informations essentielles sur les FPGA utilisés durant la thèse pour nos implantations. Nous présenterons ensuite en section 3.3

l'état de l'art des algorithmes et des implantations matérielles sur FPGA de la multiplication modulaire pour des premiers GF(P) génériques. Nous nous intéresserons particulièrement à l'algorithme de multiplication modulaire de Montgomery ainsi qu'à certaines de ses variantes les plus intéressantes. Dans la section 3.4, nous décrirons comment nous avons utilisé le principe de *l'hyper-threading* pour concevoir l'architecture de notre HTMM. La section 3.5 présentera le détail des premières implantations de HTMM 128 bits que nous avons proposées dans [GT17d] en 2017 à la conférence internationale *IEEE Asilomar Conference on Signals, Systems and Computers*. Nous décrirons en section 3.6 un ensemble d'améliorations et d'optimisations du HTMM que nous avons soumis en juin 2018 dans [GT18a] au journal *IEEE Transactions on Computers*. Dans la section 3.7, nous présenterons un outil logiciel que nous avons développé pour générer automatiquement les codes sources VHDL de HTMM avec différentes spécifications de paramètres internes. Nous détaillerons en section 3.8 les meilleurs résultats d'implantation sur 4 FPGA de Xilinx de différents HTMM 128 bits et 256 bits spécifiés et implantés grâce à notre outil et présentés dans [GT18a]. Nous comparerons ces HTMM avec les meilleurs multiplieurs de l'état de l'art pour ces tailles de premiers. Finalement, la section 3.9 conclura ce chapitre et nous fournirons dans l'annexe 3.10 l'ensemble des résultats d'implantation pour nos HTMM ainsi que quelques figures illustrant la façon dont nous avons sélectionné nos meilleures spécifications de HTMM.

3.2 Rappels sur les FPGA

Les circuits logiques programmables FPGA (pour *field-programmable gate array* en anglais) sont des circuits intégrés composés d'une matrice de blocs matériels câblés. Ces blocs matériels et les connections entre ces derniers peuvent être configurés pour implanter différents circuits matériels. Les FPGA offrent un bon compromis entre les implantations de circuits ASIC (*application-specific integrated circuits* en anglais), performants mais dédiés à une unique application, et les implantations logicielles beaucoup plus flexibles mais en général bien moins performantes. Ces FPGA permettent non seulement le prototypage, la validation et l'évaluation rapide de circuits mais aussi la modification et la correction de circuits sans nécessiter la refonte de nouveaux circuits (comme c'est le cas pour les ASIC).

Dans le cadre de la thèse, nous avons utilisé pour nos implantations différents FPGA du constructeur Xilinx :

— le petit FPGA Spartan-6 comme « cible » *low-cost* et pour de futures évaluations de robustesse face aux attaques physiques par SCA sur la carte SAKURA [sak13] ;
— les FPGA Virtex-4 et Virtex-5 pour les comparaisons avec l'état de l'art ;
— le FPGA Virtex-7 récent pour les comparaisons avec les implantations récentes de la littérature.

Nos implantations FPGA ont été réalisées par l'intermédiaire des outils de CAO de Xilinx : ISE 14.7 et SmartXplorer pour la synthèse et le placement–routage des circuits décrits en VHDL, et iSim 14.7 pour la simulation et la validation de ces derniers. En raison du choix des FPGA Spartan-6 et Virtex-4 et 5, nous n'avons pas pu utiliser le dernier logiciel de CAO de Xilinx, Vivado, qui ne prend pas en charge les FPGA antérieurs à la 7^{e} génération (Virtex-7 par exemple).

3.2.1 Structure d'un FPGA Xilinx

Les FPGA de Xilinx sont construits autour d'une matrice de blocs matériels câblés *hétérogènes*. Parmi ceux-ci, on retrouve les blocs logiques configurables (CLB), les *slices* DSP et les blocs mémoire BRAM. En figure 3.1, nous illustrons l'organisation de ces différents blocs matériels au sein d'une région d'un

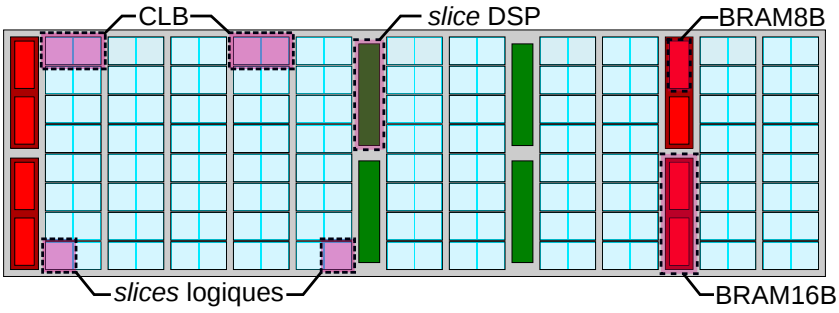


FIGURE 3.1 – Illustration schématique de l'organisation des blocs matériels câblés dans une des régions d'un FPGA Spartan-6 XC6SLX75 d'après le *floorplan* obtenu dans l'outil PlanAhead 14.7 de Xilinx. Le routage entre les différents blocs n'est pas représenté.

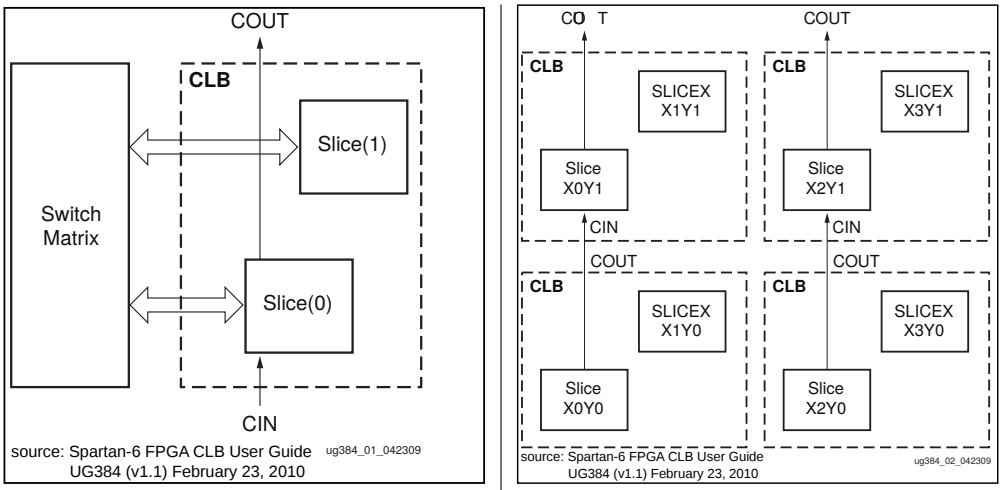


FIGURE 3.2 – Composition (a) et organisation en lignes et colonnes (b) des CLB dans un FPGA Spartan-6. Les illustrations sont tirées de la documentation officielle de Xilinx [Xil10].

FPGA Spartan-6 XC6SLX75. Dans cette figure, les rectangles verts représentent les *slices DSP*, c'est à dire des petits blocs câblés dédiés au calcul de multiplications–accumulations. Les rectangles bleus représentent les CLB, composés dans le Spartan-6 de *2 slices* que nous appellerons *slices logiques* dans la suite du document par opposition aux *slices* DSP. Enfin, les BRAM sont représentées en rouge.

3.2.2 Les blocs logiques configurables CLB

Dans les FPGA Xilinx, les CLB sont constitués d'un ensemble de *slices* logiques. La composition d'un CLB dans le Spartan-6 est illustrée en figure 3.2 (a), provenant de la documentation fournie par Xilinx pour ce FPGA [Xil10]. On peut voir dans cette figure qu'un CLB du Spartan-6 est composé des 2 *slices* logiques connectées à une matrice de commutateurs (*Switch Matrix* en figure 3.2 (a)). Ces commutateurs permettent d'interconnecter les *slices* logiques de différents CLB.

Les *slices* logiques sont utilisées pour implanter des fonctions logiques. Elles sont composées de petites *tables de correspondance programmables* (LUT pour *lookup tables* en anglais) à 4, 5 ou 6 entrées (LUT4, LUT5 ou LUT6 respectivement), de bascules ou registres de 1 bit programmables (FF pour *flip-flops* en anglais). La figure 3.3 illustre pour l'exemple l'agencement des LUT et des FF dans une *slice* logique d'un

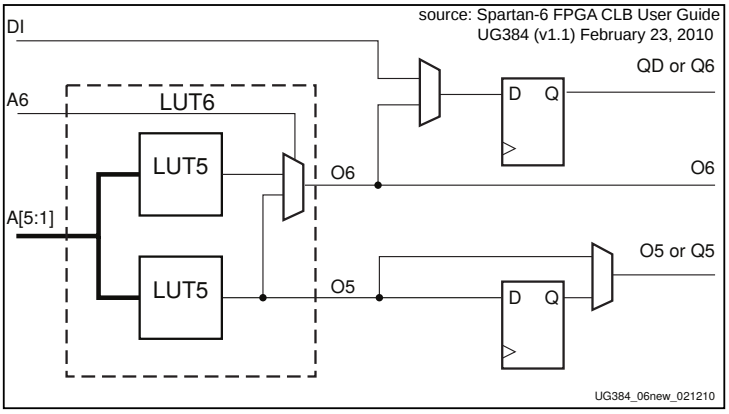


FIGURE 3.3 – Agencement de la LUT6, composée de 2 LUT5, et des FF dans une *slice* logique d'un Spartan-6. Les illustrations sont tirées de la documentation officielle de Xilinx [Xil10].

FPGA	techno. nm	slices logiques	LUT	FF	DRAM bits	SRL bits	document de référence
Virtex-4	90	4	4×2 LUT4	4×2	64	64	[Xil08a]
Virtex-5	65	2	2×4 LUT6	2×4	256	128	[Xil12a]
Spartan-6	45	2	2×4 LUT6	2×8	256	128	[Xil10]
Virtex-7	28	2	2×4 LUT6	2×8	256	128	[Xil16b]

TABLE 3.1 – Composition d'un bloc logique configurable CLB dans différents FPGA d'après la documentation officielle de Xilinx (les références sont indiquées dans le tableau).

Spartan-6. Les *slices* logiques intègrent aussi des *ressources de configuration* permettant de programmer leurs fonctions respectives. Certaines *slices* logiques intègrent aussi des dispositifs logiques câblés spécifiques permettant par exemple la propagation rapide des retenues dans des additionneurs–soustracteurs. Ce dispositif logique câblé peut être configuré pour permettre la propagation de retenues entre plusieurs CLB adjacents dans même colonne, comme illustré par les flèches verticales en figure 3.2 (b). Enfin, certaines *slices* logiques peuvent aussi être utilisées pour implanter de la mémoire RAM *distribuée* (DRAM) ou des registres à décalage (SRL).

La composition des CLB et des *slices* logiques dans un FPGA dépend de sa *famille* et de sa *génération*. Dans le tableau 3.1, nous rappelons la composition des CLB dans les FPGA utilisés dans la thèse. Ce tableau indique le nombre de *slices* logiques par CLB ainsi que la composition de ces derniers. Par exemple, les CLB d'un Virtex-4 intègrent 4 *slices* logiques composés de 2 LUT4 et de 2 FF. Ces *slices* logiques peuvent être configurées pour implanter des fonctions logiques, une mémoire DRAM d'au plus 64 bits ou un registre à décalage SRL d'au plus 64 bits.

En raison de ces différences, il est délicat de comparer les surfaces de circuits implantés sur des FPGA de différentes familles ou générations après placement et routage dans les outils de CAO. Par exemple, une LUT6 d'un Virtex-5 est équivalente à 4 LUT4 d'un Virtex-4. Comparer directement le nombre de LUT6 utilisées pour une implantation sur Virtex-5 avec le nombre de LUT4 utilisées pour une implantation sur Virtex-4 n'est donc pas possible.

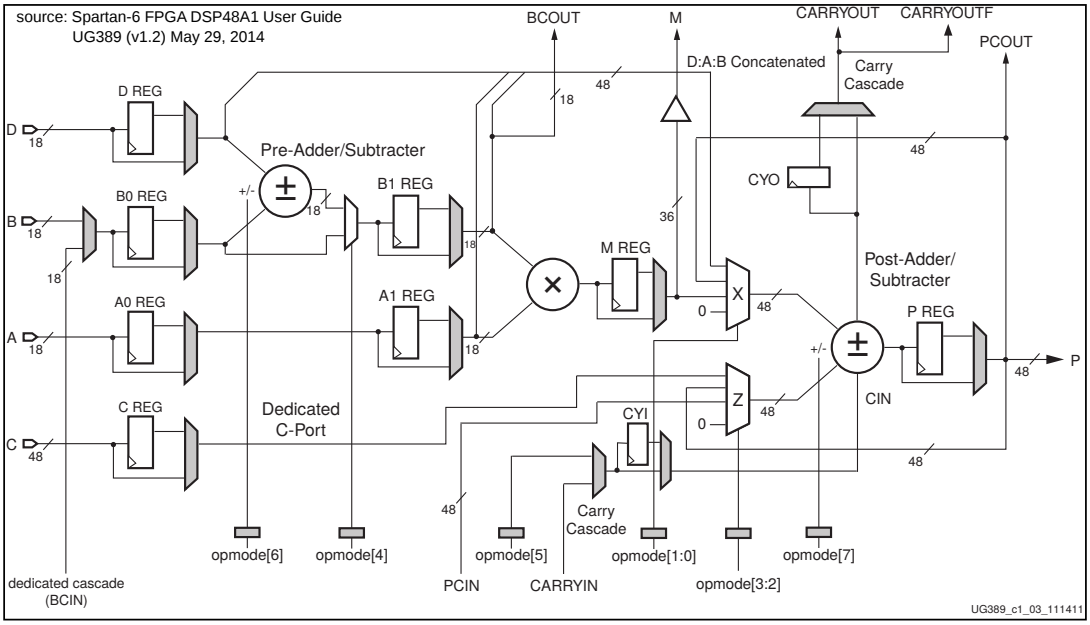


FIGURE 3.4 – Schéma d'un *slice* DSP dans un FPGA Spartan-6 (DSP48A1) d'après la documentation officielle de Xilinx [Xil11].

3.2.3 Les *slices* DSP et les BRAM

Les *slices* DSP (pour *digital signal processing* en anglais) sont des blocs matériels câblés embarquant, entre autres, un ensemble de registres et de multiplexeurs, un petit multiplieur et un additionneur–soustracteur. Ils permettent de calculer rapidement des opérations de type multiplication–accumulation sur des opérandes signés de quelques dizaines de bits : 18 bits $\times$ 18 bits $\pm$ 48 bits $\rightarrow$ 48 bits dans les Virtex-4 et les Spartan-6, ou 25 bits $\times$ 18 bits $\pm$ 48 bits $\rightarrow$ 48 bits dans les Virtex-5 et 7 par exemple. Les *slices* DSP des FPGA récents comme les Virtex-5 et 7 peuvent aussi être utilisés pour implanter des opérations plus complexes que nous ne détaillerons pas ici. Pour une description plus complète des structures et des fonctionnalités des *slices* DSP des FPGA utilisés dans cette thèse, nous renvoyons le lecteur vers la documentation fournie par Xilinx : [Xil08b] pour Virtex-4, [Xil12b] pour Virtex-5, [Xil11] pour Spartan-6 et [Xil18b] pour Virtex-7.

La figure 3.4 tirée de [Xil11] illustre la structure des *slices* DSP dans le Spartan-6. Ces *slices* DSP intègrent un pré-additionneur–soustracteur permettant d'additionner les entrées signées de 18 bits B et D, mais que n'utiliserons pas dans nos HTMM. Un petit multiplieur 18×18 bits $\rightarrow$ 48 bits permet de multiplier les entrées signées de 18 bits A et B. Enfin, un additionneur–soustracteur 48 ± 48 bits $\rightarrow$ 48 bits permet d'accumuler les résultats du multiplieur (c.-à-d. de calculer $P = A \times B \pm P$) ou d'additionner (ou de soustraire) ces derniers avec l'opérande signée de 48 bits C (c.-à-d. de calculer $P = A \times B \pm C$). Les opérandes des *slices* DSP des FPGA Xilinx sont représentés en complément à 2 et le bit de poids fort de ces opérandes est le bit de signe. Pour la multiplication–accumulation d'entiers positifs, les opérandes sont donc représentés sur 17 ou 24 bits seulement, au lieu de 18 ou 25 bits.

Les *slices* DSP peuvent être organisés *en cascade* de façon à construire des multiplieurs plus larges, comme illustré en figure 3.5. Dans ce cas, il est possible d'utiliser les ports d'entrée/sortie BCIN, PCIN, BCOUT et PCOUT pour connecter les différents *slices* DSP. Ces ports sont connectés par un routage câblé dédié, plus performant que le routage « classique » implanté entre les éléments logiques des FPGA.

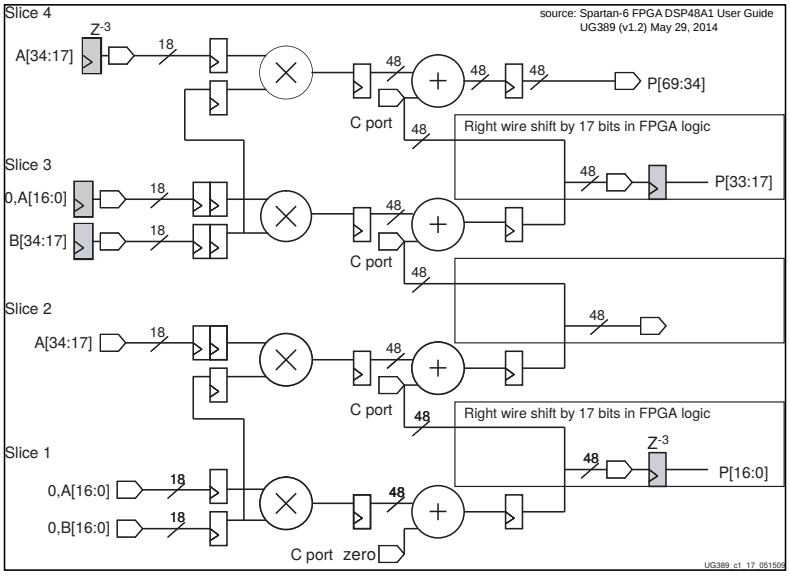


FIGURE 3.5 – Multiplieur 35×35 bits pipeliné à base de *slices* DSP organisés en cascade. Le schéma est tiré du document [Xil11] de Xilinx.

Pour pouvoir atteindre des fréquences élevées, les *slices* DSP sont construits autour d'un pipeline interne de 4 étages, comme illustré en figure 3.4 :

— le 1er étage correspond aux registres A0 et B0 respectivement placés après les entrées A et B ;
— le 2e étage correspond aux registres A1 et B1 respectivement placés après A0 et B0 ;
— le 3e étage correspond au registre M placé après le multiplieur ;
— le 4e étage correspond au registre de sortie P placé après l'additionneur–soustracteur.

Ce pipeline interne est *configurable* grâce à un ensemble de multiplexeurs *programmables à la configuration* du FPGA, indiqués en gris en figure 3.4. Les fréquences maximales atteignables dans les *slices* DSP des FPGA utilisés sont rapportées dans le tableau 3.2 d'après la documentation de Xilinx : [Xil09] pour Virtex-4, [Xil14b] pour Virtex-5, [Xil15] pour Spartan-6 et [Xil17] pour Virtex-7. Ces fréquences maximales sont données pour des *slices* DSP utilisant 4 étages de pipeline interne. Diminuer le nombre d'étages de pipeline permet de réduire la latence des *slices* DSP mais peut en contrepartie entraîner des chutes importantes de la fréquence dans ces derniers.

FPGA	type de *slice* DSP	référence du FPGA	*speed grade*	fréq. max. MHz	document de référence
Virtex-4	DSP48	XC4VLX100	-12	500	[Xil09] p.35
Virtex-5	DSP48E	XC5VLX110T	-3	550	[Xil14b] p.50
Spartan-6	DSP48A1	XC6SLX75	-3	390	[Xil15] p.52
Virtex-7	DSP48E1	XC7VX690T	-3	741	[Xil17] p.39

TABLE 3.2 – Fréquences maximales des *slices* DSP des différents FPGA utilisés pour nos implantations, issues de la documentation officielle de Xilinx.

Pour illustrer ce comportement, nous rapportons dans le tableau 3.3 les fréquences obtenues pour différentes configurations d'un *slice* DSP après placement–routage sur un Spartan-6 et un Virtex-7. Ces résultats montrent que la fréquence maximale peut être atteinte sur ces FPGA avec un pipeline interne

registres internes						lat.	fréq.
A0	B0	A1	B1	M	P	cc	MHz
√	√	√	√	√	√	4	390
	√	√		√	√	3	390
		√	√	√	√		390
√	√			√	√		349
√			√	√	√		349
	√	√			√	2	239
	√	√		√			219
				√	√		203
	√	√				1	238
				√			218
					√		180
						0	149

Spartan-6

registres internes						lat.	fréq.
A0	B0	A1	B1	M	P	cc	MHz
√	√	√	√	√	√	4	741
	√	√		√	√	3	741
		√	√	√	√		741
√	√			√	√		741
√			√	√	√		741
	√	√			√	2	412
	√	√		√			505
				√	√		534
	√	√				1	256
				√			501
					√		334
						0	278

Virtex-7

TABLE 3.3 – Impact de la configuration du pipeline des *slices* DSP d'un Spartan-6 et d'un Virtex-7 sur les latences (en cycles) et les fréquences maximales atteignables pour le calcul de l'opération $A \times B + C$. Les fréquences ont été obtenues après placement–routage dans ISE 14.7.

de 3 ou 4 étages dans les *slices* DSP. L'utilisation de moins de 3 étages de pipeline entraîne une chute de la fréquence dans les *slices* DSP d'au moins 38.7% sur Spartan-6 et d'au moins 27.9% sur Virtex-7. Sur Spartan-6, la fréquence la plus faible est atteinte quand aucun étage de pipeline n'est utilisé : $-61,8\%$ par rapport à la fréquence maximale de 390 MHz du *slice* DSP. Sur Virtex-7, la relation entre le nombre d'étages de pipeline et la fréquence du *slice* DSP est moins claire et dépend fortement des registres utilisés dans ce dernier. Par exemple, la configuration avec 1 étage de pipeline utilisant les registres A0 et B0 est légèrement moins performante que la configuration sans étages de pipeline. Les détails sur le fonctionnement des *slices* DSP fournis dans la documentation du Virtex-7 [Xil18b] et [Xil17] sont insuffisants pour expliquer ce comportement.

Les résultats du tableau 3.3 illustrent l'un des problèmes rencontrés lors de l'implantation de circuits sur FPGA, à savoir la *variation des performances* obtenues en fonction du FPGA choisi pour l'implantation. L'utilisation des registres B0, A1 et P permettra par exemple d'atteindre la meilleure fréquence dans le pipeline de 2 étages d'un *slice* DSP sur Spartan-6, mais fournira la moins bonne fréquence dans un *slice* DSP sur Virtex-7. Pour évaluer les performances d'un circuit sur différents FPGA, il est alors nécessaire d'implanter, de valider et d'évaluer ce circuit sur chacun des FPGA considérés. Cela entraîne des coûts de développement importants, notamment en termes de temps. Nous proposerons dans la section 3.7 de ce chapitre et dans la section 4.4.3 du chapitre 4 un ensemble d'outils logiciels que nous avons développé pour faciliter l'implantation et l'évaluation d'un grand nombre de circuits sur différents FPGA.

Les FPGA que nous avons utilisés pour nous implantations intègrent aussi des blocs mémoires câblés de type RAM. Ces mémoires BRAM sont souvent composées de 2 BRAM plus petites utilisables indépendamment. Par exemple, les BRAM du Spartan-6 peuvent être des BRAM de 9 Kb, notées BRAM8B en figure 3.1, ou des BRAM de 18 Kb, notées BRAM16B.

Quelles que soient leurs tailles, les mémoires BRAM possèdent 2 ports d'accès pouvant fonctionner selon *2 modes distincts* sélectionnés à la configuration. Le 1er mode est le mode *true dual-port* (TDP) dans lequel chacun des 2 ports peut être utilisé pour les accès mémoires *en lecture* et *en écriture*. Le 2e mode

FPGA	taille BRAM kbits	largeur (SDP) bits	largeur (TDP) bits	document de référence
Virtex-4	18	–	36	[Xil08a] p.142
	36	–	36	
Virtex-5	18	36	18	[Xil12a] p.122
	36	72	36	
Spartan-6	9	36	18	[Xil11] p.10
	18	36	36	
Virtex-7	18	36	18	[Xil16a] p.25
	36	72	36	

TABLE 3.4 – Tailles des BRAM et largeur maximale des mots mémoires utilisables dans les BRAM de différents FPGA de Xilinx pour les modes *simple dual-port* (SDP) et *true dual-port* (TDP).

est le mode *simple dual-port* (SDP) dans lequel l'un des 2 ports est utilisé uniquement pour les accès mémoires *en lecture* et l'autre est utilisé uniquement pour les accès mémoires *en écriture*.

Dans les BRAM, la taille des mots mémoires et la profondeur (c.-à-d. le nombre de mots mémoires pouvant être mémorisés) sont configurables. Dans le tableau 3.4 nous listons les tailles de BRAM et les largeurs maximales des mots mémoires utilisables dans les BRAM des Virtex-4, 5 et 7 et des Spartan-6. Les références vers la documentation complète des BRAM de Xilinx pour ces FPGA sont aussi précisées dans ce tableau.

3.3 Multiplication modulaire : principe et état de l'art

3.3.1 Algorithmes pour la réduction modulo P

Toutes les opérations dans GF(P) effectuées dans (H)ECC se font modulo P. De cette façon, le résultat d'une opération calculée pour des opérandes dans GF(P) est lui aussi dans GF(P).

L'opération de réduction modulo P permet la conversion d'un nombre entier quelconque vers les entiers $\{0, 1, 2, ..., P-1\}$:

$$\begin{aligned} \texttt{red\_mod} : \mathbb{N} &\rightarrow \{0, 1, 2, ..., P-1\} \\ x &\mapsto x \bmod P,\ 0 \leq (x \bmod P) < P. \end{aligned}$$

Cette opération de réduction est coûteuse car elle implique le calcul d'une division entière par P pour le calcul du reste de la division euclidienne :

$$x \bmod P = x - \left\lfloor \frac{x}{P} \right\rfloor \times P. \tag{3.1}$$

Dans le travail présenté ici, nous ne considérons pas la réduction modulaire par des premiers ayant une forme spécifique, tels les premiers de Mersenne $P = 2^n - 1$ ou les pseudo-Mersennes $P = 2^n - \epsilon$, $2^n \gg \epsilon$, qui peut être calculée de façon rapide par des enchaînements d'additions et de décalages. Ces premiers spécifiques permettent la mise en place d'implantations (H)ECC rapides (voir [GP08] par exemple), mais limitées à un seul corps fini.

Nos cryptoprocesseurs (H)ECC sur FPGA sont conçus pour être utilisés au sein de *systèmes embarqués*. Dans ces systèmes, la reconfiguration des FPGA est coûteuse en énergie et peut être compliquée à mettre

en place, par exemple à cause de la difficulté d'accéder aux dispositifs après leur commercialisation et leur déploiement. Pour des raisons de sécurité, il est de plus nécessaire de s'assurer de l'authenticité et de l'intégrité des *bitstreams* utilisés pour reconfigurer le FPGA. La mise en place des outils nécessaires à la sécurisation et à la vérification des *bitstreams* au niveau du FPGA nécessite d'implanter et d'utiliser des protocoles de cryptographie asymétriques, ce qui pose clairement un problème de poules et d'œufs.

Pour cette raison, nous avons décidé de concevoir et d'implanter des unités arithmétiques flexibles pouvant prendre en charge des nombres premiers *génériques* pour une taille fixée des entiers modulo P (128 ou 256 bits dans les exemples de HTMM proposés dans ce chapitre). L'utilisation de premiers spécifiques de Mersenne n'est donc pas envisageable dans nos travaux. On notera que de plus en plus d'implantations de la littérature suivent aussi ce principe et utilisent des premiers génériques.

Différents algorithmes et architectures matérielles ont été proposés dans la littérature pour le calcul de la multiplication modulaire pour des premiers génériques. Ils permettent pour la plupart d'optimiser le calcul de cette opération en entrelaçant les phases de multiplication des opérandes et de réduction modulaire du résultat. Certaines des solutions matérielles de l'état de l'art se basent sur l'algorithme de Blakley proposé en 1983 dans [Bla83] et dans lequel l'un des opérandes est découpé en sous-mots de 1 ou 2 bits. Chaque sous-mot est successivement multiplié par l'intégralité du deuxième opérande puis accumulé avec les résultats des produits partiels précédents. Chaque résultat intermédiaire est si nécessaire réduit modulo P après accumulation en soustrayant P si la valeur du résultat lui est supérieur. Parmi les implantations récentes utilisant l'algorithme de Blakley, on peut citer celles de Ghosh et coll. dans [GMC10] ou celles de Javeed et coll. dans [JWS15] et dans [JWS17]. Ces implantations sont particulièrement adaptées pour des cibles ASIC dans lesquelles l'utilisation de blocs multiplieurs optimisés (équivalents aux *slices* DSP sur FPGA) n'est pas possible. Un petit nombre d'autres solutions (p. ex. [KVV10] publié en 2010 par Knezevic et coll.) se basent sur l'algorithme de Barrett, proposé dans [Bar84] en 1984. Cette méthode de réduction modulaire consiste à remplacer la division par P dans l'équation 3.1 par un ensemble de décalages et une multiplication par le quotient pré-calculé (b^{2k}/P), avec $b > 3$ et $k = \lfloor \log_b P \rfloor$. Enfin, la très grande majorité des implantations logicielles et matérielles de la multiplication modulo des premiers génériques se base sur l'algorithme de multiplication modulaire de Montgomery (MMM) proposé en 1985 dans [Mon85] ou sur l'une de ses nombreuses variantes.

3.3.2 La multiplication modulaire de Montgomery

Le principe de la MMM telle que proposée par Montgomery dans [Mon85] est de remplacer la division par le premier P de n bits dans l'équation 3.1 par une division plus facile à calculer. En particulier, les divisions par des puissances de 2 sont triviales à calculer puisqu'elles consistent simplement en la suppression d'un certain nombre de bits de poids faible. La MMM de [Mon85] est rappelée dans l'algorithme 12. Dans cet algorithme, n désigne la taille en bits du premier P. Les opérandes A et B étant compris entre 0 et P, ils ont aussi une taille de n bits. Dans l'algorithme 12, l'entier m peut être choisi arbitrairement grand, tant qu'il vérifie l'inégalité $2^m > P$, et donc $m \geq n$. Pour optimiser le calcul de la MMM, on utilise généralement $m = n$ ou $m = n + 1$ dans les implantations de la MMM de [Mon85].

Le calcul d'une multiplication modulaire nécessite quelques additions et décalages logiques ainsi que trois multiplications :

1. calcul du produit $U = A \times B$ sur $2m$ bits ;
2. calcul du quotient de réduction $q = \left(U \times (-P^{-1})\right) \bmod 2^m$ sur m bits ;
3. calcul du produit $qP = q \times P$ sur $2m$ bits.

Opérandes : A et $B \in \mathbb{N}$ et P un premier de n bits avec $0 \leq A, B < P$
Prérequis : $m \in \mathbb{N}$ tel que $2^m > P$, et $P' = -P^{-1} \bmod 2^m$
Résultat : $T \equiv (AB \times 2^{-m}) \bmod P$, $0 \leq T < P$
begin

```
1  U ← A × B
2  q ← (U × P′) mod 2^m
3  T ← (q × P + U) /2^m
4  if T ≥ P then
5      T ← T − P
6  return T
```

Algorithme 12 : Multiplication modulaire de Montgomery [Mon85].

L'addition de qP au résultat U du produit $A \times B$ permet à la somme $S = U + qP$ (avant décalage) d'être divisible par 2^m tout en vérifiant $S \bmod P = A \times B \bmod P$.

On notera que le résultat de la MMM est $T = (A \times B \times 2^{-m}) \bmod P$, et non $(A \times B) \bmod P$ suite à la division de S par 2^m. Pour empêcher la propagation du facteur 2^{-m} dans des produits successifs, Montgomery utilise une représentation particulière des opérandes correspondant à leur projection dans le *domaine de Montgomery*. La projection d'un opérande $\{0, 1, 2, ..., P-1\}$ vers le domaine de Montgomery (MD) est définie comme suit :

$$\begin{aligned} \text{proj}_{\text{MD}} : \{0, 1, 2, ..., P-1\} &\to \text{MD} \\ x &\mapsto \tilde{x} = (x \times 2^m) \bmod P. \end{aligned}$$

Le résultat de la MMM pour les opérandes $\tilde{A} = (A \times 2^m) \bmod P$ et $\tilde{B} = (B \times 2^m) \bmod P$ est alors

$$\begin{aligned} \widetilde{T} &= (\tilde{A} \times \tilde{B} \times 2^{-m}) \bmod P \\ &= (A \times 2^m \times B \times 2^m \times 2^{-m}) \bmod P \\ &= (AB \times 2^m) \bmod P \\ &= \widetilde{AB} \end{aligned}$$

et est donc toujours dans le domaine de Montgomery, ce qui est important pour pouvoir enchaîner les calculs de la MMM. On notera aussi que le résultat de l'addition ou la soustraction modulaire de deux nombres dans MD est aussi dans MD. La conversion des opérandes et des résultats vers et depuis le domaine de Montgomery peut donc être effectuée une unique fois en début et une unique fois en fin de toute séquence d'opérations modulaires. De ce fait, l'algorithme de MMM est à privilégier dans le cas où de nombreuses opérations modulaires successives doivent être calculées afin de limiter le coût de la conversion des données. Dans la suite de ce document, nous considèrerons que les opérandes de la MMM ont été au préalable projetés dans MD (au prix de quelques multiplications modulaires pouvant pour la plupart être pré-calculées dans HECC) ou sont le résultat de multiplications modulaires intermédiaires.

3.3.3 Variantes et implantations de la MMM dans la littérature

L'une des premières optimisations de la MMM a été la suppression de la soustraction finale dans la MMM de [Mon85] dans le cas où $T \geq P$, proposée par Walter dans [Wal99a] en 1999.

En effet, ce dernier a démontré qu'en élargissant le domaine de Montgomery dans l'algorithme de

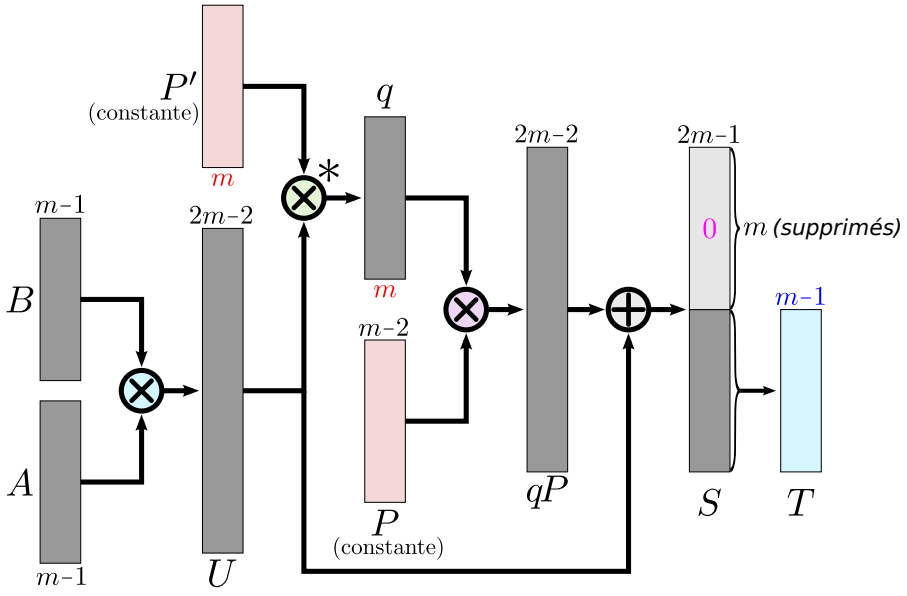


FIGURE 3.6 – Illustration des opérations dans la MMM [Mon85] optimisée par Walter [Wal99a] (sans soustraction finale) pour la valeur minimale de m, à savoir $m = n + 2$. La multiplication marquée « * » est calculée modulo 2^m (seuls les m LSB de U sont utilisés).

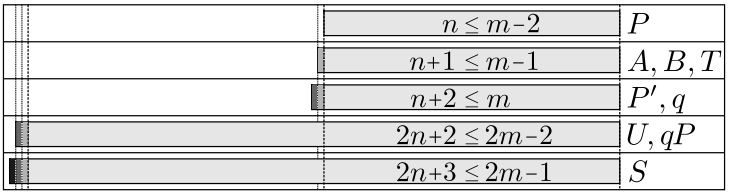


FIGURE 3.7 – Taille des valeurs manipulées dans la MMM de [Mon85] optimisée par Walter dans [Wal99a] (sans soustraction finale).

MMM en choisissant $2^m > 4P$ au lieu de $2^m > P$, le produit modulo P de deux opérandes A et B compris entre 0 et $2P$ pouvait lui aussi être borné entre 0 et $2P$. Le calcul de MMM successives peut donc être effectué pour des opérandes et résultats intermédiaires entre 0 et $2P$, la soustraction finale par P n'étant calculée qu'une unique fois si nécessaire en fin de séquence d'opérations pour ramener le résultat final entre 0 et P.

À la suite de l'extension du domaine de Montgomery, le quotient de réduction q est calculé modulo 2^m, avec $m \geq n + 2$ (n correspond à la taille de P en bits). Les produits intermédiaires U et qP sont sur $2m-2$ bits et la somme S de ces valeurs est donc sur $2m-1$ bits. Le résultat T de la MMM est quant à lui toujours compris entre 0 et $2P$ par construction, étant donné que $T = \lfloor S/2^m \rfloor$. Les opérations effectuées dans la MMM optimisée par Walter sont illustrés à la figure 3.6 et les tailles des valeurs manipulées ainsi que la relation entre m et n sont illustrées en figure 3.7.

En 1996, Koç et coll. ont implanté en logiciel et comparé dans [KAK96] cinq variantes de l'algorithme de MMM, parmi lesquelles trois variantes originales et deux issues de travaux antérieurs :

— *Separated Operand Scanning* (SOS) de [DK90] ;
— *Coarsely Integrated Operand Scanning* (CIOS) ;
— *Finely Integrated Operand Scanning* (FIOS) ;

— *Finely Integrated Product Scanning* (FIPS) de [Kal93] ;
— *Coarsely Integrated Hybrid Scanning* (CIHS).

On notera qu'aucun de ces cinq algorithmes n'intègre nativement l'optimisation de Walter, proposée 3 ans plus tard, mais que cette dernière est utilisable dans chacune des variantes.

La variante SOS de la MMM présentée dans l'algorithme 13 est la variante la plus basique et la plus coûteuse en espace mémoire. Dans cet algorithme, le calcul de la MMM est effectué en 3 boucles successives : 1) calcul du produit des opérandes A et B (lignes 2–7) ; 2) calcul du produit qP et accumulation avec le produit AB (lignes 8–14) ; et 3) division du résultat par 2^m (lignes 15–16).

Opérandes : $A = \sum_{i=0}^{s-1} a_i\, 2^{iw}$, $B = \sum_{j=0}^{s-1} b_j\, 2^{jw}$, $P = \sum_{j=0}^{s-1} p_j\, 2^{jw}$ avec $0 \leq A, B < 2P$
Prérequis : $m = s \times w$, $2^m > 4P$ et $P' = -P^{-1} \bmod 2^w$
Résultat : $T \equiv (AB \times 2^{-m}) \bmod P$, $0 \leq T < 2P$
begin

1 $t_{0\ldots 2s-1} \leftarrow 0$
2 **for** $i = 0$ **to** $s - 1$ **do**
3 $c \leftarrow 0$
4 **for** $j = 0$ **to** $s - 1$ **do**
5 $(c, u) \leftarrow t_{j+i} + a_i \times b_j + c$
6 $t_{j+i} \leftarrow u$
7 $t_{i+s} \leftarrow c$
8 **for** $i = 0$ **to** $s - 1$ **do**
9 $c \leftarrow 0$
10 $q_i \leftarrow (t_i \times P') \bmod 2^w$
11 **for** $j = 0$ **to** $s - 1$ **do**
12 $(c, u) \leftarrow t_{j+i} + q_i \times p_j + c$
13 $t_{j+i} \leftarrow u$
14 `add`(t_{i+s}, c)
15 **for** $j = 0$ **to** $s - 1$ **do**
16 $t_j \leftarrow t_{j+s}$
17 **return** $T = \sum_{j=0}^{s-1} t_j\, 2^{jw}$

Algorithme 13 : SOS (*Separated Operand Scanning*) de [DK90] tel que présenté dans [KAK96] et modifié pour supprimer la soustraction finale [Wal99a].

L'opération `add` dans l'algorithme 13 correspond à une addition avec propagation de retenue dans tous les mots $t_{k>i+s}$ du résultat intermédiaire.

Le CIOS est une version optimisée de l'algorithme SOS, proposé pour des implantations logicielles avec des petits temps de calcul et nécessitant des ressources mémoire moins importante pour le stockage des données intermédiaires en mémoire. Il est détaillé dans l'algorithme 14, dans lequel nous avons utilisé l'optimisation de Walter pour supprimer la soustraction finale. Le CIOS permet d'entrelacer efficacement le calcul de produits partiels et les étapes de réductions partielles grâce à une décomposition régulière des valeurs manipulées dans la MMM en s mots de w bits vérifiant $m = s \times w > 4P$. Ce découpage permet de décomposer les opérations modulaires sur des nombres de m bits en *opérations élémentaires* sur des entiers de w bits : additions, soustractions, multiplications et accès mémoires par exemple. On notera qu'en fonction du choix de w et de s la valeur de m peut être légèrement supérieure à $n + 2$.

En figure 3.8, nous illustrons un exemple de décomposition des opérandes et du résultat d'une MMM pour $n = 128$ bits en mots de $w = 32$ bits. Dans cet exemple, on peut voir que seulement 4 mots de

Opérandes : $A = \sum_{i=0}^{s-1} a_i\, 2^{iw}$, $B = \sum_{j=0}^{s-1} b_j\, 2^{jw}$, $P = \sum_{j=0}^{s-1} p_j\, 2^{jw}$ avec $0 \leq A, B < 2P$
Prérequis : $m = s \times w$, $2^m > 4P$ et $P' = -P^{-1} \bmod 2^w$
Résultat : $T \equiv (AB \times 2^{-m}) \bmod P$, $0 \leq T < 2P$
begin

```
1   t_{0...s-1} ← 0
2   for i = 0 to s − 1 do
3       d ← 0
4       for j = 0 to s − 1 do
5           (d, u_j) ← t_j + a_i × b_j + d
6       u_s ← d
7       q_i ← (u_0 × P') mod 2^w
8       c ← 0
9       for j = 0 to s − 1 do
10          (c, t_{j−1}) ← u_j + q_i × p_j + c
11      (c, t_{s−1}) ← u_s + c
12  return T = Σ_{j=0}^{s−1} t_j 2^{jw}
```

Algorithme 14 : CIOS basé sur [KAK96] avec optimisation de [Wal99a] (sans soustraction finale).

w bits sont nécessaires pour représenter le premier P, mais qu'il faut $s = 5$ mots pour représenter les valeurs A, B, T P' et q. Dans le CIOS rappelé dans l'algorithme 14, toutes les valeurs de moins de m bits sont représentées sur s mots. Dans le cas de P, les $m - n$ MSB du mot de poids fort sont fixés à 0. Il en va de même pour les $m - n - 1$ MSB des mots de poids fort de A, B et T. Nous reviendrons sur le choix des paramètres w et s dans la section 3.5 de ce chapitre.

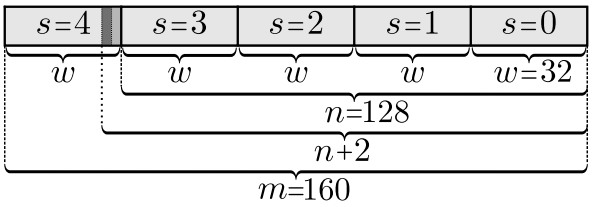


FIGURE 3.8 – Illustration de la décomposition dans le CIOS des valeurs manipulées dans la MMM de [Mon85] optimisée par Walter dans [Wal99a] (sans soustraction finale). Dans cet exemple, $n = 128$ bits et $w = 32$ bits.

Le calcul de la MMM dans l'algorithme CIOS repose sur deux niveaux de boucles. La boucle externe, d'indice i dans l'algorithme 14, itère sur les mots $\{a_0, a_1, ..., a_s\}$ de l'opérande A (c.-à-d. le *multiplieur*). La figure 3.9 décrit les séquences d'opérations effectuées dans les deux premières itérations d'indices $i = 0$ et $i = 1$ de la boucle externe du CIOS. À chaque itération de la boucle externe, le produit partiel $U_i = a_i \times B$ est calculé grâce à une boucle interne d'indice j itérant sur les mots $\{b_0, b_1, ..., b_s\}$ de l'opérande B (c.-à-d. le *multiplicande*). Ce calcul correspond à l'étape **I** dans la figure 3.9 (lignes 4–6 dans l'algorithme 14). Pour les itérations $i > 0$, on accumule aussi dans cette étape le résultat T_{i-1} de l'itération précédente au produit partiel $a_i B$. Durant l'étape **II** de l'itération courante, on calcule le quotient de réduction q_i à partir des w bits de poids faible du produit partiel U_i (ligne 7 dans l'algorithme 14). Enfin, le produit partiel $q_i P$ est calculé dans l'étape **III** grâce à une boucle interne d'indice j itérant sur les mots $\{p_0, p_1, ..., p_s\}$ du premier P, et sommé dans l'étape **IV** avec le résultat intermédiaire U_i. Le résultat intermédiaire T_i obtenu en fin d'itération correspond au résultat de la somme de $q_i P$ et de U_i dont les

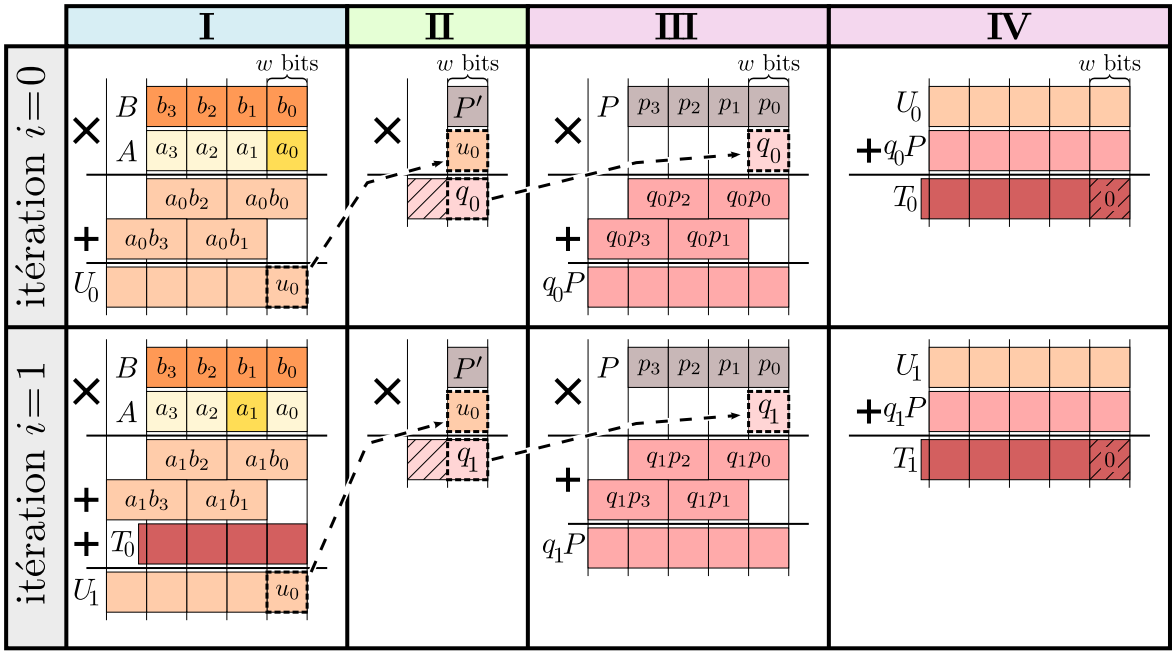


FIGURE 3.9 – Illustration des opérations dans les deux premières itérations externes du CIOS de [KAK96] (indice $i \in \{0, 1\}$) pour une décomposition des opérandes en $s = 4$ mots de w bits : **I**) calcul du produit partiel a_iB ; **II**) calcul du quotient de réduction q_i ; et **III**–**IV**) calcul du produit partiel q_iP et du résultat T_i. Les mots de w bits hachurés sont supprimés lors des calculs.

w bits du mot de poids faible sont supprimés.

La figure 3.9 permet d'illustrer les fortes dépendances entre les différentes opérations effectuées au sein et entre les itérations de la boucle externe. Par exemple, le calcul des valeurs U_i dans l'étape **I** nécessite le calcul des valeurs T_{i-1} aux itérations précédentes. Il faut donc attendre la latence d'une itération entre le calcul de 2 opérations consécutives à l'étape **I**, ce qui a tendance à augmenter le temps de calcul de la MMM. Ces dépendances peuvent aussi entraîner une baisse d'efficacité des implantations matérielles à cause de l'apparition de « bulles » dans le pipeline des unités lorsque l'on cherche à obtenir de hautes fréquences.

La variante FIOS permet d'intégrer les étapes de multiplications et de réductions partielles au sein d'une unique boucle interne (là où deux boucles internes sont nécessaires dans le CIOS). L'algorithme correspondant est décrit dans l'algorithme 15. On remarquera toutefois que l'intégration des produits et réductions au sein d'une unique boucle interne requière l'utilisation de l'opération `add` comme dans le SOS, coûteuse car générant des propagations de retenues sur plusieurs mots successifs.

L'algorithme FIPS proposé dans [Kal93] est décrit dans l'algorithme 16. De façon similaire à l'algorithme FIOS, cette variante permet d'entrelacer les calculs des produits $A \times B$ et $q \times P$ au sein d'une unique boucle mais pour un ordonnancement différent des produits partiels.

D'après les auteurs de [KAK96], le FIPS est moins performant que d'autres variantes de la MMM (p. ex. CIOS) pour des implantations sur processeurs généralistes. Ils considèrent toutefois que le FIPS pourrait être particulièrement adapté pour des implantations dans des processeurs spécialisés basés sur une architecture optimisée pour les opérations de « multiplication–accumulation ». Les processeurs dédiés au traitement du signal sont ainsi mis en avant par Koç et coll. comme étant des cibles privilégiées pour

Opérandes : $A = \sum_{i=0}^{s-1} a_i\, 2^{iw}$, $B = \sum_{j=0}^{s-1} b_j\, 2^{jw}$, $P = \sum_{j=0}^{s-1} p_j\, 2^{jw}$ avec $0 \leq A, B < 2P$
Prérequis : $m = s \times w$, $2^m > 4P$ et $P' = -P^{-1} \bmod 2^w$
Résultat : $T \equiv (AB \times 2^{-m}) \bmod P$, $0 \leq T < 2P$
begin

1 $t_{0\ldots s-1} \leftarrow 0$
2 **for** $i = 0$ **to** $s - 1$ **do**
3 $(c, u) \leftarrow t_0 + a_i \times b_0$
4 $\texttt{add}(t_1, c)$
5 $q_i \leftarrow (u \times P') \bmod 2^w$
6 $(c, u) \leftarrow u + q_i \times P_0$
7 **for** $j = 1$ **to** $s - 1$ **do**
8 $(c, u) \leftarrow t_j + a_i \times b_j + c$
9 $\texttt{add}(t_{i+j}, c)$
10 $(c, u) \leftarrow u + q_i \times p_j$
11 $t_{j-1} \leftarrow u$
12 $(c, u) \leftarrow t_s + c$
13 $t_{s-1} \leftarrow u$
14 $t_s \leftarrow t_{s+1} + c$
15 $t_{s+1} \leftarrow 0$
16 **return** $T = \sum_{j=0}^{s-1} t_j\, 2^{jw}$

Algorithme 15 : FIOS basé sur [KAK96] avec optimisation de [Wal99a] (sans soustraction finale).

l'implantation du FIPS. On notera toutefois qu'à notre connaissance, il n'existe pas d'implantations du FIPS sur FPGA utilisant les *slices* DSP de ces derniers.

Finalement, la variante CIHS introduite dans [KAK96] illustre une autre approche possible pour le calcul de la MMM. Celle-ci se base sur une modification du SOS utilisant un ordonnancement hybride des produits partiels pour la multiplication des opérandes (itérations sur les mots du produit similaires au FIPS) et pour la réduction (itérations sur les mots des opérandes similaires au CIOS).

Le CIHS permet de réduire le nombre d'opérations par rapport au SOS, ainsi que l'espace mémoire nécessaire au stockage des résultats intermédiaires. De par la décomposition des entiers modulo P utilisée et l'ordonnancement des produits partiels utilisé, il ne nécessite pas l'utilisation de l'opération `add`. Il ne souffre donc pas de l'augmentation de complexité due aux longues propagations de retenues induites par cette dernière.

Les coûts des différents algorithmes estimés dans [KAK96] sont rapportés dans le tableau 3.5. Clairement, le CIOS apparaît comme étant la variante la plus performante avec un nombre réduit d'opérations arithmétiques et d'accès mémoires et des besoins de mémoire moins importants que pour le SOS.

Des implantations efficaces de la MMM sur FPGA utilisant l'algorithme du CIOS ont été proposées en 2004 par MacLoone et coll. dans [MMM04c] et par MacIvor et coll. dans [MMM04b]. Ce dernier article propose d'ailleurs une comparaison d'architectures de multiplieurs modulaires pour le SOS, le CIOS, et le FIOS dont l'implantation est aussi détaillée dans [MMM04a] des mêmes auteurs. Ces architectures reposent sur l'utilisation des *slices* DSP 18×18 bits du FPGA Virtex-2 Pro, utilisables efficacement grâce à la décomposition des produits de $m \times m$ bits en s^2 produits partiels de $w \times w$ bits au sein des boucles internes dans ces trois algorithmes.

D'autres variantes de la MMM ont été proposées par Orup dans [Oru95] en 1995, utilisant aussi une décomposition en mots de grande base des opérandes de la multiplication modulaire. Ces algorithmes

Opérandes : $A = \sum_{i=0}^{s-1} a_i\, 2^{iw}$, $B = \sum_{j=0}^{s-1} b_j\, 2^{jw}$, $P = \sum_{j=0}^{s-1} p_j\, 2^{jw}$ avec $0 \leq A, B < 2P$
Prérequis : $m = s \times w$, $2^m > 4P$ et $P' = -P^{-1} \bmod 2^w$
Résultat : $T \equiv (AB \times 2^{-m}) \bmod P$, $0 \leq T < 2P$
begin

1 $t_{0\ldots s-1} \leftarrow 0$
2 **for** $i = 0$ **to** $s - 1$ **do**
3 **for** $j = 0$ **to** $i - 1$ **do**
4 $(c, u) \leftarrow t_0 + a_j \times b_{i-j}$
5 $\texttt{add}(t_1, c)$
6 $(c, u) \leftarrow u + q_j \times p_{i-j}$
7 $t_0 \leftarrow u$
8 $\texttt{add}(t_1, c)$
9 $(c, u) \leftarrow t_0 + a_i \times b_0$
10 $\texttt{add}(t_1, c)$
11 $q_i \leftarrow (u \times P') \bmod 2^w$
12 $(c, u) \leftarrow u + q_i \times p_0$
13 $\texttt{add}(t_1, c)$
14 $t_0 \leftarrow t_1$
15 $t_1 \leftarrow t_2$
16 $t_2 \leftarrow 0$
17 **for** $i = s$ **to** $2s - 1$ **do**
18 **for** $j = i - s + 1$ **to** $s - 1$ **do**
19 $(c, u) \leftarrow t_0 + a_j \times b_{i-j}$
20 $\texttt{add}(t_1, c)$
21 $(c, u) \leftarrow u + q_j \times p_{i-j}$
22 $t_0 \leftarrow u$
23 $\texttt{add}(t_1, c)$
24 $q_{i-s} \leftarrow t_0$
25 $t_0 \leftarrow t_1$
26 $t_1 \leftarrow t_2$
27 $t_2 \leftarrow 0$
28 **return** $T = \sum_{j=0}^{s-1} q_j\, 2^{jw}$

Algorithme 16 : FIPS de [Kal93] d'après la version de [KAK96] modifiée pour supprimer la soustraction finale [Wal99a].

Opérandes : $A = \sum_{i=0}^{s-1} a_i\, 2^{iw}$, $B = \sum_{j=0}^{s-1} b_j\, 2^{jw}$, $P = \sum_{j=0}^{s-1} p_j\, 2^{jw}$ avec $0 \leq A, B < 2P$
Prérequis : $m = s \times w$, $2^m > 4P$ et $P' = -P^{-1} \bmod 2^w$
Résultat : $T \equiv (AB \times 2^{-m}) \bmod P$, $0 \leq T < 2P$
begin

1 $t_{0\ldots s-1} \leftarrow 0$
2 **for** $i = 0$ **to** $s - 1$ **do**
3 $c \leftarrow 0$
4 **for** $j = 0$ **to** $s - i - 1$ **do**
5 $(c, u) \leftarrow t_{i+j} + a_i \times b_j + c$
6 $t_{i+j} \leftarrow u$
7 $(c, u) \leftarrow t_s + c$
8 $t_s \leftarrow u$
9 $t_{s+1} \leftarrow c$
10 **for** $i = 0$ **to** $s - 1$ **do**
11 $q_i \leftarrow (t_0 \times P') \bmod 2^w$
12 $(c, u) \leftarrow t_0 + q_i \times p_0$
13 **for** $j = 1$ **to** $s - 1$ **do**
14 $(c, u) \leftarrow t_j + q_i \times p_j + c$
15 $t_{j-1} \leftarrow u$
16 $(c, u) \leftarrow t_s + c$
17 $t_{s-1} \leftarrow u$
18 $t_s \leftarrow t_{s+1} + c$
19 $t_{s+1} \leftarrow 0$
20 **for** $j = i + 1$ **to** $s - 1$ **do**
21 $(c, u) \leftarrow t_{s-1} + a_j \times b_{s-j+i}$
22 $t_{s-1} \leftarrow u$
23 $(c, u) \leftarrow t_s + c$
24 $t_s \leftarrow u$
25 $t_{s+1} \leftarrow c$
26 **return** $T = \sum_{j=0}^{s-1} t_j\, 2^{jw}$

Algorithme 17 : CIHS basé sur [KAK96] avec optimisation de [Wal99a] (sans soustraction finale).

algo.	multiplications	additions	lectures	écritures	espace mémoire
SOS	$2s^2+s$	$4s^2+4s+2$	$6s^2+7s+3$	$2s^2+6s+2$	$2s+2$
CIOS	$2s^2+s$	$4s^2+4s+2$	$6s^2+7s+2$	$2s^2+5s+1$	$s+3$
FIOS	$2s^2+s$	$5s^2+3s+2$	$7s^2+5s+2$	$3s^2+4s+1$	$s+3$
FIPS	$2s^2+s$	$6s^2+2s+2$	$9s^2+8s+2$	$5s^2+8s+1$	$s+3$
CIHS	$2s^2+s$	$4s^2+4s+2$	$6.5s^2+6.5s+2$	$3s^2+5s+1$	$s+3$

TABLE 3.5 – Coûts respectifs des cinq algorithmes évalués dans [KAK96] en terme d'opérations arithmétiques élémentaires (sur des mots de w bits), d'accès mémoire (lectures et écritures) et de taille de mémoire (les opérandes sur m bits sont décomposés en s mots de w bits).

ont pour but d'accélérer le calcul du quotient de réduction q dans la MMM en relâchant les dépendances de données dans les itérations sur les produits partiels grâce à l'extension du domaine de Montgomery.

Cela permet de paralléliser efficacement le calcul des mots successifs q_i du quotient de réduction et celui des produits partiels $a_i \times B$ mais induit en contrepartie une augmentation du chemin de données dans le multiplieur. L'algorithme 18 présente l'algorithme le plus performant proposé par Orup pour la réduction du temps de calcul de la MMM.

Opérandes : $A = \sum_{i=0}^{s-1} a_i\, 2^{iw}$, $B = \sum_{j=0}^{s-1} b_j\, 2^{jw}$, $\widetilde{P} = \sum_{j=0}^{s-1} \widetilde{p}_j\, 2^{jw}$ avec $0 \leq A, B < 2\widetilde{P}$
Prérequis : $m = s \times w$, $2^m > 4\widetilde{P}$ avec $\widetilde{P} = P' \times P$ pour $P' = -P^{-1} \bmod 2^{w(d+1)}$ et $d \in \mathbb{N}$
Résultat : $T \equiv (AB \times 2^{-m}) \bmod P$, $0 \leq T < 2\widetilde{P}$
begin
1 $S_0 \leftarrow 0$
2 $q_{-d\ldots-1} \leftarrow 0$
3 **for** $i = 0$ **to** $s + d$ **do**
4 $q_i \leftarrow S_i \bmod 2^w$
5 $S_{i+1} \leftarrow (S_i/2^w) + q_{i-d} \times (\widetilde{P}/2^{w(d+1)}) + b_i \times A$
6 **return** $T \leftarrow 2^{sw} \times S_{s+d+1} + \sum_{j=0}^{d-1} \left(q_{s+j+1} \times 2^{wj}\right)$

Algorithme 18 : MMM avec quotient pipeliné proposé dans [Oru95]. L'entier d est le « délai » dans le pipeline avant le calcul du premier mot q_0.

Les travaux de Suzuki et de Ma et coll. publiés respectivement dans [Suz07] en 2007 et dans [MLPJ13] en 2013 s'appuient sur cet algorithme 18 pour l'implantation FPGA de multiplieurs modulaires profitant d'un parallélisme interne accru et de temps de calcul réduits mais avec des surfaces de circuit plus importantes.

Le coprocesseur 256 bits pour ECC sur des GF(P) *génériques* présenté dans [MLPJ13] est l'un des plus rapides proposés dans la littérature pour ce niveau de sécurité et utilisant une surface de circuit raisonnable [1]. Le multiplieur modulaire utilisé par Ma et coll. nécessite 37 *slices* DSP et les fréquences rapportées par les auteurs pour cette unité sont respectivement 250 MHz sur Virtex-4 et 290 MHz sur Virtex-5. Cependant, les auteurs ne fournissent pas d'autres résultats d'implantation pour cette unité.

Au vue des très bons résultats rapportés pour le processeur ECC de [MLPJ13], nous avons décidé de reproduire l'implantation de leur multiplieur pour un plus grand nombre de cibles FPGA et de tailles des éléments de GF(P). Pour les comparaisons avec nos HTMM 128 bits présentés dans [GT17d], nous

1. Pour rappel, nous ne considérons pas comme raisonnables des implantations utilisant plus de quelques dizaines de slices DSP et de BRAM.

avons d'abord implanté uniquement une version 128 bits du multiplieur de [MLPJ13]. Par la suite, nous avons étendu ces comparaisons dans [GT18a] en implantant une version 256 bits du multiplieur de Ma et coll.. Nos résultats d'implantation du multiplieur de [MLPJ13] pour 128 et 256 bits sont rapportés dans le tableau 3.9 sous la dénomination MA16. Ces résultats pour la version 256 bits implantée sur Virtex-4 et Virtex-5 sont très proches des résultats originaux rapportés par Ma et coll., avec 37 *slices* DSP utilisés et des fréquences presque identiques (mais jamais inférieures). Seul le comportement interne de nos unités est très légèrement différent : pour le calcul d'une MMM, nos implantations nécessitent 37 cycles (au lieu de 35 dans [MLPJ13]) mais l'intervalle entre les calculs de deux MMM est réduit à 28 cycles (au lieu de 29). Notre réimplantation du multiplieur utilisé dans [MLPJ13] n'est donc pas parfaite mais les résultats obtenus sont quand même très proches de ceux de l'implantation originale, et ce malgré le manque de certaines informations sur les implantations de [MLPJ13]. Nous tenons toutefois à faire remarquer que cette réimplantation n'a été rendue possible que grâce à la qualité des explications fournies par Ma et coll. qui nous ont permis de reproduire leurs résultats avec une précision satisfaisante. Ce n'est malheureusement pas le cas de toutes les implantations que l'on peut trouver dans la littérature. Nous espérons pour notre part donner l'exemple en fournissant un *générateur* de HTMM *distribué en licence libre* [GT18b] permettant de *reproduire* nos implantations du HTMM.

Plus récemment encore, l'algorithme *Iterative Digit-Digit Montgomery Multiplication* (IDDMM) a été proposé par Morales-Sandoval et coll. en 2016 dans [MSDP16] pour le calcul de la MMM modulo des premiers génériques de 256 à 1024 bits. L'algorithme proposé est très fortement inspiré du CIOS dans lequel les deux boucles internes ont été fusionnées sans impacter le calcul des différents produits partiels. En raison de sa ressemblance avec l'algorithme original de [KAK96], nous ne le retranscrirons pas ici.

Parmi les résultats rapportés dans [MSDP16], nous avons sélectionné pour comparaisons ceux obtenus pour l'implantation de leur multiplieur 256 bits avec un chemin de données interne de 64 bits sur Virtex-5. Cette implantation est nommée MO64 dans le tableau 3.9.

Dans [ACZ16] publié en 2016, Amiet et coll. proposent une implantation sur Virtex-7 d'une variante de l'algorithme IDDMM de [MSDP16]. Pour comparaisons avec nos multiplieurs, nous utilisons les résultats d'implantation de deux de leurs multiplieurs 256 bits conçus avec des largeurs de chemins de données internes de 32 et de 64 bits et notés respectivement AM32 et AM64 dans le tableau 3.9.

Certains travaux de la littérature ont été dédiés à la conception de multiplieurs permettant d'atteindre *de très hauts débits*. Ces travaux utilisent pour la plupart des techniques de déroulage de boucles pour simplifier le contrôle des itérations dans les multiplieurs au prix d'une consommation de surface plus importante. Dans ces multiplieurs la latence pour une unique MMM est en général plus élevée mais l'intervalle entre différentes multiplications est très faible.

C'est par exemple le cas dans les multiplieurs basés sur des architectures systoliques dont l'utilisation pour le calcul de la MMM a été proposée par Walter en 1993 dans [Wal93]. Les multiplieurs systoliques sont composés d'un ensemble de blocs atomiques, chacun de ces blocs prenant en charge le calcul d'une opération en 1 cycle horloge. Ils sont conçus pour recevoir de nouveaux opérandes à chaque cycle horloge et générer le résultat d'une multiplication à chaque cycle après un délai correspondant à la latence de la première MMM. Plusieurs optimisations de ces architectures ont été proposées depuis [Wal93], par exemple dans [BP01] en 2001, dans [OBPV03] en 2003 ou encore dans [MMM05] en 2005.

Des architectures récentes utilisant le principe du calcul systolique ont été proposées par Mrabet et coll. dans [MEML+17] en 2017 pour l'implantation sur FPGA Artix-7 du CIOS pour des premiers P

de 128 et 256 bits. Deux versions de ces architectures ont été proposées pour des décompositions internes des entiers modulo P en respectivement 8 et 16 mots. Les résultats d'implantation de ces deux versions sont détaillés dans le tableau 3.9 sous les références MR8 et MR16.

Finalement, le travail publié dans [MBCM17] par Massolino et coll. en 2017 propose des implantations des algorithmes CIOS et FIOS de [KAK96] pour ECC sur le FPGA *low-power* IGLOO 2 de MicroSemi. Les deux architectures ECC proposées utilisant le CIOS (notée MAS1) ou le FIOS (notée MAS2) visent à minimiser la consommation du coprocesseur en surface et utilisent pour cela des unités arithmétiques prenant en charge à la fois le calcul de la MMM et celui de l'addition/soustraction modulaire. Les résultats d'implantation de MAS1 et MAS2 sont rapportés dans le tableau 3.9.

Pour conclure cet état de l'art des implantations matérielles de la MMM de la littérature, on notera qu'il existe aussi tout un ensemble d'implantations ASIC que nous n'avons pas abordées ici. Dans ces implantations, il n'est en général pas considéré souhaitable d'utiliser des petits multiplieurs semblables aux *slices* DSP des FPGA. Plusieurs algorithmes ont été proposés pour optimiser l'implantation de la MMM pour des premiers P génériques sans utiliser de tels multiplieurs. C'est par exemple le cas de l'algorithme *Multiple-Word Radix-2 Montgomery Multiplication* (MWR2MM) proposé dans [TK99]. L'algorithme MWR2MM se base sur des itérations sur les produits partiels $a_i \times B$ dans lesquels les mots de 2 bits a_i du multiplieur A sont multipliés par la totalité des m bits du multiplicande B. Les travaux [TTK01], [TK03], [TT03] ou encore [HGEG11] sont de bons exemples d'implantations ASIC pour lesquelles les résultats rapportés par les auteurs sont intéressants. Toutefois, nos unités arithmétiques pour la MMM étant élaborées pour des implantations matérielles sur FPGA optimisant l'utilisation des *slices* DSP, nous ne considérerons pas ici ce type de variantes.

3.4 Utilisation de *l'hyper-threading* dans HTMM

Pour la conception de nos multiplieurs modulaires, nous avons choisi d'utiliser l'algorithme CIOS de [KAK96] avec suppression de la soustraction finale proposée dans [Wal99a] (cf algorithme 14). Ce choix a été principalement motivé par la simplicité et la régularité de cet algorithme, qui le rend particulièrement apte à être implanté en utilisant les ressources matérielles hétérogènes des FPGA (en particulier les *slices* DSP).

Comme nous l'avons vu précédemment, le calcul des opérations dans la boucle externe d'indice i du CIOS créent de fortes dépendances séquentielles de données (p. ex. calcul du quotient de réduction partiel q_i). À cause de ces dépendances, le calcul des opérations internes du CIOS peut créer des « bulles » dans les *slices* DSP, en particulier quand ces derniers sont configurés avec 3 ou 4 étages de pipeline pour pouvoir fonctionner à haute fréquence. En raison de ces « bulles », certains étages des *slices* DSP sont passifs lors de certains cycles, ce qui diminue l'efficacité matérielle des implantations.

Deux solutions sont classiquement proposées dans la littérature pour résoudre ce problème de dépendances. La première consiste à utiliser un algorithme plus complexe et plus coûteux pour relâcher les dépendances de données, comme celui de [Oru95] utilisé dans [MLPJ13] et rappelé dans l'algorithme 18. La deuxième solution privilégie la réutilisation des blocs matériels du FPGA au cours des itérations de la MMM pour calculer des tâches différentes, au prix d'un contrôle plus complexe des transferts de données dans l'unité, comme c'est le cas dans [MBCM17] par exemple.

Pour concevoir des multiplieurs efficaces, c.-à-d. limitant au maximum la présence de « bulles » dans le pipeline interne, nous proposons une solution alternative utilisant *l'hyper-threading* afin de recouvrir

les latences et les « bulles » dans nos multiplieurs par du calcul utile. *L'hyper-threading* est généralement utilisé pour l'implantation d'algorithmes itératifs dans lesquels le nombre d'opérations à exécuter par itération est plus petit que la profondeur du pipeline, certains étages étant alors inactifs à certains cycles (voir [KM03] par exemple). À notre connaissance, ce principe n'a cependant jamais été utilisé pour le calcul de la MMM.

Dans le multiplieur modulaire proposé dans ce chapitre, nous utilisons *l'hyper-threading* pour recouvrir les latences dans le pipeline interne par le *calcul en parallèle* d'autres MMM *indépendantes*. Ainsi, un *multiplieur physique* est partagé simultanément par plusieurs *multiplieurs logiques* (LM) indépendants se partageant à tour de rôle les ressources matérielles de l'unité. Nous avons nommé ce multiplieur modulaire HTMM, pour *hyperthreaded modular multiplier*.

Nous rappelons que dans l'algorithme 14 du CIOS utilisant l'optimisation de Walter [Wal99a] les valeurs de A, B, q, T, P et P' sont représentées sur $m = s \times w$ bits avec $2^m > 4P$. Dans la plupart des FPGA, les unités matérielles câblées de type *slice* DSP ne permettent pas de traiter directement des opérandes de grandes tailles. En particulier, on rappellera que dans nos FPGA la taille des mots dans les BRAM est au plus de 36 bits et que les *slices* DSP peuvent calculer des produits au maximum de 18×18 ou 18×25 bits. Dans les applications cryptographiques que nous visons, les valeurs manipulées font en général plus d'une centaine de bits (128 ou 256 bits par exemple). Manipuler en parallèle les m bits des opérandes de la MMM demanderait alors un grand nombre de BRAM et de *slices* DSP.

Par exemple, 4 BRAM d'une largeur de 36 bits sont nécessaires pour écrire ou lire en mémoire 128 bits en parallèle. Par contre, le nombre de valeurs à stocker en mémoire durant le calcul de la MMM est au maximum de 10 pour les applications 256 bits ECC et de 20 pour les applications 128 ou 256 bits HECC, sur les 512 valeurs pouvant être stockées en pratique dans une BRAM. Dans ce cas de figure, les BRAM implantées sont clairement sous-exploitées car en grande partie vides. On notera aussi que le calcul direct d'un produit 128×128 bits nécessite 64 *slices* DSP 18×18 bits $\rightarrow$ 48 bits.

Dans notre HTMM, nous décomposons les éléments de m bits en mots plus petits de w bits traités de manière séquentielle ($m \gg w$). Comme dans l'algorithme 14, on dénotera s le nombre de mots de w bits nécessaires à la représentation des différents opérandes, avec $m = s \times w$.

Cette décomposition nous permet d'utiliser efficacement les *slices* DSP et les BRAM des FPGA récents. Elle nous permet aussi de réduire le coût en surface des interconnexions dans nos accélérateurs (H)ECC complets (décrits dans le chapitre 4). Enfin, un dernier bénéfice de cette décomposition vient du fait qu'elle peut nous permettre de diminuer la longueur de certains chemins critiques dans nos unités. C'est en particulier le cas pour le calcul de certaines opérations élémentaires comme l'addition pour laquelle l'utilisation d'opérandes de w bits permet de limiter la propagation des retenues.

Comme nous l'avons déjà indiqué ci-dessus, nous utilisons le CIOS pour sa régularité et sa simplicité, combiné à une architecture hyper-threadée nous permettant de recouvrir les latences dans le pipeline câblé des *slices* DSP du FPGA. Un HTMM (unité *physique*) peut supporter σ LM (unités *logiques*) pour calculer jusqu'à σ MMM *indépendantes* simultanément. Dans nos HTMM, le chargement des opérandes et la génération du résultat sont limités à un seul LM par cycle (voir la figure 3.10).

Le comportement de HTMM pour $\sigma = 3$ avec une décomposition des opérandes en $s = 2$ mots est illustré dans la figure 3.10 (les cycles horloge sont notés CC) :

- CC 1 : chargement des mots de w bits (A_0, B_0) pour la première MMM ;
- CC 2 : début du calcul de $A \times B$ dans LM1 et chargement de (A_1, B_1) ;
- CC 3 : chargement des mots de w bits (C_0, D_0) pour la seconde MMM ;

- CC 4 : début du calcul de $C \times D$ dans LM2 et chargement de (C_1, D_1) ;
- CC 5 : chargement des mots de w bits (E_0, F_0) pour la seconde MMM ;
- CC 6 : début du calcul de $E \times F$ dans LM3 et chargement de (E_1, F_1) ;
- durant les CC suivants, σ produits indépendants sont calculés dans les LM ;
- CC δ : sortie du premier mot de w bits AB_0 du produit AB ;
- CC $\delta + 1$: sortie du second mot AB_1 du produit AB ;
- durant les 4 CC suivants, les s mots des produits CD et EF sont successivement générés.

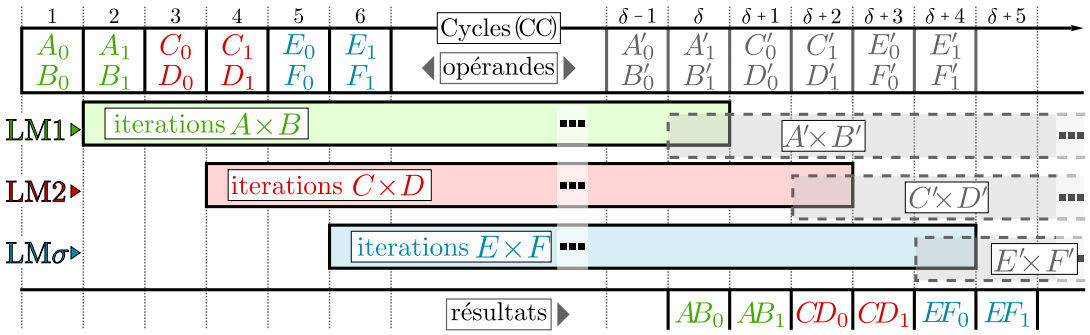


FIGURE 3.10 – Illustration des transferts des opérandes et des calculs dans HTMM pour $\sigma = 3$ LM. Les paires d'opérandes (A, B), (C, D) et (E, F) et les produits AB, CD et EF calculés sont décomposés en $s = 2$ mots de w bits. Les cycles horloges sont notés CC (clock cycles).

Après le chargement des opérandes (p. ex. $A_{0,\ldots,s-1}, B_{0,\ldots,s-1}$), tous les étages de pipeline dans chaque *slice* DSP effectuent successivement des calculs intermédiaires pour chacune des différentes MMM indépendantes (une opération est calculée par MMM et par cycle). Dans la suite du chapitre, nous appellerons λ la *latence* entre le chargement du *premier* mot des opérandes et la sortie du *dernier* mot du produit. Dans notre exemple en figure 3.10, δ désigne le nombre de cycles entre le chargement du premier mot des opérandes et la sortie du premier mot du produit. Les opérandes et le résultat étant décomposés en $s = 2$ mots, la latence λ dans cet exemple est donc de $\delta + 1$ cycles.

Dans notre HTMM, de nouveaux opérandes peuvent être chargés dans le premier étage du pipeline pendant que les derniers étages terminent le calcul de la MMM courante. Nous noterons τ l'*intervalle* entre deux MMM successives dans le même LM. La valeur de τ peut être calculée à partir du nombre s de mots, du nombre σ de LM dans le HTMM et du délai θ entre 2 MMM consécutives dans 2 LM différents : $\tau = s \times \sigma \times \theta$. Dans l'exemple de la figure 3.10, le chargement des nouveaux opérandes (A'_0, B'_0) peut commencer dans LM 1 après un intervalle $\tau = \delta - 1$ cycles.

Dans le CIOS (algorithme 14), les calculs effectués durant l'itération i de la boucle externe peuvent être décomposés en trois tâches dépendantes les unes des autres :

- tâche 1 (lignes 3–5) : produit partiel $a_i \times B$ et accumulation de l'itération précédente $(i - 1)$;
- tâche 2 (ligne 6) : calcul du « quotient de réduction partielle » q_i pour l'algorithme de Montgomery ;
- tâche 3 (lignes 8–10) : produit partiel $q_i \times P$ et accumulation du résultat de la tâche 1.

La tâche 1 correspond donc à l'étape **I**, la tâche 2 à l'étape **II**, et la tâche 3 aux étapes **III** et **IV** illustrées en figure 3.9. La division du résultat de l'itération i par 2^w dans le CIOS est effectuée entre la tâche 3 de l'itération i et la tâche 1 de l'itération $i + 1$ par suppression des w bits du mot de poids faible (voir la section 3.6.2 pour les détails d'implantation).

L'architecture de notre HTMM, représentée en figure 3.11, est conçue conformément à cette décomposition en 3 tâches, chacune de ces dernières étant prise en charge par un bloc matériel dédié. À cause

des étages de pipelines internes dans les *slices* DSP, le résultat de la tâche 3 n'est pas disponible immédiatement dans la tâche 1 pour l'itération $i + 1$ suivante et on doit attendre autant de cycles que d'étages dans le pipeline interne du HTMM. L'utilisation de *l'hyper-threading* nous aide à masquer ce délai : en calculant σ MMM indépendantes, nous pouvons remplir tous les étages du pipeline avec des calculs utiles.

3.4.1 Note sur le fonctionnement du HTMM

La gestion de *l'hyper-threading* dans nos multiplieurs implique la mise en place d'un contrôle interne un peu plus complexe permettant de s'assurer que les ressources matérielles du FPGA sont bien utilisées au bon cycle par le bon LM. Les signaux de contrôle interne sont engendrés par des petites FSM à des cycles précis dépendant des paramètres du HTMM et de la configuration de *l'hyper-threading*. Du fait de l'utilisation de ces FSM, le fonctionnement du HTMM est contraint et le calcul d'une nouvelle MMM ne peut être démarré dans un LM libre qu'à certains cycles précis. Par exemple, dans le HTMM illustré en figure 3.10, le calcul d'un produit ne peut être démarré qu'aux cycles CC 1, CC 3 et CC 5. Toute paire d'opérandes envoyée au HTMM en dehors de ces cycles sera ignorée. Les cycles pendant lesquels de nouvelles MMM peuvent être calculées dans le HTMM sont indiqués par l'activation d'un signal de 1 bit en sortie de l'unité.

L'ajout de matériel pour permettre les entrées à des instants quelconques serait coûteux pour une efficacité faible.

3.5 Premières versions du HTMM 128 bits de [GT17d]

Dans [GT17d], nous avons proposé *deux implantations* de notre HTMM sur *trois FPGA différents* de Xilinx, à savoir un Virtex-4, un Virtex-5 et un Spartan-6. Ces deux versions ont été *décrites à la main* en VHDL pour des tailles de premiers P de 128 bits. Dans la première version nous avons utilisé les blocs BRAM des FPGA pour stocker les opérandes des multiplications modulaires, et de la mémoire distribuée DRAM dans la deuxième version du HTMM.

Nous allons à présent décrire et expliquer les choix des paramètres internes que nous avons utilisés dans ce HTMM. Nous détaillerons ensuite les résultats d'implantation obtenus après placement et routage des 2 versions du HTMM sur les 3 FPGA sélectionnés. Nous verrons que grâce à l'utilisation de *l'hyper-threading* notre HTMM permet d'obtenir de très bonnes performances pour le calcul de plusieurs MMM indépendantes tout en limitant la consommation de surface de circuit. En particulier, nous verrons que notre HTMM offre de meilleurs compromis temps–surface que le multiplieur modulaire le plus rapide de l'état de l'art des *premiers quelconque*, à savoir celui de [MLPJ13].

3.5.1 Sélection des paramètres s et w dans le HTMM 128 bits

Les performances du HTMM sont impactées à la fois par s et par w. Pour rappel, s est le nombre d'itérations dans les boucles internes et dans la boucle externe de l'algorithme 14 du CIOS. L'augmentation de s implique une latence λ plus élevée (celle-ci augmente en $O(s^2)$). La décomposition des valeurs manipulées dans la MMM en mots de w bits induit le calcul de produits partiels de $w \times w$ bits dans les itérations des boucles internes du CIOS. Pour réduire la valeur de s et donc la latence λ du HTMM, w doit être sélectionné de façon à être le plus grand possible tout en permettant de remplir efficacement les *slices* DSP utilisés.

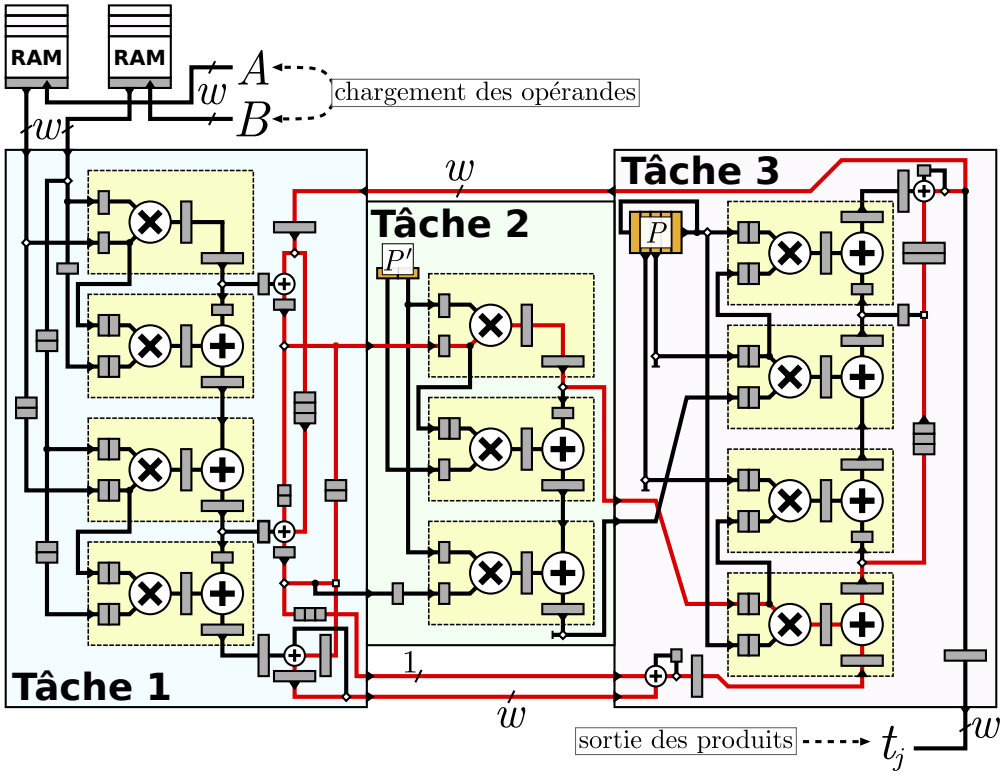


FIGURE 3.11 – Architecture du HTMM 128 bits de [GT17d] pour $\sigma = 3$ et $s = 4$ (sans le détail du contrôle). Les rectangles bleus, verts et rouges représentent les blocs matériels pour chaque tâche dans l'Algo. 14. Les rectangles jaunes sont les *slices* DSP (avec le détail du pipeline interne) et les gris sont les registres. Le chemin critique dans la boucle externe est en rouge.

On remarquera que les boucles externes et internes du CIOS utilisent une décomposition identique des opérandes A (le multiplieur) et B (le multiplicande). Les *slices* DSP48E disponibles dans les FPGA récents (Virtex-5 par exemple) peuvent utiliser des configurations rectangulaires pour calculer des produits signés de 18×25 bits. Pour maximiser l'utilisation de ces *slices* DSP, il est possible d'utiliser sur ces FPGA des décompositions asymétriques dans les boucles externes et internes du CIOS. L'implantation efficace de ce type de décomposition est cependant difficile à mettre en œuvre, en particulier quand on veut optimiser le nombre de *slices* DSP (voir [dDP09], [SC10], [GAKCL12] ou [RMIT14] par exemple). Elle implique par ailleurs la mise en place d'interfaces plus complexes pour la gestion entrées et sorties de l'unité et le stockage des opérandes en mémoire.

Pour simplifier la gestion des opérandes dans notre unité HTMM, nous avons décidé d'utiliser uniquement les configurations 18×18 bits des multiplieurs câblés disponibles dans l'ensemble des FPGA sélectionnés. La mise en place d'architectures de HTMM construites autour de multiplieurs câblés rectangulaires est une piste intéressante à étudier pour améliorer encore les performances de nos unités ou pour en réduire la surface. Par manque de temps, nous n'avons malheureusement pas pu explorer cette piste, que nous envisageons d'explorer pour de futures améliorations de nos multiplieurs modulaires.

Dans les *slices* DSP, les opérandes sont signés en complément à 2 et le signe de chaque opérande est indiqué par la valeur de son MSB. Pour le calcul de produits non signés, on doit forcer le MSB des opérandes à zéro d'après la documentation des FPGA Xilinx. Seuls les 17 LSB des opérandes sont donc

utilisables dans les multiplieurs câblés 18×18 bits pour le calcul des produits partiels dans les boucles internes de l'algorithme 14. Pour remplir efficacement les *slices* DSP 18×18 bits, la taille w des mots dans notre HTMM doit alors être 17 bits ou un multiple de 17 bits. Afin de déterminer la meilleure décomposition à utiliser dans notre HTMM, avons exploré différentes valeurs de w : 17, 34, 51 et 68 bits.

Pour $w = 17$ bits, les produits partiels $(a_i \times b_j)$, $(u_0 \times P')$ ou $(q_i \times p_j)$ dans l'algorithme 14 du CIOS nécessitent un unique *slice* DSP chacun (soit 3 au total pour 1 HTMM). De plus, les mots a_i et b_j des opérandes dans chacun des LM peuvent être mémorisés dans une unique BRAM (la largeur de 36 bits des BRAM est suffisante pour stocker deux mots de 17 bits par adresse mémoire). Cependant, le nombre s d'itérations des boucles externes et internes dans le CIOS est important dans le cas où w est petit, ce qui implique que la latence λ des LM est importante.

Quand w est supérieur à 36 bits, le stockage de chaque opérande nécessite plusieurs BRAM pour mémoriser un petit nombre de mots. Par exemple, la décomposition des éléments de GF(P) de 128 bits utilisés dans HECC en mots de $w = 68$ bits dans HTMM nécessiterait l'implantation de 2 BRAM d'une largeur de 36 bits pour le stockage d'un opérande sur seulement 2 mots. Une décomposition des opérandes en mots de $w = 51$ bits ou plus grands dans HTMM entraîne l'implantation de circuits de taille importante. Pour $w = 51$ bits, l'implantation de l'algorithme 14 nécessite 9 *slices* DSP pour les tâches 1 et 3 et 6 *slices* DSP pour le calcul de produit partiel modulo 2^w dans la tâche 2 (soit un total de 24 *slices* DSP). Pour $w = 68$ bits, la taille du circuit est encore plus importante : 16 *slices* DSP dans les blocs matériels pour les tâches 1 et 3 et 9 *slices* DSP pour le calcul de produit partiel modulo 2^w dans la tâche 2 (soit un total de 41 *slices* DSP). Pour ces 2 valeurs de w, 4 BRAM sont aussi nécessaires pour la mémorisation des opérandes.

Nous avons finalement sélectionné $w = 34$ bits pour nos implantations FPGA du HTMM 128 bits de [GT17d]. Pour cette taille de mots, 4 *slices* DSP sont requis dans chacun des blocs matériels pour les tâches 1 et 3 comme illustré en figure 3.11 et 3 *slices* DSP pour le calcul de produit partiel modulo 2^w dans le bloc pour la tâche 2. Un HTMM complet intègre donc un total de 11 *slices* DSP.

Les mots de $w = 34$ bits d'un opérande sont suffisamment petits pour être stockés dans une seule BRAM. Dans nos FPGA, la taille minimale des BRAM varie entre 9 kbit sur Spartan-6 et 18 kbit sur Virtex-4 et 5. Celles-ci peuvent donc stocker au moins 200 mots de w bits, ce qui correspond à au moins 50 opérandes par BRAM pour HECC (25 opérandes par BRAM pour ECC). En pratique, le nombre σ de LM dans nos multiplieurs hyper-threadés est bien plus petit que ces bornes (voir section 3.5.2) 2 BRAM sont alors suffisantes pour stocker les couples d'opérandes pour les différentes MMM calculées en même temps dans les σ LM.

Les s produits de mots de w bits dans les itérations des boucles internes des tâches 1 et 3 de l'algorithme 14 sont calculés séquentiellement en s cycles dans les blocs matériels dédiés (1 produit partiel est initié par cycle). Les résultats des produits partiels $w \times m$ bits en sortie de ces boucles internes dans les tâches 1 et 3 (resp. $a_i \times B$ ou $q_i \times P$) sont sur $w + m$ bits décomposés en $s + 1$ mots de w bits.

Les s mots de poids faibles des résultats des boucles internes sont engendrés séquentiellement en s cycles. Ils correspondent aux valeurs successives des variables u_j (pour la boucle interne de la tâche 1) et t_{j-1} (pour la boucle interne de la tâche 3) dans l'algorithme 14. Un cycle supplémentaire est nécessaire dans les tâches 1 et 3 pour la propagation des retenues finales d et c dans les mots de poids forts des résultats (resp. u_s en ligne 6 et t_{s-1} en ligne 11 de l'algorithme 14). L'intervalle entre 2 produits partiels calculés successivement pour 2 LM dans chacun des blocs pour les tâches 1 et 3 est alors de 5 cycles dans le HTMM 128 bits quand $w = 34$ bits.

3.5.2 Sélection du paramètre σ dans le HTMM 128 bits

Dans l'architecture du HTMM 128 bits de [GT17d], le chemin critique « tâche 1 → tâche 2 → tâche 3 → tâche 1 » indiqué en rouge en figure 3.11 pour la boucle externe d'indice i a une durée de $\alpha = 15$ cycles. La valeur du paramètre α dépend de l'architecture interne du HTMM et en particulier du nombre de *slices* DSP utilisés dans chaque bloc matériel implanté qui lui dépend uniquement de la taille w des mots internes. Dans nos multiplieurs, le paramètre α est fixe pour une taille w de mots internes donnée (c.-à-d. quelle que soit la taille du premier P ou la valeur de s).

Comme discuté dans la section précédente, les boucles internes dans les tâches 1 et 3 ont quant à elles besoin de $s+1 = 5$ cycles pour lancer les s produits $w \times w$ bits nécessaires au calcul d'un produit partiel $m \times w$ bits. Pour remplir le pipeline du HTMM 128 bits sans « bulles », nous avons fixé le paramètre σ de telle sorte que

$$\sigma = \left\lceil \frac{\alpha}{s+1} \right\rceil .$$

Si σ est plus grand, le résultat de la tâche 3 doit en effet être retardé pour attendre que tous les produits partiels $w \times m$ bits aient été lancés dans la tâche 1 pour les σ différents LM. Ce retard implique l'ajout de $\sigma(s+1) - \alpha$ registres supplémentaires entre le bloc matériel pour la tâche 3 et celui dédié à la tâche 1.

Si σ est au contraire plus petit, le nombre de produits partiels à calculer dans la tâche 1 (dans les différents LM) est insuffisant pour recouvrir la traversée des $\alpha = 15$ étages du chemin critique « tâche 1 → tâche 2 → tâche 3 → tâche 1 » dans notre HTMM 128 bits. Cela mène à la création de $\alpha - \sigma(s+1)$ « bulles » dans le pipeline interne du HTMM et donc à la sous utilisation de certaines ressources matérielles.

Dans notre HTMM 128 bits, $\alpha = 15$ cycles et $s = 4$ mots. Nous avons alors sélectionné $\sigma = 3$ pour les implantations présentées dans [GT17d]. Avec $\sigma = 3$ LM, le temps de calcul des produits partiels pour les différents LM dans la tâche 1 est $\sigma(s+1) = 15$ cycles, ce qui correspond exactement à la valeur minimale de α.

3.5.3 Gestion du premier P dans le HTMM

Dans nos versions du HTMM de [GT17d], le premier P de 128 bits est stocké dans l'architecture de notre HTMM grâce à $s+1$ registres de w bits. Il est fixé à l'implantation et ne peut donc pas être modifié à l'exécution. Il en est de même pour la constante $P' = (-P^{-1}) \bmod 2^w$, stockée au sein de l'unité dans 1 registre de w bits.

Contrairement à certains multiplieurs de l'état de l'art, comme celui de [MLPJ13] par exemple, nous avons décidé de stocker le premier P et la constante P' dans le HTMM plutôt que de les transférer vers l'unité depuis une mémoire externe à chaque calcul d'une MMM. Cela nous a permis de simplifier l'interface du HTMM et la mise en place du contrôle dédié à la gestion des opérandes dans l'unité.

Les $s+1$ registres de w bits permettent le stockage des s mots $p_{0\ldots s-1}$ de P dans le bloc dédié à la tâche 3. Ils sont organisés sous forme d'un *buffer circulaire*, représenté en orange dans le bloc matériel dédié à la tâche 3 de la figure 3.11. Comme présenté dans cette figure, les registres de w bits du *buffer* circulaire sont connectés aux *slices* DSP du bloc.

En figure 3.12, nous illustrons le fonctionnement du *buffer* pour le calcul du produit $q_i \times p_0$ de $w \times w$ bits effectué aux itérations i de la boucle externe et $j = 0$ de la 2$^{\text{e}}$ boucle interne dans l'algorithme 14. Les mots de w bits q_i et p_0 sont découpés en mots de 17 bits de la façon suivante : $q_i = q_{i\,H}\, 2^{17} + q_{i\,L}$ et

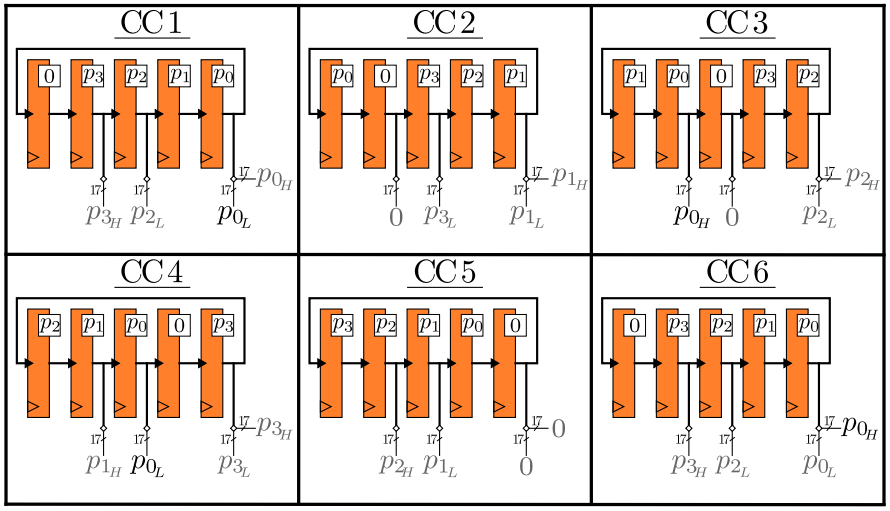


FIGURE 3.12 – Illustration du fonctionnement du *buffer* circulaire servant au stockage de P dans le bloc 3 du HTMM de [GT17d]. Le cycle d'horloge « CC 1 » correspond au début du calcul de $q_i \times p_0$ dans le premier *slice* DSP du bloc matériel pour la tâche 3.

$p_0 = p_{0H}\, 2^{17} + p_{0L}$. Si on considère que le cycle CC 1 en figure 3.12 correspond à l'arrivée de q_{iL} dans le 1er *slice* DSP du bloc 3 (c.-à-d. le *slice* DSP du bas en figure 3.11), les mots de p_0 sont parcourus de la façon suivante :

- CC 1 : les 17 LSB p_{0L} de p_0 sont envoyés au 1er *slice* DSP pour le calcul de $q_{iL} \times p_{0L}$;
- CC 3 : les 17 MSB p_{0H} de p_0 sont envoyés au 2e *slice* DSP pour le calcul de $q_{iL} \times p_{0H}$;
- CC 4 : les 17 LSB p_{0L} de p_0 sont envoyés au 3e *slice* DSP pour le calcul de $q_{iH} \times p_{0L}$;
- CC 6 : les 17 MSB p_{0H} de p_0 sont envoyés au 4e *slice* DSP pour le calcul de $q_{iH} \times p_{0H}$.

On peut aussi constater sur cette figure que pendant le CC 6 les 17 LSB p_{0L} de p_0 sont envoyés au 1er *slice* DSP pour le calcul de $q_{iL} \times p_{0L}$ dans le LM suivant.

3.5.4 Résultats d'implantation du HTMM 128 bits de [GT17d]

Le schéma d'architecture de notre HTMM 128 bits proposé dans [GT17d] avec les paramètres internes $w = 34$ bits, $s = 4$ et $\sigma = 3$ LM correspond à celui représenté en figure 3.11.

Deux versions du HTMM ont été implantées *manuellement* sur 3 FPGA différents de Xilinx : les Virtex-4 et 5 pour comparaison avec l'état de l'art récent et un Spartan-6 pour évaluation des performances sur un FPGA *low-cost*. Les résultats de ces implantations sont détaillés dans le tableau 3.6. Dans ce tableau, la latence λ est donnée en cycles pour 1 MMM mais le temps de calcul est donné pour 3 MMM afin d'illustrer les bénéfices de l'*hyper-threading*. Le temps de calcul pour 3 MMM, en ns, est égal à :

$$\frac{\lambda + 2 \times \theta}{f \times 10^{-3}},$$

avec $\theta = 5$ le nombre de cycles entre le calcul de 2 MMM indépendantes consécutives dans 2 LM et f la fréquence en MHz du circuit implanté. Les deux versions présentées diffèrent par les ressources matérielles utilisées pour l'implantation des mémoires RAM dans le HTMM (cf. figure 3.11). La première version (nommée HTMMb dans le tableau 3.6) utilise les BRAM câblées des FPGA. La deuxième version (HTMMd dans le tableau 3.6) utilise des mémoires DRAM implantées dans les *slices* logiques des FPGA.

Dans le tableau 3.7, nous rappelons aussi pour comparaison les résultats du multiplieur GF(P) MA16 que nous avons implanté à partir de la solution de [MLPJ13] pour des premiers P génériques de 128 bits.

version	FPGA	*slices* logiques	LUT	FF	BRAM	*slices* DSP	fréq. MHz	λ cc	temps 3M ns
HTMMb	V4	449	364	615	2	11	328	69	241
	V5	249	371	593	2	11	357		221
	S6	180	359	587	2	11	304		260
HTMMd	V4	1346	1128	1638	0	11	330	69	239
	V5	517	652	1616	0	11	400		198
	S6	483	1344	1631	0	11	302		261

TABLE 3.6 – Résultats d'implantation sur FPGA Spartan-6 (S6), Virtex-4 (V4) et Virtex-5 (V5) de nos HTMM 128 bits pour HECC de [GT17d]. Les BRAM ont des tailles de 9 kbits sur S6 et de 18 kbits sur V4 et V5. Le temps est donné pour le calcul de $\sigma = 3$ MMM indépendantes.

FPGA	*slices* logiques	LUT	FF	BRAM	*slices* DSP	fréq. MHz	λ cc	temps 3M ns
V4	879	1201	1311	6	21	252	27	266
V5	440	1027	1310	6	21	292		229
S6	540	1600	1280	6	21	210		319

TABLE 3.7 – Résultats d'implantation sur FPGA Spartan-6 (S6), Virtex-4 (V4) et Virtex-5 (V5) de la solution MA16 reproduite à partir de [MLPJ13] pour des premiers P de 128-bits. Les BRAM ont des tailles de 9 kbits sur S6 et de 18 kbits sur V4 et V5. Le temps est donné pour 3 MMM indépendantes.

Pour le calcul de 1 MMM, notre HTMM est moins performant que MA16 avec une latence λ de 69 cycles pour les versions HTMMb et HTMMd contre 27 cycles pour MA16. En revanche, le calcul de 3 MMM indépendantes dans notre HTMM (c.-à-d. quand tous les LM sont remplis) nécessite 79 cycles contre 65 cycles pour MA16 tout en utilisant presque 2 fois moins de *slices* DSP.

Grâce à la décomposition des opérandes et des résultats intermédiaires dans le pipeline interne du HTMM, les fréquences atteignables dans ce dernier sont supérieures à celles de MA16 : +37% dans la version HTMMd sur Virtex-5 par exemple. Cela permet aux versions HTMMb et HTMMd de proposer des temps de calcul inférieurs à ceux de MA16 quand plusieurs MMM indépendantes peuvent être calculées en même temps.

La première version du HTMM proposée dans [GT17d] permet d'atteindre de meilleures performances et de meilleurs compromis en termes de temps de calcul et de coût en surface que les multiplieurs modulaires de l'état de l'art pour des P génériques de 128 bits. Par exemple, la comparaison des résultats d'implantation sur Spartan-6 dans les tableaux 3.6 et 3.7 montre que la version HTMMb permet une réduction du temps de calcul de 15%, du nombre de *slices* DSP de 48%, du nombre de BRAM de 66% et du nombre de *slices* logiques de 33% par rapport à l'implantation de MA16 sur le même FPGA et pour les mêmes tailles de P. La version HTMMd implantée sur le même FPGA permet quant à elle la même réduction du temps de calcul et du nombre de *slices* DSP que pour HTMMb. La réduction du nombre de *slices* logiques comparé à MA16 est de seulement 10% pour HTMMd mais cette version n'utilise plus aucune BRAM.

3.6 Améliorations du HTMM proposées dans [GT18a]

Comme nous l'avons vu dans la section 3.5, notre premier HTMM présenté dans [GT17d] était *implanté à la main* sur des FPGA Virtex-4 et 5 et Spartan-6 pour des premiers P de 128 bits fixés à l'implantation et avec les paramètres internes $w = 34$ bits, $s = 4$ mots et $\sigma = 3$ LM.

Dans ces travaux, nous avions dû limiter notre exploration à ces seuls paramètres. En effet, l'exploration *à la main* des nombreux paramètres internes de nos HTMM pour des tailles de premiers différentes et des implantations sur un plus grand nombre de FPGA n'était pas possible en pratique.

Les bonnes performances du HTMM de [GT17d] nous ont toutefois encouragé à approfondir cette exploration. Nous avons donc décidé de mettre au point un outil logiciel nous permettant de rapidement *générer des codes* VHDL pour des HTMM utilisant *différentes spécifications de paramètres*. La section 3.7 est consacrée à la description de cet outil, grâce auquel nous avons pu explorer et proposer dans [GT18a] un grand nombre de HTMM différents et profitant de nombreuses améliorations.

Parmi ces améliorations, nous avons *réduit le nombre de slices DSP* de 11 à 9 dans nos HTMM en modifiant la structure du bloc 2. Cette amélioration est expliquée en section 3.6.1. Nous avons aussi proposé une optimisation de l'ordonnancement des calculs dans le HTMM, présentée en section 3.6.2, qui nous a permis de *réduire la latence* λ dans nos multiplieurs. Les modifications de la structure du HTMM nous ont permis de mettre en place un contrôle interne plus régulier et de mieux pipeliner certaines opérations. La fréquence de fonctionnement de l'unité a ainsi pu être augmentée, avec des gains allant jusqu'à +20% pour certains HTMM par rapports aux fréquences rapportées dans [GT17d].

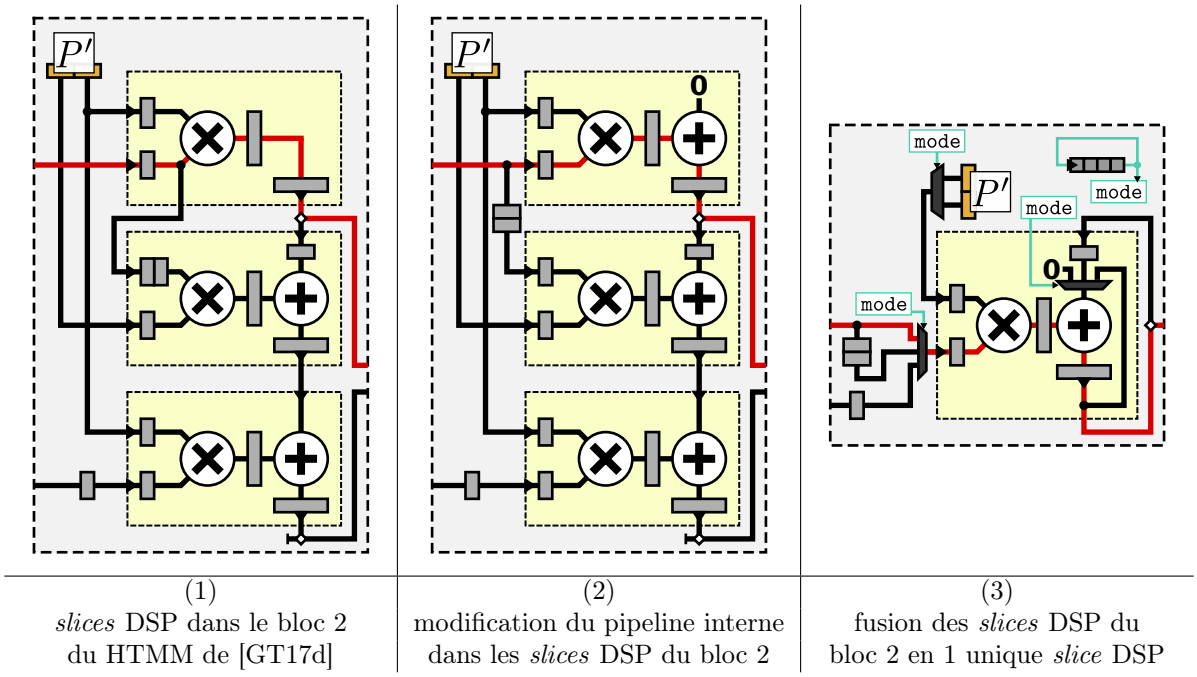
Dans le but d'évaluer l'impact de la configuration des *slices* DSP sur les performances, nous avons aussi proposé une architecture alternative pour le bloc 1 du HTMM. Dans cette nouvelle architecture, détaillée en section 3.6.3, nous avons supprimé un étage de pipeline dans le 1$^{\text{er}}$ *slice* DSP, nous permettant ainsi de réduire la latence dans le bloc ainsi que sa surface.

Enfin, nous avons modifié la gestion du premier P dans le HTMM afin de le rendre *modifiable à l'exécution*. On notera toutefois que seule la valeur du premier P peut être modifiée à l'éxécution. La taille maximale de P est, elle, fixée lors de l'implantation. Le nouveau mode de gestion de P dans HTMM sera présenté en section 3.6.4.

Notre outil de génération nous a permis d'explorer un grand nombre de spécifications des paramètres pour les nouveaux HTMM proposés. En particulier, nous avons pu illustrer le fonctionnement de notre HTMM pour différentes tailles de premiers. Nous avons ainsi proposé des versions du HTMM pour des premiers de 256 bits en plus de ceux utilisant des premiers de 128 bits. Nous rappelons que la taille maximale de P dans HTMM est définie *à l'implantation* et qu'elle ne peut pas être modifiée à l'exécution. Dans ces HTMM, nous utilisons toujours un découpage en mots de $w = 34$ bits. Pour cette raison, le nombre de *slices* DSP et de BRAM utilisés est le même dans le HTMM 128 bits et dans le HTMM 256 bits. Le paramètre s est quant à lui impacté par la modification du paramètre n et on a $s = 4$ pour $n = 128$ bits et $s = 8$ pour $n = 256$ bits. Notre outil nous a aussi permis de générer facilement des codes VHDL pour des implantations sur d'autres FPGA. Nous avons ainsi pu proposer dans [GT18a] des exemples de résultats obtenus après implantation sur un FPGA Virtex-7 récent. Enfin, nous avons aussi évalué l'impact sur les performances de calcul et la consommation de surface de différentes spécifications pour le paramètre σ dans nos HTMM 128 et 256 bits.

3.6.1 Réduction du nombre de *slices* DSP dans le bloc 2

Le bloc matériel dédié à la tâche 2 dans le HTMM calcule le « quotient de réduction » q_i dans l'algorithme de Montgomery à chaque itération de la boucle externe (ligne 7 dans l'algorithme 14). Chaque q_i est calculé grâce à un produit partiel de $w \times w$ bits réduit modulo 2^w. Comme nous l'avons vu dans la section précédente, seuls 3 *slices* DSP sont nécessaires au calcul de ce produit quand $w = 34$ bits en raison de la réduction.



(1)	(2)	(3)
slices DSP dans le bloc 2 du HTMM de [GT17d]	modification du pipeline interne dans les *slices* DSP du bloc 2	fusion des *slices* DSP du bloc 2 en 1 unique *slice* DSP

FIGURE 3.13 – Étapes de la modification du bloc 2 pour la réduction du nombre de *slices* DSP dans le HTMM.

Après une analyse fine de l'ordonnancement des opérations internes dans les *slices* DSP du bloc 2, nous avons réalisé qu'il était possible de calculer le produit partiel réduit $(u_0 \times P') \bmod 2^w$ en modifiant la structure du bloc 2 pour n'utiliser qu'un seul *slice* DSP. Nous illustrons les différentes étapes de la modification apportées au bloc 2 en figure 3.13. Comme le montre l'étape 1 en figure 3.13, les 3 *slices* DSP du bloc 2 dans le HTMM de [GT17d] utilisent des configurations différentes de leurs pipelines internes. Dans les FPGA Xilinx, il est possible de modifier le *mode de fonctionnement* des *slices* DSP *à l'exécution*. Il n'est toutefois pas possible de modifier leurs pipelines internes sans reconfigurer le FPGA. Afin de pouvoir fusionner les 3 *slices* DSP du bloc 2, nous avons tout d'abord modifié leur configuration interne comme l'illustre l'étape 2 en figure 3.13. Après cette étape, nos différents *slices* DSP utilisent des pipelines similaires de 3 étages entre les entrées du multiplieur interne et le registre de sortie. L'étape 3 de la figure 3.13 illustre la nouvelle configuration du bloc 2 après fusion des 3 *slices* DSP en un unique *slice* DSP. Le signal `mode` dans la nouvelle configuration du bloc 2 est un signal de 2 bits permettant de sélectionner à chaque cycle l'opération calculée dans le bloc. Il est généré par une petite FSM implantée sous forme de *buffer* circulaire de 2 bits de large.

Cette optimisation permet d'économiser 2 *slices* DSP dans le bloc 2 au prix de quelques registres supplémentaires (moins d'une cinquantaine de bits au total), d'un petit multiplexeur 3 entrées de 17 bits et d'un petit multiplexeur 2 entrées de 17 bits. Grâce à la mise en place d'un pipeline efficace dans le

bloc 2, la fréquence du HTMM n'est pas réduite après implantation de cette optimisation.

L'architecture du HTMM 128 bit pour $\sigma = 4$ LM et $s = 4$, obtenue après réduction du nombre de *slices* DSP et réduction de la latence (voir l'optimisation proposée dans la section suivante), est illustrée en figure 3.14.

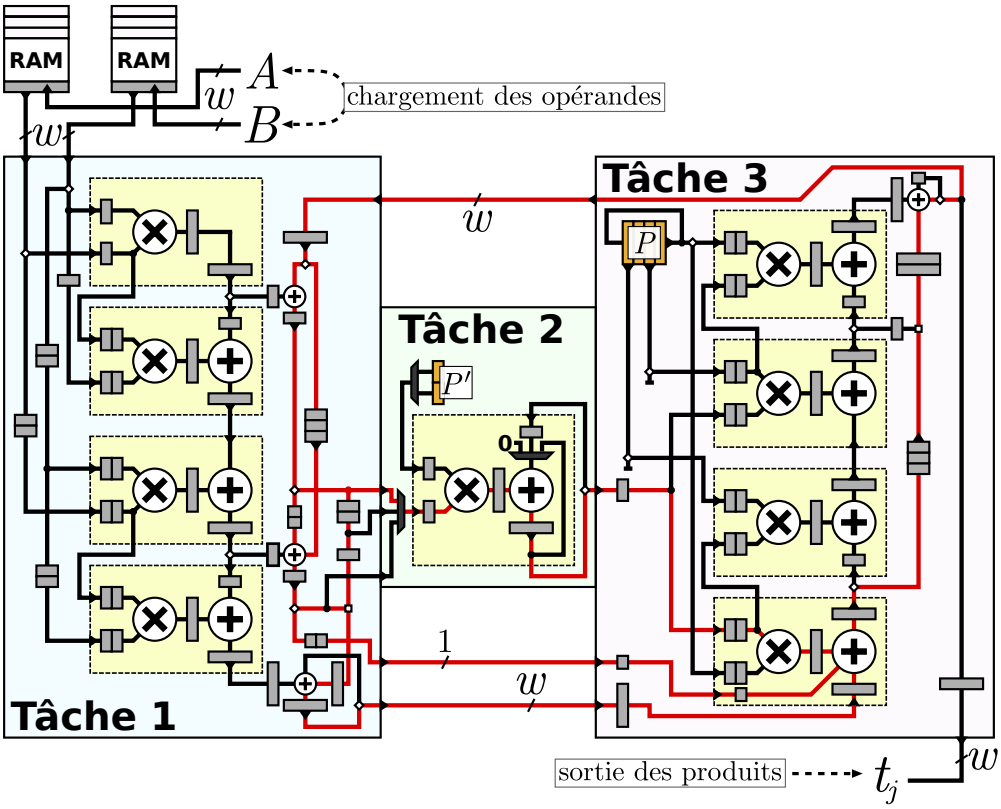


FIGURE 3.14 – Architecture du HTMM 128 bits de [GT18a] pour $\sigma = 4$ et $s = 4$ après réduction du nombre de *slices* DSP (sans le détail du contrôle).

3.6.2 Réduction de la latence de la MMM

Dans notre premier HTMM de [GT17d], les boucles internes dans les tâches 1 et 3 nécessitent chacune s cycles pour charger les s mots de l'opérande B et du premier P dans les *slices* DSP des blocs correspondants. Un cycle supplémentaire est nécessaire pour la propagation des retenues dans les mots de poids forts des résultats de la tâche 1 (c.-à-d. u_s à la ligne 6 de l'algorithme 14) et de la tâche 3 (c.-à-d. t_{s-1} à la ligne 11 de l'algorithme 14). Ce cycle supplémentaire introduit une « bulle » dans le pipeline des *slices* DSP entre les calculs de 2 MMM consécutives dans deux LM.

Dans [GT18a], nous avons modifié le contrôle du HTMM afin de supprimer cette « bulle » en permettant aux retenues propagées dans les mots de poids forts d'une MMM de « déborder » sur les mots de poids faibles calculés dans la MMM suivante, comme illustré dans le bas de la figure 3.15.

La figure 3.15 illustre l'enchaînement au sein du HTMM 128 bits des mots calculés dans les itérations des boucles internes j du CIOS. Les rectangles correspondent aux mots de w calculés pour 2 MMM consécutives dans 2 LM, respectivement coloriées en bleu et en rouge. Les itérations des boucles dans chaque LM sont données par les indices j et les sigles LSW et MSW indiquent respectivement le mot de

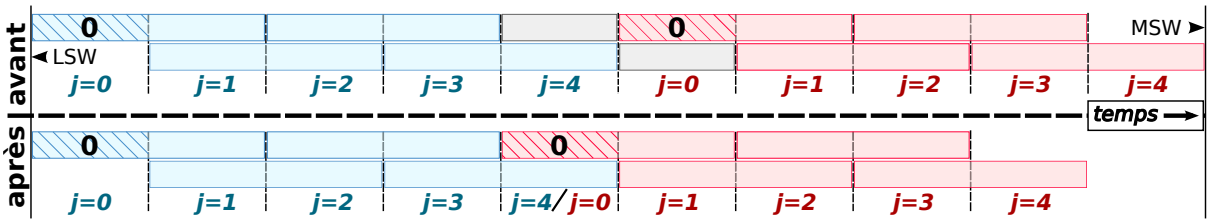


FIGURE 3.15 – Exemple d'enchaînement des mots calculés aux itérations d'indice j des boucles internes de 2 MMM, *avant* et *après* la réduction de latence dans le HTMM 128 bits ($s = 4$ mots).

poids faible pour la 1re MMM et le mot de poids fort pour la 2e MMM

La partie haute de la figure correspond à la version originale sans réduction de latence. La « bulle » introduite par la propagation de retenue dans les mots de poids forts des résultats intermédiaires est représentée en gris dans ce schéma.

La partie basse de la figure correspond à l'enchaînement des mots après la réduction de latence dans la version optimisée du HTMM 128 bits. Dans cette version, les résultats t_{s-1} et t_{-1} calculés respectivement dans le 1er LM à l'itération $j = 4$ et dans le 2e LM à l'itération $j = 0$ *partagent le même mot* de w bits.

Cette optimisation est possible car le mot de poids faible t_{-1} du résultat est toujours égal à 0 et supprimé à la fin des itérations de la boucle externe d'indice i du CIOS. Durant ces itérations dans un LM, le mot de poids faible t_{-1} peut donc être utilisé pour stocker des valeurs intermédiaires pour le LM suivant. Pour ce faire, nous avons légèrement modifié le comportement des boucles internes d'indice j dans l'algorithme du CIOS. Notre version modifiée est présentée dans l'algorithme 19.

Dans cet algorithme ainsi que dans les démonstrations suivantes, les variables marquées par $^{(\text{prev})}$, $^{(\text{curr})}$ ou $^{(\text{next})}$ désignent des valeurs intermédiaires calculées respectivement dans le LM précédent, courant ou suivant.

Opérandes : $A = \sum_{i=0}^{s-1} a_i\, 2^{iw}$, $B = \sum_{j=0}^{s-1} b_j\, 2^{jw}$, $P = \sum_{j=0}^{s-1} p_j\, 2^{jw}$ tels que $0 \leq A, B < 2P$
Prérequis : $m = s \times w$, $2^m > 4P$ et $P' = -P^{-1} \bmod 2^w$
Résultat : $T \equiv (AB \times 2^{-m}) \bmod P$, $0 \leq T < 2P$
begin

```
1  t_{0...s-1} ← 0; d ← 0; c ← 0
2  for i = 0 to s − 1 do
3      for j = 0 to s − 1 do
4          v_j ← t_0 + a_i × b_0
5          (d, u_j) ← v_j + d
6      q_i ← (v_0 × P') mod 2^w
7      for j = 0 to s − 1 do
8          (c, t_{j−1}) ← u_j + q_i × p_j + c
9  return T = t_{−1}^{(next)} 2^{(s−1)w} + Σ_{j=0}^{s−2} t_j 2^{jw}
```

Algorithme 19 : Algorithme CIOS pour la MMM, basé sur [KAK96] et [Wal99a] et modifié pour intégrer l'optimisation de latence proposée dans [GT18a].

Démonstration et validation de l'optimisation proposée

Pour valider cette optimisation, nous allons montrer que le fait que partager un mot de w bits entre 2 LM successifs ne modifie pas les résultats des MMM calculées dans ces LM.

En particulier, nous allons vérifier que $t_{-1}^{(\text{next})} = t_{s-1}^{(\text{curr})} < 2^w$ grâce aux propriétés de l'algorithme de la MMM ; et donc que permettre le débordement du mot de poids fort t_{s-1} du LM courant sur le mot de poids faible t_{-1} du LM suivant ne modifie pas le résultat calculé dans ce dernier.

Dans l'algorithme 19, les retenues d et c sont propagées entre les LM successifs. Afin de clarifier les explications, nous noterons u_s la retenue d engendrée par l'itération $j = s - 1$ de la 1re boucle interne et $t_{s-1} = u_s + c$ le résultat de l'itération $j = s - 1$ de la 2e boucle interne dans l'algorithme 19.

Le calcul de T_i dans le LM courant requière la valeur $t_{-1}^{(\text{next})}$ calculée à l'itération $j = 0$ de la 2e boucle interne du LM suivant :

$$t_{-1}^{(\text{next})} = \left(u_0^{(\text{next})} + q_i^{(\text{next})} \times p_0 + c^{(\text{curr})}\right) \bmod 2^w. \tag{3.2}$$

Nous allons à présent montrer que les valeurs $t_{-1}^{(\text{next})}$ et $t_{s-1}^{(\text{curr})} = u_s^{(\text{curr})} + c^{(\text{curr})}$ sont bien égales. Pour cela nous allons dans un premier temps calculer la valeur $t_{-1}^{(\text{next})}$ à partir des valeurs $u_0^{(\text{next})}$, $q_i^{(\text{next})}$ et $c^{(\text{curr})}$. À cause de la propagation du mot u_s entre les LM successifs, nous avons

$$u_0^{(\text{next})} = \left(T_{i-1}^{(\text{next})} + a_i^{(\text{next})} \times b_0^{(\text{next})} + u_s^{(\text{curr})}\right) \bmod 2^w. \tag{3.3}$$

Le quotient de réduction q_i est, lui, calculé *avant la propagation* des mots u_s. Sa valeur est donc

$$q_i^{(\text{next})} = \left((T_{i-1}^{(\text{next})} + a_i^{(\text{next})} \times b_0^{(\text{next})} \times P'\right) \bmod 2^w. \tag{3.4}$$

La valeur $t_{-1}^{(\text{next})}$ dans l'équation 3.2 peut être réécrite en utilisant les équations 3.3 et 3.4 :

$$\begin{aligned} t_{-1}^{(\text{next})} &= \left(T_{i-1}^{(\text{next})} + a_i^{(\text{next})} \times b_0^{(\text{next})} + u_s^{(\text{curr})} + q_i^{(\text{next})} \times p_0 + c^{(\text{curr})}\right) \bmod 2^w \\ &= \left(\left(T_{i-1}^{(\text{next})} + a_i^{(\text{next})} \times b_0^{(\text{next})} + q_i^{(\text{next})} \times p_0\right) + \left(u_s^{(\text{curr})} + c^{(\text{curr})}\right)\right) \bmod 2^w. \end{aligned} \tag{3.5}$$

Dans l'algorithme du CIOS, le mot de poids faible du résultat intermédiaire $T_i = T_{i-1} + a_i \times b_0 + q_i \times p_0$ à l'itération i de la boucle externe est égal à 0 par construction de la multiplication de Montgomery. Dans le LM suivant, on a donc

$$\left(T_{i-1}^{(\text{next})} + a_i^{(\text{next})} \times b_0^{(\text{next})} + q_i^{(\text{next})} \times p_0\right) \bmod 2^w = 0,$$

la valeur $t_{-1}^{(\text{next})}$ dans l'équation 3.5 pouvant alors être exprimée sous la forme

$$t_{-1}^{(\text{next})} = \left(u_s^{(\text{curr})} + c^{(\text{curr})}\right) \bmod 2^w.$$

La valeur $u_s^{(\text{curr})}$ est calculée de la façon suivante :

$$u_s^{(\text{curr})} = \left\lfloor \frac{T_{i-1}^{(\text{curr})} + a_i^{(\text{curr})} \times B^{(\text{curr})} + u_s^{(\text{prev})}}{2^m} \right\rfloor.$$

D'après les propriétés de la MMM de l'algorithme 19, on a

$$\begin{cases} 4P < 2^m \\ 0 \leq B, T_{i-1} < 2P < 2^{m-1} \\ 0 \leq a_i < 2^w \end{cases} \tag{3.6}$$

La valeur $u_s^{(\text{curr})}$ est alors bornée par

$$0 \leq u_s^{(\text{curr})} < 2^{w-1}. \tag{3.7}$$

De façon similaire, la valeur de $c^{(\text{curr})}$ est

$$c^{(\text{curr})} = \left\lfloor \frac{\sum_{j=0}^{s-1} \left(u_j^{(\text{curr})} 2^{jw} \right) + u_s^{(\text{prev})} + q_i^{(\text{curr})} \times P + c^{(\text{prev})}}{2^m} \right\rfloor.$$

À partir des bornes de $u_s^{(\text{curr})}$ données en équation 3.7, il est possible de déterminer les bornes de $c^{(\text{curr})}$:

$$0 \leq c^{(\text{curr})} \leq 2^{w-2}. \tag{3.8}$$

Il est alors possible de borner la valeur de $t_{-1}^{(\text{next})}$ à partir des résultats des équations 3.7 et 3.8 :

$$0 \leq t_{-1}^{(\text{next})} < 2^w.$$

Nous avons alors

$$\begin{aligned} t_{-1}^{(\text{next})} &= \left(u_s^{(\text{curr})} + c^{(\text{curr})} \right) \bmod 2^w \\ &= u_s^{(\text{curr})} + c^{(\text{curr})} \\ &= t_{s-1}^{(\text{curr})} \end{aligned}$$

Le résultat $T^{(\text{curr})}$ de l'algorithme 19 pour la MMM calculée dans le LM courant est donc

$$\begin{aligned} T^{(\text{curr})} &= t_{-1}^{(\text{next})} 2^{(s-1)w} + \sum_{j=0}^{s-2} t_j^{(\text{curr})} 2^{jw} \\ &= t_{s-1}^{(\text{curr})} 2^{(s-1)w} + \sum_{j=0}^{s-2} t_j^{(\text{curr})} 2^{jw} \\ &= \sum_{j=0}^{s-1} t_j^{(\text{curr})} 2^{jw}. \end{aligned}$$

Les résultats de MMM successives calculées dans différents LM de la version optimisée du HTMM sont donc égaux à ceux calculés pour les mêmes MMM dans la version non optimisée du HTMM utilisant l'algorithme classique du CIOS présenté en début de chapitre (cf. algorithme 14). □

Grâce à cette optimisation, nous avons pu supprimer la « bulle » engendrée dans le HTMM de [GT17d] par la propagation des retenues dans les mots de poids fort des résultats des boucles internes du CIOS. Le nombre de cycles θ entre 2 MMM consécutives calculées dans 2 LM différents du HTMM optimisé est alors de 4 cycles d'horloge pour le HTMM 128 bits, au lieu de 5 cycles dans le HTMM de [GT17d]

sans réduction de latence. Il est de 8 cycles pour la version optimisée du HTMM 256 bits et de 9 cycles dans la version de ce HTMM ne profitant pas de cette réduction de la latence. Suite à cette amélioration, nous avons $\theta = s$, ce qui est la valeur optimale atteignable. En effet, étant donné qu'au moins s cycles sont nécessaires pour le chargement des opérandes dans le HTMM, il n'est pas possible de réduire plus encore θ.

Pour mettre en place cette optimisation, nous avons dû modifier le contrôle dans nos HTMM. Par chance, cette modification nous a permis de mettre en place un contrôle plus régulier dans lequel les produits partiels dans les blocs 1 et 3 sont calculés sans interruption. Elle ne provoque par ailleurs pas de surcoût mesurable en terme de surface du FPGA consommée.

On notera enfin que la réduction de latence impacte aussi la configuration de *l'hyper-threading* dans nos HTMM et le nombre minimum de LM nécessaire pour remplir efficacement le pipeline interne du HTMM optimisé est à présent $\sigma = \lceil \alpha/s \rceil$ cycles au lieu de $\sigma = \lceil \alpha/(s+1) \rceil$.

3.6.3 Impact de la configuration des *slices* DSP sur les performances du HTMM

D'après la documentation des FPGA Xilinx, utiliser moins de 3 étages de registres dans le pipeline interne des *slices* DSP entraîne une diminution de la fréquence maximale atteignable dans ces blocs matériels câblés. Pour cette raison, l'ensemble des *slices* DSP dans nos HTMM sont configurés avec un pipeline interne d'au moins 3 étages, comme illustré dans les schémas d'architecture en figures 3.11 et 3.14.

On peut cependant constater en étudiant ces schémas que l'additionneur 48 bits câblé dans le 1$^{\text{er}}$ *slice* DSP du bloc 1, situé en haut à gauche, n'est pas utilisé.

Supprimer l'étage de pipeline intermédiaire dans ce *slice* DSP permet de réordonnancer les 4 produits 17×17 bits calculés dans les *slices* DSP du bloc 1 de façon à réduire de 1 cycle le temps de calcul des produits partiels $a_i \times B$. Ce nouvel ordonnancement permet aussi de réduire de 1 cycle les délais en entrée des 2$^{\text{e}}$, 3$^{\text{e}}$ et 4$^{\text{e}}$ *slices* DSP du bloc 1 et donc de supprimer 4 registres de w bits dans le HTMM.

Cependant, la fréquence atteignable dans ce *slice* DSP est réduite, et l'ampleur de cette réduction dépend du FPGA choisi pour l'implantation.

Ainsi, supprimer un étage de pipeline dans le 1$^{\text{er}}$ *slices* DSP du bloc 1 permet de réduire la latence et la surface du HTMM mais peut aussi provoquer une diminution de la fréquence globale dans l'unité.

Il est en pratique impossible de déterminer laquelle des configurations des *slices* DSP du bloc 1 permet d'obtenir le meilleur compromis en terme de surface du FPGA consommée et de temps de calcul dans le HTMM. Nous avons donc décidé de proposer 2 variantes du HTMM, avec 2 configurations du 1$^{\text{er}}$ *slice* DSP du bloc 1 :

— la *version rapide* avec 3 étages de pipeline pour atteindre des fréquences plus élevées ;
— la *petite version* avec 2 étages de pipeline pour réduire la latence de 1 cycle et la surface du FPGA consommée.

Dans la section 3.8, nous verrons que chacune de ces variantes a un intérêt en fonction du FPGA et des paramètres du HTMM choisis pour implantation.

3.6.4 Prise en charge de la modification du premier P à l'exécution

Contrairement à la plupart des multiplieurs modulaires utilisés les implantations ECC et HECC de la littérature, et contrairement au HTMM de [GT17d], les HTMM proposés dans [GT18a] permettent la

modification du premier P *à l'exécution*.

Dans ces HTMM, les paramètres n (taille du premier), m (taille des opérandes), w et s sont fixés lors de la conception mais P et P' peuvent être définis et chargés dans l'unité durant l'exécution sur le FPGA, sans avoir besoin de reconfigurer ce dernier. Ils possèdent 2 modes de fonctionnement, *setup* et *run*, sélectionnés par l'intermédiaire du signal de contrôle externe `set` de 1 bit.

Lorsque le signal `set` est activé, le HTMM rentre en mode *setup*. Lors de l'entrée en *setup*, les LM du HTMM sont remis à zéro et toute MMM en cours de calcul est stoppée et ignorée. Les s mots de P sont ensuite chargés séquentiellement en s cycles dans le *buffer* circulaire du bloc 3 depuis le port d'entrée de l'opérande A du HTMM. Les w bits de P' sont eux chargés dans le registre dédié du bloc 2 depuis le port d'entrée de l'opérande B du HTMM au même cycle que le mot de poids faible p_0 de P.

Une fois les s mots de P chargés dans le HTMM, l'unité quitte le mode *setup* et retourne en mode *run*, qui est son mode de fonctionnement normal. En mode *run*, le HTMM peut donc calculer de nouvelles multiplications modulo le premier P défini lors du dernier *setup* ou retourner en mode *setup* si le signal `set` est à nouveau activé.

Cette modification du HTMM a seulement nécessité l'ajout de quelques LUT et FF, ainsi que l'ajout d'un port d'entrée de 1 bit supplémentaire dans l'interface de l'unité. Elle n'a entraîné aucune diminution de la fréquence du HTMM.

3.6.5 Validation du HTMM

Toutes les versions des HTMM 128 bits et 256 bits que proposées dans [GT18a] ont été validées au niveau arithmétique et au niveau de l'architecture et des implantations sur FPGA.

Au niveau arithmétique, nous avons modifié l'algorithme original du CIOS de [KAK96] utilisant l'optimisation de [Wal99a] pour permettre la réduction de latence détaillée en section 3.6.2. La validité de cette modification a été étudiée dans cette même section.

Au niveau de l'architecture, nous avons validé l'ensemble des implantations des différentes variantes du HTMM sur les différents FPGA sélectionnés au moyen de simulations intensives de calculs de MMM dans les unités. Durant ces simulations, nous avons calculé les résultats de MMM obtenus dans nos unités pour un ensemble de vecteurs de test prédéfinis ainsi que pour plusieurs millions de vecteurs de test choisis aléatoirement. Ces résultats ont été validés après comparaison avec les résultats théoriques calculés pour l'ensemble de ces vecteurs de test grâce à l'outil mathématique *Sage* (cf. section 3.7).

3.7 Générateur de HTMM pour différents jeux de paramètres

Pour faciliter l'exploration de l'espace de conception pour nos multiplieurs, nous avons développé un outil pour la génération automatique d'un grand nombre de HTMM avec différentes spécifications des paramètres internes. Ce générateur est basé sur un ensemble de scripts `Bash` et de programmes `Python`. Il est disponible en *open source* pour les plateformes Unix depuis [GT18b].

La génération d'un HTMM se fait à partir d'une *spécification des paramètres* du multiplieur fournie sous forme de fichier texte en entrée du générateur. Cette spécification permet de fixer les valeurs choisies pour les différents paramètres du HTMM :

— taille m des opérandes ;

— nombre s de mots de $w = 34$ bits (p. ex. 4 pour 128 bits et 8 pour 256 bits) ;

— nombre σ de LM ;
— type de mémoire pour le stockage des opérandes (BRAM ou DRAM) ;
— utilisation ou non de la réduction de latence ;
— configuration des *slices* DSP (version *rapide* ou *petite*) ;
— FPGA choisi pour l'implantation.

À partir de ce fichier texte, le générateur effectue un ensemble de vérifications afin de valider la cohérence de la spécification. Cette première étape permet aussi à l'outil de calculer et de rapporter certaines propriétés du HTMM telles que la latence λ de la MMM ou l'intervalle minimum τ entre 2 MMM calculées dans un LM.

Si la spécification des paramètres est validée par le générateur, celui-ci produit un ensemble de fichiers VHDL pouvant directement être utilisés pour l'implantation du HTMM spécifié sur le FPGA sélectionné.

Pour aider l'exploration de l'espace de conception, nous fournissons aussi dans [GT18b] un ensemble de scripts permettant d'automatiser les étapes de synthèse, d'implantation et de validation dans les outils de CAO propriétaires. Le flot complet pris en charge par ces différents scripts est illustré en figure 3.16. Il supporte à l'heure actuelle les outils Xilinx suivant : ISE 14.7 pour la synthèse et l'implantation, le simulateur ISim pour la validation après synthèse et SmartXplorer pour le placement–routage.

Les étapes 1 et 2 en figure 3.16 correspondent respectivement à la spécification des paramètres par l'utilisateur et à la génération des codes VHDL.

La synthèse et le placement–routage du HTMM sur le FPGA sélectionné sont effectués dans l'étape 3. Durant cette étape, nous utilisons l'outil SmartXplorer pour sélectionner l'implantation permettant d'atteindre la fréquence la plus haute après de nombreuses exécutions de l'outil de placement–routage (c.-à-d. 100 exécutions par défaut).

Durant l'étape 4, le comportement fonctionnel du HTMM produit par le générateur est validé après synthèse grâce à des simulations intensives pour des millions de paires d'opérandes tirées au hasard et sélectionnées au préalable. Les résultats de synthèse sont validés par comparaison avec les résultats théoriques attendus. Ces derniers sont calculés pour chacune des nombreuses paires d'opérandes en utilisant le logiciel mathématique *Sage*, disponible en *open source* depuis `http://www.sagemath.org/`.

Finalement, la dernière étape, numérotée 5 dans le flot de conception illustré en figure 3.16, engendre un rapport final regroupant les résultats d'implantation : surface du FPGA consommée (p. ex. nombre de LUT, de FF, de BRAM ou de DSP) et fréquence maximale atteignable dans l'implantation. Ce rapport final inclut aussi des informations issues de l'étape de validation comme les traces d'exécution obtenues après simulation par exemple.

3.8 Résultats d'implantation sur FPGA et comparaisons

Pour évaluer et comparer différents jeux de paramètres, nous avons implanté un grand nombre de HTMM sur les FPGA sélectionnés : Virtex-4 XC4VLX100 (dénoté V4), Virtex-5 XC5VLX110T (V5), Virtex-7 XC7VX690T (V7) et Spartan-6 XC6SLX75 (S6). Les résultats d'implantation des HTMM 128 bits et 256 bits sont détaillés ci-dessous en nombre de *slices* DSP, de *slices* logiques, de LUT, de FF et de BRAM pour les métriques de surface. Pour les performances des HTMM implantés, nous rapportons les latences λ en cycles pour 1 MMM, les fréquences de fonctionnement des unités et les temps de calcul pour 8 MMM indépendantes. Le nombre de 8 MMM est choisi afin d'illustrer les bénéfices de *l'hyper-threading* pour le calcul de `xDBLADD` dans KHECC. En effet, on retrouve dans cette opération des

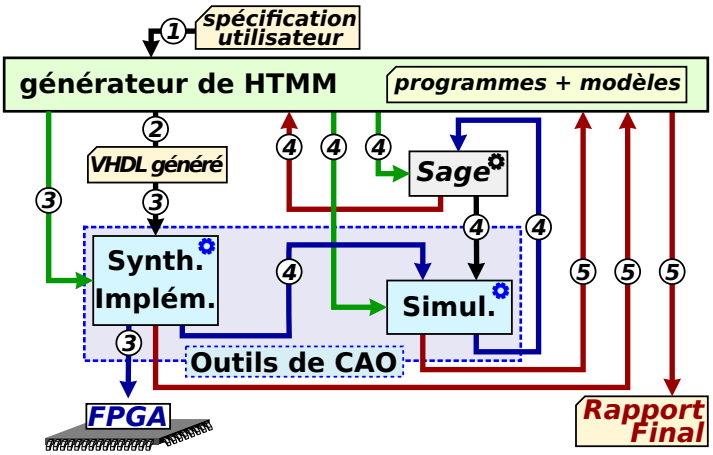


FIGURE 3.16 – Générateur de HTMM en vert et son flot d'utilisation. Les outils de CAO propriétaires sont en bleu et le programme Sage pour la validation mathématique en gris.

ensembles de 8 MMM indépendantes pouvant être calculées en parallèle (voir la section 4.2.2 du chapitre suivant).

Chaque spécification des paramètres est abrégée sous forme d'un sigle composé de 4 caractères :

— 1 lettre : « F » ou « S » pour les versions *rapide* ou *petite* respectivement ;

— 1 chiffre : σ choisi dans $\{3, 4\}$ pour les HTMM 128 bits ou dans $\{2, 3, 4\}$ pour les HTMM 256 bits ;

— 1 chiffre : $\theta = s$ ou $s + 1$ cycles pour le délai entre 2 LM avec ou sans réduction de latence ;

— 1 lettre : « B » ou « D » pour le type de mémoire BRAM ou DRAM utilisé.

Par exemple, F45B désigne la version rapide du HTMM avec $\sigma = 4$ LM, sans réduction de latence ($\theta = s+1 = 5$) et intégrant des mémoires à base de BRAM pour les opérandes. Le 2e chiffre « 5 » indique d'ailleurs que $n = 128$ bits dans le HTMM car $s = 4$ mots.

En figure 3.17 nous illustrons les compromis surface – temps pour l'ensemble des HTMM 128 bits implantés sur V4 et V7. Dans cette figure, les hexagones désignent les spécifications utilisant les DRAM et les carrés celles à base de BRAM. Les versions rapides (F) des HTMM sont indiquées en bleu et les petites versions (S) en rouge. L'ensemble des compromis surface – temps pour les différentes spécifications des HTMM 128 bits et 256 bits implantées sur les différents FPGA considérés peut être trouvé en section annexe 3.10 de ce chapitre.

D'après la figure 3.17, le plus petit HTMM sur V7 est S44B tandis que F44B est le plus rapide utilisant des BRAM et F44D le plus rapide utilisant de la DRAM. Sur V4, S44B est toujours le HTMM le plus petit et F44B le plus rapide mais F44D n'est plus sur la frontière de Pareto. Lors de l'exploration des HTMM sur différentes générations de FPGA, il est très difficile de prédire l'impact de certains paramètres sur les performances de calcul et la consommation de surface. Par exemple, sur un FPGA V7 récent, les HTMM les plus rapides utilisent des mémoires de type DRAM alors que sur un V4 plus ancien ils utilisent des BRAM. Le choix du FPGA impacte aussi grandement les différences de surfaces entre les différents HTMM implantés. Ainsi, sur V7 utilisant des LUT-6, les variations de surface en nombre de LUT entre les HTMM sont seulement de 26% alors qu'elles sont de 116% sur V4 utilisant des LUT-4. L'étude des compromis pour V4 et V7 confirme donc l'intérêt pratique de notre générateur de HTMM sans lequel il serait difficile de déterminer la meilleure spécification de paramètres pour une application donnée. La génération automatique nous permet ainsi d'explorer un grand nombre de spécifications et

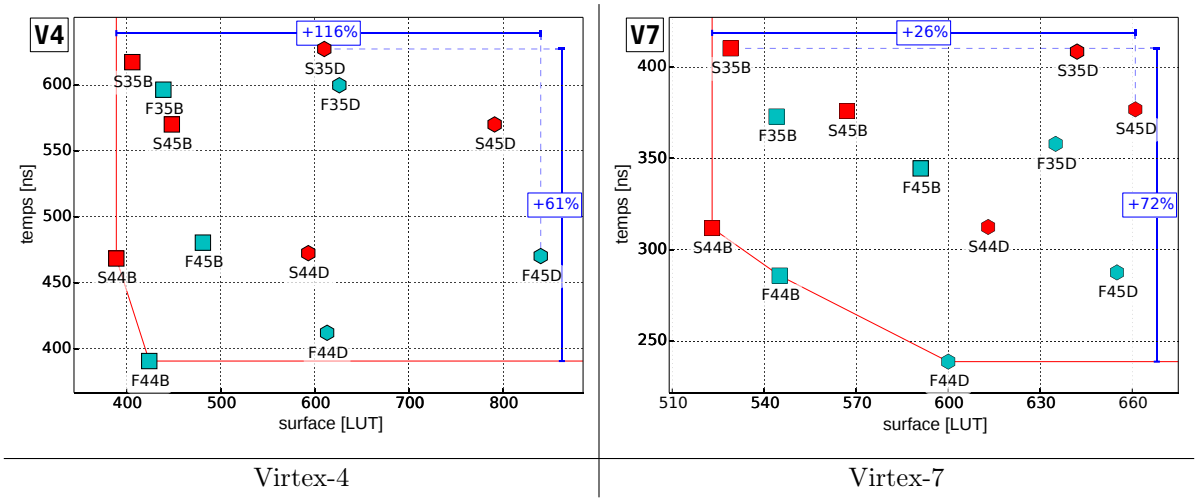


FIGURE 3.17 – Temps de calcul pour 8 MMM dans les HTMM 128 bits implantés sur V4 (à gauche) et sur V7 (à droite) en fonction du nombre de LUT consommées. Les lignes rouges indiquent les frontières de Pareto des espaces de compromis.

de compromis pour un FPGA donné. À partir de cette exploration, nous sommes en mesure de proposer le HTMM le mieux adapté aux besoins de l'application. Par exemple sur V7, on utilisera le petit HTMM S44B quand la surface est limitée dans l'application alors que le HTMM F44D sera plus adapté pour des applications nécessitant de bonnes performances de calcul.

Le tableau 3.8 liste les résultats d'implantation des HTMM 128 bits et 256 bits les plus intéressants. Ces derniers ont été sélectionnés après étude des compromis surface – temps sur les FPGA V4, V5, S6 et V7 pour différentes métriques de surface : nombres de *slices* logiques, de LUT et de FF.

Pour chaque HTMM, les valeurs en gras indiquent les métriques de surface pour lesquelles celui-ci fait partie des meilleurs compromis, c'est-à-dire les métriques pour lesquelles le compromis obtenu est sur la frontière de Pareto. La capacité des BRAM est de 9 kbits, notée $^{\ominus}$, sur S6 et de 18 kbits, notée $^{\oplus}$, sur les autres FPGA.

Le tableau 3.9 liste les résultats d'implantation des multiplieurs modulaires pour des premiers génériques les plus efficaces de l'état de l'art que nous avons listés en section 3.3 :

— MA16 de [MLPJ13] que nous avons réimplanté pour des premiers de $n = 128$ bits et $n = 256$ bits sur V4, V5, S6 et V7 ;
— MR8 de [MEML$^+$17] pour $n = 128$ bits et $n = 256$ bits sur A7 (Artix-7) ;
— MR16 de [MEML$^+$17] pour $n = 256$ bits sur A7 ;
— AM32 et AM64 de [ACZ16] pour $n = 256$ bits sur V7 ;
— MO64 de [MSDP16] pour $n = 256$ bits sur V5.

Nous avons aussi listé dans le tableau 3.9 les résultats d'implantation sur FPGA I2 (IGLOO 2) des très petits multiplieurs modulaires génériques MAS1 et MAS2 récemment proposés dans [MBCM17]. Nous n'avons cependant pas eu accès à ce FPGA pour nos implantations et nous ne nous pourrons donc pas nous comparer avec ces multiplieurs.

Comparer directement les résultats d'implantation « bruts » rapportés dans les tableaux 3.8 et 3.9 est un exercice difficile en raison de la diversité des métriques devant être considérées dans les différents FPGA. Pour cette raison, nous avons décidé dans [GT18a] d'illustrer les différences entre nos HTMM et

n	spéc. HTMM	FPGA	*slices* logiq.	LUT	FF	BRAM	*slices* DSP	fréq. MHz	λ 1M cc	λ 8M cc	tps 8M ns
128 bits	F44B	V4	**512**	**424**	**766**	$2^{\oplus}$	9	387	75	151	391
	S44B		**486**	**389**	**745**			320	74	150	468
	F44B	V5	**257**	**397**	**728**	$2^{\oplus}$	9	483	75	151	313
	S44B		**242**	**369**	**703**			397	74	150	378
	F44D	S6	**241**	**382**	**757**	0	9	349	75	151	433
	S44D		**199**	416	**721**			334	74	150	449
	S44B		203	**321**	**671**	$2^{\ominus}$		320			469
	F44D	V7	**306**	**600**	**758**	0	9	633	75	151	239
	F44B		325	**545**	725	$2^{\oplus}$		528			286
	S44B		**287**	**523**	**683**			481	74	150	312
256 bits	F48B	V4	**611**	**504**	**829**	$2^{\oplus}$	9	387	255	535	1383
	F28B		**523**	**455**	**786**			370	143	535	1445
	S28B		534	**422**	**766**			317	142	534	1682
	F48B	V5	**282**	**480**	**794**	$2^{\oplus}$	9	502	255	535	1065
	F28B		**240**	**433**	**751**			497	143	535	1076
	S48B		**232**	414	**667**			398	254	534	1342
	S28B		245	**408**	730			397	142	534	1346
	F28D	S6	**209**	**415**	**777**	0	9	349	143	535	1533
	S28B		**175**	358	**690**	$2^{\ominus}$		320	142	534	1668
	F28B		205	**335**	743			320	143	535	1671
	F28D	V7	**314**	**634**	**778**	0	9	598	143	535	895
	F48D		**291**	674	787			552	255	535	969
	F28B		296	**556**	743	$2^{\oplus}$		528	143	535	1013
	S28B		301	**552**	**703**			480	142	534	1112

TABLE 3.8 – Résultats d'implantation pour les HTMM 128 bits et 256 bits les plus intéressants.

n	Réf.	FPGA	*slices* logiq.	LUT	FF	BRAM	*slices* DSP	fréq. MHz	λ 1M cc	λ 8M cc	tps 8M ns
128 bits	MA16	V4	879	1201	1311	$6^{\oplus}$	21	252	27	167	663
		V5	440	1027	1310	$6^{\oplus}$		292			571
		S6	540	1600	1280	$6^{\ominus}$		210			795
		V7	455	1182	1305	$6^{\oplus}$		350			478
	MR8	A7	206	255	487	0	19	198	33	264*	1333*
256 bits	MA16	V4	1466	1998	2204	$10^{\oplus}$	37	250	37	233	932
		V5	698	1860	2172	$10^{\oplus}$		292			798
		S6	741	1941	2159	$10^{\ominus}$		177			1319
		V7	661	1770	2172	$10^{\oplus}$		372			626
	AM32	V7	n.r.	1917	n.r.	0	9	225	114	912*	4049*
	AM64		n.r.	2343	n.r.	0	32	161	76	608*	3776*
	MO64	V5	n.r.	699	333	$4^{\oplus}$	33	68	24	192*	2814*
	MR16	A7	402	846	1123	0	29	146	66	528*	3617*
	MR8		352	809	870	0	31	106	33	264*	2490*
	MAS1	I2	n.r.	505	257	1	1	250	583	4664*	18656*
	MAS2		n.r.	680	341	2	2		312	2496*	9984*

TABLE 3.9 – Résultats d'implantation de multiplieurs 128 et 256 bits de l'état de l'art récent (* indique un résultat estimé d'après l'article d'origine).

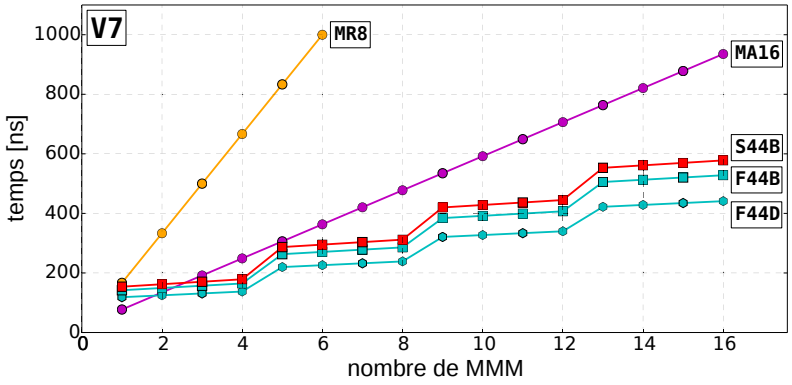


FIGURE 3.18 – Temps de calcul pour différents nombres de MMM dans des multiplieurs modulaires 128 bits implantés sur V7.

les multiplieurs de l'état de l'art en utilisant différentes figures.

Nous nous concentrerons sur le cas des multiplieurs modulaires avec $n = 128$ bits utilisables pour nos implantations de cryptosystèmes KHECC. Les comparaisons seront donc effectuées avec les multiplieurs MA16 et MR8 de l'état de l'art. Ce dernier étant implanté sur un FPGA Artix-7 (A7), nous limiterons ces comparaisons aux FPGA de génération 7, à savoir A7 et V7. Les observations que nous ferons dans la suite de cette section sont cependant aussi vraies sur les autres FPGA. Les *slices* logiques, les LUT, les BRAM et les *slices* DSP sont identiques dans les FPGA A7 et V7. Nous pouvons donc comparer directement les surfaces de circuit utilisées dans MR8 et dans nos implantations du HTMM sur V7. En revanche, la fréquence maximale possible dans A7 est inférieure à celle de V7 : 509 MHz au lieu de 741 MHz. La fréquence de MR8 étant de 198 MHz, cette différence de fréquences maximales atteignables n'impacte pas les comparaisons de MR8 avec nos implantations du HTMM sur V7.

La figure 3.18 permet de comparer les temps de calcul de différents nombres de MMM indépendantes dans des multiplieurs 128 bits implantés sur V7. Pour le calcul de 1 ou 2 MMM, MR16 est plus efficace que nos HTMM. Cependant, quand le nombre de MMM augmente, nos HTMM sont toujours plus rapides. En effet, l'augmentation du temps de calcul dans nos HTMM en fonction du nombre de MMM n'est pas linéaire mais évolue en suivant une progression en forme d'escalier. Tant que le nombre de LM occupés dans le HTMM est inférieur à σ, l'intervalle de temps entre les lancements de nouvelles MMM dans les LM libres est seulement de s ou $s + 1$ cycles par MMM. Quand σ LM sont remplis, la prochaine MMM peut être démarrée après $\tau - \sigma \times (s + 1)$ cycles si le HTMM n'utilise pas l'optimisation de réduction de latence ou $\tau - \sigma \times s$ cycles sinon. [2] Ce n'est pas le cas des multiplieurs MR8 et MA16 qui ne peuvent calculer qu'une unique MMM à la fois et dans lesquels le temps de calcul évolue donc de façon linéaire.

Pour 8 MMM notre HTMM 128 bits le plus performant sur V7, c.-à-d. F44D, est *deux fois plus rapide* que le meilleur multiplieur de l'état de l'art, à savoir MA16. F44D est aussi *beaucoup plus petit* avec seulement 9 *slices* DSP et 600 LUT utilisés contre 21 *slices* DSP et 1182 LUT dans MA16. Le HTMM F44D étant 2 fois plus petit et 2 fois plus rapide que MA16, il est par conséquent 4 *fois plus performant*.

En figure 3.19 nous illustrons l'ordonnancement de 8 MMM indépendantes dans différents multiplieurs 128 bits implantés sur V7. Sur la droite de cette figure, nous rappelons les nombres de *slices* DSP, de BRAM, de Slices logiques, de LUT et de FF ainsi que la fréquence dans les différents multiplieurs. Les

2. Pour rappel, τ est l'intervalle minimum en cycles entre 2 MMM dans 1 LM du HTMM.

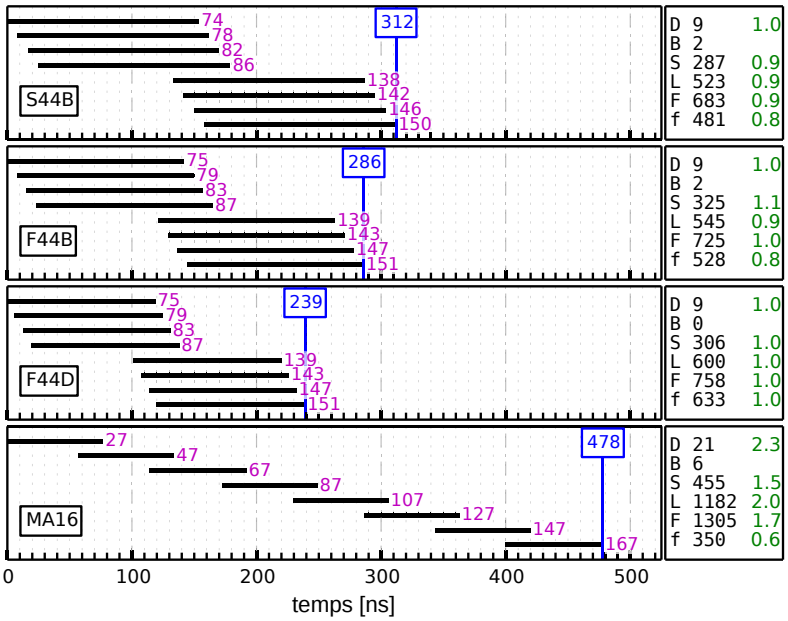


FIGURE 3.19 – Détail de l'ordonnancement de 8 MMM indépendantes dans différents multiplieurs 128 bits et résultats d'implantation pour différentes métriques sur FPGA Virtex-7. Les résultats des MMM sont générés aux cycles indiqués en violet et le temps de calcul total pour les 8 MMM, en ns, est indiqué en bleu.

nombres en vert dans cette dernière colonne correspondent aux résultats pour chacune des métriques normalisés par rapport aux résultats de F44D. L'ordonnancement au sein de MR8 n'est pas représenté en figure 3.19 pour des raisons d'échelle, le temps de calcul pour 3 MMM dans MR8 étant supérieur au temps de calcul pour 8 MMM dans MA16. Les ratios en vert montrent que les variations de surface entre nos HTMM S44B, F44B et F44D sont inférieures à 10%. *A contrario*, le multiplieur MA16 est plus gros et plus lent : 2.3 plus de *slices* DSP, 2 fois plus de LUT, 1.7 fois plus de FF et 1.5 fois plus de *slices* logiques pour un temps de calcul 2 fois plus important.

La figure 3.19 illustre clairement les raisons pour lesquelles l'utilisation de *l'hyper-threading* dans nos multiplieurs modulaires est efficace pour réduire les temps de calcul de nombreuses MMM indépendantes.

L'exploration de différentes spécifications des paramètres internes et de différentes configurations des *slices* DSP effectuée grâce à notre générateur de HTMM nous a permis d'approcher la fréquence maximale des *slices* DSP et des BRAM de nos FPGA. Sur V7 par exemple, la fréquence de fonctionnement de MA16 est de 350 MHz alors que celle atteinte dans F44D est de 633 MHz, soit 85.5% de la fréquence maximale possible de 741 MHz dans les *slices* DSP du V7.

Notre générateur nous a aussi permis d'évaluer les performances de F44B et de les comparer aux performances de F44D. Suite à cette comparaison, nous avons estimé que F44B était moins intéressante sur V7 que F44D en raison de sa fréquence de fonctionnement plus faible. Cette réduction de fréquence est imposée par les BRAM du V7 dont nous atteignons la fréquence maximale de 530 MHz dans F44B.

Afin d'évaluer et de comparer les performances de nos multiplieurs, nous avons introduit dans [GT18a] une métrique *d'efficacité matérielle*. Cette métrique nous permet de quantifier le taux d'utilisation des ressources matérielles utilisées dans nos HTMM. De telles métriques existent déjà dans la littérature, la plus connue d'entre elles étant sans doute le *produit temps–surface* qui consiste à multiplier le temps

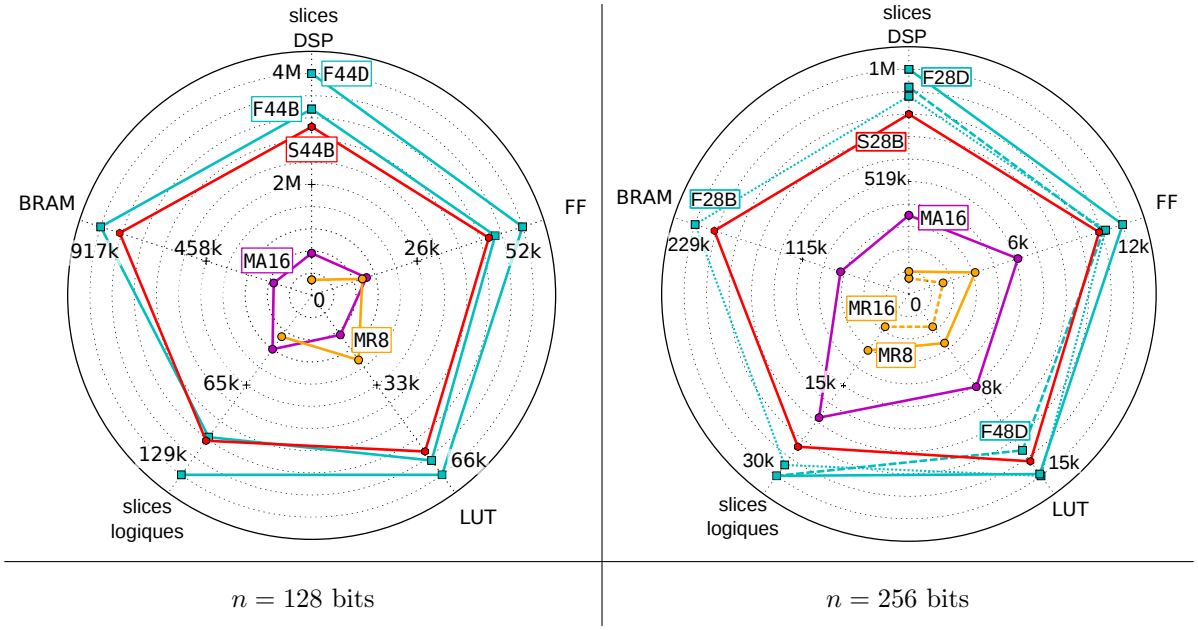


FIGURE 3.20 – Exemples d'efficacité matérielle, en nombre de MMM calculées par seconde par unité de surface, pour différents multiplieurs 128 bits (à gauche) et 256 bits (à droite) implantés sur V7 et pour différentes métriques de surface du FPGA. Les valeurs les plus grandes sont les meilleures (vers l'extérieur).

de calcul pour une opération par la surface utilisés. Cependant, cette métrique n'est pas bien adaptée pour évaluer l'efficacité de nos HTMM pour lesquels les performances dépendent du nombre d'opérations calculées. Notre métrique d'efficacité matérielle correspond au *débit moyen atteignable dans le multiplieur par unité de surface*. Elle est exprimée en nombre de MMM pouvant être calculées par seconde et par unité de surface.

Par exemple, F44D permet de calculer 129 k MMM par seconde et par *slice* logique. Ce HTMM est donc plus efficace que MA16 dont l'efficacité matérielle est de 37 k MMM calculables par secondes par *slice* logique. L'efficacité matérielle de nos meilleurs HTMM et des multiplieurs 128 bits de l'état de l'art implantés sur V7 est illustrée pour différentes métriques de surface en partie gauche de la figure 3.20. Cette dernière montre clairement que les ressources matérielles sont mieux utilisées dans nos multiplieurs hyper-threadés. L'utilisation de *l'hyper-threading* permet de réduire le nombre de « bulles » dans le pipeline interne et donc des implantations plus performantes d'algorithmes simples et réguliers tels que le CIOS.

Pour $n = 256$ bits, MA16 est le seul multiplieur modulaire de l'état de l'art permettant d'atteindre des meilleurs temps de calcul que nos HTMM, mais il est aussi beaucoup plus gros. Ainsi, MA16 est par exemple 1.48 fois plus rapide que F48B sur V4, mais il utilise 2.4 fois plus de *slices* logiques, 2.7 fois plus de FF et 4 fois plus de LUT. Les nombres de *slices* DSP et de BRAM implantés dans MA16 sur ce FPGA sont quant à eux respectivement 4.1 fois et 5 fois plus grand que dans F48B. L'utilisation des algorithmes plus complexes de [Oru95] dans MA16 permet à ce dernier d'être 50% plus rapide que notre HTMM 256 bits F48B mais au prix d'une surface 3 à 4 fois plus importante.

D'après les résultats rapportés dans le tableau 3.9, les autres multiplieurs de l'état de l'art AM32/64, MO64 et MR8/16 sont à la fois plus gros et plus lents que nos HTMM. La partie droite de la figure 3.20 illustre l'efficacité matérielle des meilleurs multiplieurs 256 bits HTMM et de l'état de l'art implantés sur V7. Elle démontre là encore clairement les bénéfices de *l'hyper-threading* sur le taux d'utilisation de

la surface du FPGA consommée : le nombre de MMM indépendantes calculées par seconde et par unité de surface dans nos HTMM 256 bits sur V7 est toujours supérieur à celui calculé dans le multiplieur optimisé MA16, quelle que soit la métrique de surface considérée.

Les implantations AM32 et AM64 ne sont pas représentées en figure 3.20 car seuls les nombres de *slices* logiques, de BRAM et de *slices* DSP sont rapportés dans [ACZ16]. On notera toutefois que AM32 est au moins 4 fois plus lent que nos HTMM F28D et F48D pour le même nombre de *slices* DSP et de BRAM, et au moins 6 fois plus de *slices* logiques. AM64 est aussi au moins 4 fois plus lent que F28D et F48D mais consomme 3.6 fois plus de *slices* DSP et au moins 7.5 fois plus de *slices* logiques.

3.9 Conclusion et perspectives pour notre HTMM

Dans ce chapitre, nous avons présenté un nouveau *multiplieur modulaire hyper-threadé* HTMM permettant d'augmenter l'efficacité des *slices* DSP dans les implantations FPGA de la multiplication modulaire de Montgomery pour des premiers génériques. *L'hyper-threading* nous permet de remplir le pipeline interne de ces *slices* DSP en limitant l'apparition de « bulles » dues aux dépendances de données dans les itérations des algorithmes de MMM simples et réguliers comme le CIOS. Ainsi, nous pouvons mettre en place un pipeline interne performant dans nos HTMM qui nous permet d'approcher les fréquences maximales dans les *slices* DSP et les BRAM : 502 MHz sur 550 MHz dans le FPGA Virtex-5 ou 633 MHz sur 740 MHz dans le FPGA Virtex-7 par exemple.

Nos premières versions du HTMM, proposées dans [GT17d], ont été implantées à la main sur Virtex-4, Virtex-5 et Spartan-6 pour des premiers de 128 bits sélectionnés à l'implantation. Elles nous ont permis d'obtenir de meilleures performances que le meilleur multiplieur 128 bits de la littérature, que nous avons réimplanté à partir de [MLPJ13].

Dans [GT18a] nous avons proposé un ensemble d'améliorations pour notre HTMM, parmi lesquelles une réduction du nombre de *slices* DSP, des optimisations de la latence des MMM calculées, la possibilité de modifier le premier P à l'exécution et l'augmentation du nombre de variantes pouvant être implantées. Nous avons aussi proposé un outil logiciel, disponible en *open source* depuis [GT18b], pour la génération automatique de HTMM avec différentes spécifications des paramètres internes qui nous a permis d'explorer de nombreux HTMM sur différents FPGA. Ce générateur nous a permis d'évaluer de nombreuses spécifications et de sélectionner les meilleures versions du HTMM sur différents FPGA.

L'exploration de l'espace de paramètres nous a permis de proposer des HTMM plus performants que les multiplieurs modulaires de l'état de l'art pour des premiers génériques de 128 et 256 bits quand plusieurs MMM indépendantes doivent être calculées. Pour des premiers de taille $n = 128$ bits, nos HTMM sont toujours plus rapides et plus petits que les multiplieurs de l'état de l'art : par exemple, le temps de calcul et le nombre de *slices* DSP sont respectivement divisés par 2 et par 2.3 dans notre HTMM F44D par rapport à MA16, la solution la plus performante de l'état de l'art. Pour des premiers de taille $n = 256$ *bits*, nos HTMM sont plus rapides et plus petits que les différents multiplieurs de l'état de l'art à l'exception de MA16. Le temps de calcul dans MA16 est en effet divisé par environ 1.5 mais ce multiplieur utilise en contrepartie 4.1 fois plus de *slices* DSP, 5 fois plus de BRAM et 3 fois plus de *slices* logiques.

Il reste cependant de nombreuses pistes à explorer et améliorations à apporter aux travaux présentés dans ce chapitre. Au niveau du générateur, il pourrait être intéressant d'étendre les scripts `Bash` afin de pouvoir prendre en charge un nombre plus importants d'outils de CAO commerciaux comme Vivado de

Xilinx ou Quartus de Intel–Altera, et des FPGA de fabricants différents (Intel–Altera ou MicroSemi par exemple). Dans un second temps, il serait aussi intéressant de modifier l'outil afin de pouvoir prendre en charge les configurations rectangulaires des *slices* DSP (25×18 bits par exemple), que nous n'avons pas utilisées dans les HTMM proposés. Enfin, le flot d'utilisation du générateur pourrait être étendu et complété par des analyses de consommation d'énergie totale dans l'unité implantée. Cela nous permettrait de compléter nos évaluations et d'ajouter une nouvelle métrique de comparaison à considérer lors de notre exploration. Nous pourrions par exemple estimer l'impact de l'utilisation du HTMM sur des systèmes embarqués autonomes en énergie ou fonctionnant sur batterie.

Au niveau des HTMM, de nombreuses pistes d'amélioration sont aussi envisageables. En particulier, il serait intéressant d'explorer et d'évaluer des spécifications de HTMM pour des tailles n de premiers différentes : $n < 100$ bits pour des applications à sécurité réduite ou pour des cryptosystèmes différents ; ou $n > 400$ bits pour les générations futures de cryptosystèmes HECC avec des niveaux de sécurité plus élevés par exemple. Enfin, il serait intéressant d'étudier l'impact sur l'architecture et sur les implantations de l'ajout de protections contre les SCA dans HTMM.

3.10 Annexe : résultats d'implantation complets

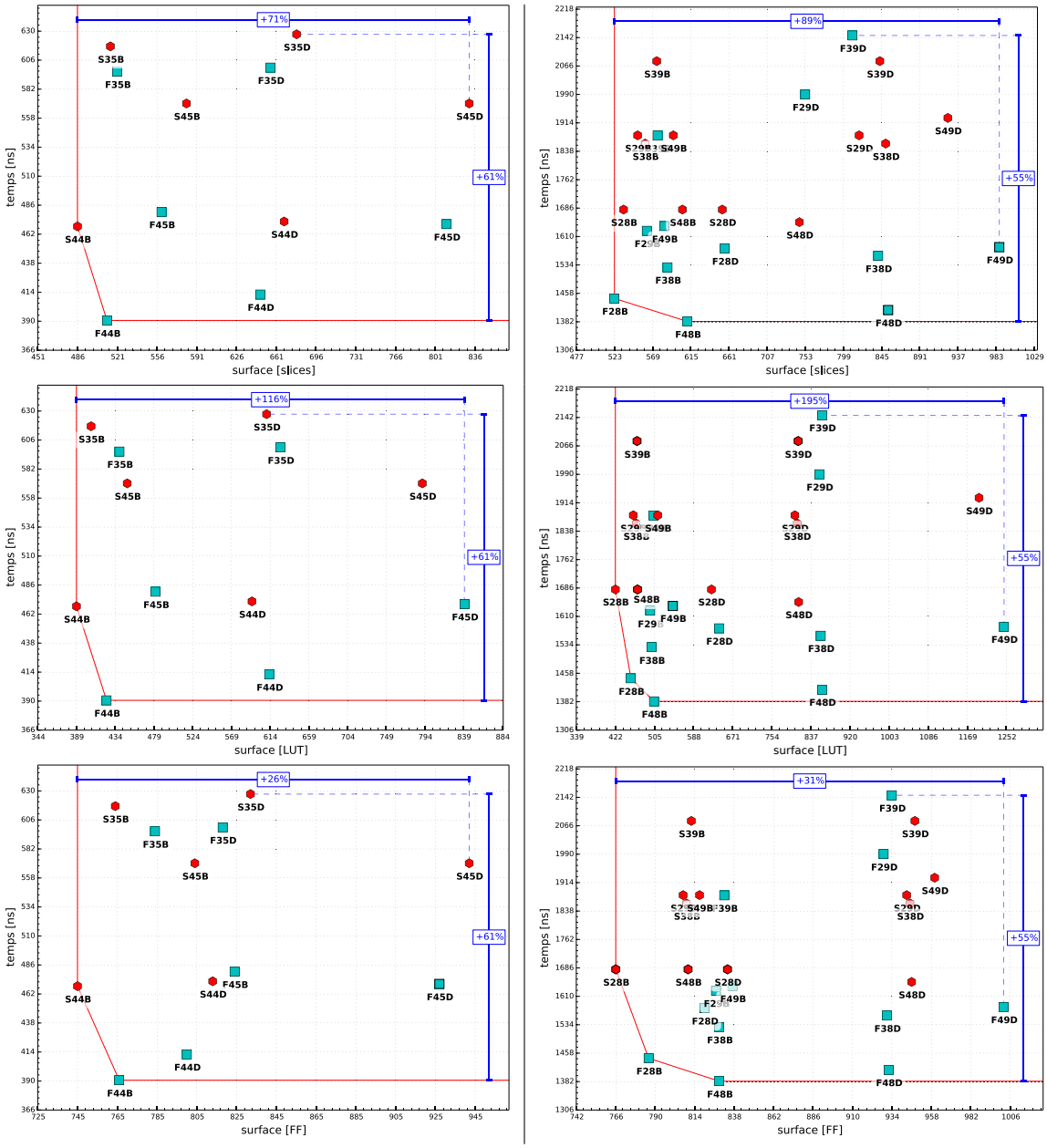


FIGURE 3.21 – Compromis surface – temps des HTMM spécifiés et implantés sur V4 pour trois différentes métriques de surface du FPGA : *slices* logiques, LUT et FF.

n	spéc. HTMM	*slices* logiques	LUT	FF	BRAM	*slices* DSP	fréq. MHz	λ cc	temps 8M ns
128 bits	F35B	521	439	784	$2^{\oplus}$	9	330	72	597
	F35D	656	626	818	0	9	328		600
	F44B	512	424	766	$2^{\oplus}$	9	387	75	391
	F44D	647	613	800	0	9	366		412
	F45B	560	481	824	$2^{\oplus}$	9	379	87	480
	F45D	811	840	927	0	9	387		470
	S35B	515	406	764	$2^{\oplus}$	9	317	71	617
	S35D	679	610	832	0	9	312		627
	S44B	486	389	745	$2^{\oplus}$	9	320	74	468
	S44D	668	593	813	0	9	317		473
	S45B	582	448	804	$2^{\oplus}$	9	317	86	570
	S45D	831	791	942	0	9	317		570
256 bits	F28B	523	455	786	$2^{\oplus}$	9	370	143	1445
	F28D	656	642	820	0	9	339		1578
	F29B	562	496	827	$2^{\oplus}$	9	368	157	1625
	F29D	753	855	929	0	9	300		1990
	F38B	587	499	829	$2^{\oplus}$	9	387	199	1528
	F38D	841	858	931	0	9	379		1558
	F39B	575	503	832	$2^{\oplus}$	9	352	220	1880
	F39D	810	861	934	0	9	308		2148
	F48B	611	504	829	$2^{\oplus}$	9	387	255	1383
	F48D	853	861	932	0	9	379		1413
	F49B	583	544	837	$2^{\oplus}$	9	365	283	1638
	F49D	987	1247	1002	0	9	378		1582
	S28B	534	422	766	$2^{\oplus}$	9	317	142	1682
	S28D	653	626	834	0	9	317		1682
	S29B	551	460	807	$2^{\oplus}$	9	317	156	1881
	S29D	818	803	943	0	9	317		1881
	S38B	560	466	809	$2^{\oplus}$	9	317	198	1859
	S38D	850	808	945	0	9	317		1859
	S39B	574	468	812	$2^{\oplus}$	9	317	219	2079
	S39D	843	810	948	0	9	317		2079
	S48B	605	469	810	$2^{\oplus}$	9	317	254	1682
	S48D	746	811	946	0	9	324		1648
	S49B	594	512	817	$2^{\oplus}$	9	317	282	1881
	S49D	925	1194	960	0	9	310		1927

TABLE 3.10 – Résultats d'implantation pour les différentes spécifications des HTMM 128 bits et 256 bits sur FPGA V4.

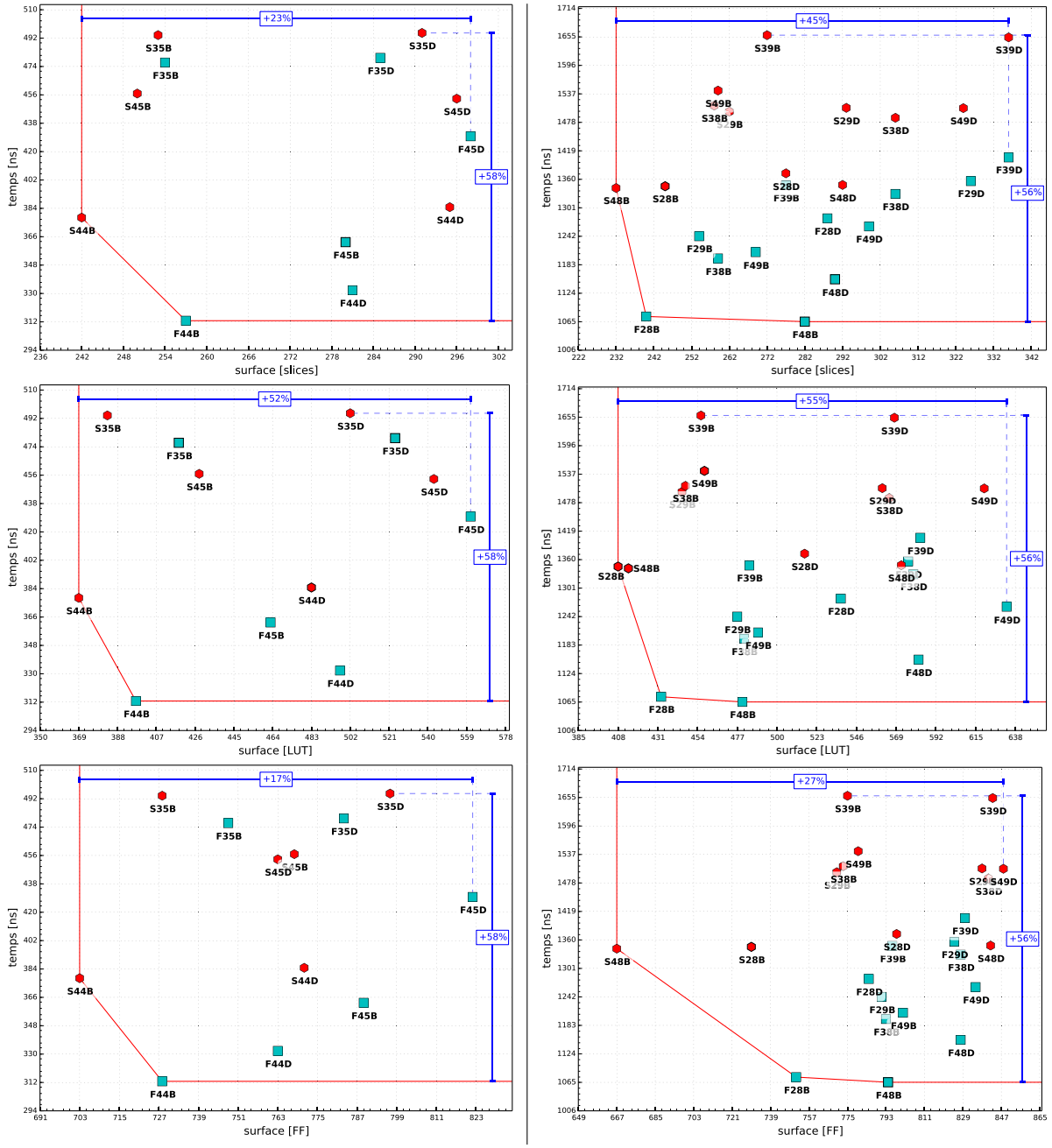


FIGURE 3.22 – Compromis surface – temps des HTMM spécifiés et implantés sur V5 pour trois différentes métriques de surface du FPGA : *slices* logiques, LUT et FF.

n	spéc. HTMM	*slices* logiques	LUT	FF	BRAM	*slices* DSP	fréq. MHz	λ cc	temps 8M ns
128 bits	F35B	254	418	748	$2^{\oplus}$	9	413	72	477
	F35D	285	524	783	0	9	411		479
	F44B	257	397	728	$2^{\oplus}$	9	483	75	313
	F44D	281	497	763	0	9	455		332
	F45B	280	463	789	$2^{\oplus}$	9	502	87	363
	F45D	298	561	822	0	9	424		430
	S35B	253	383	728	$2^{\oplus}$	9	397	71	494
	S35D	291	502	797	0	9	396		495
	S44B	242	369	703	$2^{\oplus}$	9	397	74	378
	S44D	295	483	771	0	9	390		385
	S45B	250	428	768	$2^{\oplus}$	9	396	86	457
	S45D	296	543	763	0	9	399		454
256 bits	F28B	240	433	751	$2^{\oplus}$	9	497	143	1076
	F28D	288	537	785	0	9	418		1279
	F29B	254	477	791	$2^{\oplus}$	9	481	157	1242
	F29D	326	576	825	0	9	441		1356
	F38B	259	481	793	$2^{\oplus}$	9	494	199	1196
	F38D	306	579	828	0	9	444		1330
	F39B	277	484	796	$2^{\oplus}$	9	490	220	1348
	F39D	336	583	830	0	9	470		1405
	F48B	282	480	794	$2^{\oplus}$	9	502	255	1065
	F48D	290	582	828	0	9	464		1153
	F49B	269	489	801	$2^{\oplus}$	9	495	283	1209
	F49D	299	633	835	0	9	474		1262
	S28B	245	408	730	$2^{\oplus}$	9	397	142	1346
	S28D	277	516	798	0	9	389		1372
	S29B	262	445	770	$2^{\oplus}$	9	398	156	1500
	S29D	293	561	838	0	9	396		1508
	S38B	258	447	773	$2^{\oplus}$	9	390	198	1513
	S38D	306	565	841	0	9	397		1487
	S39B	272	456	775	$2^{\oplus}$	9	398	219	1659
	S39D	336	568	843	0	9	399		1654
	S48B	232	414	667	$2^{\oplus}$	9	398	254	1342
	S48D	292	572	842	0	9	396		1348
	S49B	259	458	780	$2^{\oplus}$	9	387	282	1544
	S49D	324	620	848	0	9	396		1507

TABLE 3.11 – Résultats d'implantation des différentes spécifications des HTMM 128 bits et 256 bits sur FPGA V5.

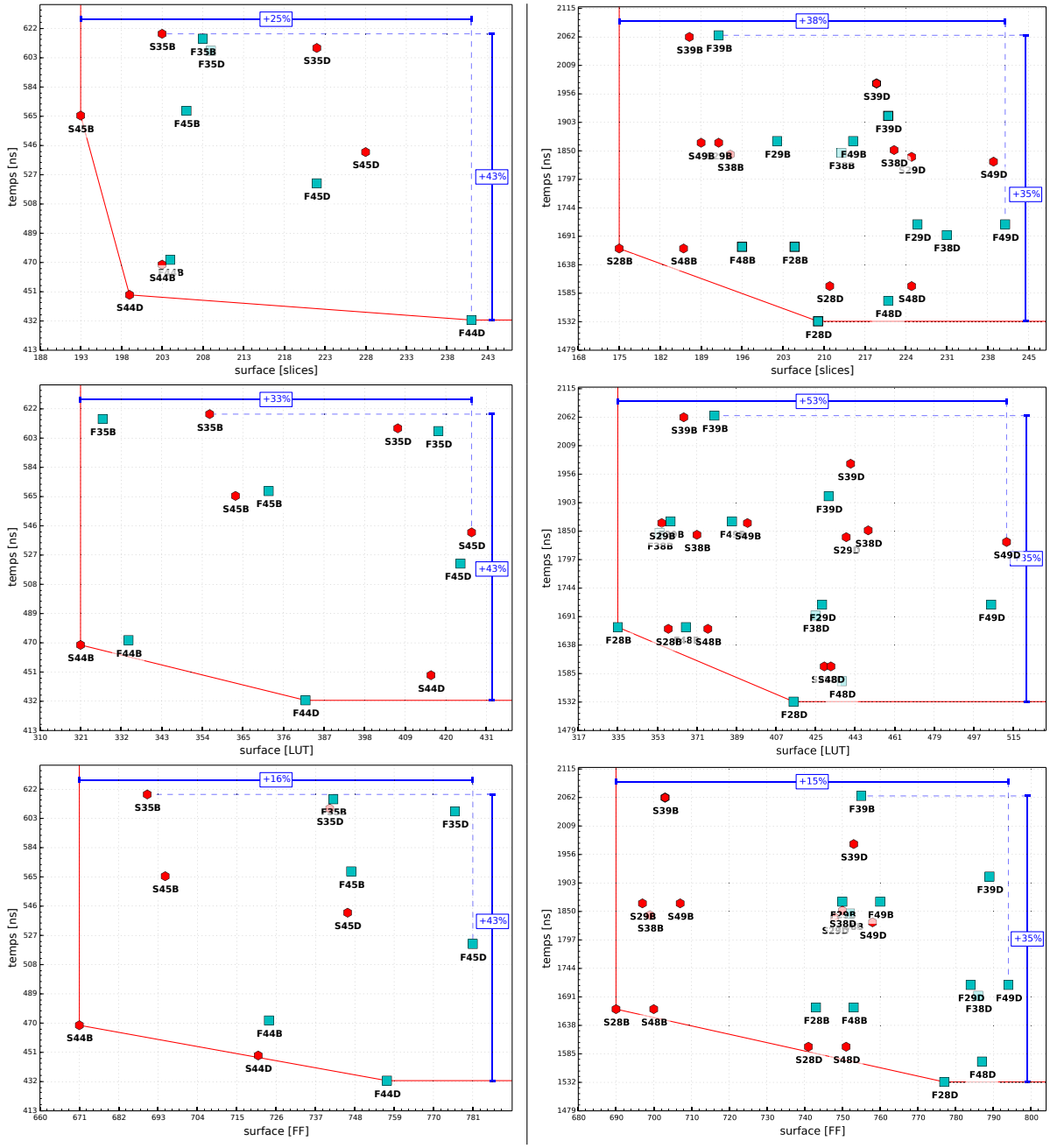


FIGURE 3.23 – Compromis surface – temps des HTMM spécifiés et implantés sur S6 pour trois différentes métriques de surface du FPGA : *slices* logiques, LUT et FF.

n	spéc. HTMM	*slices* logiques	LUT	FF	BRAM	*slices* DSP	fréq. MHz	λ cc	temps 8M ns
128 bits	F35B	208	327	742	$2^{\ominus}$	9	320	72	615
	F35D	209	418	776	0	9	324		608
	F44B	204	334	724	$2^{\ominus}$	9	320	75	472
	F44D	241	382	757	0	9	349		433
	F45B	206	372	747	$2^{\ominus}$	9	320	87	569
	F45D	222	424	781	0	9	349		521
	S35B	203	356	690	$2^{\ominus}$	9	317	71	619
	S35D	222	407	741	0	9	322		609
	S44B	203	321	671	$2^{\ominus}$	9	320	74	469
	S44D	199	416	721	0	9	334		449
	S45B	193	363	695	$2^{\ominus}$	9	320	86	565
	S45D	228	427	746	0	9	334		542
256 bits	F28B	205	335	743	$2^{\ominus}$	9	320	143	1671
	F28D	209	415	777	0	9	349		1533
	F29B	202	359	750	$2^{\ominus}$	9	320	157	1868
	F29D	226	428	784	0	9	349		1713
	F38B	213	354	752	$2^{\ominus}$	9	320	199	1846
	F38D	231	425	786	0	9	349		1693
	F39B	192	379	755	$2^{\ominus}$	9	320	220	2065
	F39D	221	431	789	0	9	345		1915
	F48B	196	366	753	$2^{\ominus}$	9	320	255	1671
	F48D	221	437	787	0	9	341		1571
	F49B	215	387	760	$2^{\ominus}$	9	320	283	1868
	F49D	241	505	794	0	9	349		1713
	S28B	175	358	690	$2^{\ominus}$	9	320	142	1668
	S28D	211	429	741	0	9	334		1598
	S29B	192	355	697	$2^{\ominus}$	9	320	156	1865
	S29D	225	439	748	0	9	325		1839
	S38B	194	371	699	$2^{\ominus}$	9	320	198	1843
	S38D	222	449	750	0	9	319		1851
	S39B	187	365	703	$2^{\ominus}$	9	320	219	2062
	S39D	219	441	753	0	9	334		1975
	S48B	186	376	700	$2^{\ominus}$	9	320	254	1668
	S48D	225	432	751	0	9	334		1598
	S49B	189	394	707	$2^{\ominus}$	9	320	282	1865
	S49D	239	512	758	0	9	326		1830

TABLE 3.12 – Résultats d'implantation des différentes spécifications des HTMM 128 bits et 256 bits sur FPGA S6.

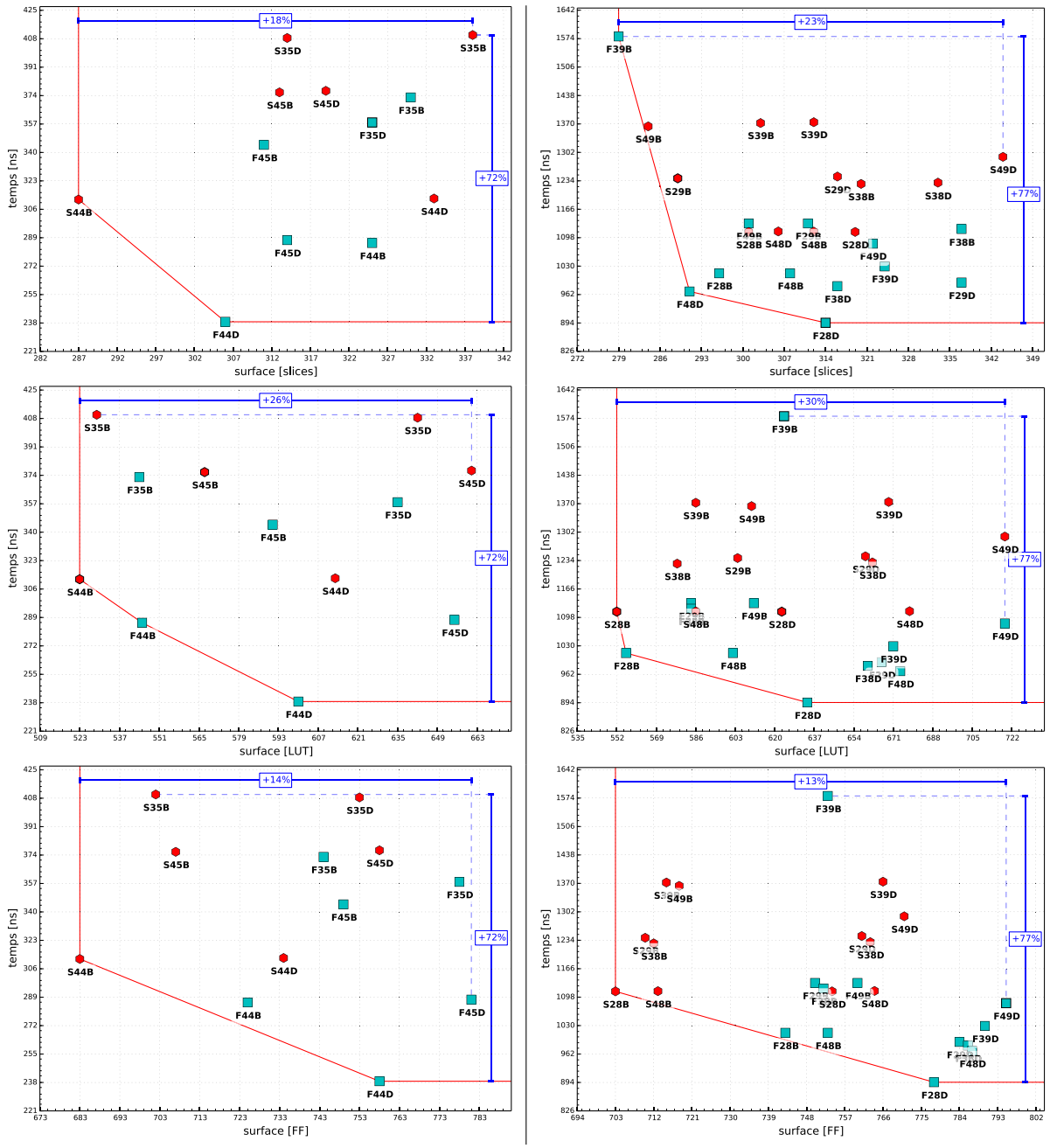


FIGURE 3.24 – Compromis surface – temps des HTMM spécifiés et implantés sur V7 pour trois différentes métriques de surface du FPGA : *slices* logiques, LUT et FF.

n	spéc. HTMM	*slices* logiques	LUT	FF	BRAM	*slices* DSP	fréq. MHz	λ cc	temps 8M ns
128 bits	F35B	330	544	744	$2^{\oplus}$	9	528	72	373
	F35D	325	635	778	0	9	550		358
	F44B	325	545	725	$2^{\oplus}$	9	528	75	286
	F44D	306	600	758	0	9	633		239
	F45B	311	591	749	$2^{\oplus}$	9	528	87	345
	F45D	314	655	781	0	9	633		288
	S35B	338	529	702	$2^{\oplus}$	9	478	71	410
	S35D	314	642	753	0	9	480		408
	S44B	287	523	683	$2^{\oplus}$	9	481	74	312
	S44D	333	613	734	0	9	480		312
	S45B	313	567	707	$2^{\oplus}$	9	481	86	376
	S45D	319	661	758	0	9	480		377
256 bits	F28B	296	556	743	$2^{\oplus}$	9	528	143	1013
	F28D	314	634	778	0	9	598		895
	F29B	311	584	750	$2^{\oplus}$	9	528	157	1132
	F29D	337	666	784	0	9	604		991
	F38B	337	584	752	$2^{\oplus}$	9	528	199	1119
	F38D	316	660	786	0	9	602		982
	F39B	279	624	753	$2^{\oplus}$	9	419	220	1579
	F39D	324	671	790	0	9	642		1029
	F48B	308	602	753	$2^{\oplus}$	9	528	255	1013
	F48D	291	674	787	0	9	552		969
	F49B	301	611	760	$2^{\oplus}$	9	528	283	1132
	F49D	322	719	795	0	9	552		1084
	S28B	301	552	703	$2^{\oplus}$	9	480	142	1112
	S28D	319	623	754	0	9	480		1112
	S29B	289	604	710	$2^{\oplus}$	9	481	156	1240
	S29D	316	659	761	0	9	480		1244
	S38B	320	578	712	$2^{\oplus}$	9	481	198	1227
	S38D	333	662	763	0	9	480		1230
	S39B	303	586	715	$2^{\oplus}$	9	481	219	1372
	S39D	312	669	766	0	9	480		1374
	S48B	312	586	713	$2^{\oplus}$	9	480	254	1112
	S48D	306	678	764	0	9	480		1113
	S49B	284	610	718	$2^{\oplus}$	9	438	282	1364
	S49D	344	719	771	0	9	462		1291

TABLE 3.13 – Résultats d'implantation des différentes spécifications des HTMM 128 bits et 256 bits sur FPGA V7.

4 Accélérateurs matériels optimisés pour Kummer-HECC

Sommaire

4.1 Introduction

Dans ce chapitre basé sur les travaux que nous avons publiés en 2017 à la conférence internationale Indocrypt (voir [GCT17]), nous nous intéresserons à l'implantation d'accélérateurs matériels pour le calcul de l'opération de multiplication scalaire $[k]\mathcal{P}$ dans HECC. Ces accélérateurs utilisent les surfaces de Kummer associées à des courbes hyperelliptiques *quelconques* sur des corps finis de grande caractéristique première et générique de 128 bits. Ils ont été conçus pour des scalaires k de 256 bits dont l'utilisation dans (K)HECC permet de garantir le niveau de sécurité théorique de 128 bits actuellement recommandé. Les architectures de coprocesseur présentées sont à notre connaissance les *premières architectures* proposées dans l'état de l'art pour l'implantation FPGA d'accélérateurs HECC sur des *corps fini de grande caractéristique première et générique* pour ce niveau de sécurité.

Dans un premier temps, nous étudierons dans la section 4.2 les différentes implantations matérielles proposées dans l'état de l'art pour HECC. Nous commencerons par les implantations « classiques » à base de Jacobiennes de courbes hyperelliptiques avant de discuter de deux implantations de la littérature utilisant les surfaces de Kummer. Certaines des principales différences entre nos implantations pour KHECC et celles de l'état de l'art seront ensuite discutées et motivées dans la section 4.3.

La section 4.4 sera consacrée à la conception et la mise en place de nos architectures. Dans cette section, nous décrirons les différentes unités implantées dans nos accélérateurs ainsi que les choix que nous avons fait quant à la topologie et au contrôle interne dans ces unités. La diversité des différents paramètres pouvant être spécifiés lors de la conception des architectures a nécessité la mise en place d'outils permettant de valider et d'estimer rapidement les performances de différentes solutions. Ceux-ci seront aussi détaillés dans cette section.

Nous avons exploré différents types et tailles d'architectures allant de petites architectures à de plus grosses architectures parallèles. Grâce aux outils que nous avons proposés, nous avons aussi exploré diverses spécifications des paramètres internes pour chaque architecture (p. ex. taille des communications entre la mémoire et les unités arithmétiques). Les performances et coûts en surface de ces différentes architectures ont été mesurées après implantations sur plusieurs FPGA. Nous présenterons dans la section 4.5 quatre architectures implantées pour différentes spécifications de paramètres : deux petites architectures et deux plus grosses architectures parallèles.

Les résultats d'implantation des accélerateurs basés sur ces architectures seront discutés et comparés avec l'état de l'art des implantations ECC et HECC dans la section 4.6. Nous verrons que nos accélérateurs les plus performants sont plus efficaces en terme de taille et/ou de temps de calcul que les meilleures implantations ECC de l'état de l'art pour des courbes quelconques. Ils sont aussi compétitifs avec les implantations ECC et HECC optimisées pour des courbes particulières.

Les accélérateurs que nous avons proposé dans [GCT17] utilisent les multiplieurs modulaires hyper-threadés que nous avons présenté en 2017 dans [GT17d]. Nous avons depuis optimisé ces multiplieurs modulaires et les nouvelles versions de ces derniers ont été soumises au journal IEEE Transactions on Computers en mai 2018 (cf. [GT18a]). Nous présenterons dans la section 4.7 un ensemble de nouvelles implantations matérielles de nos accélérateurs dans lesquelles nous utilisons nos multiplieurs optimisés de [GT18a]. Nous avons aussi étendu ces implantations à un FPGA Artix-7 récent de Xilinx.

Enfin, la section 4.8 conclura ce chapitre, nous permettant de donner quelques perspectives et de discuter de certaines pistes qu'il serait intéressant d'explorer.

4.2 État de l'art des implantations de HECC sur FPGA

4.2.1 Implantations utilisant les Jacobiennes des courbes hyperelliptiques

Bien que l'utilisation des courbes hyperelliptiques ait été proposée pour la première fois en 1988 par Koblitz dans [Kob88], il a fallu attendre le début des années 2000 pour voir apparaître les premières implantations matérielles d'accélérateurs HECC. Ces dernières, tout comme la très grande majorité des accélérateurs proposés dans la littérature, ont été conçues pour le calcul de la multiplication scalaire $[k]\mathcal{P}$ sur les Jacobiennes de *courbes hyperelliptiques binaires*, à coefficients et paramètres dans GF(2^m). L'arithmétique dans les corps de type GF(2^m) consiste à calculer des opérations sur des polynômes de degré m et à coefficients dans GF(2), modulo un *polynôme irréductible*. Elle est de ce fait plus simple que l'arithmétique dans GF(P) car il n'y a pas de propagation de retenues entre les différents coefficients. Ainsi, par exemple, l'addition de deux polynômes à coefficients dans GF(2) correspond à un simple XOR entre les bits des différents opérandes, les différents coefficients du résultat pouvant alors être calculés en parallèle.

Si l'utilisation de courbes binaires a été longtemps privilégiée pour accélérer les calculs au niveau du corps, elle est cependant moins conseillée à l'heure actuelle pour des raisons de sécurité (cf. [BL13] ou [PQ12] par exemple). Les standards récents pour les implantations de protocoles (H)ECC d'échange de clé et de signature numérique privilégient l'utilisation de courbes définies sur des corps finis GF(P) de grande caractéristique première. Par exemple, la NSA (*National Security Agency* en charge de la sécurité des systèmes d'information du gouvernement américain) recommande depuis les années 2010 l'utilisation des courbes P-256 et P-384 pour l'implantation de ces protocoles. Les courbes P-256 et P-384 sont des courbes standardisées par le NIST et définies sur des corps finis premiers de caractéristiques respectives NIST-256 $= 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ et NIST-384 $= 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$.

Les résultats d'implantation détaillés des différents accélérateurs de la littérature HECC utilisant des courbes binaires sont résumés dans le tableau 4.1 en page 99.

Les premières études portant sur la conception de coprocesseurs matériels pour le calcul de la multiplication scalaire dans HECC ont été effectuées et publiées dans la thèse de master de Wollinger [Wol01] en 2001. Cependant, la première implantation matérielle d'un coprocesseur pour HECC peut être attribuée à Boston et coll. en 2002 dans [BCLW02] (le coprocesseur correspondant est nommé Bos02 dans le tableau 4.1). Ce coprocesseur utilise l'algorithme de Cantor [Can87] modifié par Koblitz [Kob89] pour calculer l'opération de multiplication scalaire sur la Jacobienne d'une courbe hyperelliptique de genre 2 définie sur le corps fini GF(2^{113}).

Il est construit autour d'un ensemble d'unités arithmétiques dédiées au calcul des opérations dans GF(2^{113}), parmi lesquelles : 2 multiplieurs, 1 unité pour les carrés, 1 additionneur, 1 unité dédiée au calcul du pgcd, et 1 unité pour les divisions dans le corps fini. Une unité spécifique permet de contrôler les transferts des opérandes et des résultats entre les différentes unités et le banc de registres via un réseau de (dé)multiplexeurs.

Après implantation sur un FPGA Virtex-2 le coprocesseur de Boston et coll. peut calculer une multiplication scalaire en 10.1 ms en utilisant deux blocs matériels concurrents pour le calcul des opérations d'addition et de doublement de points. Les surfaces consommées sont données séparément pour chaque bloc (16600 *slices* logiques pour le bloc d'addition et 15100 *slices* logiques pour le bloc de doublement). Le nombre de *slices* logiques indiqué pour Bos02 dans le tableau 4.1 correspond à la somme des nombres de *slices* logiques pour chacun des deux blocs matériels.

En 2004, Kim et coll. [KWC+04] ont proposé et implanté sur un FPGA Virtex-2 trois différents coprocesseur HECC visant respectivement les hautes performances (version dénotée Kim04a dans le tableau 4.1), un partage efficace des ressources (version Kim04b dans le tableau 4.1) et une faible consommation de surface (version Kim04c dans le tableau 4.1). Il s'agit des premières implantations connues de HECC utilisant la version affine des formules explicites introduites dans [PWP04], plus performantes que l'algorithme original de Cantor. Elles utilisent une courbe hyperelliptique définie sur le corps fini GF(2^{89}).

Les différents coprocesseurs sont construits autour d'un ensemble d'unités arithmétiques pour les calculs modulaires dans GF(2^{89}) : multiplieur modulaire, unité pour le calcul de l'inversion dans le corps fini (on parle d'inverseur dans le corps ou d'inverseur modulaire), additionneur modulaire et unité dédiée au calcul des carrés dans le corps. Ils embarquent aussi un banc de registres (ou mémoire) connecté aux multiplieurs et inverseurs modulaires via un réseau d'interconnexion. En raison de leurs petites tailles, les unités dédiées aux carrés et aux additions sont placées entre la mémoire et les multiplieurs/inverseurs modulaires. Le contrôle dans les coprocesseurs est construit de façon hiérarchique : un contrôle central gère l'ordonnancement des opérations au niveau courbe (additions et doublements) tandis qu'un ensemble d'unités de contrôle locales prennent en charge le fonctionnement des unités arithmétiques impliquées dans le calcul de ces opérations.

La version Kim04a est décomposée en deux clusters dédiés au calcul de l'addition et du doublement de points embarquant chacun 1 banc de registres, 1 inverseur dans le corps, 1 contrôle local, et respectivement 2 et 1 multiplieurs. Les deux autres versions utilisent un banc de registres, 2 multiplieurs et 1 inverseur modulaire, partagés entre les opérations d'addition et de doublement de points. La principale différence entre Kim04b et Kim04c est l'utilisation dans cette dernière d'une mémoire distribuée de 1536 bits au lieu d'un banc de registres implanté dans les *slices* logiques.

Les trois architectures discutées dans [KWC+04] ont aussi été utilisées par Wollinger pour le calcul de la multiplication scalaire sur une courbe hyperelliptique définie sur GF(2^{81}). Les résultats d'implantation sur FPGA Virtex-2 et Virtex-2 Pro ont été rapportés dans son manuscrit de thèse [Wol04] publié en 2004 pour les trois versions (nommées respectivement Wol04a, Wol04b et Wol04c dans le tableau 4.1) : rapide, avec ressources partagées, et avec utilisation des blocs mémoires des FPGA.

En 2006, Sakiyama et coll. ont proposé dans [SBPV06] un coprocesseur superscalaire permettant d'exploiter *dynamiquement* le parallélisme d'instructions lors du calcul de l'opération de multiplication scalaire dans trois cryptosystèmes ECC et HECC différents. Le coprocesseur HECC implanté se base sur une courbe de genre 2 définie sur GF(2^{83}) et utilise un système de coordonnées mixtes pour le calcul de la multiplication scalaire.

Les opérations au niveau corps sont prises en charge par un ensemble d'unités arithmétiques (ou MALU) calculant l'une des deux opérations $AB + C$ ou $A \times (B + D) + C$ dans GF(2^{83}). L'architecture n'embarque par ailleurs aucun autre type d'unité arithmétique. En particulier, l'inversion dans le corps est calculée par l'intermédiaire d'une suite de multiplications et d'additions.

La flexibilité de l'architecture proposée par Sakiyama et coll. a permis d'explorer différentes configurations de leur coprocesseur pour HECC, parmi lesquelles trois ont été sélectionnées pour implantation sur un FPGA Virtex-2 Pro. Ces différentes versions (Sak06a, Sak06b et Sak06c dans le tableau 4.1) se basent sur la même topologie d'architecture mais embarquent respectivement 1, 2 et 3 unités arithmétiques MALU.

Toujours en 2006, Batina et coll. ont publié dans [BMPV06] une comparaison de deux accélérateurs

matériels, l'un pour ECC et l'autre pour HECC, tous deux implantés sur un FPGA Virtex-2 Pro. Le coprocesseur HECC proposé (Bat06 dans le tableau 4.1) implante l'algorithme de *double-and-add* pour la multiplication scalaire sur une courbe de genre 2 à coefficients dans GF(2^{83}), avec utilisation de coordonnées projectives.

L'architecture de l'accélérateur est construite de façon à refléter la hiérarchie des opérations dans (H)ECC. Au niveau le plus bas, on retrouve les unités arithmétiques pour le calcul des opérations dans GF(2^{83}). Celles-ci sont au nombre de deux : 1 multiplieur implantant l'algorithme de multiplication modulaire de Montgomery (pour les multiplications modulo un polynôme irréductible dans GF(2^{83})) et 1 additionneur (implantant un simple XOR bits à bits des 2 opérandes). Les deux unités arithmétiques sont contrôlées par trois unités de contrôle locales, construites autour d'automates finis (FSM) et peuvent être utilisées en parallèle. Chacune des trois FSM permettent d'ordonnancer le calcul des opérations dans GF(2^{83}) nécessaires au calcul de l'une des trois primitives suivantes : addition de points, doublement de point et inversion dans GF(2^{83}) (utilisant le petit théorème de Fermat). Finalement, une unité de contrôle global à base de FSM est en charge de l'ordonnancement de ces primitives de haut niveau dans le coprocesseur.

Les temps de calcul rapportés dans [BMPV06] pour l'accélérateur Bat06 sont légèrement supérieurs à ceux des implantations proposées dans [KWC+04] (version Kim04a en particulier). Les auteurs expliquent cette différence de performances par l'utilisation de coordonnées affines avec une unité dédiée à l'inversion dans [KWC+04]. On rappellera aussi que l'architecture de Batina et coll. est conçue pour l'implantation de coprocesseurs pour ECC *et* pour HECC. D'après les auteurs, les performances des implantations ECC et HECC pourraient être améliorées en utilisant des optimisations spécifiques à chacun de ces 2 cryptosystèmes. Ces travaux sont toutefois très intéressants car ils proposent la première comparaison de coprocesseurs pour ECC et pour HECC, celle-ci montrant que HECC peut être aussi efficace que ECC en matériel, avec -25.5% de temps de calcul pour une surface en nombre de *slices* logiques augmentée de 28.8%.

Deux autres implantations de coprocesseurs HECC sur un FPGA Virtex-2 ont été proposées en 2007 dans [EMY07] par Elias et coll. pour des courbes de genre 2 définies sur GF(2^{113}). Ces coprocesseurs reposent sur une architecture hiérarchique dans laquelle trois blocs matériels différents sont en charge de l'exécution de la multiplication scalaire, des opérations arithmétiques au niveau courbe (addition et doublement de points) et des opérations arithmétiques au niveau du corps fini. Chaque bloc intègre un ensemble de registres utilisés en tant que mémoire locale. Le dernier bloc regroupe les différentes unités arithmétiques pour le calcul des primitives gérées dans le bloc supérieur : 4 multiplieurs et 1 inverseur dans le corps pour la conversion des coordonnées dans l'accélérateur, 4 multiplieurs et 4 additionneurs pour l'addition de points, et 4 multiplieurs et additionneurs pour le doublement de points. Les blocs étant indépendants, les trois différentes primitives peuvent être évaluées en parallèle.

La première implantation (Eli07a dans le tableau 4.1) utilise un système de coordonnées projectives couplé à l'algorithme de multiplication scalaire *double-and-add* avec parcours des bits de k depuis les poids faibles (algo. 1 dans le chapitre 2). La deuxième implantation (Eli07b dans le tableau 4.1) utilise quant à elle les coordonnées mixtes et un recodage NAF du scalaire. Par rapport à la version Eli07a, l'architecture de Eli07b intègre une unité supplémentaire pour le recodage, couplée au premier bloc pour le contrôle de la multiplication scalaire.

En 2008, Fan et coll. ont décrit dans [FBV08] une architecture de coprocesseur HECC compact, visant une faible surface de circuit. Cette architecture a été implantée sur un FPGA Virtex-2 pour une courbe

hyperelliptique binaire définie sur GF(2^{83}). Elle intègre : 1 unité de contrôle basée sur un microcode de 4 instructions, 1 mémoire ROM stockant les séquences d'instructions à exécuter pour le calcul des opérations au niveau courbe, 1 unité arithmétique *unifiée* pour les multiplications et inversions dans le corps, et 1 mémoire RAM pour le stockage des paramètres de la courbe et des données intermédiaires (implantée dans les BRAM du FPGA).

La multiplication scalaire est calculée en utilisant un recodage NAF du scalaire et les coordonnées projectives des éléments de la courbe. Le coprocesseur proposé par Fan et coll. (Fan08) est plus rapide et plus petit que les différentes solutions décrites précédemment.

Des implantations plus récentes ont été proposées par Sghaier et coll. dans [SMZM16] en 2016, pour deux versions de coprocesseurs HECC se basant sur le corps fini binaire GF(2^{83}). La première version utilise les formules de Cantor [Can87] pour l'addition et le doublement de points. La seconde version (Sgh16) utilise les formules explicites de Harley [Har00], optimisées par les auteurs pour réduire le nombre d'opérations requises dans GF(2^{83}). Ces deux versions utilisent la représentation affine des coordonnées des points et sont implantées sur Virtex-2 et sur Virtex-5. Le coût en surface de circuit de la version utilisant l'algorithme de Cantor est largement plus grand que celui de la version Sgh16, soit 2.7× plus de *slices* logiques, pour un temps de calcul plus de 20 fois supérieur.

L'architecture des cryptoprocesseurs décrits est construite autour d'un contrôle central, d'un ensemble de blocs pour le calcul de primitives haut niveau, de registres d'entrées/sorties et d'un ensemble d'unités arithmétiques incluant entre autres 2 multiplieurs (implantant l'algorithme de multiplication de Montgomery), 1 unité pour les carrés, 1 unité dédiée au calcul du pgcd (uniquement dans la première version) et 1 inverseur modulaire. Nous renvoyons le lecteur vers [SMZM16] pour les détails sur les optimisations des formules ainsi que pour les détails de l'architecture.

Finalement, les travaux d'Alimi et coll. publiés en 2017 dans [ASMT17] portent sur l'exploration d'architectures d'accélérateurs pour HECC dans GF(2^{83}). Les coprocesseurs proposés utilisent l'algorithme de l'échelle de Montgomery de [Mon87] (algo. 7 dans le chapitre 2). Les opérations au niveau courbe (addition et doublement de points) sont calculées en utilisant les formules explicites de [CFA$^+$05] et un système de représentation projectives des coordonnées. La topologie d'architecture utilisée repose sur un contrôleur central pilotant un ensemble de 5 blocs matériels. Deux blocs sont respectivement en charge de la conversion « affine → projective » des coordonnées des opérandes et de la conversion « projective → affine » des coordonnées des résultats du coprocesseur. Ils font tous deux appel à un bloc implantant différentes unités arithmétiques pour calculer les 4 multiplications et l'inversion dans GF(2^{83}) nécessaires à la conversion affine–projective ou projective–affine. La configuration des unités arithmétiques au sein ce bloc n'est pas détaillée dans [ASMT17]. En particulier, aucune précision n'est apportée sur le choix de l'algorithme utilisé (p. ex. petit théorème de Fermat ou algorithme d'Euclide étendu), sur l'utilisation ou non d'une unité dédiée pour cette opération ou encore sur le nombre de multiplieurs utilisés. Enfin, deux blocs sont respectivement consacrés aux calculs de l'addition et du doublement de points, utilisant chacun ses propres unités arithmétiques (additionneurs et multiplieurs dans GF(2^{83})). Les coprocesseurs de [ASMT17] peuvent donc calculer une addition et un doublement de points en parallèle.

Les auteurs explorent plusieurs configurations de ces deux blocs en faisant varier le nombre de multiplieurs utilisés dans chaque bloc, et donc les possibilités de parallélisme entre les opérations calculées dans ces multiplieurs. Le coprocesseur le plus performant (Ali17 dans le tableau 4.1) utilise 6 multiplieurs pour l'addition de points et 7 pour le doublement et permet d'obtenir le meilleur produit temps–surface parmi les différentes versions proposées après implantation sur un FPGA Virtex-6.

réf.	corps fini	FPGA	LUT	FF	*slices* logiques	blocs RAM	fréq. MHz	temps $[k]\mathcal{P}$ ms
Bos02	GF(2^{113})	Virtex-2	n.r.	n.r.	31700	0	45	10.10
Kim04a	GF(2^{89})	Virtex-2	16459	4437	9950	0	63	0.44
Kim04b			13276	2702	7096	0	50	0.79
Kim04c			8451	2178	4995	0	51	1.02
Wol04a	GF(2^{81})	Virtex-2 Pro	n.r.	n.r.	7737	0	61	0.39
Wol04b			n.r.	n.r.	5674	0	51	0.66
Wol04c			n.r.	n.r.	4039	0	57	0.79
Wol04a	GF(2^{81})	Virtex-2	n.r.	n.r.	7785	0	57	0.42
Wol04b			n.r.	n.r.	5604	0	47	0.72
Wol04c			n.r.	n.r.	3955	0	54	0.83
Sak06a	GF(2^{83})	Virtex-2 Pro	n.r.	n.r.	2446	1*	100	0.99
Sak06b			n.r.	n.r.	4749	2*		0.55
Sak06c			n.r.	n.r.	6586	3*		0.42
Bat06	GF(2^{83})	Virtex-2 Pro	20999	n.r.	11296	0	166	0.50
Eli07a	GF(2^{113})	Virtex-2	n.r.	n.r.	25911	0	47	2.12
Eli07b			n.r.	n.r.	25271	0	45	2.03
Fan08	GF(2^{83})	Virtex-2	n.r.	n.r.	2316	6	125	0.31
Sgh16	GF(2^{83})	Virtex-2	n.r.	n.r.	5734	n.r.	145	0.30
		Virtex-5	n.r.	n.r.	5086	0	175	0.29
Ali17	GF(2^{83})	Virtex-6	n.r.	n.r.	3061	0	n.r.	0.33

TABLE 4.1 – Résultats d'implantations de HECC sur FPGA présentés dans la littérature.

Il n'existe à notre connaissance qu'une seule implantation matérielle de HECC utilisant la Jacobienne d'une courbe définie sur GF(P) (ici, P est un premier de 81 bits). Celle-ci a été proposée en 2015 par Ahmadi et coll. dans [AAKPM15] pour un ASIC 0.13 μm. Les résultats d'implantations rapportés font état d'une fréquence de 1 MHz pour un temps de calcul de la multiplication scalaire de 502.8 ms.

4.2.2 Implantations de HECC utilisant les surfaces de Kummer (KHECC)

En 2016, Renes et coll. décrivent dans [RSSB16] la première implantation *logicielle* de cryptosystèmes complets utilisant les surfaces de Kummer pour accélérer le calcul de la multiplication scalaire sur la Jacobienne d'une courbe hyperelliptique définie dans GF(P).

Dans [RSSB16], la multiplication scalaire sert de base à deux implantations de protocoles cryptographiques : protocole d'échange de clés de type Diffie-Hellman et un protocole de signature numérique. L'utilisation de la surface de Kummer permet d'obtenir des gains de performances significatifs par rapport aux meilleures implantations de ECC de l'état de l'art pour un niveau de sécurité théorique équivalent de 128 bits. Par exemple, le nombre de cycles requis pour l'échange de clé Diffie-Hellman est réduit de 27 % sur Cortex M0 et de 32 % sur AVR ATmega. Pour la signature, le gain est encore plus impressionnant, avec une réduction du nombre de cycles de respectivement 75 et 71 % sur les deux microcontrôleurs.

Pour l'implantation des deux protocoles, Renes et coll. utilisent la surface de Kummer $\mathcal{K}_{\mathcal{C}}$ associée à la courbe hyperelliptique $\mathcal{C}_{11,-22,-19,-3}$ de Gaudry et Schost [GS12] (cf. section 2.4). Nous rappelons que

cette courbe est définie sur le corps fini GF(P) avec $P = 2^{127} - 1$ un premier de 127 bits dit *de Mersenne* dont la structure particulière permet une réduction modulaire rapide et peu coûteuse. En effet, on peut constater que

$$2^{128} \equiv 2 \bmod P$$

et donc que la relation suivante est vraie pour tout $X \in \{0, ..., 2^{256}\}$ vérifiant $X = X_h\, 2^{256} + X_m\, 2^{128} + X_l$:

$$X \equiv (4\, X_h + 2\, X_m + X_l) \bmod P. \tag{4.1}$$

La réduction dans GF(P) pour ce premier particulier ne nécessite donc que quelques décalages et additions, contrairement à la réduction dans les corps finis premiers *génériques* qui nécessite des algorithmes plus complexes (p. ex. l'algorithme de réduction modulaire de Montgomery).

Le calcul de la multiplication scalaire sur $\mathcal{K}_\mathcal{C}$ utilise une adaptation de l'algorithme de l'échelle de Montgomery de [Mon87] proposée par Renes et coll.. La fonction `crypto_scalarmult` correspondante implantée dans [RSSB16] est présentée dans l'algorithme 20.

Entrées : $k = \sum_{i=0}^{n_k-1} k_i\, 2^i \in [0, 2^{n_k})$ et $(x_\mathcal{P}/y_\mathcal{P}, x_\mathcal{P}/z_\mathcal{P}, x_\mathcal{P}/t_\mathcal{P}) \in \mathrm{GF}(m)^3$ pour $\pm\mathcal{P} \in \mathcal{K}_\mathcal{C}$.
Sortie : $(\pm[k]\mathcal{P}, \pm[k+1]\mathcal{P})$
begin
1 $\quad \pm\mathcal{V}_1 \leftarrow (a : b : c : d)$
2 $\quad \pm\mathcal{V}_2 \leftarrow$ `xUNWRAP`$(x_\mathcal{P}/y_\mathcal{P}, x_\mathcal{P}/z_\mathcal{P}, x_\mathcal{P}/t_\mathcal{P})$
3 $\quad$ **for** $i = n_k - 1$ **downto** *0* **do**
4 $\quad\quad (\pm\mathcal{V}_1, \pm\mathcal{V}_2) \leftarrow$ `CSWAP`$(k_i, (\pm\mathcal{V}_1, \pm\mathcal{V}_2))$
5 $\quad\quad (\pm\mathcal{V}_1, \pm\mathcal{V}_2) \leftarrow$ `xDBLADD`$(\pm\mathcal{V}_1, \pm\mathcal{V}_2, (x_\mathcal{P}/y_\mathcal{P}, x_\mathcal{P}/z_\mathcal{P}, x_\mathcal{P}/t_\mathcal{P}))$
6 $\quad\quad (\pm\mathcal{V}_1, \pm\mathcal{V}_2) \leftarrow$ `CSWAP`$(k_i, (\pm\mathcal{V}_1, \pm\mathcal{V}_2))$
7 $\quad$ **return** $(\pm\mathcal{V}_1, \pm\mathcal{V}_2)$

Algorithme 20 : `crypto_scalarmult` : multiplication scalaire $\pm[k]\mathcal{P}$ sur $\mathcal{K}_\mathcal{C}$ utilisant l'échelle de Montgomery et implantée par Renes et coll. dans [RSSB16].

Dans l'algorithme 20, les n_k bits du scalaire k sont parcourus itérativement en commençant par le bit de poids fort (MSB). À chacune des itérations, un couple de points $(\pm\mathcal{V}_1, \pm\mathcal{V}_2)$ sur $\mathcal{K}_\mathcal{C}$ est calculé à partir des résultats de l'itération précédente à l'aide des opérations `CSWAP` et `xDBLADD`. Pour les valeurs initiales $\pm\mathcal{V}_1 = (a : b : c : d)$ [1] et $\pm\mathcal{V}_2 = \pm\mathcal{P} = (x_\mathcal{P} : y_\mathcal{P} : z_\mathcal{P} : t_\mathcal{P})$, l'algorithme de multiplication scalaire calcule les points $\pm([k]\mathcal{P})$ et $\pm([k+1]\mathcal{P})$ sur $\mathcal{K}_\mathcal{C}$.

Dans l'algorithme de multiplication scalaire proposé par Renes et coll., l'opération d'addition–doublement différentielle `xDBLADD` représente le cœur des itérations de l'échelle de Montgomery. Cette opération au niveau courbe calcule le couple de points

$$(\pm[2]\mathcal{V}_1, \pm(\mathcal{V}_1 + \mathcal{V}_2)) = \texttt{xDBLADD}(\pm\mathcal{V}_1, \pm\mathcal{V}_2, \pm\mathcal{P})$$

à partir des points $\pm\mathcal{V}_1$, $\pm\mathcal{V}_2$ et $\pm\mathcal{P}$ (le point de base) sur $\mathcal{K}_\mathcal{C}$. `xDBLADD` se base sur un ensemble d'opérations arithmétiques dans GF(P) dont les dépendances sont illustrées en figure 4.1 : multiplications `M`, carrés `S`, et *transformées de* `Hadamard` calculées grâce à un ensemble de 8 additions $\oplus$ et soustractions $\ominus$ décrit dans l'équation 4.2.

1. les éléments a, b, c et d sont des paramètres de $\mathcal{K}_\mathcal{C}$

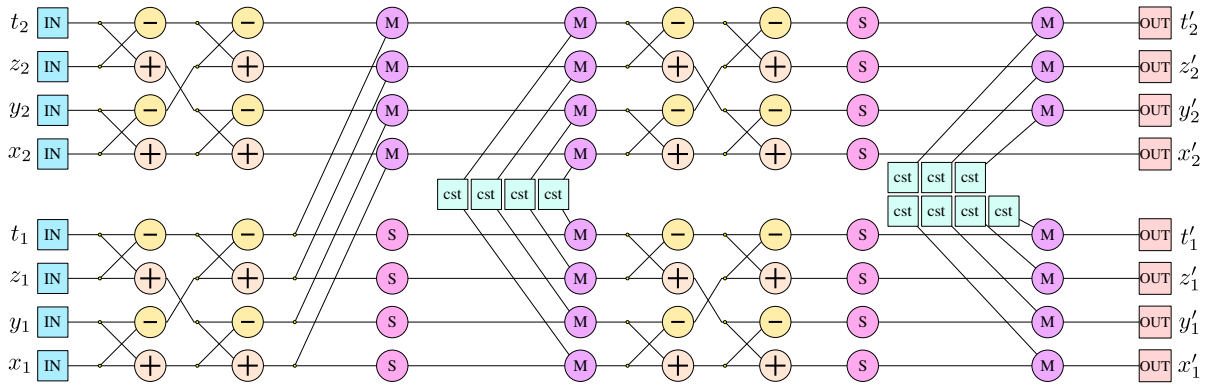


FIGURE 4.1 – Opérations au niveau corps GF(P) dans `xDBLADD` tirée de [RSSB16]. Les entrées IN correspondent aux coordonnées $(x_1 : y_1 : z_1 : t_1)$ de $\pm\mathcal{V}_1$ et $(x_2 : y_2 : z_2 : t_2)$ de $\pm\mathcal{V}_2$. Les sorties OUT correspondent aux nouvelles coordonnées $(x'_1 : y'_1 : z'_1 : t'_1)$ et $(x'_2 : y'_2 : z'_2 : t'_2)$ de $\pm\mathcal{V}_1$ et de $\pm\mathcal{V}_2$.

$$\texttt{Hadamard} : (x : y : z : t) \mapsto (x' : y' : z' : t'), \text{ avec } \begin{cases} x' = x + y + z + t, \\ y' = x + y - z - t, \\ z' = x - y + z - t, \\ t' = x - y - z + t. \end{cases} \tag{4.2}$$

Grâce à l'utilisation de la courbe $\mathcal{C}_{11,-22,-19,-3}$, certaines constantes utilisées dans le calcul de la multiplication scalaire sur $\mathcal{K}_\mathcal{C}$ sont représentables sur seulement 16 bits. Renes et coll. utilisent alors une fonction dédiée pour calculer les multiplications d'éléments de GF(P) par ces petites constantes. Cette fonction est 3.4 à 6 fois plus rapide que la fonction de multiplication de deux éléments de GF(P) en fonction du microcontrôleur utilisé. Les multiplications par ces petites constantes représentent 12 des 15 multiplications par des constantes que l'on peut trouver dans le calcul de l'opération `xDBLADD`, soit 38.7% des multiplications et carrés modulaires calculés dans cette opération.

La seconde opération intervenant dans les itérations de l'échelle de Montgomery de `crypto_scalarmult` est l'opération de `CSWAP`. Elle permet d'intervertir les paires de points en entrée et en sortie du `xDBLADD` en fonction de la valeur du bit du scalaire traité durant l'itération courante.

Enfin, on notera que le point de base $\pm\mathcal{P}$ est représenté dans `crypto_scalarmult` sous une forme compressée (trois coordonnées au lieu des quatre utilisées pour représenter les éléments sur $\mathcal{K}_\mathcal{C}$) permettant à Renes et coll. d'en réduire les coûts de stockage et de transmission. La fonction `xUNWRAP` est exécutée une fois avant le début des itérations afin de décompresser le point $\pm\mathcal{P}$ et d'en récupérer les quatre coordonnées $(x_\mathcal{P} : y_\mathcal{P} : z_\mathcal{P} : t_\mathcal{P})$, pour un coût total de 4 multiplications dans GF(P).

Une implantation matérielle de μKummer sur une plateforme Zynq-7020 de Xilinx embarquant un FPGA Artix-7 a été présentée en 2018 par Koppermann et coll. dans [KSHS18] pour le protocole d'échange de clé de Diffie-Hellmann. L'architecture proposée se compose d'une mémoire de données, d'un bloc pour le calcul des opérations dans GF(P) et d'une unité de contrôle.

La mémoire de données permet le stockage des paramètres et constantes de la courbe et des éléments intermédiaires manipulés pendant le calcul de la multiplication scalaire. Elle est basée sur une mémoire RAM distribuée double ports (implantée dans les *slices* logiques du FPGA) et sur un ensemble de registres de 127 bits. Le bloc dédié aux calculs arithmétiques est composé de plusieurs unités : 1 multiplieur,

2 additionneurs, 2 soustracteurs et 1 unité optimisée pour la multiplication par des petites constantes.

Le multiplieur modulaire est *spécifique au premier* $P = 2^{127} - 1$. Il utilise 49 *slices* DSP rectangulaires 17×24 bits du Zynq-7020 pour calculer une multiplication modulaire en 7 cycles d'horloge. Le pipeline interne de ce multiplieur permet à celui-ci de produire un résultat de multiplication modulaire par cycle. Les deux additionneurs et soustracteurs sont regroupés au sein d'un bloc conçu pour le calcul des transformées d'Hadamard. Au vue de la description de ce bloc faite par les auteurs, il ne semble pas que les additionneurs et soustracteurs puissent être utilisés de façon indépendante. Les transferts de données entre les différentes unités arithmétiques et la mémoire se font sous la forme de mots de 127 bits, soit la taille des éléments du corps. L'unité de contrôle consiste en une FSM générant à chaque cycle horloge les signaux nécessaires au pilotage de l'architecture. On notera que l'utilisation de cette FSM empêche toute modification de l'ordonnancement des calculs sans modification du *bitstream* dans le FPGA, limitant de ce fait la flexibilité de dernier.

Deux versions de l'architecture décrite ont été proposées dans [KSHS18], à savoir une version « simple cœur » (Kop18a) et une version « multi-cœur » (Kop18b). La version Kop18a implante l'architecture décrite et permet de calculer deux multiplications scalaires indépendantes en parallèle en utilisant un ordonnancement intelligent des opérations dans le coprocesseur. Dans la version Kop18b, l'architecture est répliquée quatre fois, permettant de ce fait le calcul de 8 multiplications scalaires sur des clés et des points de base différents en parallèle.

Les résultats d'implantation détaillés par Koppermann et coll. pour les deux versions de leurs coprocesseurs sont rapportés dans le tableau 4.2.

réf.	LUT	FF	*slices* logiques	*slices* DSP	BRAM	fréq. MHz	nbr. de cycles	temps μs
Kop18a	8764	6852	2657	49	0	139	11330	82
Kop18b	35015	27300	10554	196	0	129	11330	88

TABLE 4.2 – Résultats d'implantations pour les architectures KHECC Kop18a et Kop18b de [KSHS18]. Le nombre de cycles correspond à la latence d'une multiplication scalaire.

4.3 Objectifs et contraintes de nos accélérateurs matériels

Les accélérateurs matériels que nous présenterons dans le reste de ce chapitre sont dédiés au calcul de la multiplication scalaire pour KHECC. Ils sont en très grande partie inspirés des travaux sur μKummer décrits dans [RSSB16] : opérations au niveau courbe, algorithmes ou encore paramètres et constantes utilisés par exemple. Nous noterons toutefois certaines différences entre nos travaux et ceux de Renes et coll.

Contrairement à Renes et coll. et comme dans la grande majorité des travaux de la littérature, nous nous intéressons à la conception d'accélérateurs matériels pour la *multiplication scalaire* et non à l'implantation de protocoles cryptographiques complets. La multiplication scalaire est en effet l'opération la plus importante et la plus coûteuse au sein des applications embarquées de HECC, que ce soit en termes de performances de calcul ou de consommation de ressources et d'énergie. Elle est aussi cruciale dans le cadre de la protection face aux attaques par *analyse de canaux cachés* (SCA), en particulier lorsque la valeur du scalaire k doit être protégée s'il correspond à la clé secrète dans le protocole. Nous considérerons donc ici que notre accélérateur est couplé à une implémentation logicielle prenant en charge

les primitives de haut niveau, ce qui est le cas dans la plupart des cryptosystèmes embarqués utilisant (H)ECC.

Comme nous l'avons expliqué dans les chapitres précédents, nous avons pour objectif la mise en place d'accélérateurs matériels *flexibles* permettant entre autres d'augmenter la durée de vie des circuits implantés sur FPGA. Dans ces circuits, les paramètres de la courbe et du corps peuvent être *modifiés à l'exécution* pour une taille de caractéristique P donnée. Pour cette raison, et contrairement aux implantations logicielles de Renes et coll. dans [RSSB16] et à l'implantation matérielle de Koppermann et coll. dans [KSHS18], nous avons choisi d'implanter KHECC en nous basant sur des *corps finis premiers génériques* ne profitant pas de particularités permettant d'optimiser les temps de réduction modulaire.

Il est important de bien faire remarquer que ce choix augmente la complexité des opérations arithmétiques à implanter dans nos accélérateurs, tout en rendant inutilisables certaines optimisations de [RSSB16] et [KSHS18]. Par exemple, nous avons choisi d'utiliser l'algorithme de Montgomery pour les multiplications et carrés dans GF(P). Comme nous l'avons vu dans le chapitre 3, l'utilisation de cet algorithme impose de projeter les éléments de GF(P) dans le domaine de Montgomery $\{(x \times 2^m) \bmod P, \forall x \in \mathrm{GF}(P)\}$. En particulier, les « petites » constantes de 16 bits dans μKummer sont représentées dans le domaine de Montgomery par des éléments de GF(P) de 128 bits. Il nous est donc impossible d'utiliser un multiplieur optimisé pour les multiplications par ces constantes semblable à celui implanté dans [KSHS18]. On notera que ce dernier permet le calcul d'une multiplication modulaire d'un élément de 127 bits par une petite constante de 16 bits en seulement 4 cycles contre 7 cycles pour une multiplication modulo $(2^{127} - 1)$ classique. Par comparaison, le calcul d'une multiplication modulaire dans notre HTMM est d'environ 74 cycles, soit environ 18.5 fois plus long. On rappellera aussi que les multiplications par des petites constantes de 16 bits représentent 38.7% des multiplications et carrés modulaires calculés dans l'opération `xDBLADD`.

4.4 Choix des unités et exploration d'architectures d'accélérateurs

Pour la conception de nos cryptoprocesseurs nous avons choisi de nous baser sur des architectures construites à partir d'un ensemble de blocs (ou unités) fonctionnel(le)s. Nous avons alors pu nous concentrer dans un premier temps sur la sélection des unités dédiées aux opérations arithmétiques (voir la section 4.4.1). Chacune de ces unités a été complètement décrite en VHDL synthétisable, validée par un ensemble de simulations intensives et finalement implantée sur chacun des FPGA que nous avons choisi. Dans un second temps, nous nous sommes intéressés aux blocs dédiés au contrôle et à la prise en charge des données dans nos accélérateurs. Nos choix en matière de mémoires, de communications internes et de contrôle sont détaillés dans la section 4.4.2.

Rappels sur les notations

Conformément à la notation introduite dans le chapitre précédent, nous utiliserons les variables s et w pour dénoter respectivement *le nombre* et *la taille des mots dans les unités arithmétiques* dans nos cryptoprocesseurs : multiplieurs, additionneurs ou soustracteurs par exemple.

Les variables $\tilde{s}$ et $\tilde{w}$ désigneront elles respectivement *le nombre* et *la taille des mots dans les mémoires et les communications* dans nos cryptoprocesseurs.

De petites interfaces séries–parallèles placées en entrée et en sortie des unités permettront de passer d'une décomposition à l'autre quand $w \neq \tilde{w}$.

4.4.1 Choix des unités arithmétiques

Les unités arithmétiques sont dédiées au calcul des opérations au niveau du corps fini GF(P) (addition, soustraction et multiplication modulo un nombre premier générique P), ainsi qu'à l'opération `CSWAP`. Ces différentes unités sont détaillées ci-après.

Multiplieur (`Mult`) :

Pour le calcul des multiplications et carrés modulaires dans les accélérateurs KHECC de [GCT17], nous avons choisi d'implanter le multiplieur modulaire hyper-threadé HTMM, décrit au chapitre 3. Dans [GCT17], nous avons utilisé la première version non optimisée du HTMM présentée dans [GT17d] dont les caractéristiques sont les suivantes :

- les opérandes de $m = 128$ bits sont décomposés en $s = 4$ mots de $w = 34$ bits ;
- $\sigma = 3$ LM sont utilisés pour calculer jusqu'à 3 MMM indépendantes simultanément ;
- 11 *slices* DSP (17×17 bits) et 2 BRAM sont utilisés ;
- la latence λ pour une MMM est de 69 cycles.

Les versions optimisées de notre HTMM de [GT18a], proposées dans le chapitre 3, n'ont pas pu être utilisées dans les architectures de [GCT17] car elles ont été implantées plus tardivement. Toutefois, nous proposons dans la section 4.7 quelques nouvelles implantations de nos accélérateurs utilisant la version F44B optimisée du HTMM.

Comme évoqué précédemment, la projection des éléments de GF(P) dans le domaine de Montgomery rend inutilisable les optimisations pour la multiplication par des petites constantes de 16 bits. Nous n'utiliserons donc pas d'unité matérielle dédiée aux multiplications par ces constantes. Pour ce qui est du calcul des carrés dans GF(P), il n'existe pas à notre connaissance d'algorithme permettant d'accélérer de façon significative le calcul du carré d'un opérande dans MD. Dans [HMV04] (pp. 34-35), les auteurs expliquent qu'il est possible d'utiliser un algorithme optimisé pour le calcul du carré dans $\mathbb{N}$ avant de réduire le résultat modulo P. Cette méthode permet de réduire la complexité du calcul du carré, avec un nombre de produits partiels divisé par 2 par rapport à la multiplication. Elle ne permet toutefois pas l'entrelacement des étapes de calcul du carré avec celles de la réduction et nécessite plus de mémoire. En termes d'implantation matérielle, cette méthode nécessite aussi d'ajouter une nouvelle unité arithmétique dans nous unités. Nous avons donc décidé dans un premier temps de limiter la surface de circuit consommée et la complexité du contrôle dans nos architectures en utilisant l'unité HTMM `Mult` pour le calcul des carrés modulaires.

On notera ici que, de façon similaire aux solutions de l'état de l'art, la conversion finale des valeurs depuis le domaine de Montgomery ($0 \leq x < 2P$) vers le corps fini GF(P) ($0 \leq x \leq P$) est effectué à la fin de la multiplication scalaire. La conversion d'une valeur $\tilde{a}$ de MD vers GF(P) nécessite le calcul d'une unique MMM :

$$\begin{aligned} a &= \mathrm{MMM}(\tilde{a}, 1) \\ &= a\,2^m \times 1 \times 2^{-m} \bmod P \\ &= a \bmod P. \end{aligned}$$

Son impact sur le temps de calcul total de la multiplication scalaire est donc minime : moins de 200 cycles pour la conversion des 4 coordonnées respectives des points $\pm\mathcal{V}_1$ et $\pm\mathcal{V}_2$ sur les 100 à 200 mille cycles nécessaires au calcul de la multiplication scalaire. N'impliquant pas données secrètes, elle n'a d'ailleurs pas non plus d'impact sur les aspects de robustesse aux attaques par observation.

Additionneur (`AddSub`) :

Nous utilisons une unité unifiée `AddSub` pour le calcul des opérations d'addition et de soustraction modulaires $(x \pm y) \bmod 2P$ (la borne supérieure de $2P$ vient du domaine de Montgomery). L'opération exécutée pour une paire d'opérandes donnée est sélectionnée grâce à un signal de contrôle permettant de choisir le mode de fonctionnement de l'unité (ici additionneur ou soustracteur) en début de calcul.

Le résultat l'addition des opérandes A et B est compris entre 0 et $4P$ pour A et B entre 0 et $2P$. L'addition modulaire $(A+B) \bmod 2P$ peut alors être calculée simplement en soustrayant $2P$ au résultat si ce dernier est compris entre $2P$ et $4P$. Afin d'évaluer rapidement les bornes de $A+B$, nous calculons en parallèle les valeurs $R = A+B$ et $R' = A+B-2P$ et nous nous basons sur la retenue sortante de cette dernière opération pour déterminer si R' est négatif. Dans le cas où $R' < 0$, le résultat de $(A+B) \bmod 2P$ est $R = A+B$. Sinon, le résultat de $(A+B) \bmod 2P$ est $R' = A+B-2P$.

La soustraction modulaire est calculée selon le même principe, en calculant les valeurs $R = A-B$ et $R' = A-B+2P$ et en se basant sur la retenue sortante de la première opération pour déterminer si $A-B$ est négatif. Dans le cas où $R < 0$, le résultat de $(A-B) \bmod 2P$ est $R' = A-B+2P$. Sinon, le résultat de $(A-B) \bmod 2P$ est $R = A-B$.

Dans l'unité `AddSub`, les soustractions par B ou par P sont calculées à partir de leur complément à 2, notés respectivement $\overline{B}$ et $\overline{2P}$. On calcule alors $R = A+B$ et $R' = A+B+\overline{2P}$ pour l'addition modulaire, et $R = A+\overline{B}$ et $R' = A+\overline{B}+2P$ pour la soustraction modulaire. Par conséquent, notre unité `AddSub` repose sur deux additionneurs pour les calculs de R et R' (R' est calculé en sommant R à P ou $\overline{2P}$). Les compléments à 2 de B et P sont calculés en inversant les bits de B et P et en forçant à 1 la valeur de la retenue entrante dans l'additionneur concerné. Le choix du mode additionneur ou soustracteur dans l'unité `AddSub` ne repose quant à lui que sur quelques multiplexeurs et signaux de contrôle.

Clairement, le chemin critique dans le pipeline interne de notre unité est le calcul de la retenue sortante de $R' = A+B-P$ (pour l'addition) ou de $R = A-B$ (pour la soustraction) qui est nécessaire à la sélection du résultat valide. L'impact du calcul de la retenue sur les performances de notre unité est encore plus important lorsqu'on utilise une décomposition interne des $m = 136$ bits des opérandes en s mots de w bits.[2] En effet, une telle décomposition nous force à augmenter la profondeur du pipeline interne de w étages pour retarder la sortie du premier mot de poids faible du résultat tant que les derniers mots de poids fort des opérandes n'ont pas été traités et la retenue sortante calculée.

Nous avons exploré l'impact de différentes décompositions internes des opérandes sur les performances de notre `AddSub` pour les jeux de paramètres suivants : $s = 4$ mots de $w = 34$ bits, $s = 2$ mots de $w = 68$ bits ou $s = 1$ mot de $w = 136$ bits. Pour $w = 68$ ou 136 bits, nous avons observé une baisse significative de la fréquence globale de l'accélérateur (provoquée par de longues propagations de retenues dans les additionneurs de `AddSub`). Par exemple, la fréquence maximale atteinte pour un additionneur d'entiers de 34 bits implanté sur Spartan-6 est de 429 MHz. Cette fréquence sur le même FPGA diminue quand la taille des opérandes augmente : 318 MHz pour un additionneur 68 bits, et 240 MHz pour un additionneur 136 bits. Il semblerait que les additionneurs pouvant être implantés dans les FPGA que nous avons sélectionnés soient optimisés pour des tailles de mots proches de 32 bits mais pas pour des tailles supérieures. Nous avons donc choisi un découpage interne en mots de 34 bits des éléments de GF(P) dans `AddSub`, comme c'est le cas dans le HTMM de `Mult`. Cela nous permet de limiter la longueur des chemins combinatoires et d'obtenir des fréquences élevées au sein de l'opérateur.

2. les notations s, w et m utilisées pour `AddSub` sont les mêmes que pour HTMM mais les valeurs des paramètres correspondant peuvent différer entre les deux unités.

Unité de `CSWAP` :

L'opération de `CSWAP` utilisée dans la fonction `crypto_scalarmult` de [RSSB16] (algo. 20) intervertit les points $\pm\mathcal{V}_1$ et $\pm\mathcal{V}_2$ au début et à la fin de chaque itération de l'échelle de Montgomery en fonction de la valeur des bits du scalaire k parcourus successivement.

Plusieurs solutions sont possibles pour l'implantation de cette opération. Nous avons tout d'abord envisagé d'intégrer la gestion du scalaire et l'échange des points $\pm\mathcal{V}_1$ et $\pm\mathcal{V}_2$ *au sein du contrôle* de nos accélérateurs, mais avons finalement décidé de concevoir et d'implanter une unité arithmétique dédiée au `CSWAP`. L'utilisation d'une telle unité nous a permis de décorréler le décodage des instructions de la valeurs des bits du scalaire k, et ainsi de limiter les risques de fuites d'information sensible dans le contrôle de nos accélérateurs. Elle nous a aussi permis de « centraliser » la gestion du scalaire au sein d'une unique unité, facilitant de ce fait la mise en place de protections au niveau matériel.

L'unité de `CSWAP` reçoit simultanément en entrée deux éléments de GF(P) (chacun correspondant à l'une des coordonnées x, y, z ou t de $\pm\mathcal{V}_1$ et de $\pm\mathcal{V}_2$) et les intervertit, ou pas, en fonction de la valeur du bit du scalaire traité à l'itération courante. Les deux coordonnées des points résultats sont générées simultanément sur la sortie de l'unité. Après avoir fini de traiter le dernier bit de clé, l'unité de `CSWAP` active un signal `key_done` indiquant que tous les bits du scalaire k ont été parcourus. Ce signal permet de notifier à l'accélérateur la fin de la dernière itération de l'échelle de Montgomery et donc de la multiplication scalaire.

Pour améliorer la résistance de nos accélérateurs faces aux attaques par SCA, nous avons conçu l'unité de `CSWAP` de façon à décorréler au maximum l'activité électrique au sein de l'opérateur de la valeur des bits du scalaire. Pour cela, nous avons choisi de décomposer les opérandes du `CSWAP` en mots de $w = 34$ bits transférés en s cycles. Les transferts des points $\pm\mathcal{V}_1$ et $\pm\mathcal{V}_2$ en entrée et en sortie de l'unité sont effectués en parallèle et pipelinés avec le fonctionnement interne de cette dernière. Ainsi, nous pouvons « masquer » l'activité au sein de l'unité avec les communications (c.-à-d. le transfert des s mots mémoire depuis et vers l'unité). Notre unité de `CSWAP` possède en outre un comportement *uniforme* et à *temps constant* (le nombre et la durée des opérations internes sont indépendants des valeurs des opérandes et de celle des bits du scalaire). Finalement, on notera que les adresses mémoires auxquelles sont lus et écrits les opérandes et résultats du `CSWAP` sont indépendantes de la valeur des bits du scalaire. Cela qui nous permet là encore de limiter les fuites au niveau du contrôle.

4.4.2 Contrôle, mémoire et communications

Nous avons choisi de concevoir des architectures de type Harvard, classiquement utilisées dans la littérature pour les cryptoprocesseurs (H)ECC. Ces architectures reposent sur une mémoire de données, un ensemble d'unités arithmétiques, un réseau d'interconnexion, et une unité de contrôle associée à une mémoire de code. Le modèle Harvard a l'avantage de permettre la conception d'architectures modulables dans lesquelles la modification, l'ajout ou le retrait d'unités arithmétiques ou de blocs fonctionnels peut se faire relativement facilement. Cette propriété a été l'une de nos principales motivations dans le choix de ce modèle. Elle nous a permis d'explorer et d'implanter de nombreuses architectures différentes, pour différentes sélections de paramètres.

Dans les architectures Harvard, la mémoire de données et le réseau d'interconnexion servent respectivement au stockage d'éléments de GF(P) (opérandes et résultats d'opérations arithmétiques ou constantes et paramètres de courbes par exemple) et à leurs transferts vers/depuis les différentes unités arithmétiques. L'unité de contrôle génère ensemble de signaux permettant de contrôler les accès à la mémoire

config.	$\tilde{w}$ bits	$\tilde{s}$ mots	nbr. de cycle(s) par opération mémoire	BRAM
`w̃34`	34	4	4	1
`w̃68`	68	2	2	2
`w̃136`	136	1	1	4

TABLE 4.3 – Configurations mémoires explorées pour nos accélérateurs. Les éléments de GF(P) sur 136 bits sont décomposés en $\tilde{s}$ mots de $\tilde{w}$ bits dans les mémoires et le réseau d'interconnexion.

de données, les communications entre les différentes unités ainsi que le fonctionnement de ces dernières.

Mémoire :

Dans nos accélérateurs pour KHECC nous utilisons une ou plusieurs mémoire(s) interne(s) pour stocker les valeurs intermédiaires (c.-à-d. les coordonnées dans GF(P) des points manipulés), les paramètres de la courbe et les constantes nécessaires au calcul de la multiplication scalaire. Ces mémoires sont conçues de façon à utiliser efficacement les mémoires BRAM câblées des FPGA de Xilinx et pouvant stocker des mots de 1, 2, 4, 9, 18 ou 36 bits. Nous avons choisi une configuration dans laquelle la taille $\tilde{w}$ des mots mémoire est un multiple de la taille $w = 34$ bits des mots dans nos unités `Mult` et `AddSub`, ce qui nous a permis de simplifier le contrôle dans l'architecture et d'éviter l'implantation d'interfaces complexes en entrée et sortie des différentes unités arithmétiques. Nos BRAM sont alors configurées avec une largeur de 36 bits, dont les 2 bits de poids fort sont inutilisés, et une profondeur de 512 mots.

La décomposition des 136 bits des éléments de GF(P) en mots mémoire plus petits permet de réduire le nombre de BRAM parallèles dans la mémoire de nos accélérateurs, plusieurs mots du même élément pouvant être lus ou écrits de façon séquentielle dans les mêmes BRAM. Elle impacte en contrepartie le nombre de cycles nécessaires à l'accès en lecture ou en écriture des éléments stockés en mémoire.

Par exemple, pour une décomposition en mots mémoire de $\tilde{w} = 34$ bits, 1 opération mémoire (c.-à-d. lecture ou écriture d'un élément de GF(P) de 136 bits) nécessite 4 lectures (ou écritures) séquentielles de mots de 34 bits, requérant un total de 4 cycles.

Nous avons exploré 3 décompositions différentes pour le stockage des éléments de GF(P) (et pour les communication internes), décrites dans le tableau 4.3. L'ensemble des éléments de GF(P), des constantes et des paramètres à stocker en mémoire pour KHECC nécessite moins de 512 mots mémoire, quelle que soit la décomposition utilisée (configurations mémoires `w̃34`, `w̃68` ou `w̃136`). Le nombre de BRAM implantées pour chaque configuration mémoire explorée est donné dans le tableau 4.3, de même que le nombre de cycles par opération mémoire sur des éléments complets de GF(P).

Pour des raisons de sécurité, la mémoire interne de nos accélérateurs ne peut pas être accédée directement depuis l'extérieur. Les entrées/sorties dans nos accélérateurs sont prises en charge par une petite unité dédiée, désactivée pendant le calcul des opérations de multiplication scalaire. En raison de l'impact très réduit de cette unité sur la surface et les performances des architectures proposées, et pour simplifier la description de ces dernières, nous ne la mentionnerons pas dans le reste de ce chapitre.

Communications internes :

Les différentes unités et la mémoire communiquent via un réseau d'interconnexion basé sur un ensemble de multiplexeurs. Nous avons privilégié ce type de structure par rapport à des structures de bus pour des

raisons de performances (faible efficacité des bus sur les FPGA) mais aussi pour des raisons de sécurité. Les bus ont en effet tendance à générer des variations d'activité électrique importantes qui peuvent avoir un impact néfaste pour la protection contre les attaques par SCA.

Afin d'explorer différents compromis entre performances de calcul et coût en surface de circuit, nous avons implanté différentes configurations du système de communication en nous basant sur les décompositions présentées dans le tableau 4.3. Les réseaux d'interconnexion implantés dans nos différentes architectures sont décrits plus en détail dans la section 4.5. Nous avons choisi d'utiliser la même décomposition (`w̃34`, `w̃68` ou `w̃136`) des éléments de GF(P) dans la mémoire et dans les communications afin de simplifier le transfert des éléments de GF(P) entre la mémoire et le réseau d'interconnexion.

L'échange des opérandes et résultats entre le réseau d'interconnexion et les unités `Mult` et `AddSub` (utilisant une décomposition interne en $s = 4$ mots de $w = 34$ bits) est naturel pour la configuration `w̃34` dans laquelle $\tilde{w} = 34$ bits. Pour les configurations `w̃68` et `w̃136`, nous avons dû rajouter des petites interfaces séries–parallèles au niveau des entrées et sorties des unités arithmétiques, prenant en charge la conversion des mots de $\tilde{w}$ bits en mots de w bits (et inversement).

Contrôle de nos accélérateurs :

Le contrôle supervise entre autres les accès mémoires, les communications et le calcul des différentes opérations dans nos accélérateurs durant le calcul de la multiplication scalaire de KHECC. Dans nos accélérateurs, la mise en place de ce contrôle est simple comparé à celui d'un processeur généraliste « classique » : les latences dans les différents opérateurs sont fixes, l'ordonnancement des opérations étant lui indépendant des données calculées (c.-à-d. nous n'avons pas besoin d'instructions conditionnelles complexes) et donc statique à l'exécution. Les principales tâches prises en charge par le contrôle sont :

- la génération des différents signaux de contrôle des unités arithmétiques (activation ou définition du mode de fonctionnement par exemple) ;
- le contrôle des accès mémoires (définition des adresses accédées et du mode d'accès) pour la lecture (resp. écriture) des opérandes (resp. résultats) vers (resp. depuis) les unités arithmétiques ;
- la gestion des itérations de l'échelle de Montgomery et la fin de la multiplication scalaire.

Nous avons envisagé plusieurs solutions de contrôle pour nos accélérateurs parmi lesquels des solutions centralisées à base d'automates finis (FSM) ou de programmes et des solutions décentralisées dans lesquelles plusieurs unités de contrôle locales basées sur des petites FSM prennent en charge des blocs matériels indépendants. Notre choix s'est porté sur un contrôle centralisé, comme on en trouve dans la majorité des solutions de l'état de l'art, basé sur un petit jeu d'instructions (ISA pour *instruction-set architecture* en anglais). Cette solution permet la mise en place d'architectures flexibles dans lesquelles le programme peut être modifié facilement à l'exécution sans avoir à reconfigurer le FPGA (contrairement aux architectures basées sur des FSM). L'ISA se compose d'instructions représentées sur 36 bits contenant : 1 opcode de 4 bits, 1 adresse d'unité sur 3 bits, 1 mode d'opération sur 2 bits, 2 adresses mémoire sur 9 bits chacune et 1 immédiat de 9 bits. Les différentes instructions définies dans l'ISA sont listées dans le tableau 4.4. Elles sont lues depuis une *mémoire de code* et *décodées* dans l'unité de contrôle afin de générer un ensemble de signaux pilotant les différentes unités, la mémoire et le réseau d'interconnexion. L'accès à la mémoire de code et le décodage des instructions sont pipelinés et nécessitent 2 cycles chacun.

Dans le cadre de nos applications de KHECC, les programmes pour le calcul de la multiplication scalaire font moins de 512 instructions de 36 bits. La mémoire de code peut donc être implantée en utilisant une unique BRAM des FPGA.

opcode	instruction	description
`0b0000`	`jump`	Fait pointer le compteur de programme (PC) vers l'adresse `immediat` si le signal `key_done` est inactif
`0b0001`	`read`	Transfert les opérandes en mémoire vers l'unité cible (pendant $\tilde{s}$ cycles) et lance le calcul dans l'unité
`0b0010`	`write`	Transfert le résultat de l'unité cible vers la mémoire (pendant $\tilde{s}$ cycles)
`0b0011`	`wait`	Attends pendant `immediat` cycles
`0b0111`	`nop`	Pas d'opération (attends pendant 1 cycle)
`0b1111`	`end`	Déclenche la fin de la multiplication scalaire

TABLE 4.4 – Jeu d'instructions défini pour nos accélérateurs (le signal `key_done` est généré par l'unité de `CSWAP` en fin de multiplication scalaire).

Pour le décodage des instructions, nous avons implanté des *boucles matérielles* utilisant de petites FSM. Ces dernières permettent de gérer les $\tilde{s}$ cycles nécessaires aux opérations mémoires et de transferts (instructions `read` et `write`) ainsi que la durée d'attente pour l'instruction `wait`. Pour des raisons de sécurité, le décodage des instructions ne prend pas en charge ni ne dépend de la valeur des bits du scalaire (les 256 bits du scalaire sont gérés exclusivement dans le `CSWAP`).

Nous avons aussi mis au point un outil en développé en `Python` nous permettant d'explorer automatiquement divers ordonnancements des opérations arithmétiques, des accès mémoires et des communications internes dans nos accélérateurs en fonction du nombre et du type des unités utilisées ainsi que des paramètres sélectionnés pour la mémoire et le réseau d'interconnexion.

Notre outil d'ordonnancement utilise pour l'instant un algorithme de type « glouton » visant à ordonnancer les opérations dans les multiplieurs au plus vite en raison de la grande latence de ces derniers. Dans le cas où plusieurs opérations peuvent être exécutées au même moment dans une unité, notre algorithme sélectionne et ordonnance aléatoirement l'une d'entre elles. L'ordonnancement final obtenu peut alors être sous optimal. Pour limiter l'impact de cette sélection aléatoire des opérations, notre outil effectue un grand nombre d'ordonnancements pour les mêmes jeux de paramètres d'architecture et sélectionne le meilleur d'entre eux.

Par exemple, le meilleur ordonnancement trouvé pour la petite architecture (A2, $\tilde{\mathtt{w}}\mathtt{34}$) présentée en section 4.5.2 permet le calcul des opérations de `CSWAP` et de `xDBLADD`en 961 cycles. Cet ordonnancement est trouvé après 96 *runs* de notre outil. En comparaison, le 1$^{\text{er}}$ ordonnancement trouvé nécessite 1041 cycles pour le calcul de ces opérations. Le temps de calcul pour le moins bon ordonnancement trouvé est lui de 1137 cycles. Enfin, le temps de calcul médian obtenu pour 100 ordonnancements est de 1022.0 cycles et le temps moyen est de 1022.16 cycles.

Le meilleur ordonnancement produit par notre outil pour les paramètres d'un accélérateur donné est utilisé pour générer le programme exécuté dans cet accélérateur. Cet ordonnancement nous fournit aussi le nombre total de cycles nécessaires au calcul d'une multiplication scalaire pour un programme et une architecture donnés, ce qui nous permet d'évaluer et de comparer rapidement les performances de différentes solutions.

@	instruction						décodage
	opcode	unit.	mode	adresse A	adresse B	`immediat`	
0	0001	011	00	000101000	000011000	000000000	`read`(3,0,40,24)
1	0010	011	00	000111000	000110000	000000000	`write`(3,56,48)
2	0001	011	00	000100100	000000100	000000000	`read`(3,0,36,4)
3	0010	011	00	000111100	000110100	000000000	`write`(3,60,52)
4	0011	000	00	000000000	000000000	000000100	`wait`(4)
5	0001	001	00	000110100	000110000	000000000	`read`(1,0,52,48)
6	0001	001	01	000110100	000110000	000000000	`read`(1,1,52,48)
7	0010	001	00	001000000	001000000	000000000	`write`(1,64)
8	0010	001	00	001000100	001000100	000000000	`write`(1,68)
⋮							
102	0011	000	00	000000000	000000000	000000010	`wait`(2)
103	0000	000	00	000000000	000000000	000000101	`jump`(4)
104	0001	010	00	000111000	000110000	000000000	`read`(2,0,56,48)
105	0001	010	00	000111100	000110100	000000000	`read`(2,0,60,52)
106	0001	010	00	000110100	000111100	000000000	`read`(2,0,52,60)
107	0001	010	00	000110000	000111000	000000000	`read`(2,0,48,56)
108	0111	000	00	000000000	000000000	000000000	`nop`
109	1111	000	00	000000000	000000000	000000000	`end`

TABLE 4.5 – Extrait d'un programme de calcul d'une multiplication scalaire ($\pm[k]\mathcal{P}$) dans l'un de nos accélérateurs.

Exemple de programme pour la multiplication scalaire :

Afin d'illustrer la lecture et le décodage des instructions dans le contrôle de nos accélérateurs, nous listons dans le tableau 4.5 les premières et dernières instructions d'un programme pour le calcul d'une multiplication scalaire exécuté dans un accélérateur intégrant les unités suivantes :

unit. 0 : `Mult` pour la multiplication (mode de fonctionnement unique) ;

unit. 1 : `AddSub` pour l'addition (mode 0) et la soustraction (mode 1) ;

unit. 2 : registre de $\tilde{w}$ bits (pour la lecture des coordonnées de $\pm[k]\mathcal{P}$) ;

unit. 3 : unité de `CSWAP`.

Mem. : mémoire de données dans laquelle 2 mots de $\tilde{w}$ bits peuvent être lus *ou* écrits simultanément.

L'exécution du programme dans le tableau 4.5 commence par le décodage de l'instruction à l'adresse 0, puis le PC est incrémenté à chaque instruction :

\# *Préambule :*

0. lecture des opérandes aux adresses mémoires 40 et 24 vers `CSWAP` ;
1. écriture des résultats du `CSWAP` en mémoire aux adresses 56 et 48 ;
2. lecture des opérandes aux adresses 36 et 4 vers `CSWAP` ;
3. écriture des résultats du `CSWAP` aux adresses 60 et 52 ;

\# *Corps des itérations de l'échelle de Montgomery :*

4. 4 cycles d'attente ;
5. lecture des opérandes aux adresses 52 et 48 vers `AddSub` (addition) ;
6. lecture des opérandes aux adresses 52 et 48 vers `AddSub` (soustraction) ;
7. écriture du résultat de `AddSub` (addition) à l'adresse 64 ;

8. écriture du résultat de `AddSub` (soustraction) à l'adresse 68 ;

⋮

102. 2 cycles d'attente ;
103. PC = 4 si `key_done` n'est pas actif, sinon PC = 104 ;

 # Terminaison (signal `key_done` *actif) :*

104–107. lecture des coordonnées $(x : y : z : t)$ du point $\pm[k]\mathcal{P}$ vers le registre de sortie de l'accélérateur (seules les adresses A sont utilisées) ;

108–109. fin de la multiplication scalaire après 1 cycle d'attente.

4.4.3 Outils pour l'exploration d'architectures

À partir de l'ensemble des unités sélectionnées et décrites dans les sections 4.4.1 et 4.4.2, nous pouvons construire des architectures d'accélérateurs KHECC pour le calcul de la multiplication scalaire. Dans les applications de KHECC, les opérations au niveau courbe sont plus complexes que pour ECC, avec un plus grand niveau de parallélisme entre les opérations internes dans GF(P).

La conception d'accélérateurs performants implique alors de déterminer le nombre d'opérateurs arithmétiques de chaque type à implanter afin de tirer parti de ce parallélisme pour accélérer les calculs tout en limitant les coûts matériels de nos implantations sur FPGA. Le choix des paramètres pour les unités arithmétiques impacte aussi la configuration des mémoires de données (en particulier la taille $\tilde{w}$ des mots mémoire) ou celle du routage dans le réseau d'interconnexion. En pratique, déterminer les paramètres et configurations les plus intéressantes n'est pas réalisable sans avoir au préalable implanté, simulé et évalué chacune des différentes solutions possibles sur chacun de nos différents FPGA.

Le principal problème que nous avons rencontré lors de la conception de nos cryptoprocesseurs a été la taille de l'espace de paramètres possibles, pour lequel de nombreuses architectures possibles doivent être explorées et évaluées. Nous nous sommes rapidement rendu compte que l'évaluation de ces différentes architectures après description en VHDL, validation par simulations intensives et implantation sur nos différents FPGA n'était pas non plus réalisable dans un temps raisonnable.

Nous avons donc décidé de mettre en place un *ensemble d'outils* nous permettant de valider et d'évaluer rapidement de nombreuses solutions de l'espace de conception afin d'en sélectionner les plus prometteuses pour implantation sur FPGA. Ces outils sont basés sur une représentation hiérarchique et hétérogène dans laquelle nos différentes architectures sont décrites en utilisant un *modèle haut niveau* nous permettant de les valider grâce à des simulations intensives *rapides* et d'en *estimer* les performances et coûts matériels.

Modélisation haut niveau des architectures :

L'utilisation d'un modèle haut niveau pour la description de nos architectures nous permet de nous abstraire de la phase de description en VHDL, compliquée et coûteuse en temps. Nous avons choisi de mettre en place un modèle haut niveau basé sur une spécification CCABA (pour *Critical Cycle Accurate, Bit Accurate*) inspirée du TLM (*Transaction-Level Modeling*, voir [CG03]).

Dans la spécification CCABA, les *cycles critiques* dans nos architectures sont les cycles pendant lesquels il y a des transitions au niveau des signaux de contrôle reçus ou émis par les différentes unités (arithmétiques, de mémoires ou réseau d'interconnexion) ou au niveau de leurs ports d'entrée/sortie. Dans nos modèles d'architectures, le comportement des différents signaux au niveau des interfaces des unités est donc modélisé de façon exacte au *cycle près* et au *bit près* (modélisation CABA), alors que le comportement interne de ces unités est abstrait par l'utilisation de fonctions haut niveau.

L'utilisation de cette spécification nous permet d'accélérer la phase de simulation des architectures. En effet, à l'échelle d'une multiplication scalaire complète nécessitant quelques centaines de milliers de cycles, les cycles critiques au niveau de l'architecture ne représentent qu'une petite partie du temps de calcul. La majeure partie du temps de calcul est consacré aux calculs des opérations au sein des diverses unités.

Par exemple, l'ensemble des transitions des signaux au niveau des interfaces d'entrée/sortie dans l'unité `Mult` (c.-à-d. HTMM 128 bits de [GT17d]) s'effectue en 10 cycles (2×1 cycles pour les signaux de contrôle et 2×4 cycles pour la lecture des opérandes et l'écriture du résultat), soit seulement 14.5% des 69 cycles nécessaires au calcul d'une MMM.

Grâce à la spécification CCABA qui s'affranchit de la simulation CABA des calculs internes dans les unités, nous sommes capables d'évaluer et de valider rapidement de nombreuses architectures en simulant leur comportement au niveau architectural pour un grand nombre de vecteurs de test.

Modélisation du comportement interne des unités dans les architectures :

Comme expliqué ci-dessus, le comportement interne des différentes unités implantées dans nos architectures est abstrait en utilisant des fonctions haut niveau. Ces fonctions permettent le *calcul exact* des valeurs des sorties d'une unité *à chaque cycle critique* à partir des données lues sur ses ports d'entrée, sans avoir besoin de calculer les valeurs internes pendant les cycles intermédiaires.

Étant donné que nos différentes unités arithmétiques ont préalablement été complètement décrites en VHDL, implantées sur FPGA et validées par simulation intensive, nous connaissons exactement leur comportement temporel au cycle prêt. Il en va de même pour la mémoire, utilisant les BRAM du FPGA dont le comportement est parfaitement connu.

Les cycles critiques dans une unité peuvent donc être modélisés de façon exacte : à chaque transition des valeurs en entrée de l'unité, nous pouvons déterminer exactement les cycles critiques pendant lesquels auront lieu des transitions sur ses ports de sortie. Les différentes valeurs écrites lors des cycles critiques sur les ports de sortie de l'unité sont, quant à elles, calculées mathématiquement à partir des valeurs lues sur les ports d'entrée, sans avoir besoin de décrire le fonctionnement exacte de l'unité à tous les cycles.

L'unité de contrôle et le réseau d'interconnexion sont des cas particuliers : ils ne sont pas implantés préalablement en VHDL car ils sont très fortement liés aux paramètres de l'architecture. Toutefois, leurs comportements temporels respectifs sont parfaitement connus car ils sont imposés par le choix du type et de la topologie de l'architecture ainsi que par le style de conception utilisé (profondeur du pipeline interne par exemple).

Outil de simulation pour la validation des modèles d'architecture :

Pour l'exploration et la validation de nos architectures, nous avons développé un outil de simulation CCABA basé sur le modèle haut niveau décrit précédemment.

Nous ferons remarquer ici qu'il existe dans la littérature différents langages de description permettant la modélisation de systèmes au niveau comportemental, le plus connu d'entre eux étant probablement SystemC® (cf. [IEE12] et [BDBK09]). Ce dernier se base sur un ensemble de bibliothèques `C++` supportant entre autres la modélisation en TLM de systèmes concurrents complexes et intègre un cœur de simulation.

Cependant, contrairement à `Python`, SystemC ne supporte pas nativement l'utilisation d'outils mathématiques *open source* tels que Sage, nécessaires au calcul des opérations modulaires complexes (pour le HTMM par exemple) et à la vérification des accélérateurs au niveau cryptographique. De plus, nous avons

évalué que la modélisation et la simulation de nos architectures ne nécessiterait qu'un sous-ensemble très limité des nombreuses fonctionnalités et niveaux d'abstractions proposés par SystemC pour la modélisation ou la simulation. Nous avons estimé que la difficulté de la prise en main des bibliothèques `C++` de SystemC et de l'interfaçage des codes `C++` avec Sage impliqueraient des temps de développement trop importants. On notera aussi que le *debug* de codes est bien plus complexe en `C++` qu'en `Python`.

Nous avons donc préféré implanter en `Python` notre propre outil de simulation CCABA. Au sein de cet outil, chaque unité est modélisée « à la main » en `Python`/Sage pour spécifier son comportement mathématique d'une part, et son comportement durant les cycles critiques d'autre part, ce dernier étant connu d'après la description VHDL écrite « à la main » correspondante. Grâce à l'utilisation de `Python`, la taille du code de notre simulateur est relativement petite : moins de 1900 lignes de codes pour le cœur du simulateur (incluant les différents outils d'ordonnancement et de configuration des architectures). Les codes `Python` utilisés pour modéliser les différentes unités font, quant à eux, au plus 325 lignes chacun.

Pour gérer le comportement concurrent des unités et s'assurer que les opérandes des différentes unités soient correctes lors d'un cycle critique, on évalue tout d'abord les valeurs des sorties devant être mises à jour. On propage ensuite les nouvelles valeurs vers les unités dont des entrées sont routées sur ces sorties. Enfin, pour chaque unité dont au moins l'une des entrées a été modifiée, on calcule les nouveaux résultats ainsi que les cycles critiques correspondant à l'écriture de ces résultats sur les sorties de l'unité.

Dans le simulateur, un compteur permet de garder en mémoire le cycle courant. À la fin de chaque cycle critique (c.-à-d. une fois toutes les entrées mises à jour et les futures sorties évaluées dans les unités), le compteur est mis à jour. Sa nouvelle valeur est le numéro du prochain cycle critique pendant lequel au moins une sortie d'unité doit être mise à jour avec une nouvelle valeur.

Notre outil de simulation est instrumenté afin de générer une trace de simulation dans laquelle sont indiquées les nouvelles valeurs des différents ports d'entrée et de sortie modifiés à chaque cycle critique. Nous avons vérifié et validé son bon fonctionnement sur un ensemble d'architectures de test, allant d'une simple structure de multiplieur–accumulateur avec un contrôle très simple à des architectures complètes d'accélérateurs pour KHECC.

Pour chacune de ces architectures, nous avons comparé les résultats de nombreuses simulations avec les résultats théoriques calculés par Sage pour les mêmes opérandes. Durant ces différentes simulations, avons aussi pris soin de vérifier les différents opérandes et résultats de chacune des unités, connus grâce à la trace de simulation générée. Nous nous sommes pour cela basés sur le comportement temporel de l'unité (connu et vérifié après implantation) et sur des comparaisons des résultats de l'opérateur avec les résultats théoriques pour différents jeux d'opérandes.

On notera qu'en plus de cette validation effectuée grâce à notre outil de simulation, les circuits implantés en VHDL ont aussi été complètement validés par des simulations intensives dans le logiciel iSim 14.7 de Xilinx.

Évaluation des performances des architectures proposées :

Nous pouvons rapidement *estimer* le coût en surface d'une architecture en sommant les surfaces des différentes unités instanciées dans l'accélérateur (unités arithmétiques et mémoire). Le temps de calcul d'une multiplication scalaire dans l'accélérateur est *estimé* en multipliant le nombre total de cycles (mesuré après simulation et validé par comparaison avec les valeurs obtenues via l'outil d'ordonnancement) par la période de cycle horloge dans l'unité la plus lente. L'impact du contrôle, qui a ce stade n'a pas encore été décrit en VHDL, sur les performances de l'architecture est approché en nous basant sur des

comparaisons avec des implantations d'unités similaires.

Nous effectuons ce type d'estimation pour chacune des spécifications d'accélérateurs (c.-à-d. pour chaque jeu de paramètres d'architectures) que nous devons simuler et évaluer. Nous pouvons ensuite sélectionner les solutions les plus prometteuses afin de les implanter sur nos différents FPGA (après description du contrôle et du réseau d'interconnexion en VHDL), les valider par simulations intensives et en évaluer les coûts et performances précises après placement et routage. La validation des différents accélérateurs implantés se fait par le calcul d'un grand nombre de multiplications scalaires pour des premiers et des points de base ($\pm\mathcal{P}$) différents et dont les résultats sont comparés avec des valeurs de référence calculées par Sage.

4.5 Architectures proposées

Après avoir exploré différentes spécifications de paramètres et d'architectures, nous avons sélectionné les quatre architectures listées dans le tableau 4.6 et les avons complètement décrites et implantées en VHDL sur nos différents FPGA.

La première architecture implantée est la petite architecture basique A1 qui nous a principalement servi de base de comparaison. Nous avons ensuite exploré différentes optimisations ainsi que plusieurs architectures parallèles. L'architecture A2 repose sur une optimisation de l'unité de `CSWAP` (`v2`). L'architecture A3 utilise deux opérateurs dédiés pour chaque type d'opérations arithmétiques ($\pm$ et $\times$) afin d'atteindre un meilleur parallélisme entre opérations dans GF(P) (nous rappelons que l'unité `Mult` étant un HTMM, elle permet déjà le calcul de trois MMM en parallèle). Enfin, l'architecture A4 se base sur un *cluster* d'unités parallèles à la fois pour les opérations arithmétiques et pour les opérations au niveau de la mémoire. Cette architecture utilise aussi une nouvelle version de l'unité de `CSWAP` (`v3`). Chacune des architectures A1–A4 est proposée pour les trois configurations `w̃34`, `w̃68` et `w̃136` possibles des mémoires et du réseau d'interconnexion (cf. section 4.4.2).

ressources	architectures			
	A1 (Sec. 4.5.1)	A2 (Sec. 4.5.2)	A3 (Sec. 4.5.3)	A4 (Sec. 4.5.4)
`AddSub`	1	1	2	2
`Mult`	1	1	2	2
`CSWAP`	1 `v1`	1 `v2`	1 `v2`	1 `v3`
Mémoire de données	1	1	1	2
Réseau d'interconnexion	1	1	1	2 avec pont
Mémoire de programme	1	1	1	1
Contrôle	1	1	1	1

TABLE 4.6 – Caractéristiques principales des quatre architectures KHECC implantées et évaluées. `v1`, `v2` et `v3` indiquent les versions de l'opération de CSWAP.

Les différents FPGA Xilinx que nous avons choisis pour nos implantations sont les suivants : les FPGA V4 et V5 (Virtex-4 et 5) pour les comparaisons avec l'état de l'art et un petit FPGA S6 (Spartan-6) pour l'évaluation des performances sur une cible *low-cost*. Pour rappel, les caractéristiques de ces différents FPGA sont détaillées en section 3.2 du manuscrit. On notera aussi que les implantations sur S6 permettront de futures évaluations de robustesse aux attaques SCA sur la carte SAKURA [sak13]. Nous avons utilisé le logiciel ISE 14.7 de Xilinx pour la synthèse de nos différentes architectures, le

placement–routage ayant été effectué par l'outil SmartXplorer qui nous a permis d'obtenir les meilleurs résultats d'implantation après 100 placements–routages.

Afin de comparer équitablement différentes implantations sur FPGA, il est important de rappeler que la configuration interne des *slices* logiques et des LUT est fortement dépendante de la famille et de la génération des FPGA considérés, comme le montre le tableau 3.1 en page 43. En particulier, une LUT6 étant équivalente à 4 LUT4, les *slices* logiques dans V4 et dans V5/S6 ne peuvent pas être comparés directement.

4.5.1 Architecture A1 : solution de base

L'architecture A1 est décrite en figure 4.2. Elle se base sur un modèle de processeur Harvard classique intégrant une unité de contrôle, une mémoire pour le programme, une mémoire de données, un réseau d'interconnexion et un ensemble d'unités arithmétiques. L'accélérateur correspondant est le plus petit parmi les quatre proposés, avec une unique instance de chaque type d'unité `AddSub`, `Mult`, et `CSWAP-v1` décrites en section 4.4. Les résultats d'implantations pour cette architecture sont présentés dans le tableau 4.7.

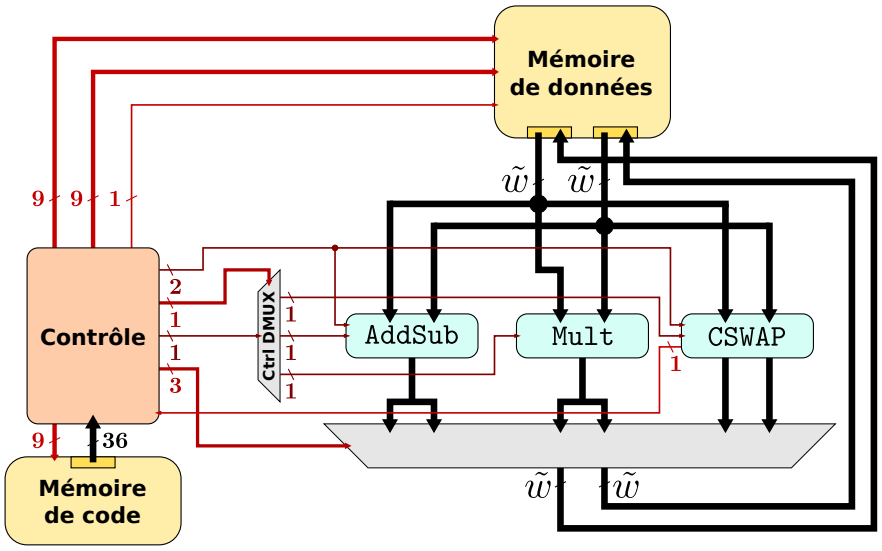


FIGURE 4.2 – Architecture A1 avec ses principales unités (arithmétiques et mémoire), le réseau d'interconnexion interne et le contrôle (les surfaces des différents blocs ne sont pas à l'échelle).

D'après les résultats dans le tableau 4.7 on peut voir que la largeur $\tilde{w}$ des communications n'a que peu d'impact sur le nombre de cycles d'horloge nécessaires au calcul d'une multiplication scalaire (au mieux 5% de réduction). Dans cet accélérateur, la majorité du temps de calcul est en effet passé dans les unités arithmétiques `Mult` et `AddSub`. Réduire la durée des accès mémoires en augmentant $\tilde{w}$ ne permet donc pas d'améliorer significativement les performances.

De plus, l'augmentation de $\tilde{w}$ impacte fortement la surface consommée en terme de LUT. Pour la version `w̃68`, on observe une augmentation de 50 à 70% du nombre de LUT comparé à la version `w̃34`, en fonction du FPGA. Cette augmentation est comprise entre 80 et 110% pour la version `w̃136` comparée à la version `w̃34`. Clairement, augmenter la largeur des communications dans le réseau d'interconnexion

augmente la consommation d'éléments logiques dans nos FPGA. Le nombre de BRAM augmente aussi avec $\tilde{w}$, en raison de l'augmentation de la largeur des mémoires dans `w̃68` et `w̃136` (voir le tableau 4.3).

FPGA	$\tilde{w}$ bits	LUT	FF	*slices* logiques	*slices* DSP	BRAM	fréq. MHz	nbr. de cycles	temps ms
V4	34	1010	1833	1361	11	4	322	194 614	0.60
	68	1750	3050	2251	11	5	305	186 911	0.61
	136	2281	3028	1985	11	7	266	184 337	0.69
V5	34	757	1816	603	11	4	360	194 614	0.54
	68	1264	3033	908	11	5	360	186 911	0.52
	136	1582	3008	940	11	7	360	184 337	0.51
S6	34	1064	1770	408	11	4	278	194 614	0.70
	68	1555	2970	705	11	5	252	186 911	0.74
	136	1910	2994	747	11	7	221	184 337	0.83

TABLE 4.7 – Résultats d'implantation de l'architecture A1 sur les différents FPGA (toutes les BRAM font une taille de 18 Kb et nous n'utilisons que les multiplieurs 17×17 dans les *slices* DSP des différents FPGA).

La relation entre $\tilde{w}$ et le nombre de FF utilisées est plus compliquée à analyser. Les versions `w̃68` et `w̃136` consomment environ 67% de FF supplémentaires par rapport à la version `w̃34`. Cela peut en partie être expliqué par le coût des interfaces séries–parallèles entre les unités arithmétiques, pour lesquelles la largeur du chemin de données internes est $w = 34$ bits, et le réseau d'interconnexion et la mémoire dans les versions `w̃68` et `w̃136`. Le nombre de FF dans ces deux dernières versions est similaire (moins de 1% d'écart quel que soit le FPGA), les interfaces dans `w̃68` étant moins larges mais aussi plus complexes à mettre en place que celles utilisées dans `w̃136`.

Quand $\tilde{w}$ augmente, on peut observer une augmentation des délais combinatoires et de la sortance logique (*fanout* en anglais) dans nos accélérateurs couplée à une diminution de leur fréquence de fonctionnement, celle-ci étant fortement liée au FPGA utilisé. Sur des FPGA plus anciens (V4 par exemple) ou plus petits (S6 p. ex.), la réduction de fréquence varie de 5 à 20%. Sur V5, plus récent et plus gros, la fréquence de 360 MHz obtenue au sein de l'accélérateur correspond à celle de l'unité la plus lente. Dans le cas d'un petit accélérateur, n'utilisant qu'une petite partie des ressources du V5, le contrôle n'impacte pas la fréquence globale.

D'après l'analyse des résultats d'implantation de notre plus petite architecture, les versions `w̃68` et `w̃136` avec des mots mémoire plus grands ne sont pas intéressantes. La réduction du nombre de cycles d'horloge pour le calcul d'une multiplication scalaire est compensée par une diminution de la fréquence globale de l'accélérateur, accompagnée d'une augmentation du nombre de LUT et de BRAM utilisées. La meilleure version pour A1 est donc toujours `w̃34`, quel que soit le FPGA considéré.

4.5.2 Architecture A2 : optimisation de l'unité de `CSWAP`

L'architecture A2 est similaire à l'architecture A1 mais avec l'unité de `CSWAP` modifiée `CSWAP-v2`. Le schéma de l'architecture A2 est le même que celui pour A1 présenté en figure 4.2.

L'implantation de l'unité de `CSWAP-v2` vient du constat suivant : dans l'échelle de Montgomery de [RSSB16] (fonction `crypto_scalarmult`), l'opération de `CSWAP` est exécutée une fois en fin de chaque itération et une fois en début de chaque itération suivante pour les mêmes opérandes $(\pm\mathcal{V}_1, \pm\mathcal{V}_2)$ et

2 bits successifs du scalaire k. Étant donné qu'aucun autre calcul n'est effectué entre ces deux opérations consécutives de `CSWAP`, nous les avons fusionnées au sein d'une opération unique afin de diviser par deux le nombre d'appels à l'unité de `CSWAP` dans l'accélérateur.

Notre version *modifiée* `CSWAP-v2` utilise 2 bits consécutifs du scalaire k : k_i et k_{i-1} (pour rappel, le scalaire k est parcouru depuis les MSB vers les LSB). Les points $\pm\mathcal{V}_1$ et $\pm\mathcal{V}_2$ en entrée sont permutés quand $k_i = k_{i-1}$, comme illustré en figure 4.3. Si $k_i = k_{i-1}$, on n'effectue pas de permutation. Cette modification de l'unité de `CSWAP` ne requiert que l'implantation d'un XOR supplémentaire pour la comparaison des 2 bits du scalaire et n'a donc pas d'impact significatif sur la surface de circuit consommée ou la vitesse de calcul.

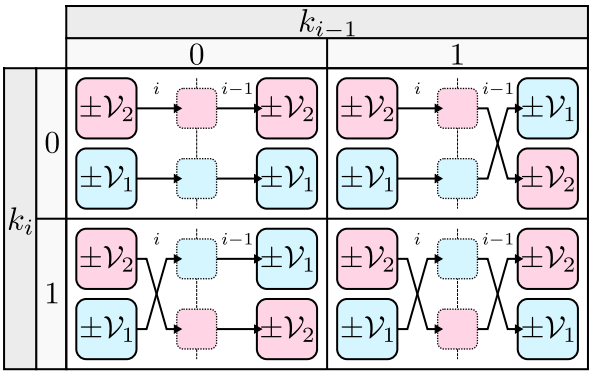


FIGURE 4.3 – Permutation des opérandes $\pm\mathcal{V}_1$ et $\pm\mathcal{V}_2$ dans l'unité `CSWAP-v2` en fonction de la valeur des bits k_i et k_{i-1} du scalaire.

Pour évaluer le tout premier `CSWAP` de la première itération de l'échelle de Montgomery, nous utilisons les bits '0' (pour le bit courant) et le MSB k_{n_k-1} du scalaire k (pour le bit suivant). De façon similaire, le dernier `CSWAP` de l'itération finale de l'échelle de Montgomery est évalué en utilisant les bits k_0 et '0'.

L'optimisation proposée ne modifie pas les propriétés de sécurité face aux SCA. L'accélérateur fonctionne toujours en temps constant (le temps de calcul ne dépend pas de la valeur des bits du scalaire) et son comportement est toujours uniforme (les opérations effectuées sont indépendantes du scalaire). Tout comme pour l'unité de `CSWAP` de base (version `v1`), nous avons conçu l'unité `CSWAP-v2` autour d'un pipeline interne possédant une activité uniforme (voir la description de l'unité de `CSWAP` dans la section 4.4.1).

L'ensemble des résultats d'implantation pour A2 sont rapportés dans le tableau 4.8. Leur analyse montre que l'impact des variations de $\tilde{w}$ dans l'architecture A2 est similaire à celui observé pour A1, avec toutefois quelques variations.

Le nombre de cycles pour le calcul d'une multiplication scalaire dans A2 est légèrement réduit par rapport à A1, en raison du nombre réduit d'appels à l'unité de `CSWAP-v2`. La fréquence est quant à elle légèrement augmentée pour la version `w̃136` : +23% sur V4 et +28% sur S6. On peut aussi constater que les variations de fréquence entre les différentes versions sont moins importantes que dans A1 sur ces FPGA. Les temps de calcul dans les accélérateurs basés sur A2 sont quant à eux réduits de 5 à 10% par rapport à A1.

Les ressources en surface (LUT, FF, *slices* logiques) consommées dans les implantations FPGA de A2 sont aussi légèrement réduites, de 5 à 13% en fonction du FPGA grâce à la simplification du contrôle interne de l'unité de `CSWAP-v2`. Les nombres de *slices* DSP et de BRAM ne sont quant à eux pas impactés

FPGA	$\tilde{w}$ bits	LUT	FF	*slices* logiques	*slices* DSP	BRAM	fréq. MHz	nbr. de cycles	temps ms
V4	34	872	1624	1121	11	4	330	184 374	0.56
	68	1556	2637	1978	11	5	290	183 071	0.63
	136	2161	3027	2100	11	7	327	183 057	0.56
V5	34	722	1605	541	11	4	360	184 374	0.51
	68	1196	2620	840	11	5	360	183 071	0.51
	136	1419	3009	944	11	7	360	183 057	0.51
S6	34	940	1559	381	11	4	293	184 374	0.63
	68	1503	2565	553	11	5	262	183 071	0.70
	136	1890	2981	667	11	7	283	183 057	0.65

TABLE 4.8 – Résultats d'implantation de l'architecture A2 sur les différents FPGA (toutes les BRAM font une taille de 18 Kb et nous n'utilisons que les multiplieurs 17×17 dans les *slices* DSP des différents FPGA).

par l'optimisation proposée. Il sont donc identiques entre A1 et A2.

Les résultats d'implantation pour l'architecture A2 montrent donc que la meilleure version est toujours `w̃34` quel que soit le FPGA considéré. On notera que le gain de temps de calcul d'environ 0.8% dans les versions `w̃68` et `w̃136` sur V5 est largement compensé par l'augmentation des ressources en surface consommées. Les implantations correspondantes ne peuvent donc pas être considérées comme étant plus efficaces que les implantations de `w̃34`.

L'optimisation de l'unité de `CSWAP-v2` permet d'obtenir une architecture un peu plus performante que l'architecture de base A1 avec environ 10% de gain de temps de calcul et de surface consommée.

4.5.3 Architecture A3 : augmentation du nombre d'unités arithmétiques

L'architecture A3 représentée en figure 4.4 intègre un plus grand nombre d'unités arithmétiques : 2 `AddSub` et 2 `Mult`. Dans cette architecture, on peut donc calculer 2 additions/soustractions en même temps et jusqu'à 6 multiplications/carrés modulaires en parallèle, contre 3 seulement dans A1 et A2, en utilisant le HTMM de [GT17d].

Le tableau 4.9 présente les résultats d'implantation pour A3. Le comportement des différentes versions `w̃34`, `w̃68` et `w̃136` pour cette architecture diffère de celui des architectures A1 et A2.

L'ajout d'une unité `AddSub` et d'une unité `Mult` provoque une augmentation de 60 à 90% du nombre de LUT utilisées en fonction du FPGA et de la valeur de $\tilde{w}$. De plus, la deuxième unité `Mult` augmente de 11 le nombre de *slices* DSP et de 2 le nombre de BRAM utilisés dans l'accélérateur. Cette augmentation de surface confirme que les unités dédiées aux calculs dans GF(P) sont les plus coûteuses en ressources dans nos architectures.

Les fréquences globales obtenues dans A3 sont légèrement plus basses que dans A2 en raison de l'augmentation de la sortance logique (*fanout*) et de la mise en place d'un contrôle plus complexe, toutes deux dues à l'augmentation du nombre d'unités dans l'architecture. La chute de fréquence inhérente à l'augmentation de $\tilde{w}$ est faible : moins de 4% sur V4 et V5, et environ 15% sur S6.

Dans l'architecture A3, l'ajout des unités `AddSub` et `Mult` supplémentaires permet d'augmenter le nombre d'opérations arithmétiques (addition/soustraction, multiplications et carrés) calculables en parallèle. Toutefois, l'augmentation du parallélisme d'opérations dans l'accélérateur entraîne une augmen-

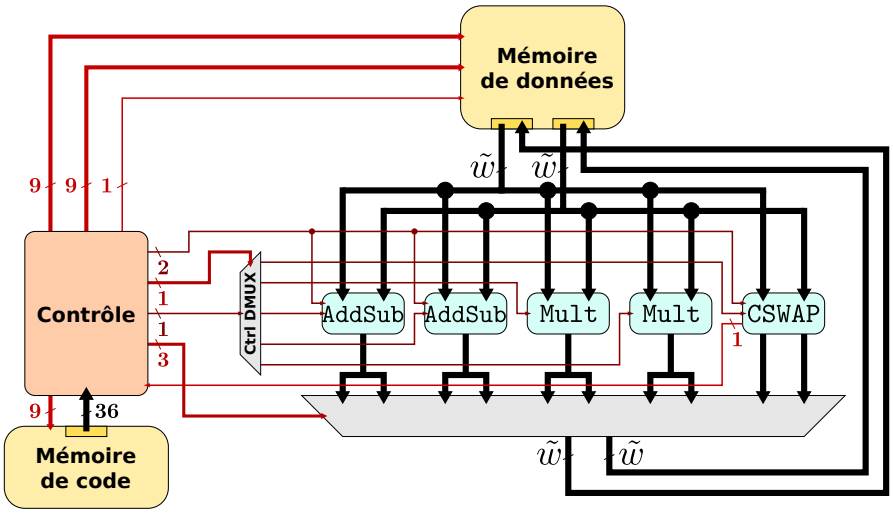


FIGURE 4.4 – Architecture A3 avec ses principales unités (arithmétiques et mémoire), le réseau d'interconnexion interne, et le contrôle (les surfaces des différents blocs ne sont pas à l'échelle).

tation de la densité des accès à la mémoire et des transferts dans le réseau d'interconnexion dans le temps. L'utilisation d'un grand $\tilde{w}$ permet des accès plus rapides à la mémoire et des temps de transfert réduits. Dans les versions basées sur des petits $\tilde{w}$, la mémoire et le réseau d'interconnexion deviennent des goulots d'étranglement de l'architecture, limitant de ce fait le parallélisme entre opérations internes. La valeur de $\tilde{w}$ impacte donc fortement le nombre de cycles dans A3 : le calcul d'une multiplication scalaire pour les versions `w̃68` et `w̃136` nécessite respectivement 34 et 36% de cycles en moins comparé à la version `w̃34`. Le temps de calcul profite lui aussi de cette réduction du nombre de cycles pour des grands $\tilde{w}$: de -25 à -35% du temps de calcul pour la version `w̃136` en fonction du FPGA.

Pour `w̃68` et `w̃136`, A3 est plus rapide que A2 avec des réductions du temps de calcul de 16 à 35% selon le FPGA. Cette augmentation de la vitesse de calcul dans A3 se fait au prix d'une augmentation de la consommation des ressources du FPGA.

Dans le tableau 4.9 on peut noter une augmentation du nombre de cycles dans la version `w̃34` de A3 par rapport à A2. Pour expliquer ce comportement, nous avons dû nous tourner vers la partie logicielle de nos accélérateurs. Nous rappelons que le nombre de cycles nécessaires au calcul d'une multiplication scalaire est directement dépendant du programme exécuté dans l'accélérateur. Ce dernier est généré par l'intermédiaire de notre outil d'ordonnancement pour une configuration d'architecture donnée. Nous avons observé que la politique d'ordonnancement « gloutonne » utilisée, visant à remplir les multiplieurs au plus vite, n'était pas adaptée pour gérer efficacement la forte pression mémoire dans les versions avec des petits $\tilde{w}$. Faute de temps, nous n'avons pas pu nous consacrer à la mise en place d'un outil d'ordonnancement plus performant. On notera toutefois que les résultats d'implantation sur FPGA de nos architectures ne sont pas impactés par la qualité de l'ordonnancement et du code exécuté. L'amélioration de notre ordonnanceur pourrait cependant permettre de réduire le nombre de cycles nécessaires au calcul des multiplications scalaires et donc d'améliorer encore les performances de nos accélérateurs.

En conclusion, l'architecture A3 offre de nouveaux compromis temps de calcul – surface consommée. Grâce à l'exploration effectuée à l'aide de nos outils de modélisation et de simulation CCABA, nous pouvons donc proposer différentes architectures pouvant répondre à différentes contraintes de performances

FPGA	$\tilde{w}$ bits	LUT	FF	*slices* logiques	*slices* DSP	BRAM	fréq. MHz	nbr. de cycles	temps ms
V4	34	1462	2611	1783	22	6	294	188 218	0.64
	68	2802	4367	3468	22	7	282	124 191	0.44
	136	3768	5017	3660	22	9	285	119 057	0.42
V5	34	1262	2607	921	22	6	358	188 218	0.53
	68	2290	4403	1409	22	7	345	124 191	0.36
	136	2737	4978	1594	22	9	348	119 057	0.34
S6	34	1527	2503	668	22	6	265	188 218	0.71
	68	2421	4267	1020	22	7	225	124 191	0.55
	136	3007	4877	1131	22	9	225	119 057	0.53

TABLE 4.9 – Résultats d'implantation de l'architecture A3 sur les différents FPGA (toutes les BRAM font une taille de 18 Kb et nous n'utilisons que les multiplieurs 17×17 dans les *slices* DSP des différents FPGA).

ou de surface. Par exemple, quand la surface de circuit est limitée, la version `w̃34` est intéressante mais le temps de calcul est augmenté de 25 à 30%. En revanche, quand la vitesse de calcul est la contrainte principale, les versions `w̃68` et `w̃136` sont plus adaptées mais consomment plus de ressources.

4.5.4 Architecture A4 : architecture cluster

Dans l'architecture A3, l'ajout d'unités arithmétiques nous a permis d'augmenter le nombre d'opérations indépendantes calculées simultanément et de profiter d'avantage du parallélisme entre opérations arithmétiques dans la formule du `xDBLADD`. Nous avons toutefois constaté que la mémoire et le réseau d'interconnexion devenaient des « goulots d'étranglement » de l'architecture, en particulier dans le cas des versions utilisant des transferts séquentiels de petits mots mémoire (version `w̃34` en particulier). Pour diminuer la congestion mémoire dans nos architectures, nous avons décidé d'ajouter une seconde mémoire de données permettant l'exécution d'opérations de lectures et écritures en parallèle.

L'ajout de cette seconde mémoire augmente cependant la complexité du contrôle et du routage dans nos accélérateurs : p. ex. gestion de deux espaces d'adresses différents, décodage d'instructions concurrentes dans l'unité de contrôle ou encore routage des données vers et depuis les différentes unités via un réseau d'interconnexion plus complexe. Nous avons décidé de ne pas utiliser le modèle d'architecture implanté dans A1-3 mais d'explorer de nouvelles topologies plus adaptées à la gestion du stockage et du transfert d'éléments de GF(P) pour le calcul d'opérations arithmétiques concurrentes.

Nous sommes partis du constat que les différentes opérations dans GF(P) sur lesquelles se base l'opération `xDBLADD` pouvaient être séparées en deux ensembles distincts, séparés par une ligne rouge en pointillés dans la figure 4.5 :

- l'ensemble d'opérations inférieur prend en entrée les 4 coordonnées $(x_1 : y_1 : z_1 : t_1)$ du point $\pm\mathcal{V}_1$ et génère en sortie les coordonnées $(x'_1 : y'_1 : z'_1 : t'_1)$ du nouveau point résultat $\pm\mathcal{V}_1$;
- l'ensemble d'opérations supérieur prend en entrée les 4 coordonnées $(x_2 : y_2 : z_2 : t_2)$ du point $\pm\mathcal{V}_2$ et génère en sortie les coordonnées $(x'_2 : y'_2 : z'_2 : t'_2)$ du nouveau point résultat $\pm\mathcal{V}_2$.

Nous avons alors construit l'architecture A4 autour de deux clusters matériels calculant chacun l'un des ensembles d'opérations nécessaires respectivement au calcul de $\pm\mathcal{V}_1$ ou de $\pm\mathcal{V}_2$ dans la formule de `xDBLADD`.

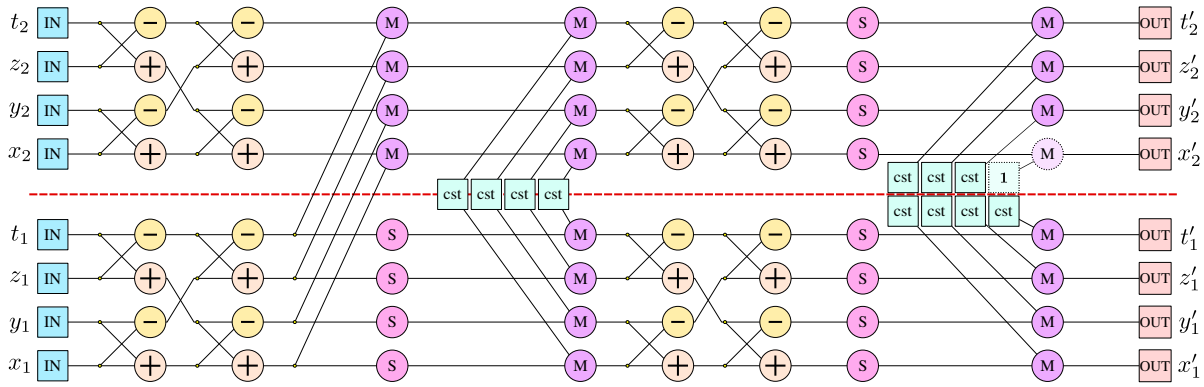


FIGURE 4.5 – Opérations au niveau GF(P) dans `xDBLADD`. La ligne rouge en pointillés indique la séparation des opérations en 2 clusters distincts.

Afin de simplifier le contrôle dans l'architecture A4, nous avons réordonnancé les opérations arithmétiques dans `xDBLADD` et les opérations de `CSWAP` (version `v1`) calculées pour chaque bit k_i du scalaire dans l'algorithme de multiplication scalaire. L'ordonnancement original est illustré dans la figure 4.6 dans laquelle les différentes opérations ont été regroupées pour en améliorer la lisibilité.

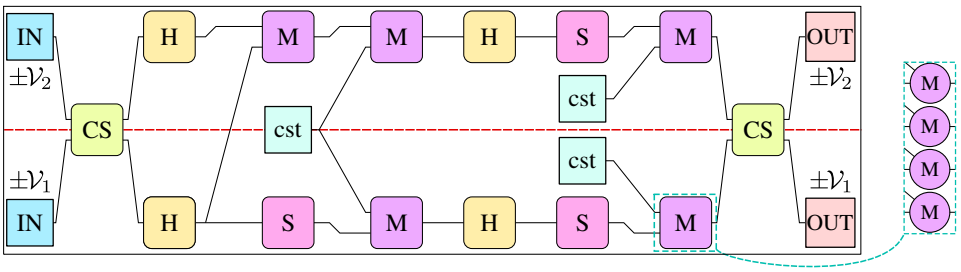


FIGURE 4.6 – Représentation abstraite des opérations modulaires calculées dans une itération de l'échelle de Montgomery pour `xDBLADD` et `CSWAP`. Les carrés `H` représentent les tranformées d'Hadamard (4 additions et 4 soustractions) et les carrés `M`/`S` représentent des groupes de 4 multiplications/carrés. Chaque arc représente 4 éléments de GF(P).

Nous avons tout d'abord éliminé les carrés en remplaçant les opérations $a^2 \times b$ par $a \times b \times a$ (cf. figure 4.7), ce qui nous a permis de calculer des opérations identiques dans les deux clusters.

Pour prendre en charge le transfert d'éléments de GF(P) entre les deux clusters (dépendance surlignée en orange dans la figure 4.7), nous avons modifié le comportement de l'opération de `CSWAP` ainsi que son ordonnancement au sein des itérations de l'échelle de Montgomery (cf. figure 4.8). L'opération de `CSWAP` a ainsi été séparée en deux opérations distinctes $\texttt{CS}_0$ (pour le `CSWAP` en début d'itération) et $\texttt{CS}_1$ (pour le `CSWAP` en fin d'itération). Comme illustré dans la figure 4.9, $\texttt{CS}_0$ et $\texttt{CS}_1$ prennent chacune en entrée 4 coordonnées dans GF(P) et les permutent en fonction de la valeur du bit k_i du scalaire considéré à l'itération courante :

- $\texttt{CS}_0(a, b, c, d)$ renvoie (a, b, c, b) si $k_i = 0$ et (c, d, a, d) sinon ;
- $\texttt{CS}_1(a, b, c, d)$ renvoie (a, b, c, d) si $k_i = 0$ et (c, d, a, b) sinon.

Avec ce nouvel ordonnancement, le nombre d'opérations à calculer et les coordonnées des points $\pm\mathcal{V}_1$ et $\pm\mathcal{V}_2$ à chaque itération de l'échelle de Montgomery sont les mêmes que pour A1. Cependant, les deux ensembles d'opérations à calculer respectivement pour $\pm\mathcal{V}_1$ et pour $\pm\mathcal{V}_2$ lors d'une itération sont

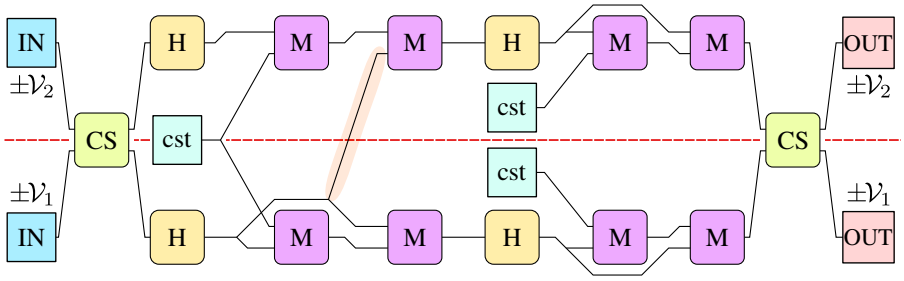


FIGURE 4.7 – Nouvel ordonnancement des opérations pour le calcul des points $\pm\mathcal{V}_1$ et $\pm\mathcal{V}_2$ à chaque itération de l'échelle de Montgomery après suppression des carrés dans `xDBLADD`.

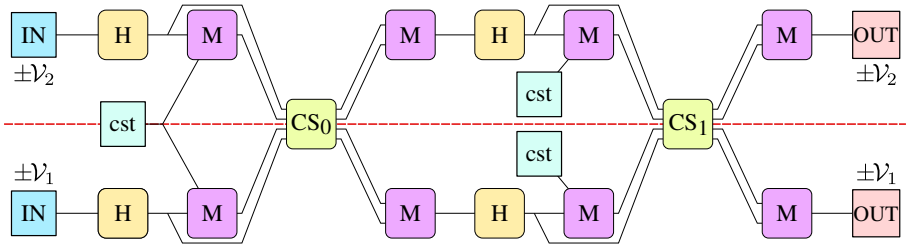


FIGURE 4.8 – Ordonnancement des nouvelles opérations de permutation $\texttt{CS}_0$ et $\texttt{CS}_1$ dans le calcul des points $\pm\mathcal{V}_1$ et $\pm\mathcal{V}_2$ à chaque itération de l'échelle de Montgomery.

identiques. Seuls les opérandes, résultats et constantes stockés en mémoire diffèrent entre les deux clusters.

L'architecture A4 illustrée dans la figure 4.10 exploite cette décomposition de l'opération de `xDBLADD` en deux ensembles identiques d'opérations dans GF(P). Afin d'éviter de surcharger la figure 4.10, nous n'avons pas détaillé le routage des signaux de contrôle, représentés par de petits cercles au niveau des diverses unités. A4 se compose de deux clusters matériels possédant chacun sa propre mémoire de données, son propre ensemble d'unités arithmétiques (1 `AddSub` et 1 `Mult`) et son propre réseau d'interconnexion.

Les deux clusters communiquent par l'intermédiaire d'une nouvelle unité de `CSWAP-v3`, prenant en charge les opérations $\texttt{CS}_0$ et $\texttt{CS}_1$. Dans le nouvel ordonnancement, les résultats de ces opérations de `CSWAP` sont uniquement utilisés comme opérandes des multiplieurs. Nous avons donc décidé de placer l'unité de `CSWAP-v3` à l'entrée des multiplieurs, ce qui nous a permis de supprimer des opérations mémoire inutiles et de simplifier le réseau d'interconnexion dans les deux clusters.

L'unité utilise 2 bits de contrôle pour choisir parmi l'un des 3 modes de fonctionnement possibles : (1) mode $\texttt{CS}_0$; (2) mode $\texttt{CS}_1$; et (3) mode sans permutation (pour les multiplications en sortie des additions et soustractions). On notera ici que les opérations $\texttt{CS}_1$ à itération i et $\texttt{CS}_0$ à l'itération $i+1$ sont séparées par une opération de multiplication. Dans A4, on ne peut donc pas utiliser l'optimisation du `CSWAP-v2` utilisée dans les architectures A2 et A3.

Le contrôle dans cette architecture est identique pour les deux clusters, chacun d'entre eux recevant les mêmes signaux de contrôles (signaux de contrôle des unités, adresses mémoires) à chaque cycle. Le programme correspond à la séquence d'instructions générée par notre outil d'ordonnancement pour les 32 opérations *d'un unique cluster*. Pour permettre l'utilisation des mêmes signaux de contrôle, les différentes constantes utilisées dans chaque cluster sont stockées en mémoire aux mêmes adresses (et dupliquées entre les deux mémoires quand cela est nécessaire).

Comme pour les architectures A1-3, nous avons exploré les trois versions `w̃34`, `w̃68` et `w̃136` des mé-

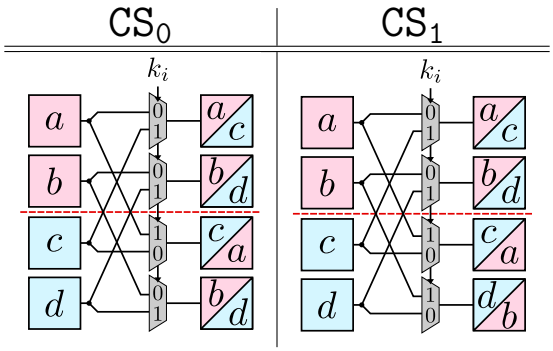


FIGURE 4.9 – Permutation des éléments (a, b, c, d) de GF(P) dans les opérations $\texttt{CS}_0$ et $\texttt{CS}_1$ en fonction de la valeur du bit k_i du scalaire à l'itération i de l'échelle de Montgomery. Les couleurs indiquent les clusters (pour resp. $\pm\mathcal{V}_1$ ou $\pm\mathcal{V}_2$) dont sont issus les différents éléments.

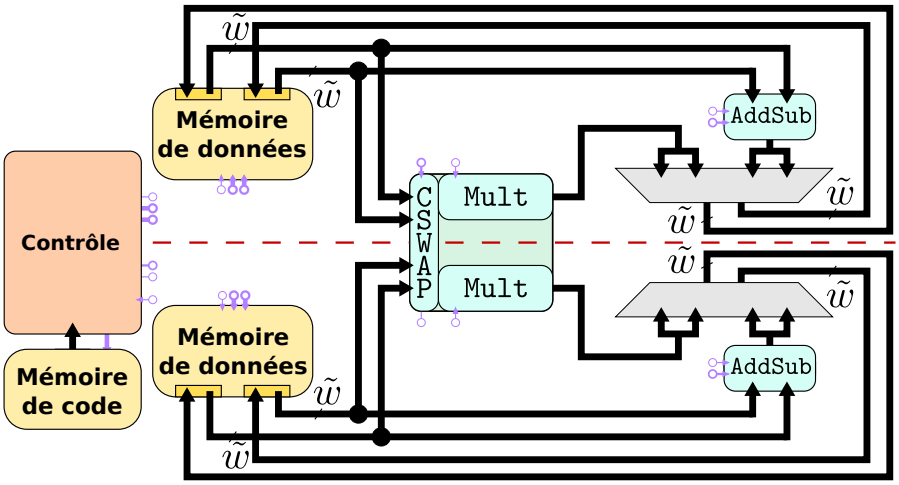


FIGURE 4.10 – Architecture A4 avec ses deux clusters (unités arithmétiques et de mémoire, réseau d'interconnexion interne), le contrôle simplifié (sans le routage des signaux) et la nouvelle modification de l'unité de `CSWAP` (les surfaces des différents blocs ne sont pas à l'échelle).

moires et des réseaux d'interconnexion décrites précédemment. Les résultats d'implantation complets sont rapportés dans le tableau 4.10.

Ces résultats montrent une réduction de 25% du nombre de cycles d'horloge dans la version `w̃34` de A4 par rapport à la version équivalente de A3. Quand $\tilde{w}$ augmente, on peut au contraire constater une légère augmentation du nombre de cycles (+3% pour `w̃68` et +5% pour `w̃136`) due aux contraintes induites dans notre outil d'ordonnancement par la séparation des opérations en clusters distincts : p. ex. parallélisme limité dans chaque cluster ou encore échanges entre clusters contraints par l'utilisation du `CSWAP-v3`.

L'utilisation de clusters matériels permet de simplifier le contrôle, de limiter la propagation des signaux dans l'architecture et de réduire le fanout entre les mémoires et les unités. En conséquence, on peut constater que la fréquence est supérieure dans A4 à celle atteignable dans A3 pour la plupart de nos implantations (elle est au pire similaire dans les autres cas).

Le temps de calcul de la multiplication scalaire est fortement réduit pour les versions utilisant des petites valeurs de $\tilde{w}$. Par exemple, la version `w̃34` permet d'obtenir des temps de calcul similaires à ceux des versions les plus grosses de A3, mais pour des coûts en surface réduits : p. ex. +2% de temps de

FPGA	$\tilde{w}$ bits	LUT	FF	*slices* logiques	*slices* DSP	BRAM	fréq. MHz	nbr. de cycles	temps ms
V4	34	1695	2950	2158	22	7	324	142 119	0.44
	68	2804	4282	3184	22	9	290	128 021	0.44
	136	3171	4994	3337	22	13	299	125 456	0.42
V5	34	1370	2953	1013	22	7	358	142 119	0.40
	68	2095	4259	1358	22	9	337	128 021	0.38
	136	2514	4952	1589	22	13	313	125 456	0.40
S6	34	1564	2089	758	22	7	262	142 119	0.54
	68	2387	4030	1060	22	9	239	128 021	0.54
	136	3181	4786	1136	22	13	251	125 456	0.50

TABLE 4.10 – Résultats d'implantation de l'architecture A4 sur les différents FPGA (toutes les BRAM font une taille de 18 Kb et nous n'utilisons que les multiplieurs 17×17 dans les *slices* DSP des différents FPGA).

calcul pour -41% de *slices* logiques, -48% de LUT et 2 BRAM en moins entre la version `w̃34` de A4 et la version `w̃136` de A3 sur S6. On remarquera que la version de A4 la plus rapide sur V5 est la version `w̃68` dont la fréquence plus élevée permet de compenser un nombre de cycles plus important que pour la version `w̃136`.

Le nombre de *slices* DSP dans A4 est le même que dans A3 car on utilise le même nombre de multiplieurs (c.-à-d. 2 unités `Mult`). En raison de l'utilisation de deux mémoires de données, le nombre de BRAM utilisées dans A4 est plus grand que dans A3. Le nombre de BRAM est proportionnel à $\tilde{w}$ et augmente conformément à la description des configurations mémoire présentées dans le tableau 4.3. On a ainsi 1 BRAM supplémentaire dans l'architecture (A4, `w̃34`) comparé à l'architecture (A3, `w̃34`), 2 BRAM supplémentaires dans (A4, `w̃68`) comparé à (A3, `w̃68`) et 4 BRAM supplémentaires dans (A4, `w̃136`) comparé à (A3, `w̃136`).

Les résultats d'implantation en nombre de LUT sont différents de ceux obtenus pour les autres architectures : la consommation de LUT est plus importante dans la version `w̃34` (jusqu'à $+15\%$) mais réduite dans la version `w̃136` (jusqu'à -16%) comparée à A3.

Pour conclure la discussion sur les résultats de l'architecture A4, on peut faire remarquer que la sélection des paramètres pour nos architectures (topologie, taille des mots mémoire, nombre d'unités arithmétiques, etc.) dépend de l'objectif recherché (vitesse de calcul ou surface de circuit) et du FPGA. Quand l'objectif principal est de sélectionner l'accélérateur le plus petit possible, A4 est moins intéressante que A3. Quand l'objectif est de sélectionner l'accélérateur le plus rapide, A4 est intéressante pour les implantations sur le FPGA *low-cost* S6 mais A3 est meilleure pour V4 et V5. Bien que les accélérateurs basés sur A4 soient un peu moins rapides que ceux basés sur A3, ils sont aussi beaucoup plus petits. Les compromis surfaces–temps dans les accélérateurs implantant l'architecture A4 sont donc meilleurs en pratique.

4.6 Comparaisons et discussions

En figure 4.11, nous représentons les compromis surface–temps pour chaque version `w̃34`, `w̃68` et `w̃136` des 4 architectures implantées sur les 3 FPGA V4, V5 et S6. Les nombres de *slices* DSP et de BRAM

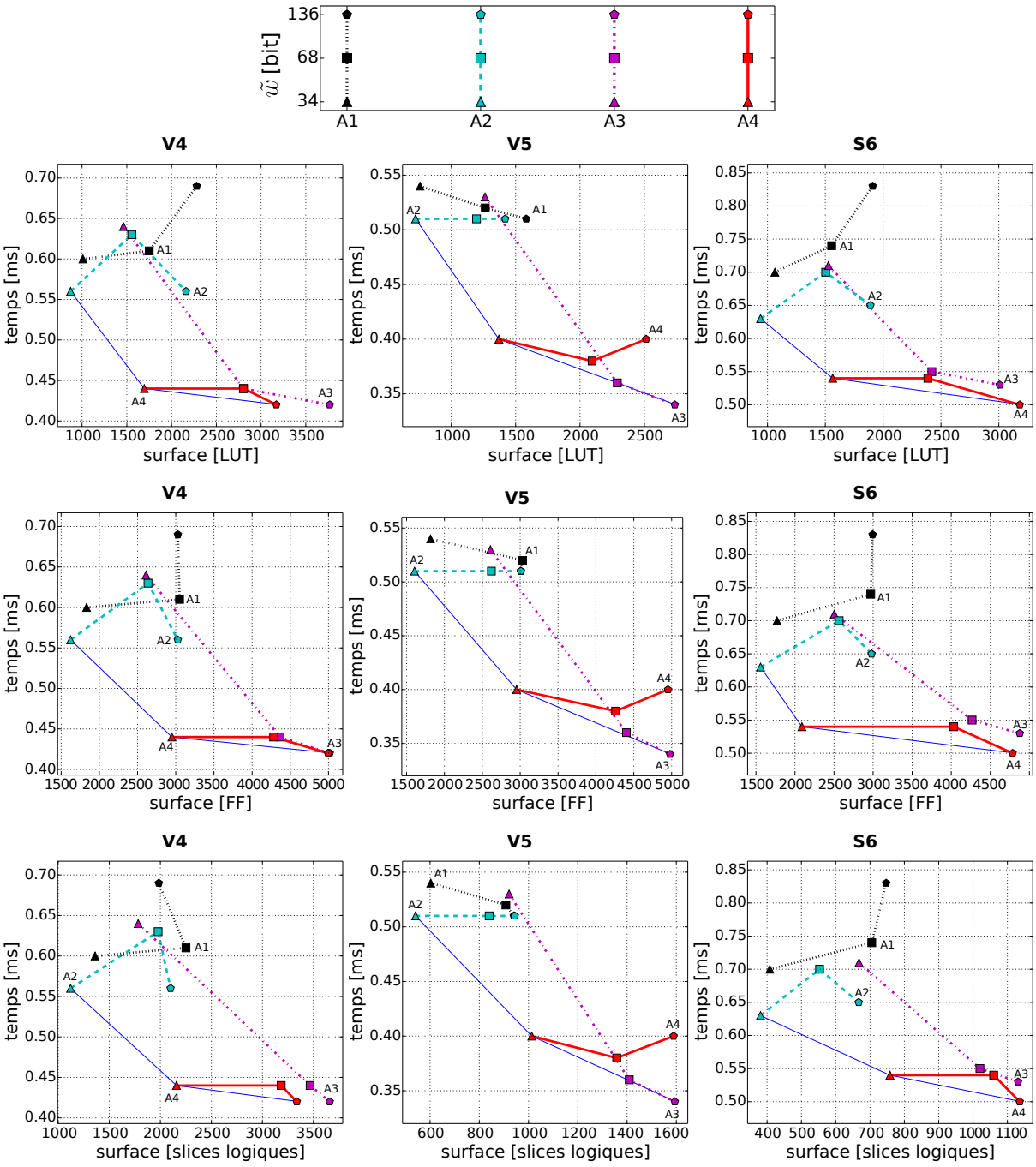


FIGURE 4.11 – Compromis pour nos architectures A1-4 en matière de surface (LUT, FF ou *slices* logiques) et de temps de calcul pour les différentes configurations de $\tilde{w}$ sur nos trois FPGA (les fronts de Pareto sont indiqués en bleu). La légende est définie dans la figure du haut.

consommés étant indépendants du FPGA, nous ne présentons que les compromis surface–temps pour les *slices* logiques, les LUT et les FF. Les meilleures configurations pour chaque FPGA et pour chacune des métriques de surface étudiées sont situées sur la frontière de Pareto, indiquée par une ligne bleue dans chacun des différents graphiques de la figure 4.11.

En étudiant ces différents graphiques, on peut constater que le comportement des frontières de Pareto est fortement corrélé au choix du FPGA. Par exemple, la comparaison des compromis pour les architectures A3 et A4 sur les différents FPGA révèle les comportements suivants :

- sur V4, les versions `w̃136` (resp. `w̃68`) de A4 et A3 ont des vitesses similaires, mais les versions `w̃68` et `w̃136` de A4 sont légèrement plus petites que les versions respectives de A3 : -15.8% de LUT et -8.8% de *slices* logiques pour la version (A4, `w̃136`) comparé à (A3, `w̃136`) ;
- sur V5, les versions `w̃68` et `w̃136` sont plus rapides pour l'architecture A3 que pour A4 (resp. -5% et -10% de temps de calcul) pour un nombre similaire de FF et de *slices* logiques consommés ;
- sur S6, les versions `w̃68` et `w̃136` sont légèrement plus rapides pour A4 que pour A3 pour des surfaces presque similaires (variations inférieures à 6% quelle que soit la métrique de surface considérée).

Les architectures les plus intéressantes sur V4 et S6 sont alors la version `w̃34` de A2 si l'objectif est la conception de petits accélérateurs, la version `w̃136` de A4 si l'objectif est la conception d'accélérateurs rapides, et finalement la version `w̃34` de A4 pour la conception d'accélérateurs présentant un bon compromis surface–vitesse.

De façon similaire, les architectures intéressantes sur V5 sont les versions `w̃34` de A2 pour des petits accélérateurs et `w̃34` de A4 pour des accélérateurs avec un très bon compromis surface–vitesse. Sur ce FPGA particulier, on préfèrera cependant la version `w̃136` de A3 pour l'implantation d'accélérateurs visant les meilleurs temps de calcul.

Le comportement des différentes implantations varient non seulement en fonction du choix des paramètres mais aussi du FPGA sélectionné. Il est alors difficile de prédire les performances d'une architecture sur un FPGA donné en se basant sur les résultats d'implantation de cette même architecture sur un FPGA d'une famille différente.

L'évaluation des performances d'un coprocesseur après les différentes phases d'implantation (synthèse, placement–routage, ...) doit donc être effectuée pour chaque FPGA, ce qui en pratique est un exercice coûteux en temps pour les développeurs et concepteurs d'accélérateurs matériels. Les outils d'exploration au niveau architectural que nous avons proposé permettent d'orienter le choix des paramètres afin de sélectionner les meilleures architectures possibles. Par exemple, ils nous ont aidé à sélectionner, tester et valider les quatre architectures présentées parmi un ensemble d'autres solutions basées entre autres sur des configurations différentes du nombre d'unités arithmétiques. Suite à cette sélection, nous avons pu implanter sur FPGA et valider par simulation les versions d'architectures les plus intéressantes et évaluer celles présentant les meilleurs compromis temps–surface. Le tableau 4.11 rappelle les résultats d'implantations pour nos accélérateurs présentant les meilleurs compromis temps–surface sur chacun des différents FPGA (c.-à-d. les solutions sur les fronts de Pareto).

La comparaison directe de nos accélérateurs avec l'état de l'art des implantations HECC sur $\mathrm{GF}(2^m)$ que nous avons présenté dans la section 4.2 n'est pas possible en raison des différences au niveau arithmétique (opérations dans $\mathrm{GF}(P)$ *versus* opérations dans $\mathrm{GF}(2^m)$). On notera aussi que les niveaux de sécurité théorique proposés par la majorité des implantations HECC de la littérature sont plus petits que ceux proposés dans nos implantations (environ 80 bits de sécurité dans les coprocesseurs de l'état de l'art *versus* 128 bits de sécurité dans nos accélérateurs).

FPGA	archi.	$\tilde{w}$ bits	LUT	FF	*slices* logiques	*slices* DSP	BRAM	fréq. MHz	temps $[k]\mathcal{P}$ ms
Virtex-4	A2	34	872	1624	1121	11	4	330	0.56
	A4	34	1695	2950	2158	22	7	324	0.44
	A4	136	3171	4994	3337	22	13	299	0.42
Virtex-5	A2	34	722	1605	541	11	4	360	0.51
	A4	34	1370	2953	1013	22	7	358	0.40
	A3	136	2737	4978	1594	22	9	348	0.34
Spartan-6	A2	34	940	1559	381	11	4	293	0.63
	A4	34	1564	2089	758	22	7	262	0.54
	A4	136	3181	4786	1136	22	13	251	0.50

TABLE 4.11 – Résultats d'implantations pour nos accélérateurs KHECC sur GF(P) présentant les meilleurs compromis temps–surface sur chaque FPGA.

Dans le tableau 4.12, nous rapportons les meilleurs résultats d'implantation FPGA d'accélérateurs ECC et HECC sur GF(P) que nous avons pu trouver dans la littérature pour un niveau de sécurité théorique de 128 bits. Nous n'avons pas intégré à cet état de l'art les accélérateurs utilisant des représentations particulières des nombres comme le RNS (*residue number system*).

Les implantations de ECC présentées en 2008 dans [GP08] (Gun08a et Gun08b) utilisent la courbe elliptique P-256 standardisée par le NIST et définie sur le corps fini premier de grande caractéristique NIST-256 $= 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ (NIST-256 est un *pseudo-Mersenne* permettant le calcul rapide de la réduction modulaire). Les coprocesseurs de [GP08] étant spécifiquement conçus pour utiliser le premier NIST-256, ils ne sont pas flexibles en terme du choix de courbe ou de corps fini GF(P).

Le coprocesseur proposé en 2014 dans [AR14] (Alr14) est aussi limité à l'utilisation des courbes du NIST mais permet en revanche de supporter 5 courbes différentes, pour différentes tailles de premiers (192 à 521). On notera que le coprocesseur Alr14 de [AR14] intègre aussi un ensemble de protections contre les attaques par SCA : utilisation de l'échelle de Montgomery, *randomisation* du scalaire k, conception d'unités ayant un comportement uniforme, *randomisation* des adresses mémoire ou encore ajout de bruit par exemple. Ces protections ont un impact non négligeable sur la consommation de *slices* logiques mais pas sur le nombre de *slices* DSP et de BRAM utilisés (très élevé dans cette implantation). Le temps de calcul rapporté pour Alr14 dans le tableau 4.12 est donnée pour une configuration de l'accélérateur utilisant la courbe P-256 et protégé « uniquement » contre les attaques par SPA et par injection fautes.

Parmi ces différentes solutions de l'état de l'art optimisées pour l'utilisation de la courbe P-256 du NIST, seule Gun08a est réalisable en pratique dans le cadre d'applications embarquées où les ressources sont limitées (Gun08b et Alr14 consomment un nombre très important de *slices* DSP et de BRAM). En comparaison avec l'implantation de Gun08a sur un FPGA Virtex-4, notre accélérateur KHECC basé sur la version $\tilde{\mathtt{w}}\mathtt{34}$ de A4 est plus rapide (-12% de temps de calcul) et consomme moins de *slices* DSP et de BRAM (respectivement -31 et -36%) pour un nombre de *slices* logiques augmenté de 26%.

Les implantations de ECC présentées en 2012 dans [LWH12] (Lai12) et en 2013 dans [MLPJ13] (Ma13) sont conçues pour des courbes non spécifiques, définies sur des corps finis de grandes caractéristiques premières génériques GEN-256 (GEN-256 désigne ici un nombre premier de 256 bits n'ayant pas de structure binaire particulière). Pour une taille de premiers fixée, ces implantations permettent la prise en

réf.	P	FPGA	LUT	FF	*slices* logiq.	*slices* DSP	BRAM	fréq. MHz	tps $[k]\mathcal{P}$ ms	protections SPA	protections DPA	crypto-système
Gun08a	NIST-256	Virtex-4	2589	2028	1715	32	11	490	0.50			ECC
Gun08b			34896	32430	24574	512	176	375	0.04			
Lai12	GEN-256	Virtex-4	n.r.	n.r.	2901	14	n.r.	227	1.09			
		Virtex-5	n.r.	n.r.	3657	10	n.r.	263	0.86			
Ma13	GEN-256	Virtex-4	5740	4876	4655	37	11	250	0.44	$\surd$	$\surd$	
		Virtex-5	4177	4792	1725	37	10	291	0.38	$\surd$	$\surd$	
Alr14	NIST-256	Virtex-6	32900	n.r.	11200	289	128	100	0.40	$\surd$	($\surd$)	
Sas15u	$2^{255}-19$	Zynq-7020	2783	3592	1029	20	2	200	0.40	$\surd$		
Sas15p			2862	3784	1180	22	2		0.42	$\surd$	$\surd$	
Jar16a	extension $(2^{127}-1)^2$	Zynq-7020	1069	1894	565	16	7	190	0.31	$\surd$		
Jar16b			4217	4413	1691	27	10		0.16	$\surd$		
Kop18a	$2^{127}-1$	Zynq-7020	8764	6852	2657	49	0	139	0.08	$\surd$		HECC

TABLE 4.12 – État de l'art des implantations sur FPGA de cryptosystèmes à base de courbes sur GF(P) pour un niveau de sécurité théorique de 128 bits. À l'exception du coprocesseur KHECC Kop18a, les différents coprocesseurs listés implantent des solutions pour ECC.

charge de différents corps finis et permettent ainsi une certaine flexibilité au niveau du choix des courbes.

Le coprocesseur Ma13 de [MLPJ13] intègre une unité de multiplication modulaire basée sur la variante de l'algorithme de Montgomery présentée par Orup dans [Oru95] (cf. chapitre 3 pour plus de détails sur le multiplieur de [MLPJ13]). Dans leurs travaux, Ma et coll. implantent un additionneur/soustracteur *sans réduction* ce qui leur permet de réduire le temps de calcul des opérations d'addition et de soustraction au prix d'une augmentation des ressources consommées. La réduction des résultats d'addition et de soustraction est effectuée dans le multiplieur grâce à une nouvelle extension du domaine de Montgomery (c.-à-d. $2^m > 64\widetilde{P}$ dans l'algo. 18). Au niveau courbe, la multiplication dans Ma13 scalaire est calculée en utilisant un recodage w-NAF du scalaire ainsi que la *randomisation* des coordonnées du point de base $\mathcal{P}$. La randomisation permet d'améliorer la résistance du coprocesseur aux attaques par SCA sans toutefois impacter de façon importante les performances de calcul et la consommation de ressources au dire des auteurs. Le coprocesseur Ma13 est à notre connaissance l'implantation la plus performante d'un accélérateur ECC sur GF(P) avec P générique pour un niveau de sécurité théorique de 128 bits.

En comparaison avec les résultats d'implantation pour ce coprocesseur, notre version `w̃34` de A4 permet de réduire de 40% le nombre de *slices* DSP et de BRAM, et de 53% le nombre de *slices* logiques pour un temps de calcul identique (0.44 ms) sur un FPGA Virtex-4 (ces résultats sont similaires sur V5). Grâce à l'utilisation de l'hyperthreading dans nos multiplieurs ainsi qu'à la décomposition des éléments de GF(P) en mémoire et dans le(s) réseau(x) d'interconnexion, nos accélérateurs sont capables d'atteindre des fréquences plus importantes et des temps de calcul réduits tout en limitant la consommation des ressources du FPGA.

Les coprocesseurs Sas15u et Sas15p proposés en 2015 dans [SG15] utilisent la courbe elliptique « Curve25519 » introduite dans [Ber06] en 2006 et définie sur GF(P), P correspondant au pseudo-Mersenne spécifique $P = 2^{255} - 19$. Les deux accélérateurs Sas15u et Sas15p sont protégés contre les attaques par SPA mais Sas15p intègre en plus certaines protections contre les attaques par DPA. Ils sont implantés sur une plateforme Zynq-7020 embarquant un FPGA Artix-7 de Xilinx. Ce FPGA est construit selon une structure similaire à celle du Virtex-5 (p. ex. même configuration des éléments logiques et des *slices* DSP) mais les fréquences maximales atteignables sont supérieures : 628 MHz dans les *slices* DSP de l'Artix-7 contre 550 MHz dans celle du Virtex-5 par exemple. Dans [SG15], les auteurs proposent aussi des versions multi-cœurs de leur architecture visant à augmenter le débit de l'accélérateur en tirant parti des ressources disponibles sur le FPGA. En raison du coût d'implantation élevé de ces coprocesseurs (220 *slices* DSP par exemple), nous ne nous y comparerons pas.

Enfin, on rappellera qu'en raison de la structure spécifique du premier utilisé dans la courbe Curve25519, l'étape de réduction modulaire dans les opérations arithmétiques est optimisée. Les unités optimisées implantées dans Sas15 permettent de réduire le temps de calcul et/ou la consommation de ressources dans le coprocesseur mais limitent son utilisation à la courbe Curve25519. Ce n'est pas le cas de nos accélérateurs qui peuvent prendre en charge des courbes quelconques pour une taille donnée d'éléments de corps finis (128 bits en l'occurrence).

En termes de performances sur Virtex-5, nos accélérateurs présentent des compromis intéressants par rapport aux implantations de Sas15u et Sas15p sur Zynq-7020. Ainsi, la version `w̃34` de la petite architecture A2 requière seulement 16% de temps de calcul en plus et 2 BRAM supplémentaires par rapport à Sas15u mais un nombre de LUT, FF et *slices* logiques divisé respectivement par 3.9, par 2.2 et par 1.94. Le nombre de *slices* DSP est quant à lui quasiment divisé par 2.

Enfin, les implantations Jar16a et Jar16b proposées en 2016 dans [JMAL16] sont conçue pour l'utilisa-

tion exclusive de la courbe elliptique « Four$\mathbb{Q}$ » définie en 2015 dans [CL15]. Four$\mathbb{Q}$ se base sur le corps fini GF $\left((2^{127}-1)^2\right)$ permettant des réductions rapides modulo le premier de Mersenne $P = 2^{127}-1$. Il est essentiel de remarquer que les opérations dans GF $\left((2^{127}-1)^2\right)$ peuvent être calculées par des *séquences d'opérations arithmétiques sur des opérandes de 127 bits*, ce qui n'est pas le cas dans les implantations ECC présentées jusqu'à présent. Grâce à la réduction des tailles des opérandes et à la réduction modulo le premier de Mersenne $2^{127}-1$, les unités arithmétiques implantées dans Jar16a et Jar16b nécessitent peu de temps de calcul (20 cycles pour une multiplication modulaire par exemple) sans augmenter la consommation des ressources du FPGA (16 *slices* DSP par multiplieur par exemple). L'utilisation de Four$\mathbb{Q}$ augmente cependant en contrepartie la complexité des opérations au niveau courbe.

Les deux versions Jar16a et Jar16b proposées diffèrent par l'algorithme utilisé pour la multiplication scalaire : échelle de Montgomery pour la première et utilisation d'endomorphismes pour la seconde. L'utilisation de l'échelle de Montgomery dans la version Jar16a permet de réduire le coût matériel du coprocesseur implanté comparé à la version Jar16b au détriment des performances de calcul. Jar16a est alors adaptée pour les applications limitées en ressources et Jar16b pour les applications dans lesquelles la réduction du temps de calcul prévaut sur la consommation de surface. Les performances de ces deux architectures après implantation sur Zynq-7020 représentent à notre connaissance les meilleures performances que l'on peut trouver dans l'état de l'art des implantations ECC sur FPGA pour un corps fini *non générique*. Comme pour les coprocesseurs Sas15u et Sa15p de [SG15], nous avons comparé les performances de Jar16a et Jar16b, implantés sur Zynq-7020, avec celles de nos accélérateurs implantés sur Virtex-5.

La comparaison des coprocesseurs optimisés Jar16a et Jar16b avec nos accélérateurs (A2, $\tilde{\mathtt{w}}\mathtt{34}$), (A4, $\tilde{\mathtt{w}}\mathtt{34}$) et (A3, $\tilde{\mathtt{w}}\mathtt{136}$) montre clairement l'impact de la généricité de nos architectures quant au choix de corps fini sur le temps de calcul de la multiplication scalaire et la consommation de ressources. Pour une consommation de ressources équivalente en nombre de *slices* logiques et un nombre de *slices* DSP et de BRAM réduit de 31% et de 43% respectivement, le temps de calcul dans notre l'accélérateur (A2, $\tilde{\mathtt{w}}\mathtt{34}$) est augmenté de 65%. (A4, $\tilde{\mathtt{w}}\mathtt{34}$) permet de réduire l'écart de temps de calcul à +29% mais au prix d'une augmentation importante du nombre de *slices* logiques (+79% par rapport à Jar16a) et de l'ajout de *slices* DSP supplémentaires. Le coprocesseur Jar16b consomme quand à lui légèrement plus de *slices* DSP et de BRAM que notre accélérateur le plus gros (A3, $\tilde{\mathtt{w}}\mathtt{136}$) mais pour un temps de calcul divisé par 2.1 pour la multiplication scalaire. Toutefois, on notera que la fréquence dans nos accélérateurs est supérieure à celle atteinte par Jar16a et Jar16b. Cela peut être expliqué par la mise en place d'un pipeline efficace non seulement au sein des diverses unités mais aussi au niveau du contrôle, de la mémoire et du réseau d'interconnexion de nos architectures.

Finalement, la comparaison avec le coprocesseur Kop18a pour KHECC montre que nos accélérateurs implantés sur Virtex-5 consomment moins de ressources et profitent d'une fréquence plus élevée. Les performances en temps de calcul de nos diverses architectures sont toutefois là encore fortement impactées par la volonté de mettre en place des accélérateurs génériques. Ainsi, on rappellera entre autres que le multiplieur de Kop18a optimisé pour $P = 2^{127}-1$ a une latence de 7 cycles (contre 69 cycles pour une multiplication modulaire de Montgomery dans notre HTMM) mais qu'il utilise 49 *slices* DSP (contre 11 dans notre HTMM). Malgré l'impact de la généricité sur les performances de nos accélérateurs, nos implantations permettent d'obtenir des compromis intéressants par rapport à Kop18a. Par exemple, notre coprocesseur (A2, $\tilde{\mathtt{w}}\mathtt{34}$) nécessite un temps de calcul 6.38 fois supérieur à celui de Kop18a mais consomme 4.45 fois moins de *slices* logiques, 4.91 fois moins de *slices* DSP et 4.27 fois moins de FF, tout

en pouvant utiliser différentes courbes et corps premiers. Le nombre de LUT utilisées dans (A2, $\tilde{w}34$) est lui divisé par 12.14 par rapport à Kop18a mais 4 BRAM supplémentaires sont utilisées (Kop18a utilise la RAM distribuée implantée dans les *slices* logiques du FPGA). Enfin, on fera remarquer que l'utilisation d'un chemin de données de 127 bits dans Kop18a permet de réduire les temps d'accès mémoire et de transfert des opérandes mais impacte aussi fortement la fréquence de fonctionnement du coprocesseur. Celle-ci est ainsi limitée à 139 MHz, ce qui représente 22% de la fréquence maximale atteignable dans les *slices* DSP de la Zynq-7020 et 39% de la fréquence atteinte dans (A2, $\tilde{w}34$).

En raison du la taille du coprocesseur Kop18b (196 *slices* DSP, 10554 *slices* logiques) dont le but n'est pas l'optimisation de la multiplication scalaire mais l'augmentation du débit de l'accélérateur, nous ne nous intéresserons pas à celui-ci pour nos comparaisons.

4.7 Nouveaux accélérateurs utilisant la version F44B de HTMM

Afin de compléter les travaux présentés dans ce chapitre, nous avons tenu à proposer quelques nouvelles architectures utilisant les dernières versions du HTMM optimisé que nous avons proposées dans [GT18a] (cf. chapitre 3). Nous avons choisi d'utiliser la version F44B de HTMM en raison de ses bonnes performances et de l'implanter dans les versions (A2, $\tilde{w}34$), (A4, $\tilde{w}34$) et (A3, $\tilde{w}136$) de nos cryptoprocesseurs.

Pour rappel, la version F44B de HTMM est basée sur $\sigma = 4$ multiplieurs logiques (LM) permettant de calculer jusqu'à 4 multiplications modulaires indépendantes en parallèle avec un intervalle $\tau = 4$ cycles entre les différents produits. Une multiplication est calculée dans un LM avec une latence λ de 75 cycles.

Un nouvel ordonnancement des opérations arithmétiques a été généré pour chacune des versions afin de prendre en compte le parallélisme de 4 opérations dans F44B (au lieu de 3 dans le HTMM utilisé précédemment). Les temps de calcul des nouveaux ordonnancement pour les différentes versions sont détaillés dans le tableau 4.13. L'exécution de chaque nouvel ordonnancement (c.-à-d. programme) dans l'architecture correspondante a été validée après modélisation CCABA dans nos outils puis cette dernière a été implantée sur chaque FPGA pour validation par simulation et évaluation des performances après placement–routage.

Les résultats d'implantation obtenus pour ces différents coprocesseurs sur les FPGA Virtex-4 (V4), Virtex-5 (V5) et Spartan-6 (S6) sont présentés dans le tableau 4.14.

accélérateur		nbr. de cycles	accélération
archi.	version		
A2	$\tilde{w}34$	175 155	×1.05
A4	$\tilde{w}34$	128 035	×1.11
A3	$\tilde{w}136$	107 539	×1.11

TABLE 4.13 – Temps de calcul d'une multiplication scalaire en nombre de cycles dans nos accélérateurs KHECC utilisant la version F44B de HTMM. L'accélération correspond au ratio des temps (en nombre de cycles) par rapport aux implantations du tableau 4.11.

D'après les résultats présentés dans le tableau 4.13, on peut voir que l'ajout d'un LM dans nos multiplieurs permet une diminution du temps de calcul de 5 à 12% pour la multiplication scalaire comparé aux versions listées dans le tableau 4.11. L'impact de cette modification est logiquement plus important dans les accélérateurs (A4, $\tilde{w}34$) et (A3, $\tilde{w}136$) que dans l'accélérateur (A2, $\tilde{w}34$) pour lequel la congestion mémoire empêche de profiter efficacement du parallélisme potentiel entre les opérations arithmétiques.

FPGA	archi.	$\tilde{w}$ bits	LUT	FF	*slices* logiq.	*slices* DSP	BRAM	fréq. MHz	tps $[k]\mathcal{P}$ ms
Virtex-4	A2	34	863	1689	1081	9	4	327	0.54
	A4	34	1699	3255	2447	18	7	328	0.39
	A3	136	3959	5251	3492	18	9	290	0.37
Virtex-5	A2	34	783	1653	558	9	4	386	0.45
	A4	34	1413	3182	1019	18	7	378	0.34
	A3	136	2658	5170	1657	18	9	356	0.30
Spartan-6	A2	34	911	1619	382	9	4	298	0.59
	A4	34	1565	3120	809	18	7	276	0.46
	A3	136	3128	5040	1182	18	9	238	0.45
Zynq-7020	A2	34	855	1619	463	9	4	347	0.50
	A4	34	1475	3020	747	18	7	360	0.36
	A3	136	3147	5033	1143	18	9	322	0.33

TABLE 4.14 – Résultats d'implantations d'accélérateurs KHECC sur GF(P) utilisant la version optimisée F44B du HTMM 128 bits avec $\sigma = 4$ LM et une latence de 75 cycles par MMM.

Pour ce qui est de la surface, le nombre de *slices* DSP est réduit de 4 grâce aux dernières optimisations de HTMM présentées dans le chapitre 3, pour un nombre de *slices* logiques légèrement augmenté. Le nombre de BRAM n'est pas impacté par la modification du multiplieur (F44B utilisant aussi 2 BRAM).

Enfin, on peut constater que l'utilisation du multiplieur hyper-threadé F44B ne permet pas d'augmenter les fréquences de nos accélérateurs (moins de 10% d'augmentation au maximum sur Virtex-5). Après une analyse plus fine de nos nouvelles implantations, nous avons pu constater que le chemin critique dans nos architectures ne se situait plus dans le multiplieur mais au niveau du contrôle et du réseau d'interconnexion sur les FPGA V5 et S6. Sur V4, le chemin critique est lui situé au niveau de la propagation des retenues sur $w = 34$ bits dans l'additionneur.

L'utilisation de la version optimisée F44B de HTMM dans nos accélérateurs permet de rendre ceux-ci compétitifs avec les meilleures implantations sur corps finis spécifiques de l'état de l'art en matière de compromis temps–surface. Par exemple, le temps de calcul d'une multiplication scalaire dans notre petit accélérateur (A2, $\tilde{\mathtt{w}}\mathtt{34}$) implanté sur V5 est maintenant 5.6 fois plus long que celui de Kop18a pour un nombre de *slices* DSP divisé par 5.4 et un nombre de *slices* logiques, de LUT et de FF divisé par 4.8, 11.2 et 4.2 respectivement et ce sans aucune optimisation dépendant du choix du premier dans nos opérateurs arithmétiques. Sur Zynq-7020, le temps de calcul dans accélérateur (A2, $\tilde{\mathtt{w}}\mathtt{34}$) est, lui, 6.3 fois plus long que celui de Kop18a mais le nombre de *slices* logiques est divisé par 7 et les nombres de LUT et de FF sontt divisés respectivement par 10.3 et 4.3.

Grâce à l'utilisation du HTMM de [GT18a], le premier P peut maintenant être modifié à l'exécution dans les nouveaux accélérateurs listés dans le tableau 4.14.

4.8 Conclusions et perspectives

Dans ce chapitre, nous avons présenté les résultats d'implantation de plusieurs accélérateurs matériels pour KHECC publiés dans [GCT17]. Ces accélérateurs se basent sur la solution « μKummer » de [RSSB16] mais peuvent être utilisés pour des courbes hyperelliptiques quelconques basées sur des

corps finis de grande caractéristique première et générique. Il s'agit à notre connaissance des premières implantations FPGA de cryptoprocesseurs pour le calcul de la multiplication scalaire dans KHECC utilisant ce type de corps proposées dans la littérature.

Grâce aux outils logiciels d'exploration d'architectures que nous avons proposés ainsi qu'à l'utilisation de notre multiplieur hyper-threadé de [GT17d], nous sommes en mesure de proposer des accélérateurs performants pour différentes contraintes et applications. Ainsi, nous pouvons proposer des solutions matérielles pour des applications dans lesquelles les ressources sont limitées, ou bien au contraire des solutions parallèles plus grosses dans le cas où de hautes vitesses sont requises. Nos résultats d'implantation montrent que nos accélérateurs atteignent des vitesses de calcul similaires à celles des toutes meilleures implantations de l'état de l'art pour des corps finis GF(P) génériques, mais pour des surfaces réduites de moitié (-40% pour les *slices* DSP et les BRAM et -60% pour les *slices* logiques).

L'utilisation d'un contrôle à base de programmes et de notre HTMM optimisé dans lequel le premier P peut être modifié à l'exécution au sein de nos cryptoprocesseurs rendent ces derniers particulièrement flexibles et adaptables. Grâce à leur flexibilité et du niveau de sécurité proposé, nous les destinons principalement à des utilisations au sein de systèmes embarqués pour lesquels la reconfiguration des FPGA est complexe et dont les durées de vie s'étendent sur plusieurs années.

Il reste cependant plusieurs pistes à explorer pour améliorer nos accélérateurs, parmi lesquelles deux nous semblent particulièrement intéressantes.

La première concerne l'analyse plus poussée de leur robustesse face aux attaques par analyse de canaux auxiliaires. En effet, l'utilisation de l'algorithme de l'échelle de Montgomery et l'implantation de notre unité de `CSWAP` sont les premières étapes dans la mise en place de cryptoprocesseurs *sécurisés* et *robustes* mais ne sont pas suffisantes.

Pour rappel, l'échelle de Montgomery assure le calcul uniforme et à temps constant de $[k]\mathcal{P}$ au niveau algorithmique : les opérations exécutées et le temps de calcul sont indépendants des bits du scalaire k.

Au niveau circuit, les fuites d'information sensible dans l'unité de `CSWAP` sont limitées par la lecture ou l'écriture parallèle des couples de coordonnées en entrée ou en sortie de l'opérateur. L'utilisation d'un pipeline interne permet lui de masquer les variations de courant dans l'unité par l'écriture sur les ports d'entrées et de sortie de paires de coordonnées indépendantes. Enfin, le fait d'avoir décorrélé le contrôle de la valeur du scalaire nous permet de limiter les fuites d'informations au niveau de la gestion de la mémoire (les adresses accédées ne sont pas liées à k) et des unités.

Nous n'avons toutefois pas eu le temps dans le cadre de la thèse d'approfondir l'étude des fuites d'informations pouvant survenir lors des transferts de données dans nos accélérateurs. De la même manière, il nous faudra à l'avenir étudier l'impact sur les performances de nos cryptoprocesseurs de l'ajout de certaines contremesures de l'état de l'art (recodage du scalaire ou randomisation des adresses mémoires par exemple).

Une deuxième piste d'amélioration que nous avons envisagé sans avoir le temps de la concrétiser concerne l'exploration de topologies et de modèles d'architectures différents. En particulier, nous aurions souhaité implanter une version d'architecture basée sur un modèle *dataflow* sans mémoire centralisée. Ce type de modèle, initialement proposé dans les années 80 pour les processeurs généralistes, a pour avantage la mise en place d'architectures nativement parallèles. Dans le modèle dataflow, les opérandes sont gérées localement au niveau des unités grâce à un ensemble de FIFO (files de registres de type « *first in, first out* ») et grâce à des échanges de messages entre unités. Il est aussi intéressant d'un point de vue sécurité car il pourrait *a priori* permettre une *randomisation* facile des calculs dans l'architecture.

5 Conclusion

La cryptographie sur courbe hyperelliptique (HECC) a été proposée comme alternative à la cryptographie sur courbe elliptique (ECC) pour l'implantation de cryptosystèmes asymétriques. Dans HECC, la taille des valeurs manipulées est divisée par 2 par rapport à celle des valeurs manipulées dans ECC, pour le même niveau de sécurité théorique. Depuis le début des années 2000 et la publication des premières formules optimisées pour HECC [Lan05, WPP05], HECC est généralement considérée comme plus performante que ECC. Les implantations logicielles récentes de μKummer présentées dans [RSSB16] confirment l'intérêt de HECC par rapport à ECC. Par exemple, l'implantation sur le Cortex M0 d'ARM du protocole de signature numérique décrite dans [RSSB16] est 75% plus rapide que la meilleure implantation ECC de l'état de l'art sur le même microcontrôleur. Il n'existe cependant à ce jour que peu d'implantations matérielles de HECC dans la littérature. De plus, la quasi-totalité de ces implantations utilisent des courbes binaires avec de faibles niveaux de sécurité théorique au regard des recommandations de sécurité actuelles. Dans ce contexte, le but du projet labex HAH (*Hardware and Arithmetic for HECC*) était d'étudier et de proposer de nouvelles unités arithmétiques et architectures de cryptoprocesseurs pour HECC sur FPGA.

Durant la thèse, nous nous sommes basés sur les solutions μKummer de [RSSB16] pour concevoir et proposer différents accélérateurs matériels *performants* et *flexibles* pour le calcul de la multiplication scalaire. Grâce à différents outils de modélisation et de simulation que nous avons développé, nous avons pu explorer, évaluer et valider différentes spécifications de paramètres d'architectures pour nos accélérateurs : topologie, configuration des mémoires et des communications, nombre et type des unités arithmétiques, etc. Nous avons validé et évalué les performances des accélérateurs les plus intéressants après implantation sur différents FPGA. Les principaux résultats obtenus ont été publiés dans [GCT17].

Dans nos cryptoprocesseurs, les paramètres de la courbe hyperelliptique sur GF(P) ainsi que le premier P peuvent être modifiés à l'exécution pour s'adapter aux évolutions de HECC (pour une taille maximale des éléments de GF(P) définie à l'implantation). Nos cryptoprocesseurs sont aussi performants mais 2 fois plus petits que les toutes meilleures implantations matérielles de la multiplication scalaire (H)ECC sur FPGA de l'état de l'art, pour des P génériques. Ils intègrent aussi certaines protections contre les attaques physiques par observation de type SPA.

L'opération de multiplication scalaire implantée dans nos cryptoprocesseurs nécessite le calcul d'opérations dans GF(P) et donc des réductions modulo P. En particulier, les multiplications modulaires sont les opérations les plus courantes et coûteuses dans (H)ECC. Leur calcul nécessite l'utilisation d'algorithmes plus complexes tels que l'algorithme de multiplication modulaire de Montgomery (MMM). L'implantation d'unités arithmétiques matérielles pour la MMM est un exercice difficile, en particulier quand des hautes fréquences sont visées. En effet, les dépendances de données internes à l'algorithme empêchent de remplir efficacement le pipeline des unités et diminuent le taux d'utilisation des ressources matérielles dans les circuits implantés sur FPGA (*slices* DSP p. ex.).

Nous avons proposé une nouvelle unité matérielle basée sur la variante CIOS [KAK96] de la MMM et

utilisant le principe de l'*hyper-threading* [KM03] pour réduire le nombre de « bulles » dans le pipeline. Ces « bulles » correspondent à des cycles pendant lesquels certains étages du pipeline ne calculent aucune opération utile. Notre multiplieur modulaire hyper-threadé (HTMM) est un *multiplieur physique* dans lequel plusieurs *multiplieurs logiques* (LM) se partagent les ressources matérielles pour calculer différentes MMM indépendantes en même temps. Dans [GT17d] nous avons proposé des premiers exemples d'implantations du HTMM, décrites à la main en VHDL pour des premiers P de 128 bits définis à l'implantation. Ces premières versions ont été implantées, validées et évaluées sur 3 FPGA différents. Pour le calcul de plusieurs MMM indépendantes, elles nous ont permis d'obtenir de meilleurs compromis temps–surface que les meilleures implantations matérielles de l'état de l'art utilisant des premiers P génériques de 128 bits. Dans [GT18a], nous avons proposé de nombreuses améliorations du HTMM, parmi lesquelles la possibilité de modifier la valeur du premier P à l'exécution (la taille maximale de P est fixée à l'implantation). Nous avons aussi proposé un générateur de HTMM nous permettant de produire rapidement les codes VHDL du HTMM pour différentes spécifications de paramètres internes : nombre de LM, taille de P, optimisations utilisées, FPGA ciblé, etc. Nous avons illustré les performances de notre nouveau HTMM en proposant de nombreuses implantations pour des tailles maximales de P de 128 et de 256 bits sur différents FPGA. Notre nouveau HTMM 128 bits est 2 fois plus rapide et 2 fois plus petit que les versions 128 bits des meilleurs multiplieurs modulaires génériques de l'état de l'art. Le HTMM 256 bits est, quant à lui, 4 à 5 fois plus petit que le meilleur multiplieur modulaire 256 bits de l'état de l'art, pour un temps de calcul multiplié seulement par 1.5.

Pour faciliter la reproduction des résultats présentés dans [GT18a], nous avons tenu à rendre accessible l'ensemble des codes VHDL des versions du HTMM implantées. Ces codes VHDL ainsi que notre générateur de HTMM sont disponibles en accès libre sur le site [GT18b].

Durant cette thèse, nous avons donc proposé de nouveaux cryptoprocesseurs HECC pour le calcul de la multiplication scalaire qui répondent aux contraintes fixées dans le cadre du projet HAH : bonnes performances de calcul et surfaces de circuit restreintes ; flexibilité dans le choix des paramètres de courbe et de corps premier ; et robustesse contre les attaques physiques SPA. Nous avons aussi proposé un nouveau type d'unité arithmétique performante permettant d'améliorer l'utilisation des ressources matérielles lors du calcul simultané de plusieurs MMM indépendantes. Les résultats d'implantations sur FPGA de nos accélérateurs matériels flexibles corroborent ceux des implantations logicielles de [RSSB16] et montrent que HECC est sensiblement plus performante que ECC en matériel.

Faute de temps, nous n'avons pas pu étudier et comparer les performances de nos accélérateurs quant à la consommation d'énergie totale pour le calcul d'une multiplication scalaire. Dans le futur, il serait particulièrement intéressant d'ajouter cette métrique à nos évaluations afin de pouvoir estimer l'impact de nos cryptoprocesseurs sur des systèmes restreints en énergie (systèmes embarqués fonctionnant sur batterie par exemple). Au niveau sécurité, il serait aussi intéressant d'étudier l'impact de l'ajout de protections contre les attaques en fautes et certaines attaques DPA sur les performances de calcul et la surface de circuit de nos cryptoprocesseurs. Enfin, nous n'avons pas eu le temps durant cette thèse d'étudier et d'implanter d'autres types d'architectures plus « exotiques » dans nos cryptoprocesseurs. Par exemple, nous aurions souhaité concevoir et évaluer des architectures de type « flot de données » nativement parallèles et permettant de limiter les problèmes de congestion mémoire. Ce type de topologie est aussi intéressante pour la mise en place de protections contre la DPA car elle pourrait faciliter la *randomisation* des calculs. Cela semble une perspective intéressante pour des travaux futurs.

Bibliographie personnelle

[GCT17] G. Gallin, T. O. Celik, and A. Tisserand. Architecture level optimizations for Kummer based HECC on FPGAs. In *Proc. 18th International Conference on Cryptology in India (Indocrypt)*, December 2017. Camera ready at `https://hal.archives-ouvertes.fr/hal-01614063/`. ⟨ pp. 10, 94, 104, 132, 135 ⟩

[GT17d] G. Gallin and A. Tisserand. Hyper-threaded multiplier for HECC. In *Asilomar Conference on Signals, Systems and Computers*. IEEE, October 2017. Camera ready at `https://hal.archives-ouvertes.fr/hal-01620046/`. ⟨ pp. i, iii, v, 8, 9, 39, 41, 57, 62, 63, 64, 65, 66, 67, 68, 69, 70, 73, 74, 83, 94, 104, 112, 118, 133, 136 ⟩

[GT18a] G. Gallin and A. Tisserand. Generation of hyper-threaded GF(P) multipliers for flexible curve based cryptography on FPGAs. *submitted to IEEE Transactions on Computers*, 2018. ⟨ pp. i, iii, v, 8, 9, 39, 41, 58, 68, 70, 71, 74, 75, 78, 81, 83, 94, 104, 131, 132, 136 ⟩

[GT18b] G. Gallin and A. Tisserand. Generator for hyper-threaded modular multipliers. `https://sourcesup.renater.fr/projects/htmm/`, March 2018. ⟨ pp. 9, 58, 75, 76, 83, 136 ⟩

Références non citées

[Gal17] G. Gallin. Architectures matérielles pour la cryptographie sur courbes hyper-elliptiques. Séminaire sécurité des systèmes électroniques embarqués DGA – IRISA [`http://securite-elec.irisa.fr`], December 2017. Invited talk.

[GT15] G. Gallin and A. Tisserand. Comparaison expérimentale d'architectures de crypto-processeurs pour courbes elliptiques et hyper-elliptiques. Journées nationales Codage et Cryptographie (JC2), October 2015. Talk by G. Gallin.

[GT16] G. Gallin and A. Tisserand. Hardware and arithmetic for hyperelliptic curves cryptography. Colloque annuel international du labex CominLabs, November 2016. Poster available at `https://hal.archives-ouvertes.fr/hal-01404755`.

[GT17a] G. Gallin and A. Tisserand. Architecture level optimizations for Kummer based HECC on FPGAs. 15th International Workshop on cryptographic architectures embedded in logic devices (CryptArchi), June 2017. Talk by A. Tisserand (slides available at `https://hal.archives-ouvertes.fr/hal-0154562`).

[GT17b] G. Gallin and A. Tisserand. Finite field multiplier architectures for hyper-elliptic curve cryptography. 12ème Colloque national du GDR SOC2, June 2017. Poster presented by A. Tisserand (available at `https://hal.archives-ouvertes.fr/hal-01545625`).

[GT17c] G. Gallin and A. Tisserand. Hardware architectures exploration for hyper-elliptic curve cryptography. 6ème Colloque national Crypto'Puces, June 2017. Talk by G. Gallin (slides available at `https://hal.archives-ouvertes.fr/hal-01547034`).

[GVCT15a] G. Gallin, N. Veyrat-Charvillon, and A. Tisserand. Comparaison expérimentale d'architectures de crypto-processeurs pour courbes elliptiques et hyper-elliptiques. In *Proc. Conférence nationale d'informatique en Parallélisme, Architecture et Système (Compas)*, June 2015. best paper award for computer architecture track (paper available at `https://hal.archives-ouvertes.fr/hal-01171094`).

[GVCT15b] G. Gallin, N. Veyrat-Charvillon, and A. Tisserand. Experimental comparison of crypto-processors architectures for elliptic and hyper-elliptic curves cryptography. 13th International Workshop on cryptographic architectures embedded in logic devices (CryptArchi), June 2015. Talk by G. Gallin (slides available at `https://hal.archives-ouvertes.fr/hal-01197048`).

[GVCT15c] G. Gallin, N. Veyrat-Charvillon, and A. Tisserand. Hardware and arithmetic for hyperelliptic curves cryptography. Rencontres nationales Arithmétiques de l'Informatique Mathématique (RAIM), 2015. Poster available at `https://hal.archives-ouvertes.fr/hal-01134020`.

[GVCT15d] G. Gallin, N. Veyrat-Charvillon, and A. Tisserand. Hardware and arithmetic for hyperelliptic curves cryptography. Colloque annuel international du labex CominLabs, March 2015. Poster.

Bibliographie générale

[AAKPM15] H.-R. Ahmadi, A. Afzali-Kusha, M. Pedram, and M. Mosaffa. Flexible prime-field genus 2 hyperelliptic curve cryptography processor with low power consumption and uniform power draw. *ETRI journal*, 37(1) :107–117, February 2015. ⟨ p. 99 ⟩

[ACZ16] D. Amiet, A. Curiger, and P. Zbinden. Flexible FPGA-based architectures for curve point multiplication over GF(p). In *Euromicro Conf. on Digital System Design (DSD)*, pages 107–114, August 2016. ⟨ pp. 58, 78, 83 ⟩

[ANS11] ANSSI. Avis relatif aux paramètres de courbes elliptiques définis par l'État français. Journal Officiel de la République Française, October 2011. NOR : PRMD1123151V. ⟨ p. 19 ⟩

[AR14] H. Alrimeih and D. Rakhmatov. Fast and flexible hardware support for ECC over multiple standard prime fields. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(12) :2661–2674, January 2014. ⟨ pp. 6, 40, 127 ⟩

[ASMT17] N. Alimi, A. Sghaier, M. Machhout, and R. Tourki. Exploring the design space of curve-based cryptographic accelerators. In *Proc. International Conference on Engineering MIS (ICEMIS)*, pages 1–5, May 2017. ⟨ p. 98 ⟩

[Ava04] R. M. Avanzi. Aspects of hyperelliptic curves over large prime fields in software implementations. In *Proc. 6th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 3156, pages 148–162, August 2004. ⟨ p. 4 ⟩

[Bar84] P. Barrett. *Communications Authentication and Security Using Public Key Encryption : A Design for Implementation*. PhD thesis, University of Oxford, 1984. ⟨ pp. 16, 48 ⟩

[BCHL16] J. W. Bos, C. Costello, H. Hisil, and K. Lauter. Fast cryptography in genus 2. *Journal of Cryptology*, 29(1) :28–60, January 2016. Online publication 15th November 2014. ⟨ pp. 4, 30, 35 ⟩

[BCLW02] N. Boston, T. Clancy, Y. Liow, and J. Webster. Genus two hyperelliptic curve coprocessor. In *Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 2523, pages 400–414, August 2002. ⟨ p. 95 ⟩

[BCR+18] E. Barker, L. Chen, A. Roginsky, A. Vassilev, and R. Davis. SP 800-56A : Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography. NIST Special Publication, April 2018. Revision 3. ⟨ p. 4 ⟩

[BDBK09] D. C. Black, J. Donovan, B. Bunton, and A. Keist. *SystemC : From the Ground Up*. 2nd edition, 2009. ⟨ p. 112 ⟩

[BDL97] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. In *Proc. International Conference on the Theory and Applications of Cryptographic Techniques – Advances in Cryptology (EUROCRYPT)*, volume 1233, pages 37–51, May 1997. ⟨ p. 7 ⟩

[BDL+11] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. In *Proc. 13th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, volume 6917, pages 124–142, September 2011. see also full version [BDL+12]. ⟨ p. 37 ⟩

[BDL+12] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2) :77–89, 2012. ⟨ p. 140 ⟩

[Ber06] D. J. Bernstein. Curve25519 : New Diffie-Hellman speed records. In *Proc. 9th International Workshop on Public Key Cryptography (PKC)*, volume 3958, pages 207–228, April 2006. ⟨ pp. 7, 18, 19, 129 ⟩

[BJ02] E. Brier and M. Joye. Weierstraß elliptic curves and side-channel attacks. In *Proc. 5th International Workshop on Public Key Cryptography (PKC)*, volume 2274, pages 335–345, February 2002. ⟨ pp. 27, 28 ⟩

[BL] D. J. Bernstein and T. Lange. Explicit-formulas database. `http://hyperelliptic.org/EFD/`. ⟨ pp. 18, 19, 20, 21, 22 ⟩

[BL13] D. J. Bernstein and T. Lange. Safecurves : choosing safe curves for elliptic-curve cryptography. `http://safecurves.cr.yp.to`, 2013. ⟨ pp. 18, 95 ⟩

[Bla83] G. R. Blakley. A computer algorithm for calculating the product A*B modulo M. *IEEE Trans. on Comp.*, C-32(5) :497 – 500, May 1983. ⟨ pp. 16, 48 ⟩

[BMPV06] L. Batina, N. Mentens, B. Preneel, and I. Verbauwhede. Flexible hardware architectures for curve-based cryptography. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 4839–4842, May 2006. ⟨ pp. 96, 97 ⟩

[BP01] T. Blum and C. Paar. High-radix Montgomery modular exponentiation on reconfigurable hardware. *IEEE Trans. on Comp.*, 50(7) :759–764, July 2001. ⟨ p. 58 ⟩

[Can87] D. G. Cantor. Computing in the Jacobian of a hyperelliptic curve. *Mathematics of Computation*, 48(177) :95–101, January 1987. ⟨ pp. 4, 32, 95, 98 ⟩

[CFA+05] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography.* Discrete Mathematics and Its Applications. Chapman & Hall/CRC, July 2005. ⟨ pp. 14, 98 ⟩

[CG03] L. Cai and D. Gajski. Transaction level modeling : An overview. In *Proc. 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS, pages 19–24. ACM, October 2003. ⟨ p. 111 ⟩

[CL15] C. Costello and P. Longa. FourQ : Four-dimensional decompositions on a Q-curve over the Mersenne prime. In *Proc. 21st International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, volume 9452, pages 214–235, November 2015. ⟨ pp. 19, 130 ⟩

[Cor99] J.-S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *Proc. 1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 1717, pages 292–302, August 1999. ⟨ pp. 27, 28 ⟩

[dDP09] F. de Dinechin and B. Pasca. Large multipliers with fewer DSP blocks. In *Proc. 19th International Conference on Field Programmable Logic and Applications (FPL)*, pages 250–255, August 2009. ⟨ p. 63 ⟩

[DH76] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6) :644–654, 1976. ⟨ pp. 2, 3, 36 ⟩

[DK90] S. R. Dusse and B. S. Kaliski. A cryptographic library for the Motorola DSP56000. In *Proc. Workshop on the Theory and Application of Cryptographic Techniques – Advances in Cryptology (EUROCRYPT)*, pages 230–244, 1990. ⟨ pp. vii, 12, 50, 51 ⟩

[EMY07] G. Elias, A. Miri, and T.-H. Yeap. On efficient implementation of FPGA-based hyperelliptic curve cryptosystems. *Computers & Electrical Engineering*, 33(5) :349–366, July 2007. ⟨ p. 97 ⟩

[FBV08] J. Fan, L. Batina, and I. Verbauwhede. HECC goes embedded : an area-efficient implementation of HECC. In *Proc. 15th Workshop on Selected Areas in Cryptography (SAC)*, volume 5381, pages 387–400, August 2008. ⟨ p. 97 ⟩

[FV12] J. Fan and I. Verbauwhede. An updated survey on secure ECC implementations : Attacks, countermeasures and cost. In *Cryptography and Security : From Theory to Applications – Essays dedicated to Jean-Jacques Quisquater on the occasion of his 65th birthday*, pages 265–282. 2012. ⟨ p. 28 ⟩

[GAKCL12] S. Gao, D. Al-Khalili, N. Chabini, and P. Langlois. Asymmetric large size multipliers with optimised FPGA resource utilisation. *IET Computers & Digital Techniques*, 6 :372–383, November 2012. ⟨ p. 63 ⟩

[Gau07] P. Gaudry. Fast genus 2 arithmetic based on theta functions. *Journal of Mathematical Cryptology*, 1(3) :243–265, August 2007. ⟨ pp. 4, 34, 35 ⟩

[GG16] S. D. Galbraith and P. Gaudry. Recent progress on the elliptic curve discrete logarithm problem. *Designs, Codes and Cryptography*, 78(1) :51–72, 2016. ⟨ p. 6 ⟩

[GH00] P. Gaudry and R. Harley. Counting points on hyperelliptic curves over finite fields. In *Proc. 4th Symposium on Algorithmic Number Theory (ANTS)*, volume 4, pages 313–332, July 2000. ⟨ pp. 32, 34 ⟩

[GMC10] S. Ghosh, D. Mukhopadhyay, and D. Chowdhury. High speed Fp multipliers and adders on FPGA platform. In *Conf. Design and Architectures for Signal and Image Processing (DASIP)*, pages 21–26, October 2010. ⟨ p. 48 ⟩

[GP08] T. Guneysu and C. Paar. Ultra high performance ECC over NIST primes on commercial FPGAs. In *Proc. 10th Internaltional Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 5154, pages 62–78, August 2008. ⟨ pp. 47, 127 ⟩

[GS12] P. Gaudry and E. Schost. Genus 2 point counting over prime fields. *Journal of Symbolic Computation*, 47(4) :368–400, April 2012. ⟨ pp. 30, 99 ⟩

[GVY17] D. Genkin, L. Valenta, and Y. Yarom. May the fourth be with you : A microarchitectural side channel attack on several real-world applications of Curve25519. In *Proc. of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 845–858, October 2017. ⟨ p. 6 ⟩

[HAH] HAH – hardware and arithmetic for hyperelliptic curves cryptography. `https://h-a-h.cominlabs.u-bretagneloire.fr/fr/presentation`. ⟨ p. 5 ⟩

[Ham15] M. Hamburg. Ed448-Goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625, June 2015. `https://eprint.iacr.org/2015/625`. ⟨ p. 19 ⟩

[Har00] R. Harley. Fast arithmetic on genus 2 curves. available at `http://cristal.inria.fr/\~harley/hyper`, 2000. ⟨ pp. 32, 98 ⟩

[HGEG11] M. Huang, K. Gaj, and T. El-Ghazawi. New hardware architectures for Montgomery modular multiplication algorithm. *IEEE Trans. on Comp.*, 60(7) :923–936, July 2011. ⟨ p. 59 ⟩

[HMV04] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. 2004. ⟨ pp. 14, 20, 22, 24, 25, 26, 37, 104 ⟩

[IEE12] IEEE. IEEE Standard for Standard SystemC Language Reference Manual - Redline. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005) - Redline*, pages 1–1163, January 2012. ⟨ p. 112 ⟩

[JMAL16] K. Jarvinen, A. Miele, R. Azarderakhsh, and P. Longa. FourQ on FPGA : New hardware speed records for elliptic curve cryptography over large prime characteristic fields. In *Proc. 18th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, volume 9813, pages 517–537, August 2016. ⟨ pp. 19, 129 ⟩

[JQ01] M. Joye and J.-J. Quisquater. Hessian elliptic curves and side-channel attacks. In *Proc. 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 2162, pages 402–410, May 2001. ⟨ p. 28 ⟩

[JWS15] K. Javeed, X. Wang, and M. Scott. Serial and parallel interleaved modular multipliers on FPGA platform. In *Internat. Conf. on Field Programmable Logic and Applications (FPL)*, pages 1–4, September 2015. ⟨ p. 48 ⟩

[JWS17] K. Javeed, X. Wang, and M. Scott. High performance hardware support for elliptic curve cryptography over general prime field. *Microprocessors and Microsystems*, 51 :331–342, June 2017. ⟨ p. 48 ⟩

[JY02] M. Joye and S.-M. Yen. The Montgomery powering ladder. In *Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 2523, pages 291–302, August 2002. ⟨ pp. vii, 28, 29 ⟩

[KAK96] C. K. Koc, T. Acar, and B. S. Kaliski, Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3) :26–33, June 1996. ⟨ pp. v, vii, 8, 11, 40, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 71, 75, 135 ⟩

[Kal93] B. S. Kaliski. The Z80180 and big-number arithmetic. *Dr Dobb's Journal-Software Tools for the Professional Programmer*, 18(9) :50–59, 1993. ⟨ pp. vii, 12, 51, 53, 55 ⟩

[KG13] C. F. Kerry and P. D. Gallagher. FIPS PUB 186-4 : Digital signature standard (DSS). Federal Information Processing Standards Publication, July 2013. ⟨ pp. 3, 19, 28, 36, 37 ⟩

[KJB99] P. C. Kocher, J. Jaffe, and Jun B. Differential power analysis. In *Proc. 19th International Cryptology Conference – Advances in Cryptology (CRYPTO)*, pages 388–397, August 1999. ⟨ pp. 7, 27 ⟩

[KM03] D. Koufaty and D. T. Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2) :56–65, March 2003. ⟨ pp. 8, 40, 60, 136 ⟩

[KM15] N. Koblitz and A. Menezes. A riddle wrapped in an enigma. Cryptology ePrint Archive, Report 2015/1018, revised May 2018, December 2015. `https://eprint.iacr.org/2015/1018`. ⟨ p. 5 ⟩

[Knu98] D. E. Knuth. *The art of computer programming, Seminumerical algorithms*, volume 2. Addison – Wesley, 3rd edition, 1998. ⟨ pp. 16, 23 ⟩

[Kob87] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177) :203–209, 1987. ⟨ pp. 3, 14 ⟩

[Kob88] N. Koblitz. A family of Jacobians suitable for discrete log cryptosystems. In *Proc. 5th Conference on the Theory and Application of Cryptography – Advances in Cryptology (CRYPTO)*, volume 403, pages 94–99, August 1988. ⟨ pp. 4, 14, 28, 29, 95 ⟩

[Kob89] N. Koblitz. Hyperelliptic cryptosystems. *Journal of Cryptology*, 1(3) :139–150, October 1989. ⟨ pp. 32, 95 ⟩

[Koc96] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proc. 16th International Cryptology Conference – Advances in Cryptology (CRYPTO)*, pages 104–113, August 1996. ⟨ pp. 7, 27 ⟩

[KSHS18] P. Koppermann, F. De Santis, J. Heyszl, and G. Sigl. Fast FPGA implementations of Diffie-Hellman on the Kummer surface of a genus-2 curve. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1) :1–17, 2018. ⟨ pp. v, 6, 101, 102, 103 ⟩

[KVV10] M. Knezevic, F. Vercauteren, and I. Verbauwhede. Faster interleaved modular multiplication based on Barrett and Montgomery reduction methods. *IEEE Trans. on Comp.*, 59(12) :1715–1721, December 2010. ⟨ p. 48 ⟩

[KWC+04] H.W. Kim, T. Wollinger, Y.J. Choi, K.I. Chung, and C. Paar. Hyperelliptic curve coprocessors on a FPGA. In *Proc. 5th International Workshop on Information Security Applications (WISA)*, volume 3325, pages 360–374, August 2004. ⟨ pp. 96, 97 ⟩

[Lac13] P. Lacharme. Sécurité du passeport électronique : 10 ans après son lancement, quelles lecons en tirer ? In *8ème conférence sur la sécurité des architectures réseaux et des systèmes d'information (SARSSI)*, 2013. ⟨ p. 4 ⟩

[Lan02] T. Lange. Inversion-Free Arithmetic on Genus 2 Hyperelliptic Curves. Cryptology ePrint Archive, Report 2002/147, 2002. ⟨ pp. 4, 33 ⟩

[Lan05] T. Lange. Formulae for Arithmetic on Genus 2 Hyperelliptic Curves. *Applicable Algebra in Engineering, Communication and Computing*, 15(5) :295–328, February 2005. ⟨ pp. 4, 33, 135 ⟩

[Len87] H. W. Lenstra. Factoring integers with elliptic curves. *Annals of Mathematics*, 126(3) :649–673, November 1987. ⟨ p. 3 ⟩

[LLMP93] A. K. Lenstra, H. W. Lenstra, M. S. Manasse, and J. M. Pollard. The number field sieve. In *The development of the number field sieve*, volume 1554 of *Lecture Notes in Mathematics*, pages 11–42, 1993. ⟨ p. 3 ⟩

[LM05] T. Lange and P. K. Mishra. SCA resistant parallel explicit formula for addition and doubling of divisors in the Jacobian of hyperelliptic curves of genus 2. In *Proc. 6th International Conference on Cryptology in India (Indocrypt)*, volume 3797, pages 403–416, December 2005. ⟨ p. 33 ⟩

[LM10] M. Lochter and J. Merkle. ECC brainpool standard curves & curve generation. `https://tools.ietf.org/pdf/rfc5639.pdf`, 2010. visited July 2018. ⟨ p. 19 ⟩

[LS01] P.-Y. Liardet and N. Smart. Preventing SPA/DPA in ECC systems using the Jacobi form. In *Proc. 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 2162, pages 391–401, May 2001. ⟨ p. 28 ⟩

[LWH12] J.-Y. Lai, Y.-S. Wang, and C.-T. Huang. High-performance architecture for elliptic curve cryptography over prime fields on FPGAs. *Interdisciplinary Information Sciences*, 18(2) :167–173, 2012. ⟨ p. 127 ⟩

[MBCM17] P. M. Massolino, L. Batina, R. Chaves, and N. Mentens. Area-optimized Montgomery multiplication on IGLOO2 FPGAs. In *Internat. Conf. Field Programmable Logic and Applications (FPL)*, pages 1–4, September 2017. ⟨ pp. 59, 78 ⟩

[MEML+17] A. Mrabet, N. El-Mrabet, R. Lashermes, J.-B. Rigaud, B. Bouallegue, S. Mesnager, and M. Machhout. A scalable and systolic architectures of Montgomery modular multiplication for public key cryptosystems based on DSPs. *Journal of Hardware and Systems Security*, 1(3) :219–236, September 2017. ⟨ pp. 58, 78 ⟩

[Mer78] R. C. Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21(4) :294–299, April 1978. ⟨ pp. 2, 3 ⟩

[Mét16] J. Métairie. *Contributions aux opérateurs arithmétiques* GF(2^m) *et leurs applications à la cryptographie sur courbes elliptiques.* PhD thesis, Université Rennes 1, 2016. ⟨ p. 17 ⟩

[Mil85] V. S. Miller. Use of elliptic curves in cryptography. In *Proc. Conference on the Theory and Application of Cryptographic techniques – Advances in Cryptology (CRYPTO)*, volume 218, pages 417–426, 1985. ⟨ pp. 3, 14 ⟩

[MLPJ13] Y. Ma, Z. Liu, W. Pan, and J. Jing. A high-speed elliptic curve cryptographic processor for generic curves over GF(p). In *Proc. International Workshop on Selected Areas in Cryptography (SAC)*, volume 8282, pages 421–437, August 2013. ⟨ pp. v, 57, 58, 59, 62, 65, 67, 78, 83, 127, 129 ⟩

[MMM04a] C. McIvor, M. McLoone, and J. McCanny. FPGA Montgomery modular multiplication architectures suitable for ECCs over GF(p). In *IEEE Internat. Symp. on Circuits and Systems (ISCAS)*, volume 3, pages 509–512, May 2004. ⟨ p. 54 ⟩

[MMM04b] C. McIvor, M. McLoone, and J. McCanny. FPGA Montgomery multiplier architectures - a comparison. In *IEEE Symp. Field-Programmable Custom Computing Machines (FCCM)*, pages 279–282, April 2004. ⟨ p. 54 ⟩

[MMM04c] M. McLoone, C. McIvor, and J. McCanny. Coarsely integrated operand scanning (CIOS) architecture for high-speed Montgomery modular multiplication. In *IEEE Internat. Conf. on Field-Programmable Technology*, pages 185–191, December 2004. ⟨ p. 54 ⟩

[MMM05] C. McIvor, M. McLoone, and J. McCanny. High-radix systolic modular multiplication on reconfigurable hardware. In *IEEE Internat. Conf. on Field-Programmable Technology*, pages 13–18, December 2005. ⟨ p. 58 ⟩

[Mon85] P. L. Montgomery. Modular multiplication without trial division. *Math. of Comp.*, 44(170) :519–521, April 1985. ⟨ pp. 8, 16, 40, 48, 49, 50, 52 ⟩

[Mon87] P. L. Montgomery. Speeding the Pollar and elliptic curves methods of factorisation. *Mathematics of Computation*, 48(177) :243–264, January 1987. ⟨ pp. 28, 98, 100 ⟩

[MOP08b] S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks : Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008. ⟨ p. 27 ⟩

[MSDP16] M. Morales-Sandoval and A. Diaz-Perez. Scalable GF(p) Montgomery multiplier based on a digit-digit computation approach. *IET Computers & Digital Techniques*, 10(3) :102–109, May 2016. ⟨ pp. 58, 78 ⟩

[NIS16] NISTIR 8105 : Report on post-quantum cryptography. `http://dx.doi.org/10.6028/NIST.IR.8105`, April 2016. ⟨ p. 5 ⟩

[OBPV03] S. Ors, L. Batina, B. Preneel, and J. Vandewalle. Hardware implementation of a Montgomery modular multiplier in a systolic array. In *Internat. Parallel and Distributed Processing Symp.*, pages 184–191, April 2003. ⟨ p. 58 ⟩

[Oru95] H. Orup. Simplifying quotient determination in high-radix modular multiplication. In *Symp. on Computer Arithmetic (ARITH)*, pages 193–199. IEEE, July 1995. ⟨ pp. vii, 54, 57, 59, 82, 129 ⟩

[PQ12] C. Petit and J.-J. Quisquater. On polynomial systems arising from a Weil descent. In *Proc. 18th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, pages 451–466, 2012. ⟨ pp. 7, 18, 95 ⟩

[PWP04] J. Pelzl, T. Wollinger, and C. Paar. High performance arithmetic for special hyperelliptic curve cryptosystems of genus two. In *Proc. International Conference on Information Technology : Coding and Computing (ITCC)*, volume 2, pages 513–517, April 2004. ⟨ p. 96 ⟩

[RMIT14] D. B. Roy, D. Mukhopadhyay, M. Izumi, and J. Takahashi. Tile before multiplication : An efficient strategy to optimize DSP multiplier for accelerating prime field ECC for NIST curves. In *Proc. 51st Annual Design Automation Conference (DAC)*, pages 177–177, June 2014. ⟨ p. 63 ⟩

[RSA77] R. L. Rivest, A. Shamir, and L. Adleman. On digital signatures and public-key cryptosystems. Technical report, Massachusetts Institute of Technology (MIT), Cambridge lab. for computer science, April 1977. ⟨ p. 3 ⟩

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2) :120–126, February 1978. ⟨ pp. 3, 12 ⟩

[RSSB16] J. Renes, P. Schwabe, B. Smith, and L. Batina. μKummer : Efficient hyperelliptic signatures and key exchange on microcontrollers. In *Proc. 18th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, volume 9813, pages 301–320, August 2016. ⟨ pp. vii, 4, 7, 9, 34, 35, 37, 99, 100, 101, 102, 103, 106, 116, 132, 135, 136 ⟩

[sak13] Side-channel AttacK User Reference Architecture (SAKURA). `http://satoh.cs.uec.ac.jp/SAKURA/hardware/SAKURA-G.html`, 2013. visited August 2018. ⟨ pp. 41, 114 ⟩

[SBPV06] K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede. Superscalar coprocessor for high-speed curve-based cryptography. In *Proc. 8th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 4249, pages 415–429, October 2006. ⟨ p. 96 ⟩

[SC10] S. Srinath and K. Compton. Automatic generation of high-performance multipliers for FPGAs with asymmetric multiplier blocks. In *Proc. 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 51–58, February 2010. ⟨ p. 63 ⟩

[Sch15] R. Scheidler. Hyperelliptic curve arithmetic. Slides presented at the 12th International Conference on Finite Fields and Their Applications – available at `www.skidmore.edu/fq12/uploads/Scheidler.pdf`, July 2015. visited August 2018. ⟨ pp. 31, 32 ⟩

[SEC10] Standards for efficient cryptography – SEC2 : Recommended elliptic curve domain parameters v2.0. Certicom Research, `www.secg.org/sec2-v2.pdf`, January 2010. visited July 2018. ⟨ pp. 19, 28 ⟩

[SG15] P. Sasdrich and T. Guneysu. Implementing Curve25519 for side-channel–protected elliptic curve cryptography. *ACM Trans. Reconfigurable Technol. Syst.*, 9(1) :3 :1–3 :15, November 2015. ⟨ pp. 19, 129, 130 ⟩

[SG17] P. Sasdrich and T. Guneysu. Cryptography for next generation TLS : Implementing the RFC 7748 elliptic curve448 cryptosystem in hardware. In *Proc. 54th Annual Design Automation Conference (DAC)*, pages 16 :1–6, June 2017. ⟨ p. 19 ⟩

[Sma99] N. P. Smart. On the performance of hyperelliptic cryptosystems. In *Proc. International Conference on the Theory and Applications of Cryptographic Techniques – Advances in Cryptology (EUROCRYPT)*, pages 165–175, 1999. ⟨ p. 4 ⟩

[SMZM16] A. Sghaier, C. Massoud, M. Zeghid, and M. Machhout. Flexible hardware implementation of hyperelliptic curves cryptosystem. *International Journal of Computer Science and Information Security (IJCSIS)*, 14(4), April 2016. ⟨ p. 98 ⟩

[SS99] N. P. Smart and S. Siksek. A fast Diffie–Hellman protocol in genus 2. *Journal of Cryptology*, 12(1) :67–73, January 1999. ⟨ pp. 33, 34 ⟩

[SSI98] Y. Sakai, K. Sakurai, and H. Ishizuka. Secure hyperelliptic cryptosystems and their performance. In *Proc. 1st International Workshop on Public Key Cryptography (PKC)*, pages 164–181, 1998. ⟨ p. 4 ⟩

[Suz07] D. Suzuki. How to maximize te potential of FPGA resources for modular exponentiation. In *Proc. 9th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 272–288, September 2007. ⟨ p. 57 ⟩

[TK99] A. F. Tenca and C. K. Koc. A scalable architecture for Montgomery multiplication. In *Proc. 1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 1717, pages 94–108, August 1999. ⟨ p. 59 ⟩

[TK03] A. F. Tenca and C. K. Koc. A scalable architecture for modular multiplication based on Montgomery's algorithm. *IEEE Trans. on Comp.*, 52(9) :1215–1221, September 2003. ⟨ p. 59 ⟩

[TT03] A. F. Tenca and L. A. Tawalbeh. An efficient and scalable radix-4 modular multiplier design using recoding techniques. In *Asilomar Conf. on Signals, Systems Computers*, volume 2, pages 1445–1450, November 2003. ⟨ p. 59 ⟩

[TTK01] A. F. Tenca, G. Todorov, and C. K. Koc. High-radix design of a scalable modular multiplier. In *Proc. 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 2162, pages 185–201, May 2001. ⟨ p. 59 ⟩

[Wal93] C. D. Walter. Systolic modular multiplication. *IEEE Trans. on Comp.*, 42(3) :376–378, March 1993. ⟨ p. 58 ⟩

[Wal99a] C. D. Walter. Montgomery exponentiation needs no final subtractions. *Electronics Letters*, 35(21) :1831–1832, October 1999. ⟨ pp. iii, 49, 50, 51, 52, 54, 55, 56, 59, 60, 71, 75 ⟩

[Wol01] T. Wollinger. Computer architectures for cryptosystems based on hyperelliptic curves. Master's thesis, Worcester Polytechnic Institute., 2001. ⟨ p. 95 ⟩

[Wol04] T. Wollinger. *Software and hardware implementation of hyperelliptic curve cryptosystems.* PhD thesis, 2004. ⟨ p. 96 ⟩

[WPP05] T. Wollinger, J. Pelzl, and C. Paar. Cantor versus Harley : optimization and analysis of explicit formulae for hyperelliptic curve cryptosystems. *IEEE Transactions on Computers*, 54(7) :861–872, July 2005. ⟨ pp. 32, 135 ⟩

[Xil08a] Xilinx. *UG070 : Virtex-4 User Guide*, December 2008. v2.6. ⟨ pp. 43, 47 ⟩

[Xil08b] Xilinx. *UG073 : XtremeDSP for Virtex-4 FPGAs User Guide*, May 2008. v2.7. ⟨ p. 44 ⟩

[Xil09] Xilinx. *Virtex-4 FPGA Data Sheet 302 : DC and Switching Characteristics*, September 2009. v3.7. ⟨ p. 45 ⟩

[Xil10] Xilinx. *UG384 : Spartan-6 Configurable Logic Block User Guide*, February 2010. v1.1. ⟨ pp. 42, 43 ⟩

[Xil11] Xilinx. *UG383 : Spartan-6 FPGA Block RAM Resources User Guide*, July 2011. v1.5. ⟨ pp. iii, 44, 45, 47 ⟩

[Xil12a] Xilinx. *UG190 : Virtex-5 User Guide*, March 2012. v5.4. ⟨ pp. 43, 47 ⟩

[Xil12b] Xilinx. *UG193 : Virtex-5 FPGA XtremeDSP Design Considerations User Guide*, January 2012. v3.5. ⟨ p. 44 ⟩

[Xil14b] Xilinx. *Virtex-5 FPGA Data Sheet 202 : DC and Switching Characteristics*, December 2014. v5.4. ⟨ p. 45 ⟩

[Xil15] Xilinx. *Spartan-6 FPGA Data Sheet 162 : DC and AC Switching Characteristics*, January 2015. v3.1.1. ⟨ p. 45 ⟩

[Xil16a] Xilinx. *UG473 : 7 Series FPGAs Memory Resources User Guide*, September 2016. v1.12. ⟨ p. 47 ⟩

[Xil16b] Xilinx. *UG474 : 7 Series FPGAs Configurable Logic Block User Guide*, September 2016. v1.8. ⟨ p. 43 ⟩

[Xil17] Xilinx. *Virtex-7 T and XT FPGAs Data Sheet 183 : DC and AC Switching Characteristics*, April 2017. v1.27. ⟨ pp. 45, 46 ⟩

[Xil18b] Xilinx. *UG479 : 7 Series DSP48E1 Slice User Guide*, March 2018. v1.10. ⟨ pp. 44, 46 ⟩

[YJ00] S.-M. Yen and M. Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers*, 49(9) :967–970, September 2000. ⟨ pp. 7, 27 ⟩

UNIVERSITE BRETAGNE LOIRE MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Unités arithmétiques et cryptoprocesseurs matériels pour la cryptographie sur courbe hyperelliptique

De nombreux systèmes numériques nécessitent des primitives de cryptographie asymétrique de plus en plus performantes mais aussi robustes aux attaques et peu coûteuses pour les applications embarquées. Dans cette optique, la cryptographie sur courbe hyperelliptique (HECC) a été proposée comme une alternative intéressante aux techniques actuelles du fait de corps finis plus petits. Nous avons étudié des cryptoprocesseurs HECC matériels performants, flexibles et robustes contre certaines attaques physiques. Tout d'abord, nous avons proposé une nouvelle architecture d'opérateurs exécutant, en parallèle, plusieurs multiplications modulaires $(A \times B) \bmod P$, où P est un premier générique de quelques centaines de bits et configurable dynamiquement. Elle permet le calcul de la grande majorité des opérations nécessaires pour HECC. Nous avons développé un générateur d'opérateurs, distribué en logiciel libre, pour l'exploration de nombreuses variantes de notre architecture. Nos meilleurs opérateurs sont jusqu'à 2 fois plus petits et 2 fois plus rapids que les meilleures solutions de l'état de l'art. Ils sont aussi flexibles quant au choix de P et atteignent les fréquences maximales du FPGA. Dans un second temps, nous avons développé des outils de modélisation et de simulation pour explorer, évaluer et valider différentes architectures matérielles pour la multiplication scalaire dans HECC sur les surfaces de Kummer. Nous avons implanté, validé et évalué les meilleures architectures sur différents FPGA. Elles atteignent des vitesses similaires aux meilleures solutions comparables de l'état de l'art, mais pour des surfaces réduites de moitié. La flexibilité obtenue permet de modifier lors de l'exécution les paramètres des courbes utilisées.

Title : Hardware arithmetic units and cryptoprocessors for hyperelliptic curve cryptography.

Many digital systems require primitives for asymmetric cryptography that are more and more efficient but also robust to attacks and inexpensive for embedded applications. In this perspective, and thanks to smaller finite fields, hyperelliptic curve cryptography (HECC) has been proposed as an interesting alternative to current techniques. We have studied efficient and flexible hardware HECC cryptoprocessors that are also robust against certain physical attacks. First, we proposed a new operator architecture able to compute, in parallel, several modular multiplications $(A \times B) \bmod P$, where P is a generic prime of a few hundred bits and configurable at run time. It allows the computation of the vast majority of operations required for HECC. We have developed an operator generator, distributed in free software, for the exploration of many variants of our architecture. Our best operators are up to 2 times smaller and twice as fast as the best state-of-the-art solutions. They are also flexible in the choice of P and reach the maximum frequencies of the FPGA. In a second step, we developed modeling and simulation tools to explore, evaluate and validate different hardware architectures for scalar multiplication in HECC on Kummer surfaces. We have implemented, validated and evaluated the best architectures on various FPGA. They reach speeds similar to the best comparable solutions of the state of the art, but for halved surfaces. The flexibility obtained makes it possible to modify the parameters of the curves used during execution.