

Étude et conception d'opérateurs arithmétiques

Arnaud Tisserand

▶ **To cite this version:**

Arnaud Tisserand. Étude et conception d'opérateurs arithmétiques. Informatique [cs]. Université Rennes 1, 2010. tel-00502465

HAL Id: tel-00502465
https://theses.hal.science/tel-00502465

Submitted on 15 Jul 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HABILITATION À DIRIGER DES RECHERCHES

présentée devant

L'Université de Rennes 1
Spécialité : Informatique

par

Arnaud Tisserand

Étude et conception d'opérateurs arithmétiques

soutenue le 6 juillet 2010 devant le jury composé de :

M. Daniel Etiemble, Professeur Université Paris Sud

M. Guy Gogniat, Professeur Université Bretagne Sud

M. Bernard Goossens, Professeur Université Perpignan

M. Dominique Lavenier, Professeur ENS Cachan Antenne de Bretagne

M. Jean-Michel Muller, Directeur de Recherche CNRS, Président du jury

M. Olivier Sentieys, Professeur Université Rennes 1

et au vu des rapports de :

M. Daniel Etiemble, Professeur Université Paris Sud

M. Bernard Goossens, Professeur Université Perpignan

M. Paolo Ienne, Professeur EPFL Lausanne Suisse

Table des matières

Avant-propos **5**

1. Introduction **7**
1.1. Arithmétique des ordinateurs . . . 7
1.1.1. Représentations des nombres . . . 8
1.1.2. Algorithmes . . . 10
1.2. Implantations . . . 12
1.3. Validation au niveau arithmétique . . . 14
1.4. Outils et support pour l'arithmétique des ordinateurs . . . 16

2. Résumé des travaux **19**
2.1. Arithmétique en ligne . . . 20
2.2. Architectures reconfigurables . . . 21
2.2.1. Architecture FPPA . . . 22
2.2.2. Architecture FPOP . . . 23
2.3. Méthodes à base de tables . . . 28
2.3.1. Méthode à base de tables et de petits multiplieurs . . . 29
2.3.2. Méthode à base de tables et additions, méthode multipartite . . . 30
2.3.3. Méthode à base de tables et polynômes très particuliers . . . 32
2.4. Division pour circuits asynchrones . . . 35
2.5. Opérateurs arithmétiques spécifiques pour FPGA . . . 36
2.6. Variantes de la multiplication . . . 37
2.6.1. Multiplication par des constantes . . . 38
2.6.2. Multiplication tronquée . . . 39
2.6.3. Fonction puissance entière x^n . . . 40
2.7. Bibliothèques flottantes pour processeurs entiers . . . 40
2.7.1. Bibliothèque flottante pour processeurs CoolRisc . . . 40
2.7.2. Bibliothèque FLIP pour processeur VLIW ST200 . . . 41
2.8. Division par des constantes . . . 43
2.9. Approximations polynomiales . . . 43
2.10. Calcul à basse consommation d'énergie . . . 44
2.10.1. Opérateurs arithmétiques pour la basse consommation . . . 44
2.10.2. Modélisation et évaluation de la consommation d'opérateurs arithmétiques 45
2.10.3. Consommation d'énergie dans les processeurs graphiques . . . 47
2.11. Opérateurs arithmétiques pour la cryptographie . . . 48
2.11.1. Arithmétique dans les corps finis . . . 48
2.11.2. Fonctions de hachage cryptographique . . . 49
2.11.3. Sécurisation d'opérateurs arithmétiques pour la cryptographie . . . 49
2.12. Générateur de diviseur `divgen` . . . 50
2.13. Bibliothèque logicielle PACE pour la cryptographie . . . 51

2.14. Maîtrise des erreurs d'arrondi dans les outils de CAO . . . 56
2.15. Générateurs aléatoires TRNG . . . 57
2.16. Arithmétique par estimation . . . 58

3. Opérateurs arithmétiques matériels pour l'évaluation de fonctions par approximation polynomiale 61
3.1. Introduction . . . 61
3.2. Notations et état de l'art . . . 62
3.2.1. Notations . . . 62
3.2.2. Erreurs . . . 62
3.2.3. Polynôme minimax . . . 63
3.2.4. Vers des polynômes d'approximation avec des coefficients représentables . 63
3.2.5. Génération des meilleurs approximants . . . 64
3.2.6. Outils pour le calcul de bornes d'erreur globale . . . 65
3.3. Méthode d'optimisation proposée . . . 66
3.3.1. Calcul du polynôme minimax . . . 67
3.3.2. Détermination des coefficients du polynôme et de leur taille . . . 68
3.3.3. Détermination de la taille du chemin de données . . . 69
3.3.4. Résumé de la méthode . . . 69
3.4. Exemples d'applications sur FPGA . . . 70
3.4.1. Fonction 2^x sur $[0, 1]$ . . . 70
3.4.2. Racine carrée sur $[1, 2]$ . . . 73
3.5. Conclusion et perspectives . . . 75

4. Conclusion et perspectives 77

5. Autres activités 81
5.1. Encadrement . . . 81
5.2. Enseignements . . . 83
5.3. Administration et tâches collectives . . . 83
5.4. Animation . . . 84
5.5. Collaborations académiques . . . 84
5.6. Collaborations et contrats industriels . . . 86
5.7. Développement de logiciels . . . 87
5.8. Participation à des comissions de spécialistes . . . 87
5.9. Édition, comités de programmes, relectures . . . 88
5.10. Expertises . . . 88

6. Bibliographie personnelle 89

7. Bibliographie générale 95

A. Sélection de publications 99
A.1. Opérateurs en arithmétique en ligne pour le contrôle numérique . . . 99
A.2. Opérateurs à base de petits multiplieurs pour l'évaluation de fonctions . . . 105
A.3. Méthode des tables multipartites . . . 116
A.4. Optimisation de la multiplication par des constantes . . . 129
A.5. Génération d'approximations polynomiales efficaces en machine . . . 142

Avant-propos

Ce mémoire résume les activités effectuées en recherche, encadrement, animation et enseignement depuis ma thèse soutenue en septembre 1997. Ces activités ont été réalisées dans plusieurs contextes. Une partie a été effectuée lors de mon séjour post-doctoral dans le groupe *Ultra Low Power* de la division microélectronique du Centre suisse d'électronique et microtechnique (CSEM) à Neuchâtel en Suisse entre octobre 1997 et septembre 1999. Entre octobre 1999 et septembre 2005, j'ai travaillé dans l'équipe Arénaire du Laboratoire de l'informatique du parallélisme (Lip) à Lyon sur un poste de chargé de recherche (CR) à l'Institut national de recherche en informatique et automatique (Inria). Entre octobre 2005 et novembre 2008, j'ai travaillé dans l'équipe Arith du Laboratoire d'informatique robotique et microélectronique de Montpellier (Lirmm) sur un poste de CR au Centre national de la recherche scientifique (CNRS). Enfin, depuis décembre 2008, je travaille, toujours comme CR CNRS, dans l'équipe Cairn de l'Institut de recherche en informatique et systèmes aléatoires (Irisa) à Lannion.

Mes travaux de recherche portent essentiellement sur l'*arithmétique des ordinateurs*, en *matériel* et en *logiciel*. Ces travaux comportent des liens avec les domaines de la conception de circuits intégrés numériques, de l'architecture des machines et du développement logiciel de bibliothèques de calcul. Les principaux domaines d'application de ces travaux sont : le calcul numérique dans les systèmes embarqués, la cryptographie asymétrique, la sécurité numérique, le traitement numérique du signal et des images et de façon plus limitée les dispositifs numériques de contrôle–commande en automatique.

Ce mémoire se décompose comme suit. Le chapitre 1 est une courte introduction à l'arithmétique des ordinateurs et au contexte des travaux. Le chapitre 2 résume les principaux travaux effectués ou en cours. Quelques articles parus dans des journaux, et reproduits en annexe A, décrivent les travaux les plus significatifs. Le chapitre 3 détaille un peu plus l'un des axes de recherche étudié ces dernières années : la conception d'opérateurs arithmétiques pour l'évaluation de fonctions en matériel. J'ai choisi ce thème car il me semble bien illustrer l'ensemble de mes travaux. En particulier, il montre les différentes *facettes* étudiées en arithmétique des ordinateurs : les représentations des nombres, les algorithmes de calculs, les implantations efficaces, les problèmes de validation et les outils d'aide à la conception de circuits arithmétiques. La conclusion, au chapitre 4, dresse un bilan des travaux effectués ou en cours et propose quelques perspectives. Une grande partie des travaux présentés dans ce mémoire a été réalisée en collaboration avec d'autres chercheurs (doctorants, permanents, stagiaires de M2R–DEA ou ingénieurs). Ces collaborations locales, nationales et internationales, seront rapidement présentées dans les différentes sections techniques. Les activités en encadrement, animation, enseignement, expertise et administration sont résumées dans le chapitre 5.

Les travaux décrits dans ce mémoire ont été publiés dans des journaux, des conférences ou en rapport de recherche. Les références à ces publications sont regroupées dans la bibliographie personnelle au chapitre 6. Les autres références bibliographiques sont regroupées au chapitre 7. La liste complète de mes publications est disponible sur ma page web [1]. Ces publications sont disponibles, quand les droits le permettent, sur cette page web. Deux ordres des auteurs sont util-

1. Page web accessible à l'adresse : `http://www.irisa.fr/prive/Arnaud.Tisserand`

isés dans mes publications. Ceci correspond à des modes de fonctionnement différents propres à chaque communauté. Dans la communauté arithmétique des ordinateurs, issue de l'informatique, l'ordre des auteurs est l'ordre alphabétique car tous les auteurs participent de façon significative aux publications. Dans la communauté issue de la micro-électronique et du traitement du signal, c'est l'ordre d'implication dans la publication qui est utilisé avec éventuellement des participations essentiellement administratives en fin de liste (p. ex. codirections de thèses). Je m'efforcerai de préciser la nature et le degré d'implication de mes participations dans les travaux collectifs.

Enfin, les annexes, à partir de la page 99, reproduisent une sélection de quelques publications illustrant les principaux résultats résumés au chapitre 2.

Chapitre 1

Introduction

Cette introduction décrit les différentes facettes de l'arithmétique des ordinateurs étudiées dans nos travaux. Elle précise aussi le contexte de ces travaux ainsi que certaines notations utilisées dans ce mémoire. Elle se décompose en quatre sections. La section 1.1 présente quelques notions d'arithmétique des ordinateurs sur les représentations des nombres et les algorithmes de calcul. La section 1.2 introduit les principales cibles utilisées pour nos implantations matérielles et logicielles. La section 1.3 présente quelques problèmes et méthodes de validation au niveau arithmétique. Enfin, la section 1.4 résume les principales difficultés et quelques solutions proposées sur les outils et le support aux utilisateurs pour la conception et l'utilisation d'opérateurs arithmétiques en matériel.

1.1. Arithmétique des ordinateurs

L'*arithmétique des ordinateurs* est la branche de l'informatique qui traite des représentations des nombres et des algorithmes pour effectuer les calculs de base en machine. Cette première définition informelle nécessite quelques précisions. Les représentations des nombres et les algorithmes seront abordés respectivement aux sous-sections 1.1.1 et 1.1.2.

Par *calculs de base*, nous désignons les opérations suivantes : addition, soustraction, comparaisons, multiplication, division, inverse, racine carrée et son inverse. À ces opérations nous ajoutons des variantes comme l'incrémentation, la multiplication par une ou des constantes, le carré, le cube et la norme euclidienne. Nous ajoutons aussi l'évaluation de fonctions plus complexes comme les fonctions trigonométriques, le logarithme, l'exponentielle, etc. On s'intéresse aussi à la composition de plusieurs de ces opérations. On considère aussi le cas de calculs sur des objets composés de plusieurs nombres comme le calcul sur les nombres complexes. À ces opérations, classiques en calcul scientifique, on ajoute des opérations spécifiques nécessaires dans certaines applications. On a, par exemple, des filtres numériques en traitement du signal, des transformées en cosinus discret (DCT) et des sommes de différences de valeurs absolues (SAD) en multimédia. Il en va de même pour certaines primitives cryptographiques comme l'exponentiation modulaire et l'addition de points d'une courbe elliptique. Par contre, la réalisation de calculs évolués sur des objets plus complexes ne rentre pas dans le cadre de notre travail (par exemple des opérations matricielles en algèbre linéaire [89]).

Par *en machine*, on entend à la fois la réalisation des calculs dans des circuits intégrés numériques et dans des processeurs généralistes ou dédiés. Nous visons des opérateurs dédiés à certaines applications (approximation de fonctions, primitives cryptographiques, filtres numériques, DCT, etc.) mais aussi des ressources généralistes (unités arithmétiques, opérateurs flottants, etc.). Plus de détails seront donnés en section 1.2.

Comme dans de nombreux domaines de l'informatique, le choix de la structure des données et des algorithmes de traitement associés influence grandement les performances obtenues. Dans la suite, nous utiliserons souvent le terme *arithmétique* comme un ensemble de systèmes de représentations des nombres et des algorithmes de calcul associés. Ces deux aspects sont étroitement liés. L'un de nos principaux objectifs est de trouver, pour un problème donné, une bonne *adéquation* entre une arithmétique et son implantation matérielle ou logicielle. Les critères

que nous regarderons pour qualifier cette adéquation sont la vitesse ou la durée des calculs, la surface de silicium, la taille des programmes, la taille mémoire, le nombre de registres intermédiaires nécessaires et la consommation d'énergie. Souvent la précision des calculs est ajoutée à ces critères. Pour nous, la précision est plus une contrainte à garantir dans nos applications. Par exemple, nous étudions des opérateurs arithmétiques permettant de réaliser des opérations avec une précision cible minimale donnée. Le problème de la validation de la précision de nos opérateurs, ou plus généralement, de leur bon comportement mathématique, sera abordé en section 1.3. La robustesse aux attaques par canaux cachés ou par injection de fautes est aussi un critère de qualité des opérateurs arithmétiques utilisés dans des cryptosystèmes. Nous verrons en section 2.11.3 que mesurer cette robustesse n'est pas une chose simple.

Plusieurs ouvrages traitent d'arithmétique des ordinateurs : [83] est la référence la plus complète actuellement en anglais, [112] est une très bonne référence en français. Avec Marc Daumas et Florent de Dinechin, nous avons été éditeurs invités pour un numéro spécial sur l'arithmétique des ordinateurs dans la revue « Réseaux et systèmes répartis, calculateurs parallèles » [21] en 2001. À cette occasion, nous avons écrit une introduction à l'arithmétique des ordinateurs [20] et sur quelques thèmes de recherche abordés dans la communauté française de ce domaine.

1.1.1. Représentations des nombres

Il existe une grande variété de *systèmes de représentations des nombres*, voir [91, 93] pour des aspects historiques et [112, 83] pour une revue des principales représentations utilisées en arithmétique des ordinateurs. Les travaux présentés dans ce mémoire utilisent principalement des représentations en numération de position. Dans cette catégorie de représentation, les nombres sont représentés par des séquences de chiffres. Les chiffres sont des éléments de l'ensemble (fini) des chiffres possibles. Chaque chiffre est associé à une puissance de la base. La base choisie est souvent 2 ou une puissance de 2 pour des raisons d'implantation. Nous utiliserons des chiffres entiers (naturels ou relatifs). Dans la suite, nous manipulerons essentiellement des entiers et des approximations des réels en virgule fixe. Nous utiliserons un peu la représentation approchée des réels en virgule flottante dans la section 2.7.2 consacrée à la bibliothèque FLIP. Cette bibliothèque fournit un support flottant pour des processeurs particuliers qui intègrent uniquement des unités entières ou en virgule fixe.

On note $()_2$ la représentation binaire d'une valeur, par exemple $12345 = (11000000111001)_2$ ou $3.125 = (11.001)_2$. La notation classique en base 10 est simplifiée : pas de $()_{10}$. La notation de la virgule suit le standard anglo-saxon, avec un point, afin de simplifier les interactions avec des programmes. On représente un entier positif a en numération simple de position, sur n bits, par la suite $(a_{n-1}a_{n-2}\dots a_1a_0)_2$, où les chiffres a_i appartiennent à l'ensemble de chiffres $\{0,1\}$ et où la valeur du nombre est $\sum_{i=0}^{n-1} a_i 2^i$. Un chiffre a_i, de rang ou position i, est associé au poids 2^i. En virgule fixe avec k bits entiers et l bits fractionnaires, on représente un nombre a dont la valeur est $\sum_{i=-l}^{k} a_i 2^i$. Pour la représentation des nombres relatifs, nous utiliserons suivant les notations suivantes : complément à deux, signe et valeur absolue (aussi appelée signe et magnitude) et redondante à chiffres signés. Le livre [112] présente en détail ces différentes représentations.

En numération de position de base β, on peut autoriser les chiffres à avoir d'autres valeurs entières que celles de l'ensemble habituel $\{0,1,2,\dots,\beta-1\}$. En 1961, Avizienis [70] a proposé des *représentations redondantes* des nombres très utiles en arithmétique des ordinateurs où, en base β, l'ensemble de chiffres contient strictement plus de β éléments et il est symétrique autour de 0. On note $()_{\beta,\mathcal{D}}$ la notation en base β avec l'ensemble de chiffres $\mathcal{D}$. Les chiffres négatifs (seulement les chiffres, pas les nombres) seront notés avec une barre au-dessus de leur valeur

absolue afin d'éviter les confusions avec le signe de l'opération de soustraction. Par exemple, le chiffre -1 est noté $\overline{1}$. Avoir une représentation redondante des nombres signifie que certains nombres possèdent plusieurs écritures. Par exemple, en base $\beta = 10$, si on utilise l'ensemble de chiffres $\mathcal{D} = \{\overline{9}, \ldots, \overline{1}, 0, 1, \ldots, 9\}$, alors le nombre 2010 peut s'écrire $(2010)_{10,\mathcal{D}}$, $(21\overline{9}0)_{10,\mathcal{D}}$, $(3\overline{99}0)_{10,\mathcal{D}}$, $(1\overline{8}010)_{10,\mathcal{D}}$, $(1\overline{8}1\overline{9}0)_{10,\mathcal{D}}$, etc. Le nombre 9999 s'écrit assez élégamment $(1000\overline{1})_{10,\mathcal{D}}$. Les représentations redondantes à chiffres signées sont très anciennes. En 1840, par exemple, Cauchy propose de les utiliser pour limiter les erreurs lors de longues séquences de calculs effectués manuellement [76]. Ce problème de validation était important à cette époque pour bon nombre de professions.

Les représentations redondantes permettent de calculer certaines opérations plus rapidement. L'addition et la soustraction s'effectuent en un temps indépendant de la taille des nombres ou temps constant. On parle souvent d'addition et de soustraction sans propagation de retenue. Les éventuelles retenues générées à un certain rang ne peuvent pas se propager plus loin que le rang immédiatement supérieur. Ceci brise, de façon systématique au niveau de chaque rang, la chaîne de dépendance des propagations de retenues, ce qui permet bien d'avoir la propriété d'addition/soustraction en temps constant. Ces représentations permettent d'accélérer notablement des suites d'additions ou de soustractions. Ceci est particulièrement utile pour effectuer la somme des produits partiels dans l'opération de multiplication. Par contre, pour une seule opération d'addition ou de soustraction, une représentation redondante n'est pas intéressante si il faut la somme (ou la différence) en représentation classique (non redondante). Le coût de la conversion d'une représentation redondante vers une représentation non redondante est celui d'une « vraie » addition avec propagation de retenues (en temps logarithmique avec la taille des opérandes pour les meilleurs algorithmes).

Dans le cas de la division, l'utilisation d'une notation redondante pour les chiffres du quotient dans l'algorithme SRT permet d'accélérer significativement les calculs [82]. À chaque itération, la sélection d'un nouveau chiffre du quotient est basée sur une estimation du reste partiel et du diviseur. Quelques chiffres de poids forts de ces deux valeurs servent d'entrée à une table donnant le nouveau chiffre du quotient. Les erreurs dues aux estimations sont corrigées dans les itérations suivantes. Si les erreurs sont limitées à chaque itération, l'écriture redondante du quotient permet de tendre vers la valeur théorique en s'autorisant des petites modifications. Dans SRT, la mise à jour du reste partiel, qui est une soustraction, s'effectue souvent en représentation redondante pour accélérer le calcul. L'algorithme SRT est un cas idéal pour étudier les représentations redondantes. La section 2.12 résume les travaux sur un générateur automatique de code VHDL pour des diviseurs SRT réalisé avec Nicolas Veyrat-Charvillon et Romain Michard.

Nous utiliserons beaucoup les représentations *carry-save* et *borrow-save*, cf. [83]. Ce sont des représentations redondantes de base 2. Les chiffres sont dans l'ensemble $\{0, 1, 2\}$ pour la représentation *carry-save* et dans l'ensemble $\{-1, 0, 1\}$ pour la représentation *borrow-save*. La représentation *carry-save*, notée $()_{\text{cs}}$, est utilisée pour effectuer la somme de plusieurs termes, comme pour la réduction des produits partiels dans les multiplieurs rapides. La représentation *borrow-save*, notée $()_{\text{bs}}$, permet d'éviter simplement des longues suites de chiffres à 1, par exemple $63 = (0111111)_2 = (100000\overline{1})_{\text{bs}}$. Elle nous sera particulièrement utile pour recoder des constantes afin de diminuer leurs nombres de bits non nuls et ainsi diminuer significativement la quantité de calculs lors de multiplications par ces constantes, cf. section 2.6.1 et chapitre 3.

Mais les représentations redondantes ont un coût. Comme il y a plus de chiffres à représenter que dans une notation non redondante, le stockage est plus coûteux. La base 2 constitue l'un des pires cas pour l'augmentation des besoins de mémorisation. On passe de 1 bit par chiffre en représentation classique à 2 bits par chiffre en *carry-save* et en *borrow-save*. Plus la base augmente moins le surcoût de mémorisation est important. Par exemple, en base 4 avec les

chiffres dans $\{-2,-1,0,1,2\}$, on a 3 bits par chiffre au lieu de 2. L'augmentation du nombre de portes logiques nécessaires pour traiter les chiffres codés sur plus de bits est souvent assez faible. En effet, la représentation redondante permet de se dispenser des algorithmes de propagation, ou de génération, rapides des retenues intermédiaires. Un additionneur *carry-save* ou *borrow-save* est bien plus petit, et régulier, qu'un additionneur rapide (non redondant). D'autre part, du fait de la redondance les comparaisons sont plus complexes (p. ex. $(2010)_{10,\mathcal{D}} = (1\bar{8}010)_{10,\mathcal{D}}$ dans l'exemple donné plus haut). Les représentations redondantes sont donc souvent utilisées dans des calculs intermédiaires mais rarement pour des résultats finaux.

Une autre représentation que nous commençons à utiliser est la représentation modulaire[1] des nombres ou RNS pour *residue number system* [118]. Elle est beaucoup utilisée en traitement du signal et un peu en cryptographie dans la mesure où elle fait apparaître un fort parallélisme. On représente un entier a par les valeurs a_i de a modulo m_i pour chacun des moduli m_i. L'ensemble des m_i est appelée base du système RNS. Le théorème des restes chinois donne une construction de a modulo $\prod m_i$ à partir des a_i quand les m_i sont premiers deux à deux. Les opérations d'addition et de multiplication s'effectuent en parallèle sur les différents moduli. La représentation RNS n'est pas un système de position. L'ordre des calculs sur les différents éléments de la base des moduli n'a aucune importance. C'est cette propriété qui utilisée dans [71] pour protéger des cryptosystèmes contre certaines attaques par canaux cachés (analyse de la consommation d'énergie, du rayonnement électromagnétique ou du temps de calcul).

Depuis quelques années, nous étudions aussi les représentations des nombres sur des corps finis $\mathbb{F}_q$ utilisés en cryptographie. Dans notre cas, nous nous limitons aux corps premiers $\mathbb{F}_p$ et aux extensions du corps binaire $\mathbb{F}_{2^m}$. Dans le cas $\mathbb{F}_p$, c'est l'arithmétique modulo un grand nombre premier p qui est utilisée. On utilise des représentations pour des grands nombres proches de la multiprécision en logiciel. Dans le cas $\mathbb{F}_{2^m}$, les éléments du corps sont représentés par des polynômes univariés avec des coefficients dans $\mathbb{F}_2$. Les nombres sont alors des suites de ces coefficients avec aussi un aspect multiprécision. Dans l'avenir, nous comptons étudier les caractéristiques de représentations pour des extensions de la forme $\mathbb{F}_{3^m}$.

1.1.2. Algorithmes

En arithmétique des ordinateurs, on classe habituellement les *opérations* ou *fonctions* à *évaluer* en plusieurs catégories :

- les *opérations arithmétiques* : addition, soustraction, multiplication et opérations dérivées (carré, cube, addition–multiplication fusionnée, multiplication par des constantes, etc.) ;
- les *fonctions algébriques* comme la division, l'inverse, la racine carrée et son inverse ou la norme euclidienne ;
- les *fonctions élémentaires* comme les fonctions trigonométriques, l'exponentielle, le logarithme, l'élévation à une puissance réelle, etc.

Cette décomposition correspond non seulement à des caractéristiques mathématiques différentes mais aussi à des complexités de mise en œuvre différentes pour les implantations des opérateurs matériels ou logiciels. Il existe d'autres catégories de fonctions à évaluer comme les fonctions spéciales (fonction gamma, fonction erreur, etc.), mais nous n'utilisons pas ces fonctions dans nos applications.

Pour les opérations arithmétiques, on sait *calculer exactement* le résultat en utilisant des algorithmes assez simples et directs (sans de nombreuses itérations qui convergent vers le résultat final). L'implantation de ces algorithmes donne lieu à des opérateurs rapides et autonomes (ne

1. À ne pas confondre avec l'arithmétique modulaire dans laquelle on effectue des opérations modulo un nombre premier.

nécessitant aucune autre ressource pour effectuer leurs calculs). En matériel, ces algorithmes s'implantent souvent comme des circuits combinatoires très efficaces. L'introduction de registres permet d'augmenter le débit ou bien de réduire la surface du circuit en « repliant » l'opérateur comme avec certaines décompositions récursives de la multiplication.

L'opération d'*addition* est de loin l'opération arithmétique la plus répandue. La recherche en arithmétique des ordinateurs est encore active pour cette opération tant le spectre des algorithmes et des implantations possibles est large [129]. En règle générale, lorsque l'on parle d'addition en arithmétique des ordinateurs, on sous-entend addition et soustraction car ces deux opérations sont très proches, en particulier dans des représentations comme le complément à deux ou le *borrow-save*.

La *multiplication* peut se ramener à une succession d'additions. En effet, il suffit d'effectuer la somme de tous les produits partiels. On nomme cette phase la *réduction* des produits partiels. La *génération* des produits partiels est massivement parallèle à un problème de sortance près. En effet, chaque chiffre de l'un des opérandes doit être multiplié par chacun des chiffres de l'autre. La sortance sur l'élément de mémorisation (souvent une bascule ou un verrou) qui stocke un bit des opérandes est alors très importante. Dans les multiplieurs de grande taille, la génération des produits partiels peut nécessiter jusqu'à 25% du temps de calcul total alors qu'en théorie une simple porte `ET` suffit pour chaque bit de produit partiel. Une utilisation astucieuse de recodages, comme celui de Booth [83], permet de réduire le nombre de produits partiels. La phase de réduction des produits partiels est théoriquement très simple. Il suffit de faire la somme en utilisant une notation redondante des nombres. En utilisant un arbre d'addition, on obtient la solution optimale en temps de calcul. Le problème ici est encore un problème de contraintes sur l'implantation et en particulier un problème de topologie. En effet, implanter un arbre sur une surface plane et si possible carrée ou rectangulaire n'est pas simple, cf. [86]. Enfin, la phase d'*assimilation* des retenues permet de transformer la représentation redondante de la somme des produits partiels en une représentation classique. En pratique, il s'agit d'une addition des composantes de l'écriture redondante de la somme des produits partiels.

Dans le cas des fonctions algébriques courantes dans les applications visées, on utilise essentiellement trois types d'algorithmes [83, 117] : les algorithmes à base d'additions et de décalages (*shift-and-add*), ceux basés sur des itérations de fonctions (style Newton-Raphson) et des approximations polynomiales. Dans les algorithmes à base d'additions et de décalages, le résultat est produit en commençant par les poids forts, chiffre par chiffre [82]. Les méthodes à additions et des décalages sont aussi appelées méthodes à récurrence de chiffres. La méthode utilise une représentation redondante du résultat, ce qui permet d'accélérer les calculs en limitant les multiples comparaisons qui doivent être faites à chaque itération pour sélectionner le nouveau chiffre du résultat. Les algorithmes à itération de fonction, comme Newton-Raphson ou Goldschmidt, utilisent une itération qui converge vers la valeur du résultat avec des additions et des multiplications [103]. On a alors une convergence quadratique (et même cubique dans certains cas). Cette méthode est très utilisée dans les processeurs généralistes mais elle est souvent désastreuse en termes de consommation énergétique par rapport à des opérateurs autonomes (comme des unités dédiées utilisant des algorithmes à additions et décalages). Nous avons travaillé sur les approximations polynomiales, cf. chapitre 3.

Les *fonctions élémentaires* sont utiles pour le calcul scientifique, mais aussi pour de plus en plus d'applications comme les applications graphiques ou le contrôle numérique. Contrairement aux opérations arithmétiques et aux fonctions algébriques, on ne sait pas calculer les fonctions élémentaires, sauf pour certains arguments triviaux, on sait seulement les approcher. Ici encore, on utilise trois types d'algorithmes pour évaluer les fonctions élémentaires [113] : les approximations polynomiales ou rationnelles, les algorithmes à base d'additions et de décalages et enfin

les méthodes à base de tables. Nous avons travaillé sur des approximations polynomiales (cf. chapitre 3) et sur des méthodes à base de tables (cf. section 2.3). Les méthodes à base de tables, très étudiées ces dernières années, permettent d'obtenir des opérateurs rapides pour des précisions jusqu'à une vingtaine de bits. Au delà, la taille des tables devient bien trop grande pour des implantations efficaces. Les algorithmes à base d'additions et de décalages fournissent des opérateurs de taille raisonnable mais nécessitent un grand nombre d'itérations pour fournir le résultat. Enfin, les approximations polynomiales s'imposent dans bon nombre d'applications pour des questions d'efficacité et de réutilisabilité des ressources de calcul. Les approximations rationnelles sont assez peu utilisées en pratique du fait du coup important de la division finale. Nous verrons en section 3.5, que ceci pourrait bien changer dans certains cas du fait des caractéristiques de circuits récents.

1.2. Implantations

La troisième facette de notre travail en arithmétique des ordinateurs concerne l'*implantation* des algorithmes de calcul et représentations des nombres. Cette facette est souvent très coûteuse en temps de travail pour des implantations d'arithmétiques évoluées. Dans nos travaux, ces implantations ont été réalisées majoritairement pour des *cibles matérielles* et un peu pour des *cibles logicielles*. Toutefois, notre implication dans des développements logiciels est importante même pour les cibles matérielles. En effet, la plupart des codes sources utilisés pour les implantations matérielles sont générés par des programmes spécifiques (cf. section 1.4). La plupart de nos réalisations matérielles a été implantée et validée sur des circuits FPGA (pour *field programmable gate arrays*). Cibler des implantations Asic (pour *application-specific integrated circuits*) ne nous était généralement pas possible pour des raisons budgétaires, en particulier, du fait des coûts de main d'œuvre sur l'installation et la maintenance des lourds outils de conception de circuits. Au CSEM, la structure de la division microélectronique nous permettait de cibler des circuits Asic. Ceci n'a plus été possible par la suite. À l'Irisa, nous espérons pourvoir cibler de nouveau des Asic. Les opérateurs présentés au chapitre 3 ont été implantés en matériel et en logiciel. Nous verrons quelles sont les implications des possibilités et contraintes des supports d'exécution sur les algorithmes de calcul et les représentations des nombres.

La bonne réussite de l'implantation pratique d'une arithmétique pour une application donnée dépend en partie des caractéristiques du support d'exécution. Par exemple, nous regardons comment proposer des dispositifs de calcul différents à partir des ressources disponibles dans les circuits FPGA actuels comme des petits blocs de multiplication câblée. Nous verrons que dans certains cas, il faudra adapter les algorithmes pour profiter au mieux des performances de certaines ressources matérielles comme pour les méthodes à base de tables (cf. section 2.3). Le travail d'implantation est souvent long et pas toujours simple à valoriser, et particulièrement lorsqu'il faut concevoir des outils pour effectuer ces implantations. Néanmoins, cet aspect de validation expérimentale par implantation de nos résultats est une motivation importante afin de prouver que nos solutions sont réellement viables en pratique. Ceci permet aussi d'obtenir des mesures de performances bien plus pertinentes que des nombres d'opérations ou de grossières évaluations de complexité.

L'implantation matérielle d'opérateurs arithmétiques et de représentations des nombres évolués est souvent difficile en pratique. Le concepteur doit jongler entre le large espace des paramètres possibles, la faiblesse des langages et outils de conception et les contraintes technologiques. L'évolution rapide des technologies demande de reconcevoir régulièrement des unités de calcul. Ceci demande beaucoup d'efforts et de temps. Malheureusement, cela limite souvent la complexité

des solutions envisagées dans bon nombre de circuits ou processeurs. Dans le cas des circuits Asic, nous nous limitons aux implantations basées sur des bibliothèques de cellules standards. À long terme, l'un de nos buts est de permettre l'implantation d'arithmétiques évoluées, à partir d'une description simple au niveau arithmétique, qui utiliseraient le type des cellules de la bibliothèque utilisée et des optimisations des outils.

Les circuits reconfigurables FPGA sont de plus en plus utilisés pour des implantations matérielles. Les plus petites performances de ces circuits par rapport aux circuits de type Asic sont souvent compensées par la rapidité de mise en œuvre, la réelle possibilité de prototypage à bas coût au niveau universitaire et le faible coût pour des petites séries lors de contrats industriels. L'implantation d'opérateurs arithmétiques en FPGA ne pose pas de problèmes théoriques plus complexes que pour des circuits intégrés classiques. C'est même un peu le contraire qui se produit en pratique. Les FPGA, bien que proches des circuits classiques, n'obéissent pas du tout aux mêmes métriques. Concernant le temps, par exemple, l'additionneur le plus simple à propagation séquentielle de retenue reste plus rapide qu'un additionneur plus sophistiqué et ce jusqu'à des tailles importantes. Ceci est lié à la logique dédiée à la propagation rapide des retenues ou *fast carry lines*. L'apparition de très nombreux blocs de multiplication câblée dans les FPGA biaise aussi un peu les choses. Par exemple, dans les FPGA Xilinx qui embarquent des dizaines ou des centaines de multiplieurs 18×18 câblés, utiliser des multiplications avec des opérandes de moins de 18 bits n'a aucun intérêt (même la réduction de la consommation est très faible). Dans les FPGA, ce sont aussi les aspects de surface qui sont modifiés. Comment comparer deux solutions dont une utilise les blocs de multiplication câblée et pas l'autre ? Nous verrons que certaines des ressources spécifiques aux circuits FPGA nous demandent d'utiliser des algorithmes un peu différents de ceux que l'on utilise pour les circuits intégrés classiques.

Le support de la virgule flottante dans les FPGA ne se pose que depuis peu. Les FPGA des générations précédentes étaient de taille trop limitée. Aujourd'hui des bibliothèques complètes et performantes, comme celle proposée par J. Detrey [79], permettent de synthétiser des opérateurs flottants en FPGA. Mais ici aussi, l'avenir des arithmétiques évoluées est incertain. Si des blocs câblés spécifiques à la virgule flottante font leur apparition dans les futures générations de FPGA, ce qui semble probable, le support d'autres représentations des nombres pour l'approximation des réels comme le système logarithmique deviendront probablement inefficaces en comparaison à des structures flottantes dédiées. En fait, pour des applications classiques, l'arithmétique à un niveau fin n'a plus vraiment de place dans les FPGA actuels où les structures arithmétiques élémentaires sont déjà câblées. Dans l'avenir, cette tendance risque de se renforcer avec des structures de FPGA dont la « granularité » est de plus en plus importante.

La basse consommation d'énergie est un critère de conception de plus en plus important, et même critique dans de nombreux cas. Il faut se poser la question de l'efficacité énergétique des calculs. Les interactions entre les algorithmes et les représentations des nombres sont nombreuses et assez peu étudiées. C'est une thématique vraiment intéressante, mais faute de réussite dans les demandes de financement de projets sur ce thème, nous ne travaillons que très peu sur ces aspects pour le moment. Nous espérons que cela changera prochainement.

Pour les implantations logicielles, nous nous sommes essentiellement intéressés au cas de processeurs dédiés (type DSP pour *digital signal processor*). En particulier, nous avons développé des algorithmes et des bibliothèques pour des processeurs embarqués dans des applications multimédia ou de traitement du signal (processeurs ST100 et ST200). Le type des représentations des nombres supportées directement par le processeur dépend des applications cibles. Les données sont souvent représentées en entier ou en virgule fixe pour des raisons de coût de silicium (les unités flottantes optimisées sont bien trop coûteuses pour ces applications). En plus d'unités arithmétiques et logiques (additions, soustractions, comparaisons, décalages et fonctions

logiques), on trouve souvent des unités de type MAC (pour *multiply and accumulate*) ou FMA (pour *fused multiply and add*) [2] dans ces processeurs. Notre travail sur la génération automatique de code de division par des constantes pour le processeur ST100, cf. section 2.8, illustre assez bien l'impact du type des unités de calcul sur les algorithmes arithmétiques mis au point. Celui sur le support flottant pour des processeurs entiers, cf. section 2.7.2, illustre l'impact des types de données supportés dans un processeur (ici ceux de la famille ST200) sur les arithmétiques. Depuis quelques temps, nous nous intéressons aussi au développement de bibliothèques arithmétiques rapides pour la cryptographie (cf. section 2.13).

1.3. Validation au niveau arithmétique

Une autre facette, complexe et multiforme, du problème de la conception et l'implantation d'algorithmes arithmétiques et de représentations des nombres est celui de leur *validation*. Ce problème est présent à différentes étapes. Lors la conception d'un opérateur arithmétique (algorithmes et représentations), la validation *a priori* est importante étant donnée la complexité des méthodes de détermination des paramètres. Ce problème est actuellement un problème important pour la communauté d'arithmétique des ordinateurs. Bon nombre de travaux, en France particulièrement, portent sur ce problème. Enfin, la validation *a posteriori* permet de vérifier le bon comportement des opérateurs arithmétiques réalisés. En plus du test traditionnel et indispensable des circuits après fabrication, il faut pouvoir garantir que les descriptions des opérateurs arithmétiques sont correctement utilisées. Les outils de synthèse et de compilation sont souvent très limités au niveau arithmétique. Il faut donc vérifier que les descriptions qui sont injectées dans ces outils sont totalement correctes et bien comprises. Par exemple, dans le domaine de l'arithmétique pour la cryptographie, l'implantation de multiplieurs modulaires de plusieurs milliers de bits ($a \times b \bmod p$) en utilisant des décompositions des opérandes en sous-mots est assez complexe. Comme garantir que la description VHDL, où dans tout autre langage, de l'algorithme est totalement correcte (découpages complets, machines d'états, manipulation des valeurs intermédiaires, etc.) ? Générer des cas de tests pathologiques pour des si grandes structures n'est pas chose facile.

La mise au point d'algorithmes arithmétiques utilisant des représentations des nombres évoluées, ou même exotiques, pose un problème important : comment garantir la validité des solutions proposées. Ce problème de validation est omniprésent à différentes étapes des travaux. Dans un premier temps, comment garantir que la preuve mathématique d'un algorithme est correcte ? Ensuite, la description de cet algorithme et des représentations des nombres utilisées dans un langage de programmation ou de description de matériel est-elle l'image fidèle de la solution théorique proposée ? En particulier, comment vérifier que les produits des outils comme les compilateurs ou les synthétiseurs sont corrects ? Loin de proposer des solutions à tous ces problèmes, nous avons essayé de « limiter les risques » en adoptant une méthodologie de validation basée sur différents niveaux de tests et différentes techniques basées sur des outils récents et proches du domaine de la preuve formelle.

La validation *a priori*, lors de la conception théorique, des opérateurs arithmétiques est souvent complexe. Ces algorithmes ne sont pas très longs en pratique, mais ils dépendent de nombreux paramètres. Par exemple, un algorithme SRT optimisé ne nécessite qu'une petite dizaine de lignes pour sa description. Mais la détermination des paramètres internes (p. ex. les tailles

2. On parle de MAC dans les processeur de type DSP (souvent pour la virgule fixe) et de FMA pour les processeurs généralistes (souvent pour la virgule flottante), mais tous effectuent des opérations de type $a \times b \pm c$ avec éventuellement accumulation.

optimales des estimations du reste partiel et du diviseur pour SRT) nécessite souvent l'écriture de programmes ou des calculs beaucoup assez longs. Prouver qu'un algorithme fraîchement conçu retourne toujours le bon résultat, et ce quels que soient les opérandes et les paramètres internes, est un problème difficile en informatique en général et pas seulement en arithmétique des ordinateurs. De nombreux travaux récents, et particulièrement en France, proposent des méthodes et des outils dans le domaine de la preuve formelle pour aider les concepteurs d'arithmétiques. En particulier, nous renvoyons aux travaux de jeunes chercheurs, S. Boldo [73] et G. Melquiond [108] pour ces aspects. Dans plusieurs travaux récents, cf. chapitre 3, nous utilisons le logiciel GAPPA [107] développé par G. Melquiond pour obtenir des preuves formelles d'opérateurs pour l'approximation de fonctions. Les preuves formelles générées par GAPPA sont ensuite vérifiées par l'assistant de preuves COQ [126]. De plus, nous utilisons le logiciel de calcul formel MAPLE [102] afin de vérifier certains calculs. Nous essayons, grâce à ces différents outils, de vérifier formellement le plus possible de nos algorithmes et méthodes d'optimisation.

L'implantation pratique d'arithmétiques en utilisant des langages de programmation ou de description de matériel pose aussi un problème de validation. En effet, il faut vérifier *a posteriori* le bon fonctionnement, ou comportement, des opérateurs arithmétiques réalisés. Les outils de synthèse pour le matériel ou bien les compilateurs pour le logiciel sont souvent très limités au niveau arithmétique. Il faut donc vérifier que les descriptions qui sont injectées dans ces outils sont bien comprises. On ne parle pas d'erreurs dans les outils, mais seulement d'un problème de la bonne forme des descriptions que l'on injecte dans ces outils. Un exemple simple est celui de l'addition binaire, normalement toute simple, en VHDL. Si on déclare A et B deux vecteurs de bits de taille n, l'opération VHDL $A + B$ retourne en fait $(A + B) \bmod 2^n$. Si l'on souhaite obtenir la somme complète sur $n + 1$ bits, il faut concaténer un zéro sur les poids forts de A et de B puis les additionner (en VHDL : `('0'\&A)+('0'\&B)`). Vérifier que l'on n'a pas oublié ce genre de détail n'est pas une chose simple en pratique étant donné de nombre de découpages de valeurs arithmétiques dans nos algorithmes. Le jour où un concepteur pourra déclarer deux signaux en représentation flottante IEEE 754 et en faire la division directement en VHDL (sans passer par des bibliothèques qui ne garantissent actuellement rien quant au respect de la norme) n'est pas encore venu.

Afin de valider nos opérateurs arithmétiques, nous utilisons des *simulations fonctionnelles* à plusieurs niveaux : exhaustif (lorsque c'est possible), vecteurs de valeurs choisies, vecteurs de valeurs aléatoires. Dans chacun de ces cas, on doit comparer la sortie du simulateur fonctionnel à une référence théorique. En pratique, nous utilisons soit le système de calcul formel MAPLE soit un programme spécifique développé en C ou C++ (utilisant des bibliothèques comme GMP [88] ou MPFR [87]) pour générer ces valeurs de référence. Remarquons que ces programmes spécifiques n'utilisent pas les mêmes algorithmes que ceux que nous testons. La probabilité de faire les « mêmes erreurs » est donc faible, mais clairement non nulle. En dehors du domaine de la cryptographie, pour bon nombre de nos opérateurs, le nombre de bits d'entrée est modéré : 8 à 24 bits typiquement. Pour quelques valeurs des paramètres, on peut très bien tester l'ensemble des entrées possibles en simulation fonctionnelle (par exemple avec un simulateur VHDL raisonnablement rapide comme MODELSIM). Dans ce cas, même si la couverture des tests effectués n'est pas totale (l'espace des paramètres internes étant souvent trop grand), ces simulations exhaustives, au niveau des opérandes, nous donnent une première garantie de la validité de nos opérateurs, au moins pour les jeux de paramètres testés.

Quand le nombre de bits d'entrée ou le nombre de paramètres devient trop important pour simuler exhaustivement nos opérateurs, on utilise des simulations à base de vecteurs de valeurs. En plus des tests classiques utilisant de très nombreuses valeurs aléatoires, nous essayons de générer automatiquement des vecteurs de test correspondants à des cas particuliers de nos algo-

rithmes. Un bon exemple est présenté dans nos travaux sur la bibliothèque FLIP, cf. section 2.7.2. Dans certains cas, nous sommes capables de générer automatiquement les cas extrêmes des découpages internes dans nos opérateurs.

Depuis quelques temps, nous nous intéressons à l'intégration de méthodes de *gestion des erreurs de calcul* dans les outils de conception assistée par ordinateur (CAO) pour les circuits intégrés numériques. En section 2.14, nous décrivons deux travaux en cours sur ce thème. Nous travaillons sur l'intégration de la gestion et l'optimisation des erreurs d'arrondi dans le langage SystemC. Nous faisons le lien entre des descriptions arithmétiques en SystemC et les bornes d'erreurs calculées par le logiciel GAPPA. Ceci permet de valider la précision d'opérateurs ou bien de calculer des tailles d'opérateurs sous contrainte de précision. D'un autre côté, avec le laboratoire VLSI CAD de l'Université du Massachusetts à Amherst aux USA, nous essayons d'intégrer la gestion des erreurs d'arrondis (calculées par GAPPA) dans leur outil de synthèse de haut niveau afin de pouvoir synthétiser des circuits avec des précisions validées à la conception.

Nous espérons pouvoir travailler, les prochaines années, sur la validation au niveau arithmétique dans les langages de haut niveau. Les progrès effectués ces dernières années sur les outils d'aide à la preuve formelle nous invitent à essayer de voir comment utiliser au mieux ces outils pour le développement et l'utilisation d'opérateurs arithmétiques. Dans le monde des circuits intégrés, la conception pour le test (ou DfT pour *design for testability* en anglais) est depuis longtemps une réalité. Dans le monde du logiciel, un grand chemin reste encore à parcourir.

1.4. Outils et support pour l'arithmétique des ordinateurs

Enfin, la dernière facette de nos travaux concerne le problème du *support utilisateur* lors de la conception de dispositifs faisant intervenir des arithmétiques évoluées. Le manque de support et d'outils au niveau arithmétique ne se fait pas uniquement ressentir lors de la phase de validation. Il apparaît dès les premières étapes de la conception d'algorithmes à un niveau fin. En effet, les outils actuels (langages de programmation et de description de matériel en particulier) ne supportent directement que des types arithmétiques très simples et un tout petit nombre d'opérations. Dans le cas du logiciel, de nombreuses bibliothèques arithmétiques sont disponibles, mais ce n'est pas du tout le cas pour le matériel.

Les seules représentations des nombres supportées dans les outils de synthèse sont la numération simple de position et le complément à deux pour les entiers et la virgule fixe. Les entiers sont assez bien pris en compte, mais c'est déjà moins évident pour la virgule fixe du fait du manque total d'aide à la gestion de la précision. Les langages de description de matériel comme VHDL et Verilog permettent même de déclarer des nombres flottants et de faire des opérations plus ou moins complexes sur ces flottants. Même si cela fonctionne bien en simulation (en utilisant les ressources flottantes du processeur qui exécute le simulateur assez souvent), aucune description utilisant des flottants n'est directement synthétisable pour le moment. Il faut alors utiliser des blocs IP (pour *intellectual property*) pour les opérateurs ou des bibliothèques spécifiques.

Au niveau des algorithmes pour effectuer les calculs de base, le constat est assez similaire. En règle générale, seuls quelques algorithmes simples, voire rudimentaires, sont utilisés dans les synthétiseurs. Une simple addition, ou soustraction, est bien gérée. Par contre, une séquence d'additions ou de soustractions de plusieurs valeurs de tailles différentes conduit souvent à un opérateur lent et gourmand en surface. Pour la multiplication, le cas des opérandes de même type et de même taille multiple d'une puissance de 2 est correcte (car souvent disponible sous forme de descriptions toutes prêtes pour ces paramètres précis). Pour des opérandes de tailles différentes et en mélangeant les représentations des opérandes, le résultat est assez souvent bien

inférieur à ce que l'on sait faire à la main même en utilisant des techniques simples. La solution retenue est basée sur des conversions et des extensions de tailles avant d'utiliser les descriptions toutes prêtes décrites plus haut. L'idée est de laisser faire les outils de simplification logique pour éliminer les portes inutiles. Mais en faisant ainsi, on se prive d'un grand nombre de simplifications au niveau arithmétique. Pour les autres opérations, les possibilités sont souvent quasiment nulles pour la synthèse.

La plupart des travaux présentés dans ce mémoire proposent des outils : générateurs automatiques ou bibliothèques. Une bibliothèque spécifique fournit soit directement des opérateurs particuliers soit un ensemble complet (opérateurs, types de données, propriétés, fonctions de test, etc.) permettant le support d'arithmétiques évoluées. Les générateurs sont des programmes autonomes qui permettent de produire automatiquement des descriptions particulièrement optimisées comme du code VHDL ou C de bas niveau (proche de l'assembleur). La plupart du temps, ces descriptions issues de générateurs ne peuvent pas être obtenues manuellement étant donnée la difficulté des problèmes mis en œuvre pour déterminer les paramètres algorithmiques évoqués plus haut. Tout comme les générateurs automatiques, le développement de bibliothèques pour améliorer le support en arithmétique des ordinateurs constitue une de nos activités importantes depuis de nombreuses années. Cet effort, long et pas toujours simple à valoriser, est nécessaire pour que les progrès réalisés en arithmétique des ordinateurs puissent se propager réellement en pratique.

Nous essayons de limiter le développement de bibliothèques d'opérateurs pour le matériel. Nous préférons développer des générateurs. Il y a plusieurs raisons dans ce choix. La principale raison est probablement la pauvreté des langages de description de matériel et en particulier de leurs sous-ensembles effectivement synthétisables. Décrire un arbre totalement générique (profondeur, type des nœuds, etc.) en VHDL reste particulièrement difficile au niveau syntaxique et de la structure de données dans le langage. Comment décrire de façon efficace en VHDL une arithmétique sur les corps finis avec de multiples représentations des nombres possibles suivant la caractéristique du corps ? Utiliser des langages de ce type pour manipuler des types de données assez complexes et garantir des propriétés de typage n'est pas vraiment la vocation première de ces langages. En fait, il n'y a pas vraiment de langage reconnu et diffusé qui soit orienté synthèse au niveau algorithmique aujourd'hui.

Afin de pouvoir utiliser des outils informatiques puissants comme le typage fort, le polymorphisme ou l'abstraction, nous utilisons des outils informatiques pour générer, plus ou moins automatiquement des descriptions de bas niveau de nos opérateurs. Toute la partie algorithmique et arithmétique est alors décrite dans des outils informatiques et c'est l'exécution de ces outils qui va produire une description optimisée d'une solution. Nous présentons dans la suite de ce mémoire différents générateurs d'opérateurs. Un problème qui commence à se produire est celui de la cohérence et de la simplicité du développement de systèmes complets quand il faut associer différents générateurs. Ceci constituera l'un de nos axes de recherche secondaires dans les toutes prochaines années.

Pour le logiciel, nous avons participé activement au développement de la bibliothèque FLIP, cf. section 2.7.2, permettant un support flottant sur des processeurs entiers. Nous participons aussi au développement d'une bibliothèque logicielle pour la cryptographie, cf. section 2.13.

Chapitre 2

Résumé des travaux

Sommaire

2.1. Arithmétique en ligne 20
2.2. Architectures reconfigurables 21
2.2.1. Architecture FPPA 22
2.2.2. Architecture FPOP 23
2.3. Méthodes à base de tables 28
2.3.1. Méthode à base de tables et de petits multiplieurs 29
2.3.2. Méthode à base de tables et additions, méthode multipartite 30
2.3.3. Méthode à base de tables et polynômes très particuliers 32
2.4. Division pour circuits asynchrones 35
2.5. Opérateurs arithmétiques spécifiques pour FPGA 36
2.6. Variantes de la multiplication 37
2.6.1. Multiplication par des constantes 38
2.6.2. Multiplication tronquée 39
2.6.3. Fonction puissance entière x^n 40
2.7. Bibliothèques flottantes pour processeurs entiers 40
2.7.1. Bibliothèque flottante pour processeurs CoolRisc 40
2.7.2. Bibliothèque FLIP pour processeur VLIW ST200 41
2.8. Division par des constantes 43
2.9. Approximations polynomiales 43
2.10. Calcul à basse consommation d'énergie 44
2.10.1. Opérateurs arithmétiques pour la basse consommation 44
2.10.2. Modélisation et évaluation de la consommation d'opérateurs arithmétiques 45
2.10.3. Consommation d'énergie dans les processeurs graphiques 47
2.11. Opérateurs arithmétiques pour la cryptographie 48
2.11.1. Arithmétique dans les corps finis 48
2.11.2. Fonctions de hachage cryptographique 49
2.11.3. Sécurisation d'opérateurs arithmétiques pour la cryptographie 49
2.12. Générateur de diviseur `divgen` 50
2.13. Bibliothèque logicielle PACE pour la cryptographie 51
2.14. Maîtrise des erreurs d'arrondi dans les outils de CAO 56
2.15. Générateurs aléatoires TRNG 57
2.16. Arithmétique par estimation 58

Les travaux résumés dans ce mémoire sont regroupés par thèmes dont la liste est donnée dans le sommaire ci-dessus. L'ordre de présentation de ces thèmes est assez proche de l'ordre chronologique de leur réalisation (ou de leur début pour les études les plus longues). Pour chacun de ces thèmes de recherche, nous terminons la description par un petit tableau récapitulatif contenant : le contexte (collaborations, période), les publications, les encadrements (niveau, étudiants, pourcentage d'encadrement, et devenir des étudiants quand il est connu), les réalisations particulières (logiciels, blocs matériels, prototypes), ainsi que les éventuels contrats (académiques ou industriels) liés au thème.

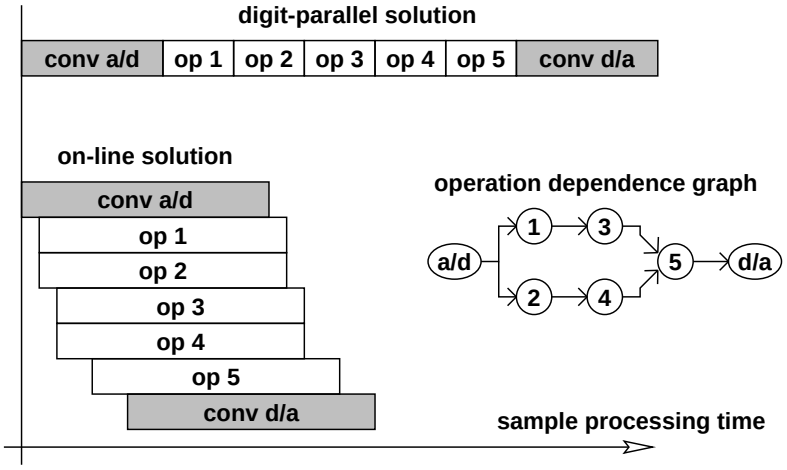


FIGURE 2.1.: Utilisation typique des arithmétiques parallèle et en ligne en contrôle numérique.

2.1. Arithmétique en ligne

L'*arithmétique en ligne*, cf. [83, chap. 9], est une arithmétique dans laquelle les nombres circulent en *série*, chiffre par chiffre, les *poids forts en tête*. Cette arithmétique permet d'obtenir des opérateurs de plus petite taille que leurs équivalents parallèles au prix d'un temps de calcul linéaire en la taille des opérandes. Le mode de communication, entre opérateurs, en série avec les poids forts en tête permet d'affiner les résultats à chaque nouveau cycle. On peut donc facilement concevoir des algorithmes où des décisions peuvent être prises en fonction de l'ordre de grandeur de certaines valeurs. Cette propriété est très intéressante pour des applications qui utilisent ou produisent des grandeurs physiques comme des signaux issus de capteurs ou des consignes d'actionneurs. Nous avons essentiellement utilisé l'arithmétique en ligne pour réaliser des contrôleurs numériques en automatique. L'arithmétique en ligne permet aussi un pipeline au niveau du chiffre très efficace lors de calculs successifs (cf. figure 2.1 et figure 1.c du papier présenté en section A.1). Toutes les opérations usuelles peuvent être effectuées en arithmétique en ligne en utilisant un système redondant de représentation des nombres. Ceci n'est pas le cas en série avec les poids faibles en tête où la comparaison et la division ne peuvent être effectuées sans devoir attendre de connaître complètement les opérandes. Il est clair que pour l'opération de comparaison, commencer par les poids forts permet de déterminer le résultat au plus vite.

Durant le séjour postdoctoral au CSEM, nous avons travaillé sur deux aspects assez différents liés à l'arithmétique en ligne :

- étude et conception d'opérateurs de calcul et de systèmes complets en arithmétique en ligne pour le contrôle numérique ;
- étude et conception d'une architecture reconfigurable, de type FPGA dédié, utilisant l'arithmétique en ligne, elle sera décrite en section 2.2.2.

L'étude et la conception d'opérateurs en arithmétique en ligne pour le contrôle numérique a débuté par une collaboration avec Martin Dimmler alors en thèse à l'Institut d'automatique de l'École polytechnique fédérale de Lausanne (EPFL) en Suisse. J'étais l'un des co-encadrants du CSEM qui finançait cette thèse. Nous avons amélioré la bibliothèque d'opérateurs de calcul en arithmétique en ligne commencée durant ma thèse [49] et nous avons implanté un système de régulation pour le contrôle numérique de la position d'un miroir dans une application de communication aérospatiale. Les résultats de ce travail ont été publiés dans [25] et [63]. Les opérateurs de base (addition, soustraction, multiplication, multiplication par une constante,

carrée et division) ont été optimisés et implantés sur des circuits FPGA Actel (technologie anti-fusible pour les applications aérospatiales). En dehors de la partie purement arithmétique, nous avons étudié les problèmes du contrôle des débuts et fin de calcul dans les réseaux d'opérateurs en ligne. Différentes solutions basées sur des contrôles centralisés ou distribués ont été étudiées et implantées. Nous avons proposé un système de contrôle totalement distribué basé sur une ligne de validité des données. Ce mécanisme, décrit dans [25] et [63], permet d'obtenir de très bonnes performances en vitesse (pas de latence additionnelle et temps de cycle minimal) pour un surcoût en surface très faible. Les détails de ce travail sont décrits dans l'article [25] publié dans le journal *IEEE/ASME Transactions on Mechatronics* en juin 1999 et reproduit en annexe A.1 (page 99). Nous avons comparé les performances et coûts d'opérateurs arithmétique en ligne en base 2 et en base 4. La version en base 4 était légèrement moins performante que la version en base 2. Ceci était lié à la petite taille des tables (*lookup table* LUT) disponibles dans les FPGA de l'époque. Il serait intéressant de refaire cette étude sur des FPGA récents (avec des LUT à 5 ou 6 entrées contre 4 à l'époque).

Avec Bernard Girau (LORIA), nous avons étudié l'implantation de réseaux de neurones, des perceptrons multi-couches, sur FPGA en utilisant l'arithmétique en ligne. Les résultats de ce travail ont été publiés dans le journal *International Journal of Systems Research and Information Science* en 1999 [32].

Entre 2005 et 2007, nous avons collaboré avec Rachid Beguenane et Stéphane Simard du laboratoire Ermétis de l'Université du Québec à Chicoutimi (UQAC) au Canada. Cette collaboration portait sur l'étude et l'implantation dans un système sur puce (SoC pour *system on chip*) de contrôle rapide mais sans capteur pour des moteurs électriques à induction. Ces systèmes sans capteurs nécessitent d'estimer en permanence différents paramètres (tensions, courants, flux, vitesse angulaire, etc.) afin d'alimenter un système de régulation assez complexe dont le temps de réponse global doit être de l'ordre de la microseconde. Seuls des systèmes basés sur plusieurs processeurs, et donc coûteux, étaient capables d'atteindre cette vitesse de régulation. Le pipeline au niveau du chiffre, permis dans des séquences de calculs en arithmétique en ligne, a été l'un des éléments clés pour l'efficacité en vitesse de ce travail. La petite taille des opérateurs de calcul en arithmétique en ligne a permis d'implanter tous les calculs sur une surface raisonnable. En particulier, nous avons travaillé sur l'optimisation et l'implantation d'opérateurs en ligne pour l'évaluation de fonctions à l'aide d'approximations polynomiales. Deux articles communs ont été publiés dans des conférences internationales : NEWCAS [2] et CCECE [1] en 2006. Nous avons commencé le co-encadrement de la thèse de Stéphane Simard sur ce thème. Bien malheureusement, il a souhaité arrêter sa thèse après une année, qui pourtant était prometteuse, pour travailler dans le secteur privé. Nous n'avons plus de nouvelles depuis. Nous espérons pouvoir retravailler avec l'UQAC dans l'avenir. L'arithmétique en ligne montre un véritable potentiel pour certaines applications d'automatique.

Récapitulatif	
Encadrements :	Martin Dimmler, doctorat 1999 [80] 30 %, maintenant chercheur ESO Stéphane Simard, doctorat, 20 %, arrêt après 1 an
Journal :	IEEE Trans. Mechatronics 1999 [25]
Conférences :	FPL 1997 [63], NEWCAS 2006 [2], CCECE 2006 [1]
Collaborations :	EPFL 1997–1999, UQAC 2005–2007

2.2. Architectures reconfigurables

L'étude et la conception d'architectures reconfigurables a été une activité importante lors du séjour postdoctoral au CSEM. Dans ce cadre, nous avons travaillé sur deux architectures reconfigurables différentes :

– une architecture reconfigurable massivement parallèle monopuce à basse consommation d'énergie et tolérante aux fautes : FPPA (cf. sous-section 2.2.1) ;
– une architecture reconfigurable et parallèle d'opérateurs de calcul en arithmétique en ligne et de convertisseurs ADC[1]/DAC[2] pour l'automatique : FPOP (cf. sous-section 2.2.2).

2.2.1. Architecture FPPA

Le projet FPPA, pour *field programmable processor array*, portait sur une architecture massivement parallèle monopuce composée d'une grille d'une centaine de petits processeurs avec leurs mémoires et d'un dispositif de communication asynchrone et reconfigurable. Le but de ce projet était l'étude et la mise en œuvre de mécanismes de tolérance aux pannes inhérentes aux grandes surfaces de silicium par reconfiguration, et de techniques permettant de limiter la consommation d'énergie associant le parallélisme et la réduction de la tension d'alimentation V_{DD}.

Le circuit FPPA est composé d'une matrice de 12×13 blocs de traitement. Chaque bloc contient un processeur CoolRisc, une mémoire Rom pour le code de différentes bibliothèques propres à FPPA, une mémoire Ram pour le code utilisateur et une mémoire Ram pour les données utilisateur. Le processeur présent dans chaque bloc est un CoolRisc 816 développé par le CSEM avec des données sur 8 bits et 16 registres internes. La mémoire Rom de 2048 instructions de code contient les bibliothèques de configuration et diagnostic du circuit ainsi qu'une bibliothèque de fonctions mathématiques. La mémoire Ram de 512 instructions pour le code utilisateur est protégée contre une erreur simple par un code correcteur linéaire de Hamming. Chaque instruction sur 22 bits est stockée avec les bits de redondance sur 27 bits ($22+\lceil\log_2 22\rceil$). La mémoire Ram de données a une taille de 512 octets.

Les blocs de processeurs et mémoires sont reliés par un système de communication asynchrone suivant une topologie en grille. Le système de communication est asynchrone car il n'est pas possible de distribuer une horloge commune à tous les blocs du circuit assez rapidement. Chaque bloc, synchrone, est cadencé par une horloge locale. Les communications se font à l'aide d'un protocole requête et acquittement implanté en matériel. La grille de communication est reconfigurable pour tolérer certains défauts. La phase de diagnostic du circuit permet de déterminer si certains blocs du circuit ont des défauts. La reconfiguration du système va invalider la ligne et la colonne de chaque bloc qui présente un défaut. La taille initiale, de 12×13 blocs, a été calculée en prenant en compte le taux de défaut de la technologie et la taille des blocs afin que le système soit au moins de 10×10 processeurs après reconfiguration dans plus de 95 % des cas. Un mécanisme de routage intégré permet de router automatiquement des paquets de 8 bits à travers la grille. Le routage est aussi reconfigurable. À chaque index, ou canal, de routage, correspond une table de routage qui précise à quels voisins (nord, sud, est ou ouest) doivent être envoyés les paquets. Les index sont stockés sur 3 bits, soit 8 canaux de routage. Différentes topologies sont envisageables pour chaque index : point à point, diffusion selon les axes totale ou limitée, anneau...

L'un des buts de ce projet était de proposer des architectures à très faible consommation d'énergie en utilisant le parallélisme et la diminution de la tension d'alimentation V_{DD}. La consommation dynamique d'un circuit CMOS (*complementary metal oxide semi-conductor*) peut être modélisée par l'équation suivante [128, 84]

$$P_{dyn} = \alpha \times C \times f \times V_{DD}^2 \tag{2.1}$$

1. *Analog to digital converter (ADC).*
2. *Digital to analog converter (DAC).*

où α est le taux d'activité moyen du circuit, C la capacité totale commutée en moyenne et f la fréquence. Les courants de fuite n'étaient pas encore trop importants pour les technologies de l'époque (0.35 ou 0.25 μm). Réduire V_{DD} et f permet donc de diminuer significativement la puissance dynamique du circuit [128]. Le parallélisme était utilisé pour augmenter les performances perdues du fait de la réduction de fréquence.

Le *layout* complet du circuit a été réalisé, mais il n'a pas été fondu faute de budget. Le circuit représente 4 cm^2 en technologie 0.35 μm soit la surface du réticule complet disponible dans cette technologie (ce qui en aurait fait un très gros circuit pour cette époque). Le circuit présente une consommation de moins de 1 W à pleine vitesse à 20 MHz sous 3 V pour une performance maximum de 500 MIPS (millions d'instructions par seconde). Une version simplifiée avec 2×2 blocs a été fondue et testée pour valider le principe.

J'ai participé à ce projet avec Bernard Girau, Pascal Nussbaum, Pierre Marchal, Christian Piguet et Fabio Restrepo. J'ai réalisé la bibliothèque de calcul présente dans la ROM des blocs de FPPA (description en section 2.7.1). Nous travaillé dans le cadre de l'encadrement de la thèse de Fabio Restrepo [122] sur des outils pour programmer la machine FPPA. Un logiciel a été développé durant cette thèse. Ma participation à ce développement était plus limitée (20 % environ). La description d'entrée était un ensemble de machines d'états (manipulées graphiquement dans l'outil Renoir de Mentor Graphics). Le logiciel produisait les tables et index de routage, les codes d'échanges de données et les codes utilisateurs, la distribution de processus et la gestion de la configuration du circuit. Les résultats de ces travaux ont été publiés dans les conférences ICES en 1998 [47] et MicroNeuro en 1999 [31].

Aujourd'hui, ce genre d'architecture intégrant de nombreux cœurs sur une même puce est courant, mais à l'époque, 1997–1999, nous étions au tout début de ce genre de circuits. Cette expérience était passionnante.

Récapitulatif	
Encadrement :	Fabio Restrepo, doctorat 2001 [122], 20 %
Conférences :	ICES 1998 [47], MicroNeuro 1999 [31]
Collaboration :	EPFL 1997–1999
Logiciels :	bibliothèque de calcul (flottant + autres) pour processeur CoolRisc outils spécifiques pour FPPA

2.2.2. Architecture FPOP

Avec Martin Dimmler, nous avons montré que l'arithmétique en ligne sur des FPGA permet d'obtenir des solutions plus rapides et plus économes en énergie qu'avec une arithmétique parallèle pour certaines applications de contrôle numérique (cf. section 2.1). Suite à ces travaux, avec Pierre Marchal et Christian Piguet, nous avons étudié et proposé une architecture de circuit reconfigurable dédiée à l'arithmétique en ligne. Cette architecture a été nommée FPOP pour *field programmable on-line operator*. Le but de ce projet était de fournir des solutions de contrôle numérique et de traitement du signal monocircuit et à faible consommation d'énergie.

L'architecture de FPOP est similaire à un FPGA mais à grain moyen. Elle est composée d'un ensemble de cellules reconfigurables pour les entrées–sorties et les calculs en arithmétique en ligne ainsi qu'un système de connexions reconfigurable entre les cellules. L'architecture est présentée à la figure 2.2. Les différents blocs de FPOP sont :

- les blocs d'entrée–sortie :
  - conversion analogique vers numérique (A/D) pour les entrées ;
  - conversion numérique vers analogique (D/A) pour les sorties ;

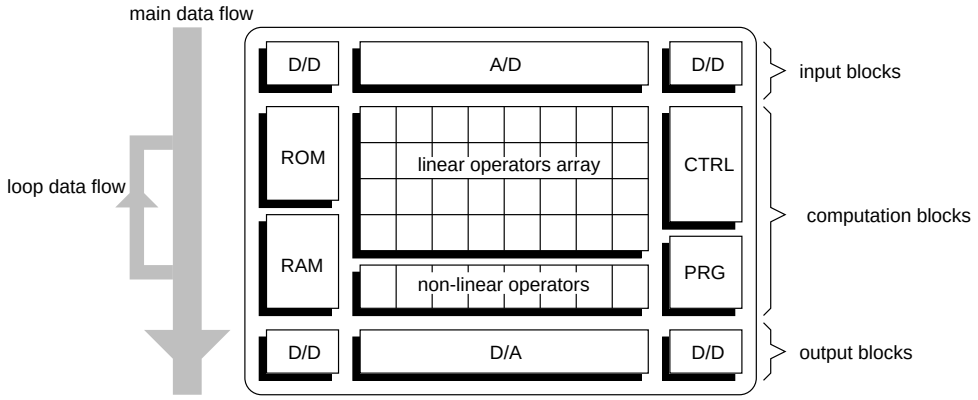


FIGURE 2.2.: Architecture de FPOP.

  - conversion numérique vers numérique (D/D) [3] pour les entrées et les sorties ;
- les blocs de calcul reconfigurables :
  - matrice de blocs pour des opérations linéaires $(\pm, \times_{cst})$;
  - vecteur de blocs pour certaines opérations non linéaires ;
- les mémoires :
  - mémoire ROM pour les paramètres fixes ;
  - mémoire RAM pour les paramètres applicatifs ;
- le bloc de contrôle (CTRL) pour le séquencement des opérations et les structures de contrôle (test, boucle) ;
- le bloc de configuration (PRG).

La représentation des nombres choisie était le *borrow-save* : base 2 avec les chiffres $\{-1, 0, 1\}$ (cf. section 1.1.1). Ce choix venait du fait que l'on avait certains blocs de calcul, des ADC et des DAC qui fournissaient ou utilisaient directement des données en représentation *borrow-save*. Un essai avec des opérateurs d'addition et des portions de multiplication par une constante en ligne dans les bases 4, 5 et 8 avait été intégré en technologie SOI [4] 1 μm. Lors de cet essai, différents codages des chiffres ont été testés : complément à deux, signe magnitude, 1 parmi n. Les résultats de mesure du circuit étaient intéressants en particulier pour la basse consommation d'énergie. Mais refaire tous les blocs, en particulier les ADC et DAC, en plus grande base n'était pas possible dans le temps du projet. Nous avons donc conservé la base 2 et le *borrow-save*.

Nous avons regardé quels étaient les types d'opérations présents dans les applications d'automatique visées (p. ex. des contrôleurs numériques) et leurs fréquences d'utilisation. On avait :

- beaucoup d'additions (de 2 à 5 termes) et de multiplications par des constantes (mais qui doivent être configurables) ;
- quelques multiplications de 2 variables ;
- un tout petit peu de divisions, de racines carrées et des approximations de fonctions (p. ex. $\cos, \sin, \exp, \log, \arctan$) ;

Nous avons donc choisi d'implanter un grand nombre de cellules reconfigurables pour l'addition et la multiplication par des constantes. Ces cellules sont regroupées dans un matrice optimisée pour les opérations linéaires de type $\sum a_i x_i$ où les a_i sont des constantes et les x_i des variables. L'opération de multiplication de deux variables étant moins fréquente que les multiplications

3. Conversions entre les représentations redondante et binaire classique (dans les deux sens), adaptation du format de tension, interface parallèle 8 bits pour des communications avec un microcontrôleur.

4. Technologie *silicon on insulator* (SOI) choisie pour sa faible consommation et sa résistance aux rayonnements dans les applications aérospatiales.

par des constantes, nous n'avons pas implanté de multiplieurs. Une configuration spécifique de deux cellules linéaires adjacentes (selon l'axe des abscisses sur la figure 2.2) permet de faire une multiplication de deux variables. En effet, le partage des ressources de deux multiplieurs par une constante permet d'implanter simplement et efficacement un multiplieur. Pour les opérations beaucoup plus rares, nous avons choisi d'implanter une rangée d'opérateurs dédiés soit à la division, la racine carrée ou l'approximation de polynômes. C'est la rangée de cellules d'opérations non linéaires en bas de l'architecture de FPOP présentée en figure 2.2.

La figure 2.3 décrit le contenu des cellules linéaires (permettant aussi la multiplication de deux variables). Les cellules sont configurables à plusieurs niveaux : taille des données, intervalle des entrées, constantes, délais additionnels, etc. L'intérêt de l'arithmétique en ligne est permettre la mise en œuvre d'opérateurs de petite taille. Le caractère sériel de l'arithmétique en ligne est aussi très intéressant pour simplifier le réseau de routage programmable. On a bien une architecture reconfigurable à grain moyen dont l'efficacité énergétique est bien meilleure que pour un FPGA qui nécessité de très grandes quantités de bits de configuration. Bien entendu, FPOP est bien moins flexible qu'un FPGA. Il n'est fait que pour certaines applications.

Pour les cellules d'opérations non linéaires, nous avons utilisé les algorithmes classiques de division et de racine carrée en ligne (cf. [83, chap. 9]). Pour l'approximation de fonctions, nous avons réalisé des cellules, illustrées en figure 2.4, permettant d'évaluer des polynômes d'un degré maximum de 7. Ce degré maximum correspondait au besoin de précision pour les applications visées. Les approximations polynomiales utilisées à cette époque étaient bien moins performantes que ce que l'on sait faire aujourd'hui avec des techniques comme celles présentées au chapitre 3.

Le contrôle dans le circuit FPOP est totalement statique et déterminé lors de la configuration. Nous utilisons des lignes de validité des données. On trouve, un peu partout dans FPOP, des cellules de délais additionnels programmables. Ces délais sont utiles pour synchroniser les entrées de certains opérandes.

Nous avons aussi travaillé sur des outils de programmation pour FPOP. Le flot de programmation est illustré en figure 2.5. La programmation se faisait à partir d'un fichier source en C ou Matlab (très utilisé en automatique). L'utilisateur devait extraire les parties devant être implantées dans FPOP : les opérations de calcul et les structures de contrôle simples nécessaires au contrôle de ces calculs (test et boucle). Ces fragments du code source étaient lus par un analyseur grammatical du langage. Le sous-ensemble des langages C et Matlab reconnu était extrêmement simple. La phase d'analyse fournissait les informations suivantes :

- la liste de tous les signaux internes avec leur nom, taille et précision (si annotée par l'utilisateur) ;
- la liste de toutes les opérations ;
- la liste les structures de contrôle reconnues par notre outil (certains tests et boucles).

À partir de ces informations, une allocation des ressources de FPOP était déterminée en utilisant un mécanisme d'affectation des opérations aux cellules par priorité. Les informations sur les signaux permettaient de configurer le routage entre les différentes cellules et blocs du circuit. Une phase d'optimisation était faite pour réduire le délai en ligne et nécessitait de modifier la configuration des éléments de délai programmable des cellules. La programmation de FPOP était très simple du fait de la grande similarité entre les équations (entrées en Matlab ou en C) et la structure des cellules. Les ressources de routage étaient dimensionnées afin de pouvoir implanter directement les principaux schémas de calcul sans ordonnancement. Le seul ordonnancement nécessaire était pour la gestion de la synchronisation des flots de données dans les boucles. Le problème de la précision des opérandes était totalement à la charge du programmeur. Aujourd'hui, il serait possible d'automatiser en partie cette prise en charge de la précision des calculs.

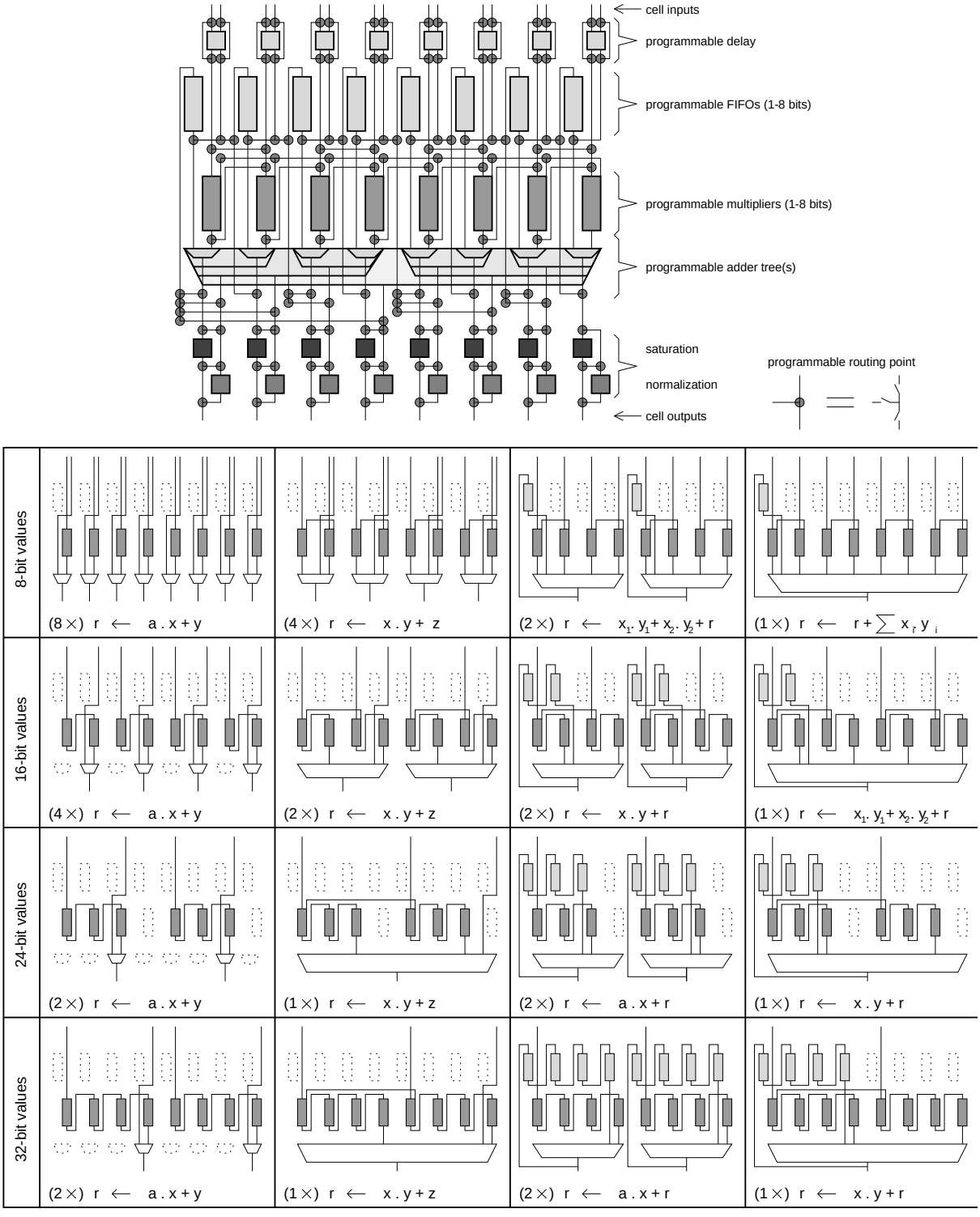


FIGURE 2.3.: Cellule linéaire et de multiplication de FPOP et quelques configurations possibles.

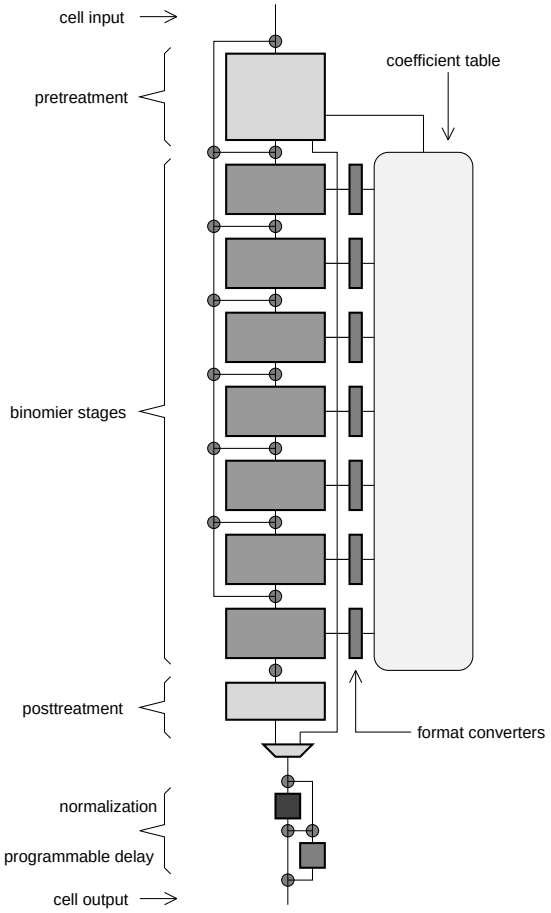


FIGURE 2.4.: Cellule d'approximation polynomiale de FPOP.

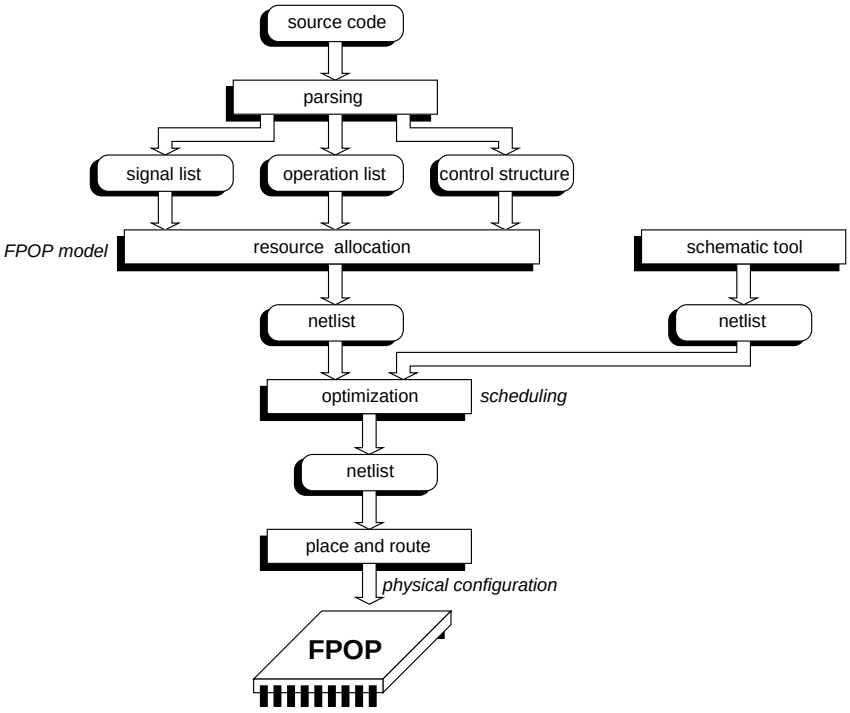


FIGURE 2.5.: Flot de programmation de FPOP.

L'architecture FPOP a été brevetée en janvier 1999 [37] et ensuite publiée dans les conférences FPL 1999 [65] et SPIE 1999 [64].

Depuis cette époque, il existe des structures efficaces d'ADC et de DAC qui travaillent avec un codage redondant en base 4 et des chiffres dans $\{-2, -1, 0, 1, 2\}$. Nous espérons pouvoir retravailler sur ce genre d'architecture dans le futur.

RÉCAPITULATIF	
	BREVET : CSEM [37]
	CONFÉRENCES : SPIE 1999 [64], FPL 1999 [65]
	LOGICIELS : outils de programmation spécifiques à FPOP

2.3. Méthodes à base de tables

L'approximation des fonctions élémentaires en matériel est une thématique importante en arithmétique des ordinateurs, en particulier en France, depuis une vingtaine d'années. Un livre complet [113] est dédié aux méthodes d'évaluation des fonctions élémentaires en logiciel et en matériel. Évaluer efficacement ces fonctions est important pour de plus en plus d'applications dans des systèmes embarqués, en multimédia, en traitement du signal, etc.

Le type de méthode à utiliser dépend beaucoup de la précision cible et de considérations au niveau de l'implantation. Les méthodes à base de tables décrites ci-dessous sont utilisables assez efficacement jusqu'à une précision[5] d'une vingtaine de bits. Jusqu'au début des années 90, seules des méthodes à base de récurrence de chiffres[6] comme CORDIC [113] étaient implantées en matériel. En logiciel, on utilisait essentiellement des approximations polynomiales

5. Pour nous, la précision est le nombre de bits significatifs μ correspondant à une erreur maximale ϵ selon la relation $\mu = -\log_2 |\epsilon|$. Par exemple, une erreur de 2^{-16} correspond à une précision de 16 bits.

6. Aussi appelées méthodes à additions et décalages dans la littérature.

ou rationnelles. Le chapitre 3 portera sur l'évaluation de fonctions par approximation polynomiale en matériel. Avec l'arrivée de FPGA de grande taille et l'augmentation de la densité d'intégration en circuit ASIC, d'autres méthodes d'évaluation sont devenues intéressantes pour l'approximation des fonctions élémentaires en matériel. En particulier, les méthodes à base de tables sont devenues une alternative possible aux approximations polynomiales et rationnelles et aux algorithmes comme CORDIC.

Bien évidemment, tabuler directement les valeurs d'une fonction $f(x)$ pour chacun des arguments possibles x conduit à une table avec 2^n valeurs si x est représenté sur n bits. En pratique, la tabulation directe, c.-à-d. avec une seule table, est limitée à des arguments d'une dizaine de bits. On trouve, dans de rares cas particuliers, des tables avec 12 ou 15 bits d'adresse. Dans les méthodes à base de tables évoluées, on utilise plusieurs tables de constantes précalculées ainsi que quelques opérations simples comme l'addition, la soustraction, les décalages et éventuellement des petites multiplications ou des multiplications rectangulaires. Une multiplication est qualifiée de *rectangulaire* lorsqu'un des deux opérandes est beaucoup plus petit que l'autre.

2.3.1. Méthode à base de tables et de petits multiplieurs

Lors d'une collaboration avec Milos Ercegovac (UCLA), Thomas Lang (UCI) et Jean-Michel Muller (CNRS–LIP), nous avons mis au point un algorithme permettant d'évaluer l'inverse, la racine carrée, la racine carrée inverse, l'exponentielle et le logarithme en utilisant des petites tables et des petits multiplieurs. Nous décrivons ici très rapidement le principe de la méthode proposée. Une description complète est présentée dans l'article [27] publié dans le journal *IEEE Transactions on Computers* en juillet 2000 et reproduit en annexe A.2 (page 105). Une version préliminaire de ce travail a été publiée à la conférence SPIE en 1998 [26].

Ci-dessous, nous utilisons les notations de [27] reproduit en annexe A.2. On cherche à approcher la fonction $g(Y)$ où Y est la mantisse d'un nombre flottant IEEE [69] sur m bits. On a donc $1 \leq Y < 2$. Le calcul de l'exposant étant très simple, il n'est pas pris en compte. Les calculs internes sont faits sur n bits avec $n > m$. La méthode utilise un développement en série de Taylor (cf. [113]) qui converge rapidement, c.-à-d. avec peu de termes, si l'argument est proche de 1. La méthode fonctionne en 3 étapes :

Étape 1 : réduction

On trouve A, proche de 1, à partir de Y tel que $-2^{-k} < A < 2^{-k}$ et k est fixé à $n/4$ (on coupera A en 4 parties égales pour la deuxième étape). Pour ceci, on utilise une approximation $\hat{R}$ de $1/Y$ avec $k+1$ bits de précision (par tabulation). On a $A = Y \times \hat{R} - 1$.

Étape 2 : évaluation

On utilise le développement de Taylor $f(A) = C_0 + C_1 A + C_2 A^2 + C_3 A^3 + C_4 A^4 + \ldots$. Comme $|A| < 2^{-k}$, on peut couper A en parties égales telles que $A = A_2 z^2 + A_3 z^3 + A_4 z^4 + \ldots$ avec $z = 2^{-k}$ et $\forall i \ |A_i| \leq 2^k - 1$. A est donc de la forme : $0.\boxed{0\ldots0}\boxed{A_2}\boxed{A_3}\boxed{A_4}\ldots$ En injectant le découpage de A dans le développement de Taylor de $f(A)$ et après suppression des termes plus petits que 2^{-4k}, on obtient l'approximation :

$$f(A) \approx C_0 + C_1 A + C_2 A_2^2 z^4 + 2 C_2 A_2 A_3 z^5 + C_3 A_2^3 z^6$$

Pour chacune des fonctions cibles on a un jeu de valeurs pour les C_i.

Étape 3 : post-traitement

Pour passer de $f(A)$ à $g(Y)$, il faut effectuer une multiplication par une valeur qui dépend de $\hat{R}$. Cette valeur est tabulée en même temps que $\hat{R}$.

Pour les différentes fonctions, on a une précision d'un peu moins de $4k$ bits. Le coût de calcul est dominé par un multiplieur de $(3k+1) \times (3k+2)$ bits et des tables de tailles $2^k \times (k+1)$ et $2^k \times (n+k+1)$ bits.

Comparativement aux méthodes à base de tables proposées ci-dessous, cette méthode est moins performante pour une seule fonction. Par contre, pour approcher les fonctions $1/Y$, $\sqrt{Y}$, $1/\sqrt{Y}$, $\exp Y$ et $\log Y$ dans le même circuit, elle est bien plus efficace, en vitesse et en surface, que la somme des blocs utilisant d'autres méthodes pour chaque fonction individuellement. Cette méthode donne d'assez bons résultats dans le cas de la simple précision IEEE ($m = 24$).

Récapitulatif	
	Journal : IEEE TC 2000 [27]
	Conférence : SPIE 1998 [26]
	Collaborations : UCLA et UCI (1997–1999)

2.3.2. Méthode à base de tables et additions, méthode multipartite

Ci-dessous, nous résumons très rapidement une méthode proposée avec Florent de Dinechin (ENS Lyon–LIP). Une description complète est présentée dans l'article [24] publié dans le journal *IEEE Transactions on Computers* en mars 2005 et reproduit en annexe A.3 (page 116). Une version préliminaire de ce travail a été publiée aux conférences SPIE en 2000 [22] (uniquement pour les fonctions sin et cos) et ARITH 15 en 2001 [23].

Ces méthodes utilisent des lectures dans plusieurs tables en parallèle et effectuent la somme (de valeurs algébriques) des contributions des différentes tables pour fournir l'approximation souhaitée. Le nombre de tables utilisées est variable suivant les méthodes.

- La méthode *bipartite* utilise deux tables comme illustré en figure 2.6. Cette méthode est utilisée depuis de nombreuses années. L'approximation effectuée est très proche d'un développement en série de Taylor à l'ordre 1 autour du point constitué des bits de poids forts de l'argument. On cherche l'approximation de $f(x)$ avec $x = x_1 + x_2 \times 2^{-k} + x_3 \times 2^{-2k}$ et $0 \leq x_i \leq 1 - 2^{-k}$ un multiple de 2^{-k}. L'argument x est donc découpé en 3 parties égales de k bits chacune : $\boxed{x_1 \mid x_2 \mid x_3}$. On utilise un développement de Taylor d'ordre 1 au point $x_1 + x_2 \times 2^{-k}$ qui est $f(x) = f(x_1 + x_2 2^{-k}) + x_3 2^{-2k} f'(x_1 + x_2 2^{-k}) + \epsilon_1$. On approche $f'(x_1 + x_2 2^{-k})$ par un développement de Taylor en x_1 qui est $f'(x_1 + x_2 2^{-k}) = f'(x_1) + x_2 2^{-k} f''(x_1) + \epsilon_2$. Finalement on a :

$$f(x) = f(x_1 + x_2 2^{-k}) + x_3 2^{-2k} f'(x_1) + \epsilon$$

  Globalement, la méthode bipartite nécessite : une table $T_1(x_1, x_2) \approx f(x_1 + x_2 2^{-k})$ de 2^{2k} mots de n bits, une table $T_2(x_1, x_3) \approx x_3 2^{-2k} f'(x_1)$ de 2^{2k} mots de p bits avec $p << n$ et une addition finale. Cette méthode permet une compression des tables d'une taille totale de seulement $(n+p) \times 2^{\frac{2n}{3}}$ bits contre $n \times 2^n$ bits dans le cas d'une tabulation directe.
  Nous avons proposé une petite amélioration dans [22] où x n'est pas découpé en parties de même taille. Nous avons implanté cette amélioration dans un générateur automatique de code VHDL pour les fonctions sinus et cosinus. Nous exploitons aussi la structure de cascades de LUT des FPGA pour optimiser le découpage.
- La méthode *multipartite* utilise $m+1$ tables comme illustré en figure 2.7 dans le cas tripartite avec $m = 2$. Différentes variantes ont été proposées pour la méthode multipartite. Mais toutes ces méthodes utilisaient un découpage en blocs de même taille pour simplifier le problème. Nous avons proposé dans [23] puis [24] un algorithme de recherche exhaustif pour parcourir toutes les combinaisons possibles des découpages. Grâce à une formulation

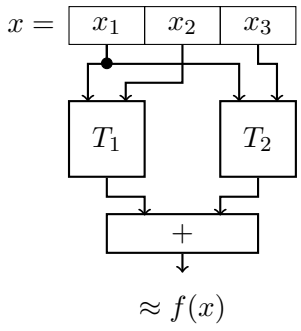


FIGURE 2.6.: Architecture de calcul pour la méthode bipartite.

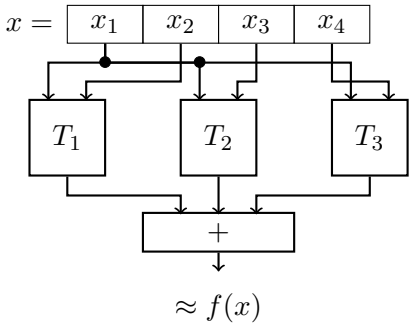


FIGURE 2.7.: Architecture de calcul pour la méthode multipartite (ici tripartite $m = 2$).

astucieuse des contributions de l'erreur d'approximation, nous pouvons élaguer l'arbre des découpages au fur et mesure de son parcours. Pour un niveau de l'arbre des découpages, on sait évaluer l'erreur d'approximation. Si cette erreur dépasse l'erreur maximale admise, il n'est pas utile de parcourir le sous-arbre. En pratique, l'accélération due à cet élagage est très importante. Notre programme de recherche parcourt l'espace des découpages possibles et utiles en quelques dizaines de secondes pour des précisions cibles de 24 bits. L'explication de la méthode est présentée dans [24] reproduit en annexe A.3. Étant donnés un nombre de tables et une erreur maximale admissible, notre méthode calcule le découpage optimal conduisant aux plus petites tailles de tables. La méthode multipartite donne de très bons résultats pour des approximations avec des précisions cibles jusqu'à une vingtaine de bits. Pour des précisions plus importantes, il faut se tourner vers des méthodes comme celles présentées au chapitre 3.

Les méthodes à base de tables sans multiplication, même petite ou rectangulaire, sont de moins en moins intéressantes sur les toutes dernières générations de circuits FPGA. Les blocs de multiplication câblée intégrés dans ces FPGA rendent les solutions à base de multiplication plus performantes que celles basées seulement sur des tables, des additions et des décalages. Par exemple, dans un circuit Xilinx Virtex-6, on trouve, suivant la taille du FPGA, entre 288 et 2 016 blocs qui intègrent tous un multiplieur 25×18 bits et un accumulateur sur 48 bits qui fonctionnent à plusieurs centaines de MHz. Dans ces conditions, évaluer une fonction avec une précision entre 20 et 32 bits de précision en utilisant des approximations polynomiales (cf. chapitre 3) ne coûte que quelques multiplieurs.

Dans un avenir proche, nous souhaitons pouvoir évaluer les caractéristiques énergétiques des méthodes à base de tables pour des implantations ASIC. Ces méthodes conduisent à des circuits

avec une faible activité par rapport à des solutions itératives. Mais si la surface de silicium nécessaire pour les tables est trop importante, la consommation statique du circuit due aux courants de fuite (cf. [84]) risque de devenir très importante dans les dernières technologies.

Récapitulatif	
	Journal : IEEE TC 2005 [24]
	Conférences : SPIE 2000 [22], ARITH 2001 [23]
	Logiciel : générateur de code VHDL

2.3.3. Méthode à base de tables et polynômes très particuliers

Lors d'une collaboration avec Milos Ercegovac (UCLA) et Jean-Michel Muller (CNRS-LIP), nous avons proposé une méthode d'évaluation de l'inverse et de la racine carrée inverse pour des faibles précisions (7 à 10 bits). La méthode utilise une petite table et des polynômes très particuliers pour lesquels il n'y a aucune multiplication à effectuer. Ce travail a été publié à la conférence Asilomar en 2005 [28]. Un générateur automatique de code VHDL optimisé a été réalisé, il est disponible sur ma page web [7]. Ce générateur, nommé `seedgen`, est écrit en C et il est disponible sous licence GPL.

L'approximation de $1/x$ et de $1/\sqrt{x}$ est particulièrement utile pour l'initialisation des méthodes itératives utilisées dans le calcul de la division et de la racine carrée en logiciel. Dans la plupart des processeurs à hautes performances actuels, la division et la racine carrée sont calculées, en utilisant les unités flottantes, par une méthode proche de celle de Newton-Raphson (cf. [83, chap. 7]). Pour le calcul de $q = a/d$, on utilise une évaluation de $t = 1/d$ avec Newton-Raphson via l'itération $x_{i+1} = x_i(2 - dx_i^2)$, c.-à-d. x_i converge vers t, puis en effectuant le produit $q = t \times a$. La méthode de Newton-Raphson permet de doubler le nombre de chiffres significatifs à chaque itération. Afin d'accélérer le processus, les premières itérations qui fournissent peu de précision, sont généralement remplacées par une lecture dans une table d'une approximation x_0 du quotient $1/d$. Cette table d'inverses approchés est implantée en matériel. Par exemple, sur le processeur Itanium, l'instruction `frcpa` fournit un germe (*seed* en anglais) de l'inverse $1/d$ avec une précision de 8.886 bits (cf. [103]).

Dans le cas de racine carrée $\sqrt{c}$, on utilise l'itération de Newton-Raphson $x_{i+1} = \frac{x_i}{2}(3 - cx_i^2)$, qui converge vers $1/\sqrt{c}$, puis une multiplication par c pour obtenir $\sqrt{c}$. Utiliser une itération de Newton-Raphson qui converge directement vers $\sqrt{c}$ est une mauvaise idée. L'itération correspondante fait intervenir une division à chaque étape. Dans le cas de racine carrée, il faut donc tabuler une approximation de $1/\sqrt{c}$ pour supprimer les premières itérations qui donnent peu de précision. Le processeur Itanium a une instruction `frsqrta` qui fournit un germe de $1/\sqrt{c}$ avec une précision de 8.831 bits (cf. [103]).

La meilleure approximation polynomiale de $f(x) = 1/x$ et de degré 1 avec $x \in [1, 2[$ est donnée par le polynôme minimax : $1.4571 - 0.5x$ pour une précision de 4.5 bits (cf. chapitre 3 pour une description rapide et des références sur l'utilisation des polynômes d'approximation minimax). Différents chercheurs avaient proposé d'utiliser le polynôme $p(x) = \frac{3}{2} - \frac{x}{2}$ proche de $1.4571 - 0.5x$. Le polynôme $p(x)$ est particulièrement simple à évaluer comme on le constate sur la figure 2.8. La division par 2 est un simple décalage à droite, $-x/2$ est obtenu, en complément à deux, en inversant tous les bits de x et en ajoutant 1 au niveau du LSB comme retenue entrante d'un additionneur très simple (qui ne fait que propager la retenue). L'ajout de 3/2 ne coûte rien comme on le voit en figure 2.8. Au final, l'évaluation de $p(x)$ ne nécessite que quelques inverseurs et un additionneur pour faire la propagation de la retenue entrante injectée au LSB. Mais la

7. `http://www.irisa.fr/prive/Arnaud.Tisserand/devel/seedgen/`

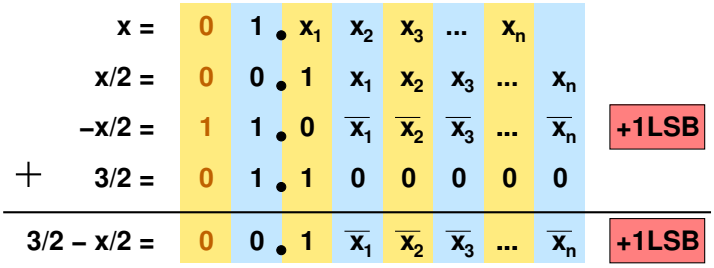


FIGURE 2.8.: Évaluation à très faible coût du polynôme $p(x) = \frac{3}{2} - \frac{x}{2}$.

précision de $p(x)$ pour approcher $1/x$ est de seulement 3.5 bits, ce qui est loin de la dizaine de bits pour la précision visée.

Nous avons proposé d'ajouter une petite table pour compenser les erreurs dues à l'utilisation de $p(x)$ seul. On a donc $s(x) = p(x) + t(x) = \frac{3}{2} - \frac{x}{2} + t(x)$ où $t(x)$ est la différence tabulée entre $1/x$ et $p(x)$. L'architecture correspondante est présentée en figure 2.9. Afin d'améliorer un peu la précision, le calcul $p(x) + t(x)$ est fait sur $n + g$ bits où n est le nombre de bits de x et g le nombre de bits de garde ajoutés. La table $t(x)$ est assez petite. Elle a 2^n bits d'adresse et des mots de w bits. w est petit car il faut juste stocker la différence entre $1/x$ et $p(x)$. Le nombre de bits d'adresse n est entre 7 et 10 pour nos précisions cibles.

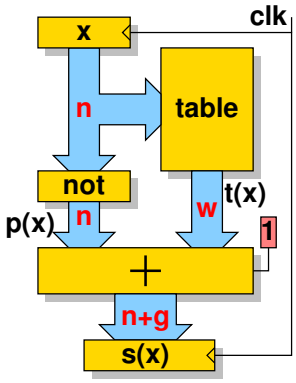


FIGURE 2.9.: Architecture pour l'évaluation de $p(x) + t(x)$.

La figure 2.10 illustre les différentes valeurs dans le cas, très simple, où x est représenté sur $n = 3$ bits et avec $g = 1$ bit de garde. On vérifie bien sur ces courbes que la différence entre $1/x$ et $p(x)$ est faible, et donc w petit. La figure 2.11 illustre l'impact du nombre de bits de garde g sur les erreurs d'approximation dans un cas particulier.

Dans le cas de la racine carrée inverse, nous avons trouvé une astuce similaire. Le polynôme minimax de degré 1 pour $f(x) = 1/\sqrt{x}$ avec $x \in [1, 2[$ est $1.2739 - 0.292x$. Il conduit à une précision de 5.7 bits. Ce polynôme est proche du polynôme $p(x) = \frac{5}{4} - \frac{x}{4}$ qui conduit à une précision de 4 bits. Ce polynôme $p(x)$ pour la racine carrée inverse est aussi très simple à évaluer comme on le constate sur la figure 2.12. L'architecture présentée en figure 2.9 peut être de nouveau utilisée avec de très légères modifications.

Cette méthode permet d'obtenir des circuits 2.5 fois plus petits et 40 % plus rapides qu'avec d'autres solutions pour $1/x$ et $1/\sqrt{x}$ approchées avec une précision de 7 à 10 bits. Dans le chapitre 3, nous verrons qu'il est maintenant possible de rechercher des polynômes avec des

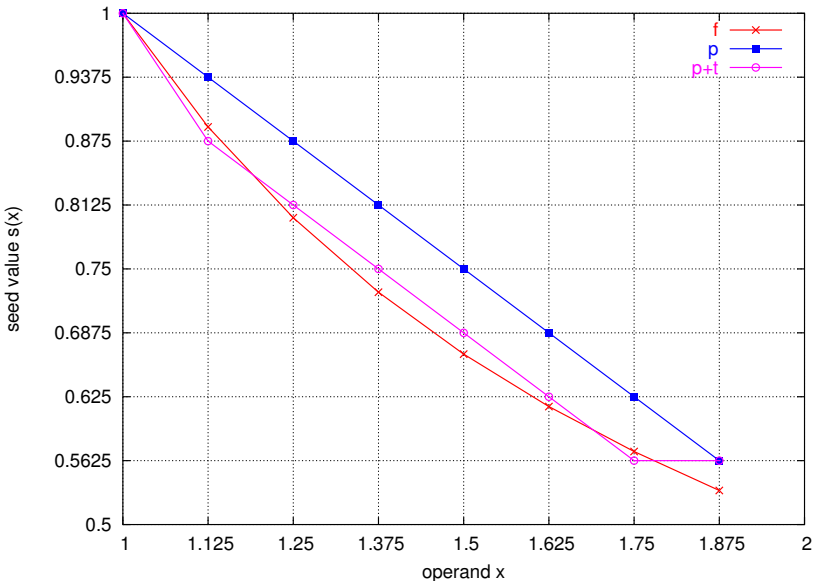


FIGURE 2.10.: Résultat de `seedgen` pour $f = 1/x$, $n = 3$ et $g = 1$.

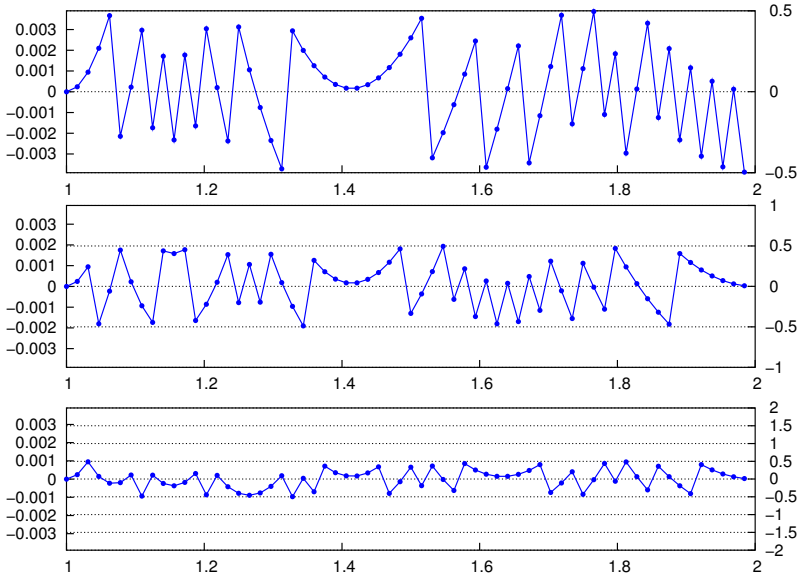


FIGURE 2.11.: Courbes d'erreur pour $f = 1/x$, $n = 6$ et différents nombres de bits de garde : $g = 1$ (haut), $g = 2$ (milieu) et $g = 3$ (bas).

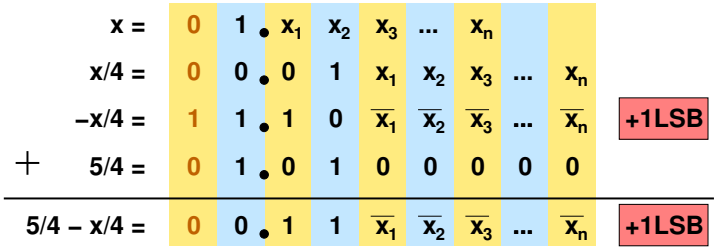


FIGURE 2.12.: Évaluation à très faible coût du polynôme $p(x) = \frac{5}{4} - \frac{x}{4}$.

écritures particulières des coefficients. Ce principe d'utilisation de polynômes très particuliers peut être probablement étendu à d'autres fonctions grâces à ces outils.

RÉCAPITULATIF

CONFÉRENCE : Asilomar 2005 [28]
COLLABORATION : UCLA (2004–2005)
LOGICIEL : générateur `seedgen`, licence GPL

2.4. Division pour circuits asynchrones

Les circuits asynchrones, ou auto-séquencés, permettent de s'affranchir des problèmes complexes de distribution d'horloge (cf. [125]). Ils présentent de nombreuses caractéristiques intéressantes : mise en veille naturellement simple et efficace, faible rayonnement électromagnétique, conception modulaire très simple, etc. Le temps de calcul d'un opérateur asynchrone dépend des valeurs des opérandes, il est donc variable. L'optimisation d'un opérateur arithmétique nécessite donc de raisonner en cas moyen et plus seulement en pire cas comme dans les circuits classiques avec horloge. Des algorithmes optimisés pour les opérations d'addition et de multiplication ont été proposés pour les circuits asynchrones. Mais pour la division, seules des solutions assez simples ont été étudiées.

Avec Nicolas Boullis, nous avons étudié l'impact d'une implantation en circuit asynchrone sur les performances de divers algorithmes de division par récurrence de chiffres [83, chap. 5]. Plusieurs études ont été faites pour des algorithmes SRT en base 2 avec les chiffres $\{-1, 0, 1\}$ et un reste partiel représenté en *carry-save* ou en *borrow-save*. Le choix du codage redondant du reste partiel est totalement justifié en synchrone. Mais est-ce le cas en asynchrone ? Nous avons évalué et comparé les algorithmes suivants :

- division restaurante en base 2, reste partiel non redondant ;
- division SRT base 2, chiffres du quotient dans $\{-1, 0, 1\}$, reste partiel non redondant ;
- division SRT base 4, chiffres du quotient dans $\{-2, \cdots, 2\}$, reste partiel non redondant ;
- division SRT base 8, chiffres du quotient dans $\{-6, \cdots, 6\}$, reste partiel non redondant ;
- division SRT base 16, chiffres du quotient dans $\{-10, \cdots, 10\}$, reste partiel non redondant ;
- division SRT base 2, chiffres du quotient dans $\{-1, 0, 1\}$, reste partiel redondant ;
- division SRT base 4, chiffres du quotient dans $\{-2, \cdots, 2\}$, reste partiel redondant.

Nous avons développé, en C++, un simulateur au niveau porte logique asynchrone avec prise en compte des délais des portes. Il n'existait pas de simulateur pour circuit asynchrone accessible à cette époque. Utiliser un simulateur VHDL avec gestion des délais ne permettait pas de faire des statistiques assez rapidement. Nous avons utilisé un modèle de circuit asynchrone quasiment insensible aux délais (cf. [125]). Notre simulateur prend en entrée une description au niveau porte, ou *netlist*, de l'opérateur à simuler et un vecteur de valeurs des opérandes. La *netlist*

d'entrée est annotée avec les caractéristiques temporelles des portes. Le simulateur alors effectue des simulations logiques avec prise en compte des délais pour chaque élément du vecteur des opérandes. Des statistiques mesurées par le simulateur permettaient d'évaluer la distribution du temps de calcul et la distribution de l'activité [8] dans l'opérateur. La surface de circuit était estimée par la surface cumulée des cellules de la *netlist*.

Les résultats de cette étude ont été publiés dans la conférence SPIE en 2001 [9]. Ils montrent que la représentation redondante du reste partiel n'est pas du tout efficace pour la division asynchrone. La propriété d'addition (ou soustraction) en temps constant n'apporte rien dans ce cas. Un additionneur même très simple a un temps de calcul moyen très faible. L'absence d'accès à des outils de conception de circuit asynchrone ne nous a pas permis d'aller plus loin sur cette étude.

Avec Frédéric Robin et Pascal Vivet de STMicroelectronics, nous avions commencé une collaboration sur la conception d'unités de calcul asynchrones. En particulier, nous avions étudié un multiplieur-accumulateur asynchrone optimisé pour la basse consommation. Bien malheureusement, cette collaboration s'est arrêtée après quelques mois suite à un changement d'objectifs chez STMicroelectronics.

Récapitulatif	
	Encadrement : Nicolas Boullis, DEA 2001, 100 %
	Conférence : SPIE 2001 [9]
	Collaboration : STMicroelectronics 2001 (seulement quelques mois)

2.5. Opérateurs arithmétiques spécifiques pour FPGA

Avec Jean-Luc Beuchat, nous avons proposé des opérateurs arithmétiques qui essayent d'utiliser au mieux les structures spécifiques de certains circuits FPGA. Nous avons travaillé sur la multiplication, la division, l'arithmétique modulaire avec des moduli particuliers et l'arithmétique série. Le résultat de ce travail a été publié dans le chapitre « Opérateurs arithmétiques pour FPGA » [8] du livre « Arithmétique des ordinateurs » en 2004 (traité I2C, Hermes, Lavoisier).

Dans l'article [6] publié à la conférence FPL en 2002, nous décrivons des méthodes de multiplication et de division qui utilisent les blocs de multiplication câblée qui sont intégrés dans les FPGA depuis une dizaine d'années. Dans les FPGA Virtex-II de Xilinx, il y a quelques dizaines de blocs dédiés pour la multiplication 18×18 bits. Une décomposition de la multiplication $n \times n$ peut se faire en utilisant un découpage des opérandes en deux avec $n = 2m$ et $XY = (X_1k + X_0)(Y_1k + Y_0) = X_1Y_1k^2 + (X_1Y_0 + X_0Y_1)k + X_0Y_0$ où $k = 2^m$. La méthode proposée par Karatsuba et Ofman [94] utilise moins de sous-multiplications avec $(X_1k + X_0)(Y_1k + Y_0) = X_1Y_1(k^2 - k) + (X_1 + X_0)(Y_1 + Y_0)k + X_0Y_0(1 - k)$. Seules 3 multiplications $m \times m$ bits sont nécessaires contre 4 avec la décomposition triviale. Dans [97], Knuth propose une modification qui permet d'éviter le problème de la croissance de la taille de certains termes intermédiaires lors d'une décomposition récursive : $(X_1k + X_0)(Y_1k + Y_0) = X_1Y_1(k^2 + k) - (X_1 - X_0)(Y_1 - Y_0)k + X_0Y_0(1 + k)$. Nous avons développé un générateur de code VHDL qui fournit une description optimisée en fonction des blocs câblés disponibles et des tailles des opérandes. Nous avons essayé d'utiliser les blocs multiplieurs avec des combinaisons d'un peu de logique pour les décompositions qui ne sont pas des multiples de 18 bits. Par exemple, pour une multiplication flottante en simple précision IEEE [69], il faut un multiplieur 23×23 bits. Dans ce cas, nous utilisons un seul bloc et un peu de logique pour le « reste » de la multiplication afin de gagner un peu en ressources. Dans les dernières générations de FPGA,

8. Changements d'état qui reflètent la consommation dynamique du circuit (facteur α dans l'équation 2.1).

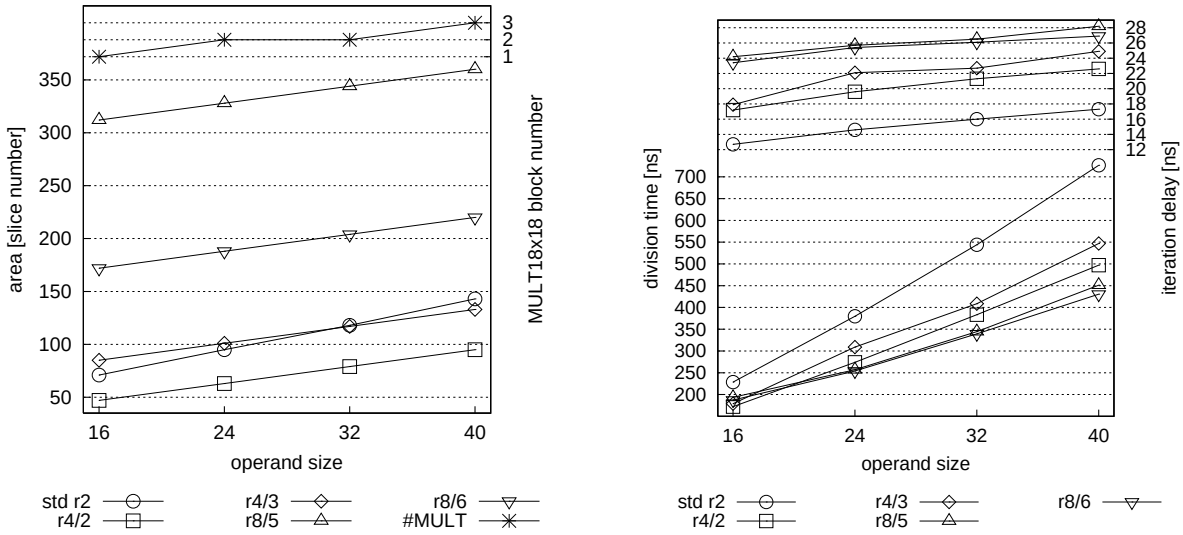


FIGURE 2.13.: Résultats d'implantation sur FPGA Virtex-II de diviseurs SRT pour différentes bases (β) et ensembles de chiffres du quotient ($\{-\alpha, \ldots, \alpha\}$, notation $\mathtt{r}\beta/\alpha$).

il y a des centaines de blocs avec des multiplieurs câblés. Ce genre d'optimisation n'a peut-être plus d'intérêt aujourd'hui. Pour la multiplication sur des Virtex-II, on obtient des gains vitesse entre 10 et 20 %.

Dans ce même article [6], nous présentons une méthode de division SRT (cf. [83, chap. 5]) en base moyenne qui utilise les blocs de multiplication 18×18 câblés. Lors de l'implantation de la méthode SRT, la base est limitée par la multiplication du nouveau chiffre du quotient et du diviseur. L'augmentation de la base permet de diminuer le nombre d'itérations à effectuer. En utilisant des blocs 18×18, on peut effectuer cette opération rapidement et avec un coût en surface raisonnable. Nous avons proposé des solutions en bases 4 et 8 avec différentes variantes pour la représentation redondante à chiffres signés des chiffres du quotient. Des gains en vitesse jusqu'à 40 % ont été obtenus pour les circuits FPGA Virtex-II. La figure 2.13 compare les temps de calcul et les surfaces des solutions proposées avec les solutions classiques.

Sur le développement de ces outils, c'est Jean-Luc qui a fait la plus grosse partie du travail sur la multiplication et moi celle sur la division.

RÉCAPITULATIF

CHAPITRE : traité I2C en 2004 [8]
CONFÉRENCE : FPL en 2002 [6]

Dans ce thème, il reste encore de nombreuses choses à faire. Les blocs DSP intégrés par centaines dans les FPGA actuels fournissent des multiplieurs asymétriques (p. ex. 25×18) et des accumulateurs (p. ex. 48 bits). Il y a aussi de nombreux blocs de mémoire pouvant être utilisés en ROM ou RAM. Ces blocs câblés changent considérablement le niveau de « granularité » auquel on était habitué en arithmétique matérielle.

2.6. Variantes de la multiplication

Nous avons travaillé sur différentes variantes de la multiplication. En particulier, la multiplication par des constantes est un problème important en pratique sur lequel nous travaillons

depuis plusieurs années.

2.6.1. Multiplication par des constantes

Ci-dessous, nous résumons rapidement un travail commencé avec Nicolas Boullis jusqu'à l'abandon de sa thèse en quatrième année. Il a écrit la première version du programme d'optimisation. Depuis, je poursuis ce travail et améliore le programme d'optimisation. Une description assez complète est présentée dans l'article [12] publié dans le journal *IEEE Transactions on Computers* en octobre 2005 et reproduit en annexe A.4 (page 129). Des versions préliminaires de ce travail ont été publiées aux conférences Sympa en 2002 [10] et ARITH 16 en 2003 [11].

De nombreux calculs courants peuvent s'écrire sous forme de multiplications par des constantes et d'additions. Ceci est particulièrement vrai dans de nombreuses applications de traitement du signal ou multimédia. Par exemple, c'est le cas des transformations comme la transformée de Fourier ou la transformée en cosinus, des filtres, etc. Mathématiquement, ces opérations peuvent s'écrire sous la forme d'une multiplication d'un vecteur d'entrée par une matrice constante.

Pour des implantations matérielles, il peut être avantageux d'implanter des multiplications par des constantes par uniquement des additions ou soustractions et des décalages. Ceci est d'autant plus vrai lorsque l'on peut combiner certains calculs, pour partager au maximum des sous-expressions communes. Le problème de la multiplication par une constante a été étudié depuis longtemps. On peut utiliser, par exemple, le célèbre recodage de Booth [74]. Il existe de nombreux algorithmes pour la multiplication par une constante, ou des problèmes proches comme l'élévation à une puissance constante. En revanche, le problème de la multiplication par une matrice constante a été nettement moins étudié. Nous traitons le cas de coefficients entiers ou en virgule fixe mais où tous les calculs sont exacts (sans aucune approximation).

La notation $x \ll k$ désigne le nombre x décalé de k bits vers la gauche, c'est-à-dire multiplié par 2^k, tous les nombres sont représentés en base 2. Les opérations d'addition et de soustraction sont considérées de coûts équivalents. On utilise un exemple unique : le calcul de p comme produit de la variable d'entrée x par la constante $c = 111463 = (11011001101100111)_2$.

L'algorithme le plus simple utilise directement la décomposition de la constante en base 2. Ainsi, dans notre exemple, le calcul se fait en 11 additions : $111463x = x \ll 16 + x \ll 15 + x \ll 13 + x \ll 12 + x \ll 9 + x \ll 8 + x \ll 6 + x \ll 5 + x \ll 2 + x \ll 1 + x$. Une amélioration simple consiste à utiliser le recodage canonique de Booth de la constante. Il s'agit d'un recodage en base 2, avec l'ensemble de chiffres $\{\overline{1}, 0, 1\}$ qui minimise le nombre de chiffres non nul. Pour notre exemple, on a $111463 = (11011001101100111)_2 = (100\overline{1}0\overline{1}0100\overline{1}0\overline{1}0100\overline{1})_2$ et le calcul se fait alors en 8 additions : $111463x = x \ll 17 - x \ll 14 - x \ll 12 + x \ll 10 - x \ll 7 - x \ll 5 + x \ll 3 - x$.

L'algorithme de Bernstein [72] recherche, par une exploration d'arbre, un très bon recodage d'une constante, en utilisant des opérations élémentaires. Le coût de chacune de ces opérations peut être précisé pour orienter les résultats en fonction de paramètres technologiques. Les opérations permises sont les suivantes : décalage $t_{i+1} = t_i \ll k$, addition de la variable d'entrée $t_{i+1} = t_i \pm x$ et addition d'un nombre avec lui-même après décalage $t_{i+1} = t_i \ll k \pm t_i$. Pour notre exemple $p = c \times x$, l'algorithme de Bernstein trouvera la solution suivante, en 5 additions :

$$\begin{aligned} t1 &= ((x \ll 3 - x) \ll 2) - x \\ t2 &= t1 \ll 7 + t1 \\ p &= (((t2 \ll 2) + x) \ll 3) - x \end{aligned}$$

L'algorithme de Lefèvre [99] permet de trouver des recodages où une même valeur peut être réutilisée « à volonté ». Il fournit la décomposition suivante, en seulement 4 additions, pour notre

exemple :

$$t1 = (x \ll 3) - (x \ll 0)$$
$$t2 = (t1 \ll 2) - (x \ll 0)$$
$$p = (t2 \ll 12) + (t2 \ll 5) + (t1 \ll 0)$$

Nous avons proposé une généralisation de l'algorithme de Lefèvre au cas de matrices constantes ainsi que diverses optimisations décrites dans [12]. Des heuristiques particulières de partage de sous-expressions communes dans un graphe ont été proposées et intégrées dans l'outil d'optimisation. Cet outil permet aussi de générer automatiquement du code VHDL. Les résultats de cet outil d'optimisation sont sensiblement meilleurs que les autres techniques de la littérature.

Avec Nicolas Boullis, nous avons utilisé notre générateur de multiplication par des constantes optimisée pour des applications à la transformée en cosinus discrète inverse (IDCT *inverse discrete cosine transform*) pour des cibles matérielles. Différentes versions sont proposées : parallèle, série (poids faibles en tête) pour les bases $2, 4, 8, 16, 64, 256$. Les résultats correspondants ont été publiés à la conférence Sympa en 2002 [10].

Nous collaborons avec l'Université de Calgary au Canada pour l'utilisation de multiplications par des constantes pour des calculs en traitement du signal. Nous commençons aussi à travailler sur la prise en compte de la précision pour réduire l'espace de recherche en se fixant une précision cible à garantir.

Récapitulatif	
Encadrement :	Nicolas Boullis, doctorat, abandon en 4ème année, 90 %, maintenant IE École centrale Paris
Journal :	IEEE TC 2005 [12]
Conférences :	Sympa 2000 [10], ARITH 2003 [11]
Collaboration :	Université Calgary 2007+
Logiciels :	générateur VHDL

2.6.2. Multiplication tronquée

Dans ce travail, c'est Nicolas Veyrat-Charvillon et Romain Michard qui ont fait la plus grande partie du travail durant leur thèse : développement des méthodes et des outils d'optimisation, expérimentations. Je me suis cantonné à fixer la thématique, donner quelques idées, suivre l'avancement des travaux et participer à la rédaction de l'article [42].

Dans un multiplieur $n \times n$ bits standard, le résultat est le produit complet sur $2n$ bits. Dans bon nombre d'applications, seule une partie des bits de poids forts de ces $2n$ bits sont utiles (souvent les n bits de poids forts qui vont servir d'opérande d'une autre opération). On peut calculer le produit complet puis arrondir ou tronquer sur k bits de poids forts ($k < 2n$). Ceci permet d'obtenir la meilleure valeur possible sur le format réduit de k bits.

La multiplication tronquée permet de calculer une approximation des k bits de poids forts du produit complet directement sans devoir calculer les $2n$ bits. Plus la partie des $2n - k$ bits de poids faibles qui ne sont pas utilisés est grande, moins bonne sera l'approximation du produit. Différentes méthodes ont été proposées pour faire cette approximation. Certaines introduisent un biais constant, d'autres utilisent, de façon plus ou moins complexe, une partie des $2n-k$ bits de poids faibles pour faire une correction. Nicolas et Romain ont proposé une description du problème qui permet d'exprimer les principales caractéristiques des méthodes de multiplication tronquée de la littérature. Ils ont proposé une méthode permettant de prédire les valeurs les plus probables de certaines retenues et de ne propager que ces retenues là. Les résultats sur FPGA montrent des surfaces plus petites que les autres solutions de multiplication tronquée pour des précisions moyennes similaires. Nous n'avons pas pu synthétiser des exemples en Asic faute d'accès à des outils de conception. Ce travail a été publié à la conférence SIPS en 2006 [42].

Récapitulatif	
	Encadrements : Nicolas Veyrat-Charvillon, doctorat 2007 [127], 90 %, maintenant postdoc UCL Romain Michard, doctorat 2008 [111], 100 %, maintenant postdoc INRIA
	Conférences : SIPS 2006 [42]

2.6.3. Fonction puissance entière x^n

Dans ce travail aussi, c'est Nicolas Veyrat-Charvillon et Romain Michard qui ont fait la plus grande partie du travail : développement des méthodes et des outils d'optimisation, expérimentations. Je me suis cantonné à fixer la thématique, donner quelques idées, suivre l'avancement des travaux et participer à la rédaction de l'article [43].

Pour évaluer le carré, le cube ou une puissance entière quelconque x^n avec n entier, de nombreuses petites optimisations ont été proposées dans la littérature. Soit x un entier naturel représenté sur n bits : $x = (x_{n-1}x_{n-2}\dots x_1x_0)_2$. Pour calculer x^2, on utilise les identités logiques suivantes afin de réduire le nombre des produits partiels à calculer (cf. [83, p. 221]) :

- règle 1 : $x_ix_i = x_i$
- règle 2 : $x_ix_j + x_jx_i = 2x_ix_j$
- règle 3 : $x_ix_j + x_i = 2x_ix_j + x_i\overline{x_j}$

Les règles 1 et 2 réduisent clairement le nombre des produits partiels à calculer. Ces règles peuvent être étendues au cube et autres valeurs de n. La règle 3 ne change pas le nombre de produits partiels mais décale certains termes d'un rang vers la droite ce qui permet de réduire la profondeur logique du circuit. Nicolas et Romain ont proposé de nouvelles identités logiques permettant la réécriture de certains termes assez complexes. Une généralisation de la règle 3 a été proposée. L'ordre de l'application des différentes règles de réécriture est un point critique du problème. Différentes stratégies assez simples ont été testées, mais une étude plus approfondie mériterait d'être menée. Les nouvelles règles proposées permettent de réduire la surface des opérateurs de 10 à 30 % selon la taille de l'opérande et la valeur de n pour des circuits FPGA. Ici aussi, nous n'avons pas pu faire d'essai sur circuit Asic. Ce travail a été publié à la conférence SPIE en 2006 [43].

Récapitulatif	
	Encadrements : Nicolas Veyrat-Charvillon, doctorat 2007 [127], 90 %, maintenant postdoc UCL Romain Michard, doctorat 2008 [111], 100 %, maintenant postdoc INRIA
	Conférences : SPIE 2006 [43]

2.7. Bibliothèques flottantes pour processeurs entiers

2.7.1. Bibliothèque flottante pour processeurs CoolRisc

Dans le cadre du projet FPPA présenté en section 2.2.1, j'ai développé une bibliothèque de calcul flottant pour les processeurs CoolRisc du CSEM. Cette bibliothèque était écrite en assembleur optimisé pour les processeurs CoolRisc. Cette bibliothèque comprenait l'ensemble des fonctions entières sur 16 bits et flottantes [9] usuelles sur 24 bits [10] ainsi que quelques primitives comme la génération de nombres pseudo-aléatoires.

- opérations arithmétiques sur des entiers signés en complément à 2 sur 16 bits (multiplication, division modulaire et comparaison) ;

9. Les processeurs CoolRisc utilisés n'intégraient en matériel qu'une unité arithmétique et logique ainsi qu'un multiplieur 8×8 bits pour les entiers.

10. Codage virgule flottante sur 24 bits (8 exposant et 15 mantisse et 1 signe). Ce codage est très proche des codages spécifiés par la norme IEEE 754 [69]. Toutefois, il n'est pas compatible avec cette norme.

– opérations arithmétiques sur des réels codés en virgule flottante sur 24 bits (addition, multiplication, division, racine carrée et comparaison) ;
– évaluation de fonctions élémentaires sur les flottants 24 bits ($\sin, \cos, \exp, \ln$ et $\arctan$) ;
– conversions entre les entiers 16 bits et les flottants 24 bits (dans les deux sens) ;
– tirages aléatoires uniformes (résultats entiers ou réels) ;
– tabulation de constantes mathématiques usuelles ($\pi, \ln 2, e, \sqrt{2}$ et $\sqrt{3}$) ;
– mise en place de schémas de routage classiques (diffusions en `XY` ou en `YX` ou par lignes ou par colonnes, au plus proche voisin) ;
– opérations de déplacement de zones de mémoire dans la grille.

Les résultats du projet FPPA ont été publiés aux conférences ICES en 1998 [47] et MicroNeuro en 1999 [31].

Récapitulatif	
	Conférences : ICES 1998 [47], MicroNeuro 1999 [31] Logiciels : bibliothèque de calcul (flottant + autres) pour processeur CoolRisc

2.7.2. Bibliothèque FLIP pour processeur VLIW ST200

Avec Claude-Pierre Jeannerod et Jean-Michel Muller, nous avons co-encadré la thèse de Saurabh K. Raina [120] entre 2003 et 2006. S. K. Raina avait une bourse de thèse de la région Rhône-Alpes. Cette thèse portait sur le développement d'algorithmes en virgule flottante adaptés à l'implantation sur des processeurs contenant uniquement des unités de calcul entier ou en virgule fixe. Dans le cadre d'une collaboration avec la société STMicroelectronics à Grenoble, nous avons développé une bibliothèque flottante simple précision IEEE 754 [69] et optimisée pour la famille de processeurs VLIW (*very large instruction word*) ST200 de STMicroelectronics (en particulier le ST220). Cette bibliothèque, développée en C, a été nommée FLIP (*floating-point library for integer processor*). L'architecture des processeurs ST200 est décrite en figure 2.14. Sur le ST220, les multiplieurs cibles étaient rectangulaires 16×32 bits.

Les cinq opérations flottantes de base ont été implantées et optimisées : addition, soustraction, multiplication, division et racine carrée. Les comparaisons et différents tests de valeurs spéciales (`NaN`, $\pm\infty$, ± 0) sont aussi implantés. D'autres fonctions ont été ajoutées : le carré, la multiplication–addition fusionnée ou FMA (pour *fused multiply and add*), l'inverse et la racine carrée inverse. Quelques approximations de fonctions élémentaires ont aussi été développées, mais uniquement pour des domaines[11] réduits : sinus, cosinus, exponentielle et logarithme.

La première version de la bibliothèque publiée à la conférence SPIE en 2004 [3] montre des performances meilleures de 20 % à 40 % en vitesse par rapport à la bibliothèque optimisée originale de STMicroelectronics. Les ingénieurs de la division compilation ne pensaient pas que de tels gains seraient possibles. Ces accélérations ont été rendues possibles en utilisant plusieurs optimisations : utilisation d'instructions particulières pour simplifier la détection des valeurs spéciales, utilisation de polynômes plus efficaces pour les approximations de fonctions (avec des techniques d'approximation présentées au chapitre 3) et un meilleur remplissage des *bundles*[12] d'instructions VLIW. L'optimisation de la détection des valeurs spéciales a permis de limiter les branchements et d'augmenter le taux de remplissage moyen des *bundles* de 47 % à 67 % pour l'addition et de 54 % à 71 % pour la multiplication. Individuellement, ces opérations ont été accélérées d'un facteur 1.3 en moyenne [3].

11. Intervalle des opérandes en entrée.
12. 4 instructions 32 bits regroupées dans un « paquet » de 128 bits et exécuté à chaque cycle.

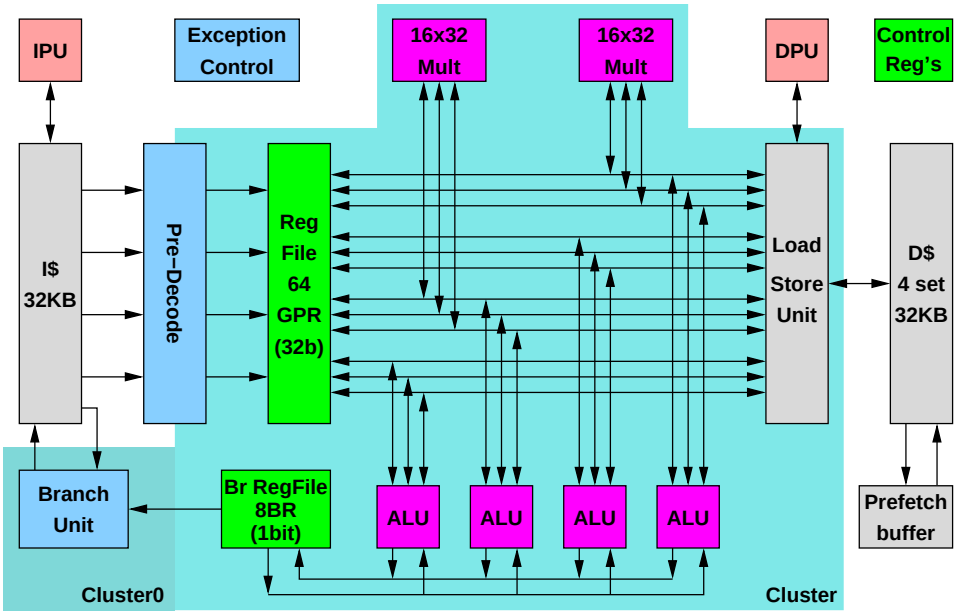


FIGURE 2.14.: Architecture des processeurs ST200 (ST220).

algorithmes de division	temps de calcul (nb. cycles)	taille du code
original (ST)	171	784
restaurante	134	676
non restaurante	117	792
SRT base 2	131	820
SRT base 4 avec $q_j \in \{-2, \ldots, 2\}$	125	716
SRT base 4 avec $q_j \in \{-3, \ldots, 3\}$	155	628
SRT base 512 + *prescaling*	60	840

TABLE 2.1.: Caractéristiques des méthodes de division testées dans FLIP.

Dans le cas particulier de la division, nous avons implanté différentes méthodes issues de la littérature [82, 116]. Nous avons aussi proposé une méthode de division en grande base avec une technique de mise à l'échelle (*prescaling*) par approximation polynomiale optimisée. Le tableau 2.1 présente les résultats d'implantation des différentes méthodes de division. Nous obtenons une accélération d'un facteur 3 avec notre méthode en grande base (512) et un *prescaling* qui utilise des polynômes bien adaptés. Les méthodes et résultats complets relatifs à la division ont été publiés à la conférence IMACS en 2005 [36].

Un autre aspect important du travail sur cette bibliothèque a porté sur la validation. Nous avons implanté une méthode de génération de cas pathologiques pour effectuer des tests. Des vecteurs de test sont générés automatiquement pour le produit cartésien de différentes valeurs pour les découpages ($\{\min, \min+1, \ldots, \min+k\}$, $\{\max-k', \ldots, \max-1, \max\}$, valeurs autour du milieu, etc.). Des tests aléatoires intensifs ont aussi été utilisés.

Travailler sur cette bibliothèque a été très intéressant. Une amélioration moyenne des performances de 10 à 15 % a été constatée sur différentes applications de calcul. Pour certaines applications, les gains étaient encore plus importants. Par exemple, pour des codeurs audio une accélération de facteur 1.48 a été obtenue. Pour le jeu `quake` sous linux, une accélération de

facteur 1.3 a été obtenue.

Récapitulatif	
Encadrement :	Saurabh K. Raina, doctorat 2006 [120], 40 %, MdC en Inde
Conférences :	SPIE 2004 [3], IMACS 2005 [36]
Collaboration :	STMicroelectronics 2003–2006
Logiciels :	bibliothèque flottante FLIP pour processeurs ST200

Récemment, la dernière version (développement sur 2007–2009) de la bibliothèque FLIP a été vendue à la société STMicroelectronics. Ma participation à cette bibliothèque a été estimée à 10 %. J'estime ma participation au développement de la première version de FLIP, celle disponible en fin de thèse de S. K. Raina, entre 30 et 40 %.

2.8. Division par des constantes

Dans le cadre d'un contrat INRIA avec la division compilation de la société STMicroelectronics à Grenoble, nous avons étudié avec Jean-Michel Muller des algorithmes logiciels pour la division par des constantes (c.-à-d. des divisions où le diviseur est connu à la compilation). Cette opération doit être particulièrement optimisée dans les cas suivants : calcul d'indice maximum de boucles complexes, calcul d'adresse d'objet (en C/C++ `/sizeof(T)`), des fonctions de hachage, des conversions entre bases 10 et 2 pour les entrées–sorties. Nos algorithmes permettent de générer du code optimisé pour les processeurs DSP de la famille ST100 de STMicroelectronics. Le point critique de ce travail porte sur l'utilisation au mieux d'unités de calcul comme le MAC (*multiply and accumulate*) sur des entiers ou en virgule fixe communes dans les DSP. Dans les ST100 cibles, on trouve un MAC de type $16 \times 16 + 40 \to 40$. Une bonne maîtrise des erreurs de calcul permet ici de gagner à la fois en précision et en quantité de calcul et donc en vitesse. En particulier, pour certaines valeurs de la constante, nous avons des algorithmes de division par ces constantes sans aucun test. Nos algorithmes ont été intégrés aux compilateurs de STMicroelectronics. Une amélioration de la vitesse de 10 % à 60 % a été obtenue sur des *benchmarks* de codage audio standards. Les résultats ont été publiés à la conférence ARITH 17 en 2005 [46].

Récapitulatif	
Conférence :	ARITH 2005 [46]
Collaboration :	STMicroelectronics (contrat INRIA)
Logiciel :	générateur de code pour DSP ST100

2.9. Approximations polynomiales

Le chapitre 3 est intégralement consacré à ce thème de recherche. Seul le tableau récapitulatif est donné ci-dessous.

Récapitulatif	
Journaux :	ACM TOMS 2006 [14], EL 2006 [15], IJHPSA 2007 [56] (article invité) TSI 2008 [45]
Conférences :	Asilomar 2004 [13], ASAP 2005 [41], ICASSP 2006 [53], Asilomar 2006 [52] NEWCAS 2007 [55]
Collaboration :	ACI GAAP
Logiciel :	générateur de code VHDL

2.10. Calcul à basse consommation d'énergie

La consommation d'énergie est devenue la principale contrainte lors de la conception de bon nombre de systèmes matériels ou logiciels. Dans un SoC, la consommation d'énergie des unités de calcul est considérée, souvent à juste titre, comme faible par rapport à celle des accès mémoires ou de certaines communications internes. Toutefois, il est quand même nécessaire de la réduire le plus possible dans de nombreux cas. Par exemple, les additionneurs et « très petits » multiplieurs d'une unité de génération d'adresses (ou AGU *address generation unit*) d'un processeur à hautes performances est l'un des points les plus chauds du circuit [104]. Les performances énergétiques de circuits spécialisés dépendent beaucoup de celles de leurs blocs de calculs intensifs.

J'ai publié un chapitre d'introduction à la consommation des opérateurs arithmétiques dans le livre collectif *Low Power Electronics Design* paru aux éditions CRC en 2004 [50]. J'ai publié un article invité sur ce même thème dans la revue TSI (Technique et Science Informatiques) en 2007 [57]. En juin 2009, j'ai été invité à donner un exposé sur la consommation des opérateurs arithmétiques à la conférence FTFC (Faible Tension Faible Consommation) à Neuchâtel en Suisse [60], les transparents sont accessibles sur ma page web [13]. J'ai donné un cours d'introduction à la consommation d'énergie dans les circuits intégrés numériques lors de l'école thématique CNRS ECOFAC2010 en mars–avril 2010 [62] dont les transparents sont accessibles sur le site web `http://ecofac2010.irisa.fr/`.

2.10.1. Opérateurs arithmétiques pour la basse consommation

Dans [25, 64, 63], nous avons montré que pour certaines applications de contrôle numérique, l'arithmétique en ligne permet d'obtenir des solutions plus économes en énergie qu'avec une arithmétique parallèle (cf. section 2.1).

Lors de nos travaux sur les approximations polynomiales (cf. chapitre 3), nous avons étudié le choix de coefficients de polynômes pour réduire la consommation d'énergie d'opérateurs matériels dédiés pour l'évaluation de fonctions. Nous privilégions l'utilisation de coefficients très creux (avec très peu de chiffres non nuls). Ceci nous permet de remplacer des multiplieurs par des séquences d'additions et de soustractions de façon similaire au cas de la multiplication par des constantes présenté en section 2.6.1. Dans certains cas, nous obtenons des réductions de la consommation de 50 %. Les résultats ont été publiés à la conférence Asilomar en 2006 [52].

J'étudie l'influence du codage des chiffres au niveau circuit. Le choix de la base et du codage interne des chiffres (nombre de fils et nombre de portes pour effectuer les calculs élémentaires) a une influence directe sur la consommation d'un opérateur de calcul. En effet, il est possible qu'avec un codage astucieux moins de transitions (changement d'états électriques) soient nécessaires au prix d'un circuit un peu plus gros pour effectuer une opération donnée. Les compromis entre l'activité électrique et la surface des opérateurs sont nombreux. Une faible activité implique une faible consommation dynamique. Mais une large surface de silicium implique, en particulier pour les technologies de plus en plus fines, une large consommation statique (courants de fuite). Il reste encore beaucoup de travail à effectuer sur ce thème. En particulier, il faut réaliser des simulations électriques intensives pour pouvoir caractériser précisément les performances énergétiques de certaines solutions de codage.

13. `http://www.irisa.fr/prive/Arnaud.Tisserand/`

Récapitulatif	
Chapitre :	dans livre CRC 2004 [50]
Journal :	TSI 2007 [57] (article invité)
Conférences :	Asilomar en 2006 [52], FTFC 2009 [60]
Collaborations :	EPFL 1997–1999, CSEM 2000, STMicroelectronics 2001
Logiciel :	générateurs de code VHDL (multiplication, division, approx. fonctions)

2.10.2. Modélisation et évaluation de la consommation d'opérateurs arithmétiques

Nous travaillons sur des modèles de la consommation d'énergie des opérateurs arithmétiques. Ces modèles se basent sur des caractéristiques de la représentation des nombres (p. ex. la distribution de probabilité des chiffres des opérandes), des algorithmes utilisés pour effectuer les opérations de base (p. ex. la décomposition en portes et en étages), et quelques informations sur l'implantation pratique (p. ex. les modèles de délais et de sortance). Cette modélisation permettra, à terme, de guider des méthodes d'optimisation pour la génération automatique, ou quasi-automatique, d'opérateurs arithmétiques.

Avec Romain Michard et Nicolas Veyrat-Charvillon, nous avons étudié l'activité d'une méthode d'évaluation de fonctions en matériel. La E-méthode [81], permet d'évaluer des polynômes et des fractions rationnelles avec une itération à base d'additions et de décalages similaire à celle utilisée pour la division. Dans cet algorithme, les chiffres du résultat sont déterminés de façon itérative par une fonction de sélection à partir des chiffres précédents du résultat et d'un résidu similaire au reste partiel dans la division. Les chiffres issus de la fonction de sélection, représentés en notation redondante, offrent une certaine latitude de choix. Dans ce travail, nous avons étudié l'activité, d'un point de vue statistique, impliquée par la fonction de sélection. Nous avons proposé une ébauche de méthode permettant de réduire cette activité en utilisant la connaissance du chiffre du résultat sélectionné à la dernière itération. Les résultats de ce travail ont été publiés à la conférence FTFC en 2005 [39].

La consommation d'énergie liée à l'activité parasite est souvent importante dans les multiplieurs [50, 57]. De bonnes modélisations de l'activité parasite sont disponibles pour les additionneurs, mais il n'en existe pas pour les multiplieurs qui soit à la fois simple, rapide et suffisamment précise pour guider les choix d'un générateur automatique. J'ai étudié la consommation des arbres de réduction de multiplieurs à base de cellules standards. J'ai proposé une formulation de l'activité parasite dans les arbres de réduction des multiplieurs rapides. Elle permet de comparer rapidement différentes stratégies d'optimisation lors de la conception. Les résultats ont été publiés à la conférence FTFC en 2007 [54]. De nombreux travaux sont encore à faire pour rendre cette méthode efficace, p. ex. l'analyse de la sensibilité à certains paramètres.

Je travaille sur une méthode d'estimation de la consommation d'énergie des opérateurs arithmétiques par mesure de l'activité logique. En insérant des compteurs d'activité comme illustré en figure 2.15, on peut mesurer très précisément, sur un circuit FPGA, l'activité logique réelle d'opérateurs de calcul. L'utilisation de FPGA permet d'atteindre des hautes vitesses et ainsi de faire des statistiques précises. Pour le moment, une des limites est liée aux temps de communications avec la carte FPGA. Les résultats ont été publiés à la conférence Asilomar en 2008 [58]. Ce travail s'est fait dans le cadre du projet ANR ROMA. Bien entendu, on n'estime pas du tout l'activité parasite avec cette méthode. Un travail restant à faire est de coupler cette mesure précise de l'activité logique et la modélisation de l'activité parasite.

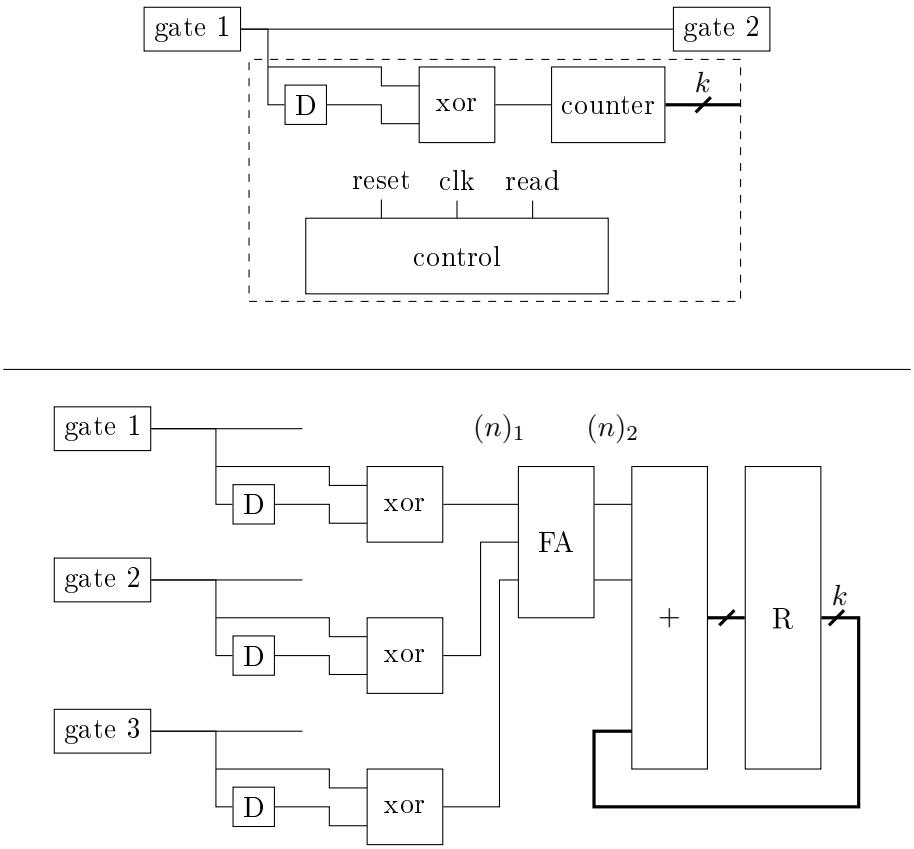


FIGURE 2.15.: Insertion de compteurs d'activité simple (haut) ou multiples (bas).

GPU	vide (W)	transpo. naïve P_m (W)	transpo. naïve T (ms)	transpo. naïve E (J)	transpo. optim. P_m (W)	transpo. optim. T (ms)	transpo. optim. E (J)	mult. matrice P_m (W)	mult. matrice T (ms)	mult. matrice E (J)	sgemm cuBLAS P_m (W)	sgemm cuBLAS T (ms)	sgemm cuBLAS E (J)
G80	68	103	2.4	0.247	127	0.30	0.038	132	25	3.3	135	11.8	1.59
G92	71	94	3.66	0.344	105	0.45	0.047	117	23.3	2.73	122	11.4	1.39
GT200	51	73	4.11	0.300	83	0.50	0.042	114	13	1.48	113	7.44	0.84

TABLE 2.2.: Puissance moyenne P_m, temps de calcul T et énergie E mesurés pour des algorithmes GPGPU : transposition de matrice naïve et optimisée, multiplication de matrice classique et sgemm de cuBLAS.

RÉCAPITULATIF
ENCADREMENTS : Nicolas Veyrat-Charvillon, doctorat 2007 [127], 90 %, maintenant postdoc UCL
Romain Michard, doctorat 2008 [111], 100 %, maintenant postdoc INRIA
CONFÉRENCES : FTFC 2005 [39], FTFC 2007 [54], Asilomar 2008 [58]
COLLABORATIONS : projet ANR ROMA

2.10.3. Consommation d'énergie dans les processeurs graphiques

En collaboration avec Sylvain Collange et David Defour de l'Université de Perpignan, nous travaillons sur l'évaluation de la consommation d'énergie dans les processeurs graphiques ou GPU (*graphic processing unit*). L'utilisation des GPU pour faire du GPGPU (*general purpose computing on GPU*) est croissante. Leur efficacité énergétique peut être assez intéressante. Par exemple, en simple précision flottante, on obtient 4 GFLOP/W pour un GPU GTX280 contre 0.8 GFLOP/W pour un CPU [14] i7 960. Les problèmes d'acheminement des énergies, d'évacuation des calories et du coût énergétique récurrent sont de plus en plus importants les centres de calcul. Les GPU peuvent devenir une alternative aux CPU dans certains cas.

Nous étudions les liens entre le courant mesuré, le temps de calcul et le type des unités qui sont utilisées pour effectuer différents calculs GPGPU. Nous utilisons des cartes graphiques Nvidia et les outils de développement CUDA en C. Pour le moment, les mesures sont assurées par une instrumentation à faible vitesse (quelques dizaines de kHz). L'analyse des mesures de courant permet d'identifier les phases de fonctionnement de différents blocs fonctionnels et de caractériser leur consommation. Les blocs considérés correspondent aux unités du GPU qui sont utilisées lors de l'exécution de noyaux (*kernels*) de calcul : bancs de registres, éléments de la hiérarchie mémoire, unités de calcul. L'analyse de la consommation électrique des GPU nous fournit des informations sur l'organisation de la hiérarchie mémoire et du comportement des unités fonctionnelles. Les résultats de ce travail ont été publiés à la conférence ICCS en 2009 [19]. Les tableaux 2.2 et 2.3 présentent des exemples de mesures obtenues par Sylvain. Jusqu'ici mon travail s'est limité à une expertise technique sur les aspects circuits et consommation d'énergie et à la participation à la rédaction de l'article [19]. Je travaille sur la mise en œuvre et l'utilisation d'un banc de mesure plus performant utilisant un oscilloscope rapide. Dans un premier temps, nous espérons pouvoir analyser plus finement l'architecture et le fonctionnement des GPU grâce à des mesures bien plus rapides. À moyen terme, nous souhaitons pouvoir étudier des algorithmes de calcul et représentations des données pour la faible consommation d'énergie sur GPU.

RÉCAPITULATIF
CONFÉRENCE : ICCS 2009 [19]
COLLABORATION : Université Perpignan 2008+

14. Pour nous c'est un processeur classique à hautes performances. CPU pour *central processing unit*.

		G80			G92			GT200		
Opérations	nb. inst.	P (W)	CPI	E (nJ/*warp*)	P (W)	CPI	E (nJ/*warp*)	P (W)	CPI	E (nJ/*warp*)
MAD	32	107	4.75	8.57	100	4.29	5.06	91	4.3	5.31
Pred	32	90	2.38	2.43	93	2.39	2.14	75	2.36	1.75
MAD+MUL	64	117	3.19	7.24	111	2.83	4.61	102	2.82	4.44
MAD+RCP	40	115	3.96	8.63	110	3.55	5.63	98	3.54	5.14
RCP	8	98	15.89	22.07	96	16	16.28	81	15.99	14.81
MOV	32	118	2.31	5.34	113	2.46	4.21	101	2.46	3.79

TABLE 2.3.: Nombre d'instructions, puissance mesurée, nombre de cycles par instruction et énergie par *warp* pour des opérations sur les GPU G80, G92 et GT200.

2.11. Opérateurs arithmétiques pour la cryptographie

Dans la cryptographie à clé publique [110], RSA [15] ou ECC [16] [92], l'arithmétique joue un rôle important dans la mise en œuvre de cryptosystèmes à la fois efficaces et sûrs. En particulier, l'arithmétique des corps finis [100], qui sera abordée à la section 2.11.1, doit être très rapide étant donnée la quantité de calculs effectués tout en nécessitant des ressources limitées (surface de circuit, taille mémoire, énergie). Enfin, l'exécution des opérations arithmétiques ne doit pas offrir de points faibles exploitables pour des attaques physiques, qui seront abordées à la section 2.11.3. J'ai été invité à donner un *tutorial* sur ce thème à la conférence ISSAC (*International Symposium on Symbolic and Algebraic Computation*) en 2005 [51].

2.11.1. Arithmétique dans les corps finis

Avec l'accroissement des besoins en cryptographie (clés de plus en plus longues, intégration dans des dispositifs portables, vitesse de chiffrement ou de signature élevées, protection contre les attaques physiques...), la conception d'opérateurs arithmétiques spécifiques est nécessaire. Nous travaillons depuis 2002 sur des opérateurs arithmétiques matériels pour la cryptographie. En particulier, nous étudions les représentations des nombres et les algorithmes de calcul pour les corps finis. Nos travaux portent essentiellement sur les corps finis premiers $\mathbb{F}_p$ (c.-à-d. où p est un grand nombre premier). L'arithmétique dans le corps $\mathbb{F}_p$ est l'arithmétique modulo p. Nous étudions à la fois des opérateurs dans $\mathbb{F}_p$ pour des p quelconques (sans structure mathématique particulière) ou bien pour des p spécifiques (p. ex. avec très peu de bits non nuls comme les p des courbes elliptiques préconisées par le NIST [92, annexe A.2.1]).

Avec Jean-Luc Beuchat (LIP) et Laurent Imbert (LIRMM), nous avons étudié des algorithmes et développé des générateurs automatiques pour la multiplication modulaire ($a \times b \bmod p$) avec des moduli de type $p = 2^n \pm 1$ (pour différentes tailles) pour des implantations sur FPGA. Cette opération est importante dans bon nombre d'applications de cryptographie. Nous avons ainsi comparé différents algorithmes de multiplication modulaire. Les résultats ont été publiés à la conférence SPIE en 2003 [4]. Ce travail s'est fait dans le cadre du projet OpAC *Opérateurs Arithmétiques pour la Cryptographie* de l'ACI *Cryptologie*.

Nous avons travaillé avec Jean-Luc Beuchat (LIP), Nicolas Sendrier (INRIA) et Gilles Villard (CNRS–LIP) sur l'implantation matérielle d'un algorithme de signature numérique avec des signatures courtes (moins de 100 bits) proposé dans le projet INRIA CODES à Rocquencourt.

15. Algorithme de cryptographie à clé publique du nom de ses inventeurs : R. Rivest, A. Shamir et L. Adleman.
16. *Elliptic curve cryptography* ou cryptographie sur les courbes elliptiques.

Cet algorithme est basé sur la théorie des codes. Son implantation logicielle optimisée (C et assembleur) nécessitait environ une minute par signature sur une machine à base de Pentium 4 à 2 GHz. Notre première version pour FPGA XCV300 de Xilinx permettait de signer en 0.86 seconde. Les premiers résultats sont regroupés dans le rapport de recherche [5]. Ce travail s'est fait dans le cadre du projet OCAM *Opérateurs Cryptographiques et Arithmétique Matérielle* de l'ACI *Sécurité Informatique*.

Récapitulatif	
	Conférence : SPIE 2003 [4], ISSAC 2005 [51]
	Collaborations : LIRMM, Université Calgary Canada, INRIA Rocquencourt ACI Cryptologie (2002–2005), ACI Sécurité Informatique (2003–2006)

2.11.2. Fonctions de hachage cryptographique

Lors d'une collaboration avec Ryan Glabb et Graham Julien du laboratoire ATIPS de l'Université de Calgary au Canada, Nicolas Veyrat-Charvillon (LIP) et Laurent Imbert (CNRS–LIRMM), nous avons étudié l'implantation optimisée sur FPGA de fonctions de hachage cryptographique de la famille SHA-2 [85] (224, 256, 384, 512 bits). Notre implantation fonctionne à haute fréquence et avec seulement deux cycles de latence. Nous avons une architecture de type « multi-modes », c.-à-d. que le même opérateur peut fonctionner dans plusieurs modes. Ici les modes sont un hachage SHA-384 ou SHA-512 ou bien deux hachages indépendants SHA-224 ou SHA-256. La plus grande partie du travail a été faite par Ryan Glabb et Nicolas Veyrat-Charvillon. Mon travail s'est limité à fixer la thématique, fournir une expertise technique sur l'implantation FPGA, suivre l'avancement des travaux et participer à la rédaction des articles. Les résultats de ce travail ont été publiés à la conférence ERSA en 2006 [33] et dans le journal JSA (*journal of systems architecture*) en 2007 [34].

Récapitulatif	
	Encadrement : Nicolas Veyrat-Charvillon, doctorat 2007 [127], 90 %, maintenant postdoc UCL
	Journal : JSA 2007 [34]
	Conférence : ERSA 2006 [33]
	Collaborations : Université Calgary Canada, ACI Sécurité Informatique (2003–2006)

2.11.3. Sécurisation d'opérateurs arithmétiques pour la cryptographie

La sécurisation de cryptoprocesseurs vis à vis d'attaques physiques comme la mesure de la consommation d'énergie [101], la mesure du temps d'exécution ou bien l'injection de fautes, est un enjeu important pour bon nombre d'applications. En effet, même si un protocole cryptographique est considéré comme mathématiquement robuste, son implantation dans un cryptoprocesseur risque d'être attaquée physiquement si on ne prend pas de précaution ou sans intégration de dispositifs de protection appelés contre-mesures. J'ai fait un cours [61] sur les opérateurs arithmétique sécurisés à l'école thématique CNRS ARCHI09 en mars 2009 (les transparents sont accessibles sur le web de l'école : `http://www.irisa.fr/archi09/`).

Lors d'une collaboration avec Nicolas Méloni (LIRMM) et le groupe « *code and crypto* » de Liam Marnane (UCC *University College Cork* en Irlande), nous avons étudié et réalisé en FPGA l'implantation des travaux de thèse de N. Méloni sur les chaînes d'addition [106]. Cette technique permet de concevoir des cryptoprocesseurs protégés contre les attaques simples par observation de la consommation qui sont dangereuses pour ECC. Ma participation portait sur l'implantation FPGA des opérateurs arithmétiques optimisés et sur la rédaction des publications communes. Ces travaux ont été publiés dans la conférence ITNG en 2007 [17] puis dans les journaux *Journal of Computers* en 2007 [18] et *International Journal of High Performance Systems Architecture*

en 2007 [16]. Un des problèmes, importants pour les prochaines années, porte sur l'analyse *a priori* de la robustesse de contre-mesures. Comment comparer des solutions de protection sans devoir faire des attaques longues et coûteuses.

Dans le cadre de la thèse de J. Francq (LIRMM, EMSE, CMP Gardanne), nous avons étudié des opérateurs arithmétiques pour ECC avec des protections contre les attaques par fautes en représentation *borrow-save*. Ce travail a été publié à la conférence FDTC en 2008 [29].

Avec deux doctorants, Thomas Chabrier et Danuta Pamula, nous étudions des protections contre les attaques physiques par canaux cachés au niveau arithmétique. En particulier, nous étudions l'implantation matérielle et l'utilisation logicielle d'unités arithmétiques reconfigurables pour la cryptographie à clé publique. Ces unités devront pouvoir implanter les principaux calculs sur les corps finis et ce de différentes façons qui pourront être changées par reconfiguration dynamique du cryptoprocesseur. Différentes représentations des nombres et différents algorithmes arithmétiques sont envisageables pour effectuer les opérations dans les corps finis. Plutôt que de fixer une représentation et un algorithme pour une opération comme dans les cryptosystèmes actuels, les unités reconfigurables étudiées devront pouvoir supporter différentes représentations des nombres et différents algorithmes arithmétiques. La reconfiguration dynamique permettant de changer la représentation des nombres et les algorithmes arithmétiques utilisés en cours de calcul doit rendre bien plus difficiles les prises de repères nécessaires pour mener à bien une attaque. En plus d'une reconfiguration fonctionnelle de haut niveau (pour supporter différents types de calculs), on peut envisager une reconfiguration fine et locale en cours d'exécution pour "brouiller" les paramètres physiques exploitables lors des attaques (activité en sortie des portes logiques, signature en courant, variations du temps de calcul...). Bien entendu, différents compromis entre reconfiguration de haut niveau, reconfiguration fine et locales et le niveau de performance et de robustesse seront à étudier.

Nous avons commencé à étudier l'utilisation de la représentation modulaire des nombres, ou RNS, pour la protection contre des attaques par canaux cachés au niveau arithmétique.

Récapitulatif	
Encadrements :	Thomas Chabrier, doctorant depuis 2009, 90 % Danuta Pamula, doctorante depuis 2009, 90 %
Journaux :	JCP 2007 [18], IJHPSA 2007 [16]
Conférences :	ITNG 2007 [17], FDTC 2008 [29]
Collaborations :	UCC Cork 2006+, CMP Gardanne, Univ. Waterloo et Calgary Canada

2.12. Générateur de diviseur `divgen`

Dans ce travail, c'est Nicolas Veyrat-Charvillon et Romain Michard qui ont fait la plus grande partie du travail de développement, d'optimisation et d'expérimentation. J'ai juste défini la thématique, supervisé les travaux, aidé sur quelques aspects techniques lié au développement et participé à la rédaction de l'article [38].

Des divisions sont présentes dans bon nombre d'applications évoluées en calcul scientifique et multimédia. Malheureusement, le faible support fourni pour son implantation en matériel oblige souvent les concepteurs à modifier leurs algorithmes afin d'éviter cette opération. Dans les processeurs à hautes performances, la division a longtemps été négligée [115]. Les diviseurs issus d'outils de synthèse ou de blocs IP ont des performances souvent médiocres. Ceci est d'autant plus regrettable que de nombreux travaux de recherche ont été menés sur les algorithmes de division efficaces [82, 116].

Nous avons donc décidé de développer un générateur de code VHDL pour implanter des diviseurs optimisés utilisant des algorithmes évolués. Ce programme, développé en C++ et

disponible sous licence GPL sur le web [17], a été nommé `divgen`. Seules les représentations entières et virgule fixe sont supportées pour les opérandes et les résultats (quotient et reste). Les cibles d'implantation sont des circuits FPGA ou Asic. Il supporte différents algorithmes de division, bases de calcul intermédiaire, représentations de nombres et diverses optimisations au niveau circuit ou architecture. Il y a de nombreux paramètres dans les algorithmes de division. Le programme `divgen` génère rapidement un code VHDL à partir des spécifications. Il est possible de faire de l'exploration d'architectures avec différents types de diviseurs assez facilement.

Les résultats de ce travail ont été publiés à la conférence SPIE en 2005 [38]. À ce jour, le programme `divgen` a été téléchargé plus de 800 fois. Il a été utilisé dans le cadre d'une collaboration entre l'Inria et le laboratoire Léti du CEA à Grenoble. Nous espérons pouvoir poursuivre ce travail dans le futur et apporter des nouvelles optimisations. En particulier, la structure des FPGA récents offre probablement de bonnes possibilités d'optimisation.

Récapitulatif	
Encadrements :	Nicolas Veyrat-Charvillon, doctorat 2007 [127], 90 %, maintenant postdoc UCL Romain Michard, doctorat 2008 [111], 100 %, maintenant postdoc INRIA
Conférence :	SPIE 2005 [38]
Collaboration :	CEA Léti
Logiciel :	générateur de code VHDL `divgen`, licence GPL

2.13. Bibliothèque logicielle PACE pour la cryptographie

Parallèlement au développement d'opérateurs arithmétiques matériels pour la cryptographie (cf. section 2.11), nous travaillons, depuis 2006, sur une bibliothèque logicielle d'arithmétique pour la cryptographie. Cette bibliothèque, développée en C++ sous licence LGPL, a été nommée Pace pour *prototyping arithmetic in cryptography easily*. Ce travail se fait en collaboration avec Pascal Giorgi, Laurent Imbert, Thomas Izard et Agostinho Peirera du Lirmm.

Dans la cryptographie sur les courbes elliptiques, ou ECC pour *elliptic curve cryptography* [92], on a besoin d'effectuer des calculs à plusieurs niveaux : sur la courbe elliptique (addition, doublement et triplement de points, multiplication scalaire), sur un corps fini $\mathbb{F}_q$ [100] sur lequel la courbe est définie et enfin sur des grands entiers ou des polynômes (eux-mêmes définis sur le corps $\mathbb{F}_q$). Aucun de ces objets mathématiques n'est directement supporté dans les langages de programmation. Il faut donc utiliser des bibliothèques logicielles. De plus, étant donnée la taille des nombres manipulés (de 160 à 600 bits pour ECC), la complexité et le nombre des opérations effectuées, il faut avoir des algorithmes et des implantations très rapides.

La bibliothèque Pace supporte différentes représentations des nombres et algorithmes arithmétiques pour effectuer des opérations sur les corps finis et en cryptographie ECC. Un de nos buts est de fournir un support simple pour le prototypage au niveau arithmétique d'ECC tout en garantissant de très bonnes performances. Par exemple, nous souhaitons pouvoir changer la représentation de nombres ou bien des algorithmes de calculs arithmétiques à certains endroits d'un programme sans devoir réécrire beaucoup de code.

La bibliothèque Pace est constituée de trois niveaux comme illustré en figure 2.16 :

- le *niveau arithmétique* qui regroupe les objets et fonctions pour le calcul sur les grands entiers et des éléments de corps finis $\mathbb{F}_q$;
- le *niveau d'évaluation des performances* ou *monitoring* qui permet de faire des statistiques automatiquement sur différentes quantités comme le temps de calcul, le nombre de cycles d'horloge (en utilisant les compteurs de cycles des processeurs), le nombre d'opérations à différents niveaux (courbe, points, éléments de $\mathbb{F}_q$, mots, dans les algorithmes, etc.), les

17. http ://lipforge.ens-lyon.fr/www/divgen/

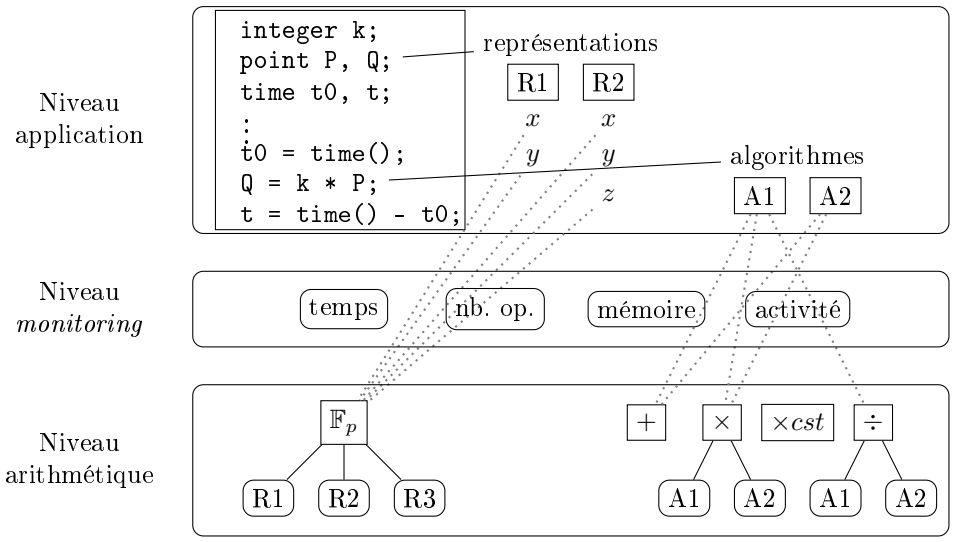


FIGURE 2.16.: Architecture en niveaux de la bibliothèque PACE.

besoins en mémoire à différents niveaux (nombre d'objets, nombre maximum de registres, etc.), variation du poids de Hamming (pour estimer l'activité électrique) ;

– le *niveau applicatif* qui regroupe les objets et fonctions de base pour la cryptographie sur un courbe elliptique $E(\mathbb{F}_q)$ et ses points P, Q (définition de la courbe, addition $P + Q$, doublement $[2]P$ et triplement $[3]P$ de points, multiplication scalaire $[k]P$, vérification $P \in E(\mathbb{F}_q)$) avec k un grand entier.

Pour pouvoir effectuer des changements de représentations et d'algorithmes simplement mais avec des bonnes performances, nous utilisons les mécanismes de méta-programmation par *template* de C++ [66, 68]. Les définitions sont statiques mais modulaires et le compilateur génère le meilleur code possible pour chaque définition utilisée. Par exemple, différentes représentations des points d'une courbe sont possibles au niveau applicatif (p. ex. les systèmes de coordonnées affines ou projectives). Le niveau *monitoring* est lui aussi implanté par méta-programmation *template* C++ pour simplifier son utilisation et n'inclure que les dispositifs de *monitoring* strictement nécessaires pour un programme donné. Les éléments de *monitoring* sont intrusifs. Ils modifient donc les performances. Nous n'implantons que ceux strictement nécessaires (demandés directement ou indirectement) grâce aux spécifications statiques par méta-programmation *template* et par recompilation du code complet.

Au niveau applicatif, il existe aussi de nombreux algorithmes pour effectuer la multiplication scalaire $[k]P$ sur une courbe elliptique. Au niveau arithmétique le plus fin, différentes représentations des éléments du corps fini $\mathbb{F}_q$ et différents algorithmes de calcul sur ce corps sont possibles. Voir l'illustration de ces possibilités en figure 2.16. La bibliothèque PACE nous permet de tester et de comparer rapidement et objectivement différentes combinaisons à partir de codes sources communs.

Nous avons aussi travaillé sur les aspects de validation numérique en fournissant des interfaces vers d'autres bibliothèques comme GMP, $\texttt{mp}\mathbb{F}_q$ ou NTL. On effectue un calcul en utilisant PACE et une autre bibliothèque avec le même code à haut niveau, puis on compare les résultats. Nos interfaces permettent de comparer des objets mathématiques qui ont des représentations différentes comme illustré en figure 2.17. Nous avons aussi validé notre bibliothèque en comparant ses résultats à ceux d'autres outils comme Maple, Pari–GP et Magma.

Les opérations supportées pour les entiers longs, pour des tailles jusqu'à 600 bits, sont : ad-

```
1  integer<100> a,b,c;
2  integer<100, RP_integer_GMP> d,e,f;
3  b = 1;
4  c = 2;
5  a = b + c;
6  e = 1;
7  f = 2;
8  d = e + f;
9  std::cout << a << ',' << d << std::endl;
10 assert(a==d);
```

FIGURE 2.17.: Exemple d'extrait de programme PACE.

dition, soustraction, multiplication, carré, division, racine carré, modulo, inverse modulaire, exponentiation modulaire, différents schémas d'accès aux chiffres, poids de Hamming (`popcount`), comparaisons, conversions depuis/vers GMP et des chaînes de caractères pour les entrées–sorties.

Les opérations supportées pour les éléments de $\mathbb{F}_p$, pour p quelconque et des tailles jusqu'à 600 bits, sont : addition, soustraction, multiplication, carré, inverse, accès aux chiffres, poids de Hamming (`popcount`), égalité, conversions depuis/vers GMP et des chaînes de caractères pour les entrées–sorties.

Au niveau courbe elliptique $E(\mathbb{F}_q)$, avec les points P, Q, nous avons : opposé d'un point $-P$, addition de points $P + Q$, doublement de point $[2]P$, triplement de point $[3]P$, multiplication scalaire $[k]P$, vérification d'appartenance à la courbe $P \in E(\mathbb{F}_q)$.

Les figures 2.17, 2.18 et 2.19 présentent des programmes, ou extraits de programmes, écrits avec PACE et éventuellement leurs résultats d'exécution.

Les résultats relatifs à la première version de la bibliothèque PACE ont été publiés à la conférence SPIE en 2007 [35]. Actuellement, seul le support du corps $\mathbb{F}_p$ pour des p quelconques est bien avancé. Nous espérons pouvoir trouver des ressources pour travailler sur $\mathbb{F}_p$ avec des p particuliers et surtout sur le corps fini $\mathbb{F}_{2^m}$.

Nous avons aussi étudié une version de certains éléments de PACE pour des processeurs graphiques GPU en C++ avec CUDA. Nous avons développé un support pour l'arithmétique modulo des grands premiers (quelconques) en C++ avec l'environnement CUDA. Nous avons les représentations des grands nombres et les opérations : $a \pm b \bmod p$, $a \times b \bmod p$ où a, b et p sont des entiers en multiprécision et p un grand nombre premier (sans structure particulière). Nous essayons d'adapter les représentations des nombres (160–600 bits) et les algorithmes à l'architecture massivement parallèle (*multi-thread*) des GPU. Le portage de tels calculs sur GPU n'est pas simple. Les résultats correspondants ont été publiés à la conférence PARCO en 2009 [30].

Les tableaux 2.4, 2.5, 2.6, 2.7 et 2.8 montrent des résultats de notre version GPU et des comparaisons avec le couplage de PACE et $\texttt{mp}\mathbb{F}_q$ sur CPU. Nous utilisons un couplage de PACE et $\texttt{mp}\mathbb{F}_q$, car $\texttt{mp}\mathbb{F}_q$ est la bibliothèque la plus rapide disponible pour les calculs sur $\mathbb{F}_q$. Dans ces tableaux, le sigle *ror* signifie *run out of register*, c.-à-d. que le compilateur `nvcc` de CUDA n'a pas été capable de générer le code demandé. Ce problème est lié à la phase d'affectation des registres en forme SSA (*static single assignment*) dans le compilateur (bien que les registres du GPU soient réellement suffisants). Nous espérons pouvoir améliorer nos résultats avec les nouvelles générations de GPU qui intègrent des mémoires un peu plus grandes et les nouvelles versions des outils de développement.

```
#include <iostream>
#include <cassert>
#include "pace.hpp"
using namespace std; // for cout & endl

integer<100> p = 29;
typedef gfp<100, p> fp_29;

int main()
{
  fp_29 x = 17, y = 20, z;
  z = x + y;
  assert(z == 8);
  cout << x << " + " << y << " = " << z << endl;
  z = x - y;
  assert(z == 26);
  cout << x << " - " << y << " = " << z << endl;
  z = x * y;
  assert(z == 21);
  cout << x << " * " << y << " = " << z << endl;
  z = inv(x);
  assert(z == 12);
  cout << x << " ^(-1) = " << z << endl;
  return 0;
}
```

FIGURE 2.18.: Exemple de programme PACE.

	addition $\mathbb{F}_p$			multiplication $\mathbb{F}_p$		
N	**local**	**shared**	**register**	**local**	**shared**	**register**
160	9	2.3	0.7	88	40	22
192	11	2.3	0.7	125	51	33
224	23	5.0	1.1	172	107	55
256	26	3.1	1.5	214	80	81
384	38	7.4	3.9	673	221	261

TABLE 2.4.: Temps de calcul en ns pour l'addition et la multiplication sur $\mathbb{F}_p$.

	addition $\mathbb{F}_p$			multiplication $\mathbb{F}_p$		
N	**notre impl.**	$\texttt{mp}\mathbb{F}_q$	**accélération**	**notre impl.**	$\texttt{mp}\mathbb{F}_q$	**accélération**
160	0.7	15	×21	22	64	×2.9
192	0.7	16	×22	33	70	×2.1
224	1.1	21	×19	55	105	×1.9
256	1.5	21	×14	81	109	×1.3
384	3.9	30	×7	261	210	×0.8

TABLE 2.5.: Comparaison des temps de calcul en ns entre notre bibliothèque et $\texttt{mp}\mathbb{F}_q$.

```
#include <iostream>
#include "pace.hpp"
using namespace std; // for cout & endl

integer<100> p = 29;
typedef gfp<100, p> fp_29;
curve<fp_29> E(4, 20);
typedef point_aff<fp_29, E> point;

int main()
{
  E.info();
  point P1(5,22);
  point P2(16,27);
  cout << "P1 = " << P1 << endl;
  cout << "P2 = " << P2 << endl;
  point P3 = P1 + P2;
  cout << "P1 + P2 = " << P3 << endl;
  point P4 = 2 * P1;
  cout << "[2] P1 = " << P4 << endl;
  return 0;
}
```

Résultat d'exécution :

```
Elliptic curve defined by y^2 = x^3 + 4*x + 20
P1 = (5 , 22)
P2 = (16 , 27)
P1 + P2 = (13 , 6)
[2] P1 = (14 , 6)
```

FIGURE 2.19.: Exemple de programme PACE et son résultat d'exécution.

	addition points			doublement point		
N	**local**	**shared**	**register**	**local**	**shared**	**register**
160	2.57	0.78	0.70	1.64	0.50	0.54
192	3.51	1.01	1.13	2.30	0.58	0.70
224	4.41	1.95	*ror*	2.73	1.01	*ror*
256	5.89	1.56	*ror*	3.71	1.09	*ror*
384	13.9	7.50	*ror*	13.3	2.42	*ror*

TABLE 2.6.: Temps de calcul en μs pour l'addition et le doublement de points.

	addition points			doublement point		
N	**notre impl.**	mp$\mathbb{F}_q$+**Pace**	*accélération*	**notre impl.**	mp$\mathbb{F}_q$+**Pace**	*accélération*
160	0.78	1.52	*1.9*	0.50	1.99	*4.0*
192	1.01	1.91	*1.9*	0.58	1.99	*3.4*
224	1.95	2.65	*1.3*	1.01	2.69	*2.6*
256	1.56	2.65	*1.7*	1.09	2.65	*2.4*
384	7.50	5.11	*0.7*	2.42	5.01	*2.0*

Table 2.7.: Temps de calcul en μs comparés entre notre version GPU et le couplage Pace+mp$\mathbb{F}_q$ sur CPU.

	temps calcul μs		débit opérations [k]P/s		
N	**notre impl.**	mp$\mathbb{F}_q$+**Pace**	**notre impl.**	mp$\mathbb{F}_q$+**Pace**	*accélération*
160	179	464	5586	2155	*2.6*
192	304	550	3289	1818	*1.8*
224	507	878	1972	1138	*1.7*
256	617	1003	1620	997	*1.6*
384	4609	2941	216	340	*0.6*

Table 2.8.: Résultats comparés pour la multiplication scalaire sur GPU et le couplage Pace+mp$\mathbb{F}_q$ sur CPU.

Récapitulatif

Conférences : SPIE 2007 [35], PARCO 2009 [30]
Collaboration : LIRMM 2008+
Logiciels : bibliothèque PACE (C++ CPU et GPU), licence LGPL

2.14. Maîtrise des erreurs d'arrondi dans les outils de CAO

Lors de la conception d'une architecture de calcul en virgule fixe, optimiser la taille d'un chemin de données puis valider sa qualité numérique est un problème fréquent. Il faut souvent arrondir (ou tronquer[18]) les valeurs intermédiaires afin de limiter la taille des circuits et la consommation d'énergie. L'erreur d'arrondi est faible pour une opération seule. Mais dans une séquence d'opérations, ces erreurs peuvent s'accumuler pour former des erreurs très importantes, parfois même en dégradant totalement le résultat. Jusqu'ici, il existe assez peu d'outils pour aider un concepteur de circuit à essayer de maîtriser les erreurs d'arrondi.

La validation de la qualité numérique est aussi problématique. Comment vérifier, et si possible garantir, que le bloc arithmétique conçu calcule effectivement la spécification mathématique avec une certaine précision ? Cette validation n'est souvent faite que très partiellement en recourant à des simulations. Ici aussi déterminer les vecteurs d'entrée qui conduisent systématiquement aux erreurs les plus importantes est un problème très difficile.

Borner finement les erreurs d'arrondi est, en général, assez complexe. Différentes méthodes ont été proposées pour le calcul flottant comme l'arithmétique stochastique [77] ou l'analyse statique [90]. Des travaux remarquables récents nous offrent des éléments de solution à ces problèmes (optimisation et validation). Dans sa thèse de doctorat [108], Guillaume Melquiond a proposé une méthode, et un outil nommé `gappa`, pour calculer des bornes très fines des erreurs d'arrondi. En injectant dans cet outil une description du calcul effectué et d'une formulation de

18. La troncature est un mode d'arrondi possible parmi d'autres (au plus près, vers $\pm\infty$, etc).

l'erreur recherchée, on a en sortie un encadrement (au sens de l'arithmétique d'intervalles) de cette erreur. Les bornes d'erreur retournées par `gappa` sont bien plus fines que ce que l'on peut calculer à la main. Nous l'utilisons dans de nombreux cas, par exemple pour dimensionner des opérateurs de calcul pour des approximations polynomiales (cf. chapitre 3).

Je développe une bibliothèque SystemC qui permet de calculer quasi automatiquement des erreurs d'arrondi en utilisant `gappa`. SystemC est un ensemble de classes et macros C++ pour la modélisation, la vérification fonctionnelle et la synthèse de haut niveau en matériel. SystemC supporte des types de représentation des données en virgule fixe, en complément à 2, pour les implantations matérielles. On a, par exemple, la classe `sc_fixed<`*wl*`,` *iwl*`,` *q_mod*`, ...>` où *wl* est la taille totale du nombre, *iwl* la taille de sa partie entière et *q_mod* le mode d'arrondi (ou quantification). Il y a une classe équivalente pour les nombres non signés : `sc_ufixed`. Une description SystemC qui utilise des types `sc_[u]fixed` peut être compilée en surchargeant certains types et en utilisant certaines techniques de méta-programmation *template*. Une description `gappa` du calcul est alors générée automatiquement. Il faut ensuite spécifier les erreurs à borner et appeler la fonction qui exécute un appel système vers `gappa` et analyse le résultat. Des extensions de cette bibliothèque, encore très embryonnaire, sont en cours d'étude afin d'offrir des techniques d'optimisation du dimensionnement de la taille de chemins de données et des facilités d'exploration architectures de calcul en virgule fixe.

Dans une collaboration naissante avec le laboratoire VLSI-CAD de l'Université du Massachusetts à Amherst, USA, nous regardons avec Daniel Gomez-Prado et Maciej Ciesielski comment intégrer dans l'outil TDS, développé dans ce laboratoire, le dimensionnement de données en virgule fixe à travers un interfaçage avec `gappa`.

Récapitulatif	
	Collaboration : Université du Massachusetts à Amherst, USA
	Logiciel : bibliothèque SystemC gestion erreurs d'arrondi pour virgule fixe

2.15. Générateurs aléatoires TRNG

Avec Renaud Santoro et Olivier Sentieys, nous travaillons sur la génération de nombres véritablement aléatoires [123, 124]. J'ai rejoint l'équipe qui travaillait sur ce thème à mon arrivée à l'IRISA. Nous étudions des TRNG (*true random number generator*) avec évaluation en ligne de la qualité de l'aléa. Un TRNG est un dispositif qui utilise une source de bruit physique, réputée réellement aléatoire, pour fournir des séquences aléatoires de bits. Différentes sources de bruit physique ont un comportement véritablement aléatoire : métastabilité de signaux, désintégration radioactive, bruit thermique, variations de la gigue temporelle dans des oscillateurs libres, etc.

Nous utilisons des TRNG à base d'échantillonnage d'oscillateurs en anneaux. Les signaux produits par plusieurs oscillateurs en anneaux, qui oscillent librement, sont combinés entre eux. Le signal ainsi formé est échantillonné à une fréquence donnée (que l'on va essayer de maximiser). La gigue aléatoire présente dans les différents oscillateurs influence le signal échantillonné. Ce dernier est une combinaison, plus ou moins complexe, d'un signal véritablement aléatoire et de signaux déterministes, mais hautement complexes, liés au TRNG lui-même et à son environnement. Une étape de post-traitement permet de séparer, dans une certaine mesure, la composante aléatoire de la composante déterministe.

La qualité de l'aléa en sortie d'un TRNG dépend beaucoup de la qualité de la source de bruit physique mais aussi de nombreux paramètres de la structure du TRNG et de son environnement de fonctionnement comme la température, les variations de la tension d'alimentation, des perturbations électromagnétiques autour du TRNG, des bruits dans le substrat du circuit, etc. Dans

un circuit intégré, des blocs proches du TRNG peuvent influencer la fréquence des oscillateurs en anneaux (qui devraient être totalement libres). Si un couplage fort s'opère, alors la composante aléatoire de la gigue diminue. La composante aléatoire du TRNG peut ne plus être suffisante ou même exploitable. Des attaques spécifiques peuvent être envisagées pour dégrader la qualité de l'aléa et ainsi fragiliser certains systèmes comme en cryptographie (génération de clés, remplissage aléatoire de blocs, chiffrement par flot, etc.). Ceci implique qu'il faut non seulement vérifier la conception des TRNG, mais aussi vérifier la qualité de l'aléa produit pendant leur fonctionnement. C'est ce sur quoi nous travaillons.

Nous implantons en matériel des tests statistiques pour évaluer en ligne (*in situ* et en temps réel) le caractère aléatoire des séquences produites par un TRNG. Pour ces tests statistiques, nous utilisons le test de Maurer [105] et les batteries de tests AIS 31 [67, 96] et FIPS 140-2 [114]. Dans un article publié à la conférence SPIE en 2009 [48], nous détaillons des approximations de fonctions spécifiques aux tests de distributions statistiques nécessaires. Nous avons implanté ces approximations en FPGA. Nous sommes les premiers à avoir pu implanter des TRNG et une analyse en ligne de la qualité de l'aléa totalement en matériel.

Mon implication sur cette thématique est croissante dans le temps. Initialement limitée à la partie mathématique des tests, elle est maintenant plus importante sur les autres aspects. Nous travaillons à la rédaction de plusieurs articles de journaux et espérons pouvoir commencer un livre sur ce thème. Nous pensons utiliser nos résultats pour réaliser des générateurs hybrides qui combinent un TRNG et un PRNG (pour *pseudo random number generator*). Un PRNG est un algorithme déterministe qui fournit une séquence pseudo aléatoire mais avec un très haut débit. Le TRNG est utilisé pour modifier l'initialisation du ou des germes du PRNG. Le couplage des deux types de générateurs permet d'avoir à la fois une séquence produite avec une très bonne qualité d'aléa et avec un débit très élevé.

Ce travail sera aussi utile pour certains aspects de la sécurisation d'opérateurs arithmétiques pour la cryptographie. Pour pouvoir utiliser des recodages des nombres avec une certaine part d'aléa dans le choix des chiffres de représentations redondantes (cf. section 2.11.3), des générateurs aléatoires seront nécessaires et si possible de bonne qualité.

Récapitulatif	Conférence : SPIE 2009 [48]

2.16. Arithmétique par estimation

Dans une collaboration avec Braden Philipps (Université d'Adelaide, Australie), nous étudions des opérateurs de calcul en arithmétique à estimation. L'arithmétique à estimation (ou *estimated arithmetic*) est une arithmétique où certaines valeurs intermédiaires sont négligées [95, 119]. Par exemple, on ne propage pas toutes les retenues intermédiaires d'un additionneur. Les circuits qui utilisent l'arithmétique à estimation sont plus petits, plus rapides et consomment moins que leurs équivalents plus précis. Toute la difficulté du problème réside dans la sélection des valeurs à négliger afin de simplifier le circuit sans trop pénaliser la précision. L'arithmétique à estimation donne des bons résultats dans des applications où une faible erreur moyenne est demandée tout en autorisant des erreurs importantes de temps en temps. Par exemple, l'article [119] présente un cas de décodeur LDPC (*low density parity check decoder*) avec des réductions de 23 % du délai, de 8 % de la surface et de 11 % en consommation sans dégrader la qualité du décodage.

J'ai étudié l'impact de l'arithmétique à estimation sur des opérateurs d'évaluation de fonctions par approximations polynomiales. Les coefficients de polynômes sont générés par la méthode

présentée au chapitre 3. Mais l'évaluation des polynômes utilise des additionneurs et multiplieurs à estimation. Les premiers résultats ont été publiés à la conférence Asilomar en 2009 [59]. On obtient des améliorations de 20 à 60 % en vitesse et de 15 à 30 % en surface sans trop pénaliser l'erreur moyenne. Une des difficultés est de pouvoir caractériser les erreurs moyennes et maximales obtenues sans devoir procéder à de longues simulations. Nous pensons aborder ce problème prochainement ainsi que la conception d'une bibliothèque d'opérateurs en arithmétique à estimation.

Récapitulatif	
Conférence :	Asilomar 2009 [59]
Collaboration :	Université Adelaide Australie
Logiciel :	générateur de code VHDL

Chapitre 3

Opérateurs arithmétiques matériels pour l'évaluation de fonctions par approximation polynomiale

Ce chapitre décrit un travail sur l'optimisation d'opérateurs arithmétiques matériels pour l'évaluation de fonctions en utilisant des polynômes d'approximation. Il reprend, en partie, le contenu et la structure de l'article [45] paru dans le journal « Technique et Science Informatiques » en 2008 et que j'ai rédigé en grande partie. Une version préliminaire a été publiée à la conférence Sympa en 2006 [44]. Des références, descriptions, variations et comparaisons à d'autres travaux autour dans ce thème ont été ajoutées tout au long de ce chapitre.

Le choix de détailler ce travail bien plus que les résumés présentés précédemment me semble justifié pour plusieurs raisons. Ce travail illustre très bien mes travaux sur les différentes facettes de l'arithmétique des ordinateurs présentées en introduction. J'ai été fortement impliqué dans toutes les étapes de ce travail : recherches, développements logiciels et matériels, expérimentations et valorisation. Je n'utilise pas les résultats de mes étudiants mais bien mes propres résultats.

3.1. Introduction

Les systèmes sur puces embarquent de plus en plus de calculs. En plus des additionneurs et multiplieurs rapides, le support matériel d'opérations évoluées est nécessaire pour certaines applications spécifiques. Des applications en traitement du signal et des images effectuent des divisions, inverses, racines carrées ou racines carrées inverses. D'autres applications utilisent des sinus, cosinus, exponentielles ou logarithmes comme en génération numérique de signaux [78].

Différents types de méthodes sont proposés dans la littérature pour l'évaluation de fonctions. On trouve dans [83] les bases d'arithmétique des ordinateurs et en particulier les opérations comme la division, l'inverse, la racine carrée et la racine carrée inverse. Dans [113], les principales méthodes d'évaluation des fonctions élémentaires (sin, cos, exp, log, etc.) sont présentées.

Parmi toutes les méthodes possibles, les approximations polynomiales sont souvent utilisées. Elles permettent d'évaluer efficacement la plupart des fonctions que l'on trouve dans les applications embarquées sur des SoC. L'évaluation pratique d'un polynôme s'effectue en utilisant des additions et des multiplications. La surface des multiplieurs est souvent très importante dans le circuit. Ceci limite le degré du polynôme d'approximation, ou alors, il faut recourir à des opérateurs qui effectuent plusieurs itérations pour parvenir au résultat final.

En section 3.2, nous présentons les notations, l'état de l'art du domaine et en particulier un outil récent permettant de borner finement les erreurs : gappa [107]. Dans ce travail, nous montrons comment obtenir des opérateurs spécifiques pour l'évaluation de fonctions par des approximations polynomiales optimisées. La méthode, proposée en section 3.3, intervient à deux niveaux

pour réaliser cette optimisation. À partir du meilleur polynôme d'approximation théorique, nous déterminons des coefficients propices à une implantation matérielle efficace. Ensuite, nous montrons comment réduire la taille des valeurs intermédiaires en utilisant GAPPA. Notre méthode permet de construire des approximations polynomiales à la fois efficaces et garanties numériquement à la conception. Dans la plupart des méthodes précédentes, seule l'erreur d'approximation était prise en compte pour l'optimisation. Ici, l'erreur totale, approximation et évaluation, est optimisée. Enfin, nous illustrons en section 3.4 l'impact de nos optimisations pour quelques exemples de fonctions sur des circuits FPGA.

3.2. Notations et état de l'art

3.2.1. Notations

La représentation en virgule fixe est la plus courante pour les applications de traitement du signal et des images. Dans la suite, nous supposons toutes les valeurs représentées en virgule fixe. Une extension de ce travail à la virgule flottante est assez simple en pratique du fait des domaines de départ et d'arrivée des fonctions cibles. La fonction à évaluer est f pour l'argument x du domaine $[a, b]$. L'argument x est dans un format sur n bits tandis que le résultat $f(x)$ est sur m bits, souvent on a $n \approx m$. Le polynôme d'approximation p utilisé pour approcher f est de degré d, ses coefficients sont notés : $p_0, p_1, \ldots, p_d$. On a donc $p(x) = \sum_{i=0}^{d} p_i x^i$. La représentation binaire des valeurs est notée $()_2$, par exemple $(11.01)_2$ représente la valeur décimale 3.25. La notation binaire signée *borrow-save* est aussi utilisée pour les coefficients (cf. section 1.1.1 ou [83]).

3.2.2. Erreurs

L'argument x est considéré comme exact. L'*erreur d'approximation* ϵ_{app} est la distance entre la fonction mathématique f et le polynôme d'approximation p utilisé pour approcher f. Pour déterminer cette erreur, nous utilisons la *norme infinie* définie ci-après et estimée numériquement par MAPLE (fonction `infnorm`) ou SOLLYA [98].

$$\epsilon_{\text{app}} = ||f - p||_\infty = \max_{a \leq x \leq b} |f(x) - p(x)|.$$

Du fait de la précision finie des calculs, des erreurs d'arrondi se produisent au cours de l'évaluation du polynôme p en machine. L'*erreur d'évaluation* ϵ_{eval} est due à l'accumulation des erreurs d'arrondi. Même si l'erreur d'évaluation commise lors d'une seule opération est toute petite (une fraction du poids du dernier bit), l'accumulation de ces erreurs peut devenir catastrophique lors de longues séries de calculs si rien n'est prévu pour en limiter les effets.

L'erreur d'évaluation est très difficile à borner finement [109]. Le problème est lié au fait que cette erreur dépend de la valeur des opérandes. Elle est donc difficile à connaître *a priori*. Souvent, on considère le pire cas d'arrondi pour chaque opération. C'est-à-dire la perte de précision correspondant à la moitié du bit de poids le plus faible (1/2 LSB (*least significant bit*) en arrondi au plus près). La perte de précision est au maximum de 1 LSB en arrondi dirigé. En virgule fixe, ce qui sera notre cas, on arrive à 1/2 LSB de précision d'arrondi en utilisant des techniques de biais. Pour l'évaluation d'un polynôme de degré d en utilisant le schéma de Horner (d additions et d multiplications), cette borne pire cas correspond à une perte de d bits de précision. Nous allons voir dans la suite que nous pouvons faire des approximations bien moins pessimistes en utilisant l'outil GAPPA développé par G. Melquiond (cf. section 3.2.6).

Il est possible de mesurer *a posteriori* l'erreur totale commise pour une valeur particulière de l'entrée x en utilisant $f(x) - \texttt{output}(p(x))$ où $\texttt{output}(p(x))$ est la sortie réelle (physique ou

simulée par un simulateur au niveau bit) de l'opérateur. Cette technique est souvent utilisée pour qualifier *a posteriori* la précision d'opérateurs par des simulations exhaustives ou pour les entrées menant aux plus grandes erreurs [1]. D'une manière générale, cette technique de qualification *a posteriori* des opérateurs de calcul est coûteuse en temps de simulation et de conception s'il faut modifier certains paramètres des opérateurs.

Dans la suite, nous exprimons les erreurs soit directement soit en termes de précision équivalente. Cette dernière est le nombre de bits n_c justes ou corrects équivalents à une certaine erreur ϵ avec $n_c = -\log_2 |\epsilon|$ pour une représentation fractionnaire du type $x_0.x_1x_2 \dots x_n$. Ceci signifie aussi que lorsque l'on a n_c bits justes, l'erreur absolue commise est alors plus petite que 2^{-n_c}. Par exemple, une erreur de 2.3×10^{-3} est équivalente à une précision de 8.7 bits corrects. Calculer avec 12 bits de précision signifie avoir une erreur d'au plus $2^{-12} = 0.00024414$.

3.2.3. Polynôme minimax

Il existe différents types de polynômes d'approximation [113]. Le polynôme *minimax* est, en quelque sorte, le meilleur polynôme d'approximation pour notre problème. Le polynôme minimax de degré d pour approcher f sur $[a, b]$ est le polynôme p^* qui satisfait :

$$||f - p^*||_\infty = \min_{p \in \mathcal{P}_d} ||f - p||_\infty,$$

où $\mathcal{P}_d$ est l'ensemble des polynômes à coefficients réels de degré au plus d. Ce polynôme peut être déterminé numériquement grâce à l'algorithme de Remes [121] (implanté dans la fonction MAPLE `minimax` ou dans l'outil très efficace SOLLYA). Le polynôme minimax est le meilleur car parmi tous les polynômes d'approximation possibles de f sur $[a, b]$, il est celui qui a la plus petite des erreurs maximales ϵ_{app} sur tout l'intervalle. On note ϵ^*_{app} l'erreur d'approximation du polynôme minimax avec $\epsilon^*_{\text{app}} = ||f - p^*||_\infty$.

Mais le polynôme minimax est un polynôme « théorique » car ses coefficients sont dans $\mathbb{R}$ et on suppose son évaluation exacte (c'est-à-dire faite avec une précision infinie). Pour pouvoir évaluer ce polynôme en pratique, il faut faire deux niveaux d'approximation supplémentaires dont les erreurs peuvent se cumuler avec l'erreur d'approximation théorique ϵ^*_{app} :

- les coefficients de p^* devront être écrits en précision finie dans le format supporté par le circuit ;
- les calculs nécessaires à son évaluation seront effectués en précision finie.

Actuellement, il n'existe pas de méthode théorique permettant d'obtenir le meilleur polynôme d'approximation en pratique (coefficients *et* évaluation dans la précision du format cible). Mais nous allons montrer dans la suite que d'importantes optimisations sont déjà possibles avec une méthode assez simple et des outils facilement accessibles. La figure 3.1 illustre la problématique de l'implantation pratique de polynômes d'approximation en matériel.

3.2.4. Vers des polynômes d'approximation avec des coefficients représentables

L'un des problèmes pour l'implantation pratique d'approximations polynomiales est le fait que les coefficients du polynôme minimax ne sont pas représentables dans le format cible en précision finie. Ceci est vrai aussi pour les autres types d'approximations polynomiales (Chebyshev, Legendre, etc.), cf. [113].

Par exemple, prenons la fonction e^x sur l'intervalle $[1/2, 1]$ avec un polynôme de degré 2 et des coefficients sur 10 bits dont 1 en partie entière, soit $x = (x_0.x_1x_2 \dots x_9)_2$. Le polynôme

1. Ces entrées particulières étant très difficiles à déterminer dans le cas général.

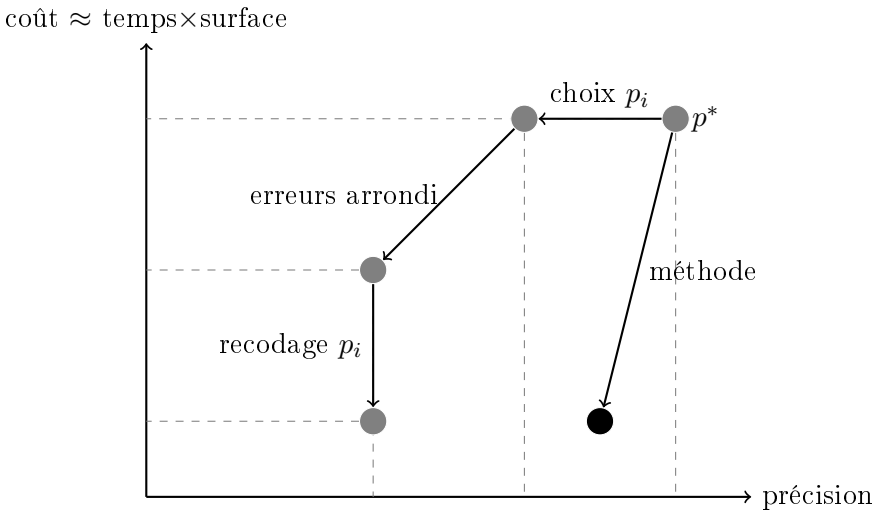


FIGURE 3.1.: Résumé du problème d'implantation de polynômes d'approximation en matériel

minimax trouvé est [2] $1.116018832 + 0.535470996x + 1.065407154x^2$, il conduit à une précision d'approximation de 9.4 bits environ. C'est la meilleure précision atteignable avec un polynôme de degré 2 sur $[1/2, 1]$. En arrondissant au plus près les 3 coefficients de ce polynôme dans le format cible, on a alors $\frac{571}{512} + \frac{137}{256}x + \frac{545}{512}x^2$ et une précision de l'approximation de 8.1 bits. Après avoir testé tous les modes d'arrondi possibles pour chacun des coefficients dans le format cible, on trouve que le meilleur polynôme est $\frac{571}{512} + \frac{275}{512}x + \frac{545}{512}x^2$ pour une précision de l'approximation de 9.3 bits, soit un gain de 1.2 bit de précision. Il semble donc que le choix des coefficients représentables dans le format cible soit un élément important pour obtenir des opérateurs précis.

Dans les dernières années, différents travaux de recherche ont été menés pour obtenir des « bons » polynômes d'approximation dont les coefficients soient exactement représentables dans le format cible. Nous avons participé à ces travaux en utilisant différentes méthodes : soit en déterminant d'assez bons polynômes à l'aide d'explorations très simples (cf. section 3.3 plus bas), soit en essayant de déterminer le ou les meilleurs polynômes d'approximations mais cette fois avec des méthodes bien plus complexes et coûteuses (cf. section 3.2.5 ci-dessous). Aujourd'hui, des outils comme SOLLYA fournissent des résultats efficaces à la fois en précision et en temps de calcul pour déterminer les polynômes. Ce qui est présenté ci-dessous a été fait avant le développement, récent, de ces outils. De plus, les outils comme SOLLYA ne prennent pas en compte les erreurs d'arrondi ni les contraintes d'implantation.

Avec Nicolas Veyrat-Charvillon et Romain Michard, nous avons étudié le cas de polynômes où pour les coefficients de degré supérieur à un, il y a, au plus, trois chiffres non nuls. Ceci permet d'utiliser les multiplications tronquées et approximations des puissances entières présentées, respectivement, aux sections 2.6.2 et 2.6.3. Les résultats de ce travail ont été publiés dans la conférence ASAP 2005 [41]. Nous avons reçu le prix du meilleur papier de cette conférence.

3.2.5. Génération des meilleurs approximants

Avec Nicolas Brisebarre et Jean-Michel Muller, nous avons proposé, dans [14], une méthode basée sur une formulation en un problème de programmation linéaire en nombres entiers. Les coefficients des polynômes sont représentés par des rationnels dont le dénominateur est une

2. En arrondissant les coefficients théoriques sur 10 chiffres décimaux soit environ 32 bits.

puissance de 2 (2^n pour un format fractionnaire sur n bits). Un polytope est construit à partir des contraintes (les tailles) sur tous les coefficients du polynôme. Chaque point du polytope est alors un polynôme dont les coefficients s'écrivent avec les tailles souhaitées. Mais seuls certains de ces polynômes sont des « bons » polynômes d'approximation de f. La liste des polynômes résultats est filtrée en deux phases. L'erreur d'approximation est mesurée en un certain nombre de points d'échantillonnage pour chaque point (polynôme) possible du polytope. Le parcours du polytope donne une liste de polynômes pour lesquels la distance entre la fonction et le polynôme testé est plus petite qu'un seuil pour chacun des points d'échantillonnage. Cette première liste de polynômes est ensuite traitée pour déterminer numériquement les normes infinies entre chacun des polynômes de la liste et la fonction. Cette méthode, séduisante sur le plan théorique, présente des temps de calcul et le volume de mémoire nécessaires importants.

Dans [13] et [15], avec Serge Torres, nous utilisons la méthode présentée dans [14] pour produire des approximations polynomiales avec des coefficients creux (avec beaucoup de 0) pour des petits degrés (3 ou 4). Toutefois, du fait des énormes besoins en temps de calcul et en mémoire de la méthode de [14], les approximations trouvées sont limitées à une douzaine de bits de précision. De plus, seule l'erreur d'approximation est prise en compte.

Ce travail s'est fait dans le cadre d'une collaboration entre l'équipe Arénaire et le laboratoire LArAL de l'Université de St-Étienne. Elle était soutenue financièrement par le projet GAAP *Génération Automatique d'Approximants Polynomiaux efficaces en machine* de l'ACI *Nouvelles Interfaces des Mathématiques* pour la période 2004–2007.

3.2.6. Outils pour le calcul de bornes d'erreur globale

L'étude de méthodes pour borner finement les erreurs de calcul est un domaine de recherche très actif depuis quelques années. Dans le domaine du traitement du signal, des méthodes assimilant les erreurs d'arrondi à du bruit sont utilisées avec succès [109]. Différents outils ont été développés pour le calcul scientifique en virgule flottante. Par exemple, FLUCTUAT [90] est un analyseur statique de code qui permet de détecter certaines pertes de précision sur des programmes flottants. Le logiciel CADNA [77] implante une méthode stochastique pour l'analyse de la précision moyenne des programmes flottants ou en virgule fixe.

Le logiciel GAPPA [107], développé par Guillaume Melquiond, permet d'évaluer et de prouver (en utilisant l'assistant de preuves COQ) des propriétés mathématiques sur des programmes numériques. La caractéristique intéressante de GAPPA dans notre problème est sa capacité à borner finement les erreurs de calcul ou à montrer que des bornes sont en dessous d'un certain seuil.

Voici un exemple simple des possibilités de GAPPA sur la fonction e^x sur $[1/2, 1]$. On suppose tous les calculs effectués en virgule fixe sur 10 bits dont 1 en partie entière. Le polynôme d'approximation utilisé est $p(x) = \frac{571}{512} + \frac{275}{512}x + \frac{545}{512}x^2$, son évaluation se décrit ainsi en GAPPA :

```
1 p0 = 571/512; p1 = 275/512; p2 = 545/512;
2 x = fixed<-9,dn>(Mx);
3 x2 fixed<-9,dn>= x * x;
4 p fixed<-9,dn>= p2 * x2 + p1 * x + p0;
5 Mp = p2 * (Mx*Mx) + p1 * Mx + p0;
6 { Mx in [0.5,1] /\ |Mp-Mf| in [0,0.001385]
7   -> |p-Mf| in ? }
```

La ligne 1 spécifie les coefficients du polynôme (choisis représentables dans le format considéré). Par convention dans la suite, les noms de variables qui commencent par un `M` majuscule

représentent les valeurs mathématiques (en précision infinie), et toutes les valeurs en minuscules représentent des variables du programme (des entrées/sorties ou des registres intermédiaires). La ligne 2 indique que `x` est la version circuit de l'argument mathématique `Mx` (en précision infinie). La construction `fixed<-9,dn>` indique que l'on travaille en virgule fixe avec le LSB de poids 2^{-9} et l'arrondi vers le bas `dn` (troncature). La ligne 3 indique que la variable `x2` est la version calculée dans le circuit de x^2. La ligne 4 décrit comment le polynôme est évalué en pratique dans le circuit tandis que la ligne 5 décrit son évaluation théorique (en précision infinie car sans l'opérateur d'arrondi `fixed<.,.>`). Enfin, les lignes 6 et 7 indiquent la propriété cherchée (entre accolades). Les hypothèses sont en partie gauche du signe `->`, elles indiquent que la valeur mathématique de x est dans $[1/2, 1]$ et que l'erreur d'approximation entre le polynôme d'approximation `Mp` (sans erreur d'arrondi) et la fonction mathématique `Mf` (la fonction cible théorique sans aucune erreur) est inférieure ou égale à 0.001385 (valeur fournie par MAPLE). La partie droite du signe `->` indique que l'on demande à GAPPA de nous dire dans quel intervalle (`in ?`) est la distance entre la valeur évaluée dans le circuit du polynôme `p` et la fonction mathématique, en incluant erreurs d'approximation et d'évaluation (arrondis).

Le résultat retourné par GAPPA (version supérieure ou égale à `0.6.1`) pour ce calcul est :

```
Results for Mx in [0.5, 1] and |Mp - F| in [0, 0.001385]:
|p - F| in [0, 232010635959353905b-64 {0.0125773, 2^(-6.31303)}]
```

Après avoir répété les hypothèses, GAPPA indique qu'il a trouvé une borne pour $|p-f|$ et qu'il y a au moins 6.31 bits corrects.

En modifiant les lignes 3 à 5 par les lignes suivantes, on cherche l'erreur totale commise en utilisant le schéma de Horner, GAPPA indique alors une précision globale de 6.57 bits.

```
3 y1 fixed<-9,dn>= p2 * x + p1;
4 p  fixed<-9,dn>= y1 * x + p0;
5 Mp = (p2 * Mx + p1) * Mx + p0;
```

3.3. Méthode d'optimisation proposée

Dans un premier temps, nous présentons rapidement la méthode, ensuite nous donnons des détails sur chacune des différentes étapes et les informations sur les outils nécessaires.

Les données nécessaires en entrée de la méthode sont :
- f la fonction à évaluer ;
- $[a, b]$ le domaine de l'argument x ;
- le format de l'argument x (nombre de bits n_x) ;
- μ l'erreur totale maximale cible (donnée en erreur absolue).

Les paramètres déterminés par la méthode sont :
- d le degré du polynôme à utiliser ;
- p_0, p_1, p_2,..., p_d les valeurs des coefficients du polynôme représentables dans le circuit ;
- n la taille utilisée pour représenter les coefficients ;
- n' la taille utilisée pour effectuer les calculs[3].

Notre méthode se résume aux 3 étapes ci-dessous avec des retours possibles à une étape antérieure (rebouclage) dans certains cas :

3. Nous verrons que dans certains cas, prendre n et n' légèrement différents peut aider à limiter la taille du circuit.

Étape 1 : calcul du *polynôme minimax*

On cherche p^* de degré d le plus petit possible tel que $\epsilon^*_{\text{app}} < \mu$ avec $\epsilon^*_{\text{app}} = ||f - p^*||_\infty$. Cette première phase donne l'erreur d'approximation ϵ^*_{app} minimale atteignable en supposant tous les calculs faits et coefficients représentés en précision infinie.

Étape 2 : détermination des *coefficients* du polynôme à implanter et de leur *taille*

On cherche ici à la fois les p_i et leur taille minimale n telle que l'erreur d'approximation ϵ_{app} du polynôme p utilisé ($p(x) = \sum_{i=0}^{d} p_i x^i$) soit strictement inférieure à μ.

Étape 3 : détermination de la *taille du chemin de données*

On cherche n' la taille minimale du chemin de données de l'étage de Horner pour effectuer les calculs. Cette dernière phase donne l'erreur d'évaluation ϵ_{eval} qui intègre les erreurs d'approximation et les erreurs d'arrondi. La valeur de n' trouvée garantit que $\epsilon_{\text{eval}} < \mu$.

Dans certains cas, il n'y a pas de solution à une étape. Il faut alors *reboucler* à l'étape précédente pour essayer d'autres solutions.

Différents types de rebouclages sont possibles. Par exemple, en fin de deuxième étape, on peut ne pas trouver des coefficients représentables pour garantir $\epsilon_{\text{app}} < \mu$. Ceci se produit lorsque la « marge » entre ϵ^*_{app} et μ était trop faible à la première étape. Il faut donc retourner à la première étape pour essayer un polynôme de degré plus grand $d \leftarrow d + 1$.

Un autre type classique de rebouclage intervient à la fin de la dernière étape. Si la taille n' trouvée est jugée trop grande devant n, il peut être intéressant de revenir à la deuxième étape pour essayer un n plus grand. Ceci permet d'avoir ϵ_{app} plus petit et donc plus de marge pour les erreurs d'arrondi.

La décision d'application de ces rebouclages est purement heuristique. Nous allons l'illustrer sur les exemples en section 3.4.

Dans les descriptions données jusqu'ici, nous utilisons des contraintes du style $\epsilon_{\text{app}} < \mu$ et pas $\epsilon_{\text{app}} \leq \mu$. En pratique, il faut de la marge entre ϵ^*_{app} et μ puis entre ϵ_{app} et μ. Le caractère strict des contraintes pour les étapes 1 et 2 est donc nécessaire. Pour la dernière étape, peut-être pouvons-nous trouver une fonction (probablement triviale et donc peu intéressante) telle que l'on ait $\epsilon_{\text{eval}} = \mu$ à la fin. Cela nous semble très improbable. De plus, l'utilisateur saurait traiter convenablement ce cas.

3.3.1. Calcul du polynôme minimax

Dans cette étape on utilise la fonction `minimax` de Maple. On commence avec $d = 1$, et on incrémente d jusqu'à ce que le polynôme minimax p^* trouvé soit tel que $\epsilon^*_{\text{app}} < \mu$.

Voici un exemple pour $f = \log_2(x)$ avec x dans $[1, 2]$:

```
1 > minimax(log[2](x),x=1..2,[1,0],1,'err'); -log[2](err);
2     -.95700010+1.0000000*x
3     4.5371245
4 > minimax(log[2](x),x=1..2,[2,0],1,'err'); -log[2](err);
5     -1.6749034+(2.0246817-.34484766*x)*x
6     7.6597968
7 > minimax(log[2](x),x=1..2,[3,0],1,'err'); -log[2](err);
8     -2.1536207+(3.0478841+(-1.0518750+.15824870*x)*x)*x
9     10.616152
```

Les lignes qui commencent par un signe supérieur (« prompt » Maple) sont les commandes entrées par l'utilisateur. Les résultats retournés par Maple sont en italique. La ligne 1 signifie que l'on cherche un polynôme de degré 1 et que l'erreur d'approximation trouvée sera placée

dans la variable `err`. Les lignes 2 et 3 représentent respectivement le polynôme trouvé et sa précision (en nombre de bits corrects).

Cette étape fournit trois éléments nécessaires pour la suite :
- d le degré du polynôme ;
- $p^*(x) = \sum_{i=0}^{d} p_i^* x^i$ le polynôme minimax (théorique) ;
- ϵ_{app}^* l'erreur *minimale* atteignable en utilisant p^* pour approcher f (en précision infinie).

Les deux autres étapes vont dégrader la qualité de l'approximation (*i.e.* fournir des erreurs plus grandes que ϵ_{app}^*). Il faut donc laisser un peu de marge entre ϵ_{app}^* et μ. Nous reviendrons sur cette marge.

Nous verrons dans l'exemple 3.4.2, que certains changements de variables permettent d'obtenir des coefficients d'ordres de grandeur voisins. Ceci limite les recadrages en virgule fixe. Si on évalue $f(x)$ sur $[a, b]$ avec $a \neq 0$, il peut être intéressant de considérer $f(x + a)$ sur $[0, b - a]$.

Toutes les fonctions ne sont pas « approchables facilement » avec des polynômes. Nous renvoyons le lecteur aux ouvrages de références sur l'évaluation de fonctions comme [113, chap. 3] pour les fonctions élémentaires. En pratique, les fonctions usuelles s'approchent bien par des polynômes.

3.3.2. Détermination des coefficients du polynôme et de leur taille

Une fois p^* déterminé, il faut trouver des coefficients représentables en précision finie. On cherche n le nombre de bits du format virgule fixe des p_i tel que n soit le plus petit possible mais avec $\epsilon_{\text{app}} < \mu$.

Nous avons vu en 3.2.4 que le choix des coefficients est important. En arrondissant simplement les coefficients du polynôme minimax sur n bits, il est peu probable de trouver un bon polynôme d'approximation. L'exemple présenté en 3.2.4 montre que tester l'ensemble des combinaisons des arrondis des coefficients de p^* permet de trouver un bon polynôme. C'est ce que nous proposons de faire systématiquement dans cette deuxième phase.

Chaque coefficient p_i^* du polynôme peut être arrondi soit vers le haut $p_i = \triangle(p_i^*)$, soit vers le bas $p_i = \triangledown(p_i^*)$. Il y a 2 choix possibles par coefficient et donc 2^{d+1} combinaisons à tester au total comme illustré en figure 3.2. Pour chaque polynôme p, il faut déterminer $\epsilon_{\text{app}} = ||f - p||_\infty$ (fonction `infnorm` de Maple).

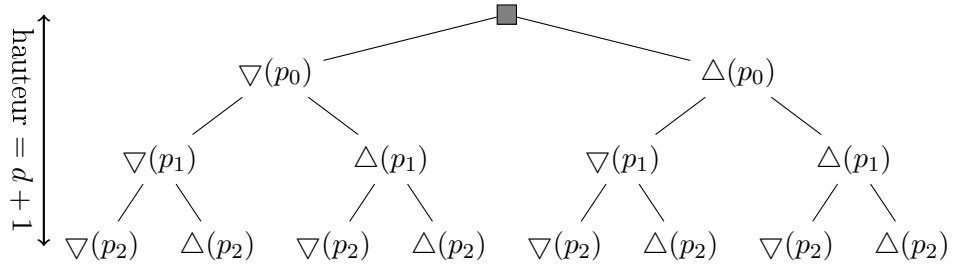


Figure 3.2.: Différentes combinaisons d'arrondi pour p^* de degré $d = 2$

Dans nos applications, d est faible ($d \leq 6$). Il y a donc au mieux quelques centaines de polynômes à tester. En pratique, chaque calcul de ϵ_{app} dure quelques fractions de seconde sur un ordinateur standard.

Nous avons fait un programme Maple qui teste les 2^{d+1} combinaisons d'arrondi des p_i^*. Le programme retourne la liste des polynômes candidats pour la troisième étape, c'est-à-dire ceux pour qui l'erreur d'approximation ϵ_{app} est minimale. On commence par $n = \lceil -\log_2 |\mu| \rceil$ (le

nombre de bits correspondant à l'erreur μ), puis on teste tous les arrondis des $d+1$ coefficients sur n bits. On recommence en incrémentant n jusqu'à ce que $\epsilon_{\text{app}} < \mu$.

À cette étape, on peut aussi souhaiter modifier plus en « profondeur » certains coefficients. Par exemple, si un coefficient retourné est $0.5002441406 = (0.100000000001)_2$ et que l'on travaille sur un peu plus d'une douzaine de bits fractionnaires, il est peut-être intéressant de fixer ce coefficient à 0.5 pour éliminer une opération. La multiplication par une puissance de 2 se réduit à un décalage. Nous verrons un exemple de ce genre de modification dans l'exemple 3.4.2. Nous ne savons pas encore formaliser simplement ce genre de traitement, mais cela constitue un de nos axes de recherche à travers l'amélioration de la méthode présentée dans [14].

Avec Jean-Luc Beuchat, nous avons utilisé ce même parcours de l'arbre des coefficients arrondis pour produire des opérateurs d'approximation polynomiale en arithmétique en ligne [7].

3.3.3. Détermination de la taille du chemin de données

Le calcul du polynôme $p(x)$ peut s'effectuer en utilisant différents schémas d'évaluation [113]. Dans la suite, nous utilisons uniquement les schémas directs et de Horner :

$$p(x) = \begin{cases} p_0 + p_1 x + p_2 x^2 + \cdots + p_d x^d & \text{direct } d \text{ add.}, d + \lceil \log_2 d \rceil \text{ mul.;} \\ p_0 + x(p_1 + x(p_2 + x(\cdots + x p_d)\cdots)) & \text{Horner } d \text{ add.}, d \text{ mul.} \end{cases}$$

Le schéma de Horner est souvent préféré au schéma direct car il nécessite moins d'opérations et donne souvent une erreur d'évaluation ϵ_{eval} plus faible [113]. Toutefois, dans certains cas particuliers avec des coefficients très creux, le schéma direct peut s'avérer intéressant (cf. exemple en 3.4.2).

La dernière étape permet de spécifier la taille du chemin de données pour l'évaluation suivant le schéma de Horner. L'étage de Horner permet d'évaluer $u \times v + z$. La taille du chemin de données de cet étage est n'. On commence avec $n' = n$ et on augmente n' tant que l'encadrement de l'erreur d'évaluation ϵ_{eval} par GAPPA avec n' bits pour le chemin de données ne donne pas $\epsilon_{\text{eval}} < \mu$. Pour le moment, le codage en GAPPA se fait à la main. Mais le schéma de Horner ou le schéma direct étant les mêmes aux coefficients près, nous avons les programmes GAPPA types pour lesquels il suffit de préciser les valeurs des coefficients et du degré. La boucle sur n' s'effectue en quelques minutes tout au plus sur un ordinateur standard.

La différence entre n' et n est appelée le nombre de bits de garde. Conserver n plus petit que n' permet, par exemple, de limiter la taille mémoire nécessaire pour stocker les coefficients.

Si la taille du chemin de données est un peu supérieure à n (1 à 3 bits), on pourrait revenir à l'étape 2 pour essayer un n plus grand. Notre expérience montre que le n' final change rarement. Par contre si la valeur de n' est bien plus grande que n, alors il peut-être intéressant de reboucler à l'étape 2 avec un n plus grand. Ici encore, nous devons encore travailler pour formaliser ce genre de test.

Avec Romain Michard et Nicolas Veyrat-Charvillon, nous étudié l'implantation matérielle de la E-Méthode en grande base pour évaluer des polynômes. Les résultats ont été publiés dans [40]. Cette méthode permet d'obtenir des opérateurs de petite taille. Mais le domaine de convergence de la E-Méthode impose des contraintes importantes sur les coefficients des polynômes utilisables. Il faudrait utiliser des techniques comme celle présentée dans [75] pour limiter ce problème.

3.3.4. Résumé de la méthode

La figure 3.3 résume graphiquement le fonctionnement de la méthode.

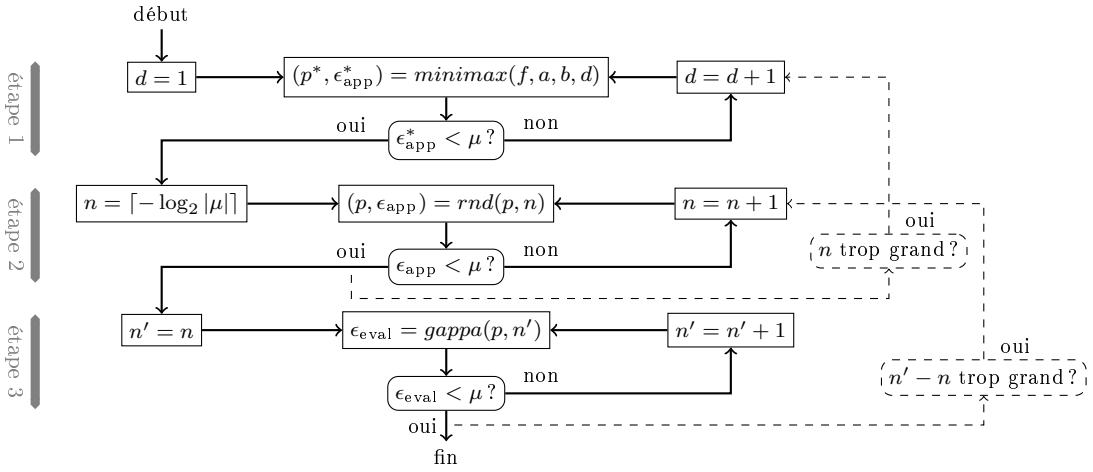


FIGURE 3.3.: Résumé de la méthode (les rebouclages en pointillés sont facultatifs)

3.4. Exemples d'applications sur FPGA

Les implantations réalisées ci-après ont été faites pour des FPGA de la famille Virtex de Xilinx (XCV200-5) avec les outils ISE8.1i de Xilinx. La synthèse et le placement/routage utilisent une optimisation en surface avec effort élevé. Les résultats indiquent toutes les ressources nécessaires pour chaque opérateur (cellules logiques et registres fonctionnels). Nous avons utilisé des circuits FPGA pour ces premières implantations du fait de leur simplicité d'utilisation. Dans l'avenir nous souhaitons optimiser et implanter nos opérateurs pour des circuits intégrés standard (ASIC). La méthode présentée dans ce chapitre est indépendante de la cible d'implantation.

Nous avons utilisé cette méthode dans d'autres cas. Par exemple, dans le projet ANR ROMA, nous avons étudié des opérateurs reconfigurables, cf. publication [55] où nous présentons un opérateur matériel d'approximation de l'inverse avec un polynôme de degré 3 mais qui ne nécessite qu'une seule multiplication de grande taille. Nous avons utilisé une approximation similaire dans la bibliothèque FLIP (cf. section 2.7.2).

3.4.1. Fonction 2^x sur $[0, 1]$

On cherche un opérateur pour évaluer la fonction 2^x, avec x dans $[0, 1]$ et une précision globale de 12 bits. La première étape est de trouver un polynôme d'approximation théorique en utilisant MAPLE. On cherche ici à minimiser le degré d du polynôme utilisé. Voici la précision d'approximation ϵ_{app} (en bits corrects) des polynômes minimax de 2^x pour d variant de 1 à 5 :

d	1	2	3	4	5
ϵ_{app}	4.53	8.65	13.18	18.04	23.15

Bien évidemment, les polynômes de degré 1 et 2 ne sont pas suffisamment précis pour notre but. La solution avec le polynôme de degré 3 conduit à une précision estimée dans le pire cas à 10 bits environ ($13.18 - d$, car on borne à d bits la perte de précision pour un schéma de Horner de degré d) ce qui semble trop faible. De plus, les 13.18 bits corrects correspondent à l'erreur d'approximation avec le polynôme minimax avec des coefficients réels (non représentables dans le format cible).

Sans outil pour aider le concepteur, il semble donc que l'on serait obligé de choisir la solution de degré 4 ou 5. Même la solution avec un polynôme de degré 4 peut conduire à une précision trop

faible. Certes sa précision théorique est au moins de $18.04-4 = 14.04$ bits corrects mais seulement pour des coefficients en précision infinie. Le polynôme minimax de degré 4 est (les coefficients sont affichés avec 10 chiffres décimaux dans le papier, mais MAPLE est capable de les calculer avec plus de précision) : $1.0000037045 + 0.6929661227x + 0.2416384458x^2 + 0.0516903583x^3 + 0.0136976645x^4$.

Pour être certain que la solution de degré 4 peut être employée, il faut trouver un format des coefficients pour lequel le polynôme minimax avec ses coefficients arrondis dans ce format ait une précision supérieure ou égale à $12 + 4 = 16$ bits. Pour un format donné, on teste tous les modes d'arrondi possibles pour les coefficients du polynôme minimax dans le format. On trouve des polynômes acceptables à partir de 14 bits fractionnaires et 1 entier. Pour montrer que le choix de coefficients représentables est important, nous présentons ci-dessous chacune des combinaisons possibles des modes d'arrondi des coefficients et la précision d'approximation du polynôme dont les coefficients s'écrivent exactement dans le format cible. Seuls deux polynômes ont la précision souhaitée (valeurs en gras).

La solution avec le polynôme de degré 4 est donc utilisable moyennant un bon choix des coefficients du polynôme d'approximation. Mais on va montrer que même celle de degré 3 l'est en pratique. Ce qui constitue une optimisation significative de l'opérateur mais nécessite des outils.

$(\nabla, \nabla, \nabla, \nabla, \nabla)$	12.00	$(\nabla, \nabla, \nabla, \nabla, \triangle)$	13.00
$(\nabla, \nabla, \nabla, \triangle, \nabla)$	13.00	$(\nabla, \nabla, \nabla, \triangle, \triangle)$	14.03
$(\nabla, \nabla, \triangle, \nabla, \nabla)$	13.00	$(\nabla, \nabla, \triangle, \nabla, \triangle)$	14.55
$(\nabla, \nabla, \triangle, \triangle, \nabla)$	14.99	$(\nabla, \nabla, \triangle, \triangle, \triangle)$	13.00
$(\nabla, \triangle, \nabla, \nabla, \nabla)$	13.00	$(\nabla, \triangle, \nabla, \nabla, \triangle)$	**16.13**
$(\nabla, \triangle, \nabla, \triangle, \nabla)$	**17.12**	$(\nabla, \triangle, \nabla, \triangle, \triangle)$	13.00
$(\nabla, \triangle, \triangle, \nabla, \nabla)$	15.71	$(\nabla, \triangle, \triangle, \nabla, \triangle)$	13.00
$(\nabla, \triangle, \triangle, \triangle, \nabla)$	13.00	$(\nabla, \triangle, \triangle, \triangle, \triangle)$	12.00
$(\triangle, \nabla, \nabla, \nabla, \nabla)$	13.00	$(\triangle, \nabla, \nabla, \nabla, \triangle)$	13.00
$(\triangle, \nabla, \nabla, \triangle, \nabla)$	13.00	$(\triangle, \nabla, \nabla, \triangle, \triangle)$	13.00
$(\triangle, \nabla, \triangle, \nabla, \nabla)$	13.00	$(\triangle, \nabla, \triangle, \nabla, \triangle)$	13.00
$(\triangle, \nabla, \triangle, \triangle, \nabla)$	12.99	$(\triangle, \nabla, \triangle, \triangle, \triangle)$	12.00
$(\triangle, \triangle, \nabla, \nabla, \nabla)$	12.99	$(\triangle, \triangle, \nabla, \nabla, \triangle)$	12.98
$(\triangle, \triangle, \nabla, \triangle, \nabla)$	12.91	$(\triangle, \triangle, \nabla, \triangle, \triangle)$	12.00
$(\triangle, \triangle, \triangle, \nabla, \nabla)$	12.79	$(\triangle, \triangle, \triangle, \nabla, \triangle)$	12.00
$(\triangle, \triangle, \triangle, \triangle, \nabla)$	12.00	$(\triangle, \triangle, \triangle, \triangle, \triangle)$	11.41

Le polynôme minimax de degré 3 qui approche le mieux théoriquement 2^x sur $[0, 1]$ est :

$$p^*(x) = 0.9998929656 + 0.6964573949x + 0.2243383647x^2 + 0.0792042402x^3.$$

On a alors $\epsilon_{\text{app}} = ||f - p^*||_\infty = 0.0001070344$ soit 13.18 bits de précision. Ceci signifie que, quelle que soit la précision des coefficients utilisés pour représenter p^* et celle utilisée pour son évaluation, on ne pourra pas avoir un opérateur avec une précision meilleure que 13.18 bits.

Afin de déterminer la version représentable de p^* que nous allons implanter, il faut trouver la taille des coefficients. Étant donné la fonction et son domaine, le format cherché est constitué d'un bit de partie entière et $n-1$ bits de partie fractionnaire. On cherche n minimal pour une erreur d'approximation ϵ_{app} correspondante la plus proche possible du maximum théorique de 13.18.

$n-1$	12	13	14	15	16
ϵ_{app}	12.38	12.45	13.00	13.00	13.02
nb. candidats	0	0	2	2	7

En effet, pour $n-1=14$ bits, tous les modes d'arrondis possibles des coefficients donnent :

$(\nabla,\nabla,\nabla,\nabla)$	11.41	$(\nabla,\nabla,\nabla,\triangle)$	12.00
$(\nabla,\nabla,\triangle,\nabla)$	12.00	$(\nabla,\nabla,\triangle,\triangle)$	12.84
$(\nabla,\triangle,\nabla,\nabla)$	12.00	$(\nabla,\triangle,\nabla,\triangle)$	**13.00**
$(\nabla,\triangle,\triangle,\nabla)$	**13.00**	$(\nabla,\triangle,\triangle,\triangle)$	12.36
$(\triangle,\nabla,\nabla,\nabla)$	12.00	$(\triangle,\nabla,\nabla,\triangle)$	12.25
$(\triangle,\nabla,\triangle,\nabla)$	12.23	$(\triangle,\nabla,\triangle,\triangle)$	12.23
$(\triangle,\triangle,\nabla,\nabla)$	12.13	$(\triangle,\triangle,\nabla,\triangle)$	12.12
$(\triangle,\triangle,\triangle,\nabla)$	12.05	$(\triangle,\triangle,\triangle,\triangle)$	11.64

Les deux polynômes candidats sont donc :

$$\frac{8191}{8192}+\frac{2853}{4096}x+\frac{1837}{8192}x^2+\frac{649}{8192}x^3 \qquad \text{et} \qquad \frac{8191}{8192}+\frac{2853}{4096}x+\frac{919}{4096}x^2+\frac{649}{8192}x^3.$$

Tous deux conduisent à une erreur d'approximation de 0.0001220703 (13.00 bits de précision). Il reste maintenant à vérifier que l'évaluation d'au moins un de ces polynômes donne une précision finale d'au moins 12 bits. Voici ci-dessous la précision totale (approximation + évaluation) retournée par GAPPA en utilisant le schéma de Horner et le schéma direct pour évaluer $\frac{8191}{8192}+\frac{2853}{4096}x+\frac{1837}{8192}x^2+\frac{649}{8192}x^3$ pour différentes tailles du chemin de données n' :

n'	14	15	16	17	18	19	20
ϵ_{eval} Horner	11.32	11.93	12.36	12.65	12.81	12.90	12.95
ϵ_{eval} direct	11.24	11.86	12.32	12.62	12.79	12.89	12.94

Les valeurs obtenues pour l'autre polynôme ($p_2=\frac{919}{4096}$) sont équivalentes. Le schéma de Horner présente un comportement légèrement meilleur que l'évaluation directe (qui en plus est plus coûteuse en nombre d'opérations). Il faut un chemin de données sur 16 bits au moins pour obtenir un opérateur avec 12 bits de précision au final et ce à partir d'une approximation avec 13.18 bits de précision.

Passer les coefficients sur 16 bits ne modifie pas beaucoup la précision totale car on voit dans la table qui donne ϵ_{app} en fonction de n que l'on passe de 13.00 à 13.02 bits de précision seulement en passant $n-1$ de 14 à 16 pour l'approximation. Avec des coefficients et un chemin de données sur 16 bits, GAPPA indique une précision de 12.38 bits en évaluant le polynôme $p(x)=\frac{32765}{32768}+\frac{22821}{32768}x+\frac{7351}{32768}x^2+\frac{649}{8192}x^3$.

Deux solutions ont été implantées pour cet opérateur : celle optimisée (degré 3, chemin de données de 16 bits) et celle de base (degré 4, chemin de données de 18 bits). La seconde version correspond à celle que l'on aurait implantée sans l'aide de notre méthode. Le tableau 3.1 donne les différentes caractéristiques des deux implantations pour un étage de Horner (logique et registres). L'optimisation permet d'obtenir un circuit 17 % plus petit mais surtout d'utiliser une approximation de degré 3 plutôt que 4 et donc de gagner 38 % en temps de calcul.

version	surface [*slices*]	période [ns]	nb. cycles	durée du calcul [ns]
degré 3, $n'=16$	193	21.9	3	65.7
degré 4, $n'=18$	233	26.9	4	107.6

TABLE 3.1.: Résultats de synthèse pour 2^x sur $[0,1]$

3.4.2. Racine carrée sur $[1,2]$

Pour ce deuxième exemple, nous cherchons à concevoir un opérateur très rapide pour évaluer $\sqrt{x}$ avec x dans $[1,2]$ et une précision d'au moins 8 bits au final (approximation et évaluation). Le polynôme minimax de degré 1 ne conduit qu'à 6.81 bits de précision, il faut au moins un polynôme de degré 2.

Le polynôme minimax de degré 2 pour $\sqrt{x}$ avec x dans $[1,2]$ est $0.4456804579+0.6262821240x-0.0711987451x^2$. Il fournit une erreur d'approximation théorique de 0.0007638369 soit 10.35 bits corrects. La précision théorique supérieure à 10 bits permet de supposer que l'on devrait atteindre notre but si on trouve des coefficients représentables sans trop diminuer l'erreur d'approximation.

Toutefois, implanter directement ce polynôme, n'est pas une bonne idée du point de vue du format. Avec x dans $[1,2]$, il faut travailler un nombre de bits variable pour la partie entière. En effet, l'opération x^2 nécessite deux bits entiers alors que les autres seulement un. Pour éviter ceci, on utilise un changement de variable pour évaluer $\sqrt{1+x}$ avec x dans $[0,1]$. Le polynôme minimax correspondant est : $1.0007638368 + 0.4838846338x - 0.0711987451x^2$. On obtient la même erreur d'approximation de 10.35 bits corrects. Ceci est tout à fait normal car le changement de variable $x \leftarrow 1+x$ utilisé ne modifie pas la qualité du polynôme minimax.

À partir de ce polynôme, on pourrait procéder comme pour l'exemple 2^x, mais les coefficients p_0 et p_1 semblent très proches de puissances de 2 et on va essayer de l'utiliser. La première chose à faire est de remplacer p_0 par 1. Le polynôme $1 + 0.4838846338x - 0.0711987451x^2$ offre une précision d'approximation de 9.35 bits, ce qui nous semble satisfaisant.

Le coefficient p_1 semble proche de 0.5. Le polynôme $1+0.5x-0.0711987451x^2$ offre une précision d'approximation de 6.09 bits seulement. p_1 ne peut donc pas être remplacé par 0.5. Toutefois nous allons essayer d'écrire p_1 avec peu de bits à 1 ou -1. Le coefficient p_1 est très proche de $(0.10000\overline{1})_2$. Le polynôme $1+(0.10000\overline{1})_2x-0.711987451x^2$ offre une précision d'approximation de 9.45 bits et en plus le produit p_1x est remplacé par la soustraction $\frac{1}{2}x - \frac{1}{2^6}x$.

Nous procédons à une recherche d'une version avec peu de bits non nuls de p_2 et nous trouvons $(0.0001001)_2$. Donc le produit p_2x^2 est remplacé par l'addition $\frac{1}{2^4}x^2 + \frac{1}{2^7}x^2$. Le polynôme $1 + (0.10000\overline{1})_2x+(0.0001001)_2x^2$ fournit une précision d'approximation de 9.49 bits. Il ne reste donc plus qu'une seule multiplication pour le calcul de x^2. De plus, on constate qu'avec un coefficient p_2 moins précis (passage de 0.0711987451 à $(0.0001001)_2$), la qualité de l'approximation est légèrement meilleure. Ceci s'explique car le coefficient p_2 quantifié à $(0.0001001)_2$ permet de compenser les erreurs de la quantification des autres coefficients. L'exploration d'une partie de l'espace des coefficients permet de trouver ce genre de polynôme. Mais encore une fois, il s'agit d'une méthode purement heuristique sans aucune garantie ni sur le résultat ni sur le temps de calcul nécessaire pour potentiellement améliorer le résultat. Pour le moment, nous ne voyons pas de méthode pour trouver automatiquement ces petites améliorations.

On va donc déterminer la précision finale intégrant l'erreur d'évaluation en utilisant GAPPA. En cherchant la taille n' du chemin de données on trouve 10 bits. Le programme à faire prouver par GAPPA est le suivant.

```
1 p0 = 1; p1 = 31/64; p2 = -9/128;
2 x = fixed<-10,dn>(Mx);
3 x2  fixed<-10,dn>= x * x;
4 p   fixed<-10,dn>= p2 * x2 + p1 * x + p0;
5 Mp = p2 * (Mx*Mx) + p1 * Mx + p0;
6 { Mx in [0,1] /\ |Mp-Mf| in [0,0.0013829642]
7   -> |p-Mf| in ? }
```

GAPPA retourne une erreur totale de 8.03 bits. Mais ce programme GAPPA correspond à l'utilisation de multiplieurs pour effectuer les produits $p_1 x$ et $p_2 x^2$. En pratique nous remplaçons ces multiplieurs par des additions/soustractions. Il faut donc donner à GAPPA une description exacte de ce qui est fait par notre architecture. La détermination de la taille minimale du chemin de données est faite en partant de $n = 8$ et en incrémentant n jusqu'à ce que la précision finale soit supérieure ou égale à 8 bits. La recherche donne $n = 13$.

```
p0  = 1;
p11 = 1/2;    p12 = -1/64;
p21 = -1/16; p22 = -1/128;
x = fixed<-8,dn>(Mx);
x2  fixed<-16,dn>= x * x;
p   fixed<-13,dn>= p21 * x2 + p22 * x2 + p11 * x
                     + p12 * x + p0;
Mx2 = Mx * Mx;
Mp  = p21 * Mx2 + p22 * Mx2 + p11 * Mx + p12 * Mx
      + p0;
{ Mx in [0,1] /\ |Mp-Mf| in [0,0.0013829642]
  -> |p-Mf| in ? }
```

Dans ce cas, GAPPA retourne une précision de 8.07 bits avec une seule vraie multiplication pour x^2. L'architecture de l'opérateur est présentée en figure 3.4. Les cercles gris indiquent un décalage vers la droite du nombre de bits indiqué à l'intérieur du cercle (routage uniquement).

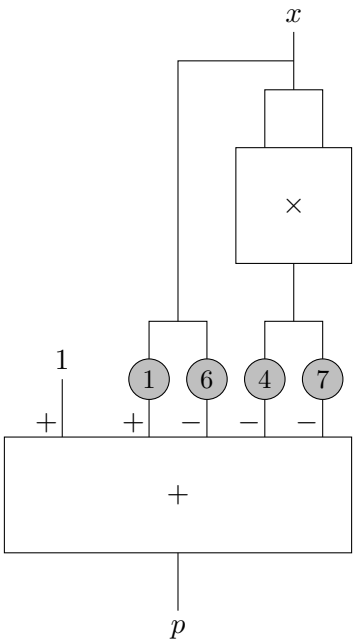


FIGURE 3.4.: Architecture de l'opérateur optimisé pour $\sqrt{1+x}$ sur $[0, 1]$

Nous avons implanté la solution optimisée présentée en figure 3.4 et celle que l'on aurait implantée sans l'aide de la méthode (degré 2, étage de Horner avec chemin de données sur 11 bits). Les résultats de synthèse correspondants sont présentés dans le tableau 3.2. Ici aussi, les

gains sont intéressants puisque l'on obtient une amélioration de 40 % en surface et de 51 % en temps de calcul.

version	surface [*slices*]	période [ns]	nb. cycles	durée du calcul [ns]
degré 2 Horner	103	19.9	2	39.8
degré 2 optimisée	61	19.4	1	19.4

TABLE 3.2.: Résultats de synthèse pour $\sqrt{1+x}$ sur $[0,1]$

3.5. Conclusion et perspectives

Nous avons proposé une méthode pour concevoir et optimiser des opérateurs arithmétiques matériels dédiés à l'évaluation de fonctions par approximation polynomiale. Notre méthode permet, à l'aide d'outils récents, de déterminer une solution avec :

- un degré d petit ;
- une taille de coefficients représentables n petite ;
- et une taille du chemin de données n' petite.

La méthode ne fournit pas des valeurs de d, n et n' optimales. L'optimum pour ce problème n'est pas connu actuellement même sur le plan théorique. Sur les exemples testés, la méthode permet d'obtenir des circuits plus petits et plus rapides que ceux que l'on pouvait concevoir avant. Toutefois, il reste beaucoup à faire pour les améliorer encore.

De plus, notre méthode permet, grâce à l'utilisation du logiciel GAPPA [107], d'obtenir des opérateurs valides numériquement dès la conception. En effet, la méthode permet de déterminer à la fois les erreurs d'approximation et les erreurs d'évaluation. Il n'est plus nécessaire de qualifier *a posteriori* le circuit avec de longues simulations ou tests. La contrainte de précision est vérifiée à la conception.

Dans l'avenir, nous pensons travailler dans plusieurs directions. Par exemple, utiliser en matériel d'autres schémas d'évaluation de polynômes comme celui proposé par Estrin pour certaines valeurs de d. De plus, toutes les heuristiques présentées pour explorer l'espace des coefficients pour potentiellement trouver de meilleurs polynômes doivent être encore étudiées. Enfin, nous travaillons sur l'intégration de cette méthode dans des outils pour générer automatiquement des circuits. La structure des FPGA actuels avec des blocs de mémoires et de multiplications câblées demandent d'adapter notre méthode à ces blocs. En particulier, en combinant les techniques de division (ou d'inverse) et les évaluations de polynômes, on peut, peut-être, envisager d'utiliser des approximations rationnelles.

Chapitre 4

Conclusion et perspectives

Dans l'introduction de ce mémoire 1.1, l'arithmétique des ordinateurs était définie par *la branche de l'informatique qui traite des représentations des nombres et des algorithmes pour effectuer les calculs de base en machine.* Dans cette première définition, deux aspects importants manquent ou sont, peut-être trop, implicites : l'*efficacité* des calculs et la *validité* des résultats.

Bon nombre de nos travaux en arithmétique des ordinateurs portent sur l'amélioration des performances des opérateurs de calcul. Nous désignons par *performances* des quantités assez facilement mesurables comme la vitesse (durée, latence, débit) et le coût des calculs (surface de silicium, taille de l'empreinte mémoire, énergie, puissance maximum, etc.). On ajoute à ces performances des aspects plus difficilement quantifiables, aujourd'hui, comme les fuites d'information par canaux cachés (ou auxiliaires) dans des applications sécurisées. Nous avons proposé de multiples solutions pour augmenter les performances d'opérateurs ou supports de calcul arithmétique : méthodes à base de tables, opérateurs reconfigurables, algorithmes dédiés aux FPGA, variantes de la multiplication et de la division, approximations polynomiales, opérateurs cryptographiques, bibliothèques flottantes ou pour la cryptographie, outils d'aide à la conception de « circuits » arithmétiques, etc.

Une autre part importante du travail était de pouvoir qualifier, ou quantifier, la validité des solutions proposées. En clair, est-ce qu'un opérateur calcule correctement et avec les performances souhaitées ? Cette question peut paraître simpliste pour des petits opérateurs entiers ou en virgule fixe. Mais, lorsqu'il s'agit de prendre en compte la précision par rapport à une approximation des nombres réels, les choses sont bien plus complexes. Même pour des objets mathématiques qui ne posent pas de problème de « précision », comme le calcul dans des corps finis, la validation n'est pas simple. En effet, comment garantir que l'implantation d'un algorithme complexe de multiplication de deux grands entiers de 600 bits modulo un grand nombre premier de 600 bits est bien correcte ? La simulation exhaustive est impossible. La génération de vecteurs de test offrant une couverture totale est un problème ouvert dans la plupart des cas. Dans l'ensemble de nos travaux, nous avons travaillé sur la validation de nos solutions en combinant plusieurs techniques : preuves mathématiques des algorithmes, simulations intensives aléatoires ou avec des vecteurs spécifiques générés automatiquement, tests intensifs sur circuits FPGA et utilisation d'outils d'aide à la preuve automatique de programmes.

La vitesse de calcul doit pouvoir continuer d'augmenter pour pouvoir traiter les énormes quantités de données des nouvelles applications. L'augmentation des fréquences des processeurs semblant ne plus être le principal moteur de la conception de processeurs, l'algorithmique va devoir reprendre une place de choix dans nos études. En particulier, le parallélisme semble être le maître mot pour les prochaines années. Que peut-il nous apporter ? Des arithmétiques avec un fort parallélisme interne, comme RNS, peuvent être assez facilement utilisées dans des circuits du fait des surfaces disponibles. Le parallélisme alors disponible au niveau de l'arithmétique est-il utile en pratique ? Ou bien, le parallélisme au niveau de l'application sera-t-il plus simple à exploiter ?

La surface de silicium reste un élément important du coût d'un circuit intégré. Même pour les très gros circuits, comme des processeurs hautes performances, réduire la taille des blocs de calcul est crucial pour pouvoir augmenter le parallélisme interne. De plus, toute réduction

de la surface d'un circuit engendre une réduction de sa consommation statique. L'optimisation d'opérateurs arithmétiques dédiés à des calculs spécifiques restera un axe de travail important.

Les aspects énergétiques des opérateurs arithmétiques seront, très probablement, une bonne source de problématiques de recherche. Les travaux de modélisation doivent être renforcés pour pouvoir guider des choix au niveau de l'utilisateur ou bien dans des outils de conception. Il reste encore à étudier les performances énergétiques de certaines de nos propositions. Par exemple, les méthodes à base de tables semblent intéressantes pour maintenir une faible activité dans le circuit (contrairement à une solution itérative). Mais la consommation statique des tables peut pénaliser lourdement le bilan énergétique global. Une étude importante qui reste à faire est une comparaison de l'efficacité énergétique des différentes méthodes d'évaluation des fonctions élémentaires (approximations polynomiales ou rationnelles, méthodes à base de tables, algorithmes à récurrence de chiffres ou bien des combinaisons). Par ailleurs, les liens entre la précision des calculs, les algorithmes utilisés et la consommation d'énergie seront une source de compromis et d'optimisations certaine.

Les architectures reconfigurables constituent une voie intéressante pour offrir de bonnes performances. Elles permettent de limiter à la fois la surface de silicium et la consommation statique. La vitesse n'est pas en reste puisque l'« adaptation » du support d'exécution aux besoins permet de calculer plus rapidement des opérations plus complexes. La définition de la bonne « granularité » est probablement un des points clés de ce problème. Par exemple, dans le cas des FPGA classiques, la grande flexibilité se paye par des performances souvent faibles. Les architectures reconfigurables à grain moyen offrent des atouts importants. Un objectif à moyen terme est d'étudier une architecture reconfigurable pour la cryptographie. L'évolution des protocoles, des tailles de clés et de certains paramètres cryptographiques rend difficile la conception de circuits figés. De plus, les travaux autour de la sécurisation d'opérateurs semblent indiquer que la reconfiguration peut aider à se protéger contre certaines attaques. À plus long terme, j'envisage de reprendre les travaux sur l'architecture FPOP pour des applications en automatique et en traitement du signal.

Dans les toutes prochaines années, un enjeu important pour la conception de cryptosystèmes matériels est de pouvoir mesurer clairement l'impact des diverses représentations des nombres et des divers algorithmes arithmétiques sur les performances (vitesse, taille, consommation d'énergie, niveau de sécurité). Par exemple, quelle arithmétique est la mieux appropriée pour des cryptosystèmes à base de courbes elliptiques, hyper-elliptiques ou à base de couplages ? Sur cette question, un de nos buts à moyen terme est de fournir des opérateurs arithmétiques performants pour le calcul dans les corps finis avec des nombres de 150 à 600 bits. Dans le cas particulier de la résistance aux attaques physiques, les travaux entrepris donneront, sans aucun doute, des nouvelles solutions de sécurisation.

Une autre voie de recherche semble intéressante pour les prochaines années : les problèmes de tolérance aux pannes et de variabilité des caractéristiques d'un circuit intégré dans les technologies très fines. Nous avons soumis un projet ANR dont une partie porte sur la conception d'unités de calcul qui intègrent des mécanismes de détection ou de tolérance aux pannes directement au niveau arithmétique. Avec la diminution de la finesse des gravures, les caractéristiques des éléments d'un circuit varient beaucoup dans une même puce. Est-il toujours justifié de concevoir des algorithmes de calcul pour un pire cas correspondant à des caractéristiques de portes ou de transistors fixes ? Les travaux menés sur la modélisation de l'activité parasite dans les opérateurs arithmétiques seront, peut-être, utiles sur ce sujet. L'activité parasite dépend, en effet, des caractéristiques temporelles de l'arrivée relative de signaux. Il y a, peut-être, des liens entre les modèles d'activité et ceux pour la variabilité.

Le support d'arithmétiques évoluées lors de la conception de circuits reste un problème im-

portant. Depuis le début de mes travaux, je me suis impliqué dans l'étude et la mise en œuvre logicielle de méthodes et d'outils pour ce support. Je compte poursuivre ces travaux. En particulier, je travaille sur un environnement de développement et d'optimisation d'opérateurs arithmétiques matériels.

Chapitre 5

Autres activités

5.1. Encadrement

Encadrement et co-encadrement de thèses

Thomas Chabrier, étudiant de l'Université de Bordeaux, bourse de 3 ans de la région Bretagne et du conseil général des Côtes-d'Armor, thèse débutée en octobre 2009. Sujet : *unités arithmétiques reconfigurables pour cryptoprocesseurs robustes aux attaques.* Thèse co-encadrée à 90% par moi et administrativement à 10% par E. Casseau. Thèse effectuée à l'IRISA.

Danuta Pamula, étudiante de l'Université de Gliwice en Pologne, bourse cotutelle France-Pologne de 3 ans débutée en septembre 2009. Sujet : *opérateurs arithmétiques pour la cryptographie.* Thèse co-encadrée à 90% par moi et administrativement à 10% par O. Sentieys. Thèse effectuée à mi-temps à l'IRISA et mi-temps à l'Université de Gliwice en Pologne.

José Lopes, étudiant Université Montpellier 2, bourse ANR de 3 ans débutée en octobre 2007. Sujet : *opérateurs arithmétiques reconfigurables.* Accréditation de l'Université de Montpellier 2 pour encadrer cette thèse seul (donc 100% par moi). Thèse abandonnée en fin de première année, effectuée au LIRMM.

Julien Francq, ingénieur Polytech'Montpellier, bourse CEA de 3 ans débutée en octobre 2006 et soutenue en décembre 2009. Sujet : *Conception et sécurisation d'unités arithmétiques hautes performances pour courbes elliptiques.* Thèse co-encadrée à 45% par moi avec Jean-Baptiste Rigaud (École des Mines de St-Étienne) à 45% et Jean-Claude Bajard à 10% et effectuée au Centre Microélectronique de Provence à Gardanne et au LIRMM. Il est maintenant expert en cryptographie matérielle chez EADS.

Stéphane Simard, ingénieur de l'Université du Québec à Chicoutimi au Canada, bourse de 3 ans débutée en septembre 2005. Sujet : *système sur puce en arithmétique en ligne pour le contrôle vectorielle sans capteur.* Thèse co-encadrée à 20% par moi et Rachid Bugénane à 80%. Thèse effectuée au laboratoire ERMETIS à l'UQAC Chicoutimi Canada. Changement de sujet après deux ans et arrêt du co-encadrement.

Romain Michard, ingénieur du Corps des Télécommunications, bourse de 4 ans en détachement INRIA, débutée en juillet 2004 et soutenue en juin 2008. Sujet : *opérateurs arithmétiques pour la basse consommation d'énergie.* Accréditation de l'ENS Lyon pour encadrer cette thèse seul (donc 100% par moi). Thèse effectuée au LIP, ENS Lyon. Il est maintenant en détachement INRIA au laboratoire de CITI de l'INSA Lyon.

Nicolas Veyrat-Charvillon, étudiant ENS Lyon, bourse MENRT de 3 ans débutée en septembre 2004 et soutenance en juin 2007. Sujet : *opérateurs arithmétiques pour des applications spécifiques.* Thèse co-encadrée à 90% par moi et administrativement par J.-M. Muller à 10%. Thèse effectuée au LIP, ENS Lyon. Il est maitenant postdoctorant dans le groupe de recherche sur la cryptographie de l'Université catholique de Louvain en Belgique.

Saurabh-Kumar Raina, étudiant Indien, bourse Région Rhône-Alpes de 3 ans et soutenue en septembre 2006. Sujet : *bibliothèque flottante pour processeurs entiers.* Thèse co-encadrée à 40% par moi, à 50% par C.-P. Jeannerod (encadrant principal) et à 10% par J.-M. Muller (encadrant administratif). Thèse effectuée au LIP, ENS Lyon. Il est maintenant *associate professor* au *Galgotias College of Engineering and Technology* en Inde.

Nicolas Boullis, élève normalien ENS Lyon, allocation couplée. Sujet : *génération d'opérateurs de multiplication par des constantes.* Thèse abandonnée en début de 4ème année, co-encadrée à 90% par moi et administrativement à 10% par J.-M. Muller. Thèse commencée au LIP, ENS Lyon. Il est maintenant ingénieur d'études à l'Ecole Centrale de Paris.

Fabio Restrepo, étudiant de l'Université de Cali, Colombie, thèse EPFL numéro 2457, soutenue en 2001. Titre : *outils de programmation d'une machine parallèle réalisée sur un seul circuit intégré.* Thèse financée par le CSEM et effectuée au Laboratoire de Systèmes Logiques de l'EPFL. Encadrement partiel à 30% par moi pour le CSEM en 1997–1999.

Martin Dimmler, étudiant de l'Université de Karlsruhe, Allemagne, thèse EPFL numéro 2050, soutenue en 1999. Titre : *Digital Control of Micro-Systems using On-Line Arithmetic.* Thèse financée par le CSEM et effectuée à l'Institut d'Automatique de l'EPFL. Encadrement partiel à 30% par moi pour le CSEM en 1997–1999. Il est maintenant expert en contrôle numérique à l'« *European Southern Observatory* ».

Encadrement de stages de Master 2 ou de DEA

Andrianina Andriamanga, étudiant de l'Université de Rennes 1, ENSSAT. Sujet : *opérateurs arithmétiques pour cryptoprocesseur ECC résistant aux attaques en faute.* Stage effectué à l'IRISA, février–septembre 2010, sous ma direction et celle de Stanislaw Piestrak.

Nicolas Veyrat-Charvillon, étudiant ENS Lyon, février–juillet 2004. Stage effectué au LIP, ENS Lyon. Sujet : *algorithmes de multiplication pour circuits asynchrones.*

Nicolas Boullis, élève normalien, février–juillet 2001. Stage effectué au LIP, ENS Lyon. Sujet : *algorithmes de division pour circuits asynchrones.*

Rivo Randrianarivoni, étudiant ENS Lyon, mars–juillet 1997, co-encadrement avec J.-M. Muller. Stage effectué au LIP, ENS Lyon. Sujet : *bibliothèque de calcul de fonctions élémentaires en multi-précision.*

Autres encadrements

Christophe Riolo, élève 1ère année à l'antenne de Bretagne de l'ENS Cachan, mai–juin 2009. Stage effectué à l'IRISA (Lannion). Sujet : *Etude de la consommation d'opérateurs arithmétiques.*

Florent Botella, étudiant 2ème année IUT Informatique de Montpellier, mars–mai 2007. Stage effectué au LIRMM. Sujet : *Bibliothèque arithmétique pour la cryptographie.*

Daria Tioc, étudiant Université Technique de Cluj-Napoca en Roumanie, mars–avril 2005. Stage effectué au LIP, ENS Lyon. Sujet : *Implantation FPGA de l'algorithme RSA.*

Zsolt Mathe, étudiant Université Technique de Cluj-Napoca en Roumanie, mars–avril 2005. Stage effectué au LIP, ENS Lyon. Sujet : *Implantation FPGA de l'algorithme RSA.*

Gaetan Leurent, élève École Polytechnique 2ème année, juin–août 2004. Stage effectué au LIP, ENS Lyon. Sujet : *Opérateurs matériels pour la cryptographie.*

Julien Robert, étudiant ENS Lyon 1ère année, juin–juillet 2004. Stage effectué au LIP, ENS Lyon. Sujet : *Implantation du produit modulaire sur FPGA.*

Guillaume Duveau, élève École Polytechnique 2ème année, avril–juillet 2003. Stage effectué au LIP, ENS Lyon. Sujet : *implémentation FPGA d'un algorithme de signature par code correcteur d'erreurs.*

Guillaume Melquiond, élève normalien 1ère année, mai–juin 2001. Stage effectué au LIP, ENS Lyon. Sujet : *bibliothèque flottante efficace en VHDL synthétisable pour FPGA.*

5.2. Enseignements

2008–2010 : Cours d'arithmétique des ordinateurs dans le master recherche 2ème année à l'ENSSAT–Université Rennes 1. Durée : 10 h par an, effectif : entre 5 et 10 étudiants chaque année.

2003–2005 : Cours de *circuits intégrés numériques* dans le master recherche (DEA en 2003–2004) 2ème année, spécialité Informatique Fondamentale, à l'ENS Lyon. Durée : 30 h par an, effectif : environ 10 étudiants chaque année.

2001–2002 : Cours de *conception d'architectures matérielles* dans le DEA d'Informatique de l'ENS Lyon. Cours commun avec F. de Dinechin. Durée par personne : 12 h par an, effectif : environ 10 étudiants chaque année.

1999–2001 : Formation continue pour SUN Microsystems : « Unix », « installation et administration système », et « administration système avancée ».

1997–1999 : Formateur interne au CSEM sur les outils Unix, administration système Solaris et l'environnement de programmation C/C++.

5.3. Administration et tâches collectives

Responsable de l'équipe–projet ARITH, 2007–2008 : équipe de 10 permanents et 5 doctorants. Site web de l'équipe : `http://www.lirmm.fr/arith/`.

Membre élu du conseil de laboratoire du LIRMM, 2006–2008

Représentant CURI, 2005–2007 : représentant du département informatique du LIRMM à la CURI.

Administration serveurs et outils CAO au LIP, 1999–2004 : Gestion, installation et maintenance des serveurs (4 machines sur 2 OS) et des outils (environ 12) pour la CAO de circuit VLSI et FPGA au LIP. Environ 15 utilisateurs sur deux équipes.

Responsable moyens informatiques LIP, 2001–2003 : Direction équipe de 3 ingénieurs. Administration système des machines et outils logiciels du LIP (50 machines et 100 utilisateurs environ).

Président commission informatique LIP, 2000–2003.

Organisateur séminaires LIP, 1999–2000 : 2 fois par mois.

Chef de projet, CSEM, 1999 : Organisation et administration d'un projet de recherche de base du CSEM (préparation et animation des réunions, valorisation, contacts industriels, rapport internes...). Équipe de 4 personnes.

Administration système et outils informatiques, CSEM, 1997-1999 : Mise en œuvre et maintenance d'applications informatiques (compilateurs, bibliothèques graphiques, outils de calcul numérique et symbolique, éditeurs...).

5.4. Animation

Membre comité scientifique écoles thématiques d'architecture, depuis 2002 : écoles thématiques du CNRS *Architectures des systèmes matériels enfouis et méthodes de conception associées.*

Co-organisateur ECOFAC2010, Plestin les Grèves, mars-avril 2010 : 2ème école thématique *Conception faible consommation pour les systèmes embarqués temps réel* (`http://http://ecofac2010.irisa.fr/`). Co-organisation avec D. Chillet et O. Sentieys.

Co-organisateur ARCHI09, Pleumeur-Bodou, mars 2009 : 5ème école thématique *Architectures des systèmes matériels enfouis et méthodes de conception associées* (`http://www.irisa.fr/archi09/`). Co-organisation avec O. Sentieys et S. Pillement.

Co-organisateur ARITH18, Montpellier, juin 2007 : *18th IEEE Symposium on Computer Arithmetic* (`http://www.lirmm.fr/arith18/`). Co-organisation avec L. Imbert.

Co-organisateur ARCHI07, Boussens, mars 2007 : 4ème école thématique *Architectures des systèmes matériels enfouis et méthodes de conception associées* (`http://www.lirmm.fr/archi07/`). Co-organisation avec C. Rochange.

Organisateur ARCHI05, Autrans, mars 2005 : 3ème école thématique *Architectures des systèmes matériels enfouis et méthodes de conception associées* (`http://www.ens-lyon.fr/ARCHI05/`).

Co-organisateur ARCHI03, Roscoff, octobre 2002 : 2ème école thématique *Architectures des systèmes matériels enfouis et méthodes de conception associées* (`http://www.irisa.fr/archi03/`). Co-organisation avec F. Charot et O. Sentieys.

5.5. Collaborations académiques

Nationales

ANR Architecture du Futur, 2007–2010 : Responsable local pour le LIRMM du projet ROMA *Reconfigurable Operators for Multimedia Applications*. Travail commun avec l'IRISA (7 pers.), le CEA LIST (2 pers.), Thomson (2 pers.) et le LIRMM (2 pers.).

LIRMM, 2002–2005 : Travail commun avec J.-C. Bajard, L. Imbert sur des aspects arithmétique matérielle pour la cryptographie : représentations des nombres (corps finis, représentation modulaire) et opérateurs arithmétiques utilisées dans les algorithmes cryptographiques.

ACI Nouvelles Interfaces des Mathématiques, 2004–2007 : Responsable local du projet baptisé GAAP *Génération Automatique d'Approximants Polynomiaux* efficaces en machine. Travail commun avec le laboratoire LArAL de l'Université de St-Étienne. Étude et développement d'algorithmes et d'une bibliothèque pour la génération automatique d'approximations polynomiales de fonctions avec des contraintes sur les coefficients. Effectifs : 4 personnes à Lyon et 2 à St-Étienne.

Site web : `http://lipforge.ens-lyon.fr/www/meplib/gaap/`

ACI Sécurité Informatique, 2003–2006 : Responsable local du projet baptisé OCAM *Opérateurs Cryptographiques et Arithmétique Matérielle* avec le projet INRIA CODES à (coord. N. Sendrier) et l'équipe Arithmétique Informatique du LIRMM à Montpellier (coord. J.-C. Bajard). Étude et implantation matérielle de primitives cryptographiques utilisant la théorie algébrique des codes. Implantation d'un prototype de système de signature

numérique à sécurité élevée avec des signatures courtes. Effectifs : 5 personnes à Lyon, 3 à Rocquencourt et 2 à Montpellier.

Site web : `http://www-rocq.inria.fr/codes/OCAM/`

ACI Cryptologie, 2002–2005 : Responsable local du projet baptisé OpAC *Opérateurs Arithmétiques pour la Cryptographie*. C'est un projet joint avec des membres des laboratoires LIRMM (départements informatique et micro-électronique, coord. J.-C. Bajard) et GTA (coord. P. Elbaz-Vincent) de Montpellier, travaillant dans divers domaines (géométrie arithmétique, théorie des nombres, calcul symbolique, arithmétique des ordinateurs, micro-électronique, architectures des ordinateurs, algorithmique). Effectifs : 3 à Lyon et 11 à Montpellier (7 LIRMM et 4 GTA).

Site web : `http://www.lirmm.fr/~bajard/ACI_CRYPTO/`

ACI Jeunes Chercheurs, 2000–2003 : Action commune avec F. de Dinechin sur l'*arithmétique pour circuits FPGA*. Travaux sur les méthodes d'approximation de fonctions à base de tables et d'additions, l'arithmétique en ligne, les opérations en virgule flottante et en système logarithmique. Acquisition d'un serveur, d'outils de CAO et de cartes FPGA.

Participation à l'AS STIC *arithmétiques des ordinateurs* du CNRS créée en 2002 regroupant Arénaire, LIP6, LORIA, LIAFA, LIRMM, MANO Univ. Perpignan et LASTI ENSSAT Lannion. Travail sur l'arithmétique en virgule fixe.

Participations multiples aux journées Arinews : réunions de la communauté nationale en arithmétique des ordinateurs 1 à 2 fois par an.

Internationales

Université du Massachusetts à Amherst, USA, 2009–maintenant : Collaboration avec le laboratoire VLSI CAD sur les outils pour la conception de circuits intégrés numériques.

Séjour : invitation de 10 jours en novembre 2009.

Université de Calgary, Canada, 2004–maintenant : Collaboration avec le laboratoire ATIPS dans l'équipe de G. Julien et V. Dimitrov. Travail commun sur des opérateurs arithmétiques pour la cryptographie et le traitement du signal.

Séjours : 10 jours en octobre 2008, 5 jours en octobre 2006, 10 jours en octobre 2005, invitation 1 mois en mai/juin 2005, 10 jours en novembre 2004.

Université de Waterloo, Canada, 2006–2007 : Collaboration avec le groupe de cryptographie.

Séjour : invitation de 10 jours en août 2007

Université de Cork, Irlande, 2006–2009 : Collaboration avec le groupe de recherche sur les codes et la cryptographie. Bourses EGIDE PAI Ulysses en 2006, 2007 et 2008 pour financer des séjours de 2 semaines dans chaque sens. Travail commun sur l'étude et la conception d'opérateurs arithmétiques sur les courbes elliptiques et résistants à l'analyse simple de consommation.

Séjours : 10 jours en novembre 2008, 8 jours en mai 2007, 7 jours en mai 2006.

Université du Québec à Chicoutimi, Canada, 2005-2007 : Collaboration avec le laboratoire ERMETIS (groupe de recherche en microélectronique et en traitement numérique du signal). Travail commun sur les opérateurs arithmétiques en ligne pour le contrôle numérique.

Séjour : invitation de 15 jours en juin/juillet 2006.

University of California at Davis, U.S.A., 2005 : Collaboration avec le laboratoire Advanced Computer Engineering sur le thème des opérateurs arithmétiques pour la basse consommation d'énergie.

Séjour : invitation de 15 jours en novembre 2005.

Université de Cardiff, Pays de Galles, 2001–2002 : Coordinateur français d'une action intégrée dans le cadre du programme franco-britannique Alliance avec l'équipe de N. Burgess. Travail commun sur l'implantation FPGA du système logarithmique et d'opérateurs arithmétiques à basse consommation d'énergie. Accueils réciproques d'étudiants.

Séjour : 10 jours en mai 2002.

University of California at Los Angeles, U.S.A., 1995–aujourd'hui : Séjour de 1 mois, en février 2000, dans l'équipe de M. Ercegovac. Petits séjours en 1995, 1997, 2001, 2003 et 2004. Travail commun sur des opérateurs de division en très grande base, l'arithmétique en ligne, l'évaluation des fonctions algébriques et élémentaires, les méthodes à base de tables et de petites multiplications. Cette collaboration a été financée par un projet PICS en 1997–1999.

Séjours : 1 mois en février 2000, multiples séjours d'une à deux semaine en 2007, 2005, 2004, 2003, 2001, 1996 et 1995.

University of California at Irvine, U.S.A., 2000 : Séjour d'un mois, en mars 2000, dans l'équipe de T. Lang. Travail commun sur ses opérateurs de division en très grande base et méthodes à base de petites multiplications et des tables.

Séjours : 1 mois en mars 2000, 1 semaine en 1995 et 1996

EPFL, Lausanne, Suisse, 1997–1999 : Collaboration avec le Laboratoire de Systèmes Logiques et l'Institut d'Automatique. Travail commun sur l'arithmétique en ligne et les systèmes parallèles mono-puce. Co-encadrement de thèses et des stages.

5.6. Collaborations et contrats industriels

BEA Technologies, 2008 : Contrat dans le cadre de l'incubation de la société BEA Technologies par Languedoc Roussillon Incubation sur l'implantation matérielle d'algorithmes de chiffrement par flot.

STMicroelectronics, 2003–2006 : Contrat avec STMicroelectronics à Grenoble. Étude et implantation d'une bibliothèque flottante (opérations et fonctions élémentaires) pour la famille de processeurs entiers VLIW ST200. Travail commun avec C.-P. Jeannerod, J.-M. Muller et S. K. Raina.

STMicroelectronics, 2002 : Collaboration avec STMicroelectronics à Crolles sur l'étude et le développement d'une unité de calcul de multiplication/addition asynchrone pour le processeur ST20.

POSIC, 2001–2002 : Contrat INRIA avec POSIC S.A. à Neuchâtel en Suisse. Étude et implantation d'algorithmes et d'architectures de calcul rapide pour les capteurs de position. Participation à la réalisation d'un logiciel de test et d'une carte prototype FPGA.

STMicroelectronics, 2000–2001 : Contrat INRIA avec STMicroelectronics à Montbonnot. Sujet : algorithmes de division par des constantes pour le processeur DSP ST100. Travail commun avec J.-M. Muller et implanté dans la version de juin 2001 du compilateur vendu par STMicroelectronics.

CSEM, 2000 : Contrat INRIA avec le Centre Suisse d'Électronique et de Microtechnique à Neuchâtel en Suisse. Étude et développement de blocs de calcul spécifiques pour le traitement d'image pour un prototype à base de FPGA et un circuit ASIC.

5.7. Développement de logiciels

- **PACE**, *Prototyping Arithmetic in Cryptography Easily* :
  Bibliothèque C++ modulaire et haute performance pour la représentation des nombres et les calculs en cryptographie.
  Développement commun avec P. Giorgi, L. Imbert et A. Perreira.
- **Seedgen**, *générateur d'approximations rapides pour l'inverse et la racine carrée inverse en matériel* :
  Programme C, sous licence GPL, pour la génération automatique d'opérateurs matériels pour l'approximation rapide de l'inverse et la racine carrée inverse.
  L'algorithme utilisé dans ce programme est issu d'une collaboration avec M. Ercegovac (UCLA) et J.-M. Muller (LIP).
  Site web : `http://www.lirmm.fr/~tisseran/devel/seedgen/`
- **MEPLib**, *machine-efficient polynomial library* :
  Bibliothèque C, sous licence LGPL, pour la génération automatique d'approximations polynomiales de fonctions avec des contraintes sur les coefficients : formats des coefficients, valeurs ou domaines de valeurs.
  Développement commun avec N. Brisebarre, F. Hennecart, J.-M. Muller et S. Torres dans le cadre de l'ACI *nouvelles interfaces des mathématiques.*
  Site web : `http://lipforge.ens-lyon.fr/projects/meplib/`
- **Divgen**, un générateur de diviseurs matériels :
  Programme, distribué sous licence GPL, de génération de descriptions VHDL synthétisables et optimisées d'opérateurs de division. Paramètres : type d'algorithme, base, type de représentation intermédiaire, optimisations au niveau architectural/circuit, circuits cibles ASIC ou FPGA.
  Développement commun avec R. Michard et N. Veyrat-Charvillon.
  Ce programme a été utilisé dans le cadre d'une collaboration entre le CEA-Léti et l'INRIA.
  Site web : `http://lipforge.ens-lyon.fr/projects/divgen/`
- **FLIP**, *floating-point library for integer processor* :
  Bibliothèque C de support pour le calcul flottant sur des processeurs entiers ou en virgule fixe (i.e. sans unité flottante). Les différentes opérations de base sont implantées et optimisées pour la simple précision et la famille de processeurs VLIW ST200 de STMicroelectronics.
  Développement commun avec C.-P. Jeannerod, J.-M. Muller et S. K. Raina.
  Cette bibliothèque est développée dans le cadre d'une collaboration avec la société STMicroelectronics à Grenoble et fait l'objet d'un support financier de la région Rhône-Alpes (bourse de thèse de S. K. Raina).

5.8. Participation à des comissions de spécialistes

- Membre extérieur de la commission de spécialistes de l'Université du Sud, Toulon-Var, 27ème section, 2007–2009.
- Membre extérieur de la commission de spécialistes de l'Université Joseph Fourier à Grenoble, 27ème section, 2003–2004.

5.9. Édition, comités de programmes, relectures

2008–aujourd'hui : Membre du comité de programme des conférences : *IEEE Symposium on Computer Arithmetic* (ARITH 18, 19, 20), International Conference on ReConFigurable Computing and FPGAs (ReConFig 2008–2009) Faible Tension Faible Consommation (FTFC 20008–2009), Symposium d'architecture des machines (SympA 2008–2009).

2006–aujourd'hui : éditeur associé de la revue *International Journal of High Performance Systems Architecture* (`http://www.inderscience.com/ijhpsa`).

2001 : Co-édition avec M. Daumas et F. de Dinechin d'un numéro spécial de la revue Réseaux Systèmes Répartis, Calculateurs Parallèles sur l'*arithmétique des ordinateurs*. Vol. 13, n° 4–5. 2001. Introduction de 29 pages et 7 chapitres invités soit un total de 200 pages environ.

Participation à la relecture d'articles pour différents journaux : IEEE Transactions on Computers, IEEE Transactions on Circuits and Systems, IEEE Transactions on VLSI Systems, IEEE Journal on Solid State Circuits, IEEE Design and Test, IEEE Transactions on Parallel and Distributed Systems, Journal of VLSI Signal Processing, ACM Transactions on Embedded Computing Systems, ACM Transactions on Design Automation of Electronic Systems, International Journal of High Performance Systems Architecture, Integration the VLSI Journal, Computer Journal, Electronic Letters, Journal of Systems Architecture, IET Circuits Devices & Systems, Techniques et Science Informatique, Microelectronics Journal, ASP Journal of Low Power Electronics, Journal of Universal Computer Science, International Journal of Reconfigurable Computing.

Participation à la relecture d'articles pour différentes conférences : ARITH, FPL, ASAP, PATMOS, ISPLED, ICCS, ISSAC, ISCAS, DATE, SAC, ReConfig, DDECS, SYMPA, FTFC, FPT, RNC, SCAN, ICES, IWLAS, WAIFI, STACS.

5.10. Expertises

Participations à des jurys de thèse :
- Co-directeur sur la thèse de Julien Francq en 2009 à l'Université de Montpellier 2.
- Examinateur sur la thèse de Benoît Bradignans en 2009 à l'Université de Montpellier 2.
- Membre invité sur la thèse de Robin Perrot en 2007 à l'Université Montpellier 2.
- Examinateur sur la thèse de Florent Bernard en 2007 à l'Université Paris 8.
- Rapporteur sur la thèse de Jean-Luc Beuchat en 2001 à l'EPFL.
- Rapporteur sur la thèse d'Eméka Mosanya en 1998 à l'EPFL.

Autres expertises :

ANR : projets en 2007, 2008 et 2010

ANRT projet en 2009

Oseo-Anvar : projet en 2006

FNRS, Belgique : projet en 2010

NSERC/CRSNG, Canada : projet en 2008 et 2009

Chapitre 6

Bibliographie personnelle

Les références ci-dessous correspondent aux travaux et publications décrits dans ce mémoire. La liste complète de mes publications est disponible, ainsi que leur contenu quand les droits le permettent, sur la page web `http://www.irisa.fr/prive/Arnaud.Tisserand`.

[1] R. Beguenane, J.-G. Mailloux, S. Simard, and A. Tisserand. Towards the system-on-chip realization of a sensorless vector controller with microsecond-order computation time. In *Proc. Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 908–912, Ottawa, Canada, May 2006. IEEE. ⟨ p. 21 ⟩

[2] R. Beguenane, S. Simard, and A. Tisserand. Function evaluation on FPGAs using on-line arithmetic polynomial approximation. In *Proc. 4th International Northeast Workshop on Circuits and Systems (NEWCAS)*, pages 21–24, Gatineau, Canada, June 2006. IEEE. ⟨ p. 21 ⟩

[3] C. Bertin, N. Brisebarre, B. Dupont de Dinechin, C.-P. Jeannerod, C. Monat, J.-M. Muller, S. K. Raina, and A. Tisserand. A floating-point library for integer processors. In F. T. Luk, editor, *Proc. Advanced Signal Processing Algorithms, Architectures and Implementations XIV*, volume 5559, pages 101–111, Denver, Colorado, U.S.A., August 2004. SPIE. ⟨ pp. 41, 43 ⟩

[4] J.-L. Beuchat, L. Imbert, and A. Tisserand. Comparison of modular multipliers on FPGAs. In F. T. Luk, editor, *Proc. Advanced Signal Processing Algorithms, Architectures and Implementations XIII*, volume 5205, pages 490–498, San Diego, California, U.S.A., August 2003. SPIE. ⟨ pp. 48, 49 ⟩

[5] J.-L. Beuchat, N. Sendrier, A. Tisserand, and G. Villard. FPGA implementation of a recently published signature scheme. Research Report RR2004-14, Laboratoire de l'Informatique du Parallélisme (LIP), March 2004. Also available as INRIA Research Report RR-5158. ⟨ p. 49 ⟩

[6] J.-L. Beuchat and A. Tisserand. Small multiplier-based multiplication and division operators for Virtex-II devices. In M. Glesner, P. Zipf, and M. Renovell, editors, *Proc. 12th International Conference on Field-Programmable Logic and Applications (FPL)*, volume 2438 of *LNCS*, pages 513–522, Monptellier, France, September 2002. Springer. ⟨ pp. 36, 37 ⟩

[7] J.-L. Beuchat and A. Tisserand. Évaluation polynomiale en-ligne de fonctions élémentaires sur FPGA. *Technique et Science Informatiques*, 23(10) :1247–1267, 2004. ⟨ p. 69 ⟩

[8] J.-L. Beuchat and A. Tisserand. Opérateurs arithmétiques pour FPGA. In J.-C. Bajard and J.-M. Muller, editors, *Arithmétique des ordinateurs*, Traité I2C : Information–Commande–Communication, chapter 4, pages 109–152. Hermes, Lavoisier, 2004. ⟨ pp. 36, 37 ⟩

[9] N. Boullis and A. Tisserand. On digit-recurrence division algorithms for self-timed circuits. In F. T. Luk, editor, *Proc. Advanced Signal Processing Algorithms, Architecture and*

Implementations XI, volume 4474, pages 115–125, San Diego, California, U.S.A., August 2001. SPIE. ⟨ p. 36 ⟩

[10] N. Boullis and A. Tisserand. Génération automatique d'architectures de calcul pour des opérations linéaires : application à l'IDCT sur FPGA. In *8ème SYMPosium en Architectures nouvelles de machines (SYMPA)*, pages 283–290, Hamamet, Tunisie, April 2002. ⟨ pp. 38, 39 ⟩

[11] N. Boullis and A. Tisserand. Some optimizations of hardware multiplication by constant matrices. In J.-C. Bajard and M. Schulte, editors, *Proc. 16th Symposium on Computer Arithmetic (ARITH)*, pages 20–27, Santiago de Compostela, Spain, June 2003. IEEE Computer Society. ⟨ pp. 38, 39 ⟩

[12] N. Boullis and A. Tisserand. Some optimizations of hardware multiplication by constant matrices. *IEEE Transactions on Computers*, 54(10) :1271–1282, October 2005. ⟨ pp. 38, 39, 129 ⟩

[13] N. Brisebarre, J.-M. Muller, and A. Tisserand. Sparse-coefficient polynomial approximations for hardware implementations. In *Proc. 38th Asilomar Conference on Signals, Systems and Computers*, pages 532–535, Pacific Grove, California, U.S.A., November 2004. IEEE. ⟨ pp. 43, 65 ⟩

[14] N. Brisebarre, J.-M. Muller, and A. Tisserand. Computing machine-efficient polynomial approximations. *ACM Transactions on Mathematical Software*, 32(2) :236–256, June 2006. ⟨ pp. 43, 64, 65, 69, 142 ⟩

[15] N. Brisebarre, J.-M. Muller, A. Tisserand, and S. Torres. Hardware operators for function evaluation using sparse-coefficient polynomials. *IEE Electronics Letters*, 42(25) :1441–1442, December 2006. ⟨ pp. 43, 65 ⟩

[16] A. Byrne, F. Crowe, W. P. Marnane, N. Meloni, A. Tisserand, and E. M. Popovici. SPA resistant elliptic curve cryptosystem using addition chains. *Int. J. High Performance Systems Architecture*, 1(2) :133–142, October 2007. ⟨ p. 50 ⟩

[17] A. Byrne, N. Meloni, F. Crowe, W. P. Marnane, A. Tisserand, and E. M. Popovici. SPA resistant elliptic curve cryptosystem using addition chains. In *Proc. 4th International Conference on Information Technology (ITNG)*, pages 995–1000, Las Vegas, Nevada, U.S.A., April 2007. IEEE. ⟨ pp. 49, 50 ⟩

[18] A. Byrne, N. Meloni, A. Tisserand, E. M. Popovici, and W. P. Marnane. Comparison of simple power analysis attack resistant algorithms for an elliptic curve cryptosystem. *Journal of Computers*, 2(10) :52–62, 2007. ⟨ pp. 49, 50 ⟩

[19] S. Collange, D. Defour, and A. Tisserand. Power consumption of GPUs from a software perspective. In *Proc. 9th International Conference on Computational Science (ICCS)*, volume 5544 of *LNCS*, pages 914–923, Baton Rouge, Louisiana, U.S.A., May 2009. ⟨ p. 47 ⟩

[20] M. Daumas, F. de Dinechin, and A. Tisserand. *L'arithmétique des ordinateurs*, chapter Introduction au numéro spécial sur l'arithmétique des ordinateurs, pages 327–356. Number 13 :4–5 in Réseaux et systèmes répartis, calculateurs parallèles. Hermes, December 2001. ⟨ p. 8 ⟩

[21] M. Daumas, F. de Dinechin, and A. Tisserand, editors. *L'arithmétique des ordinateurs*. Number 13 :4-5 in Réseaux et systèmes répartis, calculateurs parallèles. Hermes, December 2001. ⟨ p. 8 ⟩

[22] F. de Dinechin and A. Tisserand. Table-based methods comparison for low-precision evaluation of the sine and cosine functions on FPGAs. In F. T. Luk, editor, *Proc. Advanced*

Signal Processing Algorithms, Architectures, and Implementations X, volume 4116, pages 226–234, San Diego, California, U.S.A., August 2000. SPIE. ⟨ pp. 30, 32 ⟩

[23] F. de Dinechin and A. Tisserand. Some improvements on multipartite tables methods. In N. Burgess and L. Ciminiera, editors, *Proc. 15th Symposium on Computer Arithmetic (ARITH)*, pages 128–135, Vail, Colorado, U.S.A., June 2001. IEEE Computer Society. ⟨ pp. 30, 32 ⟩

[24] F. de Dinechin and A. Tisserand. Multipartite table methods. *IEEE Transactions on Computers*, 54(3) :319–330, March 2005. ⟨ pp. 30, 31, 32, 116 ⟩

[25] M. Dimmler, A. Tisserand, U. Holmberg, and R. Longchamp. On-line arithmetic for real-time control of microsystems. *IEEE/ASME Transactions on Mechatronics*, 4(2) :213–217, June 1999. ⟨ pp. 20, 21, 44, 99 ⟩

[26] M. D. Ercegovac, T. Lang, J.-M. Muller, and A. Tisserand. Reciprocation, square root, inverse square root, and some elementary functions using small multipliers. In F. T. Luk, editor, *Proc. Advanced Signal Processing Algorithms, Architectures, and Implementations VIII*, volume 3461, pages 543–554, San Diego, California, U.S.A., June 1998. SPIE. ⟨ pp. 29, 30 ⟩

[27] M. D. Ercegovac, T. Lang, J.-M. Muller, and A. Tisserand. Reciprocation, square root, inverse square root, and some elementary functions using small multipliers. *IEEE Transactions on Computers*, 49(7) :628–637, July 2000. ⟨ pp. 29, 30, 105 ⟩

[28] M. D. Ercegovac, J.-M. Muller, and A. Tisserand. Simple seed architectures for reciprocal and square root reciprocal. In *Proc. 39th Asilomar Conference on Signals, Systems and Computers*, pages 1167–1171, Pacific Grove, California, U.S.A., October 2005. IEEE. ⟨ pp. 32, 35 ⟩

[29] J. Francq, J.-B. Rigaud, P. Manet, A. Tria, and A. Tisserand. Error detection for borrow-save adders dedicated to ECC unit. In *5th Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 77–86, Washington, DC, U.S.A., August 2008. IEEE. ⟨ p. 50 ⟩

[30] P. Giorgi, T. Izard, and A. Tisserand. Comparison of modular arithmetic algorithms on GPUs. In *Proc. International Conference on Parallel Computing ParCo*, pages x–y, Lyon, France, September 2009. ⟨ pp. 53, 56 ⟩

[31] B. Girau, P. Marchal, P. Nussbaum, A. Tisserand, and H. F. Restrepo. A massively parallel one-chip architecture : Towards evolvable array processing. In *Proc. International Conference on Microelectronics for Neural Networks and Fuzzy Systems (MicroNeuro)*, pages 187–193, Granada, Spain, April 1999. IEEE. ⟨ pp. 23, 41 ⟩

[32] B. Girau and A. Tisserand. MLP computing and learning on FPGA using on-line arithmetic. *International Journal of Systems Research and Information Science*, 9(2–4), 1999. ⟨ p. 21 ⟩

[33] R. Glabb, L. Imbert, G. Jullien, A. Tisserand, and N. Veyrat-Charvillon. Multi-mode operator for SHA-2 hash functions. In *Proc. International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 207–210, Las Vegas, Nevada, U.S.A., June 2006. ⟨ p. 49 ⟩

[34] R. Glabb, L. Imbert, G. Jullien, A. Tisserand, and N. Veyrat-Charvillon. Multi-mode operator for SHA-2 hash functions. *Journal of Systems Architecture*, 53(2-3) :127–138, February 2007. Special issue on "Embedded Hardware for Cryptosystems". ⟨ p. 49 ⟩

[35] L. Imbert, A. Peirera, and A. Tisserand. A library for prototyping the computer arithmetic level in elliptic curve cryptography. In F. T. Luk, editor, *Proc. Advanced Signal Processing Algorithms, Architectures and Implementations XVII*, volume 6697, pages 1–9, San Diego, California, U.S.A., August 2007. SPIE. ⟨ pp. 53, 56 ⟩

[36] C.-P. Jeannerod, S.-K. Raina, and A. Tisserand. High-radix floating-point division algorithms for embedded VLIW integer processors. In *Proc. 17th World Congress on Scientific Computation, Applied Mathematics and Simulation IMACS*, Paris, France, July 2005. ⟨ pp. 42, 43 ⟩

[37] P. Marchal, A. Tisserand, and C. Piguet. Réseaux programmables d'opérateurs arithmétiques en-ligne. Brevet du Centre Suisse d'Electronique et de Microtechnique (CSEM), Neuchâtel, Switzerland, January 1999. Validity Domain : Switzerland, Europe, U.S.A. ⟨ p. 28 ⟩

[38] R. Michard, A. Tisserand, and N. Veyrat-Charvillon. Divgen : a divider unit generator. In F. T. Luk, editor, *Proc. Advanced Signal Processing Algorithms, Architectures and Implementations XV*, volume 5910, pages 1–12, San Diego, California, U.S.A., August 2005. SPIE. ⟨ pp. 50, 51 ⟩

[39] R. Michard, A. Tisserand, and N. Veyrat-Charvillon. Étude statistique de l'activité de la fonction de sélection dans l'algorithme de e-méthode. In *5ième journées d'études Faible Tension Faible Consommation (FTFC)*, pages 61–65, Paris, France, May 2005. ⟨ pp. 45, 47 ⟩

[40] R. Michard, A. Tisserand, and N. Veyrat-Charvillon. Evaluation de polynômes et de fractions rationnelles sur FPGA avec des opérateurs à additions et décalages en grande base. In *10ième SYMPosium en Architectures nouvelles de machines (SYMPA)*, pages 85–96, Le Croisic, France, April 2005. ⟨ p. 69 ⟩

[41] R. Michard, A. Tisserand, and N. Veyrat-Charvillon. Small FPGA polynomial approximations with 3-bit coefficients and low-precision estimations of the powers of x. In S. Vassiliadis, N. Dimopoulos, and S. Rajopadhye, editors, *Proc. 16th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 334–339, Samos, Greece, July 2005. IEEE Computer Society. Best Paper Award. ⟨ pp. 43, 64 ⟩

[42] R. Michard, A. Tisserand, and N. Veyrat-Charvillon. Carry prediction and selection for truncated multiplication. In *Workshop on Signal Processing Systems (SiPS)*, pages 339–344, Banff, Canada, October 2006. IEEE. ⟨ pp. 39, 40 ⟩

[43] R. Michard, A. Tisserand, and N. Veyrat-Charvillon. New identities and transformations for hardware power operators. In F. T. Luk, editor, *Proc. Advanced Signal Processing Algorithms, Architectures and Implementations XVI*, volume 6313, pages 1–10, San Diego, California, U.S.A., August 2006. SPIE. ⟨ p. 40 ⟩

[44] R. Michard, A. Tisserand, and N. Veyrat-Charvillon. Optimisation d'opérateurs arithmétiques matériels à base d'approximations polynomiales. In *11ième SYMPosium en Architectures nouvelles de machines (SYMPA)*, pages 130–141, Perpignan, France, October 2006. ⟨ p. 61 ⟩

[45] R. Michard, A. Tisserand, and N. Veyrat-Charvillon. Optimisation d'opérateurs arithmétiques matériels à base d'approximations polynomiales. *Technique et Science Informatiques*, 27(6) :699–718, June 2008. ⟨ pp. 43, 61 ⟩

[46] J.-M. Muller, A. Tisserand, B. Dupont de Dinechin, and C. Monat. Division by constant for the ST100 DSP microprocessor. In P. Montuschi and E. Schwarz, editors, *Proc. 17th*

Symposium on Computer Arithmetic (ARITH), pages 124–130, Cape Cod, MA., U.S.A, June 2005. IEEE Computer Society. ⟨ p. 43 ⟩

[47] P. Nussbaum, B. Girau, and A. Tisserand. Field programmable processor arrays. In M. Shipper, D. Mange, and A. Perez-Uribe, editors, *Proc. 2nd International Conference on Evolvable Systems (ICES) : from biology to hardware*, volume 1478 of *LNCS*, pages 311–322, Lausanne, Switzerland, September 1998. Springer. ⟨ pp. 23, 41 ⟩

[48] R. Santoro, A. Tisserand, O. Sentieys, and S. Roy. Arithmetic operators for on-the-fly evaluation of TRNGs. In *Proc. Advanced Signal Processing Algorithms, Architectures and Implementations XVIII*, volume 7444, pages 1–12, San Diego, California, U.S.A., August 2009. SPIE. ⟨ p. 58 ⟩

[49] A. Tisserand. *Adéquation Arithmétique Architecture : problèmes et études de cas.* Thèse de doctorat, Ecole Normale Supérieure de Lyon, Lyon, France, September 1997. ⟨ p. 20 ⟩

[50] A. Tisserand. Low-power arithmetic operators. In C. Piguet, editor, *Low Power Electronics Design*, chapter 9. CRC Press, November 2004. ⟨ pp. 44, 45 ⟩

[51] A. Tisserand. Algorithms and number systems for hardware computer arithmetic. In *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, Beijing, China, July 2005. Invited tutorial. ⟨ pp. 48, 49 ⟩

[52] A. Tisserand. Automatic generation of low-power circuits for the evaluation of polynomials. In *Proc. 40th Asilomar Conference on Signals, Systems and Computers*, pages 2053–2057, Pacific Grove, California, U.S.A., October 2006. IEEE. ⟨ pp. 43, 44, 45 ⟩

[53] A. Tisserand. Hardware operator for simultaneous sine and cosine evaluation. In *Proc. International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, volume 3, pages 992–995, Toulouse, France, May 2006. IEEE. ⟨ p. 43 ⟩

[54] A. Tisserand. Estimation rapide de l'activité parasite pour l'optimisation des arbres de réduction de multiplieurs. In *6ième journées d'études Faible Tension Faible Consommation (FTFC)*, pages 127–130, Paris, France, May 2007. ⟨ pp. 45, 47 ⟩

[55] A. Tisserand. Hardware reciprocation using degree-3 polynomials but only 1 complete multiplication. In *Proc. 5th International Northeast Workshop on Circuits & Systems (NEWCAS/MWSCAS)*, pages 301–304, Montréal, Canada, August 2007. IEEE. ⟨ pp. 43, 70 ⟩

[56] A. Tisserand. High-performance hardware operators for polynomial evaluation. *Int. J. High Performance Systems Architecture*, 1(1) :14–23, March 2007. invited paper. ⟨ p. 43 ⟩

[57] A. Tisserand. Introduction aux représentations des nombres et opérateurs arithmétiques à basse consommation d'énergie. *Technique et Science Informatiques*, 26(5) :639–646, May 2007. ⟨ pp. 44, 45 ⟩

[58] A. Tisserand. Fast and accurate activity evaluation in multipliers. In *Proc. 42th Asilomar Conference on Signals, Systems and Computers*, pages 757–761, Pacific Grove, California, U.S.A., October 2008. IEEE. ⟨ pp. 45, 47 ⟩

[59] A. Tisserand. Function approximation based on estimated arithmetic operators. In *Proc. 43th Asilomar Conference on Signals, Systems and Computers*, pages 1798–1802, Pacific Grove, California, U.S.A., October 2009. IEEE. ⟨ p. 59 ⟩

[60] A. Tisserand. Low-power arithmetic operators. In *8èmes journées d'études Faible Tension Faible Consommation*, Neuchâtel, Switzerland, June 2009. Invited Talk. ⟨ pp. 44, 45 ⟩

[61] A. Tisserand. Opérateurs arithmétiques sécurisés. Cours École Thématique ARCHI09, March 2009. ⟨ p. 49 ⟩

[62] A. Tisserand. Introduction à la consommation d'énergie dans les circuits intégrés numériques. Cours École Thématique ECOFAC2010, March 2010. ⟨ p. 44 ⟩

[63] A. Tisserand and M. Dimmler. FPGA implementation of real-time digital controllers using on-line arithmetic. In *Proc. 7th International Workshop on Field Programmable Logic and Applications (FPL)*, volume LNCS-1304, pages 472–481, London, England, August 1997. Springer. ⟨ pp. 20, 21, 44 ⟩

[64] A. Tisserand, P. Marchal, and C. Piguet. FPOP : Field programmable on-line operators. In F. T. Luk, editor, *Proc. Advanced Signal Processing Algorithms, Architectures and Implementations IX*, volume 3807, pages 31–42, Denver, Colorado, U.S.A., September 1999. SPIE. ⟨ pp. 28, 44 ⟩

[65] A. Tisserand, P. Marchal, and C. Piguet. An on-line arithmetic based FPGA for low-power custom computing. In *Proc. 9th International Workshop on Field Programmable Logic and Applications (FPL)*, volume 1673 of *LNCS*, pages 264–273, London, England, September 1999. Springer. ⟨ p. 28 ⟩

Chapitre 7

Bibliographie générale

[66] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming : Concepts, Tools, And Techniques From Boost And Beyond*. Addison-Wesley Professional, 2004. ⟨ p. 52 ⟩

[67] AIS 31 : Functionality classes and evaluation methodology for physical random number generators, September 2001. version 1. ⟨ p. 58 ⟩

[68] A. Alexandrescu. *Modern C++ Design : Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001. ⟨ p. 52 ⟩

[69] American National Standards Institute and Institute of Electrical and Electronics Engineers. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard, Std 754-1985*, 1985. ⟨ pp. 29, 36, 40, 41 ⟩

[70] A. Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, 10 :389–400, 1961. Reprinted in E. Swartzlander (ed.), Computer arithmetic, vol. II, IEEE Computer Society Press, 1990. ⟨ p. 8 ⟩

[71] J.-C. Bajard, L. Imbert, P.-Y. Liardet, and Y. Teglia. Leak resistant arithmetic. In *Proc. Cryptographic Hardware and Embedded Systems (CHES)*, volume 3156 of *LNCS*, pages 62–75, 2004. ⟨ p. 10 ⟩

[72] R. Bernstein. Multiplication by integer constants. *Software – Practice and Experience*, 16(7) :641–652, July 1986. ⟨ p. 38 ⟩

[73] S. Boldo. *Preuves formelles en arithmétique à virgule flottante*. PhD thesis, École Normale Supérieure de Lyon, November 2004. ⟨ p. 15 ⟩

[74] A. D. Booth. A signed binary multiplication technique. *Quarterly Journal of Mechanics and Applied Mathematics*, 4(2) :236–240, 1951. ⟨ p. 38 ⟩

[75] N. Brisebarre and J.-M. Muller. Functions approximable by e-fractions. In *38th Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 1341–1344, November 2004. ⟨ p. 69 ⟩

[76] A. Cauchy. Sur les moyens d'éviter les erreurs dans les calculs numériques. *Comptes rendus de l'Académie des Sciences*, Oeuvre complètes (tome 5, série 1) :431–442, November 1840. extrait n. 105. ⟨ p. 9 ⟩

[77] J.-M. Chesneaux, L.-S. Didier, F. Jézéquel, J.-L. Lamotte, and F. Rico. CADNA : Control of accuracy and debugging for numerical applications. `http://www-pequan.lip6.fr/cadna/`. LIP6–UPMC. ⟨ pp. 56, 65 ⟩

[78] L. Cordesses. Direct digital synthesis : A tool for periodic wave generation (part 1). *IEEE Signal Processing Magazine*, 21(4) :50–54, July 2004. ⟨ p. 61 ⟩

[79] J. Detrey and F. de Dinechin. A tool for unbiased comparison between logarithmic and floating-point arithmetic. *Journal of VLSI Signal Processing*, 2007. ⟨ p. 13 ⟩

[80] M. Dimmler. *Digital Control of Micro-Systems using On-Line Arithmetic*. PhD thesis, Swiss Federal Institute of Technology Lausanne, 1999. ⟨ p. 21 ⟩

[81] M. D. Ercegovac. A general hardware-oriented method for evaluation of functions and computations in a digital computer. *IEEE Transactions on Computers*, C-26(7) :667–680, 1977. ⟨ p. 45 ⟩

[82] M. D. Ercegovac and T. Lang. *Division and Square-Root Algorithms : Digit-Recurrence Algorithms and Implementations.* Kluwer Academic, 1994. ⟨ pp. 9, 11, 42, 50 ⟩

[83] M. D. Ercegovac and T. Lang. *Digital Arithmetic.* Morgan Kaufmann, 2003. ⟨ pp. 8, 9, 11, 20, 25, 32, 35, 37, 40, 61, 62 ⟩

[84] H. Fanet. *Micro et nano-électronique : Bases, composants, circuits.* Dunod, 2006. ⟨ pp. 22, 32 ⟩

[85] Fips 180-2 : Secure hash standard. National Institute of Standards and Technology (NIST), 2002. ⟨ p. 49 ⟩

[86] M. J. Flynn and S. F. Oberman. *Advanced Computer Arithmetic Design.* Wiley-Interscience, 2001. ⟨ p. 11 ⟩

[87] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR : A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2) :13 :1–13 :15, June 2007. ⟨ p. 15 ⟩

[88] GMP. GNU MP : The GNU multiple precision arithmetic library, 2010. ⟨ p. 15 ⟩

[89] G. H. Golub and C. F. Van Loan. *Matrix Computations.* John Hopkins University Press, 3rd edition, 1996. ⟨ p. 7 ⟩

[90] E. Goubault, M. Martel, and S. Putot. FLUCTUAT : Static analysis for numerical precision. `http://www-list.cea.fr/labos/fr/LSL/fluctuat/index.html`. CEA–LIST. ⟨ pp. 56, 65 ⟩

[91] G. Guitel. *Histoire comparée des numérations écrites.* Flammarion, 1975. ⟨ p. 8 ⟩

[92] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer, 2004. ⟨ pp. 48, 51 ⟩

[93] G. Ifrah. *Histoire universelle des chiffres.* Robert Lafond, 1994. ⟨ p. 8 ⟩

[94] A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Phys. Doklady*, 7(7) :595–596, January 1963. ⟨ p. 36 ⟩

[95] D. R. Kelly and B. J. Phillips. Arithmetic data value speculation. In *Proc. Advances in Computer Systems Architecture (ACSAC)*, volume 3740 of *LNCS*, pages 353–366. Springer, October 2005. ⟨ p. 58 ⟩

[96] W. Killmann and W. Schindler. Functionality classes and evaluation methodology for true (physical) random number generators. Technical Report AIS 31, BSI, September 2001. version 3.1. ⟨ p. 58 ⟩

[97] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley, 3rd edition, 1997. ⟨ p. 36 ⟩

[98] C. Lauter, S. Chevillard, M. Joldes, and N. Jourdan. Safe floating-point code development tool, 2010. ⟨ p. 62 ⟩

[99] V. Lefèvre. Multiplication par une constante. *Réseaux et Systèmes Répartis, Calculateurs Parallèles*, 13(4-5) :465–484, 2001. ⟨ p. 38 ⟩

[100] R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications.* Cambridge University Press, second edition, 1994. ⟨ pp. 48, 51 ⟩

[101] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks : Revealing the Secrets of Smart Cards*. Springer, 2007. ⟨ p. 49 ⟩

[102] Maplesoft. Maple computer algebra system. http ://www.maplesoft.com/. ⟨ p. 15 ⟩

[103] P. Markstein. *IA-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000. ⟨ pp. 11, 32 ⟩

[104] S. Mathew, M. Anders, R. K. Krishnamurthy, and S. Borkar. A 4-GHz 130-nm address generation unit with 32-bit sparse-tree adder core. *IEEE Journal of Solid-State Circuits*, 38(5) :689–695, May 2004. ⟨ p. 44 ⟩

[105] U. M. Maurer. A universal statistical test for random bit generators. *Journal of Cryptology*, 5(2) :89–105, January 1992. ⟨ p. 58 ⟩

[106] N. Méloni. *Arithmétique pour la Cryptographie basée sur les Courbes Elliptiques*. PhD thesis, Université Montpellier 2, 2007. ⟨ p. 49 ⟩

[107] G. Melquiond. GAPPA : génération automatique de preuves de propriétés arithmétiques. http ://gappa.gforge.inria.fr/. INRIA. ⟨ pp. 15, 61, 65, 75 ⟩

[108] G. Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. PhD thesis, École Normale Supérieure de Lyon, November 2006. ⟨ pp. 15, 56 ⟩

[109] D. Ménard and O. Sentieys. Automatic evaluation of the accuracy of fixed-point algorithms. In C. D. Kloos and J. da Franca, editors, *Proc. Design, Automation and Test in Europe (DATE)*, pages 529–537, March 2002. ⟨ pp. 62, 65 ⟩

[110] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. ⟨ p. 48 ⟩

[111] R. Michard. *Opérateurs arithmétiques matériels optimisés*. PhD thesis, École Normale Supérieure de Lyon, June 2008. ⟨ pp. 40, 47, 51 ⟩

[112] J.-M. Muller. *Arithmétique des ordinateurs*. Masson, 1989. ⟨ p. 8 ⟩

[113] J.-M. Muller. *Elementary Functions : Algorithms and Implementation*. Birkhäuser, 2nd edition, 2006. ⟨ pp. 11, 28, 29, 61, 63, 68, 69 ⟩

[114] NIST. Security requirements for cryptographic modules. FIPS 140-2, May 2001. ⟨ p. 58 ⟩

[115] S. F. Oberman and M. J. Flynn. An analysis of division algorithms and implementations. Technical Report CSL-TR-95-675, Computer Systems Laboratory, Dept. on Electrical Engineering and Computer Science Stanford University, 1995. ⟨ p. 50 ⟩

[116] S. F. Oberman and M. J. Flynn. Design issues in division and other floating-point operations. *IEEE Transactions on Computers*, 46(2) :154–161, February 1997. ⟨ pp. 42, 50 ⟩

[117] S.F. Oberman and M.J. Flynn. Division algorithms and implementations. *IEEE Transactions on Computers*, 46(8) :833–854, August 1997. ⟨ p. 11 ⟩

[118] A. Omondi and B. Premkumar. *Residue Number Systems : Theory and Implementation*, volume 2 of *Advances in Computer Science and Engineering*. Imperial College Press, 2007. ⟨ p. 10 ⟩

[119] B. J. Phillips, D. R. Kelly, and B. W. Ng. Estimating adders for a low density parity check decoder. In F.T. Luk, editor, *Proc. Advanced Signal Processing Algorithms, Architectures, and Implementations XVI*, volume 6313, page 631302, San Diego, CA, USA, August 2006. SPIE. ⟨ p. 58 ⟩

[120] S. K. Raina. *FLIP : a Floating-Point Library for Integer Processors*. PhD thesis, École Normale Supérieure de Lyon, September 2006. ⟨ pp. 41, 43 ⟩

[121] E. Remes. Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *C.R. Acad. Sci. Paris*, 198 :2063–2065, 1934. ⟨ p. 63 ⟩

[122] F. Restrepo. *Programming Tools for Mono-Chip Parallel Computers.* PhD thesis, Swiss Federal Institute of Technology Lausanne, 2001. ⟨ p. 23 ⟩

[123] W. Schindler. *Cryptographic Engineering*, chapter 2. Random Number Generators for Cryptographic Applications. Springer, 2009. ⟨ p. 57 ⟩

[124] W. Schindler. *Cryptographic Engineering*, chapter 3. Evaluation Criteria for Physical Random Number Generators. Springer, 2009. ⟨ p. 57 ⟩

[125] J. Sparso and S. Furber, editors. *Principles of Asynchronous Circuit Design : A Systems Perspective.* Kluwer, 2001. ⟨ p. 35 ⟩

[126] The Coq Development Team. The Coq proof assistant. http ://coq.inria.fr/. INRIA. ⟨ p. 15 ⟩

[127] N. Veyrat-Charvillon. *Opérateurs arithmétiques matériels pour des applications spécifiques.* PhD thesis, École Normale Supérieure de Lyon, June 2007. ⟨ pp. 40, 47, 49, 51 ⟩

[128] N.H.E. Weste and D. Harris. *CMOS VLSI Design : A Circuits and Systems Perspective.* Addison Wesley, third edition, 2004. ⟨ pp. 22, 23 ⟩

[129] R. Zimmermann. *Binary Adder Architectures for Cell-Based VLSI and their Synthesis.* PhD thesis, Swiss Federal Institute of Technology Zurich, 1998. ⟨ p. 11 ⟩

Annexe A

Sélection de publications

A.1. Opérateurs en arithmétique en ligne pour le contrôle numérique

Reproduction de l'article court [25] paru dans le journal *IEEE/ASME Transactions on Mechatronics* en juin 1999 et résumé en section 2.1.

auteurs	M. Dimmler, A. Tisserand, U. Holmberg et R. Longchamp
titre	*On-Line Arithmetic for Real-Time Control of Microsystems*
journal	*IEEE/ASME Transactions on Mechatronics*
date	juin 1999
volume	4
numéro	2
pages	213–217
numéro DOI	10.1109/3516.769548

Short Papers

On-Line Arithmetic for Real-Time Control of Microsystems

Martin Dimmler, Arnaud Tisserand, Ulf Holmberg, and Roland Longchamp

***Abstract*—The integration of microcontrollers within mechanical systems is a current trend. However, decreasing the size of the system and satisfying higher precision requirements make it necessary to reevaluate the common signal processing techniques for controller implementations, because limited controller size, computation speed, and power consumption become major topics. In this paper, we demonstrate that serial computations with the most significant digits first, that is, *on-line arithmetic*, offer an important potential for real-time control. They enable a combination of traditional functions, such as analog-to-digital converters and control data computations. This leads to very efficient controller implementations with small size, high speed, and low power consumption. After a brief description of the requirements and challenges of microsystem controller design, the use of on-line arithmetic for real-time control is proposed. A short introduction to on-line arithmetic is given and control-specific implementation guidelines are presented and finally applied to a simple test system.**

***Index Terms*—Low power consumption, microsystem control, miniaturization, on-line arithmetic, real-time control.**

I. Introduction

The design and manufacturing of mechanical components and systems have reached a very high standard. Combined with the low-cost integration of microelectronics, this offers new possibilities for compact high-precision mechanisms. Several applications have already appeared on the market, e.g., drives, robots, or fine-positioning devices. They are mostly controlled by digital controllers, such as microcontrollers, *digital signal processors* (DSP's), or *application-specific integrated circuits* (ASIC's) with fixed parameters. These circuits are generally based on digit-parallel arithmetic operators which are sequentially scheduled by an instruction set in the memory [see Fig. 1(a)]. However, in most mechatronic systems, these general purpose solutions are only necessary during controller development. Afterwards, at run time, the controller repeats a certain number of operations cyclically with very few user interactions. The whole control algorithm could be implemented using a complex operator. This avoids communication delays between memory and arithmetic logic unit (ALU) and offers, in particular, for multiple input multiple output (MIMO) systems, a potential for efficient parallel computation of independent terms [see Fig. 1(b) and (c)]. The inherent disadvantage of using digit-parallel arithmetic for these special operators is the high gate number, resulting in increased space and power consumption. However, in particular, in microsystem technology, small dimensions, low power consumption, and high controller speed are the most important controller requirements. For example, in many mobile and aerospace applications, battery lifetime and system dimensions play a major role.

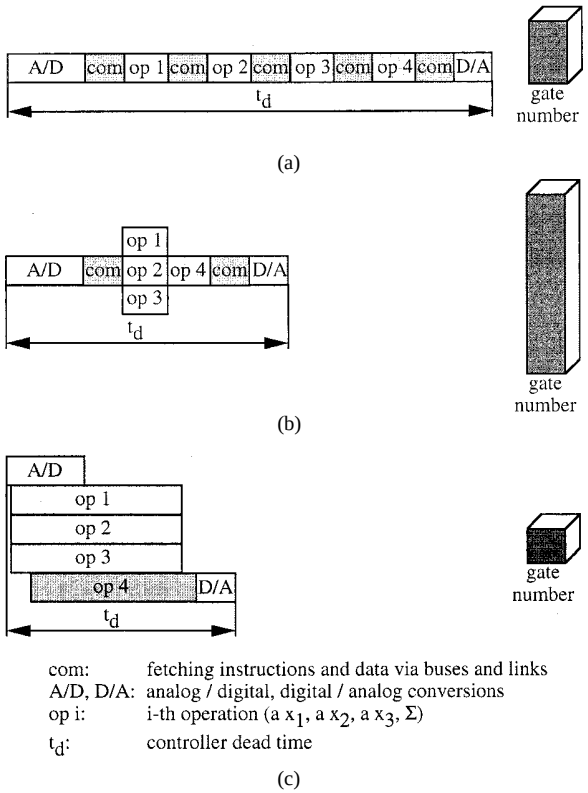


Fig. 1. Timing and size aspects of the computation $ax_1 + bx_2 + cx_3$ with different operational schemes. (a) Sequential processing with simple operators. (b) Complex digit parallel operator. (c) On-line arithmetic operators (MSDF).

In the past, digit-serial *least significant digits first* (LSDF) arithmetic has often been suggested to reduce the computation complexity [1], [2]. Its main advantages are as follows:

- simplicity of the basic operators (digit level);
- serial communication (few I/O pins);
- potential overlapping of several operations.

However, two main disadvantages make the LSDF approach too slow for real-time control of microsystems. First, sequential A/D converters work in the *most significant digits first* (MSDF) direction. Consequently, large delays are necessary to transform their outputs to LSDF form. Secondly, multiplications in the LSDF mode produce the unused least significant half of the result first. In particular, for control algorithms with many multiplications, computation time or necessary clock speed increase significantly.

Herein, the new concept of using a known MSDF serial arithmetic, so called on-line arithmetic, in real-time control systems will be

Manuscript received February 25, 1997; revised January 21, 1998 and December 11, 1998. Recommended by Technical Editor K.-M. Lee. This work was supported in part by the Centre Suisse d'Electronique et de Microtechnique.

M. Dimmler and R. Longchamp are with the Institut d'Automatique, Ecole Polytechnique Fédérale de Lausanne, CH-1015 Lausanne, Switzerland (e-mail: martin.dimmler@epfl.ch; roland.longchamp@epfl.ch).

A. Tisserand is with the Advanced Microelectronics Division, Centre Suisse d'Electronique et Microtechnique, CH-2007 Neuchâtel, Switzerland (e-mail: Arnaud.Tisserand@csemne.ch).

U. Holmberg is with the Centre for Computer System Architecture, Halmstad University, S-30118 Halmstad, Sweden (e-mail: Ulf.Holmberg @cca.hh.se).

Publisher Item Identifier S 1083-4435(99)04820-6.

1083–4435/99$10.00 © 1999 IEEE

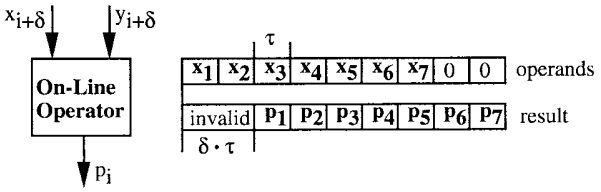


Fig. 2. Delay and clock period of on-line operations.

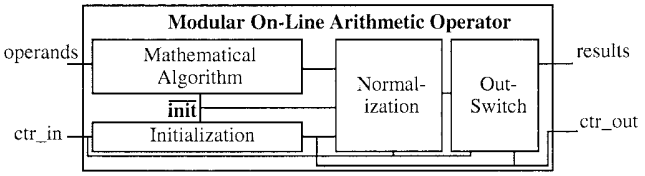


Fig. 3. Modular on-line arithmetic operator.

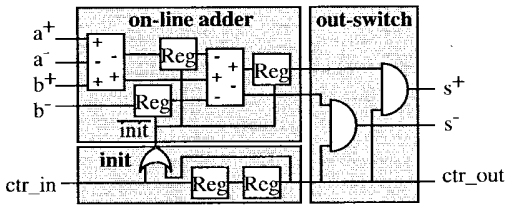


Fig. 4. On-line adder modified for real-time control.

introduced. The change of calculation direction permits an overlap of computation and A/D conversion [see Fig. 1(c)]. This property, undiscovered until now, offers an additional time interval for direct calculations which has not been used previously, neither by parallel nor by digit-serial LSDF arithmetic. Moreover, for on-line arithmetic, the advantages mentioned above of serial computation are still valid and the additional multiplication delays disappear. Therefore, the designs have low gate number, small calculation time, and low power consumption. In previous studies, Ercegovac [3] and Moran [4] have tried to bring the theoretical results of on-line arithmetic closer to practical use. However, until now, some important implementation concepts have been omitted. For example, the need for a normalization algorithm after operations was not mentioned in [3]. In [4] and [5], normalization was pointed out to be essential, but the given solution does not solve the general form of the problem (no multiadder considered). In this paper, real-time control-specific issues are introduced which make the given concept useful for controller implementations. These guidelines are demonstrated herein with a proportional integral differential (PID) controller. A complex aerospace application is currently being investigated, but in order to focus more on the conceptual perspective, the presentation is restricted to this simple implementation example.

This paper is organized as follows. In Section II, a short introduction to on-line arithmetic is given. In Section III, implementation issues are discussed, leading to a modular on-line library. In Section IV, this library is used for the PID controller. The result is compared to an optimized parallel design. A conclusion and future developments are presented in Section V.

II. Introduction to On-Line Arithmetic

For the reason of size restrictions, only a short introduction is given here. Further details can be found in [3], [6], and [7].

In on-line arithmetic, the operands, as well as the results, flow through arithmetic units in a digit serial fashion starting with the MSDF.

Important characteristics of on-line operators are their *delay* (δ) and *period* (τ), as illustrated in Fig. 2. Serial computations with the MSDF become possible owing to a change in the number system. The used redundant number systems [8] permits negative digit values, i.e., the number $0.a_1a_2 = \sum_{i=1}^{2} a_i r^{-i}$ of radix $r = 2$ can have negative a_i. This leads to several representations for some numbers ($\frac{1}{4} = 0.a_1a_2$ with $a_1 = 0$ and $a_2 = 1$ or $a_1 = 1$ and $a_2 = -1$). With radix 2, usually a 2-bit representation (a_i^+, a_i^-), called borrow–save, of the digits (a_i) is defined as follows: $a_i = a_i^+ - a_i^-$. Thus, digit 1 is represented by (1, 0), digit -1 by (0, 1), while digit 0 has two possible representations, namely (0, 0) and (1, 1).

This special digit-serial arithmetic was introduced by Ercegovac and Trivedi in 1977 [9]. Nowadays, on-line algorithms are available for all common arithmetic operations, in the fixed-point representation, as well as in the floating-point representation, but they have been rarely used in hardware applications. This is mainly due to the different original motivation (high-precision computation) and the lack of a convenient formulation for an efficient hardware implementation. In recent years, more effort has been spent on implementation issues (e.g., [6] and [10]). Two different approaches have been chosen. One follows the recursive formulation of Ercegovac [10] and the other is based on Avizienis' parallel adder (Fig. 8). The latter leads to much smaller implementations, but is limited to a few operations (addition, multiplication). The application presented here is mainly concerned with size and power consumption requirements, and additions/multiplications represent the majority of the operations. Therefore, the second approach has been focused on. The functional schemes of the multiplier and adder, as well as their building blocks, are shown in Figs. 7, 4, 8, and 9, respectively.

III. Implementation of On-Line Arithmetic for Control Algorithms

Previous literature focused rather on single on-line operations than on their interconnection to implement complex algorithms. Consequently, no uniform framework existed and, usually, arithmetic experts were needed for the implementation of specific algorithms. In this section, we supply the implementation guidelines necessary for the realization of a modular library of basic on-line operations. Each operator is composed of four main building blocks: initialization, mathematical algorithm, normalization, and output switch (see Fig. 3). The different mathematical algorithms can be found in the literature (e.g., [6]). The other three blocks are explained in more detail below. The common interface of these *modular on-line arithmetic operators* enables system designers to construct mechatronic controllers in on-line arithmetic without advanced knowledge in computer arithmetic.

A. Initialization of On-Line Operators

In digit-serial arithmetic, the operands are distributed over several subsequent operations (operators work digit wise), and there is an internal state update in the operators at each clock period. Therefore, a clear indication of every operation start is necessary for initialization. A simple way to achieve this is by a distributed control scheme in the form of an additional control line synchronized to the operands. The line is kept high if significant operand digits are present and is otherwise low. Internal state and status values (e.g., intermediate results in multiplications) are thereby reset as soon as an operator is unused. In Fig. 4, the initialization (init) is shown for an on-line adder. The two registers in the init block are necessary to compensate for the operator delay ($\delta_{adder} = 2$). As soon as $ctr_{\text{in}} = ctr_{\text{out}} = 0$,

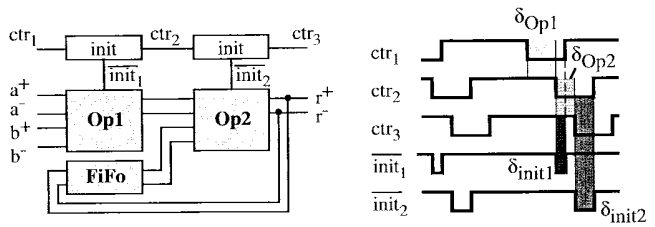


Fig. 5. Initialization and synchronization of on-line operators, $\overline{init} = ctr_{in} \vee ctr_{out}$.

the three registers of the adder are reset. The initialization takes at least one clock cycle (see Fig. 5).

In the initialization scheme, the digits of the result must have left the operator entirely before the reset was achieved. Otherwise, the last δ (on-line delay) digits of the result would be wrong. Therefore, at least $(\delta_{\max} + 1)$ intermediate zeros between the operands must be inserted at algorithm entry, where $\delta_{\max}$ is the largest delay of all of the operators in the entire algorithm. In Fig. 5, an algorithm is supposed to have two operators and $\delta_{\max} = \delta_{Op1} > \delta_{Op2}$. The additional delay is introduced for the initialization. The zeros increase the sampling period. However, in order to ensure the correct timing of the controller (sampling instant and termination of the D/A conversion at the same time), many more intermediate zeros are needed anyway. In the case where converter and arithmetic have the same resolution, the number of zeros is determined by $\delta_{\text{zeros}} = \delta_{\text{D/A}} + \delta_{\text{Sampler}} + \delta_{\text{Arithmetic}}$, where $\delta_{\text{D/A}}$, δ_{Sampler}, $\delta_{\text{Arithmetic}}$ are the delays of the D/A converter, the sampler of the A/D converter, and the controller arithmetic, respectively.

In order to avoid an interference from subsequent numbers, these intermediate zeros have to be maintained, even after several operations in the algorithm. This output switching can be realized with the additional control line (see Fig. 4, out-switch block).

This distributed control scheme improves the modularity of the design. Controller execution can be stopped easily by resetting the registers in the initialization block.

B. Synchronization

Implementations of dynamic systems give rise to loops in the signal flows due to the states. These states need to be synchronized to the signals in the forward flows. In Fig. 5, the output of the second operator (Op2) is a state that is used as input to Op2 at the next sampling instant. For synchronization, shift registers of size n (number resolution) are included in the backward branch. The shift operation is controlled by simple glue logic based on the control line ctr_3.

C. Normalization

In redundant number systems, some numbers can have several representations (e.g., $1-1/4 = 1.0(-1) = 0.11 = 1/2+1/4$ in radix 2, notations as in Section II). This property allows that in additions the sum can be represented by $n+1$ valuable digits whereas the operands and the theoretical result, using a nonredundant number system, only need n digits. In multiadders, even several digits in front of the point are possible. In order to avoid a continuously growing number of digits after additions, especially in state loops (growing number of additions), a conversion to a limited representation is necessary. In the literature, two approaches have been given. One is the complete on-the-fly normalization algorithm by Ercegovac and Lang [11] which converts the redundant numbers into conventional digital representations. Another is Merrheim's normalization algorithm [5] which generates a redundant fractional number with zero unit part (i.e., $0.s_1 s_2 s_3 \cdots$). However, the former causes a delay of n clock cycles and the latter is only appropriate for additions of two numbers (but without on-line delay). Herein, an extension of Merrheim's algorithm also suitable for multiadditions is proposed.

Proposition: With an appropriate scaling of the operands ($|s| < 1$) only two types of result need conversion (notations as in the numerical examples above)

$$\begin{aligned} & 1(1-r)\cdots(1-r).0\cdots 0(-a)s_{m+1}\cdots s_n \\ \rightarrow\ & 0.(r-1)\cdots(r-1)(r-a)s_{m+1}\cdots s_n \qquad (1) \\ & (-1)(r-1)\cdots(r-1).0\cdots 0as_{m+1}\cdots s_n \\ \rightarrow\ & 0.(1-r)\cdots(1-r)(a-r)s_{m+1}\cdots s_n \qquad (2) \end{aligned}$$

where r is the radix and a, $(-a)$ satisfying $0 < a < r$, are the first nonzero digits after the decimal point.

Proof: Consider s_i and s_i' to be the digits before and after the conversion, respectively. Suppose there are $k+1$ nonzero digits before the decimal point in (1). Then, up to the mth digit $\sum_{i=-k}^{m} s_i r^{-i} = r^k + (1-r)\sum_{i=0}^{k-1} r^i - ar^{-m} = 1 - ar^{-m} = (r-1)\sum_{i=1}^{m-1} r^{-i} + (r-a)r^{-m} = \sum_{i=1}^{m} s_i' r^{-i}$. Conversion of (2) is shown similarly. □

Remark: In case of overflow, the closest possible value appears on the output ($0.(r-1)\cdots(r-1)$ or $0.(1-r)\cdots(1-r)$, respectively). The algorithm was implemented for radix 2 in a field programmable gate array (FPGA) and requires approximately the space of 20 Actel_2 cells. This is not significant if used only a few times in a design (operator combinations reduce occurrence, see Section III-E).

Two additional properties should be mentioned in order to use most advantageously the introduced *modular on-line arithmetic operators*. First, appropriate controller representations, and secondly, simplifications by using multioperations are presented.

D. Appropriate Controller Representation

The controller representation has a significant influence on calculation speed and implementation complexity. A state-space representation based on the Jordan form offers many advantages for the implementation in on-line arithmetic. The state updates and the output equations can be calculated in parallel and the constants in the dynamic matrix are scaled ($|a_{ii}| < 1$ for stable controllers). In particular, the latter is important because it keeps the delay of the multipliers small (for fixed-point operands, the range of multiplicative constants has a significant influence on the operator delay). The controller deadtime for fast sampling designs is exactly one sampling period. Consequently, the controller has no direct term. The deadtime of one period has to be taken into account during controller design.

This transformation is highlighted by an implementation example. The aim herein is to introduce the concept of on-line arithmetic for real-time control, rather than to describe a specific application. Therefore, the simple PID algorithm with filtered differential part is treated

$$u_k = K\left(e_{k-1} + \frac{h}{T_I}x_{i,k-1} + T_D x_{d,k-1}\right)$$

with

$$\begin{aligned} x_{i,k-1} &= e_{k-1} + x_{i,k-2} \\ x_{d,k-1} &= \gamma(e_{k-1} - e_{k-2} + \tau x_{d,k-2}) \end{aligned} \qquad (3)$$

where $e_k = s_k - y_k$ is the difference between set point s_k and measurement y_k, $x_{i,k}$ and $x_{d,k}$ the controller states, u_k the controller output, h the sampling period, τ the time constant of the differential part filter, $\gamma = 1/(h+\tau)$ and K, T_I, T_D are the proportional, integral, and differential gain, respectively.

Transformation to Jordan form gives

$$z_{k+1} = Az_k + Be_{k-1}$$
$$u_k = Cz_k + De_{k-1} \quad (4)$$

where

$$A = \begin{pmatrix} \tau\gamma & 0 \\ 0 & 1 \end{pmatrix}$$
$$B = \begin{pmatrix} -T_D\gamma(1-\tau\gamma) \\ \dfrac{h}{T_I} \end{pmatrix} K$$
$$C = (1 \quad 1)$$
$$D = \left(1 + \frac{h}{T_I} + T_D\gamma\right) K.$$

E. Simplification by Using Multioperations

For $Matrix \times Vector$ computations, on-line arithmetic offers an efficient simplification. The static logic part of all multiplications can be executed in parallel without on-line delay, and the final addition can be performed by a simplified final adder with a much smaller delay than a binary tree of adders. The algorithm is given in [6]. This simplification keeps the overall delay of the controller, as well as the number of individual operators, very small. The consequence is a reduced calculation time and a small number of necessary normalization units (scaling is limited to a few intermediate results). In our example, every row of (4) is calculated by a simplified multiadder (see Fig. 6).

F. Hardware and Software Support

With programmable logic, such as FPGA's, efficient implementations of algorithms in hardware can be made with software-like design principles (for examples, see [12]). These gate arrays can be used as testbeds for later implementations in ASIC's or directly as mechatronic controllers. The controller algorithms are herein composed in a graphical or formal description language (e.g., VHDL[1]). A formal description is generally preferred because of its greater possibilities for validation and implementation on different hardware platforms. Once the basic arithmetic modules are available, implementing a controller using on-line arithmetic is very similar to standard arithmetic approaches.

IV. Implementation

A prototype board holding 2 Actel FPGA's (1 × 1240A, 1 × 1280A), some logic for a PC bus interface, as well as 2 A/D and 2 D/A converters was built. It enables the implementation of different controllers and the monitoring of run-time values via a PC bus interface. In order to verify our design rules , we implemented the PID scheme for resolutions of 12 bits and 24 bits in the Actel 1280A device (a low-cost FPGA in antifuse technology; for specification details see [13]). The resulting operational scheme for our PID example is shown in Fig. 6 for a resolution of n bits. It includes two inner loops with appropriate registers and three $Vector \times Vector$ operators with simplified final adders. In order to reduce the length of the critical combinatorial paths between the entry of the subtraction and the output of the multipliers, the inputs of the multipliers are delayed by one clock cycle. This reduces the period τ. Due to the choice of reasonable controller constants and an analog output gain, the on-line delay of the controller is $\delta_{\text{arithmetic}} = 6$. For the chosen converters, the times $\tau \times \delta_{\text{sampler}} = 1\ \mu$ and $\tau \times \delta_{D/A} = 0.4\ \mu$s

[1]VHDL stands for VHSIC (very-high-speed integrated circuit) hardware description language.

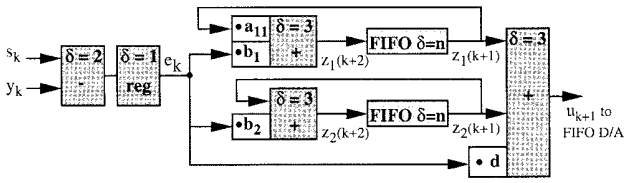


Fig. 6. Arithmetic scheme of the PID controller. The control line is not indicated, and the coefficients correspond to the matrices in (4).

TABLE I
PID in On-line and Digit-Parallel Arithmetic

Criteria	Digit-Parallel	On-Line
12 bits		$\delta_{zeros} = 7$
Clock Speed	640 kHz	760 kHz
Actel Cells	1450 (118 %)	645 (54%)
24 bits		$\delta_{zeros} = 8$
Clock Speed	1.2 MHz	1.3 MHz
Actel Cells	3700 (300 %)	1100 (89 %)

are given. The sampling time was fixed to a reasonable value of 25 μs. Resulting clock frequencies and number of intermediate zeros are listed in Table I.

We compared our design to a sequential realization in digit-parallel arithmetic (see Table I). For the digit-parallel operators, a carry-look-ahead adder and a Wooley [14] multiplier were used. Only u is calculated immediately after the A/D conversion. The state updates of *z1* and *z2* are made in parallel to the following conversion. Owing to the simplicity of the PID controller, the clock frequencies are comparable for the two approaches. However, the on-line solutions are much smaller and fit on one single FPGA, even for 24-bit resolution. The power consumption could not be evaluated explicitly (FPGA implementation), but it would certainly be higher in the parallel case (more gates and bus traffic).

Note that more drastic savings in size and power consumption can be expected for a higher complexity of the controller, e.g., multivariable controllers with coordinate transforms and sensor preconditioning. In particular, nonlinear operations, as for example trigonometric functions in control of rotational magnetic bearings or divisions in advanced friction compensation based on the new *LuGre model* [15], are very costly for digit-parallel arithmetic (in size and speed). In on-line arithmetic, these operations still grow only linearly with resolution and can be put in parallel with other computations.

V. Conclusions and Future Developments

After a discussion of the main challenges in microsystems controller design, it has been shown that a digit-serial processing scheme based on *on-line arithmetic* is particularly well suited for real-time control. Some guidelines for their control-specific implementation have been presented and verified for a simple implementation example. Their small size, high speed, and low power consumption make them well suited to microsystems.

In parallel to this work, the use of radix 4 was investigated. In [16], it has been shown that radix 4 offers a lower maximum delay and calculation time due to the much shorter representation on the digit level. However, operator complexity is significantly higher than in radix 2. Therefore, more attention was given to radix 2.

With the PID example, the concept was verified. The next challenge will be a controller implementation for a complex mechatronic system, including sensor signal calibration and transformation. Demands

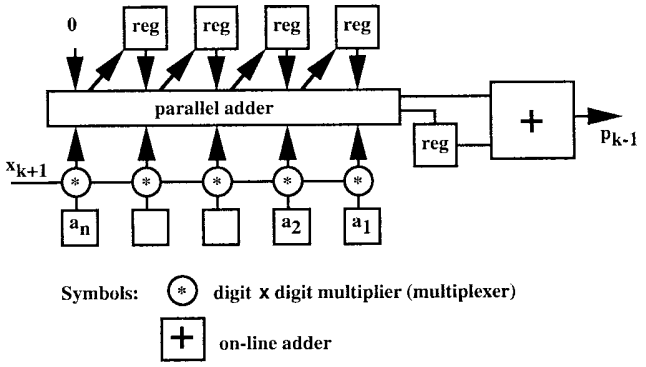


Fig. 7. On-line arithmetic multiplier by a constant number.

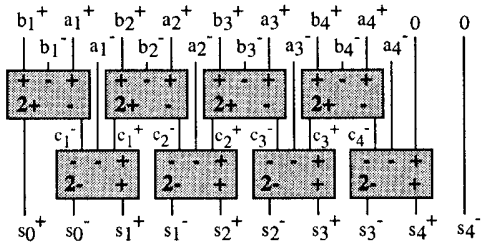


Fig. 8. Avizienis' parallel adder (indexes indicate ranks) [6].

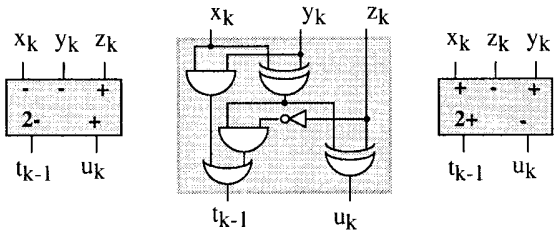


Fig. 9. Minus–minus–plus (MMP) and plus–plus–minus (PPM) cells (indexes indicate ranks).

have arisen from two application areas. One is an optical terminal for satellite communication where a triangular pointing mirror, driven by three actuators, one in each mirror corner, has to follow a fast reference signal. The other is a magnetic bearing for the actuation and guidance of a computer hard disk. In both cases, very fast controllers with strict size and power consumption constraints are required. An on-line implementation for these systems has, in addition to the arguments presented above, the important advantage that a large number of sensor interfaces can be realized in a serial manner. This reduces significantly interconnection constraints and communication delays.

APPENDIX

The functional scheme of the on-line arithmetic multiplication with a constant number and its necessary building blocks are displayed in Figs. 7–9. The on-line addition can be found in Fig. 4. For the construction of multiadders and more details, see [6].

ACKNOWLEDGMENT

The authors would like to thank the members of the Centre Suisse d'Electronique et de Microtechnique, in particular, Dr. J. Moerschell, for many fruitful discussions, as well as Prof. J.-M. Muller and Prof. J.-D. Nicoud for their comments.

REFERENCES

[1] R. Hartley and P. Corbett, "Digit-serial processing techniques," *IEEE Trans. Circuits Syst.,* vol. 37, pp. 707–719, June 1990.
[2] P. B. Denyer and S. G. Smith, "Advanced Serial-Data Computation," *J. Parallel Distributed Comput., (Special Issue on Parallelism in Computer Arithmetic),* vol. 5, no. 3, pp. 228–249, 1988.
[3] M. D. Ercegovac and T. Lang, "On-Line arithmetic: A design methodology and applications in digital signal processing," in *VLSI Signal Processing,* R. W. Brodersen and H. S. Moscowitz, Eds. New York: IEEE Press, 1988, vol. III, pp. 252–263.
[4] J. Moran, I. Rios, and J. Meneses, "Signed digit arithmetic on FPGA's," presented at the Int. Workshop on FPGA: Logic and Applications, Oxford, U.K., Sept. 1993.
[5] X. Merrheim, "Bases discretes et calcul des fonctions elementaires par materiel," Ecole Normale Superieure de Lyon, Lyon, France, Feb. 1994.
[6] J. C. Bajard, J. Duprat, S. Kla, and J. M. Muller, "Some operators for on-line radix-2 computations," *J. Parallel Distrib. Comput.*, vol. 22, no. 2, pp. 336–345, 1994.
[7] M. D. Ercegovac, "On-line arithmetic: An overview," in *Proc. SPIE Conf. Real-Time Signal Processing,* 1984, pp. 86–93.
[8] A. Avizienis, "Signed-digit number representations for fast parallel arithmetic," *IRE Trans. Electron. Computers.*, vol. 10, pp. 389–400, 1961.
[9] M. D. Ercegovac and K. S. Trivedi, "On-line algorithms for division and multiplication," *IEEE Trans. Computers,* vol. C-26, pp. 681–687, July 1977.
[10] P. Tu, "On-line arithmetic algorithms for efficient implementation," Comput. Sci. Dep., Univ. California, Los Angeles, 1990.
[11] M. D. Ercegovac and T. Lang, "On-the-fly Conversion of redundant into conventional representations," *IEEE Trans. Comput.*, vol. C-36, pp. 895–897, July 1987.
[12] R. Kasper, "Motion control with field programmable gate arrays," in *MOVIC Conf. Tutorial: Tools for Real-Time Motion Control,* Zurich, Switzerland, 1998, pp. 1–20.
[13] *ACT Family FPGA Databook*, Actel Corp., Sunnyvale, CA, 1996.
[14] J. M. Muller, *Arithmetique des Ordinateurs.* Paris, France: Masson, 1989.
[15] C. Canudas de Wit, H. Olsson, K. J. Aström, and P. Lischinsky, "A model for control of systems with friction," *IEEE Trans. Automat. Contr.*, vol. 40, pp. 419–425, Mar. 1995.
[16] R. Forest, "Arithmetique on-line en base 4 pour les controlleurs digitaux en automatique," Ecole Polytechnique de Lausanne, Lausanne, Switzerland, June 1997.

A.2. Opérateurs à base de petits multiplieurs pour l'évaluation de fonctions

Reproduction de l'article [27] paru dans le journal *IEEE Transactions on Computers* en juillet 2000 et résumé en section 2.3.1.

auteurs	M. D. Ercegovac, T. Lang, J.-M. Muller et A. Tisserand
titre	*Reciprocation, Square Root, Inverse Square Root, and Some Elementary Functions Using Small Multipliers*
journal	*IEEE Transactions on Computers*
date	juillet 2000
volume	49
numéro	7
pages	628–637
numéro DOI	10.1109/12.863031

Reciprocation, Square Root, Inverse Square Root, and Some Elementary Functions Using Small Multipliers

Milos D. Ercegovac, *Member*, *IEEE*, Tomás Lang, *Member*, *IEEE Computer Society*, Jean-Michel Muller, *Member*, *IEEE Computer Society*, and Arnaud Tisserand, *Member*, *IEEE*

Abstract—This paper deals with the computation of reciprocals, square roots, inverse square roots, and some elementary functions using small tables, small multipliers, and, for some functions, a final "large" (almost full-length) multiplication. We propose a method, based on argument reduction and series expansion, that allows fast evaluation of these functions in high precision. The strength of this method is that the same scheme allows the computation of all these functions. We estimate the delay, the size/number of tables, and the size/number of multipliers and compare with other related methods.

Index Terms—Reciprocal, square root, inverse square root, logarithm, exponential, single-/double-precision operations, small multipliers, Taylor series.

1 Introduction

The computation of reciprocal and square root has been considered of importance for many years since these functions appear in many applications. Recently, inverse square root has also received attention because of the increased significance of multimedia and graphics applications. Moreover, because of their similar characteristics, it is considered advantageous to have a single scheme to implement all three functions. We consider such a scheme here. In addition, it allows the computation of logarithms and exponentials.

The progress in VLSI technology now allows the use of large tables with short access time. As a consequence, many methods using tables have emerged during the last decade: high-radix digit-recurrence methods for division and square root [1], [15], inverse square root [16], convergence methods for division and square root [9], combination of table-lookup and polynomial approximation for the elementary functions [8], [6], [12], or (for single precision) use of table-lookups and addition only [14], [3], [10].

We are interested in computations in high precision, such as IEEE-754 double-precision (53-bit significand) format. For double precision, these are hard to achieve with today's technology by direct table lookup, tables and additions, or linear approximations.

The standard scheme to compute reciprocal, square-root, and inverse square root with high precision is based on Newton-Raphson iterations. Although the scheme has a quadratic convergence, the iterations consist of multiplications and additions and are therefore relatively slow. A variation of this method is presented in [5].

We now briefly review other methods. In [2], a method to compute reciprocal, square root, and several elementary functions is presented (and probably could also implement inverse square root). The method is based on series expansion and the implementation consists of several tables, two multipliers, and an adder. For an approximation with relative error 2^{-m}, the tables have about $m/3$ input bits, which is too large for double precision.

In [13], a method is proposed for double-precision calculations. This requires several tables with an input of 10 bits and rectangular multiplications (typically of 16×56 bits). In Section 5, we compare this scheme with the one presented here.

The bipartite table methods [10], [3], [11] require the use of tables with approximately $2m/3$ address bits and do not need multiplications to get the result (an addition suffices). These methods might be attractive for single precision calculations, but, with currently available technology, they would require extensively large tables for double precision calculations.

In this paper, we propose a unified algorithm that allows the evaluation of reciprocal, square root, inverse square root, logarithm, and exponential, using one table access, a few "small" multiplications, and at most one "large" multiplication. To approximate a function with about m-bit accuracy, we use tables with $m/4$ address bits. This makes our method suitable up to and including double precision.

- *M.D. Ercegovac is with the Computer Science Department, 4731 Boelter Hall, University of California at Los Angeles, Los Angeles, CA 90095. E-mail: milos@cs.ucla.edu.*
- *T. Lang is with Department of Electrical and Computer Engineering, University of California at Irvine, Irvine, CA 92697. E-mail: tlang@ece.uci.edu.*
- *J.-M. Muller is with CNRS-LIP, Ecole Normale Supérieure de Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France. E-mail: jmmuller@ens-lyon.fr.*
- *A. Tisserand is with INRIA-LIP, Ecole Normale Supérieure de Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France. E-mail: Arnaud.Tisserand@ens-lyon.fr.*

Manuscript received 1 Sept. 1999; revised 1 Feb. 2000; accepted 10 Mar. 2000. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 111784.

0018-9340/00/$10.00 © 2000 IEEE

As in other methods of this type, it is possible to obtain an error which is bounded (say by 2^{-m}). However, it is not possible in general to obtain results that can be directly rounded to nearest. It has been shown [4] that, for the special cases of reciprocal and square root to round to m bits, it is sufficient to compute a result with an error of less than 2^{-2m}. Similarly, for inverse square root, the error has to be less than 2^{-3m}. Since this overhead in accuracy might be prohibitive, another alternative is to produce an error of less than 2^{-m-1} and determine the exact value by computing the corresponding remainder, which is possible for division and square-root, but not for the transcendental functions [8], [7]. We do not discuss this issue further in this paper and, in the sequel, we will aim to obtain an error which is less than 2^{-m}, for m-bit operands.

2 Reciprocal, Square Root, and Inverse Square Root

We want to evaluate reciprocals, square roots, and inverse square roots for operands and results represented by m-bit significands. To achieve an error which is smaller than 2^{-m}, we use an internal datapath of n bits (to be determined) with $n > m$. We do not consider the computation of the exponent since this is straightforward.

Let us call the generic computation $g(Y)$, where Y is the m-bit input significand and, as in the IEEE standard, $1 \le Y < 2$. The method is based on the Taylor expansion of the function to compute, which converges with few terms if the argument is close to 1. Consequently, the method consists of the following three steps:

1. **Reduction.** From Y, we deduce an n-bit number A such that $-2^{-k} < A < 2^{-k}$. To produce a simple implementation that achieves the required precision, we use $k = n/4$. For the functions considered, we obtain A as

$$A = Y \times \hat{R} - 1, \tag{1}$$

where $\hat{R}$ is a $(k+1)$-bit approximation of $1/Y$. Specifically, define $Y^{(k)}$ as Y truncated to the kth bit. Then,

$$Y^{(k)} \le Y < Y^{(k)} + 2^{-k}.$$

Hence,

$$1 \le \frac{Y}{Y^{(k)}} < 1 + 2^{-k}. \tag{2}$$

Using one lookup in a k-bit address table, one can find $\hat{R}$ defined as $1/Y^{(k)}$ rounded *down* (i.e., truncated) to $k+1$ bits. Then,

$$-2^{-k-1} < \hat{R} - \frac{1}{Y^{(k)}} \le 0.$$

Therefore, since $1 \le Y^{(k)} < 2$,

$$1 - 2^{-k} < \hat{R}Y^{(k)} \le 1. \tag{3}$$

From (2) and (3), we get

$$1 - 2^{-k} < \hat{R}Y < 1 + 2^{-k}. \tag{4}$$

The *reduced argument* A is such that $g(Y)$ can be easily obtained from a value $f(A)$, which is computed during the next step.

2. **Evaluation.** We compute an approximation of $B = f(A)$ using the series expansion of f, as described below.
3. **Postprocessing.** This is required because of the reduction step. Since reduction is performed by multiplication by $\hat{R}$, we obtain $g(Y)$ from $B = f(A)$ as

$$g(Y) = M \times B,$$

where $M = h(\hat{R})$. The value of M depends on the function and is obtained by a similar method as $\hat{R}$. Specifically,

- for reciprocal $M = \hat{R}$,
- for square root $M = 1/\sqrt{\hat{R}}$,
- for inverse square root $M = \sqrt{\hat{R}}$.

Hence, although $\hat{R}$ is the same for all functions considered here, M depends on the function being computed. There is a different table for M for each function we wish to implement. Let us now consider the evaluation step.

2.1 Evaluation Step

In the following, we assume that we want to evaluate $B = f(A)$, with $|A| < 2^{-k}$. The Taylor series expansion of f is

$$f(A) = C_0 + C_1A + C_2A^2 + C_3A^3 + C_4A^4 + \dots \tag{5}$$

at the origin where the C_is are bounded.

Since $-2^{-k} < A < 2^{-k}$, A has the form

$$A = A_2z^2 + A_3z^3 + A_4z^4 + \dots, \tag{6}$$

where $z = 2^{-k}$ and $|A_i| \le 2^k - 1$.

Our goal is to compute an approximation of $f(A)$, correct to approximately $n = 4k$ bits, using small multiplications. From the series (5) and the decomposition (6), we deduce

$$\begin{aligned} f(A) = C_0 &+ C_1\left(A_2z^2 + A_3z^3 + A_4z^4\right) \\ &+ C_2\left(A_2z^2 + A_3z^3 + A_4z^4\right)^2 \\ &+ C_3\left(A_2z^2 + A_3z^3 + A_4z^4\right)^3 \\ &+ C_4\left(A_2z^2 + A_3z^3 + A_4z^4\right)^4 + \dots . \end{aligned} \tag{7}$$

After having expanded this series and dropped out all the terms of the form $W \times z^j$ that are less than or equal to 2^{-4k}, we get (see the Appendix)

$$f(A) \approx C_0 + C_1A + C_2A_2^2z^4 + 2C_2A_2A_3z^5 + C_3A_2^3z^6. \tag{8}$$

We use this last expression to approximate reciprocals, square roots, and inverse square roots. In practice, when computing (8), we make another approximation: after having computed A_2^2, obtaining A_2^3 would require a $2k \times k$ multiplication. Instead of this, we take only the k most-significant bits of A_2^2 and multiply them by A_2.

Now, we determine the coefficients for the three functions

- For reciprocal, $|C_i| = 1$ for any i, and

$$\begin{aligned}\frac{1}{1+A} &\approx 1 - A_2z^2 - A_3z^3 + \left(-A_4 + A_2^2\right)z^4 \\ &\quad + 2A_2A_3z^5 - A_2^3z^6 \\ &\approx (1-A) + A_2^2z^4 + 2A_2A_3z^5 - A_2^3z^6.\end{aligned} \tag{9}$$

- For square root, $C_0 = 1$, $C_1 = 1/2$, $C_2 = -1/8$, $C_3 = 1/16$. This gives

$$\sqrt{1+A} \approx 1 + \frac{A}{2} - \frac{1}{8}A_2^2z^4 - \frac{1}{4}A_2A_3z^5 + \frac{1}{16}A_2^3z^6. \tag{10}$$

- For inverse square root, $C_0 = 1$, $C_1 = -1/2$, $C_2 = 3/8$, $C_3 = -5/16$. This gives

$$1/\sqrt{1+A} \approx 1 - \frac{A}{2} + \frac{3}{8}A_2^2z^4 + \frac{3}{4}A_2A_3z^5 - \frac{5}{16}A_2^3z^6. \tag{11}$$

2.2 Error in the Evaluation Step

We now consider the error produced by the evaluation step described above. In the Appendix, we prove the following result:

Theorem 1. *f(A) can be approximated by*

$$C_0 + C_1A + C_2A_2^2z^4 + 2C_2A_2A_3z^5 + C_3A_2^3z^6$$

(where we use the k most-significant bits[1] only of A_2^2 when computing A_2^3), with an error less than

$$2^{-4k}\left(\frac{C_{max}}{1-2^{-k}} + 3|C_2| + 4|C_3| + 8.5\max\{|C_2|, |C_3|\} \times 2^{-k}\right)$$

with $C_{max} = \max_{i\geq 4}|C_i|$, and $k \geq 5$.

In particular, assuming $|C_i| \leq 1$ for any i (which is satisfied for the functions considered in this paper), this error is less than

$$\epsilon = 2^{-4k}(1.04C_{max} + 3|C_2| + 4|C_3| + 0.27).$$

Now, we determine the error bound ϵ for the three functions, *assuming A is exactly equal to*

$$A = A_2z^2 + A_3z^3 + A_4z^4.$$

- For reciprocal, since $|C_i| = 1$ for all i,

$$\epsilon = 8.31 \times 2^{-4k}.$$

- For square root, $C_2 = 2^{-3}$, $C_3 = 2^{-4}$, and $C_{max} = 5 \times 2^{-7}$,

$$\epsilon = 0.94 \times 2^{-4k}.$$

- For inverse square root, $C_2 = 3 \times 2^{-3}$, $C_3 = -5 \times 2^{-4}$, and $C_{max} = 35 \times 2^{-7}$,

TABLE 1
Upper Bound on the Total Absolute Error

Operation	$M_{\max}$	Total Error
Reciprocal	1	9.31×2^{-4k}
Square root	$\sqrt{2}$	2.39×2^{-4k}
Inv. square root	1	3.68×2^{-4k}

$$\epsilon = 2.93 \times 2^{-4k}.$$

These errors are committed by evaluating (8) in infinite precision arithmetic (and using the k most-significant bits of A_2^2 only). To this, we have to add the following two errors:

- A has more than $4k$ bits. Consequently, we have to add the error $2^{-4k-1}\max_A f'(A)$ due to having rounded A to $A_2z^2 + A_3z^3 + A_4z^4$.
- If the evaluation step returns a value rounded to the nearest multiple of 2^{-4k}, we have to add the maximum error value 2^{-4k-1} due to this rounding.

All this gives an upper bound ϵ_{eval} due to the evaluation step.

2.3 Total Error and Value of k

We now take into account the postprocessing step (multiplication by M). To get an upper bound ϵ_{total} on the total computation error, we multiply ϵ_{eval} by the maximum possible value of M. We do not include an error due to rounding M to n bits: It is preferable to round M to m bits directly. Table 1 gives the value of ϵ_{total}. If a $(3k+1) \times (3k+2)$ multiplier is used for the postprocessing step (as suggested in Section 3.3), then we need to add 0.5×2^{-4k} to this value.

Now, let us determine the value of k. Since the computed final result $g(Y)$ is between $1/2$ and 1 for reciprocation, between 1 and $\sqrt{2}$ for square root, and between $1/\sqrt{2}$ and 1 for inverse square-root, the first nonzero bit of the result is of weight 2^0 for square-root and of weight 2^{-1} for the other two functions. Considering the error given in Table 1, the required values of n and k are given in Table 2.

3 Implementation

We now describe implementation aspects of the proposed method. Fig. 1 shows a functional representation of the general architecture. In the sequel, we assume that A is in the sign-magnitude form which requires complementation. The multiplications produce products in the signed-digit form, and the addition of the four terms in the evaluation

TABLE 2
Values of k for Functions Evaluated ($n = 4k$)

Format	Reciprocal	Square root	Inverse square root
SP ($m = 24$)	7	7	7
DP ($m = 53$)	15	14	14

SP—single precision with faithful rounding;
DP—double precision with faithful rounding.

1. It would be more accurate to say *digits* since it is likely that, in a practical implementation, A_2^2 will be represented in a redundant (e.g., carry-save or borrow-save) representation.

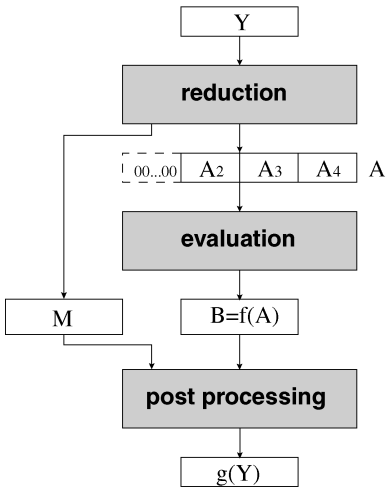


Fig. 1. General organization.

step is performed using signed-digit adders. Modifications for using different number representations are straightforward.

3.1 Reduction

Fig. 2 shows a functional representation of the reduction module which computes A and M from Y. The factor M is obtained by table lookup from block `ctb` for functions other than reciprocal.

3.2 Evaluation

The evaluation step computes (9), (10), and (11). All three require the computation of A_2^2, A_2A_3, and A_2^3. As indicated before, for A_2^3 we use the approximation

$$A_2^3 \approx (A_2^2)_{high} \times A_2.$$

Consequently, these terms can be computed by three k by k multiplications. Moreover, the first two can be performed in parallel. Alternatively, it is possible to compute the terms by two multiplications as follows:

$$B_1 = A_2^2 + 2A_2A_3z = A_2 \times (A_2 + 2A_3z)$$
$$\text{and } A_2^3 \approx (B_1)_{high} \times A_2.$$

The first of the two multiplications is of $k \times 2k$ bits and the second is of $k \times k$.

Then, the terms (either the output of the three multiplications or of the two multiplications) are multiplied by the corresponding factors which depend on the function, as shown in Table 3. Note that, for division and square root, these factors correspond to alignments, whereas, for inverse square root, multiplications by 3 and 5 are required.[2] Finally, the resulting terms are added to produce B.

Fig. 3 shows the weights of these terms in the case of the reciprocal function. The sum of these terms is rounded to the nearest multiple of 2^{-4k}. As shown in Fig. 3, this gives a $(3k+1)$-bit number $\hat{B}$. Then, B is equal to $1+\hat{B}$. An implementation is shown in Fig. 4.

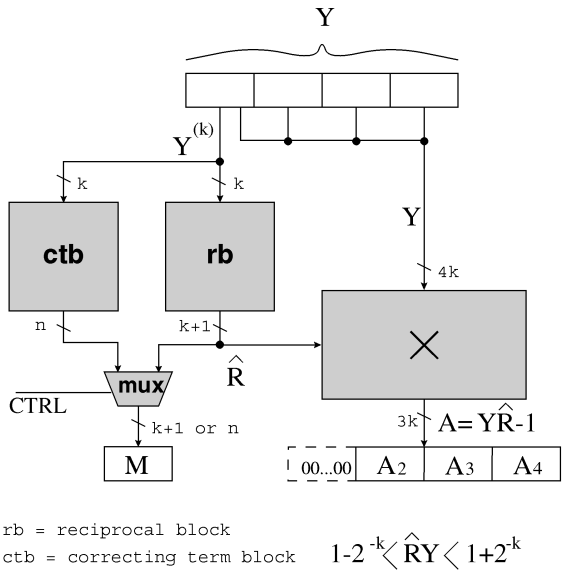


Fig. 2. Organization of the reduction module. M depends on the function being computed.

TABLE 3
Multiplication Factors

Function	B_1	A_2^3
Reciprocal	1	1
Square root	-1/8	1/16
Inverse Square root	3/8	-5/16

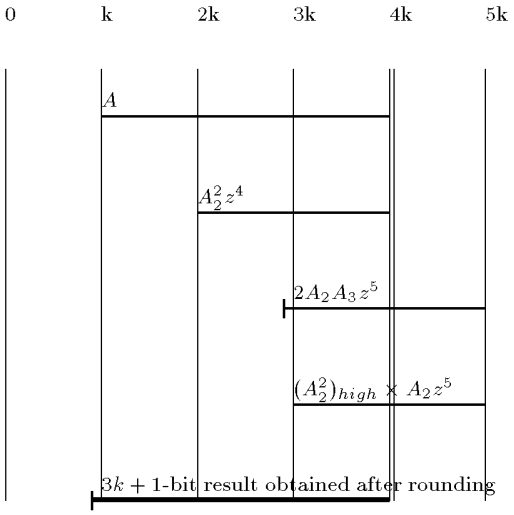


Fig. 3. The terms added during the evaluation step for reciprocal.

3.3 Postprocessing

The postprocessing (Fig. 5) consists in multiplying B by M, where $M = h(\hat{R})$ depends on the function and is computed during the reduction step. Since $B = 1 + \hat{B}$ and $|\hat{B}| < 2^{-k+1}$, to use a smaller multiplier it is better to compute

$$g(Y) = M \times B = M + M \times \hat{B}. \tag{12}$$

2. These "multiplications" will be implemented as (possibly redundant) additions since $3 = 2 + 1$ and $5 = 4 + 1$.

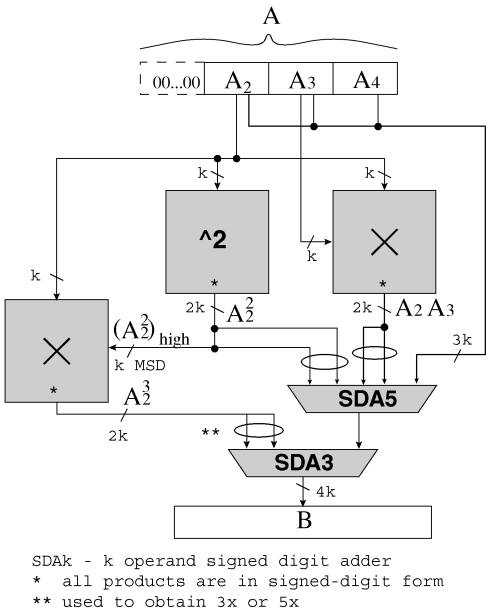


Fig. 4. Functional representation of the evaluation module. (The computations of A_2^2 and A_2A_3 can be regrouped into one rectangular multiplication.)

Note also that, for square root and inverse square root, in the multiplication it suffices to use the bits of M of weight larger than or equal to 2^{-3k-t}, where t is a small integer. Since the error due to this truncation is smaller than or equal to $2^{-4k+1-t}$, choosing $t = 2$ makes the error bounded by 0.5×2^{-4k} and allows the use of a $(3k+1) \times (3k+2)$-bit multiplier. Hence, although we need to store n bits of M (to be added in (12)), only $3k+2$ bits will be used in the multiplication $M \times \hat{B}$.

Table 4 shows the operation that must be performed during the postprocessing step and the value of M that must be used.

4 Estimation of Execution Time and Hardware

We now evaluate the method proposed in terms of execution time and hardware required. This evaluation serves for the comparisons presented in the next section.

4.1 Execution Time

The critical path is given by the following expression:

$$T_{crit} = t_{rb} + tm_{3k\times k} + 2tm_{k\times k} + ta_{4k} + tm_{3k\times 3k},$$

where t_{rb} is the table access time, tm multiplication, and ta addition time.

For instance, for double precision with faithful rounding and implementation of rb directly by table, we obtain the sum of the following delays:

- Access to table of 15 or 14 input bits.
- One multiplication of 46×16 bits (with product in conventional form).
- Two multiplications of 15×15 bits (with product in redundant form).

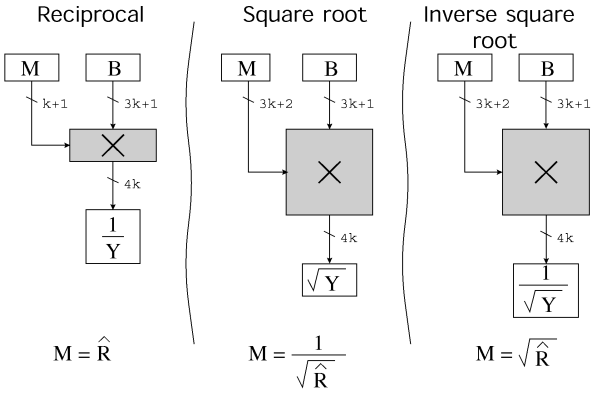


Fig. 5. Functional representation of the postprocessing module.

- Signed-digit addition of the four terms. This addition is the most complex for inverse square root (10): All three multiplications produce signed-digit results and, because the coefficients 3/8, 3/4, and -5/16 are replaced by shifts and adds, this leads to 3×2 signed-digit operands with a total of $1 + 6 = 7$. The multiplications $\frac{3}{8}A_2^2$ and $\frac{3}{4}A_2A_3$ are performed in parallel, followed by a 5-to-1 signed-digit addition SDA5. This addition is performed concurrently with the multiplication $\frac{5}{16}A_2^3$, which produces two operands, so a 3-to-1 signed-digit addition SDA3 completes the critical path in the evaluation module. The result of this addition is used as the multiplier of the next multiplication; consequently, it is possible to directly recode the signed-digit form to radix-4 multiplier.
- Multiplication of 45×43 bits (with product in conventional form).

4.2 Hardware

Table 5 gives the table sizes and number of various operations required by our method, depending on the function being computed, and the value of k. Table 6 gives the required table sizes depending on the function computed and the format (single precision and double precision).

5 Comparison with Other Methods

We restrict our comparison to three methods which also deal with reciprocals, square roots, and inverse square roots. We briefly review these methods and then compare

TABLE 4
Operation Performed during the Postprocessing Step

Function	Value of M	Size of M	Operation
$1/Y$	$\hat{R}$	$k+1$ bits	$M \times B$
$\sqrt{Y}$	$1/\sqrt{\hat{R}}$	n bits	$M \times B$
$1/\sqrt{Y}$	$\sqrt{\hat{R}}$	n bits	$M \times B$

TABLE 5
Table Sizes and Number of Operations for Our Method

Reciprocal	
Table size [bits]	$(k+1)\times 2^k$
Small/large multipliers	5 (2*) / 0
Square and inverse square root	
Table size [bits]	$(k+1+n)\times 2^k$
Small/large multipliers	4 (2*) / 1

"Small": $k \times n$ or $k \times k$ multiplication;
"large": $(3k+1)\times(3k+2)$ multiplication; * - in parallel.

TABLE 6
Tables Required by Our Method (in Bytes)

Format	Reciprocal	Square root	Inverse square root	All 3 functions
SP	128	576	576	1024
DP	65K	142K	142K	289K

SP - single precision ($m = 24$)with faithful rounding;
DP - double precision ($m = 53$)with faithful rounding.

estimates of latency (delay) and of cost (mainly of multipliers and tables) for 53-bit precision.

5.1 Newton-Raphson Iteration

The well-known Newton-Raphson (NR) iteration for reciprocal

$$x_{i+1} = x_i + x_i(1 - Yx_i) \tag{13}$$

converges quadratically to $1/Y$ provided that x_0 is close enough to $1/Y$. We use a k-bit address table to obtain x_0 and perform the intermediate calculations using an n-bit arithmetic. To compare with our method, we assume $n \approx 4k$. The first approximation x_0 is the number $\hat{Y}$ of Section 2, a k-bit approximation of $1/Y$. To get x_1, two $k \times n$-bit multiplications are required. Since x_1 is a $2k$-bit approximation of $1/Y$, it suffices to use its $2k$ most-significant bits to perform the next iteration. After this, one needs to perform two $2k \times n$-bit multiplications to get x_2, which is an n-bit approximation of $1/Y$. For $k = 15$, the NR method requires:

- one lookup in a 15-bit address table;
- two 15×30-bit multiplications (Y truncated to 30 bits);
- two 30×60-bit multiplications.

The multiplications that occur cannot be performed in parallel.

The NR iteration for reciprocal square-root

$$x_{i+1} = \frac{1}{2}x_i\left(3 - Yx_i^2\right) \tag{14}$$

has convergence properties similar to those of the NR iteration for reciprocal. Assuming (as previously) that we use a k-bit address table and that we perform the intermediate calculations using an n-bit arithmetic, with $k = 14$ and $n = 56$ (which are the values required for faithfully rounded double precision square root or inverse square root), computing an inverse square-root using the NR iteration requires:

- one lookup in a 14-bit address table;
- three 14×56-bit multiplications;
- three 28×56-bit multiplications.

In the implementation, we assume using a shared 30×60 multiplier.

Computing a square-root requires the same number of operations plus a final "large" (56×56-bit) multiplication.

5.2 Wong and Goto's Method

The method presented by Wong and Goto in [14] requires tables with $m/2$ address bits, where m is the number of bits of the significand of the floating-point arithmetic being used. This makes that method inconvenient for double-precision calculations. In [13], they suggest another method using table-lookups and rectangular multipliers.

The method for computing reciprocals is as follows: Let us start from the input value $Y = 1.y_1y_2\ldots y_{53}$. The first 10 bits of Y are used as address bits to get from a table

$$r_0 = \left\lfloor \frac{1}{1.y_1y_2\ldots y_{10}} \right\rfloor.$$

Then, compute $r_0 \times Y$ using a rectangular multiplier. The result is a number A of the form:

$$A = 1 - 0.000\ldots0a_9a_{10}\ldots a_{18}\ldots a_{56}.$$

Then, using a rectangular multiplier, compute:

$$\begin{aligned} B &= A \times (1 + 0.000\ldots0a_9a_{10}\ldots a_{18}) \\ &= 1 - 0.000000\ldots00b_{17}b_{18}\ldots b_{26}\ldots b_{56}. \end{aligned}$$

Again, using a rectangular multiplier, compute:

$$\begin{aligned} C &= B \times (1 + 0.000000\ldots00b_{17}b_{18}\ldots b_{26}) \\ &= 1 - 0.00000000000\ldots0000c_{25}c_{26}\ldots c_{56}. \end{aligned}$$

In parallel, use the bits $b_{27}b_{28}\ldots b_{35}$ as address to get from a table β consisting of the nine most significant bits of $(0.0000\ldots b_{27}b_{28}\ldots b_{35})^2$. The final result is:

$$\begin{aligned} \frac{1}{Y} \approx\ & r_0 \times 1.00000\ldots a_9a_{10}\ldots a_{18} \\ & \times 1.000000\ldots00b_{17}b_{18}\ldots b_{26} \\ & \times (1.000000000\ldots000c_{25}c_{26}\ldots c_{56} + \beta). \end{aligned} \tag{15}$$

Fig. 6 illustrates the computational graph for 56-bit reciprocal computation.

Therefore, this method for reciprocation requires one table look-up in a 10-bit address table, one look-up in a 9-bit address table, five rectangular 10×56 multiplications, and one 56×56 multiplication. The critical path is roughly

$$t_{WG} = t_{LUT10} + 3 \times t_{MULT(10\times56)} + t_{MULT(56\times56)}. \tag{16}$$

To compute reciprocal square-roots, the Wong-Goto method uses one look-up in an 11-bit address table, one look-up in a 9-bit address table, nine rectangular multiplications, and one full multiplication. The critical path

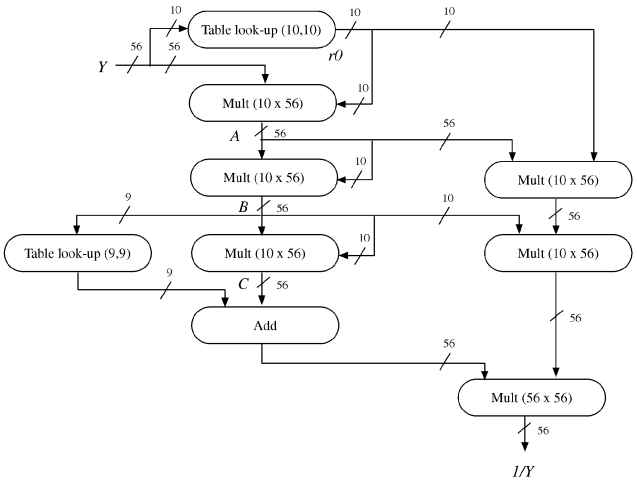


Fig. 6. The Wong-Goto reciprocal computation.

consists of one 11-bit table lookup, five rectangular multiplications, and one full multiplication.

5.3 Method Proposed by Ito, Takagi, and Yajima

This proposal [5] considers the operations division (actually reciprocal) and square root performed using a multiply-accumulate unit (as in Newton-Raphson's method). It discusses a linear initial approximation and proposes accelerated convergence methods. Since the linear initial approximation can also be used for the method proposed in this paper, for comparison purposes we do not include it and use the direct table lookup approach.

For reciprocal, the proposal in [5] results in a speedup with respect to traditional NR by modifying the recurrence so that additional accuracy is achieved by each multiply-accumulate and by adding a cubic term, which requires an additional table look-up. For square root, a novel direct algorithm is proposed, instead of going through inverse square root, as done in NR.

For the comparison of reciprocal, we use a look-up table of 15 input bits for the initial approximation (direct table method). In this case, three multiply-accumulate operations are required and the cubic term uses a table of nine input bits.

For square root, also with a table of 15 input bits for the initial approximation, four multiply-accumulate operations are required.

5.4 Estimates of Delay and Cost

We now estimate the delays and costs (size of tables and size of multipliers) of the schemes. Following [14], the delays are expressed in terms of τ—the delay of a complex gate, such as one full adder. In this unit, we estimate the delays of multipliers and tables; these delays can vary somewhat depending on the technology and implementation, but, since all schemes use the same modules, the relative values should not vary significantly.

The delay on the critical path of a multiplier is the sum of 1, 2, and 3:

1. Radix-4 multiplier recoding, multiple generation and buffering: 2τ.
2. Partial product reduction array: $1\tau \times$ number of SDA (Signed-Digit Adder) stages;
3. Final CPA (Carry Propagate Adder)—when needed: 4τ for > 30 bits; 3τ for ≤ 30 bits.

TABLE 7
Delays of Multipliers: $t_{recode} + t_{reduce} + t_{CPA}$ (in τs)

Size	Delay $[\tau]$
10×56	2+3+4=9
15×15	2+4+3=9
15×30	2+4+4=10
16×56	2+4+4=10
$30(28) \times 60(56)$	2+5+4=11
45×45	2+6+4=12
56×56	2+6+4=12

The delays of the various multipliers used are summarized in Table 7.

For the delay of a look-up table with 14-15 address bits, we estimate around 8τ and with 10-12 bits around 5τ. The size is given directly in Table 8.

From the description of the methods given above, we summarize their characteritics in Table 8 and obtain the estimates of Table 9. We conclude that, for reciprocal, our method has a similar delay as the other schemes, but is significantly faster for square root and for inverse square root. On the other hand, the Wong-Goto method requires smaller tables.

TABLE 8
Modules Required for Double Precision Computation of Reciprocal, Square Root, and Inverse Square Root in Related Methods

NR	
Table delay $[\tau]$: 8	Multipliers:
Table 1: $2^{15} \times 15$	15×30
Table 2: $2^{14} \times 14$	14×56
	30×60
(for square root)	56×56
Wong and Goto	
Table delay $[\tau]$: 5	Multipliers:
Table 1: $2^{10} \times 10$	10×56 (2)
Table 2: $2^{9} \times 9$	56×56
Table 3: $2^{11} \times 22$	
Table 4: $2^{11} \times 11$	
Table 5: $2^{9} \times 9$	
Ito, Takagi and Yajima	
Table delay $[\tau]$: 8	Multipliers:
Table 1: $2^{15} \times 15$	56×56
Table 2: $2^{9} \times 9$	
Table 3: $2^{10} \times 119$	
Table 4: $2^{8} \times 8$	
Table 5: $2^{14} \times 14$	

TABLE 9
Estimation of Total Delays (in τs) for Double Precision Computation of Reciprocals, Square Roots, and Inverse Square Roots

Method	Reciprocal	Square root	Inverse square root
NR	58	87	75
Wong and Goto	48	78	66
Ito, Takagi and Yajima	47	60	75*
Our	48	48	48

$*$ assumed to be implemented as in NR method; τ - the delay of complex gate.

NR
Recip. : 8 (table) + 2x10 (15x30 mults) + 4 (add) + 2x11 (30x60 mults) + 4 (add) = 58
Inv. sqrt: 8 (table) + 3x10 (14x56 mults) + 2 (sub) + 3x11 (28x56 mults) + 2 (sub) = 75
Sqrt: Inv. SQRT + 12 (56x56 mult) = 87
Wong and Goto
Recip. : 5 (table) + 3x9 (10x56 mults) + 4 (add) + 12 (56x56 mult) = 48
Inv. sqrt: 5 (table) + 5x9 (10x56 mults) + 4 (add) + 12 (56x56 mult) = 66
Sqrt: Inv. SQRT + 12 (56x56 mult) = 78
Ito, Takagi and Yajima
Recip: 8 (table) + 3x12 (56x56 mult) + 3 (acc) = 47
Sqrt: 8 (table) + 4x12 (56x56 mult) + 4 (acc) = 60
Our
Recip: 8 (table) + 10 (16x56 mult) [a] + 2x6 (15x15 mult with SD product) + 4 (SDA) [b] + 4 (CPA) + 10 (16x46 mult) [c] = 48
Sqrt/ Inv. Sqrt: 18 (reduction) + 16 (evaluation) + 2 (recode) + 12 (44 x 43 mult) = 48
[a] reduction; [b] evaluation; [c] post-processing.

6 Elementary Functions

Using the same basic scheme, our method also allows computation of some of the elementary functions. We briefly describe this below. Implementation is not discussed: It is very similar to what we have previously described for reciprocal, square root, and inverse square root.

6.1 Computation of Logarithms

In a similar fashion, we get:

$$\ln(1+A) \approx A - \frac{1}{2}A_2^2 z^4 - A_2 A_3 z^5 + \frac{1}{3}A_2^3 z^6.$$

Again, we only need to compute A_2^2, A_2A_3, and A_2^3. The multiplication by $\frac{1}{3}$ can be done with a small multiplier. The postprocessing step is performed as $g(Y) = M + B$, where $M = -ln(\hat{R})$ and $B = ln(1+A)$. When the argument is close to 1, no reduction is performed and, consequently, there is no cancellation.

6.2 Computation of Exponentials

Now, let us assume that we want to evaluate the exponential of an n-bit number $Y = 1 + A_1 z + A_2 z^2 + A_3 z^3 + A_4 z^4$, where $z = 2^{-k}$ ($k = n/4$), and the A_is are k-bit integers. We suggest first computing the exponential of

$$A = A_2 z^2 + A_3 z^3 + A_4 z^4,$$

TABLE 10
Summary of the Proposed Method

Reciprocal	
Table size [bits]	$(k+1)\times 2^k$
Small/large multipliers	5 (2*) / 0
Square/inverse square root	
Table size [bits]	$(k+1+n)\times 2^k$
Small/large multipliers	4 (2*) / 1
Logarithm	
Table size [bits]	$(k+1+n)\times 2^k$
Small/large multipliers	4 (2*) / 0
Exponential	
Table size [bits]	$n\times 2^k$
Small/large multipliers	3 (2*) / 0
Combined (all five)	
Table size [bits]	$(k+1+4n)\times 2^k$
Small/large multipliers	5 (2*) / 1

"Small": $k \times n$ or $k \times k$ multiplication; "large": $(3k+1)\times(3k+2)$ multiplication; * - in parallel $k = 7$ for single precision ($m = 24$); $k = 15(14)$ for double precision ($m = 53$); $n = 4k$.

using a Taylor expansion, and then to multiply it by the number

$$M = \exp(1 + A_1 z).$$

M will be obtained by looking up in a k-bit address table.

The exponential of A can be approximated by:

$$1 + A + \frac{1}{2}A_2^2 z^4 + A_2 A_3 z^5 + \frac{1}{6}A_2^3 z^6. \tag{17}$$

This shows that the same architecture suggested in Section 3 can be used as well for computing exponentials, with similar delay and accuracy.

7 Conclusion

We have proposed a method for computation of reciprocals, square-roots, inverse square-roots, logarithms, and exponentials. Table 10 summarizes the key implementation requirements in evaluating these functions. The strength of our method is that the same basic computations are performed for all these various functions. As shown in the section on comparisons, in double precision for reciprocal our method requires a computational delay quite close to other related methods, but it is significantly faster for square root and for inverse square root. We have considered only faithful rounding.

Appendix

To prove the theorem, let us start from the series (7):

$$\begin{aligned} f(A) = C_0 &+ C_1\left(A_2 z^2 + A_3 z^3 + A_4 z^4\right) \\ &+ C_2\left(A_2 z^2 + A_3 z^3 + A_4 z^4\right)^2 \\ &+ C_3\left(A_2 z^2 + A_3 z^3 + A_4 z^4\right)^3 \\ &+ C_4\left(A_2 z^2 + A_3 z^3 + A_4 z^4\right)^4 + \ldots . \end{aligned} \tag{18}$$

Let us keep in mind that $A = A_2z^2 + A_3z^3 + A_4z^4$ is obviously less than 2^{-k}. If we drop out from the previous series the terms with coefficients $C_4, C_5, C_6, C_7, \ldots$, the error will be:

$$\left|\sum_{i=4}^{\infty} C_i\left(A_2z^2 + A_3z^3 + A_4z^4\right)^i\right|,$$

which is bounded by

$$\epsilon_1 = C_{max}\sum_{i=4}^{\infty}\left(2^{-k}\right)^i = C_{max}\frac{2^{-4k}}{1-2^{-k}}, \tag{19}$$

where $C_{max} = \max_{i\geq 4}|C_i|$.

Now, let us expand the expression obtained from (7) after having discarded the terms of rank ≥ 4. We get:

$$\begin{aligned} f(A) \approx\ & C_0 + C_1A + C_2A_2^2z^4 + 2C_2A_2A_3z^5 \\ & + \left(2C_2A_2A_4 + C_2A_3^2 + C_3A_2^3\right)z^6 \\ & + \left(2C_2A_3A_4 + 3C_3A_2^2A_3\right)z^7 \\ & + \left(C_2A_4^2 + 3C_3A_2^2A_4 + 3C_3A_2A_3^2\right)z^8 \\ & + \left(6C_3A_2A_3A_4 + C_3A_3^3\right)z^9 \\ & + \left(3C_3A_2A_4^2 + 3C_3A_3^2A_4\right)z^{10} \\ & + 3C_3A_3A_4^2z^{11} + C_3A_4^3z^{12}. \end{aligned} \tag{20}$$

In this rather complicated expression, let us discard all the terms of the form $W \times z^j$ such that the maximum possible value of W multiplied by $z^j = 2^{-kj}$ is less than or equal to z^4. We then get (8), that is:

$$f(A) \approx C_0 + C_1A + C_2A_2^2z^4 + 2C_2A_2A_3z^5 + C_3A_2^3z^6.$$

To get a bound on the error ϵ obtained when approximating (20) by (8), we replace the A_is by their maximum value $2^k - 1$ and we replace the C_is by their absolute value. This gives:

$$\begin{aligned} \epsilon_2 &\leq (3|C_2| + 3|C_3|)2^{-4k} + (2|C_2| + 6|C_3|)2^{-5k} \\ &\quad + (|C_2| + 7|C_3|)2^{-6k} \\ &\quad + 6|C_3|2^{-7k} + 3|C_3|2^{-8k} + |C_3|2^{-9k} \\ &\leq \left(3|C_2| + 3|C_3| + 8.5\max\{|C_2|, |C_3|\} \times 2^{-k}\right)2^{-4k}, \end{aligned}$$

assuming that $8 \times 2^{-k} + 6 \times 2^{-2k} + 3 \times 2^{-3k} + 2^{-4k} < 0.5$, which is true for $k \geq 5$.

As explained in Section 2, when computing (8), we will make another approximation: after having computed A_2^2, the computation of A_2^3 would require a $2k \times k$ multiplication. Instead of this, we will take the most k significant bits of A_2^2 only and multiply them by A_2. If we write:

$$A_2^2 = \left(A_2^2\right)_{low} + 2^k\left(A_2^2\right)_{high},$$

where $\left(A_2^2\right)_{low}$ and $\left(A_2^2\right)_{high}$ are k-bit numbers, the error committed is

$$C_3\left(A_2^2\right)_{low}A_2z^6,$$

whose absolute value is bounded by $\epsilon_3 = |C_3|2^{-4k}$.

By adding the three errors due to the discarded terms, we get the bound given in the theorem.

ACKNOWLEDGMENTS

This research has been partially funded by a joint PICS grant from the French CNRS and MAE, the US National Science Foundation Grant "Effect of Redundancy in Arithmetic Operations on Processor Cycle Time, Architecture, and Implementation." We thank the reviewers for their careful and helpful comments.

REFERENCES

[1] M.D. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations.* Boston: Kluwer Academic, 1994.

[2] P.M. Farmwald, "High Bandwidth Evaluation of Elementary Functions," *Proc. Fifth IEEE Symp. Computer Arithmetic,* K.S. Trivedi and D.E. Atkins, eds., 1981.

[3] H. Hassler and N. Takagi, "Function Evaluation by Table Look-Up and Addition," *Proc. 12th IEEE Symp. Computer Arithmetic,* S. Knowles and W. McAllister, eds., pp. 10-16, July 1995.

[4] C. Iordache and D.W. Matula, "On Infinitely Precise Roundings for Division, Square-Root, Reciprocal and Square-Root Reciprocal," *Proc. 14th IEEE Symp. Computer Arithmetic,* pp. 233-240, 1999.

[5] M. Ito, N. Takagi, and S. Yajima, "Efficient Initial Approximation and Fast Converging Methods for Division and Square Root," *Proc. 12th IEEE Symp. Computer Arithmetic,* S. Knowles and W. McAllister, eds., pp. 2-9, July 1995.

[6] V.K. Jain and L. Lin, "High-Speed Double Precision Computation of Nonlinear Functions," *Proc. 12th IEEE Symp. Computer Arithmetic,* S. Knowles and W. McAllister, eds., pp. 107-114, July 1995.

[7] V. Lefèvre, J.M. Muller, and A. Tisserand, "Towards Correctly Rounded Transcendentals," *IEEE Trans. Computers,* vol. 47, no. 11, pp. 1,235-1,243, Nov. 1998.

[8] J.M. Muller, *Elementary Functions, Algorithms and Implementation.* Boston: Birkhauser, 1997.

[9] S.F. Oberman and M.J. Flynn, "Division Algorithms and Implementations," *IEEE Trans. Computers,* vol. 46, no. 8, pp. 833-854, Aug. 1997.

[10] D.D. Sarma and D.W. Matula, "Faithful Bipartite ROM Reciprocal Tables," *Proc. 12th IEEE Symp. Computer Arithmetic,* S. Knowles and W. McAllister, eds., pp. 17-28, July 1995.

[11] M. Schulte and J. Stine, "Symmetric Bipartite Tables for Accurate Function Approximation," *Proc. 13th IEEE Symp. Computer Arithmetic,* T. Lang, J.M. Muller, and N. Takagi, eds., 1997.

[12] P.T.P. Tang, "Table Lookup Algorithms for Elementary Functions and Their Error Analysis," *Proc. 10th IEEE Symp. Computer Arithmetic,* P. Kornerup and D.W. Matula, eds., pp. 232-236, June 1991.

[13] W.F. Wong and E. Goto, "Fast Hardware-Based Algorithms for Elementary Function Computations Using Rectangular Multipliers," *IEEE Trans. Computers,* vol. 43, pp. 278-294, Mar. 1994.

[14] W.F. Wong and E. Goto, "Fast Evaluation of the Elementary Functions in Single Precision," *IEEE Trans. Computers,* vol. 44, no. 3, pp. 453-457, Mar. 1995.

[15] T. Lang and P. Montuschi, "Very High Radix Square Root with Prescaling and Rounding and a Combined Division/Square Root Unit," *IEEE Trans. Computers,* vol. 48, no. 8, pp. 827-841, Aug. 1999.

[16] E. Antelo, T. Lang, and J.D. Bruguera, "Computation of $\sqrt{x/d}$ in a Very High Radix Combined Division/Square-Root Unit with Scaling," *IEEE Trans. Computers,* vol. 47, no. 2, pp. 152-161, Feb. 1998.



Milos D. Ercegovac earned his BS in electrical engineering (1965) from the University of Belgrade, Yugoslavia, and his MS (1972) and PhD (1975) in computer science from the University of Illinois, Urbana-Champaign. He is a professor in the Computer Science Department, School of Engineering and Applied Science at the University of California, Los Angeles. Dr. Ercegovac specializes in research and teaching in digital arithmetic, digital design, and computer system architecture. His recent research is in the areas of arithmetic design for field programmable gate arrays (FPGAs). His research contributions have been extensively published in journals and conference proceedings. He is a coauthor of two textbooks on digital design and of a monograph in the area of digital arithmetic. Dr. Ercegovac has been involved in organizing the IEEE Symposia on Computer Arithmetic. He served as an editor of the *IEEE Transactions on Computers* and as a subject area editor for the *Journal of Parallel and Distributed Computing*. He is a member of the ACM and the IEEE Computer Society.



Tomás Lang received an electrical engineering degree from the Universidad de Chile in 1965, an MS from the University of California (Berkeley) in 1966, and the PhD from Stanford University in 1974. He is a professor in the Department of Electrical and Computer Engineering at the University of California, Irvine. Previously, he was a professor in the Computer Architecture Department of the Polytechnic University of Catalonia, Spain, and a faculty member of the Computer Science Department at the University of California, Los Angeles.

Dr. Lang's primary research and teaching interests are in digital design and computer architecture with current emphasis on high-speed and low-power numerical processors and multiprocessors. He is coauthor of two textbooks on digital systems, two research monographs, one IEEE Tutorial, and author or coauthor of research contributions to scholarly publications and technical conferences. He is a member of the IEEE Computer Society.



Jean-Michel Muller received the Engineer degree in applied mathematics and computer science in 1983 and the PhD in computer science in 1985, both from the Institut National Polytechnique de Grenoble, France. In 1986, he joined the CNRS (French National Center for Scientific Research). He is with LIP Laboratory, Ecole Normale Supérieure de Lyon, where he manages the CNRS/ENSL/INRIA Arenaire project. His research interests are in computer arithmetic. Dr. Muller served as co-program chair of the 13th IEEE Symposium on Computer Arithmetic and general chair of the 14th IEEE Symposium on Computer Arithmetic. He has been an associate editor of the *IEEE Transactions on Computers* since 1996. He is a member of the IEEE Computer Society.



Arnaud Tisserand received the MSc degree and the PhD degree in computer science from the Ecole Normale Supérieure de Lyon, France, in 1994 and 1997, respectively. In 1999, he joined INRIA (National Institute for Computer Science and Control, France), posted to the CNRS/ENSL/INRIA Arenaire project, at the Ecole Normale Supérieure de Lyon. His research interests include computer arithmetic, computer architecture, and VLSI design. He is a member of the IEEE.

A.3. Méthode des tables multipartites

Reproduction de l'article [24] paru dans le journal *IEEE Transactions on Computers* en mars 2005 et résumé en section 2.3.2.

auteurs	F. de Dinechin et A. Tisserand
titre	*Multipartite Table Methods*
journal	*IEEE Transactions on Computers*
date	mars 2005
volume	54
numéro	3
pages	319–330
numéro DOI	10.1109/TC.2005.54

Multipartite Table Methods

Florent de Dinechin, *Member*, *IEEE*, and Arnaud Tisserand, *Member*, *IEEE*

Abstract—A unified view of most previous table-lookup-and-addition methods (bipartite tables, SBTM, STAM, and multipartite methods) is presented. This unified view allows a more accurate computation of the error entailed by these methods, which enables a wider design space exploration, leading to tables smaller than the best previously published ones by up to 50 percent. The synthesis of these multipartite architectures on Virtex FPGAs is also discussed. Compared to other methods involving multipliers, the multipartite approach offers the best speed/area tradeoff for precisions up to 16 bits. A reference implementation is available at www.ens-lyon.fr/LIP/Arenaire/.

Index Terms—Computer arithmetic, elementary function evaluation, hardware operator, table lookup and addition method.

1 Introduction

Table-lookup-and-addition methods, such as the bipartite method, have been the subject of much recent attention [1], [2], [3], [4], [5]. They allow us to compute commonly used functions with low accuracy (up to 20 bits) with significantly lower hardware cost than that of a straightforward table implementation, while being faster than shift-and-add algorithms *à la* CORDIC or polynomial approximations. They are particularly useful in digital signal or image processing. They may also provide initial seed values to iterative methods, such as the Newton-Raphson algorithms for division and square root [6], [7], which are commonly used in the floating-point units of current processors. They also have recently been successfully used to implement addition and subtraction in the logarithm number system [8].

The main contribution of this paper is to unify two complementary approaches to multipartite tables by Stine and Schulte [4] and Muller [5]. Completely defining the implementation space for multipartite tables allows us to provide a methodology for selecting the best implementation that fulfills arbitrary accuracy and cost requirements. This methodology has been implemented and is demonstrated on a few examples. This paper also clarifies some of the cost and accuracy questions which are incompletely formulated in previous papers. This paper is an extended version of an article published in the *Proceedings of the 15th IEEE International Symposium on Computer Arithmetic* [9].

After some notations and definitions in Section 2, Section 3 presents previous work on table-lookup-and-addition methods. Section 4 presents our unified multipartite approach in all the details. Section 5 gives results and compares them to previous work. Section 6 concludes.

• *The authors are with the Arénaire Project (CNRS-ENSL-INRIA-UCBL) LIP, École Normale Supérieure de Lyon, 46 allée d'Italie, F-69364 Lyon, France. E-mail: {Florent.de.Dinechin, Arnaud.Tisserand}@ens-lyon.fr.*

Manuscript received 1 Dec. 2003; revised 16 June 2004; accepted 17 Sept. 2004; published online 18 Jan. 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-0242-1203.

0018-9340/05/$20.00 © 2005 IEEE Published by the IEEE Computer Society

2 Generalities

2.1 Notations

Throughout this paper, we discuss the implementation of a function with inputs and outputs in fixed-point format. We shall use the following notations:

- We note $f : [a, b[\rightarrow [c, d[$, the function to be evaluated with its domain and range.
- We note w_I and w_O, the required input and output size.

In general, we will identify any word of p bits to the integer in $\{0, \ldots, 2^p - 1\}$ it codes, writing such a word in capital letters. When needed, we will provide explicit functions to map such an integer into the real domain or range of the function. For instance, an input word X will denote an integer in $\{0, \ldots, 2^{w_I} - 1\}$, and we will express the real number $x \in [a, b[$ that it codes by $x = a + (b - a)X/2^{w_I}$. Note that no integer maps to b, the right bound of the input interval, which explains why we define this interval as open in b. Such a mapping should be part of the specification of a hardware function evaluator and several alternatives exist, depending on the function to be evaluated and the needs of the application. Some applications may require that the integer X denotes $x = a + (b - a)(X + 1/2)/2^{w_I}$, some may require that it denotes $x = a + (b - a)X/(2^{w_I} - 1)$. For other applications to floating-point hardware, the input interval may span over two consecutive binades, in which case, we will consider two input intervals with different mappings. The reader should keep in mind that all the following work can be straightforwardly extended to any such mapping between reals and integers. A general presentation would degrade readability without increasing the interest of the paper. Our implementation, however, can accommodate arbitrary styles of discretization of the input and output intervals.

2.2 Errors in Function Evaluation

Usually, three different kinds of error sum up to the total error of an evaluation of $f(x)$:

- The *input discretization* or *quantization* error measures the fact that an input number usually represents a

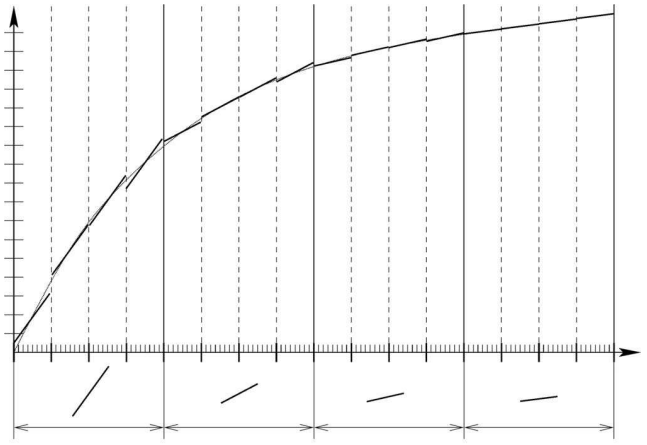


Fig. 1. The bipartite approximation.

small interval of values centered around this number.

- The *approximation* or *method* error measures the difference between the pure mathematical function f and the approximated mathematical function (here, a piecewise affine function) used to evaluate it.
- Finally, the actual computation involves *rounding* errors due to the discrete nature of the final and intermediate values.

In the following, we will ignore the question of input discretization by considering that an input number only represents itself as an exact mathematical number. Again, all the following work could probably be extended to take quantization errors into account.

3 Previous and Related Works

An approximation of a function may be simply stored in a look-up table containing 2^{w_I} values. This approach becomes impractical as soon as w_I exceeds 10-12 bits. In this section, we explore various methods which allow us to approximate functions with much less memory and very little computation.

The present paper improves on the bipartite idea and its successors, which are first presented in detail (Sections 3.1 to 3.3). As our results should be compared to other competitive hardware approximation methods, we then also present these methods (Sections 3.4 to 3.6). We leave out of this survey methods more specifically designed for a particular function, such as the indirect bipartite method for postscaled division [10], many methods developed for evaluating $f(x) = \log_2(1 \pm 2^x)$ for Logarithm Number System (LNS) arithmetic [11], [12], [13], [14], and many others.

3.1 The Bipartite Method

First presented by Das Sarma and Matula [1] in the specific case of the reciprocal function and generalized by Schulte and Stine [3], [4] and Muller [5], this method consists of approximating the function by affine segments, as illustrated in Fig. 1.

The 2^α segments (16 in Fig. 1) are indexed by the α most significant bits of the input word, as depicted in Fig. 2. For each segment, one initial value is tabulated and the other values are interpolated by adding, to this initial value, an offset computed out of the $w_I - \alpha$ least significant bits of the input word.

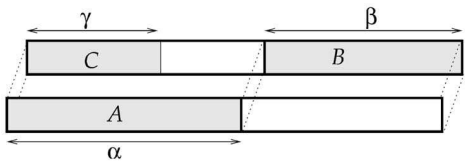


Fig. 2. Bipartite input word decomposition.

The idea behind the bipartite method is to group the 2^α input intervals into 2^γ (with $\gamma < \alpha$) larger intervals (four in Fig. 1) such that the slope of the segments is considered constant on each larger interval. These four constant slopes are figured in Fig. 1 and may be tabulated: Now, there are only 2^γ tables of offsets, each containing 2^β offsets. Altogether, we thus need to store $2^\alpha + 2^{\gamma+\beta}$ values instead of $2^{w_I} = 2^{\alpha+\beta}$.

In all the following, we will call the table that stores the initial points of each segment the *Table of Initial Values (TIV)*. This table will be addressed by a subword A of the input word, made of the α most significant bits. A *Table of Offsets (TO)* will be addressed by the concatenation of two subwords of the input word: C (the γ most significant bits) and B (the β least significant bits). Fig. 2 depicts this decomposition of the input word.

Previous authors [5], [4] have expressed the bipartite idea in terms of a Taylor approximation, which allows a formal error analysis. They find that, for $\gamma \approx \beta \approx \alpha/2$, it is possible to keep the error entailed by this method in "acceptable bounds" (the error obviously depends on the function under consideration). We develop in this paper a more geometrical approach to the error analysis with the purpose of computing the approximation error exactly, where Taylor formulas only give upper bounds.

3.2 Exploiting Symmetry

Schulte and Stine have remarked [3] that it is possible to exploit the symmetry of the segments on each small interval (see Fig. 3, which is a zoom view of Fig. 1) to halve the size of the TO: They store the value of the function in the middle of the small interval in the TIV and the offsets for a half segment in the TO. The offsets for the other half are computed by symmetry. The extra hardware cost (mostly a few XOR gates) is usually more than compensated by the reduction in the TO size (see the SBTM paper, for *Symmetric Bipartite Table Addition Method* [3]).

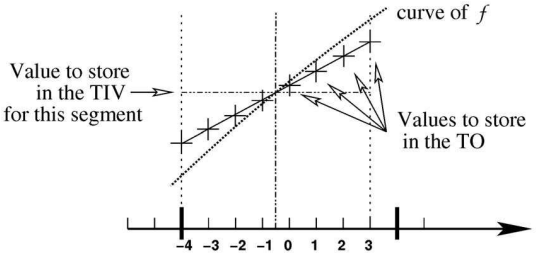


Fig. 3. Segment symmetry.

Note that the initial bipartite paper [1] suggested using an "average curve" approximation instead of a linear one for the TO. This idea wouldn't improve the maximum error, but would bring a small improvement to the average error (a fraction of half an ulp, as Section 4 will show). However, in this case, Fig. 3 is no longer symmetric and the table size reduction discussed here is no longer possible. Therefore, this idea will not be considered further.

3.3 Multipartite Methods

In another paper [4], Stine and Schulte have remarked that the TO can be decomposed into several smaller tables: What the TO computes is a linear function $\mathrm{TO}(CB) = s(C) \times B$, where $s(C)$ is the slope of the segment. The subword B can be decomposed (as seen in Fig. 6) into m subwords, B_i, of sizes β_i for $0 \leq i < m$:

$$B = B_0 + 2^{\beta_0} B_1 + \ldots + 2^{\beta_0+\beta_1+\ldots+\beta_{m-2}} B_{m-1}.$$

Let us define $p_0 = 0$ and $p_i = \sum_{j=0}^{i-1} \beta_j$ for $i > 0$. The function computed by the TO is then:

$$\begin{aligned} \mathrm{TO}(CB) &= s(C) \times \sum_{i=0}^{m-1} 2^{p_i} B_i \\ &= \sum_{i=0}^{m-1} 2^{p_i} s(C) \times B_i \qquad (1) \\ &= \sum_{i=0}^{m-1} 2^{p_i} \mathrm{TO}_i(CB_i). \end{aligned}$$

Thus, the TO can be distributed into m smaller tables, $\mathrm{TO}_i(CB_i)$, resulting in much smaller area (symmetry still applies for the m TO_is). This comes at the cost of $m-1$ additions. This improvement thus entails two tradeoffs:

- A cost tradeoff between the cost of the additions and the table size reduction.
- An accuracy tradeoff: Equation (1) is not an approximation, but it will lead to more discretization errors (one per table), which will sum up to a larger global discretization error unless the smaller tables have a bigger output accuracy (and, thus, are bigger). We will formalize this later.

Schulte and Stine have termed this method STAM, for *Symmetric Table and Addition Method*. It can still be improved: Note, in (1) that, for $j > i$, the weight of the LSB of TO_j is $2^{p_j - p_i}$ times the weight of the LSB of TO_i. In other terms, TO_i is more accurate than TO_j. It will be possible, therefore, to build even smaller tables than Schulte and Stine by compensating for the (wasted) higher accuracy of TO_i by a rougher approximation on $s(C)$, obtained by removing some least significant bits from the input C.

A paper from Muller [5], contemporary to that of Stine and Schulte, indeed exploits this idea in a specific case. The *multipartite* method presented there is based on a decomposition of the input word into $2p+1$ subwords $X_1, \ldots, X_{2p+1}$ of identical sizes. An error analysis based on a Taylor formula shows that equivalent accuracies are obtained by a table addressed by X_{2p+1} and a slope determined only by X_1, a table addressed by X_{2p} and a slope determined by X_1X_2, and, in general, a table addressed by X_{2p+2-i} and the i most significant subwords.

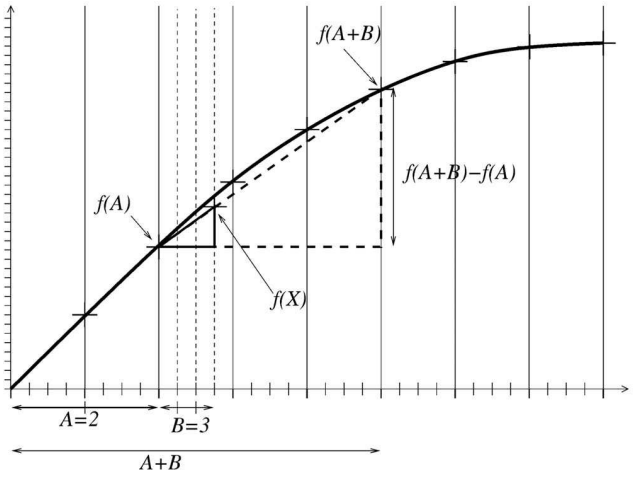


Fig. 4. First order ATA.

Muller claims (although without any numerical support) that the error/cost tradeoffs of this approach are comparable to Schulte and Stine's method. His decomposition, however, is too rigid to be really practical, while his error analysis is based on potentially overestimated error bounds due to the Taylor approximation. Besides, he doesn't address the rounding issue.

3.4 ATA Methods

The *Addition-Table-Addition* methods allow additions before and after the table look-ups. They are termed after Wong and Goto [15]; however, a whole range of such methods is possible and, to our knowledge, unpublished. This section is a survey of these methods.

Let us note $X = A + 2^{-\beta}B = a_{\alpha-1} \ldots a_0 b_{\beta-1} \ldots b_0$, where $\alpha > \beta$.

To compute $f(A + 2^{-\beta}B)$, it is possible to use the first-order Taylor approximation:

$$f(A + 2^{-\beta}B) \approx f(A) + 2^{-\beta} B f'(A)$$

with

$$Bf'(A) \approx f(A+B) - f(A).$$

Finally,

$$f(A + 2^{-\beta}B) \approx f(A) + 2^{-\beta}(f(A+B) - f(A)).$$

In other terms, this first-order ATA method approximates, in a neighborhood of A, the graph of f with a homotetic reduction of this graph with a ratio of 2^{β}, as pictured in Fig. 4.

Evaluating $f(x)$ thus involves

- one α-bit addition to compute $a_{\alpha-1} \ldots a_0 + b_{\beta-1} \ldots b_0$,
- two lookups in the same table,
  - $f(a_{\alpha-1} \ldots a_0)$ and
  - $f(a_{\alpha-1} \ldots a_0 + b_{\beta-1} \ldots b_0)$,
- one subtraction to compute the difference between the previous two lookups on less than α bits (the size

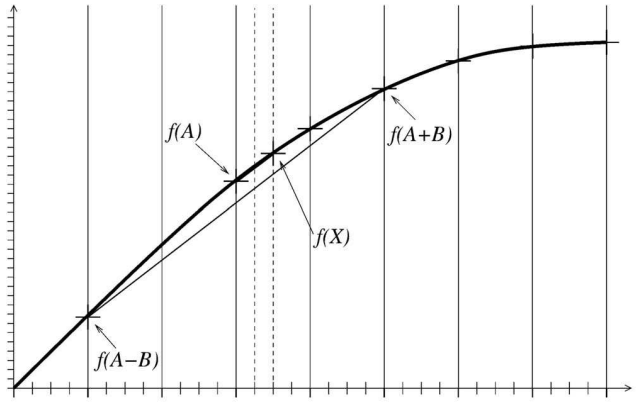


Fig. 5. Second order ATA.

of this subtraction depends on f and β and may be computed exactly),
- one shift by β bits to perform the division by 2^β, and
- one final addition on w_O bits.

Both table lookups can be performed in parallel in a dual-port table or in parallel using two tables or in sequence/pipeline (one read before the addition and one read after). This leads to a range of architectural tradeoffs.

This method can be extended to the second order by using a *central difference formula* to compute a much better approximation of the derivative (the error is a third-order term) as depicted in Fig. 5.

The formula used is now

$$Bf'(A) \approx \frac{f(A+B) - f(A-B)}{2}$$

and the algorithm consists of the following steps:

- Compute (in parallel) $A+B$ and $A-B$;
- Read in a table $f(A)$, $f(A+B)$, and $f(A-B)$;
- Compute

$$f(A + 2^{-\beta}B) \approx f(A) + 2^{-\beta-1}(f(A+B) - f(A-B)).$$

We now need three lookups in the same table and seven additions. Here again, a range of space/time tradeoffs is possible.

The original method by Wong and Goto [15] is actually more sophisticated: As in the STAM method, they split B into two subwords of equal sizes, $B = B_1 + 2^{\beta/2}B_2$, and distribute $Bf'(A)$ using two centered differences, which reduces table sizes. Besides they add a table which contains second and third-order corrections, indexed by the most-significant half-word of A and the most-significant half-word of B. For 24 bits of precision, their architecture therefore consists of six tables with 12 or 13 bits of inputs and a total of nine additions.

Another option would be to remark that, with these three table lookups, it is also possible to use a second-order Taylor formula:

$$f(A + 2^{-\beta}B) \approx f(A) \; + \; 2^{-\beta}Bf'(A) \; + \; \frac{(2^{-\beta}B)^2}{2} f''(A).$$

Indeed, we may approximate the term $f''(A)$ by

$$\begin{aligned} f''(A) &\approx \frac{f'(A+B/2) - f'(A-B/2)}{B} \\ &\approx \frac{\frac{f(A+B)-f(A)}{B} - \frac{f(A)-f(A-B)}{B}}{B} \\ &\approx \frac{f(A+B) - 2f(A) + f(A-B)}{B^2}. \end{aligned}$$

And, finally,

$$\begin{aligned} f(A + 2^{-\beta}B) \approx\; & f(A) \\ & + 2^{-\beta-1}(f(A+B) - f(A-B)) \\ & + 2^{-2\beta-1}(f(A+B) - 2f(A) + f(A-B)). \end{aligned}$$

However, the larger number of look-ups and arithmetic operations entails more rounding errors, which actually consume the extra accuracy obtained thanks to this formula.

Finally, the ATA methods can be improved using symmetry, just like multipartite methods.

These methods have been studied by the authors and found to perform better than the original bipartite approaches, but worse than the generalized multipartite approach which is the subject of this paper. This is also true of the initial ATA architecture by Wong and Goto [15], as will be exposed in Section 5.1.

3.5 Partial Product Arrays

This method is due to Hassler and Takagi [2]. The idea is to approximate the function with a polynomial of arbitrary degree (they use a Taylor approximation). Writing X and all the constant coefficients as sums of weighted bits (as in $X = \sum x_i 2^{-i}$), they distribute all the multiplications within the polynomial, thus rewriting the polynomial as the sum of a huge set of weighted products of some of the x_i. A second approximation then consists of neglecting as many of these terms as possible in order to be able to partition the remaining ones into several tables.

This idea is very powerful because the implementation space is very wide. However, for the same reason, it needs to rely on heuristics to explore this space. The heuristic chosen by Hassler and Takagi in [2] leads to architectures which are less compact than their multipartite counterpart [4] (and are interestingly similar). The reason is probably that the multipartite method exploits the higher-level property of continuity of the function, which is lost in the set of partial products.

3.6 Methods Involving Multipliers

The previous two methods involve higher order approximation of the function, but the architecture involves only adders and tables. If this constraint is relaxed, a huge range of approximations becomes possible. The general scheme is to approximate the function with one or several polynomials and trade table size for multiplications. Papers relevant to this work include (but this list is far from exhaustive):

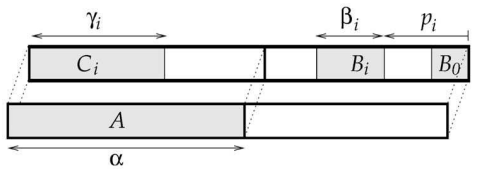


Fig. 6. Multipartite input word decomposition.

- an architecture by Defour et al. [16] involving only two small multipliers (small meaning that their area and delay are comparable to a few adders as one of the inputs is only $w_I/5$ bits wide);
- an architecture by Piñeiro et al. [17] using a squarer unit and a multiplication tree;
- several implementations of addition and subtraction in the logarithm number system with (among others) approximation of order zero [11], order one [12], and order two [13], [14]. As already mentioned, the function to be evaluated is $f(x) = \log_2(1 \pm 2^x)$ and lends itself to specific tricks, like replacing multiplications with additions in the log domain.

These methods will be quantitatively compared to the multipartite approach in Section 5.5.

3.7 Conclusion: Architectural Consideration

A common feature of all the methods presented in this section is that they lead to architectures where the result is the output of an adder tree. This adder tree lends itself to a range of area/time tradeoffs which depends on the architectural target and also on the time/area constraints of the application.

However, as initially noted by Das Sarma and Matula, there are many applications where the last stage of the adder tree (which is the most costly as it involves the carry propagation) is not needed: Instead, the result may be provided in redundant form to the operator that consumes it. It is the case when a table-and-addition architecture provides the seed to a Newton-Raphson iteration, for instance: The result can be recoded (using Booth or modified Booth algorithm) without carry propagation to be input to a multiplier.

This remark shows that the cost of the adder tree depends not only on the target, but also on the application. For these reasons, the sequel focuses on minimizing the table size.

4 The Unified Multipartite Method

4.1 A General Input-Word Decomposition

Investigating what is common to Schulte and Stine's STAM and Muller's multipartite methods leads us to define a decomposition into subwords that generalize both (see Fig. 6):

- The input word is split into two subwords, A and B, of respective sizes α and β, with $\alpha + \beta = w_I$.
- The most significant subword A addresses the TIV.
- The least significant subword B will be used to address $m \geq 1$ TOs.

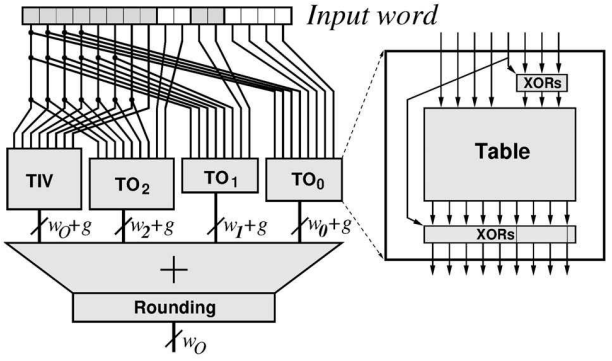


Fig. 7. Multipartite architecture.

  - B will in turn be decomposed into m subwords $B_0, \ldots, B_{m-1}$, the least significant being B_0.
  - A subword B_i starts at position p_i and consists of β_i bits. We have $p_0 = 0$ and $p_{i+1} = p_i + \beta_i$.
  - The subword B_i is used to address the TO_i, along with a subword C_i of length γ_i of A.
- Finally, to simplify notations, we will denote $\mathcal{D} = \{\alpha, \beta, m, (\gamma_i, p_i, \beta_i)_{i=0\ldots m-1}\}$ such a decomposition.

The maximum approximation error entailed by TO_i will be a function of (γ_i, p_i, β_i) which we will be able to compute exactly in Section 4.3. The TOs implementation will exploit their symmetry, just as in the STAM method.

The reader may check that the bipartite decomposition is a special case of our multipartite decomposition with $m = 1$, $\alpha = 2w_I/3$, $\gamma = w_I/3$, $\beta = \beta_0 = w_I/3$. Similarly, Stine and Schulte's STAM [4] is a multipartite decomposition where all the C_is are equal and Muller's multipartite approach [5] is a specific case of our decomposition where the γ_is are multiples of some integer.

Fig. 7 shows a general multipartite implementation, using symmetry. It should be clear that general decompositions are more promising than Stine and Schulte's in that they allow us to reduce the accuracy of the slopes involved in the TOs (and, thus, their size). They are also more promising than Muller's as they are more flexible (for example, the size of the input word need not be a multiple of some $2p + 1$). Our methodology will also be slightly more accurate than both in computing the slopes and in the error analysis. Section 5 will quantify these improvements.

4.2 An Algorithm for Choosing a Decomposition

Having defined the space of all the possible multipartite decompositions, we define in this section an efficient methodology to explore this space. The purpose of such an exploration is to select the best decomposition (in terms of speed or area) that fulfills the accuracy requirement known as *faithful rounding*: The returned result should be one of the two fixed-point numbers closest to the mathematical value. In other words, the total error should be smaller than the value ϵ_{total} of one unit in the last place (ulp):

$$\epsilon_{\text{total}} = (d - c)2^{-w_O}. \tag{2}$$

This error will be the sum of an approximation error, which depends only on the decomposition, and the various rounding errors.

Unfortunately, the tables cannot be filled with results rounded to the target precision: Each table would entail a maximum rounding error of $0.5\epsilon_{\text{total}}$, meaning that the total error budget of ϵ_{total} is unfeasible as soon as there is more than one table. The tables will therefore be filled with a precision greater than the target precision by g bits (guard bits). Thus, rounding errors in filling one table are now

$$\epsilon_{\text{rnd_table}} = 2^{-g-1}\epsilon_{\text{total}} \tag{3}$$

and can be made as small as desired by increasing g. The sum of these errors will be smaller than

$$\epsilon_{\text{rnd_m_tables}} = (m+1)\epsilon_{\text{rnd_table}}, \tag{4}$$

where $(m+1)$ is the number of tables.

However, the final summation is now also performed on g more bits than the target precision. Rounding the final sum to the target precision now entails a rounding error up to $\epsilon_{\text{rnd_final}} = 0.5\epsilon_{\text{total}}$. A trick due to Das Sarma and Matula [1] allows us to improve it to

$$\epsilon_{\text{rnd_final}} = 0.5\epsilon_{\text{total}}(1 - 2^{-g}). \tag{5}$$

This trick will be presented in Section 4.6.2.

This error budget suggests the following algorithm:

1. Choose the number of tables m. A larger m means smaller tables, but more additions.
2. Enumerate the decompositions
$$\mathcal{D} = \{\alpha, \beta, m, (\gamma_i, p_i, \beta_i)_{i=0\ldots m-1}\}.$$
3. For each decomposition $\mathcal{D}$,
   a. Compute the bounds $\epsilon_i^{\mathcal{D}}$ on the approximation errors entailed by each TO_i (see Section 4.3) and sum them to get $\epsilon_{\text{approx}}^{\mathcal{D}} = \sum_{i=0}^{m-1} \epsilon_i^{\mathcal{D}}$. Keep only those decompositions for which this error is smaller than the error budget.
   b. As the two other error terms $\epsilon_{\text{rnd_final}}$ and $\epsilon_{\text{rnd_m_tables}}$ depend on g, compute the smallest g allowing to match the total error budget. This will be detailed in Section 4.4.
   c. Knowing g allows precise evaluation of the size of the implementation of $\mathcal{D}$, as will be detailed in Section 4.5.
4. Synthesize the few best candidates to evaluate their speed and area accurately (with target constraints).

Enumerating the decompositions is an exponential task. Fortunately, there are two simple tricks which are enough to cut the enumeration down to less than a minute for 24-bit operands (the maximum size for which multipartite methods architectures make sense).

- The approximation error due to a TO_i is actually only dependent on the function evaluated, the input precision, and the three parameters p_i, β_i, and γ_i of this TO_i. It is therefore possible to compute all these errors only once and store them in a three-dimensional array $\epsilon_{\text{TO}}[p][\beta][\gamma]$. The size of this array is at most 24^3 double-precision floating-point numbers.
- For a given pair (p_i, β_i), this error grows as γ_i decreases. There exists a $\gamma_{\min}$ such that, for any $\gamma_i \leq \gamma_{\min}$, this error is larger than the required output precision. These $\gamma_{\min}(p_i, \beta_i)$ may also be computed once and stored in a table.

Finally, the enumeration of the (p_i, β_i) is limited by the relation $p_{i+1} = p_i + \beta_i$ and the enumeration on γ_i is limited by $\gamma_{\min} < \gamma_i < \alpha$. Note that we have only left out decompositions which were unable to provide faithful rounding. It would also be possible, in addition, to leave out decomposition whose area is bigger than the current best. This turns out not to be needed.

The rest of this section details the steps of this algorithm.

4.3 Computing the Approximation Error

Here, we consider a monotonic function with monotonic derivative (i.e., convex or concave) on its domain. This is not a very restrictive assumption: It is the case, after argument reduction, of all the functions studied by previous authors.

The error function we consider here is the difference $\varepsilon(x) = f(x) - \widetilde{f}(x)$ between the exact mathematical value and the approximation. Note that other error functions are possible, for example, taking into account the input discretization. The formulas set up here would not apply in that case, but it would be possible to set up equivalent formulas.

Using these hypotheses, it is possible to exactly compute, using only a few floating-point operations in double precision, the minimum approximation error which will be entailed by a TO_i with parameters p_i, β_i, and γ_i, and also the exact value to fill in these tables as well as in the TIV to reach this minimal error.

The main idea is that, for a given (p_i, β_i, γ_i), the parameters that can vary to get the smallest error are the slope $s(Ci)$ of the segments and the values $\text{TIV}(A)$. With our decomposition, several $\text{TIV}(A)$ will share the same $s(Ci)$. Fig. 8 (another zoom of Fig. 1) depicts this situation.

As the figure suggests, with our hypothesis of a monotonic (decreasing on the figure) derivative, the approximation error is maximal on the borders of the interval on which the segment slope is constant. The minimum $\epsilon_i^{\mathcal{D}}(C_i)$ of this maximum error is obtained when

$$\varepsilon_1 = -\varepsilon_2 = -\varepsilon_3 = \varepsilon_4 = \epsilon_i^{\mathcal{D}}(C_i) \tag{6}$$

with the notations of the figure. This system of equations is easily expressed in terms of $s(C_i)$, p_i, β_i, γ_i, TIV, and f. Solving this system gives the optimal slope[1] and the corresponding error:

$$s_i^{\mathcal{D}}(C_i) = \frac{f(x_2) - f(x_1) + f(x_4) - f(x_3)}{2\delta_i}, \tag{7}$$

$$\epsilon_i^{\mathcal{D}}(C_i) = \frac{f(x_2) - f(x_1) - f(x_4) + f(x_3)}{4}, \tag{8}$$

where (using the notations of Section 2.1)

1. Not surprisingly, the slope that minimizes the error is the average value of the slopes on the borders of the interval. Previous authors considered the slope at the midpoint of this interval.

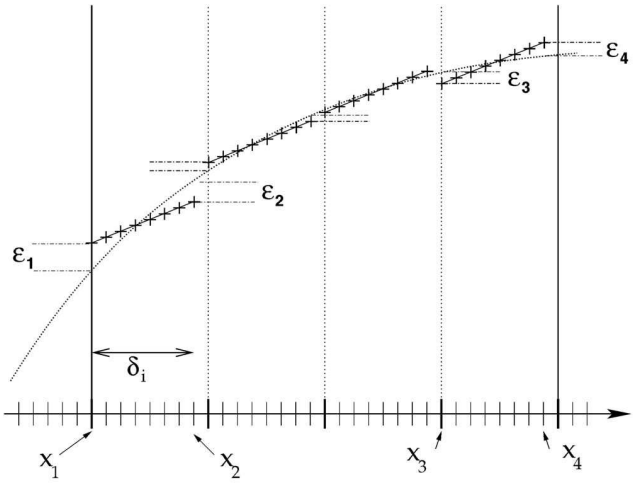


Fig. 8. Computing the approximation error.

$$\delta_i = (b-a)2^{-w_I+p_i}(2^{\beta_i}-1), \tag{9}$$

$$x_1 = a + (b-a)2^{-\gamma_i}C_i, \tag{10}$$

$$x_2 = x_1 + \delta_i, \tag{11}$$

$$x_3 = x_1 + (b-a)(2^{-\gamma_i} - 2^{-w_I+p_i+\beta_i}), \tag{12}$$

$$x_4 = x_3 + \delta_i. \tag{13}$$

Now, this error depends on C_i, that is, on the interval on which the slope is considered constant. For the same argument of convexity, it will be maximum either for $C_i = 0$ or for $C_i = 2^{\gamma_i} - 1$. Finally, the maximum approximation error due to TO_i in the decomposition $\mathcal{D}$ is:

$$\epsilon_i^{\mathcal{D}} = \max(|\epsilon_i^{\mathcal{D}}(0)|, |\epsilon_i^{\mathcal{D}}(2^{\gamma_i}-1)|). \tag{14}$$

In practice, it is easy to compute this approximation error by implementing (8) to (14). Altogether, it represents a few floating-point operations per TO_i.

4.4 Computing the Number of Guard Bits

The condition to ensure faithful rounding, $\epsilon_{\mathrm{rnd_m_tables}} + \epsilon_{\mathrm{rnd_final}} + \epsilon_{\mathrm{approx}}^{\mathcal{D}} < \epsilon_{\mathrm{total}}$ is rewritten using (2), (3), (4), and (5) as:

$$g > -w_O - 1 + \log_2((d-c)m) - \log_2((d-c)2^{-w_O-1} - \epsilon_{\mathrm{approx}}^{\mathcal{D}}).$$

If $\epsilon_{\mathrm{approx}}^{\mathcal{D}} \geq (d-c)2^{-w_O-1}$, $\mathcal{D}$ is unable to provide the required output accuracy. Otherwise, the previous inequality gives us the number g of extra bits that ensures faithful rounding:

$$g = \left\lceil -w_O - 1 + \log_2 \frac{(d-c)m}{(d-c)2^{-w_O-1} - \epsilon_{\mathrm{approx}}^{\mathcal{D}}} \right\rceil. \tag{15}$$

Our experiments show that it is very often possible to decrease this value by one and still keep faithful rounding. This is due to the actual worst-case rounding error in each table being smaller than the one assumed above, thanks to the small number of entries for each table. This question will be discussed in Section 5.2.

4.5 The Sizes of the Tables

Evaluating precisely the size and speed of the implementation of a multipartite decomposition is rather technology dependent and is out of the scope of the paper. We can, however, compute exactly (as other authors) the number of bits to store in each table.

The size in bits of the TIV is simply $2^{\alpha}(w_O + g)$. The TO_is have a smaller range than the TIV: Actually, the range of $\mathrm{TO}_i(C_i, *)$ is exactly equal to $|s_i(C_i) \times \delta_i|$. Again, for convexity reasons, this range is maximum either on $C_i = 0$ or $C_i = 2^{\gamma_i} - 1$:

$$r_i = \max(|s_i(0) \times \delta_i|, |s_i(2^{\gamma_i}-1) \times \delta_i|). \tag{16}$$

The number of output bits of TO_i (without the guard bits) is therefore

$$w_i = \left\lceil w_O + g - \log_2\left(\frac{d-c}{r_i}\right) \right\rceil. \tag{17}$$

In a symmetrical implementation of the TO_i, the size in bits of the corresponding table will be $2^{\gamma_i+\beta_i-1}(w_i - 1)$.

The actual costs (area and delay) of implementations of these tables and of multioperand adders are technology dependent. We present in Section 5.4 some results for Virtex-II FPGAs, showing that the bit counts presented above allow a predictive enough evaluation of the actual costs.

4.6 Filling the Tables

4.6.1 The Mathematical Values

An initial value $\mathrm{TIV}(A)$ provided by the TIV for an input subword A will be used on an interval $[x_l, x_r]$ defined (using the notations of Sections 2.1 and 4.3) by:

$$x_l = a + (b-a)2^{-\alpha}A, \tag{18}$$

$$x_r = x_l + \sum_{i=0}^{m-1} \delta_i. \tag{19}$$

On this interval, each TO_i provides a constant slope, as its C_i is a subword of A. The approximation error, which is the sum of the $\epsilon_i^{\mathcal{D}}(C_i)$ defined by (8), will be maximal for x_l and x_r (with opposite signs).

The TIV exact value that ensures that this error bound is reached is therefore (before rounding):

$$\widetilde{\mathrm{TIV}}(A) = \frac{f(x_l) + f(x_r)}{2}. \tag{20}$$

The TO_i values before rounding are (see Fig. 3):

$$\widetilde{\mathrm{TO}_i}(C_iB_i) = s(C_i) \times 2^{-w_I+p_i}(b-a)\left(B_i + \frac{1}{2}\right). \tag{21}$$

4.6.2 Rounding Considerations

This section reformulates the techniques employed by Stine and Schulte in [4] and using an idea that seems to appear first in the paper by Das Sarma and Matula [1].

The purpose is to fill our tables in such a way as to ensure that their sum (which we compute on $w_O + g$ bits)

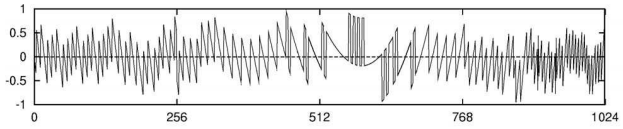


Fig. 9. Measured error (10-bit sine and $m = 2$).

always has an implicit 1 as its $(w_O + g + 1)$th bit. This reduces the final rounding error from $\epsilon_{\text{rnd_final}} = 2^{-w_O-1}$ to $\epsilon_{\text{rnd_final}} = 2^{-w_O-1} - 2^{-w_O-g-1}$.

To achieve this trick, remark that there are two ways to round a real number to $w_O + g$ bits with an error smaller than $\epsilon_{\text{rnd_table}} = 2^{-w_O-g-1}$. The natural way is to round the number to the nearest $(w_O + g)$-bit number. Another method is to truncate the number to $w_O + g$ bits and assume an implicit 1 in the $(w_O + g + 1)$th position.

To exploit the symmetry, we will need to compute the opposite of the value given by a TO_i. In two's complement, this opposite is the bitwise negation of the value, plus a 1 at the LSB. This leads us to use the second rounding method for the TO_i. Knowing that its LSB is an implicit 1 means that its negation is a 0 and, therefore, that the LSB of the opposite is also a 1. We therefore don't have to add the sign bit at the LSB. We store and bitwise negate the $w_i + g - 1$ bits of the TO_i and assume in all cases an implicit 1 at the $(w_O + g + 1)$th position.

Now, in order to reach our goal of always having an implicit 1 at the $(w_O + g + 1)$th bit of the sum, we need to consider the parity of m, the number of TO_is. If m is odd, the first rounding method is used for the TIV, if m is even, the second method is used. This way we always have $\lfloor m/2 \rfloor$ implicit ones, which we simply add to all the values of the TIV to make them explicit.

Finally, after summing the TIV and the TO_i, we need to round the sum, on $(w_O + g)$ bits with an implicit 1 at the $(w_O + g + 1)$th bit, to the nearest number on w_O bits. This can be done by simply truncating the sum (at no hardware cost), provided we have added half an LSB of the final result to the TIV when filling it.

Summing it up, the integer values that should fill the TO_is are

$$\text{TO}_i(C_iB_i) = \left\lfloor \frac{2^{w_O+g}}{d-c} \widetilde{\text{TO}}_i(C_iB_i) \right\rfloor \tag{22}$$

and the values that should fill the TIV are, if m is odd:

$$\text{TIV}(A) = \left\lfloor 2^{w_O+g} \times \frac{\widetilde{\text{TIV}}(A) - c}{d-c} + \frac{m-1}{2} + 2^{g-1} \right\rfloor \tag{23}$$

and, if m is even:

$$\text{TIV}(A) = \left\lfloor 2^{w_O+g} \times \frac{\widetilde{\text{TIV}}(A) - c}{d-c} + \frac{m}{2} + 2^{g-1} \right\rfloor. \tag{24}$$

4.7 Implementation

The methodology presented above has been implemented in a set of Java and C++ programs. These programs enumerate the decompositions, choose the best one with respect to accuracy and size, compute the actual values of the tables, and, finally, generate synthesizable VHDL.

TABLE 1
The Functions Tested, with Actual Values of w_I and w_O for 16-Bit Precision

function	input	output	w_I	w_O
sin	$[0, \pi/4[$	$[0, 1[$	16	16
2^x	$[0, 1[$	$[1, 2[$	16	15
$1/x$	$[1, 2]$	$[1/2, 1]$	15	15

Our tools also perform various additional checks. Storing the $\widetilde{\text{TIV}}$ and $\widetilde{\text{TO}}_i$, they measure the actual value of $\epsilon^{\mathcal{D}}_{\text{approx}}$. We find that the predicted values are indeed accurate to 10^{-7}. They similarly compute the maximal final error and check that this error is really smaller than the expected accuracy (see Fig. 9 for an example of output).

5 Results

This section studies the size and area of operators obtained using this methodology. The functions used are given in Table 1 with their input and output intervals. Some of these functions are identical to those in [4]. Notice that the bits that are constant over the input or output interval are not counted in w_I or w_O. Also notice that the output interval for the sine function is not the image of the input interval (which would be $[0, 1/\sqrt{2}[$), but, rather, a larger interval which will allow easy argument reduction using trigonometric identities.[2]

5.1 Comparison with Previous Work

Tables 2 and 3 present the best decomposition obtained for 16-bit and some 24-bit operands for a few functions. In these tables, we compare our results with the best-known results from the work of Schulte and Stine [4]. We can notice a size improvement up to 50 percent. The size for $1/x$ and $m = 1$ is larger than the reference size. After investigation, this is due to rounding errors compensating, in this specific case leading to an overestimated g.

5.2 Further Manual Optimization

The results obtained by the automatic method presented above can usually be slightly improved, up to 15 percent in terms of table sizes. The reason is that the automatic algorithm assumes that worst-case rounding will be attained in filling the tables, which is not the case. As we have many TO_is with few entries (typically, 2^5 to 2^8 entries for 16-bit operands in Table 2), there is statistically a good chance that the sum of the table-filling rounding errors is significantly smaller than predicted. This is a random effect which can only be tested by an exhaustive check of the architecture. However, in a final stage, it is worth trying several slight variations of the parameters, which can be of two types:

2. The specification, in the two papers by Schulte and Stine [3], [4] of the input and output intervals for the sine function is inconsistent. The input interval should probably read $[0, \pi/4[$ instead of $[0, 1[$. The output mapping is also unclear. Therefore, the comparisons concerning the sine function in this paper may be misleading.

TABLE 2
Best Decomposition Characteristics and Table Sizes for 16-Bit Operands

f	m	α	β	γ_i	β_i	g	tables	size	ref size
sin	1	10	6	5	6	1	$17.2^{10}+6.2^{10}$	23552	32768
	2	8	8	6 5	4 4	3	$19.2^{8}+10.2^{9}+6.2^{8}$	11520	20480
	3	8	8	7 5 4	2 3 3	2	$18.2^{8}+9.2^{8}+7.2^{7}+4.2^{6}$	8064	17920
	4	8	8	6 6 5 5	2 2 2 2	3	$19.2^{8}+10.2^{7}+8.2^{7}+6.2^{6}+4.2^{6}$	7808	na
2^x	1	10	6	5	6	1	$16.2^{10}+6.2^{10}$	22528	24576
	2	8	8	7 5	3 5	2	$17.2^{8}+9.2^{9}+6.2^{9}$	12032	14592
	3	8	8	7 6 4	2 3 3	2	$17.2^{8}+9.2^{8}+7.2^{8}+4.2^{6}$	8704	13568
	4	8	8	7 6 5 4	2 2 2 2	2	$17.2^{8}+9.2^{8}+7.2^{7}+5.2^{6}+3.2^{5}$	7968	na
$1/x$	1	10	5	7	5	1	$16.2^{10}+6.2^{11}$	28672	24576
	2	9	6	7,6	3,3	3	$18.2^{9}+9.2^{9}+6.2^{8}$	15360	16896
	3	9	6	8,7,5	2,2,2	2	$17.2^{9}+8.2^{9}+6.2^{8}+4.2^{6}$	14592	15872

TABLE 3
Best Decomposition Characteristics and Table Sizes for 24-Bit Operands

f	m	α	β	γ_i	β_i	g	tables	size	ref size
sin	1	15	9	7	9	3	$27.2^{15}+11.2^{15}$	1245184	1998848
	2	13	11	10 6	4 7	3	$27.2^{13}+13.2^{13}+9.2^{12}$	364544	753664
	3	12	12	10 9 6	3 4 5	4	$28.2^{12}+15.2^{12}+12.2^{12}+8.2^{10}$	233472	610304
	4	12	12	10 10 8 7	2 2 4 4	4	$28.2^{12}+15.2^{11}+13.2^{11}+11.2^{11}+7.2^{10}$	201728	507904
	5	12	12	10 10 9 7 6	2 2 2 3 3	4	$28.2^{12}+15.2^{11}+13.2^{11}+11.2^{10}+9.2^{9}+6.2^{8}$	189440	491520
	6	12	12	10 10 9 8 7 5	2 2 2 2 2 2	4	$28.2^{12}+15.2^{11}+13.2^{11}+11.2^{10}+9.2^{9}+7.2^{8}+5.2^{6}$	190016	na
2^x	1	15	9	8	9	1	$24.2^{15}+9.2^{16}$	1376256	1474560
	2	13	11	10 8	5 6	2	$25.2^{13}+12.2^{14}+7.2^{13}$	458752	581632
	3	12	12	11 9 7	3 4 5	3	$26.2^{12}+14.2^{13}+11.2^{12}+7.2^{11}$	280576	425984
	4	12	12	11 10 8 8	2 3 3 4	3	$26.2^{12}+14.2^{12}+12.2^{12}+9.2^{10}+6.2^{11}$	234496	360448
	5	12	12	11 10 9 8 8	2 2 2 3 3	3	$26.2^{12}+14.2^{12}+12.2^{11}+10.2^{10}+8.2^{10}+5.2^{10}$	211968	356352
	6	12	12	11 10 9 8 8 8	2 2 2 2 2 2	3	$26.2^{12}+14.2^{12}+12.2^{11}+10.2^{10}+8.2^{9}+6.2^{9}+4.2^{9}$	207872	na
$1/x$	1	15	8	9	8	5	$28.2^{15}+13.2^{16}$	1769472	1933312
	2	14	9	11 8	3 6	3	$26.2^{14}+12.2^{13}+9.2^{13}$	598016	884736
	3	13	10	12 10 8	2 3 5	4	$27.2^{13}+14.2^{13}+12.2^{12}+9.2^{12}$	421888	688128
	4	13	10	11 11 10 9	2 2 3 3	4	$27.2^{13}+14.2^{12}+12.2^{12}+10.2^{12}+7.2^{11}$	382976	524880
	5	13	10	11 11 10 9 8	2 2 2 2 2	5	$28.2^{13}+15.2^{12}+13.2^{12}+11.2^{11}+9.2^{10}+7.2^{9}$	379392	651264

- g may be decremented (as this is a variation of one parameter, it should actually be automatically performed).
- Some of the γ_i can be decremented (meaning less accurate slope). Remark that this negative effect on the mathematical error may be compensated by the halving of the number of values in the corresponding TO_i, which doubles the expected distance to the worst-case rounding.

Table 4 gives the example of a lucky case, with 11.5 percent improvement in size. These values of the γ_i even produce a method error of more than 0.5 ulp, which the lucky rounding compensates.

TABLE 4
Effect of Manual Optimization of the Parameters (Sine, 16 Bits, $m = 4$)

	γ_i	g	size	max. measured error in ulp
automatic	6 6 5 5	3	7808	0.915
fine-tuned	6 **5 4 3**	3	**6912**	0.939

Such a size improvement is reflected in the FPGA implementation: See Table 8 in Section 5.4.

5.3 Multipartite Are Close to Optimal among Order-One Methods

We remark in Table 2 and Table 3 that, for large values of m, the parameter α is close to $w_I/2$. Consider the family of linear (order-one) approximation schemes using some α bits of the input to address a TIV. There is an intrinsic lower bound on α for this family and it is the α for which the (mathematical) approximation error prevents faithful rounding. Generally speaking, this bound is about $w_I/2$, as given by the Taylor formula (and $w_I/3$ for order-two methods, etc.). This bound can be computed for each function exactly and we find that the best multipartite decomposition almost always exhibits the smallest α compatible with a faithful approximation.

Combined with the observation that the main contribution to the total size is always the TIV, this allows us to claim that our best multipartite approximations are close to the global optimal in the family of linear approximation schemes. More accurately, even the availability of a costless

TABLE 5
Virtex-II FPGA Implementation for Some Functions (16-Bit)

f	m	slices	(% of chip)	delay	T_{synth}	CF
sin $[0, \pi/4[$	1	711	13.9%	24.2 ns	49 s	16.6
	2	311	6.1%	21.2 ns	22 s	16.9
	3	265	5.2%	22.3 ns	16 s	15.2
	4	280	5.5%	24.8 ns	16 s	13.9
2^x	1	677	13.2%	25.7 ns	44 s	16.6
	2	376	7.3%	24.4 ns	25 s	16.0
	3	283	5.5%	22.8 ns	16 s	15.4
	4	295	5.8%	21.2 ns	18 s	15.6
$\frac{1}{x}$	1	821	16.0%	27.6 ns	69 s	17.4
	2	490	9.6%	28.1 ns	36 s	15.7
	3	474	9.3%	27.1 ns	27 s	15.4

perfect multiplier to implement a linear scheme will remove only the TO_is, which accounts for less than half the total size.

5.4 FPGA Implementation Results

In this section, the target architecture is a Virtex-II 1000 FPGA from Xilinx (XC2V1000-fg456-5). All the synthesis, place, and route processes have been performed using the Xilinx ISE XST 5.2.03i tools. The generated VHDL operators have been optimized for area with a high effort (the results are very close using a speed target). Area is measured in number of slices (two LUTs with four address bits per slice in Virtex-II devices), there are 5,120 available slices in a XC2V1000. The delay is expressed in nanoseconds. We also report the delay of the operator and its complete synthesis time (including place and route optimizations) T_{synth}. The compression factor CF is the ratio number of bits/number of LUTs; it measures the additional compression capabilities of the logical optimizer. In the target FPGAs, look-up tables may hold 16 bits, so a CF larger than 16 indicates such a size improvement.

Table 5 presents some synthesis results for the functions described in Table 1. The time required to compute the optimal decomposition (using the algorithm presented in Section 4.2) is always negligible compared to T_{synth}.

Table 6 details the evolution of area and delay with respect to input size for the sine function. Note that, in the Xilinx standard sine/cosine core [18], which uses a simple tabulation, the input size is limited to 10 bits, meaning an 8-bit table after quadrant reduction.[3]

Some results for 24-bit are also given in Table 7, showing that 24-bit precision is the practical limit of multipartite methods on such an FPGA. The economical limit, of course, is probably less than 20 bits, as suggested by Table 6.

These results show that, when the number of TO_is m increases, the operator size (the number of LUTs) decreases. The size gain is significant when we use a tripartite method ($m = 2$) instead of a bipartite one ($m = 1$). For larger values of m, this decrease is less important. Sometimes, a slight increase is possible for even larger values of m (e.g., $m = 3$ to $m = 4$ for the sine and 2^x function). This is due to the extra cost of the adder with an additional input, the XOR gates, and the sign extension mechanism that is not compensated by the tables size reduction. This is also reflected in the operator delay.

3. Using on-chip RAM blocks, the simple table approach allows up to 16 bits, meaning $w_I = 14$ after quadrant reduction.

TABLE 6
Virtex_II FPGA Implementation of the Sine Function for Various Sizes

f	w_I	m	slices	delay	T_{synth}	CF
sin $[0, \pi/4[$	8	2	19	16.6 ns	7 s	5.6
	12	3	76	18.0 ns	12 s	9.5
	16	4	280	24.8 ns	18 s	13.9
	20	5	1209	34.5 ns	96 s	16.5
	24	5	4954	43.0 ns	660 s	19.1

The compression factor CF is more or less constant (just a slight decrease with m). This fact can be used to predict the size after synthesis on the FPGA from the table size in bits. As each LUT in a Virtex FPGA can only store 16 bits of memory, we can deduce from these tables that the synthesizer performs some logic optimization inside each table. The compression factor decreases when m increases because the minimization potential is smaller on small tables than on larger ones. The synthesis time also decreases when m increases.

We investigated in [19] the use of ad hoc table-compression techniques. For this, we used JBits, a low-level hardware description language developed by Xilinx. Compression factors of up to 19 could be obtained for 16-bit and 20-bit sines at the expense of two months of development.

An important remark is that smaller operators turn out to be faster on FPGAs: Using a multipartite compression improves both speed and area.

5.5 Comparisons with Higher-Order Methods

Results for 24-bit operands should also be compared to the ATA architecture published by Wong and Goto for this specific case [15]. They use six tables for a total of 868,352 bits and, altogether, nine additions. Our results are thus both smaller and faster. However, it should be noted that five of the six tables in their architecture have the same content, which means that a sequential access version to a unique table should be possible (provided the issue of rounding is studied carefully). This sequential architecture would involve only about 16Kbits of tables, but it would be five times slower.

The remainder of this section compares with recently published methods involving multipliers. Such methods have to be used for $w_I > 24$ bits: If we consider that the maximum admissible table size is 2^{12} entries, this limit is reached by the multipartite approach for $w_I = 24$. Our aim here is to give a more quantitative idea of the domains of

TABLE 7
Virtex_II FPGA Implementation of the Sine Function (24-Bit)

f	m	slices	delay	T_{synth}	CF
sin $[0, \pi/4[$	1	640%	–	–	–
	2	170%	–	–	–
	3	105%	–	–	–
	4	101%	–	–	–
	5	4954	43.0 ns	660 s	19.1
	6	5004	36.1 ns	520 s	19.0

TABLE 8
Effect of Fine-Tuning on Virtex-II Implementation

f	m	automatic		fine-tuned	
		slices	delay	slices	delay
sin on $[0, \pi/4[$	4	280	24.8 ns	**258**	**24.6** ns

TABLE 9
Comparison with Defour et al.'s Approach

f	$w_I = w_O$	table size in [16]	multipartite size
sin $[0, \pi/4[$	15	3536	4608 ($m = 4$)
	20	15936	40704 ($m = 4$)
	24	71776	210728 ($m = 4$)

relevance of the various methods. Of course, this will depend on the function and on the target hardware.

The method published recently by Defour et al. uses two small multipliers in addition to tables and adders [16]. Note that recent FPGAs include small $18 \times 18 \rightarrow 35$-bit multipliers which can be used for Defour et al.'s architecture. This method has several restrictions: It is more rigid than the multipartite approach as it uses a decomposition of the input word into five subwords of the same size. As a consequence, for some functions, it is unable to provide enough precision for faithful rounding when $w_O = w_I$. Table 9 gives some comparisons of this method with the multipartite approach. We chose $m = 4$ so that the number of additions is the same in both approaches. According to this table, a multipartite approach will be preferred for precisions smaller than 15 bits and Defour et al.'s approach will be preferred for precisions greater than 20 bits, as far as size only is considered. For $w_I = 15$, Defour et al.'s tables are still smaller, but the size of the multipliers will compensate, so multipartite should be both smaller and faster. If speed is an issue, the delay of the multipliers will play in favor of multipartite tables.

The architecture by Piñeiro et al. [17] involves a squarer and a multiplier and 12,544 bits of tables for 24-bit $1/x$. An FPGA implementation sums up to 565 slices, which is only slightly more than our 16-bit implementation at 474 slices. This again suggests that multipartite methods are not relevant for more than 16 bits of precision, as far as size only is concerned.

Finally, we have recently compared a second-order approach using two multipliers and the multipartite approach on the specific case of addition/subtraction in the logarithm number system. The functions involved are then $\log_2(1 + 2^x)$ and $\log_2(1 - 2^x)$ and a restricted form of faithful rounding is used. In this case, we have only one point of comparison, corresponding to about 12 bits of precision. The multipartite approach is better both in terms of speed and area in this case [8].

In all these cases, it should be noted that the simplicity and generality of the multipartite approach may be a strong argument. Implementing a new function is a matter of minutes from the start down to VHDL code. This code is then efficiently synthesized, at least on FPGAs, because it only contains additions. Comparatively, approaches relying on multiplications need much more back-end work, typically requiring to design custom operators as Piñeiro et al. does.

5.6 Limits of the Method

5.6.1 Nonmonitonicities

Our approach maximizes the approximation error (within the bounds of faithful rounding) to minimize the hardware cost. This has the drawback of entailing nonmonotonicities at some of the borders between intervals: See, for instance, Fig. 8 around $x = x_3$. These nonmonotonicities are never bigger than one ulp thanks to faithful rounding. It is a problem of all the faithful approximation schemes, but the multipartite method as presented makes it happen quite often.

The subject of monotonicity in the context of bipartite tables has been studied by Iordache and Matula [7]. They reverse-engineered the AMD K6-2 implementation of fast reciprocal and reciprocal square root instructions, part of the 3D-Now instruction set extensions. They found that bipartite approximations were used, that the reciprocal was monotonic, and that the reciprocal square root was not. They also showed that the latter could be tuned to become monotonic, at the expense of a larger table size (7.25 KB instead of 5.5). This tuning involves increasing the output size of the TIV and an exhaustive exploration of what value these extra bits should take.

In general, if monotonicity is an important property, it can be enforced simply in a multipartite approximation by using appropriate slopes and TIV values. For instance, monotonically increasing functions with decreasing derivatives (as on our figures) may use the slope on the right of the interval instead of the middle, ensuring that the approximation is monotonic. This means a larger maximum approximation error, however. Rounding errors can then be kept within bounds that ensure monotonicity by increasing g as in [7]. All this entails increased hardware cost. A general and systematic study of this question remains to be done.

5.6.2 Infinite Derivative

There are also functions for which this methodology will not work. The square root function on $[0, 1[$, for example, although it may perfectly be stored in a single table, has an infinite derivative in 0 which breaks multipartite methods. We have never seen any mention of this problem in the literature, either. One solution in such cases is to split the input interval into two intervals $[0, 2^{-\zeta}[$ (on which the function is tabulated in a single table) and $[2^{-\zeta}, 1[$, where the multipartite method is used. The optimal ζ can probably be determined by enumeration.

6 Conclusion

We have presented several contributions to table-lookup-and-additions methods. The first one is to unify and generalize two complimentary approaches to multipartite tables by Stine and Schulte and by Muller. The second one is to give a method for optimizing such bipartite or multipartite tables which is more accurate than what could be previously found in the literature. Both these improvements have been implemented in general tools that can generate optimal multipartite tables from a wide range of specifications (input and output accuracy, delay, area).

These tools output VHDL which has been synthesized for Virtex FPGAs. Our method provides up to 50 percent smaller solutions than ones of the best literature results. This paper also discusses the limits of this approach. By comparing with higher-order methods, we conclude that multipartite methods provide the best area/speed tradeoff for precisions from 8 to 16 bits.

With the observation that multipartite methods are optimal among first-order methods, this paper leaves little room for improvement in such methods. Future work should now aim at providing methods for exploring the design space of higher-order approximations with the same ease as multipartite methods allow for first-order approximations.

ACKNOWLEDGMENTS

This work was partially supported by an ACI grant from the French Ministry of Research and Education, and by Xilinx University Programme. The authors would like to thank the anonymous referees for many useful remarks and comments.

REFERENCES

[1] D. Das Sarma and D. Matula, "Faithful Bipartite ROM Reciprocal Tables," *Proc. 12th IEEE Symp. Computer Arithmetic,* S. Knowles and W. McAllister, eds., pp. 17-28, 1995.
[2] H. Hassler and N. Takagi, "Function Evaluation by Table Look-Up and Addition," *Proc. 12th IEEE Symp. Computer Arithmetic,* S. Knowles and W. McAllister, eds., pp. 10-16, 1995.
[3] M. Schulte and J. Stine, "Approximating Elementary Functions with Symmetric Bipartite Tables," *IEEE Trans. Computers,* vol. 48, no. 8, pp. 842-847, Aug. 1999.
[4] J. Stine and M. Schulte, "The Symmetric Table Addition Method for Accurate Function Approximation," *J. VLSI Signal Processing,* vol. 21, no. 2, pp. 167-177, 1999.
[5] J.-M. Muller, "A Few Results on Table-Based Methods," *Reliable Computing,* vol. 5, no. 3, pp. 279-288, 1999.
[6] J.-M. Muller, *Elementary Functions, Algorithms and Implementation.* Boston: Birkhauser, 1997.
[7] C. Iordache and D.W. Matula, "Analysis of Reciprocal and Square Root Reciprocal Instructions in the AMD K6-2 Implementation of 3DNow!" *Electronic Notes in Theoretical Computer Science,* vol. 24, 1999.
[8] J. Detrey and F. de Dinechin, "A VHDL Library of LNS Operators," *Proc. 37th Asilomar Conf. Signals, Systems, and Computers,* Oct. 2003.
[9] F. de Dinechin and A. Tisserand, "Some Improvements on Multipartite Table Methods," *Proc. 15th IEEE Symp. Computer Arithmetic,* N. Burgess and L. Ciminiera, eds., pp. 128-135, June 2001.
[10] D. Matula, "Improved Table Lookup Algorithms for Postscaled Division," *Proc. 15th IEEE Symp. Computer Arithmetic,* N. Burgess and L. Ciminiera, eds., pp. 101-108, June 2001.
[11] F.J. Taylor, R. Gill, J. Joseph, and J. Radke, "A 20 Bit Logarithmic Number System Processor," *IEEE Trans. Computers,* vol. 37, no. 2, Feb. 1988.
[12] D.M. Lewis, "An Architecture for Addition and Subtraction of Long Word Length Numbers in the Logarithmic Number System," *IEEE Trans. Computers,* vol. 39, no. 11, Nov. 1990.
[13] J.N. Coleman and E.I. Chester, "Arithmetic on the European Logarithmic Microprocessor," *IEEE Trans. Computers,* vol. 49, no. 7, pp. 702-715, July 2000.
[14] B. Lee and N. Burgess, "A Dual-Path Logarithmic Number System Addition/Subtraction Scheme for FPGA," *Proc. Field-Programmable Logic and Applications,* Sept. 2003.
[15] W. Wong and E. Goto, "Fast Evaluation of the Elementary Functions in Single Precision," *IEEE Trans. Computers,* vol. 44, no. 3, pp. 453-457, Mar. 1995.
[16] D. Defour, F. de Dinechin, and J.-M. Muller, "A New Scheme for Table-Based Evaluation of Functions," *Proc. 36th Asilomar Conf. Signals, Systems, and Computers,* Nov. 2002.
[17] J.A. Piñeiro, J.D. Bruguera, and J.-M. Muller, "Faithful Powering Computation Using Table Look-Up and a Fused Accumulation Tree," *Proc. 15th IEEE Symp. Computer Arithmetic,* N. Burgess and L. Ciminiera, eds., pp. 40-47, June 2001.
[18] *Sine/Cosine Lookup Table V4.2,* Xilinx Corp., Nov. 2002, www.xilinx.com/ipcenter/.
[19] F. de Dinechin and J. Detrey, "Multipartite Tables in Jbits for the Evaluation of Functions on FPGAs," *Proc. IEEE Reconfigurable Architecture Workshop, Int'l Parallel and Distributed Symp.,* Apr. 2002.



Florent de Dinechin received the DEA from the École Normale Supérieure de Lyon (ENS-Lyon) in 1993 and the PhD degree from the Université de Rennes 1 in 1997. After a postdoctoral position at Imperial College, London, he is now a permanent lecturer at ENS-Lyon in the Laboratoire de l'Informatique du Parallèlisme (LIP). His research interests include computer arithmetic, software and hardware evaluation of functions, computer architecture, and FPGAs. He is a member of the IEEE and the IEEE Computer Society.



Arnaud Tisserand received the MSc degree and the PhD degree in computer science from the École Normale Supérieure de Lyon, France, in 1994 and 1997, respectively. He is with the French National Institue for Research in Computer Science and Control (INRIA) and the Laboratoire de l'Informatique du Parallèlisme (LIP) in Lyon, France. He teaches computer architecture and VLSI design at the École Normale Supérieure de Lyon, France. His research interests include computer arithmetic, computer architecture, and VLSI design. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

A.4. Optimisation de la multiplication par des constantes

Reproduction de l'article [12] paru dans le journal *IEEE Transactions on Computers* en octobre 2005 et résumé en section 2.6.1.

auteurs	N. Boullis et A. Tisserand
titre	*Some Optimizations of Hardware Multiplication by Constant Matrices*
journal	*IEEE Transactions on Computers*
date	octobre 2005
volume	54
numéro	10
pages	1271–1282
numéro DOI	10.1109/TC.2005.168

Some Optimizations of Hardware Multiplication by Constant Matrices

Nicolas Boullis and Arnaud Tisserand, *Member*, *IEEE*

Abstract—This paper presents some improvements on the optimization of hardware multiplication by constant matrices. We focus on the automatic generation of circuits that involve constant matrix multiplication, i.e., multiplication of a vector by a constant matrix. The proposed method, based on number recoding and dedicated common subexpression factorization algorithms, was implemented in a VHDL generator. Our algorithms and generator have been extended to the case of some digital filters based on multiplication by a constant matrix and delay operations. The obtained results on several applications have been implemented on FPGAs and compared to previous solutions. Up to 40 percent area and speed savings are achieved.

Index Terms—Computer arithmetic, multiplication by constants, common subexpressions sharing, FIR filter.

1 Introduction

Important optimizations of the speed, area, and power consumption of circuits can be achieved by using dedicated operators instead of general ones whenever possible. Multiplication by constant is a typical example. Indeed, if one operand of the multiplication is constant, one can use some shifts and additions/subtractions to perform the operation instead of using a complete multiplier. This usually leads to smaller, faster, and less power-consuming circuits.

Applications involving multiplication by constant are common in digital signal processing, image processing, control, and data communication. Finite impulse response (FIR) filters, discrete cosine transform (DCT), and discrete Fourier transform (DFT), for instance, are central operations in high-throughput systems and they use a huge amount of such operations. Their optimization widely impacts the performance of the global system that uses them. In [1], there is an analysis of the frequency of such operations.

The problem of the optimization of multiplication by constant has been studied for a long time. For instance, the famous recoding presented by Booth in [2] can simplify both the multiplications by constants and the complete multiplications. This recoding and the algorithm proposed by Bernstein in [3] were widely used on processors without a multiplication unit.

The main goal in this problem is the minimization of the computation quantity. The multiplication by constant problem seems to be simple, but its resolution is a hard problem due to its combinatorial properties. This problem can occur in more or less complex contexts. In the case of a single multiplication of one variable by one small constant, it may be possible to explore the whole parameter space. But, in the case of the multiplication of several variables by several constants, the space to explore is so huge that one has to use heuristics.

A first solution proposed to optimize multiplication by constant was the use of the constant recoding, such as Booth's. This solution just avoids long strings of consecutive ones in the binary representation of the constant. Better solutions are based on the factorization of common subexpressions, simulated annealing, tree exploration, pattern search methods, etc.

Our work deals with multiplication of constant matrix, i.e., one useful form of the multiplication of several variables by several constants. A lot of applications involve such linear operations. This method is based on constants recoding followed by some dedicated common subexpression factorization algorithms. We also extended our method to the case of some digital filters. Our solution is able to handle filters based on constant matrix multiplication and delay operations (such as FIR filters). The proposed method was implemented in a VHDL generator. The generated results for several applications have been implemented on Xilinx FPGAs (field programmable gate arrays) and compared to other solutions. Some significant improvements have been obtained: up to 40 percent area saving in the DCT case and from 20 percent up to 30 percent in the case of some FIR filters, for instance.

This paper is an extended version of the paper [4] presented at the 16th IEEE Symposium on Computer Arithmetic (ARITH16) in June 2003. It is organized as follows: The problem is presented in Section 2. In Section 3, some related works are presented. Our algorithm is presented in Section 4. The developed generator and the target architectures are discussed in Section 5. The results of the implementation of some applications and their comparison to other solutions are presented in Section 6. Finally, the specific case of digital filters is presented in Section 7.

2 Problem Definition

In this paper and in the related works, the central problem is the substitution of complete multipliers by an optimized

• *The authors are with the Arénaire Project (CNRS-ENSL-INRIA-UCBL) LIP, École Normale Supérieure de Lyon, 46 allée d'Italie, F-69364 Lyon, France. E-mail: {nicolas.boullis, arnaud.tisserand}@ens-lyon.fr.*

Manuscript received 24 Nov. 2003; revised 24 Mar. 2005; accepted 6 Apr. 2005; published online 16 Aug. 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-0225-1103.

0018-9340/05/$20.00 © 2005 IEEE Published by the IEEE Computer Society

sequence of shifts and additions and/or subtractions. We focus on integers, but all the results can be easily extended to other fixed-point representations.

All the values are represented using a standard unsigned radix-2 notation or two's complement unless it is specified. The notation $x \ll k$ denotes the k-bit left shift of the variable x (i.e., $x \times 2^k$). As we look at hardware implementations, we assume that shift is just routing and that addition and subtraction have the same area and speed cost.

As an example, let us compute p as the product of the input variable x by the constant $c = 111463 = 11011001101100111_2$. The simplest algorithm uses the distributive property of multiplication. There is one addition of x (after some potential shift) for each one in the binary representation of c. In the case $c = 111463$, it leads to 10 additions:

$$\begin{aligned} 111463x = &(x \ll 16) + (x \ll 15) + (x \ll 13) + (x \ll 12) \\ &+ (x \ll 9) + (x \ll 8) + (x \ll 6) + (x \ll 5) \\ &+ (x \ll 2) + (x \ll 1) + x. \end{aligned}$$

The central point in this problem is the minimization of the total number of operations. It can be significantly reduced by using a recoding of the constant and/or subexpression elimination and sharing. The theoretical complexity of this problem still seems to be unknown.

Depending on the target application, this problem can occur at different levels of complexity. It starts with the multiplication of one variable by one constant. After, the multiple constant multiplication (MCM) problem appears with the multiplication of one variable by several constants [5]. In this present work, we deal with a more general version of this problem with the multiplication of one variable vector by one constant matrix: the constant matrix multiplication. We also deal with the case of some digital filters that involve multiplication by a constant matrix and delay operations.

3 Related Works

There are at least four types of methods to address the multiplication by constant problem:

- direct recoding methods,
- evolutionary methods,
- cost-function-based search methods, and
- pattern search methods.

3.1 Direct Recoding Methods

The recoding of the multiplier operand is very frequently used in multipliers. The famous Booth's recoding [2] replaces long strings of ones by values with more zeros. The modified Booth recoding is often used in variable multipliers because it reduces the area of the operators. See [6] or [7] for the use of Booth or modified Booth recodings for multiplication. But, the Booth recoding is generally not used in constant multipliers because the number of nonzero digits of the recoded operand is not minimal.

A minimal recoding ensures that the number of nonzero digits in the recoded value is as small as possible. In the radix-2 signed digit (SD) representation, the digits belong to the set $\{\bar{1} = -1, 0, 1\}$. A number is said to be in the canonical signed digit (CSD) format if no two nonzero digits are consecutive; such a code is minimal. Using a minimal recoding, such as CSD, on an n-bit unsigned value, the number of nonzero digits is bounded by $(n+1)/2$ and it tends asymptotically to an average value of $n/3 + 1/9$, as shown in [8]. For our example, using CSD recoding we have: $111463 = 11011001101100111_2 = 100\bar{1}0\bar{1}0100\bar{1}0\bar{1}0100\bar{1}_2$ and the product $p = c \times x$ is reduced to seven additions/subtractions:

$$\begin{aligned} 111463x = &(x \ll 17) - (x \ll 14) - (x \ll 12) + (x \ll 10) \\ &- (x \ll 7) - (x \ll 5) + (x \ll 3) - x. \end{aligned}$$

The KCM algorithm [9] was specifically designed for LUT-based FPGAs (LUT means look-up table). It decomposes the binary representation of the variable into 4-bit chunks (a radix-16 representation). Each partial product, deduced by the product of the constant by one radix-16 digit of the variable, is read in a small multiplication table. Those tables are addressed by one radix-16 digit, which perfectly fulfills the 4-input LUT resources of the target FPGAs. There is a more general version of this decomposition problem with distributed arithmetic. For instance, in [10], distributed arithmetic was used on a 16-point DCT operator with an area saving of 17 percent compared to the direct implementation of the whole computation.

There are some recent works on the use of high-radix recoding. For instance, in [11], the authors implement some FIR filters using a radix-8 representation with punctured coefficients. Those coefficients are represented using digits in the set $\{0, \pm1, \pm2, \pm4\}$ instead of the set $\{0, \pm1, \pm2, \pm3, \pm4\}$. This is a lossy representation, so they have to deal with some additional accuracy requirements. In our case, we want to study this problem for a lossless representation, but this approach seems to be interesting for future research.

The recoding of the constants using sum of power of two (SOPOT) values is a standard method. In this method, the theoretical coefficients are quantified to values that can be expressed using a small number of nonzero bits (compared to the whole word length). This method is often used in signal processing filters, see [12] and [13] for recent filter applications.

Another recoding solution was proposed with the use of multiple-radix representations and especially with the double-base number system (DBNS) [14]. In this solution, the authors use both radices 2 and 3 simultaneously, i.e., the values are expressed by $a = \sum_{i,j} a_{i,j} 2^i 3^j$ with $a_{i,j} \in \{0, 1\}$. This multiple-radix representation, sometimes useful in some analog circuits, does not seem to be efficient in the multiplication by constant problem in digital circuits. In [15], multiple-radix or mixed-radix representations have been used in the implementation of FIR multirate converters. A small area gain is reported using this kind of representation.

3.2 Evolutionary Methods

Some evolutionary methods, such as evolutionary graph generation [16], have been proposed to generate arithmetic circuits and especially for constant multipliers. These methods based on genetic algorithms seem to provide very poor results. For instance, in [16], the results are slightly better than the straightforward CSD encoding, which is

very far from the best known results. Furthermore, it seems that these methods are limited to the problem of multiplication by a few constants and have never been used to produce more complex circuits.

3.3 Cost-Function-Based Search Methods

The algorithm presented by Bernstein in [3] allows some intermediate values that are used only once in recoding methods to be reused. A more detailed and corrected version of this algorithm can be found in [17]. The algorithm, based on a tree exploration, defines three kinds of operations: $t_{i+1} = (t_i \ll k)$, $t_{i+1} = (t_i \pm x)$, and $t_{i+1} = ((t_i \ll k) \pm t_i)$. A cost can be specified for each operation according to the target technology. The cost function used to guide the exploration is the sum of the costs of all the involved operations. This algorithm only shares some common subexpressions. For our example, $p = c \times x$ with $c = 111463$, this algorithm gives a 5-addition solution:

$$\begin{aligned} t_1 &= (((x \ll 3) - x) \ll 2) - x, \\ t_2 &= t_1 \ll 7 + t_1, \\ p &= (((t_2 \ll 2) + x) \ll 3) - x. \end{aligned}$$

There are some other cost-function-based search methods such as simulated annealing. In [18], this technique was used to produce multiplication by a small set of constants. The same multiplier is used for a small set of different coefficients. This problem is different from ours.

In [19], a greedy algorithm is used to determine a solution with a low total operation cost. A 28 percent average area saving is achieved on some controllers and elliptic filters. This solution seems to be limited due to local attraction of the greedy algorithm.

3.4 Pattern Search Methods

Most of the pattern search methods are based on the same general idea. The algorithm recursively builds a set of constants to be optimized. This set is initialized with the recoded initial constants (generally using the CSD format). The different methods differ in the way they match the common subexpressions and the way they share and reuse them.

The multiple constant multiplication (MCM) solution presented in [5] performs a tree exploration with selection of matching parts of the SD representation of the constants. This paper is the most cited one and it presents a lot of details about the algorithm as well as about the comparisons.

In [20], the matches between constants are represented using a graph. The exploration and some transformations of this graph are used to produce a specific form of FIR filters with a reduced number of adders/subtractors while controlling the operator delay.

A solution based on an algebraic formulation of the possible matches between constants is presented in [21]. Unfortunately, the authors use random filters for their tests without specifying the coefficients. So, it is difficult to compare their results to other solutions.

A recent work [22] proposes sharing digits in the CSD representation of the coefficient matrix both in a horizontal and in a vertical way. This solution allows circuits to be designed with 10 percent fewer adders/subtracters than the straightforward CSD horizontal subexpression factorization.

In [23], a pattern search method is proposed. Some optimizations on the result architecture are done such as the transformation of multiple subtractions of the same value into the negation of this value and several adders. This kind of optimization can lead to significant improvement in ASICs where subtractors are larger than adders. This is not the case in our FPGAs.

In [24], a factorization method based on the selection of the best pair of matching digits is used. This solution can be easily extended to the selection of common parts of words larger than two digits.

We will base our solution on extensions and improvements of the algorithms presented in [25] and [26]. A detailed description of this idea is presented below. One can notice that, among all the abundant bibliography about the multiplication by constant problem, there is no general solution to the multiplication by constant matrix problem.

4 Proposed Algorithms

4.1 Lefèvre's Algorithm

In 2001, Lefèvre proposed a new algorithm to efficiently multiply a variable integer by a given set of integer constants [25]. As a special case, this algorithm was used to multiply a variable by a single constant.

4.1.1 Definitions

The principle of the algorithm is to handle a list of constants to be optimized and to find a "pattern" that appears several times in the set of constants. The constants are recoded using the CSD format in the very beginning. A pattern is a sequence of digits in $\{\bar{1}, 0, 1\}$. The number of nonzero digits in the pattern is called its weight.

A pattern P is said to occur in a constant C with a shift α when, for each 1 in position k of P, there is a 1 in position $k + \alpha$ in C and, for each $\bar{1}$ in position k of P, there is a $\bar{1}$ in position $k + \alpha$ in C. And, a pattern is said to occur negatively when there is a $\bar{1}$ in C for each 1 in P and a 1 in C for each $\bar{1}$ in P. This last point is one of the main differences between the two papers, [25] and [26]. Lefèvre's algorithm allows us to use patterns negatively, which leads to slightly better optimizations.

When two occurrences of the same pattern or of different patterns match the same nonzero digit of the constant, the two occurrences are said to conflict. For example, in the number $51 = 10\bar{1}010\bar{1}_2$, the pattern $10\bar{1}$ occurs positively with shift 0, negatively with shift 2, and positively with shift 4. The first and third occurrences both conflict with the second one. And, the pattern 10001 occurs negatively with shift 0 and positively with shift 2. Those occurrences overlap, but do not conflict. Moreover, every occurrence of the $10\bar{1}$ pattern conflicts with every occurrence of the 10001 pattern.

4.1.2 Description of the Algorithm

The principle of the algorithm can be described by the pseudocode presented in Algorithm 1.

Algorithm 1 : Principle of Lefèvre's algorithm.

While (there are some patterns with weight ≥ 2 and at least 2 non-conflicting occurrences)

- choose a pattern with maximal weight and 2 non-conflicting occurrences
- remove the chosen pattern from both occurrences
- add the pattern as a new constant

Then, multiplication by each constant in the final set can be implemented in the usual way: For each 1 ($\bar{1}$) in position p, add (subtract) x shifted by p bits to the left. And then, by rolling back the algorithm, each constant can be computed by shifts and additions/subtractions, with roughly one addition/subtraction for each chosen occurrence of a pattern.

On our previous example $p = c \times x$, Lefèvre's algorithm gives a solution with only four additions:

$$t_1 = (x \ll 3) - x,$$
$$t_2 = (t_1 \ll 2) - x,$$
$$p = (t_2 \ll 12) + (t_2 \ll 5) + t_1.$$

4.2 Extensions and Enhancements to Lefèvre's Algorithm

Mathematically speaking, Lefèvre's algorithm deals with the multiplication of a number by a constant vector. The first thing to do is to extend it for the multiplication of a vector by a constant matrix. This extension is rather straightforward: We simply replace each constant with a constant vector. Patterns are then replaced by vectors of patterns and shifts are performed componentwise. With this algorithm, it is possible to share all kinds of expressions.

For example, let us consider the computation of $y_1 = 5x_1 + 5x_2 + x_3$ and $y_2 = 5x_1 + 5x_2 + 4x_3$. The algorithm will first share the computation of $5x_1 + 5x_2$ between y_1 and y_2. After that, it will share $x_1 + x_2$ in $(5x_1 + 5x_2) = 4(x_1 + x_2) + (x_1 + x_2)$, effectively sharing the multiplication by 5 between x_1 and x_2. This example shows that the algorithm deals with both dimensions of the constant matrix.

A detailed description of our extended algorithm is given in the Appendix. This description, in C-like pseudocode, presents the overall behavior of our algorithm.

One point is kept unspecified in Lefèvre's algorithm: Which maximal pattern and which occurrences should we choose? In his original implementation, Lefèvre simply chose the first maximal pattern he found, with the first two occurrences. This solution is probably not the best, so we tried to find something better.

The first idea was to find all the maximal-weight patterns with at least two nonconflicting occurrences and all their occurrences. And then, we try to choose a set of patterns and, for each pattern, a set of at least two occurrences such that two chosen occurrences (of the same pattern or of different patterns) do not conflict. The choice is performed in order to maximize the gain in the weight of all the constants; with a constant with weight w and i occurrences, we gain $(i-1)(w-1)$ times its weight. As all the chosen patterns have the same maximal weight, we want to maximize the sum, for each pattern, of the number of occurrences diminished by one.

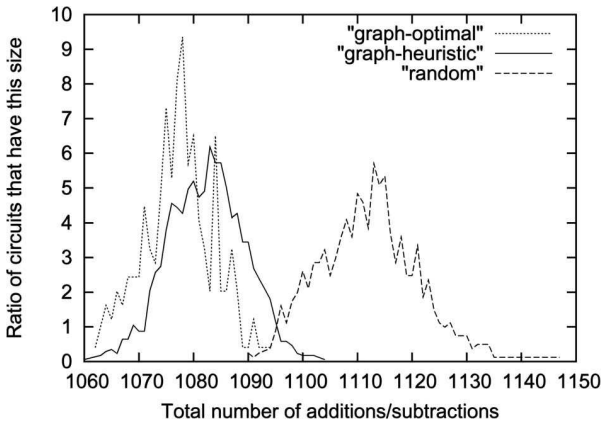


Fig. 1. Heuristics influence distributions (from many optimizations of one large 2D IDCT operator).

We tried three different solutions for this. The first one, called "random," and which is the closest to the original algorithm, is to recursively choose, at random, a pattern with two nonconflicting occurrences and to remove everything that conflicts with these occurrences. The second one, called "graph-heuristic," is to recursively choose a pattern with a maximal set of nonconflicting occurrences and a minimal set of conflicts with the other patterns and then remove everything that conflicts with these occurrences. And the third one, called "graph-optimal," is to build all the maximal sets of patterns and nonconflicting occurrences and to choose the best one. This last solution can be very computationally intensive.

We tried to compare those three solutions, by running them several times for the same constant matrix: a huge standard 8×8 points 2D IDCT (inverse DCT) operator with 14-bit words. The results in Fig. 1 show that the "graph-optimal" and "graph-heuristic" are roughly equivalent and better than the "random," with a tiny advantage to "graph-optimal." The time required to generate these results is less than one minute for "graph-heuristic" and "random," while it can grow to hours for "graph-optimal." Hence, we generally choose the "graph-heuristic" solution so we can perform lots of tries (thanks to its speed) and then choose the best solution. Similar results have been obtained using other applications.

4.3 Beyond the Mathematical Optimization

The improvements described above only deal with the minimization of the total number of additions and subtractions. Translated to hardware, this is not enough. Some additions and subtractions can be reordered without changing their total number thanks to properties such as associativity and commutativity.

First of all, one may want to have a small circuit. When three numbers a, b, and c are added, the order in which they are added influences the size of the adders. For example, if a and b are narrow numbers, while c is wide, computing $(a+b)+c$ leads to a smaller circuit than $(a+c)+b$ or

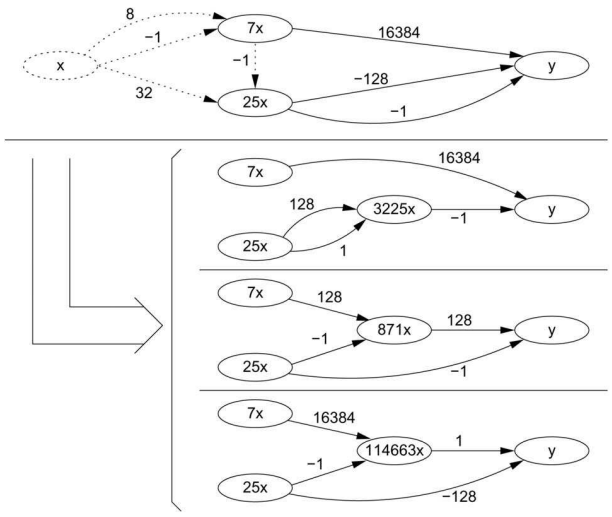


Fig. 2. Postoptimization of the multiplication by 111463 for area target.

$(b + c) + a$. Hence, for space optimization, it is generally better to add the narrowest numbers first.

On the other hand, one may want to have a fast circuit. The order in which three numbers are added also influences the worst-case delay of the circuit. For example, if a and c are available early while b is available late, the result is available earlier if we compute $(a + c) + b$ than $(a + b) + c$ or $(b + c) + a$. Hence, for speed optimization, it is preferable to add the earliest available values first.

Those optimizations are performed in two steps. At first, we unconstrain the order of the operations as much as possible. When a value is used only once, its computation is merged into the value that uses it. For example, if we had $t_1 = (x \ll 2) - x$ and $t_2 = (x \ll 6) - (t1 \ll 2) + x$, it becomes $t_2 = (x \ll 6) - (x \ll 4) + (x \ll 2) + x$. Similarly, if a term is composed of only one term, its computation is also merged into any value that uses it. This part is done by a traversal of the dataflow graph that represents the computation.

After removing all those useless constraints, we want to set new constraints that meet our goal of high speed or low area. In the data-flow graph, this is done by splitting nodes with more than two terms. We do this with a hierarchical traversal of the graph: A node is only considered after all its ancestors. This is possible because our dataflow graphs are acyclic. When we meet a node with more that two terms during the traversal, we extract two of its terms to make a new node, as illustrated in Fig. 2. We consider all the pairs of terms of the node. Each such pair can be assembled to a new intermediate value. It is possible to symbolically compute each corresponding value and deduce how wide the corresponding adder would be. And, as we are using a hierarchical traversal, we can also compute when each of those values would be available.

If we want to optimize for area, we select a pair from among those that require the smaller adder. On the other hand, if we want to optimize for speed, we select it from among those that would be available the earliest. Then, the corresponding new node is generated and replaces the two former terms: Now, there is one fewer term. We iterate that extraction of two terms until the considered node has only two terms.

Back to our example in Fig. 2, there are three possibilities that use $3225x$, $871x$, or $114663x$ as new intermediate values. Obviously, computing $871x$ requires a smaller adder than the other two; that solution would be the chosen one for area optimization. About speed, all three intermediate values would be available after three adder steps (from the input x), so they are equivalent. This simple example shows that the postoptimizations lead to significant improvements. In Section 6, a larger example (based on an IDCT operator) confirms these improvements using postoptimizations (see Table 5).

When we try to optimize for area, if several possibilities are equivalent, we choose among them with the speed criterion, so the circuit is not uselessly slow. The opposite is, of course, true as well.

Moreover, the algorithmic optimization is not enough. We need to generate some real circuits. Hence, we decided to generate some VHDL code. Although it may work for any target, our VHDL code generator is currently optimized for Xilinx FPGAs. So, additions and subtractions are performed using the dedicated fast carry-propagate adders and subtractors. The generator is able to produce VHDL code for fully parallel circuits or for digit-serial circuits with radix 2^n for any n. Only parallel architectures are available when delays are involved (e.g., filters).

5 Implementation

Our implementation is mainly in two parts. The first part performs the mathematical optimization, with our extended and enhanced version of Lefèvre's algorithm. This part was written in C++ and is approximately 1,500 lines long. This part is not a program by itself, but a collection of simple classes that can be easily interfaced with any C++ program. Hence, it would be easy, for example, to interface this with a program that computes coefficients for FIR filters. Then, the user would simply choose the type of filter and the required frequencies and attenuations and the program would compute the coefficients and generate some efficient VHDL code for it.

After the mathematical optimization, everything is implemented as plug-ins. Hence, there are, for example, plug-ins that optimize the order of the additions and subtractions or plug-ins that generate the output VHDL code. This structure with plug-ins makes the whole thing very modular. Hence, if someone wants, for example, to generate some Verilog code or some assembly language code for a DSP, it is sufficient to write a new output plug-in. Then, if someone wants to get pipelined circuits, a new pipelining plug-in can be written and it can then be used with any output plug-in. Those plug-ins are also written in C++. The collection of plug-ins is currently approximately 2,500 lines long.

The plug-ins communicate between themselves and with the main program with simple interfaces that describe the circuit as a data-flow graph. In this representation, vertices represent mathematical values. Hence, there are vertices for input values, for output values, and also for intermediate

```
#include "decompose.h"
#include <iostream>

int main(int argc, char **argv) {

  int value[] = {111463};

  Expr::initialize(argc, argv);

  CombinationSet<int> work(1);
  work.addLine(value);

  Key<int> *k;
  k = work.findBestPattern();
  while ((k!=NULL)&&(k->weight()>1)) {
    work.applyKey(*k, &std::cout);
    delete k;
    k = work.findBestPattern();
  }
  work.finish();
  return 0;
}
```

Fig. 3. Multiplication by 111463 optimization source code.

values. Then, there is an edge, from vertex x to vertex y, tagged with $(shift, sign, delay)$, if x shifted $shift$ bits to the left and a delay of $delay$ clock cycles is a positive or negative part (according to $sign$) of y. The $delay$ part is used for filters or pipelined circuits. This representation has the quality of being independent of the desired output.

Let us give a simple example of how this can be used and what is generated. As a simple example, we will consider building a constant multiplier by 111463. The corresponding source code and generated VHDL are presented in Fig. 3 and Fig. 4, respectively.

6 Results and Comparisons

The syntheses of this section have been performed using Xilinx ISE XST 4.2.03i tools for a Virtex XCV200-5 FPGA.

```
library IEEE;
  use IEEE.STD_LOGIC_1164.all;
  use IEEE.STD_LOGIC_arith.all;

entity MULT_111463 is
  port (
    x1  : in  std_logic_vector (7 downto 0);
    y1  : out std_logic_vector (24 downto 0)
  );
end MULT_111463;

architecture struct of MULT_111463 is
  signal t3     : std_logic_vector (22 downto 0);
  signal t2     : std_logic_vector (15 downto 0);
  signal t1     : std_logic_vector (22 downto 0);
begin -- struct
-- t3    = 20641*x1
  t3 <= ( t2(15)&t2(15)&t2(15 downto 0) & "00000" )
       +( t1(22 downto 0) );
-- t2    = 129*x1
  t2 <= ( x1(7)&x1(7)&x1(7)&x1(7)&x1(7)&x1(7)&x1(7)
          &x1(7)&x1(7 downto 0) )
       +( x1(7)&x1(7 downto 0) & "0000000" );
-- t1    = 16513*x1
  t1 <= ( t2(15)&t2(15)&t2(15)&t2(15)&t2(15)&t2(15)
          &t2(15)&t2(15 downto 0) )
       +( x1(7)&x1(7 downto 0) & "00000000000000" );
-- y1    = 111463*x1
  y1 <= ( t1(21 downto 0) & "000" )
       -( t3(22)&t3(22)&t3(22 downto 0) );
end struct;
-- Number of additions: 4
-- Estimated number of FA cells:  61
```

Fig. 4. Multiplication by 111463 generated VHDL.

TABLE 1
Number of Additions/Subtractions Comparison for Some 1D 8-Point DCT Operators

operator	initial	[5]	[24]	our
DCT 8b	300	94	73	56
DCT 12b	368	100	84	70
DCT 16b	521	129	114	89
DCT 24b	789	212	—	119

TABLE 2
Synthesis Results of Some 1D 8-Point DCT Generated Operators

operator	synthesis		generator	
	# slices	delay [ns]	# ±	# FA
DCT 8b	401 (17%)	19.5	56	739
DCT 12b	647 (27%)	21.7	70	1202
DCT 16b	1085 (46%)	25.7	89	2009
DCT 24b	2106 (89%)	27.9	119	3934

The operators are not pipelined. The area is measured in number of slices (two LUTs with 4 address bits per slice in Virtex devices). The required area compared to the 2,352 available slices in an XVC200 device is also reported in parentheses. The delay is expressed in nanoseconds. The number of additions/subtractions and the number of FA cells are computed by our generator (see the two last lines of Fig. 4); the number of FA cells is only an estimation (assuming the use of carry ripple adders).

Only a few papers give enough elements to compare to our solutions. In [5] and [24], there are useful values for the DCT application. Table 1 presents the number of additions/subtractions for the 1D 8-point DCT for several word sizes. Our generator improves the previous results from 17 percent to 44 percent. Table 2 gives the synthesis results for the corresponding generated operators.

We performed some other comparisons on some error-correcting codes from [5] and [24]: the 8×8 Hadamard matrix transform, $(16, 11)$ Reed-Muller, $(15, 7)$ BCH, and $(24, 12, 8)$ Golay codes. The comparison with the previous works in [5] and [24] is presented in Table 3 and the corresponding synthesis results are presented in Table 4. These results show that, for very simple operators such as a small BCH code, some improvements are still possible. In the case of the 8×8 Hadamard matrix transform, we obtained the same results as in the previous work [5].

Table 5 presents the synthesis results of the same IDCT operator with the three possible postoptimizations of our generator: none, area, or speed. The operator is a 1D 8-point IDCT for 14-bit constants and 8-bit inputs. From the same initial additions/subtractions number, the optimizations presented in Section 4.3 lead to significant improvements, 40 percent for the speed optimization for instance. The generation time for all these operators is around a few seconds on a standard desktop computer.

In Section 4.3, we explained that our generator can produce digit-parallel as well as digit-serial circuits using different output plug-ins. Table 6 presents the synthesis results of a 1D 8-point IDCT operator for several solutions: digit-parallel and radix-2, 4, 8, 16, 64, and 256 digit-serial versions. Digit-serial implementations lead to small area

TABLE 3
Number of Additions/Subtractions Comparison for Some Error-Correction Benchmarks

operator	initial	[5]	[24]	our
8×8 Had.	56	24	—	24
$(16, 11)$ R.-M.	61	43	31	31
$(15, 7)$ BCH	72	48	47	44
$(24, 12, 8)$ Golay	76	—	47	45

TABLE 4
Synthesis Results for Some Error-Correction Benchmarks

operator	synthesis		generator	
	# slices	delay [ns]	# $\pm$	# FA
8×8 Had.	128 (5%)	11.9	24	240
$(16, 11)$ R.-M.	39 (1%)	12.1	31	86
$(15, 7)$ BCH	461 (19%)	18.2	44	861
$(24, 12, 8)$ Golay	63 (2%)	12.2	45	136

and short cycle time operators. But, in order to fairly compare digit-serial versus digit-parallel solution, we should compare with the pipelined parallel operator.

In [27], an algorithm for designing multiplication by matrix operators is presented. The proposed algorithm has been tested on $n \times n$ matrices with random 8-bit integer coefficients. In Fig. 5, we compare our generator with the results from [27] (only values for n between 2 and 6 are reported in [27]). Our complete results on random matrices with 8-bit integer coefficients are reported in Table 7. The average number of additions and subtractions and its standard deviation have been evaluated on 100 random matrices for each size. The reported generation time is the average value for the generation time of one matrix. Our results show slightly better performances.

7 Extension to Digital Filters

Digital filters are a very specific case of multiplication by a constant matrix. They are linear combinations of the input, delayed several times:

$$y[t] = \sum_{i=0}^{n} a_i x[t-i],$$

where $x[i]$ is the ith value of the sampled signal x.

Such filters are generally implemented using one of two different kinds of architectures. The first one delays the input to compute all the $x[t-i]$ and then computes their linear combination (the multiplication by the constant matrix). The second one computes all the $a_i x[t-i]$ and then delays them and adds them to form the result as depicted in Fig. 6. In signal processing, the first form of the filter is called the *direct form*, while the second one is called the *transposed form*. We call the gray part of Fig. 6 the multiplication block (MB).

TABLE 5
Influence of the Generator Optimizations on a 1D 8-Point IDCT Operator

operator	synthesis		generator	
	# slices	delay [ns]	# $\pm$	# FA
IDCT	769 (32%)	30.2	81	1382
IDCT area	665 (28%)	19.8	81	1196
IDCT speed	666 (28%)	18.0	81	1241

TABLE 6
Synthesis Results for 1D 8-Point Digit-Parallel and Digit-Serial IDCT Operators

operator	# slices [ns]	delay
parallel	614	40
serial radix-2	85	22
serial radix-4	153	36
serial radix-8	194	46
serial radix-16	242	47
serial radix-64	349	47
serial radix-256	446	48

These implementation solutions consider the computations to be independent and do not allow sharing results between consecutive computations. We extended our algorithm to be able to apply such optimizations. As an example, let us consider the following trivial low-pass FIR filter:

$$y[t] = x[t] + 5x[t-1] + 5x[t-2] + x[t-3].$$

The direct form of the filter leads to three delay units to compute the $x[t-i]$ and then five additions to compute $y[t]$ (Fig. 7A). This can be reduced to three delay units and four additions by using the symmetry of the coefficients (Fig. 7B). The transposed form leads to one addition to compute the values $x[t]$ and $5x[t]$ and then three delay units and three additions to compute $y[t]$, which gives a total of three delay units and four additions (Fig. 7C).

If we allow sharing of intermediate results between computations using our generator, we can first compute $z[t] = x[t] + x[t-1]$, which requires one delay unit and one addition, and then compute $y[t] = z[t] + 4z[t-1] + z[t-2]$, which requires two delay units and two additions; this gives a total of three delay units and three additions (Fig. 7D). It is, of course, equivalent to first computing $z'[t] = x[t] + 4x[t-1] + x[t-2]$ and then $y[t] = z'[t] + z'[t-1]$; this is the architecture found by our generator (Fig. 7E). An extract of the generated VHDL code corresponding to this last architecture is shown in Fig. 8.

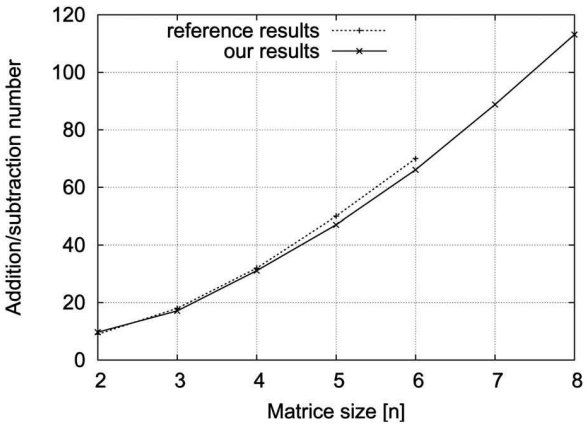


Fig. 5. Comparison of the results from [27] with ours on $n \times n$ random matrices with 8-bit integer coefficients.

TABLE 7
Results for $n \times n$ Random Matrices with 8-Bit Integer Coefficients

n	average # $\pm$	standard deviation	generation time [s]
2	9.7	1.3	.04
3	17.1	0.9	.07
4	31.2	2.2	.14
5	47.1	3.3	.37
6	66.1	4.0	.80
7	88.9	5.3	1.54
8	113.2	6.7	2.69
9	141.6	7.0	4.55
10	172.4	8.5	7.28
11	207.1	10.8	10.62
12	241.6	11.9	16.21
13	279.6	13.3	23.86
14	322.9	17.0	32.59
15	370.0	20.0	44.94
16	412.4	19.4	57.30

Of course, this trivial example only shows that it may be possible to reduce the computational cost of an FIR filter by sharing some results between consecutive computations. It is not supposed to establish a rule about how efficient it is; this will be shown by implementing some real FIR filters.

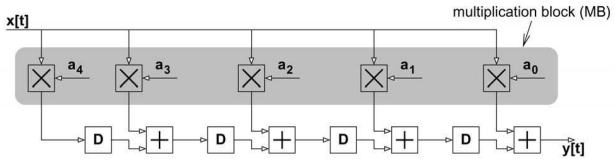


Fig. 6. The transposed form of an FIR filter.

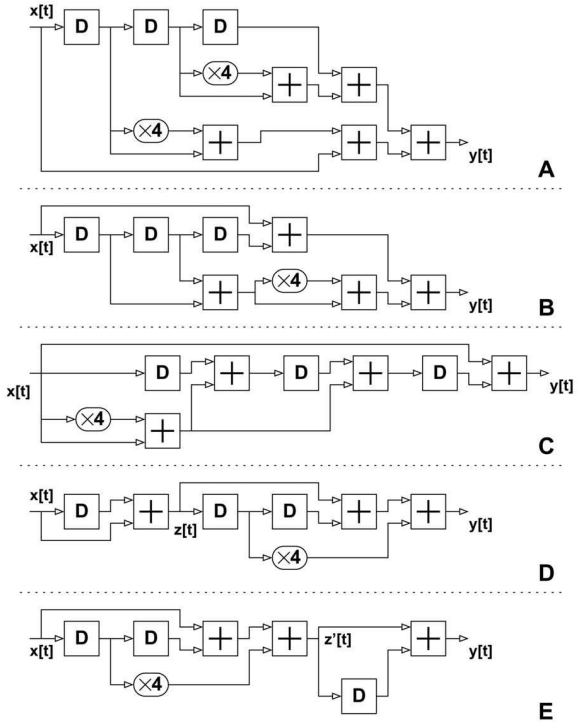


Fig. 7. FIR (1, 5, 5, 1) architectures.

```
...
architecture struct of fir1551 is
...
begin
  x1_d(0) <= x1;
  dl1: for k in 0 to 1 generate
    r1: reg generic map (n=>8)
            port map (Ck=>clk,i=>x1_d(k),o=>x1_d(k+1));
  end generate;
  t2_d(0) <= t2;
  dl2: for k in 0 to 0 generate
    r2: reg generic map (n=>11)
            port map (Ck=>clk,i=>t2_d(k),o=>t2_d(k+1));
  end generate;
-- t1     = x1+x1[2]
  t1     <=  SXT(x1_d(2) , 9) + SXT(x1 , 9);
-- t2     = x1+4*x1[1]+x1[2]
  t2     <=  SXT(x1_d(1) & "00" , 11) + SXT(t1 , 11);
-- y1     = x1+5*x1[1]+5*x1[2]+x1[3]
  y1     <=  SXT(t2 , 12) + SXT(t2_d(1) , 12);
end struct;
```

Fig. 8. FIR (1, 5, 5, 1) generated VHDL (extract).

For this new extension, we use the Z transform of the FIR filter, which is very common in digital signal processing: FIR filters are represented by polynomials in Z^{-1}. For example, our (1, 5, 5, 1) FIR filter is represented by the polynomial $1 + 5Z^{-1} + 5Z^{-2} + Z^{-3}$. In those polynomials, multiplying a signal by Z^{-1} means delaying that signal by one delay unit. Such a polynomial represents one single FIR filter. But, as our algorithm is already able to deal with several inputs and several outputs, we thought it would be useful to be able to deal with filters with several inputs and outputs. Such complex filters may be used, for example, to implement a digital audio equalizer or a digital DTMF (dual-tone-multi-frequency) decoder.

Therefore, an extension works by replacing the matrix of constants by a matrix of polynomials in Z^{-1}. Then, if we perform the optimization exactly as before, no pattern is shared between subsequent computations; this corresponds to the direct form of the filter. To implement such sharings, we must allow multiplication of the patterns by Z^{-1}, just as we allowed shifting them to the left. By doing so, the introduced delays are not taken into account for the optimization and only the number of additions/subtractions is optimized.

This generally results in a huge increase of the number of registers, with little to no gain to the number of additions/subtractions. This may be acceptable when programming some DSP processors, but it is not for hardware implementations. To prevent that huge increase, it is possible to set a limit to the number of multiplication of a pattern by Z^{-1}. Thus, it is possible to control the number of added registers.

TABLE 8
Specifications of the Example Filters Presented in [28] and Matlab Command Used to Generate the Coefficients (Attenuation and Ripple Values Are Theoretical Values)

filter	# tap	normalized frequencies pass-band	normalized frequencies stop-band	stop-band attenuation	pass-band ripple
example 1	25	0.15	0.25	46.0 dB	0.05 dB
	remez(24, [0 0.3 0.5 1], [1 1 0 0])				
example 2	59	0.021	0.07	61.7 dB	0.2 dB
	remez(59, [0 0.042 0.14 1], [1 1 0 0], [1 14])				

TABLE 9
Comparison of the Implementation of Low-Pass FIR Filters from [28]

Filter	coefficients # bits		# ±				implemented filters	
			[28] results		our results		stop-band	pass-band
	total	non-zero	MB	total	MB	total	attenuation	ripple
[28] example 1	9	2 (3)	11	36	6	30	43.8 dB	0.05 dB
[28] example 2	14	3 (4)	57	116	33	89	60.5 dB	0.2 dB

In [28], the optimization of low-pass FIR filters using sum of power of two (SOPOT) coefficients is presented. The method is demonstrated on two filters. The specifications of these two filters are reported in Table 8. This table also report the `remez` Matlab function call used to generate the theoretical coefficients of the filters.

In Table 9, we compare the implementation results from [28] with our method. For our generator input we use the optimized SOPOT coefficients presented in [28] in order to achieve the same stop-band attenuation and pass-band ripple values. On the first example from [28], nine digits ({-1, 0, 1}) SOPOT coefficients are used with at most two nonzero digits expect for large values where three digits are allowed. In the second example, 14 bits SOPOT coefficients are used with at most three (or four) nonzero digits. In Table 9, two values are reported for the number of addition/subtraction: *total* for the whole filter and *MB* only for the multiplication block of the transposed form (see Fig. 6).

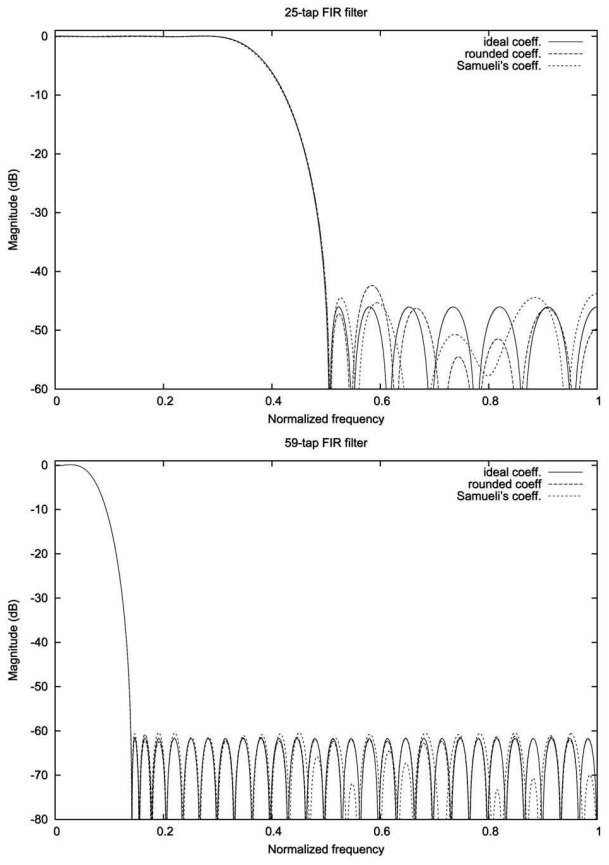


Fig. 9. Frequency response of the filters compared with [28].

The normalized frequency response of the two filters (theoretical, rounded, and generated filters) are reported in Fig. 9.

We also implemented in FPGA some low-pass FIR filters with specifications derived from [20]. The corresponding results are presented in Table 11. The specifications of those filters are presented in Table 10. The coefficients have been generated using the `remez` Matlab functions $c_1 = \texttt{remez}(\#\text{tap}, [0 f_p f_s 1], [1100])$ and $c_2 = \texttt{round}((2 \wedge \text{width}) * c_1)$. The values reported in Table 10 represent the complete filter, while the number of adders reported in [20] only represent the multiplication block, an additional adder should added for each tap of the filter.

For each filter from [20], we tried to implement it with a delay limit (denoted by DL in the result tables) set to 0 (no sharing between consecutive calculations), 1, 2, or ∞ and the resulting VHDL code was optimized for speed using Xilinx ISE XST 5.2.03i tools for a Virtex-II 1000 FPGA (XC2V1000-5) on 1.7 GHz Pentium4 PC with 1GB RAM. The operators are not pipelined. The required area, compared to the 5,120 available slices in a XC2V1000 device, is reported in parentheses. We also report the delay of the operator and

TABLE 10
Low-Pass FIR Filters Specifications from [20]

filter	f_p	f_s	#tap	width
FIR 1	0.15	0.25	60	14
FIR 2	0.15	0.20	60	16
FIR 3	0.10	0.15	60	14
FIR 4	0.10	0.12	100	18

TABLE 11
FPGA Synthesis Results of the Low-Pass FIR Filters from [20]

filter	DL	# slices	delay [ns]	T_{synth} [s]
FIR 1	0	1364 (26%)	37.2	51
	1	1377 (26%)	36.2	57
	2	1392 (27%)	35.2	57
	∞	2033 (39%)	33.5	84
FIR 2	0	1655 (32%)	38.4	60
	1	1762 (34%)	37.1	67
	2	1911 (37%)	35.8	76
	∞	4141 (80%)	36.5	215
FIR 3	0	1293 (25%)	34.8	49
	1	1420 (28%)	32.3	60
	2	1503 (30%)	32.1	77
	∞	2238 (43%)	32.9	105
FIR 4	0	3126 (61%)	41.7	114
	1	3247 (63%)	47.7	129
	2	3486 (68%)	41.8	145
	∞	(>250%)	—	—

TABLE 12
Parks-McClellan Filter Coefficients Specification

coefficient	value
$h(0)$	-0.00933078669
$h(1)$	0.07628237421
$h(2)$	0.03135623682
$h(3)$	0.01374432164
$h(4)$	-0.00948598843
$h(5)$	-0.03358586396
$h(6)$	-0.04680063247
$h(7)$	-0.03819695824
$h(8)$	-0.00271831937
$h(9)$	0.05563093697
$h(10)$	0.12405515375
$h(11)$	0.18473033065
$h(12)$	0.22024453765

TABLE 13
Number of Additions/Subtractions Comparison for the Parks-McClellan Filter

operator	initial	[22]	our
8 bits	35	32	24
16 bits	72	70	46

its total synthesis time (including place and route optimizations) using high effort constraints.

In Table 11, one can see that the operator period is generally reduced when increasing the delay limit (DL). This effect is due to the additional registers that break long paths in the circuit. This permits reorganization of the schedule with a shorter critical path. But, when the number of additional registers is too large, new long lines of routing are involved in the circuit. This explains the larger delay that sometimes occurs for large values of DL.

We did another experiment on an FIR filter from [22], implemented both with 8 and 16 bits of accuracy. This filter is based on the Parks-McClellan design of a low-pass 26-tap FIR filter with pass-band and stop-band edges at 0.2 and 0.25, respectively. The corresponding coefficients are presented in Table 12 (only the first 13 coefficients are reported because of the symmetry); those coefficients can be computed using Matlab with the command `remez`$(25, [00.20.251], [1100])$.

TABLE 14
Parks-McClellan Filters Synthesis Results

operator	DL	# slices	delay [ns]	T_{synth} [s]
8 bits	0	297 (6%)	27.0	22
	1	306 (6%)	27.7	23
	2	275 (5%)	22.4	22
	∞	415 (8%)	22.4	25
16 bits	0	711 (13%)	34.4	33
	1	747 (14%)	31.2	34
	2	755 (15%)	33.3	35
	∞	1569 (31%)	33.5	66

The comparison of our results with those from [22] is presented in Table 13. Our solution leads to a reduction of the operation count of about 25 percent for 8-bit coefficients and 34 percent for 16-bit coefficients. The results of the FPGA implementation of generated architectures are presented in Table 14.

Even with no sharing between consecutive computations, our algorithm already gives a very small number of operations (less than two additions/subtractions per tap). This gives little room for improvement. Hence, when the delay limit rises, the number of operations does not shrink much, while many registers are added to share intermediate results. This explains why the size rises with the delay limit. On the other hand, the delay is generally reduced, around 9 percent on average and up to 17 percent. This proves that this sharing is still useful when speed is a main concern.

8 Conclusion

A new algorithm for the problem of multiplication by constant was presented. We generalized the previous results by dealing with the problem of the optimization of multiplication of one vector by one constant matrix. Our algorithm is based on extensions and enhancements of previous algorithms from [25] and [26]. Compared to the best previous results, our solution leads to a significant drop in the total number of additions/subtractions, up to 40 percent.

```
procedure optimize(args: constants):
  constants := [csd(constants)]
  names := ["y"]
  share := search_best_shares(constants)
  while share != FAILURE
    (pattern,occurrences) := share
    pattern_name := new_name()
    for occ in occurrences
      (index,shift,sign) := occ
      constants[index] := constants[index] - sign * pattern << shift
      display(names[index]," := ",names[index],sign,"(",pattern_name," << ",shift,")")
    constants := constants + [pattern]
    names := names + [pattern_name]
    share := search_best_shares(constants)
  # all optimizations are done
  # ''flush'' the last constants
  for index in [1..size(constants)]
    c := constants[index]
    for rank in [0..size(c)]
      if c[rank] = 1
        display(names[index]," := ",names[index]," + (x << ",rank,")")
      else if c[rank] = -1
        display(names[index]," := ",names[index]," - (x << ",rank,")")
    display(names[index]," := 0")
```

Fig. 10. Pseudocode of procedure `optimize`.

```
function search_best_shares(lines):
  best_gain := 2
  best_patterns := {}
  for (index1,index2) in [1..size(lines)]^2
      such that index1 <= index2
    l1 := lines[index1]
    l2 := lines[index2]
    for (delay1,delay2) in [(0,0)..(+infinity,0)] union [(0,0)..(0,+infinity)]
        such that l1 div (Z-1)^delay1 != 0
             and l2 div (Z-1)^delay2 != 0
             and (index1 != index2 or delay1 <= delay2)
      l11 := l1 div (Z-1)^delay1
      l12 := l2 div (Z-1)^delay2
      for (shift1,shift2) in [(0,0)..(+infinity,0)] union [(0,0)..(0,+infinity)]
          such that l1 >> shift1 != 0
               and l2 >> shift2 != 0
               and (index1 != index2 or delay1 != delay2 or shift1 < shift2)
        line1 := l11 >> shift1
        line2 := l12 >> shift2
        for sign in {+,-}
          pattern := line1 & sign * line2
          share := (pattern,[(index1,delay1,shift1,+),(index2,delay2,shift2,sign)])
          if estimated_gain(share) >= best_gain and index1 = index2
            # eliminate internal conflicts
            true_patterns := eliminate_internal_conflicts(pattern,sign,shift1,
                                                          shift2,delay1,delay2)
          else
            # there is no internal conflict : this is a true pattern
            true_patterns := {pattern}
          for word in true_patterns
            share := (word,[(index1,delay1,shift1,+),(index2,delay2,shift2,sign)])
            gain := estimated_gain(share)
            if gain >= best_gain
              if gain > best_gain
                best_shares := {}
                best_gain := gain
              # Normalization of the pattern
              share := normalize(share)
              best_shares := best_shares union {share}
  return best_shares
```

Fig. 11. Pseudocode of function `search_best_shares`.

```
function eliminate_internal_conflicts(args: pattern,sign,shift1,shift2):
  conflicts := pattern << shift1 & sign * pattern << shift2
  while conflicts != 0
    pattern := pattern - least_significant_digit(conflicts) >> shift1
    conflicts := pattern << shift1 & sign * pattern << shift2
  return pattern
```

Fig. 12. Pseudocode of function `eliminate_internal_conflicts`.

We implemented this algorithm in a VHDL generator. Based on a simple mathematical description of the computation, the generator produces an optimized VHDL code for Xilinx FPGAs. At the moment, the generated operators are nonpipelined parallel or digit-serial ones. We will extend our generator to produce pipelined circuits to reach higher clock frequencies.

We also extended our algorithm and generator to the case of some digital filters. We are now able to handle filters involving a multiplication by constant matrix and delay operations (such as FIR filters). In the case of a 26-tap 16-bit FIR filter, a 34 percent reduction of the operation count is achieved, compared to recent results from [22]. These first results on filter optimization are promising; we now plan to work on the synthesis of filters in the near future.

We want to extend our algorithm and generator to standard-cell-based ASICs. The way to implement the adders/subtracters would widely impact the performance of the complete operator. The optimization required for low-power consumption may also change our solutions.

Another area to explore in the future is the use of lossy representations, such as [11]. In a lot of applications, the models include some approximations and the quantization of the coefficients. It may be a good idea to allow small perturbations of the coefficients.

APPENDIX

DETAILED ALGORITHMS

Figs. 10, 11, and 12 are C-like pseudocode versions of our extended algorithm presented in Section 4. Procedure `optimize`, Fig. 10, is the main entry point.

ACKNOWLEDGMENTS

The authors would like to thank the "*Ministère Français de la Recherche*" (grant # 1048 CDR 1 "*ACI jeunes chercheurs*") and the Xilinx University Program for their support. They also want to thank the anonymous reviewers. Their comments and corrections were very useful in improving the paper.

REFERENCES

[1] D.J. Magenheimer, L. Peters, K.W. Pettis, and D. Zuras, "Integer Multiplication and Division on the HP Precision Architecture," *IEEE Trans. Computers*, vol. 37, no. 8, pp. 980-990, Aug. 1988.

[2] A.D. Booth, "A Signed Binary Multiplication Technique," *Quarterly J. Mechanical Applications of Math.*, vol. IV, no. 2, pp. 236-240, 1951.
[3] R. Bernstein, "Multiplication by Integer Constants," *Software—Practice and Experience*, vol. 16, no. 7, pp. 641-652, July 1986.
[4] N. Boullis and A. Tisserand, "Some Optimizations of Hardware Multiplication by Constant Matrices," *Proc. 16th IEEE Symp. Computer Arithmetic (ARITH 16)*, J. -C. Bajard and M. Schulte, eds., pp. 20-27, June 2003.
[5] M. Potkonjak, M.B. Srivastava, and A.P. Chandrakasan, "Multiple Constant Multiplications: Efficient and Versatile Framework and Algorithms for Exploring Common Subexpression Elimination," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 2, pp. 151-165, Feb. 1996.
[6] M.D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2003.
[7] M.J. Flynn and S.F. Oberman, *Advanced Computer Arithmetic Design*. Wiley-Interscience, 2001.
[8] R.I. Hartley, "Subexpression Sharing in Filters Using Canonic Signed Digit Multipliers," *IEEE Trans. Circuits and Systems II: Analog and Digital Signal Processing*, vol. 43, no. 10, pp. 677-688, Oct. 1996.
[9] K.D. Chapman, "Fast Integer Multipliers Fit in FPGAs," *EDN Magazine*, May 1994.
[10] S. Yu and E.E. Swartzlander, "DCT Implementation with Distributed Arithmetic," *IEEE Trans. Computers*, vol. 50, no. 9, pp. 985-991, Sept. 2001.
[11] P. Boonyanant and S. Tantaratana, "FIR Filters with Punctured Radix-8 Symmetric Coefficients: Design and Multiplier-Free Realizations," *Circuits Systems Signal Processing*, vol. 21, no. 4, pp. 345-367, 2002.
[12] C.K.S. Pun, S.C. Chan, K.S. Yeung, and K.L. Ho, "On the Design and Implementation of FIR and IIR Digital Filters with Variable Frequency Characteristics," *IEEE Trans. Circuits and Systems II: Analog and Digital Signal Processing*, vol. 49, no. 11, pp. 689-703, Nov. 2002.
[13] S.C. Chan and W.L.K.L. Ho, "Multiplierless Perfect Reconstruction Modulated Filter Banks with Sum-of-Powers-of-Two Coefficients," *Signal Processing Letters, IEE*, vol. 8, no. 6, pp. 163-166, 2001.
[14] V.S. Dimitrov, G.A. Jullien, and W.C. Miller, "Theory and Applications of the Double-Base Number System," *IEEE Trans. Computers*, vol. 48, no. 10, pp. 1098-1106, Oct. 1999.
[15] J. Li and S. Tantaratana, "Multiplier-Free Realizations for FIR Multirate Converters Based on Mixed-Radix Number Representation," *IEEE Trans. Signal Processing*, vol. 45, no. 4, pp. 880-890, Apr. 1997.
[16] N. Homma, T. Aoki, and T. Higuchi, "Evolutionary Graph Generation System with Transmigration Capability and Its Application to Arithmetic Circuit Synthesis," *IEE Proc.*, vol. 149, no. 2, pp. 97-104, Apr. 2002.
[17] P. Briggs and T. Harvey, "Multiplication by Integer Constants," technical report, Rice Univ., 1994.
[18] M.F. Mellal and J.-M. Delosme, "Multiplier Optimization for Small Sets Of Coefficients," *Proc. Int'l Workshop Logic and Architecture Synthesis*, pp. 13-22, Dec. 1997.
[19] H.T. Nguyen and A. Chatterjee, "Number-Splitting with Shift-and-Add Decomposition for Power and Hardware Optimization in Linear DSP Synthesis," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 4, pp. 419-424, Aug. 2000.
[20] H.-J. Kang and I.-C. Park, "FIR Filter Synthesis Algorithms for Minimizing the Delay and the Number of Adders," *IEEE Trans. Circuits and Systems II: Analog and Digital Signal Processing*, vol. 48, no. 8, pp. 770-777, Aug. 2001.
[21] M. Martínez-Peiró, E.I. Boemo, and L. Wanhammar, "Design of High-Speed Multiplierless Filters Using a Nonrecursive Signed Common Subexpression Algorithm," *IEEE Trans. Circuits and Systems II: Analog and Digital Signal Processing*, vol. 49, no. 3, pp. 196-203, Mar. 2002.
[22] A. Vinod, E.-K. Lai, A. Premkumar, and C. Lau, "FIR Filter Implementation by Efficient Sharing of Horizontal and Vertical Common Subexpresions," *Electronics Letters*, vol. 39, no. 2, pp. 251-253, Jan. 2003.
[23] A. Yurdakul and G. Dündar, "Fast and Efficient Algorithm for the Multiplierless Realisation of Linear DSP Transforms," *IEE Proc. Circuits, Devices, and Systems*, vol. 149, no. 4, pp. 20-211, Aug. 2002.
[24] A. Matsuura, M. Yukishita, and A. Nagoya, "A Hierarchical Clustering Method for the Multiple Constant Multiplication Problem," *IEICE Trans. Fundamentals of Electronics, Comm., and Computer Sciences*, vol. E80-A, no. 10, pp. 1767-1773, Oct. 1997.
[25] V. Lefèvre, "Multiplication par une Constante," *Réseaux et Systèmes Répartis, Calculateurs Parallèles*, vol. 13, nos. 4-5, pp. 465-484, 2001.
[26] R. Paško, P. Schaumont, V. Derudder, S. Vernalde, and D. Ďuračková, "A New Algorithm for Elimination of Common Subexpressions," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 1, pp. 58-68, Jan. 1999.
[27] A.G. Dempster, O. Gustafsson, and J.O. Coleman, "Towards an Algorithm for Matrix Multiplier Blocks," *Proc. European Conf. Circuit Theory Design*, Sept. 2003.
[28] H. Samueli, "An Improved Search Algorithm for the Design of Multiplierless FIR Filters with Power-of-Two Coefficients," *IEEE Trans. Circuits and Systems*, vol. 36, no. 7, pp. 1044-1047, July 1989.



Nicolas Boullis was a student at the École Normale Supérieure de Lyon, where he learned computer science. During his studies, he showed some interest in various research topics in computer science, such as multicast in active networks, designing some arithmetic blocs for FPGA, design of asynchronous dividers, and, more recently, hardware multiplication by constants, which is the topic of his PhD thesis (which he will be defending in mid-2005). He now works at the École Centrale de Paris as a system engineer.



Arnaud Tisserand received the MSc degree and the PhD degree in computer science from the École Normale Supérieure de Lyon, France, in 1994 and 1997, respectively. He is with the French National Institue for Research in Computer Science and Control (INRIA) and the Laboratoire de l'Informatique du Parallèlisme (LIP) in Lyon, France. He teaches computer architecture and VLSI design at the École Normale Supérieure de Lyon, France. His research interests include computer arithmetic, computer architecture, and VLSI design. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

A.5. Génération d'approximations polynomiales efficaces en machine

Reproduction de l'article [14] paru dans le journal *ACM Transactions on Mathematical Software* en juin 2006 et résumé en section 3.2.5.

auteurs	N. Brisebarre, J.-M. Muller et A. Tisserand
titre	*Computing Machine-Efficient Polynomial Approximations*
journal	*ACM Transactions on Mathematical Software*
date	juin 2006
volume	32
numéro	2
pages	236–256
numéro DOI	10.1145/1141885.1141890

Computing Machine-Efficient Polynomial Approximations

NICOLAS BRISEBARRE
Université J. Monnet, St-Étienne and LIP-E.N.S. Lyon
JEAN-MICHEL MULLER
CNRS, LIP-ENS Lyon
and
ARNAUD TISSERAND
INRIA, LIP-ENS Lyon

Polynomial approximations are almost always used when implementing functions on a computing system. In most cases, the polynomial that best approximates (for a given distance and in a given interval) a function has coefficients that are not exactly representable with a finite number of bits. And yet, the polynomial approximations that are actually implemented do have coefficients that are represented with a finite—and sometimes small—number of bits. This is due to the finiteness of the floating-point representations (for software implementations), and to the need to have small, hence fast and/or inexpensive, multipliers (for hardware implementations). We then have to consider polynomial approximations for which the degree-i coefficient has at most m_i fractional bits; in other words, it is a rational number with denominator 2^{m_i}. We provide a general and efficient method for finding the best polynomial approximation under this constraint. Moreover, our method also applies if some other constraints (such as requiring some coefficients to be equal to some predefined constants or minimizing relative error instead of absolute error) are required.

Categories and Subject Descriptors: G.1.0 [**Numerical Analysis**]: General—*Computer arithmetic*; G.1.2 [**Numerical Analysis**]: Approximation; B.2.4 [**Arithmetic and Logic Structures**]: High-Speed Arithmetic

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Polynomial approximation, minimax approximation, floating-point arithmetic, Chebyshev polynomials, polytopes, linear programming

Authors' addresses: N. Brisebarre, LArAl, Université J. Monnet, 23, rue du Dr P. Michelon, F-42023 Saint-Étienne Cedex, France and LIP/Arénaire (CNRS-ENS Lyon-INRIA-UCBL), ENS Lyon, 46 Allée d'Italie, F-69364 Lyon Cedex 07 France; email: Nicolas.Brisebarre@ens-lyon.fr; J.-M. Muller, LIP/Arénaire (CNRS-ENS Lyon-INRIA-UCBL), ENS Lyon, 46 Allée d'Italie, F-69364 Lyon Cedex 07 France; email: Jean-Michel.Muller@ens-lyon.fr; A. Tisserand, INRIA, LIP/Arénaire (CNRS-ENS Lyon-INRIA-UCBL), ENS Lyon, 46 Allée d'Italie, F-69364 Lyon Cedex 07 France; email: Arnaud.Tisserand@ens-lyon.fr.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2006 ACM 0098-3500/06/0600-0236 $5.00

ACM Transactions on Mathematical Software, Vol. 32, No. 2, June 2006, Pages 236–256.

1. INTRODUCTION

The basic floating-point operations that are implemented in hardware on modern processors are addition/subtraction, multiplication, and sometimes division and/or fused multiply-add, that is, an expression of the form $xy + z$, computed with one final rounding only. Moreover, division is frequently much slower than addition or multiplication (see Table I), and sometimes, for instance, on the Itanium, it is not hardwired at all. In such cases, it is implemented as a sequence of fused multiply and add operations, using an iterative algorithm.

Therefore, when trying to implement a given regular enough function f, it seems reasonable to try to avoid divisions and to use additions, subtractions, multiplications and possibly fused multiply-adds. Since the only functions of one variable that one can implement using a finite number of these operations and comparisons are piecewise polynomials, a natural choice is to focus on piecewise polynomial approximations to f. Indeed, most recent software-oriented elementary function algorithms use polynomial approximations [Markstein 2000; Muller 1997; Story and Tang 1999; Cornea et al. 2002].

Two kinds of polynomial approximations are used: the approximations that minimize the average error, called *least squares approximations*, and the approximations that minimize the worst-case error, called *least maximum approximations*, or *minimax approximations*. In both cases, we want to minimize a distance $\|p - f\|$, where p is a polynomial of a given degree. For least squares approximations, that distance is:

$$\|p - f\|_{2,[a,b]} = \left(\int_a^b w(x)(f(x) - p(x))^2 dx \right)^{1/2},$$

where w is a continuous *weight function* that can be used to select parts of $[a, b]$ where we want the approximation to be more accurate. For minimax approximations, the distance is:

$$\|p - f\|_{\infty,[a,b]} = \sup_{a \le x \le b} |p(x) - f(x)|.$$

One could also consider distances such as

$$\|p - f\|_{\mathrm{rel},[a,b]} = \sup_{a \le x \le b} \frac{1}{|f(x)|} |p(x) - f(x)|.$$

The least squares approximations are computed by a projection method using orthogonal polynomials. Minimax approximations are computed using an algorithm credited to Remez [Remes 1934; Hart et al. 1968]. See Markstein [2000] and Muller [1997] for recent presentations of elementary function algorithms.

In this article, we are concerned with minimax approximations using distance $\|p - f\|_{\infty,[a,b]}$. And yet, our general method in Section 3.2 also applies to distance $\|p - f\|_{\mathrm{rel},[a,b]}$. Our approximations will be used in finite-precision arithmetic. Hence, the computed polynomial coefficients are usually rounded: let $m_0, m_1, \ldots, m_n$ be a fixed finite sequence of natural integers, the coefficient p_i of the minimax approximation

$$p(x) = p_0 + p_1 x + \cdots + p_n x^n$$

Table I. Latencies (in number of cycles) of Double Precision Floating-Point Addition, Multiplication, and Division on Some Recent Processors

Processor	FP add	FP mult.	FP div.
Pentium IV	5	7	38
PowerPC 750	3	4	31
UltraSPARC III	4	4	24
Alpha21264	4	4	15
Athlon K6-III	3	3	20

is rounded to, for instance, the nearest multiple of 2^{-m_i}. By doing this, we obtain a slightly different polynomial approximation $\hat{p}$. But we have no guarantee that $\hat{p}$ is the best minimax approximation to f among the polynomials whose degree-i coefficient is a multiple of 2^{-m_i}. The aim of this article is to present a way of finding this best truncated approximation. We have two goals in mind:

—rather low precision (e.g., around 24 bits), hardware-oriented, for specific-purpose implementations. In such cases, to minimize multiplier sizes (which increases speed and saves silicon area), the values of m_i, for $i \geq 1$, should be very small. The degrees of the polynomial approximations are low. Typical recent examples are given in Wei et al. [2001] and Pineiro et al. [2001]. Roughly speaking, what matters here is reducing the cost (in terms of delay and area) without making the accuracy unacceptable;

—single-precision or double-precision, software-oriented, for implementation on current general purpose microprocessors. Using table-driven methods, such as the ones suggested by Tang [1989, 1990, 1991, 1992], the degree of the polynomial approximations can be made rather low. Roughly speaking, what matters in this case is to get very high accuracy without making the cost (in terms of delay and memory) unacceptable.

One could object that the m_i's are not necessarily known a priori. For instance, if one wishes a coefficient to be exactly representable in double precision arithmetic, one needs to know the order of magnitude of that coefficient to know what value of m_i corresponds to that wish. And yet, in practice, good approximations of the same degree to a given function have coefficients that are very close (the approach given in Section 3.1 shows this) so that using our approach, with possibly two different values of m_i if the degree-i coefficient of the minimax approximation is very close to a power of i, suffices.

It is important to notice that our polynomial approximations will be computed once only and will be used very frequently (indeed, several billion times for an elementary function program in a widely distributed library). Hence, if it remains reasonably fast, the speed of an algorithm that computes adequate approximations is not extremely important. However, in the practical cases we have studied so far, our method will very quickly give a result.

In this article, we provide a general and efficient method for finding the best truncated approximation(s) (it is not necessarily unique). It consists in building a polytope $\mathfrak{P}$ of $\mathbb{R}^{n+1}$ to which the numerators of the coefficients of this (these) best truncated approximation(s) belong, such that $\mathfrak{P}$ contains a number

as small as possible of points of $\mathbb{Z}^{n+1}$. Once it is achieved, we do an exhaustive search by computing the norms[1]

$$\left\| \frac{a_0}{2^{m_0}} + \frac{a_1}{2^{m_1}}x + \cdots + \frac{a_n}{2^{m_n}}x^n - f \right\|_{\infty,[a,b]}$$

with $(a_0, a_1, \ldots, a_n) \in \mathfrak{P} \cap \mathbb{Z}^{n+1}$.

The method presented here is very flexible since it also applies when we impose supplementary constraints on the truncated polynomials, and/or when distance $\|.\|_{\mathrm{rel},[a,b]}$ is considered. For example, the search can be restricted to odd or even polynomials or, more generally, to polynomials with some fixed coefficients. This is frequently useful: one might for instance, wish the computed value of $\exp(x)$ to be exactly one if $x = 0$ (hence, requiring the degree-0 coefficient of the polynomial approximation to be 1).

Of course, one would like to take into account the roundoff error that occurs during polynomial evaluation: getting the polynomial, with constraints on the size of the coefficients, that minimizes the total (approximation plus roundoff) error would be extremely useful. Although we are currently working on that problem, we do not yet have a solution. First, it is very algorithm-and-architecture dependent (for instance, some architectures have an extended internal precision). Second, since the largest roundoff error and the largest approximation error are extremely unlikely to be attained at exactly the same points, the total error is difficult to predict accurately.

And yet, here are a few observations that lead us to believe that, in many practical cases, our approach will give us polynomials that will be very close to these ideal approximations. Please note that these observations are merely intuitive feelings, and that one can always build cases for which the ideal approximations differ from the ones we compute.

(1) Good approximations of the same degree to a given function have coefficients that are very close in practice. Indeed, the approach given in Section 3.1 shows this.

(2) When evaluating two polynomials whose coefficients are very close on variables that belong to the same input interval, the largest roundoff errors will be very close, too.

(3) In all practical cases, the approximation error oscillates slowly, whereas the roundoff error varies very quickly so that, if the input interval is reasonably small, an error very close to the maximum error is reached near any point. This is illustrated in Figures 1 and 2: we have defined p as the polynomial obtained by rounding to the nearest double precision number each coefficient of the degree-5 minimax approximation to e^x in $[0, 1/128]$. Figure 1 shows the difference $p(x) - e^x$ (approximation error), and Figure 2 shows the difference between the computed value and the exact value of $p(x)$, assuming Horner's scheme is used, in double precision arithmetic.

[1]So far, we have computed these norms using the `infnorm` function of Maple. Our research group is working on a C implementation that will use multiple precision interval arithmetic to get certified upper and lower bounds on the infinite norm of a regular enough function.

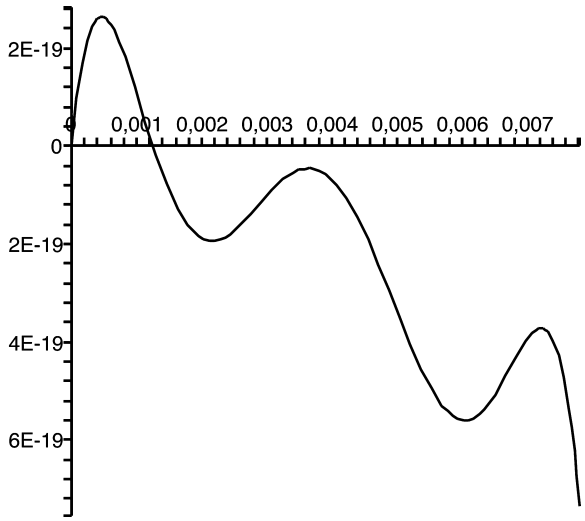


Fig. 1. Approximation error. We have plotted the difference $p(x) - e^x$.

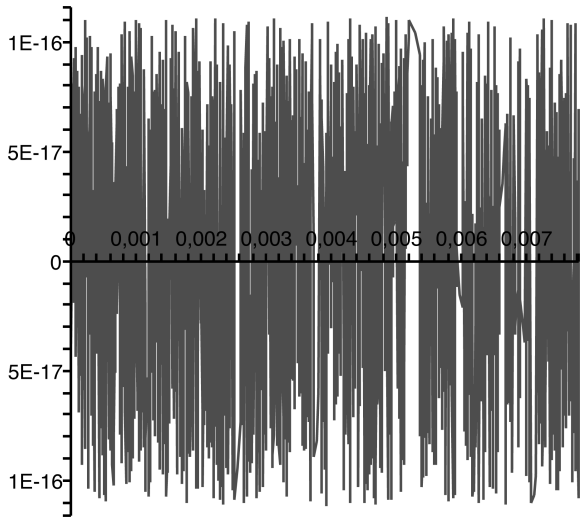


Fig. 2. Roundoff error. We have plotted the difference between the exact and computed values of $p(x)$. The computations are performed using Horner's scheme, in double precision, without using a larger internal format.

These observations tend to indicate that, for all candidate polynomials, the roundoff errors will be very close, and the total error will be close to the sum of the approximation and roundoff errors. Hence, the best polynomial when considering the approximation error only, will be very close to the best polynomial when considering approximation and roundoff errors.

Of course, these observations are not proofs; they are just result from some experiments, and we are far from being able to solve the general problem of

finding the best polynomial, with size constraints on coefficients, when considering approximation and roundoff errors. Hopefully, these remarks will one day help to build a more general method.

The outline of the article is the following. We give an account of Chebyshev polynomials and some of their properties in Section 2. In Section 3, we first provide a method based on Chebyshev polynomials that partially answers the problem, and then we give a general and efficient method based on polytopes that finds a best truncated approximation of a function f over a compact interval $[a, b]$. Despite the fact that it is less efficient and general than the polytope method, we present the method based on Chebyshev polynomials because this approach seems interesting in itself, is simple, and gives results that are easy to use and, moreover, might be useful in other problems. We end Section 3 with a remark illustrating the flexibility of our method. We finish with some examples in Section 4. We complete the article with three appendices. In the first one, we collect the proofs of the statements given in Section 2. In the second one, we prove a lemma used in Section 3.2 that implies, in particular, the existence of a best truncated polynomial approximation. In the last one, we give a worked example of the methods presented here.

To end this introduction, let us mention that a C implementation of our method is in process and also that the method applies to some signal processing problems, namely, finding the rational linear combination of cosines with constraints on the size in bits of the rational coefficients in order to implement (in software or hardware) digital FIR filters. This will be the purpose of a future article.

As we only deal with the supremum norm, wherever there is no ambiguity, we will write $\|\cdot\|_I$ instead of $\|\cdot\|_{\infty,I}$, where I is any real set.

2. SOME REMINDERS ON CHEBYSHEV POLYNOMIALS

Definition 1 (*Chebyshev Polynomials*). The Chebyshev polynomials can be defined either by the recurrence relation

$$\begin{cases} T_0(x) = 1, \\ T_1(x) = x, \\ T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x); \end{cases}$$

or by

$$T_n(x) = \begin{cases} \cos(n \cos^{-1} x) \text{ for } |x| \leq 1, \\ \cosh(n \cosh^{-1} x) \text{ for } x > 1. \end{cases}$$

A presentation of Chebyshev polynomials can be found in Borwein and Erdélyi [1995] and especially in Rivlin [1990]. These polynomials play a central role in approximation theory. The following property is easily derived from Definition 1.

PROPERTY 1. *For $n \geq 0$, we have*

$$T_n(x) = \frac{n}{2} \sum_{k=0}^{\lfloor n/2 \rfloor} (-1)^k \frac{(n-k-1)!}{k!(n-2k)!} (2x)^{n-2k}.$$

Hence, T_n has degree n and its leading coefficient is 2^{n-1}. It has n real roots, all strictly between -1 and 1.

We recall that a *monic* polynomial is a polynomial whose leading coefficient is 1. The following statement is a well known and remarkable property of Chebyshev Polynomials.

PROPERTY 2 (MONIC POLYNOMIALS OF SMALLEST NORM). *Let $a, b \in \mathbb{R}$, $a < b$. The monic degree-n polynomial having the smallest $\| \cdot \|_{[a,b]}$ norm is*

$$\frac{(b-a)^n}{2^{2n-1}} T_n\left(\frac{2x-b-a}{b-a}\right).$$

In the following, we will make use of the polynomials

$$T_n^*(x) = T_n(2x-1).$$

We have (see Fox and Parker [1972, Chapter 3], e.g.) $T_n^*(x) = T_{2n}(x^{1/2})$, hence all the coefficients of T_n^* are nonzero integers.

Now, we state two propositions that generalize Property 2 when dealing with intervals of the form $[0, a]$ and $[-a, a]$.

PROPOSITION 1. *Let $a \in (0, +\infty)$, define*

$$\alpha_0 + \alpha_1 x + \alpha_2 x^2 + \cdots + \alpha_n x^n = T_n^*\left(\frac{x}{a}\right).$$

Let k be an integer, $0 \leq k \leq n$, the polynomial

$$\frac{1}{\alpha_k} T_n^*\left(\frac{x}{a}\right)$$

has the smallest $\| \cdot \|_{[0,a]}$ norm among the polynomials of degree at most n with a degree-k coefficient equal to 1. That norm is $|1/\alpha_k|$.

Remark 1. Moreover, when $k = n = 0$ or $1 \leq k \leq n$, we can show that this polynomial is the only one having this property. We do not give the proof of this uniqueness property in this article since we only need the existence result in the sequel.

PROPOSITION 2. *Let $a \in (0, +\infty)$, $p \in \mathbb{N}$, define*

$$\beta_{0,p} + \beta_{1,p} x + \beta_{2,p} x^2 + \cdots + \beta_{p,p} x^p = T_p\left(\frac{x}{a}\right).$$

Let k and n be integers, $0 \leq k \leq n$.

—If k and n are both even or odd, the polynomial

$$\frac{1}{\beta_{k,n}} T_n\left(\frac{x}{a}\right)$$

has the smallest $\|\cdot\|_{[-a,a]}$ *norm among the polynomials of degree at most* n *with a degree-*k *coefficient equal to* 1. *That norm is* $|1/\beta_{k,n}|$.

—*Else, the polynomial*

$$\frac{1}{\beta_{k,n-1}} T_{n-1}\left(\frac{x}{a}\right)$$

has the smallest $\|\cdot\|_{[-a,a]}$ *norm among the polynomials of degree at most* n *with a degree-*k *coefficient equal to* 1. *That norm is* $|1/\beta_{k,n-1}|$.

3. GETTING THE TRUNCATED POLYNOMIAL THAT IS CLOSEST TO A FUNCTION ON A COMPACT INTERVAL

Let a, b be two real numbers, let f be a function defined on $[a, b]$ and $m_0, m_1, \ldots, m_n$ be $n+1$ integers. Define $\mathcal{P}_n^{[m_0,m_1,\ldots,m_n]}$ as the set of the polynomials of degree less than or equal to n whose degree-i coefficient is a multiple of 2^{-m_i} for all i between 0 and n (we will call these polynomials truncated polynomials), that is to say,

$$\mathcal{P}_n^{[m_0,m_1,\ldots,m_n]} = \left\{ \frac{a_0}{2^{m_0}} + \frac{a_1}{2^{m_1}}x + \cdots + \frac{a_n}{2^{m_n}}x^n,\ a_0, \ldots, a_n \in \mathbb{Z} \right\}.$$

Let p be the minimax approximation to f on $[a, b]$. Define $\hat{p}$ as the polynomial whose degree-i coefficient is obtained by rounding the degree-i coefficient of p to the nearest multiple of 2^{-m_i} (with an arbitrary choice in case of a tie) for $i = 0, \ldots, n$: $\hat{p}$ is an element of $\mathcal{P}_n^{[m_0,m_1,\ldots,m_n]}$.

Also define ϵ and $\hat{\epsilon}$ as

$$\epsilon = \|f - p\|_{[a,b]} \text{ and } \hat{\epsilon} = \|f - \hat{p}\|_{[a,b]}.$$

We assume that $\hat{\epsilon} \neq 0$.

We state our problem as follows. Let $K \geq \epsilon$, we are looking for a truncated polynomial $p^\star \in \mathcal{P}_n^{[m_0,m_1,\ldots,m_n]}$ such that

$$\|f - p^\star\|_{[a,b]} = \min_{q \in \mathcal{P}_n^{[m_0,m_1,\ldots,m_n]}} \|f - q\|_{[a,b]}$$

and

$$\|f - p^\star\|_{[a,b]} \leq K. \tag{1}$$

Lemma 2 in Appendix 2 implies that the number of truncated polynomials satisfying (1) is finite.

When $K = \hat{\epsilon}$, this problem has a solution since $\hat{p}$ satisfies 1. It should be noted that, in that case, $p^\star$ is not necessarily equal to $\hat{p}$.

We can put, for example, $K = \lambda\hat{\epsilon}$ with $\lambda \in [\epsilon/\hat{\epsilon}, 1]$.

3.1 A Partial Approach Through Chebyshev Polynomials

The term partial refers to the fact that the intervals we can deal with in this section must be of the form $[0, a]$ or $[-a, a]$, where $a > 0$. This restriction comes from the following two problems.

(1) We do not have in the general case $[a, b]$ a result analogous to Propositions 1 and 2 and simple to state.

(2) From a given polynomial and a given interval, a simple change of variable allows us to reduce the initial approximation problem to another approximation problem with an interval of the form $[0, a]$ or $[-a, a]$. Unfortunately, this change of variables (that does not keep the size of the coefficients invariant) leads to a system of inequalities of the form we faced in Section 3.2. Then, we have to perform additional operations in order to produce candidate polynomials. In doing so, we lose the main interest—the simplicity—of the approach through Chebyshev polynomials in the cases $[0, a]$ and $[-a, a]$.

We will only deal with intervals $[0, a]$ where $a > 0$ since the method presented in the following easily adapts to intervals $[-a, a]$ where $a > 0$.

In this section, we compute bounds such that, if the coefficients of a polynomial $q \in \mathcal{P}_n^{[m_0,m_1,\ldots,m_n]}$ are not within these bounds, then

$$\|p - q\|_{[0,a]} > \epsilon + K.$$

Knowing these bounds will make an exhaustive searching of $p^\star$ possible. To do this, consider a polynomial q whose degree-i coefficient is $p_i + \delta_i$. From Proposition 1, we have

$$\|q - p\|_{[0,a]} \geq \frac{|\delta_i|}{|\alpha_i|},$$

where α_i is the nonzero degree-i coefficient of $T_n^*(x/a)$. Now, if q is at a distance greater than $\epsilon + K$ from p, it cannot be $p^\star$ since

$$\|q - f\|_{[0,a]} \geq \|q - p\|_{[0,a]} - \|p - f\|_{[0,a]} > K.$$

Therefore, if there exists i, $0 \leq i \leq n$, such that

$$|\delta_i| > (\epsilon + K)|\alpha_i|,$$

then $\|q - p\|_{[0,a]} > \epsilon + K$ and therefore $q \neq p^\star$. Hence, the degree-i coefficient of $p^\star$ necessarily lies in the interval $[p_i - (\epsilon + K)|\alpha_i|, p_i + (\epsilon + K)|\alpha_i|]$. Thus we have

$$\underbrace{\left\lceil 2^{m_i}(p_i - (\epsilon + K)|\alpha_i|)\right\rceil}_{c_i} \leq 2^{m_i} p_i^\star \leq \underbrace{\left\lfloor 2^{m_i}(p_i + (\epsilon + K)|\alpha_i|)\right\rfloor}_{d_i}, \tag{2}$$

since $2^{m_i} p_i^\star$ is an integer. Note that, as $0 \in [0, a]$, Condition (1) implies in particular

$$f(0) - K \leq p_0^\star \leq f(0) + K,$$

that is, since $2^{m_0} p_0^\star$ is an integer,

$$\underbrace{\lceil 2^{m_0}(f(0) - K)\rceil}_{c_0'} \leq 2^{m_0} p_0^\star \leq \underbrace{\lfloor 2^{m_0}(f(0) + K)\rfloor}_{d_0'}.$$

We replace c_0 with $\max(c_0, c_0')$ and d_0 with $\min(d_0, d_0')$.

For $i = 0, \ldots, n$, we have $d_i - c_i + 1$ possible values for the integer $2^{m_i} p_i^\star$. This means that we have $\prod_{i=0}^n (d_i - c_i + 1)$ candidate polynomials. If this amount is small enough, we search for $p^\star$ by computing the norms $\|f - q\|_{[0,a]}$, q running among the possible polynomials. Otherwise, we need an additional step to decrease the number of candidates. Hence, we now give a method for this purpose.

It allows us to reduce the number of candidate polynomials dramatically and applies more generally to intervals of the form $[a, b]$ where a and b are any real numbers.

3.2 A General and Efficient Approach Through Polytopes

From now on, we will deal with intervals of the form $[a, b]$ where a and b are any real numbers.

We recall the following definitions from Schrijver [2003].

Definitions 1. Let $k \in \mathbb{N}$.

A subset $\mathfrak{P}$ of $\mathbb{R}^k$ is called a polyhedron if there exists an $m \times k$ matrix A with real coefficients and a vector $b \in \mathbb{R}^m$ (for some $m \geq 0$) such that $\mathfrak{P} = \{x \in \mathbb{R}^k | Ax \leq b\}$.

A subset $\mathfrak{P}$ of $\mathbb{R}^k$ is called a polytope if $\mathfrak{P}$ is a bounded polyhedron.

A polyhedron (or a polytope) $\mathfrak{P}$ is called rational if it is determined by a rational, respectively, system of linear inequalities.

The $n + 1$ inequalities given by 2 define a rational polytope of $\mathbb{R}^{n+1}$ which the numerators of $p^\star$ (i.e. the $2^{m_i} p_i^\star$) belong to. The idea[2] is to build a polytope $\mathfrak{P}$, still containing the $2^{m_i} p_i^\star$, such that $\mathfrak{P} \cap \mathbb{Z}^{n+1}$ is the smallest possible, which means that the number of candidate polynomials is the smallest possible, in order to reduce as much as possible the final step of computation of supremum norms. Once we get this polytope, we need an efficient way of producing these candidates, that is, an efficient way of scanning the integer points (i.e., to points with integer coordinates) of the rational polytope we built. Several algorithms allow us to achieve this. The one given in Ancourt and Irigoin [1991] uses the Fourier-Motzkin pairwise elimination, the one given in Feautrier [1988] and Collard et al. [1995] is a parameterized version of the Dual Simplex method and the one given in Le Verge et al. [1994] is based on the dual representation of polyhedra used in Polylib [The Polylib Team 2004]. The last two algorithms allow us to produce in an optimized way[3] the loops in our final program of exhaustive search. Note that these algorithms have, at worst, the complexity of integer linear programming[4]. Now, let us give the details of the method.

First we notice that the previous approach handles the unknowns $p_i^\star$ separately which seems unnatural. Hence, the basic aim of the method is to construct a polytope defined by inequalities that take into account in a more satisfying way the dependence relations between the unknowns. This polytope should contain fewer points of $\mathbb{Z}^{n+1}$ than the one built from the Chebyshev polynomials' approach. The examples of Section 4 indicate that this seems to be the case (and the improvements can be dramatic).

[2]After the submission of this article, we read that this idea has already been proposed in Habsieger and Salvy [1997], a paper dealing with number-theoretical issues, but the authors did not have any efficient method for scanning the integer points of the polytope.

[3]By optimized, we mean that all points of the polytope are scanned only once.

[4]In fact, the algorithm given in Feautrier [1988] and Collard et al. [1995] has, in the situation we are facing, the complexity of rational linear programming.

Condition (1) means

$$f(x) - K \leq \sum_{j=0}^{n} p_j^\star x^j \leq f(x) + K \tag{3}$$

for all $x \in [a, b]$.

The idea is to consider inequalities (3) for a certain number (chosen by the user) of values of $x \in [a, b]$. As we want to build rational polytopes, these values must be rational numbers. We propose the following choice of values. Let A be a rational approximation to a greater than or equal to a, let B be a rational approximation to b less than or equal to b and such that $B > A$, let d be a nonzero natural integer chosen by the user. We show in Lemma 2 in Appendix 2 that we must have $d \geq n$ in order to ensure that we get a polytope (which implies the finiteness of the number of the best truncated approximation(s)). We consider the rational values[5] $x_i = A + \frac{i}{d}(B - A)$, $i = 0, \ldots, d$. Then again, since the polytope has to be rational, we compute, for $i = 0, \ldots, d$, two rational numbers l_i and u_i that are rational approximations to, respectively, $f(x_i) - K$ and $f(x_i)+K$ such that $l_i \leq f(x_i)-K$ and $u_i \geq f(x_i)+K$. The rational polytope $\mathfrak{P}$ searched is therefore defined by the inequalities

$$l_i \leq \sum_{j=0}^{n} p_j^\star x_i^j \leq u_i, \quad i = 0, \ldots, d. \tag{4}$$

If $\mathfrak{P} \cap \mathbb{Z}^{n+1}$ is small enough (this can be estimated thanks to Polylib), we start our exhaustive search by computing the norms

$$\left\| \frac{a_0}{2^{m_0}} + \frac{a_1}{2^{m_1}} x + \ldots \frac{a_n}{2^{m_n}} x^n - f \right\|_{[a,b]}, \tag{5}$$

with $(a_0, a_1, \ldots, a_n) \in \mathfrak{P} \cap \mathbb{Z}^{n+1}$. This set can be scanned efficiently thanks to one of the algorithms given in Ancourt and Irigoin [1991], Feautrier [1988] and Collard et al. [1995] or Le Verge et al. [1994] that we have previously quoted. Or else, we increase the value of the parameter d in order to construct another rational polytope $\mathfrak{P}'$ that contains fewer elements of $\mathbb{Z}^{n+1}$. We must point out that, assuming that the new parameter is greater than d, does not necessarily lead to a new polytope with fewer elements of $\mathbb{Z}^{n+1}$ inside (it is easy to show such counterexamples), but it is reasonable to expect that, generally, a polytope built with a greater parameter should contain fewer elements of $\mathbb{Z}^{n+1}$ inside since it is defined from a larger number of inequalities (4) or, in other words, as our discretization method is done using a larger number of rational points, which should allow us to restrict the number of possible candidates since there are more conditions to satisfy. It is indeed the case if we choose any positive integer multiple of d as new parameter. In this case, the new polytope $\mathfrak{P}'$, associated with the parameter νd, with $\nu \in \mathbb{N}^*$, is a subset of $\mathfrak{P}$: $\mathfrak{P}$ is built from the set of rational points $\{A + \frac{i}{d}(B - A)\}_{i=0,\ldots,d}$ which is a subset of $\{A + \frac{j}{\nu d}(B - A)\}_{j=0,\ldots,\nu d}$ from which $\mathfrak{P}'$ is built.

[5]Choosing equally-spaced rational values seems a natural choice when dealing with regular enough functions. And yet, in a few cases, we get a better result with very irregularly-spaced points. This is something we plan to investigate in the near future.

Remark 2. As we said in the introduction, this method is very flexible. We give some examples to illustrate it.

—If we restrict our search to odd truncated polynomials (in this case, we put $n = 2k + 1$ and we must have $d \geq k$), it suffices to replace inequalities (4) with

$$l_i \leq \sum_{j=0}^{k} p_j^\star x_i^{2j+1} \leq u_i, \quad i = 0, \ldots, d$$

to create a polytope $\mathfrak{P}$ of $\mathbb{R}^{k+1}$ whose points with integer coordinates we scan.

—If we restrict our search to truncated polynomials some of whose coefficients have fixed values, (e.g., if we assume that the truncated polynomials sought have constant term equal to 1) it suffices to replace inequalities (4) with

$$l_i \leq 1 + \sum_{j=1}^{n} p_j^\star x_i^{2j+1} \leq u_i, \quad i = 0, \ldots, d$$

to create a polytope $\mathfrak{P}$ of $\mathbb{R}^n$ whose points we scan with integer coordinates (we must have $d \geq n - 1$).

—Our method also applies to the search for the best truncated polynomial with respect to the relative error distance $\| \cdot \|_{\mathrm{rel},[a,b]}$ defined in the introduction. In this case, we can state the problem as follows. Let $K \geq 0$, we search for a truncated polynomial $p^\star \in \mathcal{P}_n^{[m_0,m_1,\ldots,m_n]}$ such that

$$\| f - p^\star \|_{\mathrm{rel},[a,b]} = \min_{q \in \mathcal{P}_n^{[m_0,m_1,\ldots,m_n]}} \| f - q \|_{\mathrm{rel},[a,b]}$$

and

$$\| f - p^\star \|_{\mathrm{rel},[a,b]} \leq K. \tag{6}$$

It suffices to consider the inequalities

$$-K|f(x)| - f(x) \leq \sum_{j=0}^{n} p_j^\star x^j \leq K|f(x)| + f(x)$$

for at least $n + 1$ distinct rational values of $x \in [a, b]$ to make a polytope to which we apply the method presented.

4. EXAMPLES

We implemented in Maple the method given in Section 3.1, and we started developing a C library that implements the method described in Section 3.2.

For producing the results presented in this section, we implemented the approach through polytopes in a C program of our own, based on the polyhedral library Polylib [The Polylib Team 2004]. This allows us to treat a lot of examples, but it is too roughly done to tackle examples with approximations of degree 15 to 20, for instance. Our goal is to implement the method presented here in a C library which would use Polylib and PIP Feautrier [2003] that implements the parametric integer linear programming solver given in Feautrier [1988] and Collard et al. [1995] for scanning the integer points of the polytope. The choice of PIP instead of the algorithm given in Le Verge et al. [1994] is due to

Table II. Examples

	f	$[a,b]$	m	ϵ	$\hat{\epsilon}$	K
1	cos	$\left[0, \frac{\pi}{4}\right]$	[12, 10, 6, 4]	$1.135...10^{-4}$	$6.939...10^{-4}$	$\hat{\epsilon}/2$
2	exp	$\left[0, \frac{1}{2}\right]$	[15, 14, 12, 10]	$2.622...10^{-5}$	$3.963...10^{-5}$	$< \hat{\epsilon}$
3	exp	$\left[0, \log\left(1+\frac{1}{2048}\right)\right]$	[56, 45, 33, 23]	$1.184...10^{-17}$	$2.362...10^{-17}$	$< \hat{\epsilon}$
4	$\arctan(1+x)$	$\left[0, \frac{1}{4}\right]$	[24, 21, 18, 17, 16]	$2.381...10^{-8}$	$3.774...10^{-8}$	$< \hat{\epsilon}$
5	exp	$\left[-\frac{\log(2)}{256}, \frac{\log(2)}{256}\right]$	[25, 17, 9]	$8.270...10^{-10}$	$3.310...10^{-9}$	$< \hat{\epsilon}$
6	exp	$\left[-\frac{\log(2)}{256}, -\frac{\log(2)}{256}\right]$	[28, 19, 9]	$8.270...10^{-10}$	$3.310...10^{-9}$	$< \hat{\epsilon}$
7	$\frac{\log(3/4+x)}{\log(2)}$	$[-1/4, 1/4]$	[12, 9, 7, 5]	$6.371...10^{-4}$	$7.731...10^{-4}$	$< \hat{\epsilon}$
8	$\frac{\log(\sqrt{2}/2+x)}{\log(2)}$	$\left[\frac{1-\sqrt{2}}{2}, \frac{2-\sqrt{2}}{2}\right]$	[12, 9, 7, 5]	$6.371...10^{-4}$	$9.347...10^{-4}$	$< \hat{\epsilon}$

Table III. Corresponding Results

	Chebyshev	Polytope (d)	T_1	T_2	Gain of Accuracy in Bits
1	330	1 (4)	0.62	0.26	≈ 1.5
2	84357	9 (20)	0.51	2.18	≈ 0.375
3	9346920	15 (20)	0.99	1.77	≈ 0.22
4	192346275	1 (20)	0.15	0.55	≈ 0.08
5	1	0 (4)	0.05	0	0
6	4	1 (4)	0.03	0.10	≈ 0.41
7	38016	2 (15)	0.13	0.69	≈ 0.06
8		12 (20)	0.59	5.16	≈ 0.26

the fact that our polytope may, in some cases, have a large amount of vertices (due to the need for a large amount of points x_j, i.e., constraints, when building the polytope) which can generate memory problems if we use Le Verge et al. [1994]. We also need the MPFR multiple-precision library [The Spaces Project 2004] since we face precision problems when forming the polytope, and the GMP library [Granlund 2002] since the polytope is defined by a matrix and a vector which may have very large integer coefficients. The GMP library is also used inside some Polylib computations.

We put some examples in Table II, and we group the corresponding results in Table III. In the last column of Table II, the notation $< \hat{\epsilon}$ means that we chose a value slightly smaller than $\hat{\epsilon}$, namely $\hat{\epsilon}$ rounded down to four fractional digits.

In the first column of Table III, one can find the number of candidates given by Chebyshev's approach. In the second, we give the number of candidates given by the approach through polytopes and the corresponding parameter d. T_1 denotes the time in seconds for producing the candidate polynomials with the polytope method. T_2 denotes the time in seconds for computing the norms (5) which, for the moment (cf. footnote 1), is done with Maple 9 in which the value of `Digits` was fixed equal to 20. In the last column, we give the gain of accuracy in bits that we get if we use the best truncated polynomial instead of polynomial $\hat{p}$.

All the computations were done on a 2.53GHz Pentium 4 computer running Debian Linux with 256MB RAM. The compiler used was `gcc` without optimization.

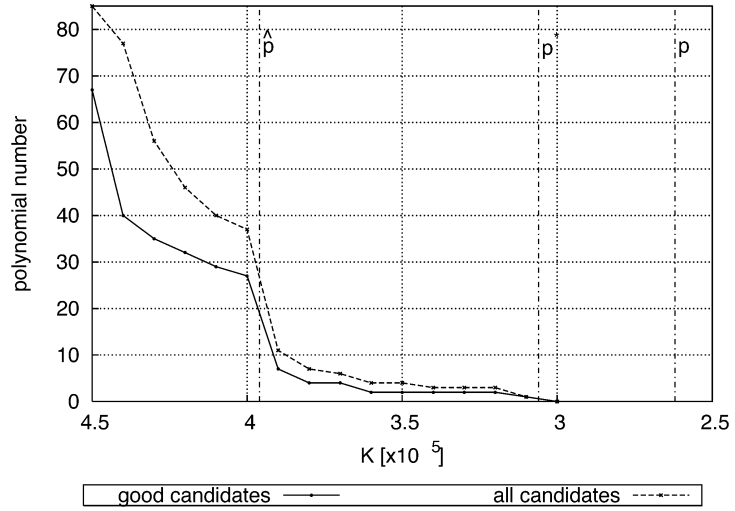


Fig. 3. This figure shows how the number of integer points of the polytope drops when K decreases, in the case of example 2. The term all candidates means the integer points of the polytope. By good candidate, we mean the points that fulfill the requirement (1).

4.1 Choice of the Examples

These examples correspond to common cases that occur in elementary and special function approximation [Muller 1997]. Examples 1, 2, and 4 are of immediate interest. Example 3 is of interest for implementing the exponential function in double precision arithmetic, using a table-driven method such as Tang's method [Tang 1991, 1992]. Examples 5 and 6 also correspond to a table-driven implementation of the exponential function in single precision. Examples 7 and 8 aim at producing very cheap approximations to logarithms in [1/2, 1], for special purpose applications.

Figure 3 shows how the number of integer points of the polytope drops when K decreases, in the case of example 2.

APPENDIX 1. PROOFS OF PROPOSITIONS 1 AND 2

Proving those propositions first requires the following two results. The first one is well known.

PROPERTY 3. *There are exactly $n+1$ values $x_0, x_1, x_2, \ldots, x_n$ such that*

$$-1 = x_0 < x_1 < x_2 < \cdots < x_n = 1,$$

which satisfy

$$T_n(x_i) = (-1)^{n-i} \max_{x\in[-1,1]} |T_n(x)| \quad \forall i,\ i = 0, \ldots,\ n.$$

That is, the maximum absolute value of T_n is attained at the x_i's, and the sign of T_n alternates at these points.

PROOF. These extreme points are simply the points $\cos(k\pi/n)$, $k = 0, \ldots, n$. □

LEMMA 1. *Let $(\delta_i)_{i=0,\ldots,n}$ be an increasing sequence of nonnegative integers and*

$$P(x) = a_0 x^{\delta_0} + \cdots + a_n x^{\delta_n} \in \mathbb{R}[x],$$

then either $P = 0$ or P has at most n zeros in $(0, +\infty)$.

PROOF. By induction on n. For $n = 0$, it is straightforward. Now we assume that the property is true until the rank n. Let

$$P(x) = a_0 x^{\delta_0} + \cdots + a_n x^{\delta_n} + a_{n+1} x^{\delta_{n+1}} \in \mathbb{R}[x]$$

with $0 \leq \delta_0 < \cdots < \delta_{n+1}$ and $a_0 a_1 \ldots a_{n+1} \neq 0$. Assume that P has at least $n+2$ zeros in $(0, +\infty)$. Then $P_1 = P/x^{\delta_0}$ has at least $n+2$ zeros in $(0, +\infty)$.

Thus, the nonzero polynomial

$$P_1'(x) = (\delta_1 - \delta_0) a_1 x^{\delta_1 - \delta_0} + \cdots + (\delta_{n+1} - \delta_0) a_{n+1} x^{\delta_{n+1} - \delta_0}$$

has, from Rolle's Theorem, at least $n+1$ zeros in $(0, +\infty)$ which contradicts the induction hypothesis. □

PROOF OF PROPOSITION 1. From Property 3, there exist $0 = \eta_0 < \eta_1 < \cdots < \eta_n = 1$ such that

$$\alpha_k^{-1} T_n^*(\eta_i) = \alpha_k^{-1} (-1)^{n-i} \|T_n^*(\cdot/a)\|_{[0,a]} = \alpha_k^{-1} (-1)^{n-i}.$$

Let $q(x) = \sum_{\substack{j=0, \\ j \neq k}}^{n} c_j x^j \in \mathbb{R}[x]$ satisfy $\|x^k - q(x)\|_{[0,a]} < |\alpha_k^{-1}|$. Then the polynomial $P(x) = \alpha_k^{-1} T_n^*(x) - (x^k - q(x))$ has the form $\sum_{\substack{j=0, \\ j \neq k}}^{n} d_j x^j$ and is not identically zero. As it changes sign between any two consecutive extrema of T_n^*, it has at least n zeros in $(0, +\infty)$. Hence, from Lemma 1, it must vanish identically which is the contradiction desired. □

PROOF OF PROPOSITION 2. We assume that n is even since a straightforward adaptation of the proof in this case gives the proof of the case where n is odd.

First, we suppose k even. Let

$$P(x) = a_n x^n + \cdots + a_{k+1} x^{k+1} + x^k + a_{k-1} x^{k-1} + \cdots + a_0$$

such that $\|P\|_{[-a,a]} < |1/\beta_{k,n}|$. Then, for all $x \in [-a, a]$,

$$-\frac{1}{|\beta_{k,n}|} < Q(x) := \frac{P(x) + P(-x)}{2} < \frac{1}{|\beta_{k,n}|}$$

that is, since k and n are both even, for all $x \in [-a, a]$,

$$-\frac{1}{|\beta_{k,n}|} < Q(x) = a_n x^n + \cdots + a_{k+2} x^{k+2} + x^k + a_{k-2} x^{k-2} + \cdots + a_0 < \frac{1}{|\beta_{k,n}|}.$$

From Property 3 and the inequality $\|Q\|_{[-a,a]} < |1/\beta_{k,n}| = \|T_n(\cdot/a)\|_{[-a,a]}$, the polynomial $R(x) = Q(x) - T_n(x/a)/\beta_{k,n} = \sum_{\substack{j=0, \\ j \neq k/2}}^{n/2} c_j x^{2j}$ changes sign between two consecutive extrema of $T_n(x/a)$. Then, it has at least n distinct zeros in $[-a, a]$ and, more precisely, in $[-a, 0) \cup (0, a]$ since 0 is an extrema of $T_n(x/a)$ as n is even. Hence, the polynomial $R(\sqrt{x}) = \sum_{\substack{j=0, \\ j \neq k/2}}^{n/2} c_j x^j$ has at least $n/2$ distinct zeros in $(0, \sqrt{a}]$: it is identically zero from Lemma 1.

We have just proved that

$$P(x) = \frac{1}{\beta_{k,n}} T_n\left(\frac{x}{a}\right) + \frac{P(x) - P(-x)}{2}.$$

We recall that $|P(x)| < 1/|\beta_{k,n}|$ for all $x \in [-a, a]$. Therefore, we have, by substituting 1 and -1 to x

$$-\frac{1}{|\beta_{k,n}|} - \frac{1}{\beta_{k,n}} T_n\left(\frac{1}{a}\right) < \frac{P(1) - P(-1)}{2} < \frac{1}{|\beta_{k,n}|} - \frac{1}{\beta_{k,n}} T_n\left(\frac{1}{a}\right)$$

and

$$-\frac{1}{|\beta_{k,n}|} - \frac{1}{\beta_{k,n}} T_n\left(-\frac{1}{a}\right) < \frac{P(-1) - P(1)}{2} < \frac{1}{|\beta_{k,n}|} - \frac{1}{\beta_{k,n}} T_n\left(-\frac{1}{a}\right)$$

As n is even, we know that $T_n(1/a) = T_n(-1/a) = 1$. Hence, we obtain

$$0 < \frac{P(1) - P(-1)}{2} < 0$$

which is the contradiction desired.

Now, we suppose k odd. Let

$$P(x) = a_n x^n + \cdots + a_{k+1} x^{k+1} + x^k + a_{k-1} x^{k-1} + \cdots + a_0$$

such that $\|P\|_{[-a,a]} < |1/\beta_{k,n-1}|$. Then, for all $x \in [-a, a]$,

$$-\frac{1}{|\beta_{k,n-1}|} < Q(x) := \frac{P(x) - P(-x)}{2} < \frac{1}{|\beta_{k,n-1}|}$$

that is, for all $x \in [-a, a]$,

$$-\frac{1}{|\beta_{k,n-1}|} < Q(x) = a_{n-1} x^{n-1} + \cdots + a_{k+2} x^{k+2} + x^k + a_{k-2} x^{k-2} + \cdots + a_1 < \frac{1}{|\beta_{k,n-1}|}.$$

From Property 3 and the inequality $\|Q\|_{[-a,a]} < |1/\beta_{k,n-1}| = \|T_{n-1}(\cdot/a)\|_{[-a,a]}$, the polynomial $R(x) = Q(x) - T_{n-1}(x/a)/\beta_{k,n-1} = \sum_{\substack{j=0,\\ j\neq(k-1)/2}}^{n/2-1} c_j x^{2j+1}$ changes sign between two consecutive extrema of $T_{n-1}(x/a)$. Then, it has at least $n-1$ distinct zeros in $[-a, a]$ and, in particular, at least $n-2$ distinct zeros in $[-a, 0) \cup (0, a]$. Hence, the polynomial $R(\sqrt{x})/\sqrt{x} = \sum_{\substack{j=0,\\ j\neq(k-1)/2}}^{n/2-1} c_j x^j$ has at least $n/2 - 1$ distinct zeros in $(0, \sqrt{a}]$: it is identically zero from Lemma 1.

We have just obtained that

$$P(x) = \frac{1}{\beta_{k,n-1}} T_{n-1}\left(\frac{x}{a}\right) + \frac{P(x) + P(-x)}{2}.$$

Here again, we recall that $|P(x)| < 1/|\beta_{k,n-1}|$ for all $x \in [-a, a]$. Thus, it follows, by substituting 1 and -1 to x

$$-\frac{1}{|\beta_{k,n-1}|} - \frac{1}{\beta_{k,n-1}} T_n\left(\frac{1}{a}\right) < \frac{P(1) + P(-1)}{2} < \frac{1}{|\beta_{k,n-1}|} - \frac{1}{\beta_{k,n-1}} T_{n-1}\left(\frac{1}{a}\right)$$

and

$$-\frac{1}{|\beta_{k,n-1}|} - \frac{1}{\beta_{k,n-1}} T_{n-1}\left(-\frac{1}{a}\right) < \frac{P(-1) + P(1)}{2} < \frac{1}{|\beta_{k,n-1}|} - \frac{1}{\beta_{k,n-1}} T_{n-1}\left(-\frac{1}{a}\right)$$

As n is even, we have $T_{n-1}(1/a) = -T_{n-1}(-1/a) = 1$. Hence, we obtain

$$0 < \frac{P(1) + P(-1)}{2} < 0$$

which is the contradiction desired. □

APPENDIX 2

We now prove the following statement.

LEMMA 2. *Let $d, n \in \mathbb{N}$, let $x_0, \ldots, x_d, l_0, \ldots, l_d, u_0, \ldots, u_d \in \mathbb{R}$ (resp. $\mathbb{Q}$) such that $x_i \neq x_j$ if $i \neq j$, let $\mathfrak{P}$ the set defined by*

$$\mathfrak{P} = \left\{ (\alpha_0, \ldots, \alpha_n) \in \mathbb{R}^{n+1} \text{ (resp. } \mathbb{Q}^{n+1}) : l_i \leq \sum_{j=0}^{n} \alpha_j x_i^j \leq u_i \text{ for } i = 0, \ldots, d \right\}.$$

If $d \geq n$, then $\mathfrak{P}$ is a polytope (resp. rational polytope). If $d < n$, then either $\mathfrak{P}$ is empty or $\mathfrak{P}$ is unbounded.

PROOF. The set $\mathfrak{P}$ is obviously a polyhedron (respectively rational polyhedron). So, if we want to prove that $\mathfrak{P}$ is a polytope (respectively rational polytope), it suffices to show that it is bounded.

First, we assume $d \geq n$. Then, $\mathfrak{P}$ is contained in the set $\mathfrak{P}'$ defined by

$$\mathfrak{P}' = \left\{ (\alpha_0, \ldots, \alpha_n) \in \mathbb{R}^{n+1} : l_i \leq \sum_{j=0}^{n} \alpha_j x_i^j \leq u_i \text{ for } i = 0, \ldots, n \right\}.$$

Let

$$\begin{array}{rcl} \varphi : \mathbb{R}^{n+1} & \longrightarrow & \mathbb{R}^{n+1} \\ (\alpha_0, \ldots, \alpha_n) & \mapsto & \left(\displaystyle\sum_{j=0}^{n} \alpha_j x_0^j, \ldots, \sum_{j=0}^{n} \alpha_j x_n^j \right) \end{array}$$

The linear application φ is an isomorphism for the matrix

$$\begin{pmatrix} 1 & x_0 & \cdots & x_0^n \\ 1 & x_1 & \cdots & x_1^n \\ \vdots & \vdots & & \vdots \\ 1 & x_n & \cdots & x_n^n \end{pmatrix}$$

is a nonsingular Vandermonde matrix since $x_i \neq x_j$ if $i \neq j$. The linear application φ^{-1}, defined on a finite dimensional $\mathbb{R}$-vector space, is continuous and the set $I_n = \prod_{i=0}^{n} [l_i, u_i]$ is a compact of $\mathbb{R}^{n+1}$. Thus, the set $\mathfrak{P}'$ which is equal to $\varphi^{-1}(I_n)$ is a compact of $\mathbb{R}^{n+1}$, which implies that $\mathfrak{P}$ is necessarily bounded.

Now, we assume $d < n$ and $\mathfrak{P}$ non empty. Then, we notice that $\mathfrak{P}$ is defined by the inequalities

$$\begin{array}{ccccc} l_0 - \sum_{j=d+1}^{n} \alpha_j x_0^j & \le & \sum_{j=0}^{d} \alpha_j x_0^j & \le & u_0 - \sum_{j=d+1}^{n} \alpha_j x_0^j, \\ \vdots & \le & \vdots & \le & \vdots \\ l_d - \sum_{j=d+1}^{n} \alpha_j x_d^j & \le & \sum_{j=0}^{d} \alpha_j x_d^j & \le & u_d - \sum_{j=d+1}^{n} \alpha_j x_d^j. \end{array}$$

As $x_i \neq x_j$ if $i \neq j$, the matrix

$$A = \begin{pmatrix} 1 & x_0 & \cdots & x_0^d \\ 1 & x_1 & \cdots & x_1^d \\ \vdots & \vdots & & \vdots \\ 1 & x_d & \cdots & x_d^d \end{pmatrix}$$

is nonsingular. Hence, in particular, the polyhedron $\mathfrak{P}$ contains the family of vectors

$$\left\{ A^{-1} \begin{pmatrix} u_0 - \sum_{j=d+1}^{n} \alpha_j x_0^j \\ u_1 - \sum_{j=d+1}^{n} \alpha_j x_1^j \\ \vdots \\ u_d - \sum_{j=d+1}^{n} \alpha_j x_d^j \end{pmatrix}, (\alpha_{d+1}, \ldots, \alpha_n) \in \mathbb{Z}^{n-d} \right\},$$

which proves that $\mathfrak{P}$ can not be bounded. □

APPENDIX 3. A WORKED EXAMPLE

Let us give the details of the first example of Table II. We focus on the degree-3 approximation to the cosine function in $[0, \pi/4]$.

First, we determine (here using the `numapprox` package of Maple with `Digits` equal to 10) the degree-3 minimax polynomial associated to cos on $[0, \pi/4]$. We get

$$p = 0.9998864206 + (0.00469021603 + (-0.5303088665 + 0.06304636099x)x)x.$$

Then, $\epsilon = \|f - p\|_{[0,\pi/4]} = 0.0001135879209$. This means that such an approximation is not good enough for single-precision implementation of the cosine function. It can be of interest for some special purpose implementations. We have

$$\hat{p} = \frac{1}{16}x^3 - \frac{17}{32}x^2 + \frac{5}{1024}x + 1 \text{ and } \hat{\epsilon} = \|f - \hat{p}\|_{[0,\pi/4]} = 0.0006939707.$$

Let us choose $K = \hat{\epsilon}/2$. Then, the approach that uses Chebyshev polynomials gives

—3 possible values between 4095/4096 and 4097/4096 for the degree-0 term,
—22 possible values between $-3/512$ and $15/1024$ for the degree-1 term,
—5 possible values between $-9/16$ and $-1/2$ for the degree-2 term,
—1 possible value equal to $1/16$ for the degree-3 term.

Hence, the approach that uses Chebyshev polynomials provides 330 candidate polynomials. This is a reasonably small number the time necessary for the exhaustive computation of the norms 5 is small[6]. There is no need to use the approach based on polytopes; we use it anyway in the following to show how it works.

First, we define a subinterval $[A, B]$ of $[0, \pi/4]$ with rational bounds. Let $A = 0$, we choose a rational approximation B of $\pi/4$ such that $B \leq \pi/4$ and its numerator and denominator are small in order to speed up the computations. The bigger the integers occuring in the data defining the polytope are, the slower the computations are performed. Hence, let $B = 7/9$, which is a convergent of the continued fraction expansion of $\pi/4$.

We choose a value of d equal to 4, that is, the polytope is built from five rational points which are $0, \frac{1}{4}\cdot\frac{7}{9}, \frac{2}{4}\cdot\frac{7}{9}, \frac{3}{4}\cdot\frac{7}{9}, \frac{7}{9}$. Let $a_j^\star = 2^{m_j} p_j^\star$ for $j = 0, \ldots, 3$. Therefore, we want the inequalities hereafter to be satisfied:

$$\cos\left(\frac{7i}{36}\right) - K \leq \sum_{j=0}^{3} \frac{a_j^\star}{2^{m_j}} \left(\frac{7i}{36}\right)^j \leq \cos\left(\frac{7i}{36}\right) + K \text{ for } i = 0, \ldots, 4. \qquad (7)$$

We still assume that $K = \hat{\epsilon}/2$. These inequalities define a polytope, but it is not a rational one. Hence, we compute rational approximations of the lower and upper bounds in 7: for $i = 0, \ldots, 4$, we compute l_i and $u_i \in \mathbb{Q}$ such that $l_i \leq \cos(\frac{7i}{36}) - K$, $\cos(\frac{7i}{36}) + K \leq u_i$. Here again, our target is to prevent the increase of the size of the involved integers. Therefore, we can either compute convergents of the l_i and u_i or we can impose that, for each i, l_i and u_i have the same denominator as the $\frac{a_j^\star}{2^{m_j}}(\frac{7i}{36})^j$ for $j = 0, \ldots, 3$. We choose the second possibility, hence, for all $i = 0, \ldots, 4$, we put

$$l_i = \left\lceil D_i \left(\cos\left(\frac{7i}{36}\right) - K\right)\right\rceil / D_i \text{ and } u_i = \left\lfloor D_i \left(\cos\left(\frac{7i}{36}\right) + K\right)\right\rfloor / D_i$$

where D_i is the least common multiple of the denominators of the $\frac{a_j^\star}{2^{m_j}}(\frac{7i}{36})^j$ for $j = 0, \ldots, 3$.

We obtain a rational polytope defined by $AX \leq B$ with

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 729 & 567 & 1764 & 1372 \\ -729 & -567 & -1764 & -1372 \\ 729 & 1134 & 7056 & 10976 \\ -729 & -1134 & -7056 & -10976 \\ 27 & 63 & 588 & 1372 \\ -27 & -63 & -588 & -1372 \\ 729 & 2268 & 28224 & 87808 \\ -729 & -2268 & -28224 & -87808 \end{pmatrix}, X = \begin{pmatrix} a_0^\star \\ a_1^\star \\ a_2^\star \\ a_3^\star \end{pmatrix}, B = \begin{pmatrix} 4097 \\ -4095 \\ 2930749 \\ -2928678 \\ 2764059 \\ -2761988 \\ 92341 \\ -92266 \\ 2128473 \\ -2126402 \end{pmatrix}.$$

[6]Around 1.5 second on a 2.53GHz Pentium 4 computer running Debian Linux with 256MB RAM.

Table IV. Number of Points, and Computational Times (in C and in Maple) for Various Choices of d and K in Example 1

d	K	Time [s] T_1	Time [s] T_2	# Polynomials
9	$7.00\,10^{-4}$	2.60	1.68	7
9	$5.00\,10^{-4}$	1.59	0.98	4
9	$2.50\,10^{-4}$	0.25	0.26	1
9	$6.93\,10^{-4}$	2.55	1.40	6
18	$6.93\,10^{-4}$	2.84	1.40	6
36	$6.93\,10^{-4}$	3.37	1.40	6
72	$6.93\,10^{-4}$	7.39	1.40	6
$p^\star$: (4095, 6, −34, 1) $2.44\,10^{-4}$ (gain 1.5 bits)				

Then, our current C implementation returns only one candidate (4095, 6, −34, 1). We check that it satisfies condition (1). We therefore directly obtain

$$p^\star = \frac{1}{16}x^3 - \frac{17}{32}x^2 + \frac{3}{512}x + \frac{4095}{4096} \quad \text{and} \quad \|f - p^\star\|_{[0,\pi/4]} = 0.0002441406250.$$

In this example, the distance between f and $p^\star$ is approximately 0.35 times the distance between f and $\hat{p}$. Using our approach saves around $-\log_2(0.35) \approx 1.5$ bits of accuracy.

Table IV gives some figures (number of points of the polytope, delay of computation) depending on the choices of d and K in this example. Here again, T_1 denotes the time in seconds for producing the candidate polynomials, and T_2 denotes the time in seconds for computing the norms 5.

ACKNOWLEDGMENTS

We would like to thank the referees, who greatly helped to improve the quality of the manuscript. We would also like to thank Nicolas Boullis and Serge Torres, who greatly helped computing the examples, and Paul Feautrier and Cédric Bastoul, whose expertise in polyhedric computations has been helpful.

REFERENCES

Ancourt, C. and Irigoin, F. 1991. Scanning polyhedra with do loops. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'91)*. ACM Press, New York, NY, 39–50.

Borwein, P. and Erdélyi, T. 1995. *Polynomials and Polynomials Inequalities*. Graduate Texts in Mathematics, 161. Springer-Verlag, New York, NY.

Collard, J.-F., Feautrier, P., and Risset, T. 1995. Construction of do loops from systems of affine constraints. *Parall. Process. Lett. 5*, 421–436.

Cornea, M., Harrison, J., and Tang, P. T. P. 2002. *Scientific Computing on Itanium-Based Systems*. Intel Press.

Feautrier, P. 1988. Parametric integer programming. *RAIRO Rech. Opér. 22*, 3, 243–268.

Feautrier, P. 2003. PIP/Piplib, a parametric integer linear programming solver. `http://www.prism.uvsq.fr/~cedb/bastools/piplib.html`.

Fox, L. and Parker, I. B. 1972. *Chebyshev Polynomials in Numerical Analysis*. Oxford Mathematical Handbooks. Oxford University Press, London, UK.

GRANLUND, T. 2002. GMP, the GNU multiple precision arithmetic library, version 4.1.2. http://www.swox.com/gmp/.

HABSIEGER, L. AND SALVY, B. 1997. On integer Chebyshev polynomials. *Math. Computat. 66*, 218, 763–770.

HART, J. F., CHENEY, E. W., LAWSON, C. L., MAEHLY, H. J., MESZTENYI, C. K., RICE, J. R., THACHER, H. G., AND WITZGALL, C. 1968. *Computer Approximations*. John Wiley & Sons, New York, NY.

LE VERGE, H., VAN DONGEN, V., AND WILDE, D. K. 1994. Loop nest synthesis using the polyhedral library. Tech. Rep. INRIA Research Report RR-2288, (May) INRIA.

MARKSTEIN, P. 2000. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall.

MULLER, J.-M. 1997. *Elementary Functions, Algorithms and Implementation*. Birkhäuser, Boston, MA.

PINEIRO, J., BRUGUERA, J., AND MULLER, J.-M. 2001. Faithful powering computation using table look-up and a fused accumulation tree. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (Arith-15)*. Burgess and Ciminiera Eds. IEEE Computer Society Press, Los Alamitos, CA, 40–58.

REMES, E. 1934. Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *C.R. Acad. Sci. Paris 198*, 2063–2065.

RIVLIN, T. J. 1990. *Chebyshev Polynomials. From Approximation Theory to Algebra* 2nd Ed. Pure and Applied Mathematics. John Wiley & Sons, New York, NY.

SCHRIJVER, A. 2003. *Combinatorial optimization. Polyhedra and efficiency. Vol. A*. Algorithms and Combinatorics, 24. Springer-Verlag, Berlin, Germany.

STORY, S. AND TANG, P. T. P. 1999. New algorithms for improved transcendental functions on IA-64. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*. Koren and Kornerup, Eds. IEEE Computer Society Press, Los Alamitos, CA, 4–11.

TANG, P. T. P. 1989. Table-driven implementation of the exponential function in IEEE floating-point arithmetic. *ACM Trans. Math. Soft. 15*, 2 (June), 144–157.

TANG, P. T. P. 1990. Table-driven implementation of the logarithm function in IEEE floating-point arithmetic. *ACM Trans. Math. Soft. 16*, 4 (Dec.), 378–400.

TANG, P. T. P. 1991. Table lookup algorithms for elementary functions and their error analysis. In *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*. P. Kornerup and D. W. Matula, Eds. IEEE Computer Society Press, Los Alamitos, CA, 232–236.

TANG, P. T. P. 1992. Table-driven implementation of the expm1 function in IEEE floating-point arithmetic. *ACM Trans. Math. Soft. 18*, 2 (June), 211–222.

THE POLYLIB TEAM. 2004. Polylib, a library of polyhedral functions, version 5.20.0. http://icps.u-strasbg.fr/polylib/.

THE SPACES PROJECT. 2004. MPFR, the multiple precision floating point reliable library, version 2.0.3. http://www.mpfr.org.

WEI, B., CAO, J., AND CHENG, J. 2001. High-performance architectures for elementary function generation. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (Arith-15)*. Burgess and Ciminiera Eds. IEEE Computer Society Press, Los Alamitos, CA, 136–146.

Received April 2004; revised May 2005; accepted September 2005