UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

STÉPHANE SIMARD

ÉTUDE DES COMPROMIS ESPACE/TEMPS DANS LES SYSTÈMES RECONFIGURABLES, POUR LA TECHNOLOGIE PSC

MÉMOIRE PRÉSENTÉ
COMME EXIGEANCE PARTIELLE
DE LA MAÎTRISE EN INGÉNIERIE

MAI 2006

Bibliothèque Paul-Émile-Boulet

UQAC Université du Québec à Chicoutimi

Mise en garde/Advice

Afin de rendre accessible au plus grand nombre le résultat des travaux de recherche menés par ses étudiants gradués et dans l'esprit des règles qui régissent le dépôt et la diffusion des mémoires et thèses produits dans cette Institution, **l'Université du Québec à Chicoutimi (UQAC)** est fière de rendre accessible une version complète et gratuite de cette œuvre.

Motivated by a desire to make the results of its graduate students' research accessible to all, and in accordance with the rules governing the acceptation and diffusion of dissertations and theses in this Institution, the **Université du Québec à Chicoutimi (UQAC)** is proud to make a complete version of this work available at no cost to the reader.

L'auteur conserve néanmoins la propriété du droit d'auteur qui protège ce mémoire ou cette thèse. Ni le mémoire ou la thèse ni des extraits substantiels de ceux-ci ne peuvent être imprimés ou autrement reproduits sans son autorisation.

The author retains ownership of the copyright of this dissertation or thesis. Neither the dissertation or thesis, nor substantial extracts from it, may be printed or otherwise reproduced without the author's permission.

Table des matières

Table des figures

Liste des tableaux

Résumé

Cette recherche a été effectuée dans le cadre du développement du langage psC, de la compagnie saguenéenne Novakod Technologies, un projet majoritairement subventionné par l'Agence spatiale canadienne, et auquel ont collaboré des chercheurs de l'Université du Québec à Chicoutimi.

Le langage psC est un nouveau langage, supporté par un compilateur et un environnement intégré de développement, visant à faciliter la programmation de systèmes en temps réel, parallèles et reconfigurables.

L'objectif de cette recherche était d'étudier, de réaliser et de comparer divers compromis espace-temps fondamentaux, dans le but de permettre au langage psC d'accélérer des calculs numériques habituellement réalisés par logiciel, au moyen de circuits reconfigurables, et à développer une bibliothèque d'opérateurs arithmétiques matériels, de fonctions logiques et de fonctions mathématiques répondant, dans la mesure du possible, aux besoins de psC dans le cadre de ce projet. Notre étude dresse, pour la première fois, un panorama général des mises en œuvres d'opérateurs arithmétiques sur FPGA et apporte plusieurs constats importants.

Une bibliothèque de composants arithmétiques et de fonctions élémentaires a été développée, testée, évaluée et livrée. Le développement de cette bibliothèque a donné lieu à une participation importante au déboguage et à l'amélioration des outils psC, qui étaient eux-mêmes en cours de développement.

Les résultats de notre recherche nous ont permis de découvrir une architecture simple et économique pour la mise en œuvre sérielle des opérations arithmétiques avec des opérandes et des résultats circulant bit de poids fort en tête. Cette architecture présente un intérêt pour la réalisation d'opérateurs arithmétiques à virgule flottante économiques.

Nous présentons une stratégie d'analyse pour l'estimation de la puissance de calcul d'un réseau d'opérateurs massivement parallèle.

Les résultats de cette étude seront repris, dans le cadre d'un doctorat, en vue de réaliser un contôleur sur puce haute-performance pour la commande de moteurs AC à induction.

Abstract

This research has been conducted within the framework of the development of the psC language, from Novakod Technologies, a Saguenay company, a project mainly subsidized by the Canadian Space Agency, and to which researchers from the University of Quebec at Chicoutimi have collaborated.

The psC language is a new language, supported by a compiler and an integrated development environment, aiming to facilitate real-time, parallel and reconfigurable systems programming.

The objective of this research was to study, implement, and compare various fundamental space–time tradeoffs, in order to allow the psC language to accelerate numerical computations usually carried out by software, by means of reconfigurable circuits, and to develop a library of hardware arithmetic operators, logic and mathematical functions answering, as far as possible, psC's needs in the context of this project.

Our study draws up, for the first time, a general panorama of arithmetic operator implementations on FPGA, and brings several significant findings.

A library of arithmetic components and elementary functions was developed, tested, evaluated and delivered. The development of this library have led to a significant participation in the debugging and the improvement of the psC tools, which were themselves under development.

Our research results enabled us to discover a simple and economic architecture for the serial implementation of arithmetic operations with the operands and results flowing in a most-significant-bit-fisrt manner. This architecture is certainly of interest for the realization of economic floating point operators.

The results of this study will further serve, in the context of a doctorate, in order to devise a high-performance system-on-chip AC induction motor controller.

Remerciements

Je remercie mon directeur et mon codirecteur de recherche, Luc Morin et Rachid Beguenane, pour leurs suggestions et leur support au cours de cette recherche, et la compagnie Novakod Technologies de nous avoir confié ce projet.

Un merci particulier va au professeur Beguenane, qui a consacré un soin et des ressources considérables afin de rehausser ma formation en micro-électronique ainsi qu'en recherche. Notamment, en sa qualité de représentant de CMC Microsystems pour l'UQAC, il m'a permis de participer à plusieurs séminaires et ateliers de formation organisés par CMC. Les gens de CMC font un travail magnifique, et leur support, le matériel et les outils de développement qu'ils fournissent aux chercheurs du domaine des micro-systèmes, encouragent et rehaussent des recherches telles que celle-ci.

Un merci particulier va également à Jean-Luc Beuchat, du projet Arénaire, pour ses conseils et sa collaboration, et pour m'avoir initié à l'arithmétique modulaire matérielle pour la cryptographie.

Le support financier de Novakod Technologies à l'Université du Québec à Chicoutimi, m'employant en tant qu'assistant de recherche du professeur Beguenane, au sein du groupe ERMETIS, a été crucial pour la réussite de ce projet.

Enfin, merci à mon collègue Éric Larouche pour ses conseils, aux membres du comité d'évaluation, D. Audet, R. Beguenane, O. Ait Mohamed, et L. Morin, et à tous ceux qui, de près ou de loin, ont contribué à la réussite de ce mémoire.

Chicoutimi, Québec — Stéphane Simard
Mai 2006

Liste des abréviations

ASIC	Circuit intégré à application spécifique (de l'anglais : Application-Specific Integrated Circuit)
BS	Borrow-save
C2	Complément à 2
CL	Cellule logique
CLB	Bloc logique configurable (de l'anglais : Configurable Logic Bloc)
CORDIC	COordinate Rotation DIgital Computer
DSP	Processeur de signaux numériques (de l'anglais : Digital Signal Processor)
FPGA	Réseau prédiffusé programmable par l'utilisateur (de l'anglais : Field-Programmable Gate Array)
GOPS	Milliards d'opérations par seconde (GOPS)
ITGE	Intégration/intégré à très grande échelle (anglais : VLSI)
LUT	Table de conversion matérielle (de l'anglais : Look-Up Table) 4-LUT : LUT à quatre entrées
MAC	Multiplieur–accumulateur
MEPS	Millions d'échantillons par seconde
MOPS	Millions d'opérations par seconde (MOPS)
MSBF	Bit de poids fort en tête (de l'angalis : most significant bit first)
MULT18x18	Bloc multiplieur ITGE embarqué 18×18 bits
PAM	Mémoire active programmable (de l'anglais : Programmable Active Memory)

PPOE	Programmation Parallèle Orientée Événements
psC	Parallel Synchronous C
RCAD	Retenue conservée–addition–décalage
RPAD	Retenue propagée–addition–décalage
RVM	Machine virtuelle Rodin (de l'anglais : Rodin Virtual Machine)
SIMD	Single Instruction Multiple Data
SISO	Shift-in, shift-out (registre à décalage sériel, utilisé comme simple délai)
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuit (circuit intégré pouvant fonctionner à très haute vitesse)

Chapitre 1

Introduction

1.1 Le langage psC

Le langage psC se veut le premier environnement de programmation entièrement parallèle basé sur un nouveau paradigme nommé *programmation parallèle orientée événement* (PPOE). Le langage et son paradigme découlent de la recherche du professeur Luc Morin, de l'Université du Québec à Chicoutimi (UQAC), président et fondateur de Novakod Technologies inc., une compagnie en démarrage, fondée en 2003, qui développe et commercialise désormais les outils de développement psC. En tant que professeur, M. Morin est, de plus, le directeur de la présente recherche, et le professeur Rachid Beguenane, également de l'UQAC, en est le codirecteur.

L'approche de psC, entièrement parallèle, consiste en un réseau de composants reliés par des fils via des ports d'entrée/sortie, et ne comporte pas les séquences d'instructions ni les boucles propres aux langages informatiques. Tout comme en conception numérique, les séquences et les boucles doivent être réalisées par des composants branchés en série ou par des machines à états finis. Puisque, en psC, le parallélisme est explicite, le concepteur a tout avantage à maîtriser l'algorithmique parallèle en plus de posséder au moins quelques notions de conception numérique.

Le langage se distingue cependant des langages de description matérielle en ce qu'il est entièrement synchrone et apporte des constructions et des types de données de plus haut niveau. Les différents blocs de code parallèle au sein des composants sont déclenchés par des *événements* arrivant sur les ports d'entrée déclarés *actifs*. Un signal d'événement est une impulsion circulant de port en port sur un fil implicite associé à toute donnée. Les ports actifs laissent passer les événements, alors que les ports *passifs* ne les laissent pas passer. Les événements transmis sur les ports de sortie actifs d'un composant sont gérés explicitement par le concepteur.

On trouve des explications plus détaillés dans le guide du programmeur psC [72].

1.2 Historique

L'origine de psC remonte à 1984, à l'Université Concordia, alors que M. Morin travaillait à la rédaction de sa thèse de doctorat en informatique intitulée : *A modeling technique for specification and simulation of digital systems.*

Dans les années 1990, une version modifiée du langage, connue sous le nom de Rodin, ayant perdu ses caractéristiques purement fonctionnelles tout en conservant sa syntaxe postfixe, vise désormais à faciliter la programmation des systèmes concurrents et en temps réel sur micro-processeur.

L'approche par composants parallèles orientée événements a été conçue en 1997. Le parallélisme était alors simulé par une machine virtuelle (la *Rodin Virtual Machine* — RVM) qui interprétait un code machine compilé. Par la suite, le langage a évolué lentement grâce à des projets confiés à des étudiants de l'UQAC.

En janvier 2003, M. Morin reçoit une première subvention de l'Agence Spatiale Canadienne, dans le cadre du Programme de développement des technologies spatiales (projet PDTS-2003), donnant le coup d'envoi d'une première phase des développements qui vont conduire au projet actuel. À ce stade, le langage ne comporte que des types de données entiers. Un groupe d'étudiants est embauché dans le cadre de stages/projets au Laboratoire de développement de logiciels (LDL) de l'UQAC afin d'ajouter d'une vaste gamme de nouveaux types et d'apporter diverses améliorations. Concurremment, un stage/projet du présent auteur donne lieu à un travail de recherche menant à la conception et au développement d'exemples d'applications du langage Rodin à l'algorithmique parallèle et à la commande en temps réel.

Suite aux résultats obtenus, M. Morin décide de commercialiser le produit. Il entame une procédure de prise de brevet et fonde, à Chicoutimi, la compagnie Novakod Technologies, embauchant 4 des étudiants du LDL, désormais gradués.

Il est important de souligner les efforts de tous ceux qui ont contribué, de près ou de loin, et ce, parfois gracieusement, aux travaux de recherche et de développement qui ont conduit à cette commercialisation. La liste est longue, mais, parmi les contributeurs connus du présent auteur, il convient de mentionner, notamment, ceux dont la contribution a immédiatement précédé la commercialisation (par ordre alphabétique) : François Blackburn, Jean-Michel Fortin, Bruno Gagnon, Michel Héraud, Éric Larouche, Patrick Latour, Jean-François Michaud, l'auteur du présent mémoire, Stéphane Simard, Simon Tremblay, ainsi qu'une foule d'autres contributeurs.

En 2004–05, Novakod obtient un contrat majeur de l'Agence Spatiale Canadienne, et le projet actuel, que l'on conviendra d'appeler simplement le Projet, est lancé. Au cours du Projet, le compilateur est réécrit en entier pour introduire une nouvelle syntaxe, proche de celle du langage C. Le langage subit alors une transformation radicale et devient psC (Parallel Synchronous C). L'objectif visé est désormais de faciliter la programmation des systèmes reconfigurables à base de puces FPGA, et l'environnement de développement psC, appelé Novakod Studio, entend supporter les applications parallèles et en temps réel tant sur les plateformes à FPGA que celles à micro-processeurs.

1.3 Les FPGA

Les FPGA (de l'anglais : *field-programmable gate arrays*), ou *réseaux prédiffusés programmables,* se composent d'un grand nombre de blocs logiques combinatoires, mis en œuvre par des multiplexeurs, des tables de conversion et des registres, pouvant être reprogrammés et interconnectés les uns aux autres un nombre arbitraire de fois dans n'importe quelle configuration. Une fois conçues les fonctions logiques de configuration, la procédure d'adaptation est très rapide et peu coûteuse comparativement au processus traditionnel de conception sur mesure d'un circuit intégré.

Les structures de calcul à grain fin présentes dans les FPGA conviennent bien à la réalisation matérielle d'algorithmes pouvant exploiter un parallélisme massif au niveau du bit ou s'adaptant bien à des architectures hautement pipelinées. La flexibilité architecturale des FPGA permet aussi aux concepteurs d'ajuster les largeurs de données afin de s'adapter à différents besoins d'un même algorithme. Les progrès récents ont d'ailleurs fait des FPGA des plateformes idéales pour la mise en œuvre du calcul reconfigurable, qui consiste à adapter sur le vif un circuit à des calculs complexes.

Réalisés sur FPGA, des calculateurs dédiés et des machines à structure massivement parallèle d'architecture SIMD (de l'anglais : *single instruction, multiple data*), capables de traiter une instruction sans affectation préalable des processeurs de traitement, peuvent potentiellement atteindre des performances du calibre d'un superordinateur pour un certain nombre d'applications spécifiques, et ce, à seulement une fraction du prix [18]. Ces calculateurs massivement parallèles peuvent exploiter les avantages d'une architecture sérielle afin d'obtenir un pipeline à grain très fin tout en minimisant la taille des éléments individuels de traitement. Comme les architectures sérielles tendent à faire grand usage de registres tout en ne nécessitant que peu de ressources de routage, elles produisent habituellement des réalisations sur FPGA à la fois économiques et relativement rapides et efficaces.

1.4 Problématique

L'arithmétique sur FPGA, à cause des contraintes technologiques, a longtemps été restreinte aux opérations sur les entiers et aux opérations sur les nombres fractionnaires à virgule fixe. Au stade actuel du progrès technologique, grâce à leur architecture configurable au niveau du bit, il est devenu possible de réaliser efficacement sur FPGA une vaste gamme d'opérations arithmétiques et de fonctions mathématiques.

Les opérateurs arithmétiques se classifient en deux grands types d'architectures : les architectures parallèles et les architectures sérielles. Dans les microprocesseurs courants, les opérations arithmétiques de base s'effectuent en parallèle, et tous les bits des opérandes sont traités en même temps. À l'opposé, lorsque l'on effectue des calculs sur papier, on procède de manière *sérielle*, c'est-à-dire chiffre à chiffre, en progressant soit de droite à gauche (chiffres de poids faible en tête), soit de gauche à droite (chiffres de poids fort en tête), selon la méthode employée.

Lorsqu'une application particulière nécessite une grande précision ou un grand nombre d'opérateurs de calcul, la surface occupée par un module dans un circuit peut devenir un critère aussi important, sinon plus, que sa vitesse. L'arithmétique sérielle est bien adaptée à de telles applications, car elle a l'avantage d'économiser l'espace et de minimiser les interconnexions, les chiffres étant calculés et émis un à un. Dans d'autres applications, par contre, où c'est la vitesse d'un opérateur isolé qui importe, le recours à une mise en œuvre parallèle, habituellement coûteuse, peut s'avérer le seul moyen d'atteindre la vitesse requise.

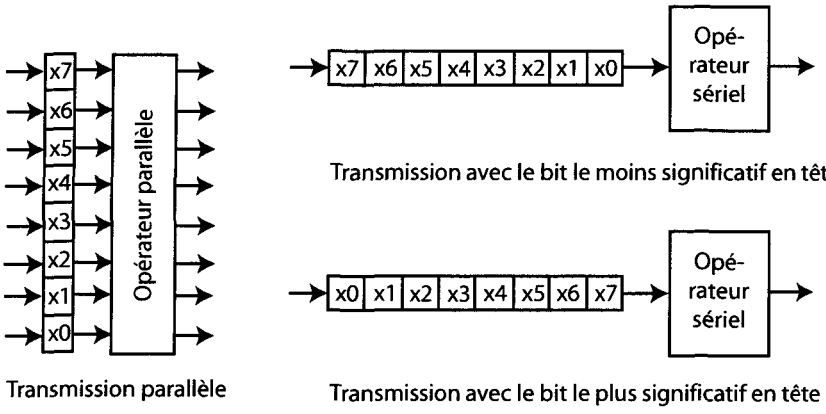

FIG. 1.1 – Transmission parallèle versus sérielle

En arithmétique parallèle, les bits des opérandes et des résultats circulent en parallèle, ce qui permet, bien qu'à un coût élevé, d'obtenir une haute performance, car

les modules arithmétiques parallèles occupent un espace considérable sur une puce et il est par conséquent nécessaire de les réutiliser au maximum. En arithmétique sérielle, les opérandes et les résultats circulent en série, chiffre à chiffre, à travers les modules. La direction de circulation des chiffres engendre les deux *modes d'opération* sériels déjà mentionnés, selon que la circulation s'effectue chiffre de poids faible (de droite à gauche) ou chiffre de poids fort (de gauche à droite) en tête (Fig. 1.1).

Additionner sériellement deux nombres chiffre de poids faible en tête est particulièrement simple. Il faut cependant tenir compte du fait que la retenue d'une addition se propage toujours de droite à gauche et que le contraire n'est pas possible. À mesure que chaque paire d'opérandes est reçue, il suffit de calculer un chiffre de somme et un autre de retenue. Le chiffre de retenue est ensuite additionné à la prochaine paire d'opérateurs, et ainsi de suite. La réalisation d'un tel additionneur ne nécessite qu'un simple additionneur complet et une bascule. La multiplication est essentiellement une somme de produits partiels, et, étant donné le sens de la propagation de retenue dans les additions, elle, fonctionne, elle aussi, naturellement chiffre de poids faible en tête. Le mode de calcul sériel chiffre de poids faible en tête semble donc, a priori, être une manière simple, rapide et économique de calculer, car elle procède dans le même sens que la propagation des retenues.

Si l'arithmétique sérielle présente certains avantages, en particulier parce qu'elle offre une économie d'espace considérable, son utilisation et ses performances sont toutefois limités par deux caractéristiques fondamentales contradictoires, contradiction qui est à l'origine de difficultés majeures ayant entraîné des décennies de recherches plus ou moins fructueuses, sachant bien que l'on ne pourrait jamais arriver qu'à des solutions de compromis. Effectuer les opérations d'addition et de multiplication dans le sens contraire de la propagation des retenues entraîne des complications qui affectent

considérablement le coût et la performance des mises en œuvre. Or, il n'est pas possible d'effectuer les opérations de division et d'extraction de la racine carrée en commençant par les chiffres les moins significatifs, car les chiffres les moins significatifs du résultat dépendent des chiffres les plus significatifs des opérandes. Il en résulte une incompatibilité intrinsèque, en arithmétique sérielle, entre les modules de la famille de l'addition et de la multiplication, d'un côté, et, de l'autre, les modules de la famille de la division et des fonctions élémentaires, dont l'extraction de la racine carrée.

Dans certaines applications, telles que le traitement de signal, où seuls des additionneurs, des soustracteurs, des multiplieurs et des modules d'élévation au carré sont nécessaires, des calculs chiffre de poids faible en tête peuvent s'avérer intéressants [32].

Il faut cependant considérer le fait que le résultat d'une multiplication de deux nombres de N chiffres comporte $2N$ chiffres, de sorte que le nombre de chiffres du produit final croît de manière exponentielle lorsque l'on fait des multiplications répétées. Dans le cas d'un multiplieur sériel produisant les chiffres les moins significatifs en premier (chiffre de poids faible en tête), il faut attendre au moins $2N-1$ cycles d'horloge, avant que le chiffre le plus significatif du produit ne soit émis en sortie, ce qui nécessite une mise en tampon durant tout le processus.

Si l'on fait abstraction des problèmes de compatibilité entre différents types d'opérateurs, un traitement en série est beaucoup plus économique en espace qu'un traitement en parallèle, mais prend, en contrepartie, un plus grand nombre de cycles d'horloge pour effectuer une opération isolée. La performance de différents réseaux de modules parallèles varie beaucoup comparativement à celle de réseaux sériels équivalents, et peut même, dans certains cas, être surpassée par un réseau sériel, pourvu que ce dernier ait une profondeur suffisante et que la latence des modules sériels ainsi cascadés soit relativement faible.

Comme les modules sériels tendent à faire grand usage de registres tout en ne demandant que peu de ressources de routage, ils donnent habituellement lieu à des réalisations sur FPGA à la fois efficaces et rapides. Par la reconfiguration de leur degré de sérialisation, les modules sériels peuvent s'adapter aux exigeances de diverses applications en termes de performances ou de ressources, ainsi qu'aux variations des conditions d'opération ou aux différentes phases de calcul d'une même application. Le choix de la largeur de chiffre influe aussi grandement sur le coût et les performances de la mise en œuvre.

Pour améliorer la fréquence d'horloge maximale d'opérateurs parallèles, qui tendent, de par leur nature même, à comporter un long chemin critique, on a généralement recours à une technique nommée *pipelinage*, qui consiste à insérer des étages de bascules au sein d'un circuit afin de scinder en tranches plus courtes le long chemin combinatoire qui le traverse. Le principal intérêt du pipelinage est d'obtenir un fort débit de calcul. Un module ainsi pipeliné aura une latence initiale de cycles d'horloge égale au nombre d'étages de pipeline, mais produira ensuite à chaque cycle, à fréquence élevée, un résultat en parallèle à sa sortie.

Du côté des opérateurs sériels, puisqu'ils commencent à calculer des chiffres du résultat d'après une entrée partielle des chiffres des opérandes, ceux-ci peuvent être pipelinés au niveau du chiffre, permettant ainsi, dans certains cas, de réduire la latence lors du calcul d'expressions composées, comparativement à une mise en œuvre parallèle.

L'arithmétique en-ligne est une arithmétique sérielle ayant la particularité d'effectuer les calculs en commençant par les chiffres les plus significatifs (chiffre le plus significatif en tête). Elle favorise le pipelinage des opérations arithmétiques, qui peuvent alors s'effectuer simultanément, à mesure que chaque chiffre devient disponible, permettant ainsi de réduire le temps de calcul de longues séquences d'opérations tout en minimisant les interconnexions entre modules. Elle tente, de plus, de solutionner les problèmes

mentionnés ci-dessus découlant du sens de propagation des retenues et de l'incompatibilité des deux différents groupes d'opérateurs arithmétiques entre eux. Nous verrons plus loin que, malgré un arsenal de sophistications théoriques imposant, l'arithmétique en-ligne n'atteint ce dernier objectif que partiellement.

1.5 Objectifs et méthodologie de recherche

Le Projet commandait l'étude des compromis espace–temps dans les systèmes reconfigurables en vue d'ouvrir la voie au développement de modules arithmétiques et de fonctions mathématiques pour FPGA dans le langage psC. En supportant, en psC, la synthèse de réseaux d'arithmétique dédiés sur FPGA, la compagnie Novakod entend permettre d'accélérer des algorithmes tournant habituellement sur microprocesseur. Il ne s'agit pas là, cependant, de synthèse directe à partir d'une description algorithmique classique, mais plutôt d'une approche s'apparentant à la conception numérique.

Cette recherche visait à étudier, à mettre en œuvre et à comparer des compromis espace-temps fondamentaux, dans le but de comprendre comment accélérer des calculs numériques grâce à des mises en œuvre sur FPGA, et à développer un ensemble d'opérateurs arithmétiques, de fonctions logiques et de fonctions mathématiques de base.

La liste d'opérateurs arithmétiques à développer et les choix de conception ont fait l'objet de discussions dans le cadre de multiples réunions, et reflètent strictement les directives reçues, fondées sur la base des exigences et des contraintes du Projet. Ces réunions se sont généralement déroulées en présence d'un représentant de la compagnie, du directeur de la compagnie et directeur de recherche à l'UQAC, ainsi que du codirecteur de recherche et superviseur du Projet pour l'UQAC.

La cueillette des données s'est déroulée concurremment au développement. Les données consistent en des mesures permettant d'évaluer et de comparer le coût en temps et

en espace des diverses mises en œuvre. Ces mesures sont basées sur des caractéristiques architecturales et sur des résultats de synthèse fournis par les outils de dévelopmment commerciaux, incluant, pour chaque opérateur arithmétique ou fonction élémentaire, sa latence (en tops d'horloge), son débit maximal (en MHz) et son coût en cellules logiques (CL).

Dans une première étape, une revue du domaine de l'arithmétique des ordinateurs a été effectuée, avec emphase sur les réalisations sur FPGA. Cette revue a démontré, dans un premier temps, que la mise en œuvre d'opérateurs arithmétiques et de fonctions élémentaires parallèles et en série sur FPGA avait suffisamment d'assises dans la littérature et était réalisable au stade actuel du progrès technologique.

Outre l'apport d'une revue du domaine, une part importante de la contribution du présent travail consiste dans la mise en œuvre et la comparaison de plusieurs architectures d'opérateurs arithmétiques et de fonctions élémentaires, constituant, dans le cadre du Projet, une bibliothèque de modules optimisés pour FPGA et codés en psC.

Les critères que nous définissons pour évaluer la performance et faire des comparaisons entre différentes mises en œuvre, sont les suivants :

- *Uniformité :* La capacité d'interconnecter des modules individuels sans avoir à introduire des structures intermédiaires pour reformater les données ;
- *Espace :* Le nombre d'unités physiques occupées par un module sur une puce. Pour les circuits réalisés sur FPGA étudiés dans le présent mémoire, l'espace est mesuré en cellules logiques (CL), constituées chacune d'une table de vérité (LUT) à 4 entrées, d'une bascule D et d'éléments logiques destinés à assurer la propagation d'une retenue ;
- *Débit :* Le débit d'un module est défini comme le nombre d'ensembles d'entrées qu'il peut traiter par unité de temps ;

- *Latence :* La latence est le nombre d'unités de temps entre l'application d'entrées et l'obtention d'une sortie. Dans la littérature sur l'arithmétique en-ligne, la latence est souvent exprimée sous forme de *délai en-ligne*, δ, mais la définition du délai en-ligne n'est pas compatible avec d'autres types d'arithmétique. Pour les mises en œuvre sérielles, nous définissons la latence comme le temps, en cycles d'horloge, séparant l'entrée du premier bit des opérandes de la sortie du dernier bit du résultat.

Chapitre 2

Revue de l'arithmétique des FPGA

Les FPGA sont devenus, au cours des dernières années, l'option favorite pour la mise en œuvre matérielle des systèmes numériques, car ils permettent d'adapter le matériel à une application donnée. Puisque les opérations arithmétiques sont les constituantes fondamentales des systèmes numériques, les spécificités de leur mise en œuvre dans la logique programmable a fait l'objet, jusqu'à présent, de recherches approfondies, et de nombreuses propositions ont été avancées en vue d'atteindre une haute performance pour des applications spécifiques. J.-L. Beuchat et A. Tisserand ont rédigé, en 2004, un chapitre de livre sur la mise en œuvre des opérateurs arithmétiques sur FPGA [26].

Il convient de remarquer que les modèles de délais et les analyses de coût développés pour la conception de circuits intégrés à application spécifique (ASIC) ne sont pas utiles pour la conception et la réalisation de dispositifs sur FPGA [47, 108]. Si les unités de mesures utilisées en conception ITGE sont le nombre de portes logiques et leur délai, des mesures de performance données en termes du nombre d'éléments logiques configurables et des délais logiques et de routage sont plus appropriées à la terminologie FPGA.

2.1 Arithmétique parallèle

Les contributions à l'arithmétique parallèle sont abondantes, et il est bon de consulter un ouvrage de référence tel que [40] pour en avoir un panorama complet.

R. Zimmermann [109] a développé, en VHDL, une bibliothèque de modules arithmétiques parallèles paramétrisables. Cette bibliothèque, indépendante de la plateforme, est dans le domaine public et elle fournit, pour ASIC, des circuits dont la performance est comparable aux bibliothèques commerciales, mais avec une plus grande flexibilité et une plus grande diversité d'opérateurs arithmétiques. Cette bibliothèque se limite toutefois aux opérations sur les entiers et n'est pas optimisée pour FPGA avec logique de propagation de retenues dédiée.

D'autres mises en oeuvres de modules d'arithmétique parallèle peuvent être obtenues sur le site http://www.opencores.org et d'une multitude d'autres sources. On en trouve plusieurs dans les références données ci-après.

2.2 Arithmétique sérielle

Les modules d'arithmétique sérielle ont une structure très simple et économique en espace. Les opérandes et les résultats des calculs circulent en série, chiffre à chiffre, à travers les modules. Un traitement sériel nécessite un plus grand nombre de cycles d'horloge qu'un traitement parallèle pour effectuer une opération, mais, excepté dans le cas de l'utilisation de blocs arithmétiques intégrés, un flux sériel épouse mieux la structure logique des FPGA [79].

L'approche sérielle conventionnelle, qui consiste à effectuer les calculs en commençant par les chiffres les moins significatifs, semble être une manière simple et rapide de calculer. Cependant, il existe une difficulté majeure : il n'est pas possible d'effectuer les opérations de division et de racine carrée en commençant par les chiffres les moins significatifs, car les chiffres les moins significatifs du résultat dépendent des chiffres les plus significatifs des opérandes. Il serait donc nécessaire de mémoriser tous les chiffres des opérandes jusqu'au dernier avant de pouvoir commencer à calculer le quotient. Si cet inconvénient, ajouté à l'incompatibilité intrinsèque qui existe entre l'addition et la multiplication, d'une part, et la division et la racine carrée, d'autre part, rend l'arithmétique sérielle bit de poids faible en tête non généralisable, cette dernière est néanmoins d'un usage facile, elle est compacte, et convient bien aux applications de traitement de signal, où seuls des additionneurs/soustracteurs, des multiplieurs et/ou des opérateurs d'élévation au carré sont requis [32].

L'approche *en-ligne,* introduite par M.D. Ercegovac en 1977 [91], a la particularité

d'effectuer les calculs dans la direction opposée, en commençant par les chiffres les plus significatifs, ce qui est possible en ayant recours à une représentation numérique spéciale, dite *redondante*, permettant de ne pas avoir à propager les retenues. Il est possible d'effectuer en-ligne, de cette manière, une succession de plusieurs opérations reliées par un *pipeline de chiffres*. Le temps de calcul total du pipeline est alors la somme des temps de calcul des modules individuels, et les interconnexions entre modules sont minimales. L'approche en-ligne apporte une solution, bien qu'imparfaite, aux problèmes reliés à la compatibilité de l'addition et de la multiplication avec la division et la racine carrée [40,77].

Les opérateurs sériels conventionnels utilisent une représentation numérique binaire standard et fonctionnent en mode chiffre de poids faible en tête. L'un des principaux intérêts de l'utilisation de ce type d'opérateurs vient du fait qu'un additionneur *bit-sériel* (Fig. 2.3) peut fonctionner à très haute fréquence, et que sa taille ne change pas selon la longueur des opérandes. De plus, les multiplieurs bit-sériels sont les composants les plus critiques en traitement de signal haute fréquence, et sont aussi requis dans la plupart des applications parallèles.

L'un des premiers modules d'arithmétique bit-sérielle à avoir été proposé pour FPGA, en 1993, était systolique et utilisait une approche hybride où les bits d'un opérande entrent, à chaque deux cycles, en commençant au cycle 1, dans le tableau linéaire par la cellule d'extrême gauche, bit le moins significatif en tête, et les bits du second opérande entrent de manière similaire, en commençant au cycle 2, par la cellule d'extrême droite, bit de poids fort en tête [82]. Un tel module est économique au niveau de l'opérateur isolé, mais, à cause de son flux de données bidirectionnel en directions contraires, souffre d'un manque de régularité pour une utilisation en système.

L'objectif de ce chapitre est de faire une revue de l'état actuel de l'art de l'arith-

métique pour FPGA. Sont couverts, les opérateurs entièrs d'addition/soustraction, de multiplication, d'élévation au carré, de division et de racine carrée, en parallèle et dans les deux modes sériels (chiffre de poids faible et chiffre de poids fort en tête).

2.3 Opérateurs entiers

2.3.1 Additionneurs et soustracteurs

Les FPGA modernes comportent des circuits de propagation de retenues très rapides permettant à une cellule logique de fonctionner comme un additionneur complet. Ces circuits n'occupent pas d'espace dans la logique standard, puisqu'ils sont réalisés à l'extérieur des LUTs [1,29,101]. Bien que la vitesse des additions sur FPGA soit toujours limitée par les circuits de propagation des retenues, ces circuits sont si rapides et efficaces comparativement à la logique à base de tables de vérité et aux délais d'interconnexion que les méthodes d'accélération traditionnelles utilisées en conception de circuits ITGE deviennent inutiles et coûteuses.

À cause de leur coût élevé, de leur complexité, de leurs structures irrégulières, de leur entrance et de leur sortance élevées et du fait qu'ils sont incapables de profiter de la logique de propagation de retenue dédiée, les additionneurs conventionnels à retenue anticipée (*carry-lookahead* et à retenue sélectionnée (*carry-select*), qui donnent habituellement les mises en œuvre les plus rapides en conception ITGE, ne conviennent pas à une réalisation sur FPGA.

Dans une étude de 1998, Xing et Yu [108] ont comparé des additionneurs à retenue complétée *carry-complete* et à retenue ignorée and *carry-skip* optimisés pour FPGA, mais n'ont constaté aucune amélioration du délai pour des additionneurs de moins de 48 et 56 bits, respectivement, et ce, malgré une augmentation considérable de la

surface occupée. Selon cette étude, ce sont les additionneurs à retenue propagée qui ont le plus faible coût et le meilleur rapport performance/coût, à cause de leur structure très régulière et de leur utilisation efficace de la logique de propagation de retenues dédiée. Pour la réalisation d'additionneurs de plus de 48 bits, l'étude conclut que des additionneurs optimisés de type sélection–propagation–propagation (*select-ripple-ripple — (S-R-R)*) semblent être le meilleur choix lorsqu'un meilleur temps de fonctionnement est désiré, moyennant une augmentation raisonnable du coût en espace. Par ailleurs, une nouvelle structure d'additionneur à retenue ignorée, donnant des mises en oeuvre FPGA efficientes, a été introduite par Kantabutra et al. en 2002 [49].

Les soustracteurs se construisent à partir d'additionneurs avec retenue entrante, dont le second opérande passe par un circuit de complémentation (complément à 1). On peut construire un additionneur/soustracteur combiné en faisant en sorte qu'un même signal de sélection serve à la fois de retenue entrante et désactive (0) ou active (1) la complémentation. Selon que le signal de sélection vaille 0 ou 1, le circuit effectue, respectivement, une addition ou une soustraction.

Additionneurs standards

Plusieurs études [35, 47, 55, 108] ont confirmé le fait que sur FPGA, il convenait d'effectuer l'addition de manière standard, en utilisant des structures à propagation de retenue plutôt que des additionneurs redondants à retenue conservée *(carry-save adders)* (Fig. 2.1). D'autre part, il ressort de ces études, qu'étant donné que le routage a une influence significative sur le coût de l'addition, il y a une différence notoire dans la méthode appropriée pour pipeliner des aditionneurs pour ASIC que pour FPGA.

Pour la conception d'ASIC, afin de subdiviser un additionneur en segments plus couts, les bascules de pipeline devraient être insérés à l'intérieur de l'additionneur à des

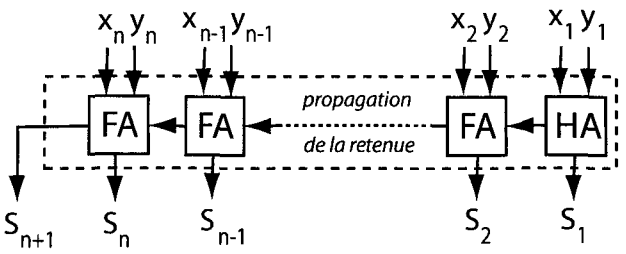

FIG. 2.1 – Additionneur à retenue propagée standard

intervalles réguliers de cellules logiques.

Sur FPGA, cette technique n'est utile que pour réduire le délai des additionneurs larges. Autrement, les bascules ne devraient être insérées qu'à des intervalles réguliers entre les additionneurs. Cette méthode a, de plus, l'avantage d'utiliser les bascules présentes à l'intérieur des cellules logiques déjà allouées à un additionneur sans augmenter sa taille ou sa complexité.

Additionneurs larges

Pour des additionneurs larges, Jamro et Wiatr [47] ont proposé une solution hybride qui consiste à la fois à diviser les additionneurs larges en de plus petits segments, comme dans la méthode ASIC, et d'insérer des bascules de pipeline à des intervalles réguliers entre additionneurs (Fig. 2.2). Ils ont suggéré que cette solution aparaissait meilleure que la technique d'addition retardée introduite par Luo et Martonosi en 1998 [62], qui utilise des arbres de Wallace pour accumuler des valeurs sans effectuer une propagation de retenue complète.

Pour des applications demandant une plus grande vitesse, telles que la cryptographie, une méthode alternative rapide pour la réalisation d'additionneurs larges sur FPGA de la famille Virtex a été présentée dans un article en 2002 par Perri et al. [78]. Cette méthode est aussi utile pour les FPGA des familles XC4000X, Spartan et Spartan-

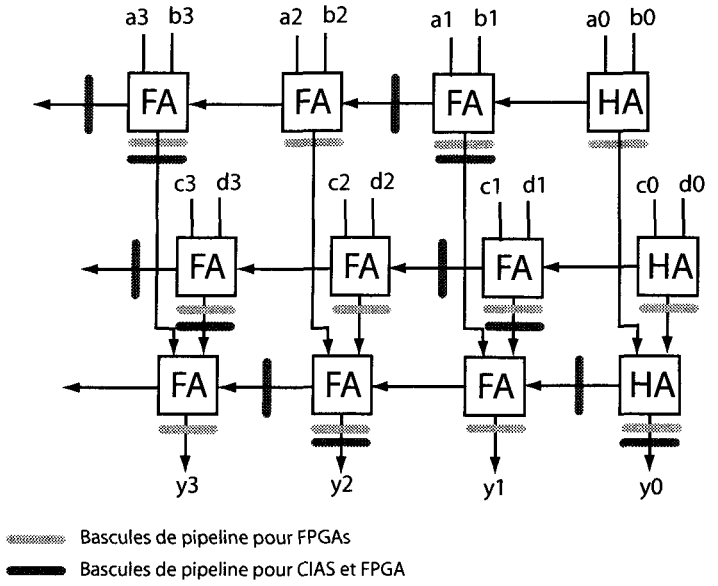

FIG. 2.2 – Pipelinage des additionneurs

XL. Les auteurs ont utilisé des éléments logiques de contournement pour éliminer les délais de routage difficiles du chemin critique. Pour un additionneur 128 bits réalisé par cette méthode, les auteurs rapportent un gain de performance allant jusqu'à 47 % comparativement à un additionneur à propagation de retenue conventionnel, avec un surcoût en espace de 28 %.

Additionneurs et soustracteurs bit-sériels

L'addition sérielle de deux nombres est particulièrement simple. À mesure que chaque paire d'opérandes est reçue, il suffit de calculer un chiffre de somme et un chiffre de retenue, puis le chiffre de retenue est additionné avec la prochaine paire d'opérandes, et ainsi de suite. La réalisation d'un tel additionneur ne requiert rien de plus qu'un additionneur complet et une bascule (Fig. 2.3).

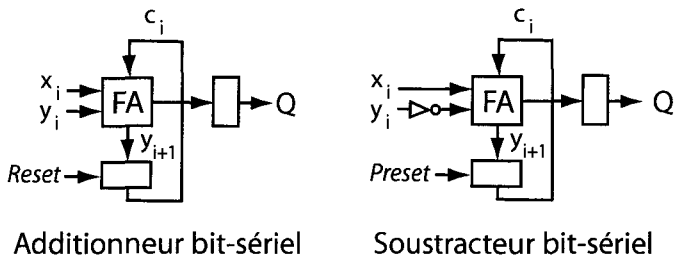

FIG. 2.3 – Additionneur et soustracteur bit-sériels

Le principal intérêt de l'utilisation d'un additionneur bit-sériel est qu'il peut fonctionner à très haute fréquence et que sa taille, d'environ deux cellules logiques sur FPGA, ne varie pas en fonction de la longueur des opérandes.

2.3.2 Multiplieurs

La multiplication est l'une des opérations les plus complexes et les plus critiques ayant à être réalisées sur FPGA. Comparativement aux ASIC, les FPGA présentent l'avantage d'être reconfigurables, mais, pour plusieurs types de multiplieurs, ils sont beaucoup plus lents que des puces ITGE conçues sur mesure. Une spécialisation et une concurrence accrues sont donc nécessaires pour obtenir un quelconque gain de performance par rapport aux processeurs de signaux numériques (DSP) et aux ASICs [79]. Il existe cinq principales structures de multiplieurs :

Décalage et addition — Un opérande est décalé à gauche d'un bit par cycle et est appliqué à un additionneur lorsque le bit correspondant du second opérande est haut. Cette technique donne des mises en œuvre compactes, mais très lentes, nécessitant N cycles d'horloge pour se compléter, et ne permet pas le chevauchement des calculs.

Arbre logique – Chacun des bits résultants sont une fonction logique des bits associés de chaque opérande. Cela donne une grande structure à entrance élevée, envisageable jusqu'à 32 bits (Fig. 2.4).

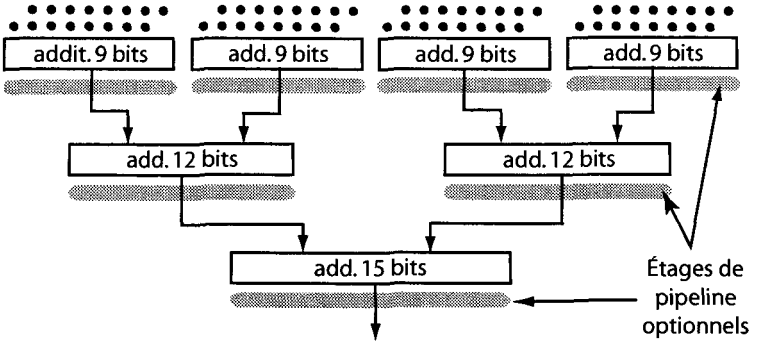

FIG. 2.4 – Multiplieur 8 bits sous forme d'arbre d'additionneurs à propagation de retenues

Structures en tableaux – Des tableaux parallèles donnent des multiplieurs rapides, avec une délai minimum, mais donne des mises en œuvre coûteuses, une majeure partie de la logique étant inactive au cours d'un calcul.

Tables de conversion – Les opérandes sont appliqués comme adresses à une mémoire pré-programmée qui donne le résultat en sortie. Cette technique est l'une des plus rapides, mais son coût en espace est si élevé qu'elle est impraticable pour des multiplieurs plus grands que 4×4. Par exemple, une mise en œuvre 16×16 nécessiterait une mémoire de $2^{16} \times 2^{16} \times 32$ bits $= 4\,\mathrm{G} \times 32$ bits. On peut, notamment, utiliser les blocs multiplieurs ITGE incorporés dans plusieurs FPGA récents. Lorsqu'ils sont disponibles, leur utilisation est parmi les manières les plus rapides d'effectuer des multiplications sur FPGA, et, de plus, elle n'utilise que peu d'éléments logiques.

Multiplieurs à base de tables

Les premiers multiplieurs entiers à base de tables pour FPGA, rapides et relativement compacts, ont été introduits en 1993–94 par l'ingénieur de Xilinx K. Chapman, pour l'architecture XC4000 [28,29]. La solution idéale proposée dans cette contribution était de nature hybride, et consistait à utiliser de petites tables de valeurs pour les produits

partiels et à combiner les résultats par addition. L'idée de base était de subdiviser le premier opérande en sections, à multiplier par consultation de table chacune des sections avec le second opérande au complet et d'utiliser un arbre d'additionneurs pour additionner les produits partiels.

Les multiplieurs à base de tables ont, par la suite, été étudiés dans plusieurs autres travaux, dont [48, 53, 70, 81, 98]. Il convient notamment de souligner la contribution de Miomo et al. [70] qui ont présenté, dans FPL 2000, un multiplieur très rapide, mais aussi de taille considérable, pour FPGA munis de LUT à 4 entrées. Ce concept utilise une méthode de compaction basée sur une représentation numérique redondante. Les auteurs en ont comparé une mise en œuvre 16 bits, avec compaction sur 4 bits, avec le plus récent multiplieur 16 bits fourni par Xilinx à l'époque et ont rapporté un gain en vitesse de trois fois celle obtenue par Xilinx, moyennant un accroissement par cinq de la surface du circuit.

Multiplieurs à coefficient constant

Les multiplieurs à coefficient constant sont économiques sur FPGA, à la fois en espace et en temps. La multiplication par une constante se présente dans un grand nombre d'applications, incluant les fonctions d'amplification, les contrôleurs numérique, la conversion d'espaces de couleurs et le filtrage numérique. Un multiplieur à coefficient constant n'utilise généralement qu'un quart à un tiers de la taille d'un multiplieur optimisé pour l'espace et est au moins aussi rapide qu'un multiplieur optimisé pour la vitesse.

Le concept consistant à utiliser une structure spéciale pour la multiplication par une constante sur FPGA a été introduite pour la première fois chez Xilinx, en 1996, par K. Chapman [30] et des collègues [50].

Sur FPGA à base de LUTs, les mises en œuvre de multiplieurs à coefficient constant

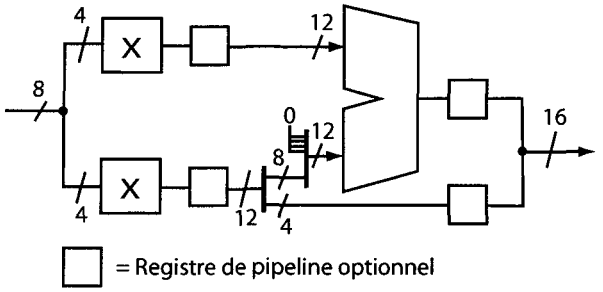

FIG. 2.5 – Un multiplieur 8 bits par une constante

bénéficient du fait que les LUTs peuvent être utilisées comme des mémoires à lecture seule. Une LUT à 4 entrées, par exemple, peut être utilisée comme une ROM 16×1. Steve Knapp (1998) a adapté cette approche pour maximiser l'utilisation des LUTs à 4 entrées en travaillant en hexadecimal (base 16) et en utilisant un arbre d'additionneurs pour des entrées plus larges [52].

Une méthode pour réduire le temps de conception de multiplieurs-additionneurs rapides à coefficient constant pour le traitement de signaux numériques a été décrite dans [53].

En 2002, Wiatr et Jamro ont presenté une étude comparative de deux techniques de multiplication à coefficient constant [48] : la multiplication sans multiplieur (à décalage addition/soustraction) (SM) utilisant une représentation numérique à chiffres signés canonique (CSC) et la multiplication à base de LUTs par table de conversion (ML). Cette étude a aussi introduit un nouvel algorithme pour la conversion de nombres en complément à deux en représentation CSC. Une représentation CSC contient approximativement 33 % moins de bits différents de zéro que sa contrepartie binaire, se traduisant par des gains du même pourcentage par coefficient. ML, en théorie, donne les mises en œuvre les plus rapides, étant donné qu'aucune arithmétique proprement dite n'est requise, mais la taille des tables de conversion croît rapidement à de grandes

dimensions avec la largeur des opérandes. En conséquence, les auteurs arrivent à la conclusion que ML est habituellement la structure la plus efficace lorsque la largeur des opérandes et des coefficients est petite (moins de 5) et que, dans tous les autres cas, SM est généralement plus économique. En 2001–2002, Jamro and Wiatr ont étendu leur étude à des structures de multiplieurs à coefficient constant visant la réalisation de la convolution sur FPGA [46, 48].

Arithmétique distribuée

L'*Arithmétique distribuée* (DA) a traditionnellement joué un rôle clé dans les systèmes de traitement de signaux numériques à base de FPGA, car elle fournit une méthode de conception à base de LUT sans multiplieur, appelée DALUT. L'approche DA est spécifiquement destinée au calcul de la somme des produits (que l'on appelle aussi le produit point vectoriel) que l'on retrouve dans plusieurs fonctions de filtrage et de transformation [43]. L. Mintzer a été le pionnier de l'arithmétique distribuée sur FPGA. En 1992, il a montré que, combiné avec l'architecture à base de LUT Xilinx, l'algorithme DA produisait des concepts de filtres très efficaces [68].

En l'an 2000, avec l'avènement de la radio et des MODEM définis par logiciel sur FPGA, Mintzer a remarqué que, hormis avec l'approche DA, même le plus rapide multiplieur réalisé sous forme de tableau configuré dans un FPGA n'arrivait pas à égaler le coût et la performance d'un processeur DSP peu coûteux [69]. Avec l'apparition récente de multiplieurs matériels, de structures multiplieur–accumulateur (MAC) dans des blocs DSP et de *RocketIOs*^{TM} ayant un débit de gigabits par seconde, intégrés à la fabrication, les FPGA modernes sont maintenant beaucoup plus performants que les autres dispositifs.

Pipelinage des multiplieurs

Lorsque l'on utilise un outil de synthèse supportant le pipelinage, il est possible de décrire le pipelinage d'un multiplieur de manière comportementale en faisant suivre sa description par autant de registres que le nombre de niveaux de pipeline désirés. L'outil de synthèse peut ainsi équilibrer le multiplieur en utilisant les registres afin d'améliorer la fréquence d'horloge. Le code VHDL suivant demande 2 niveaux de pipeline.

```
process (CLK)
begin
    if (CLK'event and CLK='1') then
        Reg1 <= A * B;
        Reg2 <= Reg1;
    end if;
end process;
```

Plusieurs techniques de pipelinage de multiplieurs pour FPGA ont été proposées [44, 57, 99]. Certains auteurs [50, 100] ont même eu recours à la reconfiguration dynamique pour mettre à jour le contenu des LUT afin de définir une nouvelle valeur de multiplicande.

Dans FPL'99, une étude, soumise par M. Wojko, a comparé, sur des FPGA Xilinx® XC4036EX-2 et Altera EPF10K70RC-2, la performance en pipeline de trois techniques de multiplication : par tableau parallèle, par addition parallèle à base de vecteurs, et de type Wallace à retenue conservée, et a décrit deux techniques de multiplication alternatives pour FPGA, utilisant une approche hybride à retenue conservée/propagée [99].

Dans une étude plus récente, Panato et al. [76] ont envisagé l'usage de piplines très profonds pour les mises en œuvre sur FPGA, chaque étage de pipeline étant limité à une seul élément logique, et ont réalisé un muliplieur-tableau en utilisant cette technique.

2.3.3 Multiplieurs bit-sériels

Des modules de multiplication bit-sérielle ont fait l'objet de recherches approfondies dans la littérature de l'ITGE, en particulier pour le traitement de signal haute fréquence. Pour des applications, telles que le traitement d'images en temps réel et les opérations matricielles, où le plus grand nombre possible de multiplieurs doivent calculer en parallèle (calcul massivement parallèle), une « règle du pouce »suggère que le coût en cellules logiques d'un multiplieur ne devrait pas excéder cinq fois la longueur de ses opérandes.

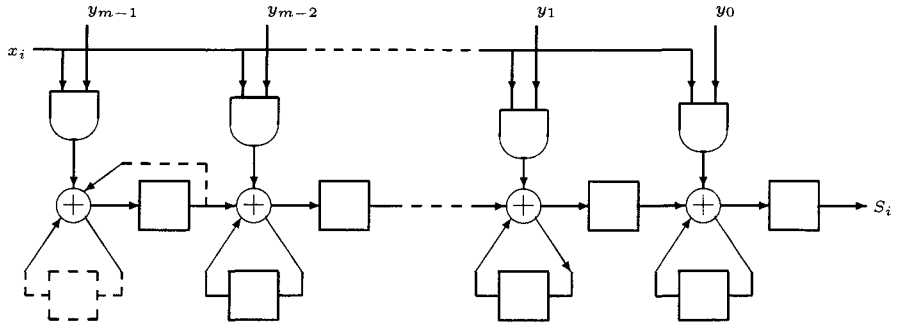

FIG. 2.6 – Multiplieur parallèle–série RCAD conventionnel

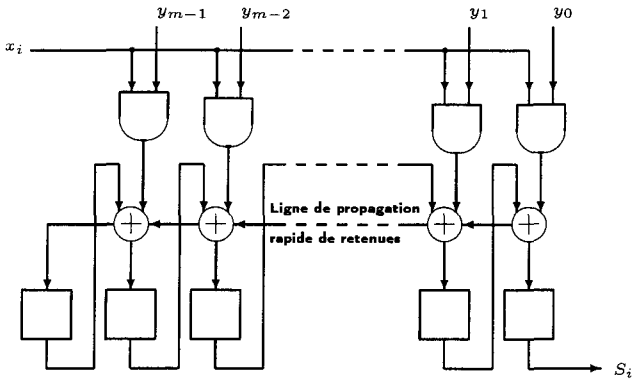

FIG. 2.7 – Multiplieur parallèle–série RPAD exploitant les lignes de propagation de retenues dédiées

La mise en œuvre la plus classique des multiplieurs sériels conventionnels (bit de

poids faible en tête), est la multiplication *série–parallèle* (Fig. 2.6 et 2.7), qui, comme son nom l'indique, accepte un opérande en série, l'autre en parallèle, et produit un résultat sériel. Ces multiplieurs ne sont basés que sur des additions et des décalages, et on en distingue deux variantes principales, selon que la retenue soit conservée (retenue conservée–addition–décalage – RCAD) (Fig. 2.6) ou propagée (retenue propagée–addition–décalage – RPAD) (Fig. 2.7) [9]. Comme il ne propage pas la retenue, un multiplieur RCAD peut fonctionner à très haute fréquence. Un multiplieur RPAD est beaucoup plus lent, mais, grâce à son exploitation des lignes de propagation de retenues dédiées, occupe deux fois moins d'espace qu'un multiplieur RCAD sur FPGA.

Hormis l'usage de multiplieurs ITGE embarqués ou de petits multiplieurs à base de table, la multiplication série–parallèle demeure la mise en œuvre la plus simple, la plus efficace, la moins coûteuse et la plus employée, autant pour FPGA que pour ASIC. Sur FPGA, elle peut être mise en correspondance directe avec l'architecture des tranches.

Valls et al. ont présenté, en 1998, une étude systématique de multiplieurs série–parallèles pour la réalisation de filtres numériques sur FPGA [95]. L'année suivante, cette étude a été enrichie d'autres multiplieurs série–parallèles chiffre-à-chiffre pipelinés, rapides et économiques [96]. En 2003, l'étude a été complétée par des résultats de réalisations optimisées pour FPGA à tables de vérité de 4 entrées, mais pouvant aussi se généraliser à d'autres tailles d'entrées [94]. Un guide de conception de plusieurs différents types de structures de multiplieurs série–parallèles, pour le traitement de signaux numériques, a été publié, en l'an 2000, par Ashour et Saleh [9].

Un concept de multiplieur reconfigurable, nommé DigiFAB, introduit en 2001 par Visavakul et al. [97], combine un type de multiplieur–tableau appelé Flexible Array Blocks (FABs) avec des techniques numériques pour produire des multiplieurs de taille arbitraire avec des ressources limitées.

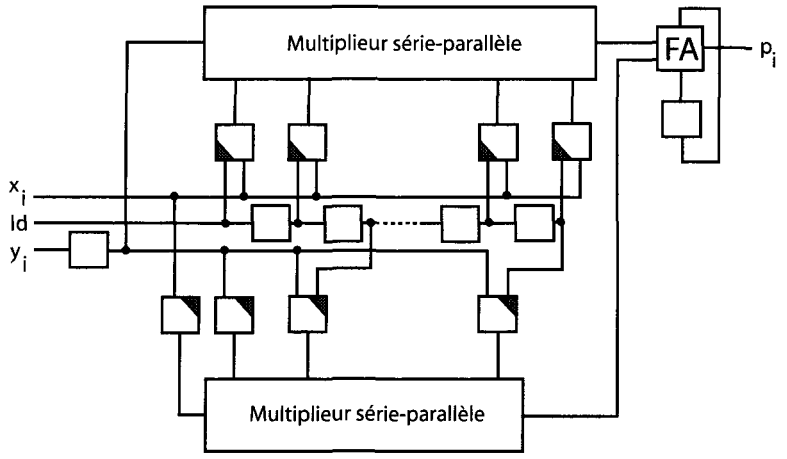

FIG. 2.8 – Un multiplieur série–série construit à partir de deux multiplieurs série–parallèle

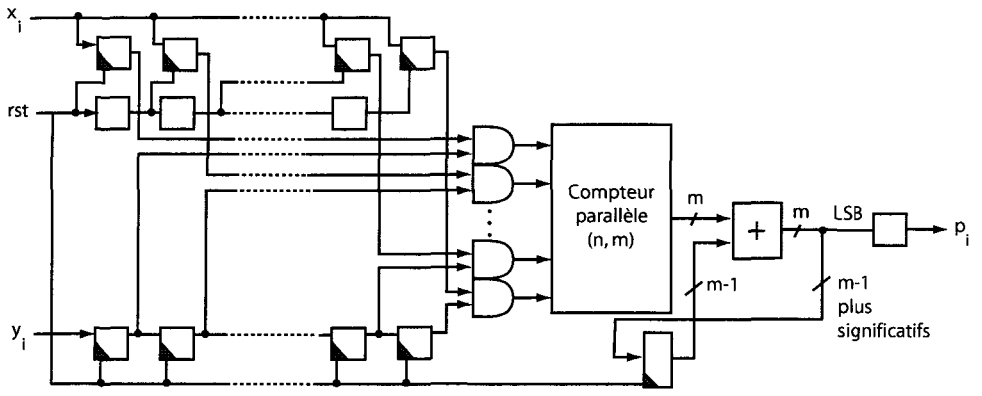

FIG. 2.9 – Un multiplieur série–série utilisant un compteur parallèle

Les travaux consacrés à la réalisation de multiplieurs sériels conventionnels acceptant les deux opérandes en série sont plutôt rares, encore davantage pour FPGA. On peut en trouver des exemples dans [45] et [26] (Fig. 2.8 et 2.9).

Ces multiplieurs sont généralement plus lents que des multiplieurs série–parallèles, et leur mise en œuvre s'avère aussi plus coûteuse. Il est d'ailleurs possible d'adapter ces derniers au moyen d'un convertisseur série–parallèle pour l'opérande parallèle, et d'un

délai de durée appropriée pour l'opérande sériel. On peut utiliser l'architecture de la Fig. 2.8 pour doubler le débit.

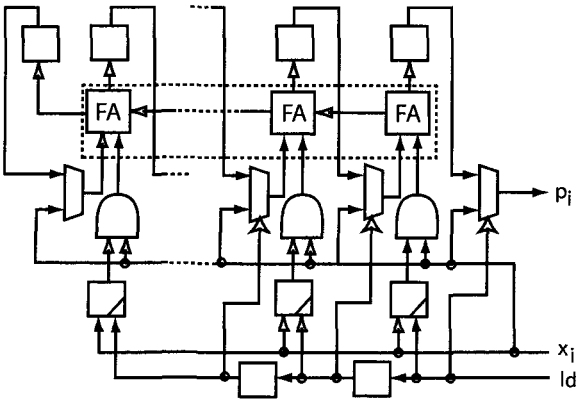

FIG. 2.10 – Opérateur bit-sériel d'élévation au carré

La Fig. 2.10 présente une architecture similaire pour l'élévation au carré bit-sérielle.

Multiplieurs bit-sériels constants

Les multiplieurs bit-sériels avec un opérande constant sont particulièrement utilisés dans les applications de traitement numérique du signal et le traitement vidéo, notamment pour la mise en œuvre en temps réel de la convolution pour le filtrage et des transformations telles que la FFT et la DCT.

De tels multiplieurs se construisent directement à partir d'un multiplieur série–parallèle, en maintenant l'opérande parallèle constant. Le circuit peut être simplifié là où la constante comporte des zéros. Par ailleurs, une structure de multiplication constante offrant un compromis espace–temps optimal a été décrit par Dittmann et al. en 2003 [34].

2.3.4 Additionneurs et multiplieurs modulaires

L'arithmétique modulaire joue un rôle crucial dans les applications de traitement de signal qui profitent de l'arithmétique en système RNS et dans divers algorithmes cryptographiques à clé publique, tels que RSA, où le principal calcul est la modulation exponentielle $c = m \bmod n$.

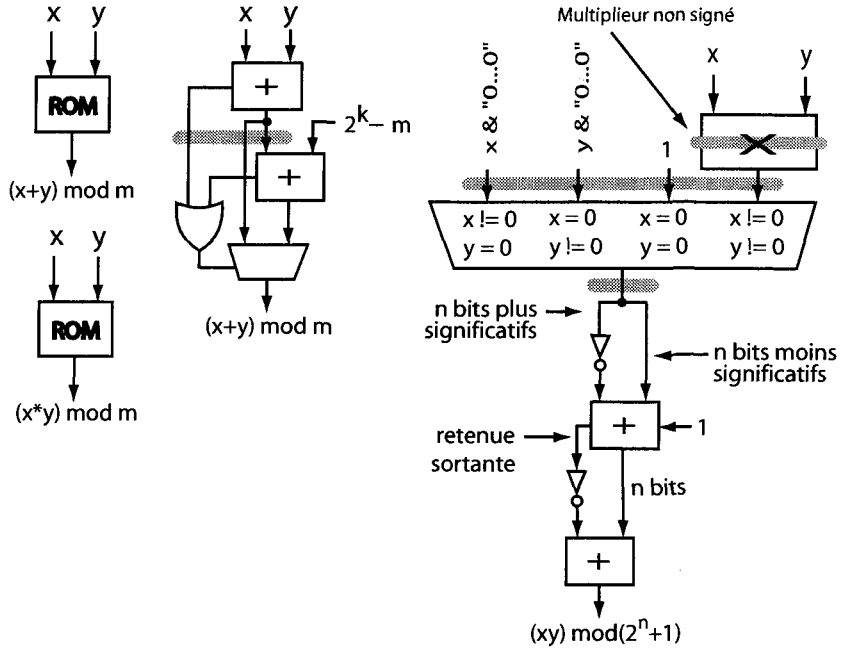

FIG. 2.11 – Quelques additionneurs et multiplieurs modulaires

J.-L. Beuchat a été parmi les premiers à contribuer à l'addition et à la multiplication modulaire sur FPGA [20]. Avec J.-M. Muller, il a proposé une famille d'algorithmes en base 2 conçus pour les FPGA à LUT à 4 entrées et à lignes de propagation de retenues dédiées [21]. Quelques additionneurs et multiplieurs modulaires découlant des travaux de Beuchat et Muller sont montrés à la Fig. 2.11. Ces multiplieurs sont efficaces pour des moduli allant jusqu'à 32 bits.

Pour les grands opérandes, les additionneurs à retenue conservée sont cependant

beaucoup plus rapides [14]. On trouve aussi dans [7] une étude d'architectures matérielles efficaces pour la mise en œuvre de la multiplication modulaire sur FPGA.

Multiplieurs modulaires bit-sériels

Une architecture de multiplication modulaire haute-vitesse basée sur l'algorithme de multiplication de Montgomery a été présentée par W. P. Marnane, en 1998 [63].

2.3.5 Opérateurs arithmétiques à base de petits blocs multiplieurs

Plusieurs FPGA apparus sur le marché récemment contiennent de petits multiplieurs ITGE ultra-rapides incorporés, et comportant également d'autres caractéristiques architecturales dédiées à la réalisation de multiplieurs haute-performance (Fig. 2.12). Dans cette catégorie, on compte (en ordre alphabétique par nom de fabriquant) les : Cyclone™ II, Stratix®, Stratix GX et Stratix II d'Altera, [2,4–6] ; Spartan™-3, Virtex-II, Virtex-II Pro, Pro X et Virtex-4 de Xilinx [104–107].

FIG. 2.12 – Multiplieur 18x18 embarqué

Altera et Xilinx ont publié des documents *application note* sur la mise en œuvre de multiplieurs haute performance dans ces FPGA [3, 102].

Certains des FPGA Altera intègrent des multiplieurs-accumulateurs (MACs) dans des blocs DSP, pouvant fonctionner à des débits de données supérieurs à 300 millions d'échantillons par seconde (MEPS) (Fig. 2.13). Chez Xilinx, les Virtex-4 contiennent des tranches XtremeDSP™, incluant un multiplieur 18x18, des étages de pipeline optionnels et un accumulateur et additionneur/soustracteur intégrés. Les blocs de mémoire

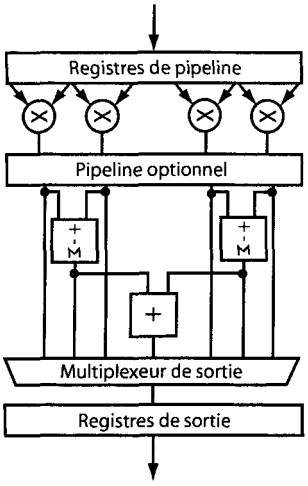

FIG. 2.13 – Bloc DSP d'un FPGA Stratix d'Altera

TriMatrix^{TM} d'Altera et les blocs RAM de Xilinx peuvent être utilisés comme des tables pour réaliser, notamment, des multiplieurs haute-performance à largeur/profondeur variable pour des applications de traitement de signaux numériques à gros volume et à faible coût. Un exemple de multiplieur de 36 bits réalisé au moyen de multiplieurs 18x18 embarqués est montré à la Fig. 2.14.

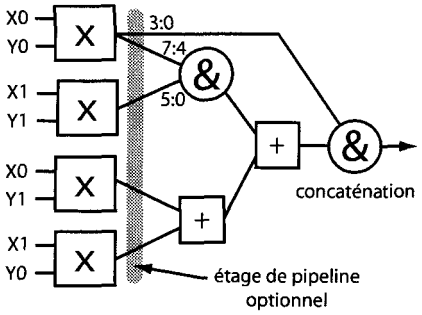

FIG. 2.14 – Multiplieur de 36 bits réalisé au moyen de multiplieurs 18x18 embarqués

Pour tirer profit des multiplieurs intégrés, Beuchat et Tisserand ont présenté, en 2002, quelques opérateurs de multiplication et de division, en divers compromis espace–temps, basés sur les blocs multiplieurs 18×18 disponibles dans la famille Virtex-II de

Xilinx® (mais généralisables à n'importe quel FPGA comportant de tels blocs) [24]. Les opérateurs présentés conduisent à des améliorations de vitesse allant jusqu'à 18 % pour la multiplication et 40 % pour la division. Ces résultats ont encore été améliorés par une contribution similaire présenté l'année suivante par Lee et Burgess [56]. Cette nouvelle contribution couvre, de plus, l'élévation au carré et la division fractionnaire, tout en réalisant une économie d'espace de 20 % et une réduction du délai de 30 % dans le cas de la multiplication. Une économie d'espace pouvant aller jusqu'à 50 % peut ainsi être obtenue dans le cas de la division.

2.3.6 Diviseurs et extracteurs de racine carrée

Les algorithmes restaurant et non restaurant sont les méthodes de base de division et d'extraction de la racine carrée par récurrence. Elles produisent un nombre fixe de bits par itération. La division non restaurante conserve toujours la différence dans le registre du reste partiel, alors que la division restaurante utilise une étape de supplémentaire pour restaurer conditionnellement le reste partiel à sa valeur précédente dans le cas où la différence est négative. L'algorithme SRT est une modification de l'algorithme non restaurant permettant de calculer en grande base dans un but d'accélération.

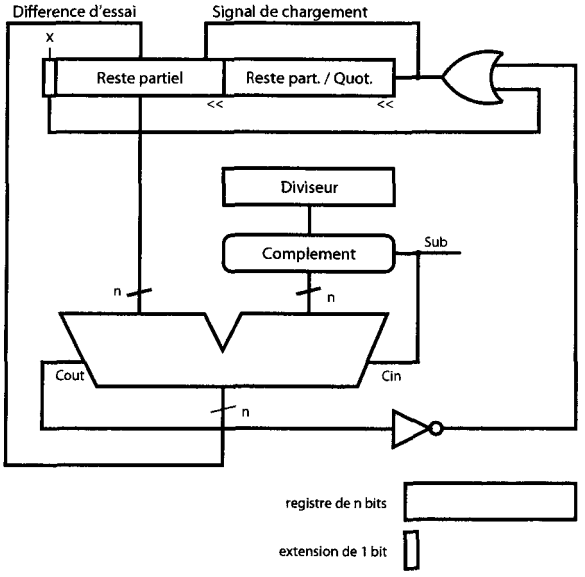

FIG. 2.15 – Diviseur séquentiel restaurant

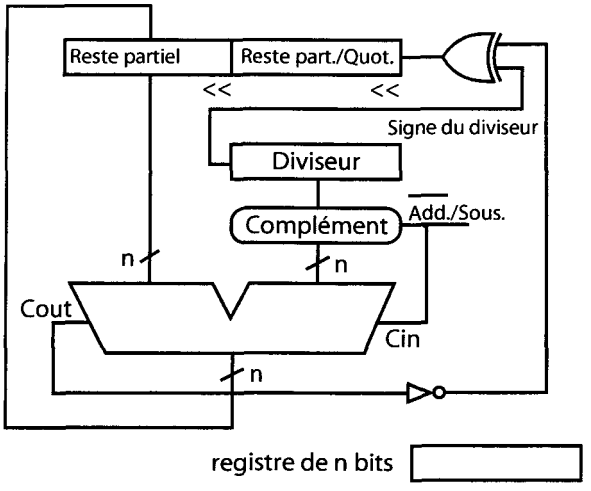

FIG. 2.16 – Diviseur séquentiel non restaurant

La Fig. 2.15 montre un diviseur restaurant séquentiel, et la Fig. 2.16, un diviseur non restaurant séquentiel. On peut obtenir un diviseur pseudo-combinatoire ou un diviseur pipeliné en déroulant l'itération de base, mais la division demeure toujours un processus fondamentalement itératif, où chaque étape d'itération ne produit qu'un seul chiffre du

résultat. Il n'existe pas de réseau combinatoire pour la division tel qu'il en existe pour l'addition et la multiplication.

La division non-restorante en base 2^n d'opérandes signés s'énonce comme suit :

$$\begin{array}{lll} x & \text{Dividende} & x_0x_1\cdots x_{nm-n-1}x_{nm-n}\cdots x_{nm-2}x_{nm-1} \\ d & \text{Diviseur} & d_0d_1\cdots d_{nm-n-1}d_{nm-n}\cdots d_{nm-2}d_{nm-1} \\ d & \text{Quotient} & q_0q_1\cdots q_{nm-n-1}q_{nm-n}\cdots q_{nm-2}q_{nm-1} \\ w & \text{Reste} & w_0w_1\cdots w_{nm-n-1}w_{nm-n}\cdots w_{nm-2}w_{nm-1} \end{array}$$

où $q = x/d$, $w = x - (d \times q)$, sachant que x et d sont tous deux de m chiffres, où chaque chiffre comporte n bits, et x_0 et d_0 sont les chiffres de signe de x et d, respectivement, comportant chacun n bits. L'expression $x-(d\times q)$ décrivant le reste w découle de l'équation de base de division $x = (d\times q)+w$. Cette équation, conjointement à la condition $w < d$, définit complètement la division entière. Le problème de la division binaire se réduit à soustraire du dividende x un ensemble de nombres, chacun étant soit 0 ou une version décalée du d.

$$Y = \begin{cases} 0 & \text{si positif} \\ 2^n - 1 & \text{si négatif} \end{cases}$$

Le quotient est obtenu en ajoutant un nombre de zéros à la suite du dividende z. La décision d'effectuer soit une addition, soit une soustraction à chaque étape de l'algorithme 1 est basée sur le chiffre de signe du reste partiel courant w_{m-1} ou sur le dividende, x_{m-1}.

Exemple 1 *Une simulation avec les entrées :* $n = 16$, $d = 289$ *and* $x = 4128$ *donnant pour résultat :* $q = (0000000000001110.010001010001\ldots)_2 = (14.283721923828125)_{10}$ *est montrée à la figure 2.17.*

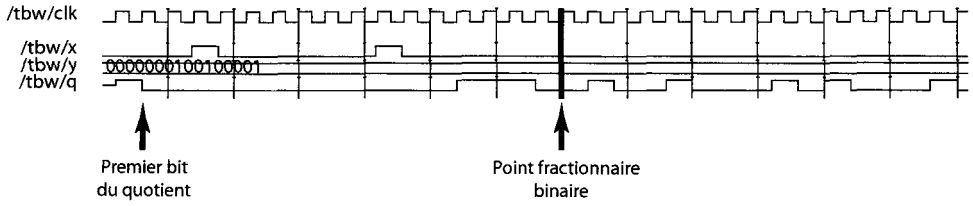

FIG. 2.17 – Chronogramme de l'exemple 1

Exemple 2 *Une simulation avec les entrées :* $n = 16$, $d = 17000$ *et* $x = 1025$ *donnant pour résultat :* $q = (0000000000001110.010001010001\ldots)_2 = (14.283721923828125)_{10}$ *est montrée à la figure 2.18.*

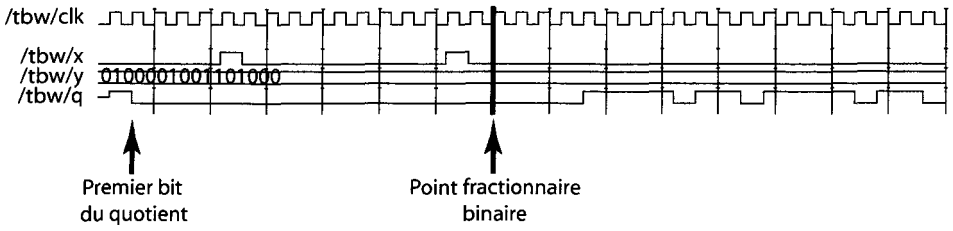

FIG. 2.18 – Chronogramme de l'exemple 2

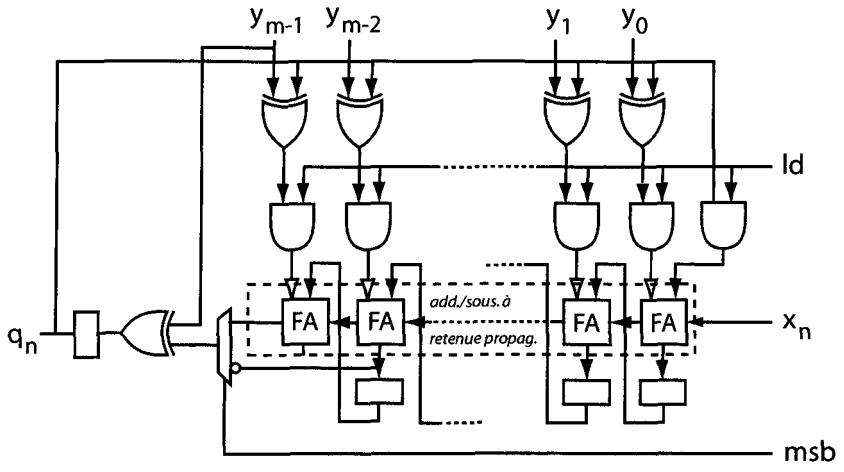

FIG. 2.19 – Diviseur série–parallèle non restaurant utilisant les lignes de propagation de retenues dédiées

Algorithme 1 Division non restaurante

[Initialisation]
$w[0] = x$
$w[1] = 2w[0] - d^*$
[Récurrence]
pour $j = 1 \ldots n-1$ **faire**
  si $w[j] \geq 0$ **alors**
    $q_{n-1-j} = 1; w[j+1] = 2w[j] - d^*$
  sinon
    $q_{n-1-j} = 0; w[j+1] = 2w[j] + d^*$
  fin si
  [Correction]
  si $w[n] < 0$ **alors**
    $q_0 = 0; w[n] = w[n] + d^*$
  sinon
    $q_0 = 1$
  fin si
fin pour

Un diviseur non restaurant acceptant un opérande en série, et l'autre en parallèle, se conçoit aisément en réorganisant le circuit de la Fig. 2.16 pour obtenir le circuit de la Fig. 2.19. Ce circuit peut être mis en correspondance directe avec l'architecture logique des FPGA, utilise la logique de propagation de retenues dédiées, donnant ainsi une mise en œuvre très compacte. D'autres mises en œuvre de diviseurs sériels ont été présentés dans [12, 13, 17, 64].

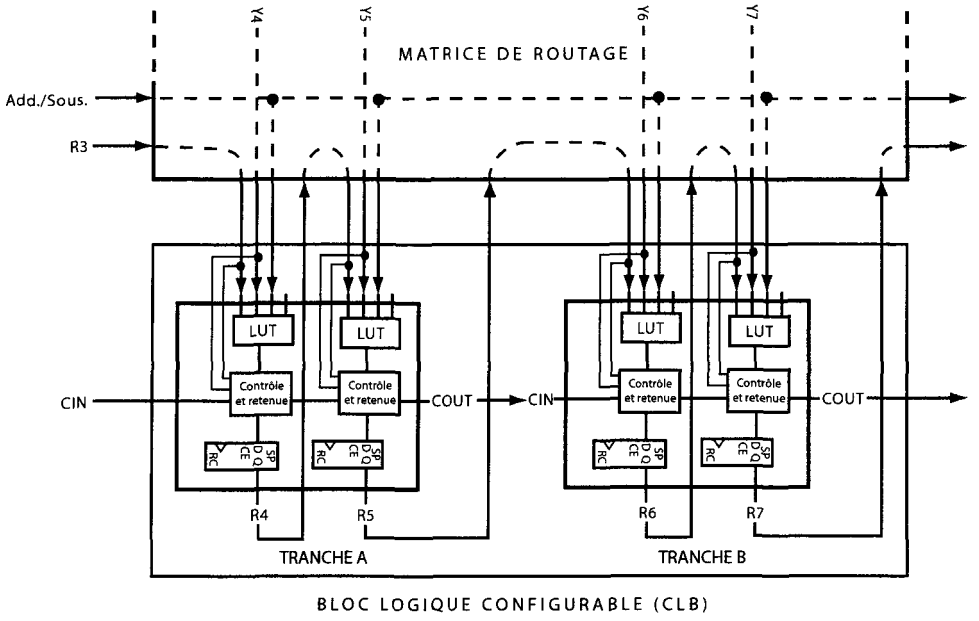

FIG. 2.20 – Correspondance directe d'un diviseur série-parallèle non restaurant avec l'architecture physique d'un FPGA

La figure 2.20, montre la correspondance exacte de cette architecture de diviseur serie-parallèlle non restaurant avec le matériel sous-jacent d'un FPGA. Cette architecture a une latence de bit d'un seul cycle d'horloge, et occupe un espace minimal de n cellules logiques.

Pour des mises en œuvre complètement sérielles, de simples registres de délai économiques de type serial-in/serial-out (SISO), présents dans certains FPGA, peuvent être utilisés. Notamment, les LUT des architectures Xilinx modernes peuvent être utilisés comme des registres à décalage 16 bit (SRL16) de ce type [103].

L'algorithme SRT

L'algorithme SRT est l'un des algorithmes de division et d'extraction de la racine carrée les plus efficaces actuellement. Les initiales SRT sont tirées des noms de Sweeney, Robertson et Tocher, qui ont développé l'algorithme de manière indépendante à peu

près en même temps. Cet algorithme est de la famille non restaurante et son objectif est d'obtenir une accélération accrue par rapport à l'algorithme non restaurant fondamental. L'algorithme SRT peut aussi se classer comme un algorithme en-ligne, étant donné qu'il utilise une représentation numérique redondante à chiffres signés, produit le résultat en série, chiffre le plus significatif en tête, et requiert ensuite une conversion à la volée et un arrondi.

L'algorithme fonctionne en permettant à un chiffre du quotient d'être 0, en plus des -1 ou +1 de la division non restaurante fondamentale, ce qui simplifie les comparaisons en posant $D = 1/2$. La règle de sélection du chiffre du quotient s'en trouve modifiée ainsi :

$$q_i = \begin{cases} 1 & \text{si } 2 \cdot r_{i-1} \geq D \\ 0 & \text{si } -D \leq 2 \cdot r_{i-1} < D \\ \bar{1} & \text{si } 2 \cdot r_{i-1} < -D \end{cases}$$

Le quotient lui-même s'en trouve cependant compliqué, et, pour reconvertir en binaire standard, on doit alors avoir recours à une méthode de conversion à la volée utilisant deux registres, introduite au début des années 1980.

Un premier exemple de réalisation sur FPGA de l'algorithme SRT en base 4 a été présenté dans [27], et une étude comparative complète des diviseurs restaurants, non-restaurants et SRT sur FPGA a été publiée dans FPL 2004 [83].

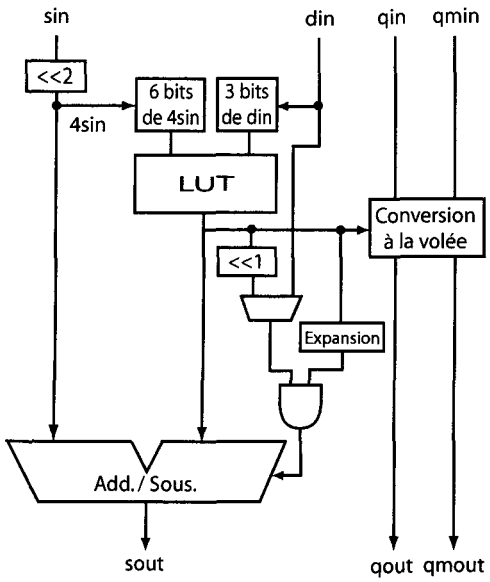

FIG. 2.21 – Étape de division SRT en base 4

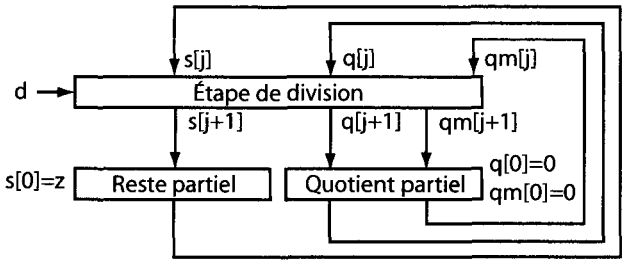

FIG. 2.22 – Mise en œuvre itérative de la division SRT

Un étage d'itération pour la division SRT en base 4 est montré à la Fig. 2.21.

Tout comme dans le cas de l'algorithme non restaurant conventionnel, une mise en œuvre itérative de la division SRT (Fig. 2.22) offre un compromis espace–temps intéressant sur FPGA.

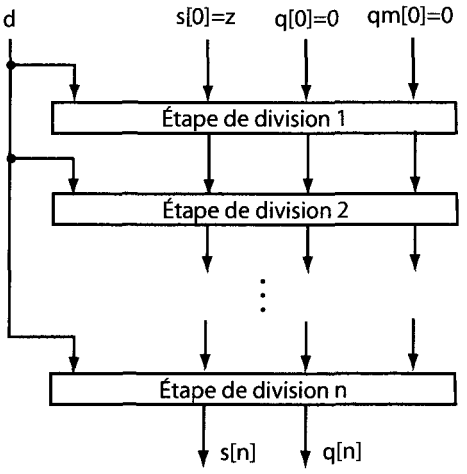

FIG. 2.23 – Mise en œuvre de la division SRT sous forme de tableau

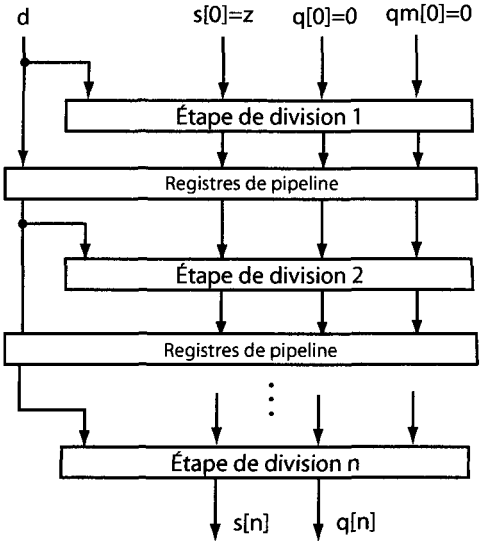

FIG. 2.24 – Mise en œuvre de la division SRT pipelinée

Les figures 2.22, 2.23 et 2.24 donnent le schéma générique de différentes mises en œuvre de diviseurs SRT.

Racine carrée

La racine carrée est une opération essentielle dans les applications telles que les graphiques par ordinateur, les calculs scientifiques et certains algorithmes de commande, mais on la considère difficile à réaliser en matériel.

Les algorithmes connus pour calculer une racine carrée incluent : la méthode de Newton–Raphson, la méthode SRT redondante et non-redondante, l'algorithme non restaurant fondamental et l'algorithme CORDIC.

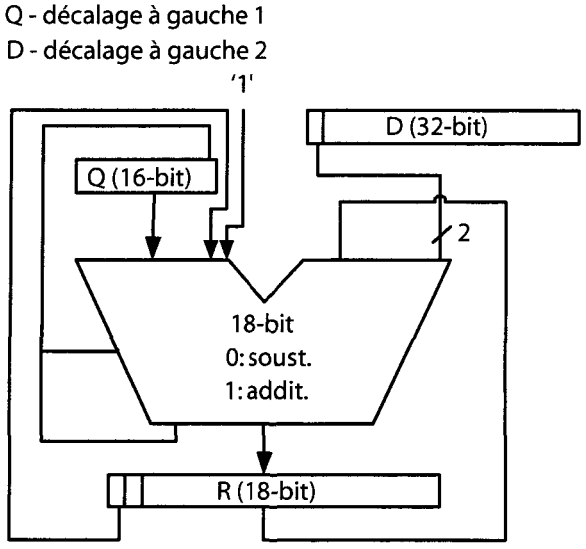

FIG. 2.25 – Racine carrée non restaurante

En 2001, Piromsopa et al. [80] ont présenté une mise en œuvre FPGA 32-bit de l'algorithme non restaurant (Fig. 2.25). De toutes les approches possibles, les auteurs ont retenu cette dernière comme la méthode la plus appropriée pour calculer des racines carrées sur FPGA, car c'est celle qui permet d'atteindre une haute performance au moindre coût.

2.4 Opérateurs à virgule fixe

Les représentations en virgule fixe, contrairement aux représentations en virgule flottante, qui comportent un exposant géré automatiquement, mais n'offrent pas autant de précision pour un même nombre de bits de représentation, peuvent représenter de manière exacte toutes les valeurs inférieures à la valeur maximale possible suivant le nombre de bits utilisé dans la représentation, pour autant que la valeur à représenter ne prenne pas plus que le nombre de chiffres supporté après la virgule.

La plus grande étendue dynamique offerte par une représentation en virgule flottante n'est possible qu'au coût d'une précision moindre et d'une plus grande complexité comparativement à une représentation en virgule fixe. Avec une performance nettement supérieure, une occupation de surface de circuit bien plus réduite et une moindre consommation d'énergie, les formats à virgule fixe offrent une performance numérique sinon égale, du moins, à un facteur prévisible moindre, que les formats à virgule flottante.

Les applications nécessitant une vaste étendue dynamique, cependant, ne peuvent se contenter d'une représentation à virgule fixe et amènent le besoin d'utiliser soit une représentation à virgule flottante standard, soit une représentation logarithmique. Une représentation logarithmique est un type de représentation à virgule flottante où la mantisse est égale à 1 et où l'exposant est représenté en virgule fixe.

Si la virgule flottante est de loin le système de représentation des nombres réels le plus couramment utilisé, une représentation logarithmique permet d'effectuer de manière simple et rapide les opérations arithmétiques de multiplication, de division et d'extraction de la racine carrée [65]. Cependant, la réalisation des opérations d'addition et de soustraction est complexe et fait croître de manière exponentielle la surface de ces circuits suivant la précision désirée [33, 65].

2.4.1 Opérateurs à virgule fixe parallèles

Un nombre à virgule fixe est une valeur comprenant une partie entière et une partie fractionnaire. Une tel format permet de représenter de manière exacte des fractions décimales, même en utilisant à l'interne une arithmétique en base 2, qui est plus performante en circuits numériques. Le nombre de bits attribués à la partie entière peut être différent de celui attribué à la partie fractionnaire.

Les additionneurs et soustracteurs à virgule fixe sont identiques à leurs équivalents entiers. La position de la virgule est implicite, et il suffit d'aligner au préalable les opérandes de manière à ce que les bits de même poids soient vis-à-vis. La virgule du résultat est à la même position que celle des opérandes.

Pour les opérations de multiplication et de division, une mise à l'échelle est nécessaire après l'opération. Une mise à une échelle supérieure revient à multiplier le résultat par une constante, ou, plus simplement, à le décaler vers la gauche. Une mise à une échelle inférieure consiste en une multiplication par une constante N suivie d'une division par une constante D. Dans ce cas, la division peut être avantageusement remplacée par un décalage à droite si l'on choisit des constantes N et D telles que D soit une puissance de 2.

Pour effectuer un arrondi, il faut ajouter un facteur correctif avant que la troncature ne se produise. C'est le cas notamment en ce qui concerne la division. De plus, un inconvénient non négligeable est dû au fait que les opérateurs de multiplication et de division doivent avoir une précision interne suffisante pour mémoriser les résultats intermédiaires, qui sont N fois plus grands que leurs opérandes. Tel que mentionné, le choix d'une échelle puissance de deux est le meilleur choix pour les opérations de mise à l'échelle requises par la multiplication et la division, car cela permet de les réaliser au moyen de simples décalages.

2.4.2 Opérateurs en-ligne

L'arithmétique en-ligne est une arithmétique à virgule fixe, car elle opère toujours sur des nombres fractionnaires. Elle a été étudiée par de nombreux auteurs, à commencer, dans l'article original de Trivedi et Ercegovac [91], par les opérations de multiplication et de division.

Un opérateur en-ligne reçoit ses données d'entrée en série, en commençant par le chiffre le plus significatif [36,37]. C'est en ayant recours à une représentation redondante des nombres qu'il est possible d'effectuer toutes les opérations chiffre de poids fort en tête. Les chiffres de la sortie doivent appartenir à un ensemble redondant de chiffres $\{-a, \ldots, -1, 0, 1, \ldots, a\}$, où $(r/2) \leq a \leq r-1$, pour r représentant la base numérique. Pour fins de compatibilité, les chiffres de l'entrée sont supposés être dans le même ensemble que ceux de la sortie. Puisque les chiffres peuvent avoir des poids positifs ou négatifs, on les appelle ainsi des *chiffres signés* [10]. En base 2, l'ensemble de chiffres est $\{-1, 0, 1\}$.

Des codes particuliers sont utilisés pour représenter les chiffres signés, qui sont habituellement représentés soit en code complément à deux (C2), soit en code *borrow-save* (BS). Pour prendre la base 2 comme exemple, dans les deux cas, un chiffre d est codé par une paire de bits (d_1, d_2). À partir de cette paire de bits, la valeur du chiffre est calculée comme $-2d_1 + d_2$ lorsqu'on utilise le code C2, et $d_1 - d_2$ lorsqu'on utilise le code BS.

L'utilisation d'une représentation numérique redondante évite d'avoir à propager la retenue des additions, ce qui permet d'effectuer successivement plusieurs opérations en-ligne, reliées par un pipeline de chiffres. Le délai global est alors la somme des délais individuels, et les interconnexions entre les modules sont minimales. La Fig. 2.26 présente un exemple de réseau d'arithmétique pour le calcul en-ligne d'une équation

donnée.

$$z = \sqrt{\frac{e-f}{(a+b)cd}}$$

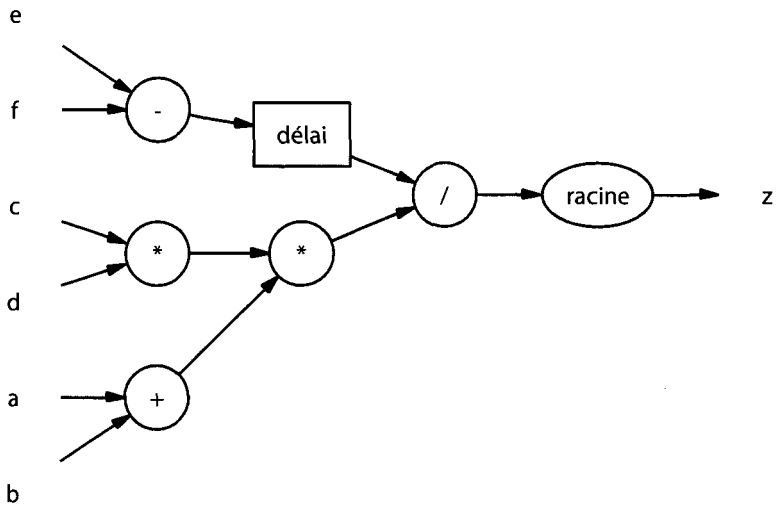

FIG. 2.26 – Exemple de réseau d'opérateurs en-ligne

Les systèmes en-ligne sont caractérisés par leur *délai en-ligne,* δ, et leur période d'horloge, τ, qui est le temps requis pour calculer un nouveau chiffre du résultat à chaque étape. Toutes les opérations arithmétiques peuvent ainsi commencer à produire des chiffres du résultat dès la réception d'un petit nombre de chiffres de leurs opérandes (habituellement 2–5 chiffres), ce qui rend possible un flux de données régulier et parfaitement uniforme.

La conception d'opérateurs en-ligne implique souvent le choix d'un compromis entre la valeur de δ et de τ, car, tel que mentionné par S. Kla, pour certaines fonctions, un algorithme en-ligne de plus court délai aura une plus longue période, et, à l'inverse, le débit d'un opérateur en-ligne peut être augmenté, moyennant un plus long délai δ, par l'ajout de bascules [51]. La Fig. 2.27 montre le déroulement temporel d'un calcul

parallèle et d'un calcul en-ligne de l'équation de l'exemple de la Fig. 2.26.

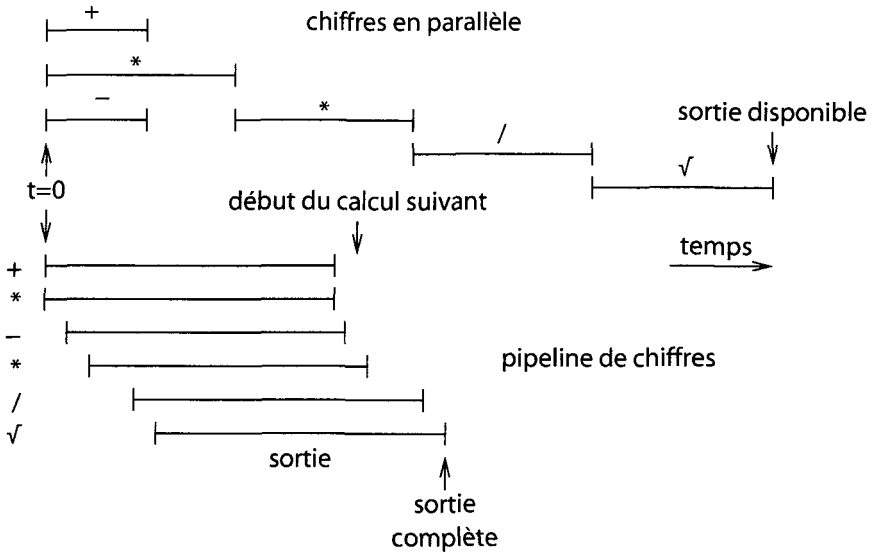

FIG. 2.27 – Déroulement temporel comparé d'un calcul parallèle et d'un calcul en-ligne

Après la réception de δ chiffres des opérandes d'entrée, l'algorithme est capable de générer le premier chiffre du résultat. Les chiffres de la sortie sont donc retardés par $\delta + 1$ cycles d'horloge par rapport aux entrées.

Afin d'éviter toute confusion lorsque le premier chiffre de la sortie a un poids différent de celui du premier chiffre de l'entrée, le délai en-ligne δ se définit comme le nombre de cycles d'horloge entre le premier chiffre fractionnaire de l'entrée et le premier chiffre fractionnaire de la sortie. Le nombre δ est tel qu'il faut $p + \delta$ chiffres en entrée pour calculer p chiffres du résultat.

L'arithmétique en-ligne offre une solution efficace aux problèmes de restriction de surface de puce fréquemment rencontrés en conception numérique et facilite le pipelinage des opérations arithmétiques, qu'elle permet d'effectuer simultanément à mesure que les chiffres deviennent disponibles. Cela peut réduire le temps de calcul de longues séquences d'opérations tout en minimisant les interconnexions entre les opérateurs.

Dès 1984, M. D. Ercegovac, le *père* de l'arithmétique en-ligne, avait constaté que, grâce à son caractère hautement modulaire et à ses interconnexions simples, l'arithmétique en-ligne paraissait attrayante dans les réseaux reconfigurables [36].

TAB. 2.1 – Comparaison de la taille des opérateurs et de la période d'horloge d'opérateurs en-ligne et parallèles

	En-ligne		Parallèlle	
Opération	ADD	MULT	ADD	MULT
taille	$O(1)$	$O(n)$	$O(n)$	$O(n^2)$
Période	$O(1)$	$O(1)$	$O(log(n))$	$O(log(n))$

De plus, l'arithmétique en-ligne permet de supporter à la fois une reconfiguration dynamique de la précision et d'obtenir un fort débit. Tel que montré en Table 2.1, la taille des modules en-ligne n'augmente que très peu en fonction de la longueur des opérandes, comparativement à des modules parallèles équivalents, et la période d'horloge de ce type de modules est indépendante de la longueur des opérandes.

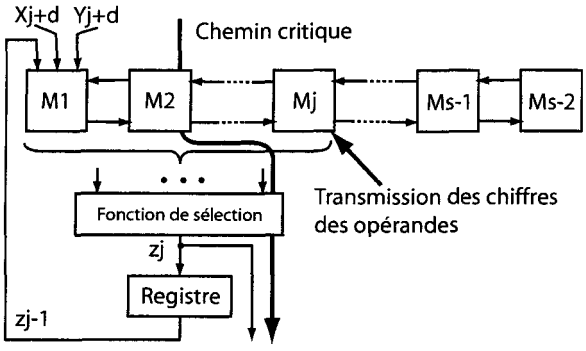

FIG. 2.28 – Structure générale d'un opérateur en-ligne

La Fig. 2.28 montre la structure générale d'un opérateur en-ligne. Chaque opérateur en-ligne réalise une équation de récurrence différente. Un opérateur en-ligne pour une fonction à deux entrées $g(X, Y)$ calcule le résultat en plusieurs cycles d'horloge au

moyen d'une equation de récurrence de la forme :

$$W[j] = rW[j-1]\mathcal{F}\left(X[j], Y[j], Z[j]\right)$$

où $\mathcal{F}$ est une fonction qui change selon l'opération arithmétique à réaliser (g). $\mathcal{F}$ se compose uniquement de multiplications, d'additions/soustractions et de décalages, et, fondamentalement, elle calcule la différence entre la valeur actuelle de la fonction $g(X[j], Y[j])$ et la sortie $Z[j]$. $X[j], Y[j]$, et $Z[j]$ représentent les vecteurs de chiffres reçus/générés à l'étape j, ainsi :

$$\begin{aligned} X[j] &= \sum_{i=0}^{\delta+j} x_i r^{-i} = X[j-1] + x_{j+\delta} r^{-(j+\delta} \\ Y[j] &= \sum_{i=0}^{\delta+j} y_i r^{-i} = Y[j-1] + y_{j+\delta} r^{-(j+\delta)} \\ Z[j] &= \sum i = 0^{j} z_i r^{-i} \end{aligned}$$

où x_i, y_i, et $z_i \in \{-a, \ldots, -1, 0, 1, \ldots, a\}$. Les entrées doivent être des fractions.

Une fonction de sélection est utilisée pour calculer les chiffres de sortie à chaque étape tout en gardant le reste W à l'intérieur des bornes. Cette fonction se décrit sous la forme générale suivante :

$$z_j = S\left(\hat{W}[j-1], x_{j+\delta}, y_{j+\delta}\right)$$

où $\hat{W}[j-1]$ est un estimé du reste à l'étape $j-1$ (voir [36]), qui est la valeur de $W[j-1]$ avec une précision de t chiffres fractionnaires. La valeur de t qui garantit la convergence pour chaque algorithme en-ligne doit être calculée lors de la conception.

Une méthode pour développer des opérateurs en-ligne a été présentée en [37].

Une première étude sur la mise en œuvre de l'arithmétique en-ligne sur PAM *(Programmable Active Memories)*, une machine virtuelle dynamiquement reconfigurable à base de FPGA, a été présentée en 1994 par Daumas, Muller et Vuillemin [31]. La même année, un certain nombre d'opérateurs en base 2, incluant des additionneurs (Fig. 2.29), des multiplieurs, un multiplieur-accumulateur (MAC) et un opérateur d'élévation au carré ont été décrits par Bajard et al. (1994) [11]. Ces opérateurs ont été repris, mis en œuvre sur FPGA et évalués dans les thèses d'A. Tisserand [90] et de J.-L. Beuchat [19], avec plusieurs contributions nouvelles.

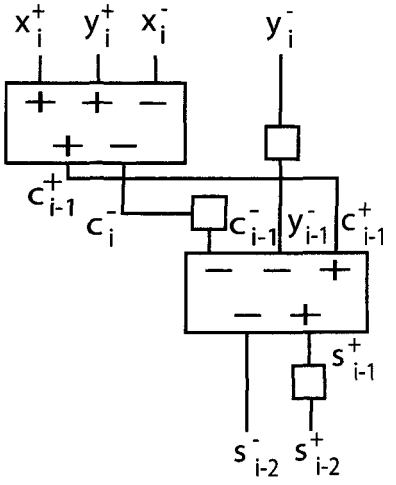

FIG. 2.29 – Additionneur en-ligne fait de deux cellules PPM

L'arithmétique en-ligne est particulièrement utile dans les applications de traitement de signal, car les modules n'ont pas à calculer les résultats en pleine précision. Deux articles publiés, respectivement, en 1999 et en 2001, présentant des bibliothèques d'opérateurs en-ligne pour le traitement de signal [42, 55]. Ces articles font figure de références pour l'évaluation de mises en œuvre de l'arithmétique en-ligne sur FPGA.

Le premier, par Ercegovac et al., remarque que l'arithmétique redondante est moins attrayante pour FPGA que pour ASIC, où elle peut raccourcir le chemin critique sans

obliger le concepteur à recourir à un pipelinage profond. Les auteurs mentionnent que cela est dû au fait que la logique de propagation rapide de retenues présente dans les FPGA donnent aux additionneurs à retenue propagée un léger avantage de performance jusqu'à une certaine largeur d'opérandes. Ce fait est maintenant bien connu. De plus, nous savons qu'utiliser la ligne de propagation rapide des retenues donne lieu à des mises en œuvre plus compactes.

Dans le deuxième article, se basant sur les résultats du premier, Galli et Tenca [42] ont presenté des mises en œuvre d'opérateurs en-ligne hautement optimisés pour la technologie FPGA cible et apportant un gain de performance significatif par rapport au précédent.

L'arithmétique en-ligne permet la réalisation des opérations de division et de racine carrée [73, 91], dont le fonctionnement propre requiert l'arrivée des chiffres les plus significatifs en premier. Effectuer ces opérations dans la direction opposée entraînerait des délais trop importants, obstruant le flux du pipeline de chiffres.

Dans le mode en-ligne, comme dans les autres modes parallèles et sériels, la division et la racine carrée sont basées sur des algorithmes de récurrence, qui calculent le résultat chiffre à chiffre ; mais, cette fois, sous forme redondante. Des mises en œuvre classiques de la division en-ligne ont été présentées dans [38] et [92]. Des adaptations d'un algorithme de récurrence pour la division [60] et l'extraction de la racine carrée à l'architecture du Xilinx XC4010 ont été présentées par Louie et Ercegovac, en 1993 [59].

La sérialisation des algorithmes de division par récurrence, en particulier les algorithmes non restaurants, peut donner lieu à des correpondances FPGA de taille minimale, et des diviseurs sériels pourraient s'avérer convenir à certaines applications de traitement ou de commande de signaux numériques.

Les manières possibles de combiner un diviseur à un réseau d'arithmétique en-ligne

sont au nombre de trois : utiliser un diviseur en-ligne (redondant), un diviseur sériel en représentation binaire standard ou en complément à 2, ou un diviseur parallèle couplé par des convertisseurs.

Un diviseur en-ligne (redondant) est toujours plus volumineux qu'un diviseur sériel conventionnel et opère à plus basse fréquence que les autres opérateurs en-ligne. Nous avons mentionné, dans l'introduction, que la solution qu'apportait l'arithmétique en-ligne au problème de compatibilité, en arithmétique sérielle, de l'addition et de la multiplication, d'une part, avec la division et la racine carrée, d'autre part, était imparfaite. C'est qu'il existe un fait méconnu qui complique le couplage d'un diviseur en-ligne à un réseau d'arithmétique en-ligne : la division en-ligne requiert que le diviseur se trouve dans une certaine plage de valeurs et que le numérateur soit strictement plus petit que le diviseur. Les structures de contrôle requises pour assurer ces contraintes rendent alors variable le délai du réseau d'arithmétique en-ligne tout entier [41]. C'est ce délai variable ainsi que la fréquence d'opération moins élevée d'un diviseur en-ligne qui sont les principaux facteurs limitant les performances de l'arithmétique en-ligne et son succès à uniformiser le pipelinage au niveau du chiffre de toutes les opérations.

Galli et Tenca ont proposé la possibilité de coupler à un réseau d'arithmétique en-ligne un diviseur parallèle SRT, plutôt qu'un diviseur en-ligne, moyennant une conversion consistant à paralléliser et à normaliser les deux opérandes avant traitement [42]. L'étape de conversion introduit cependant un délai d'autant de cycles d'horloge que le nombre de chiffres des opérandes, N. Ce compromis peut s'avérer intéressant pour certaines applications.

Des mises en œuvre en-ligne de la division ont été présentées dans [38,39,89,91,92], et de la racine carrée pour FPGA, dans [38,39].

Pour les applications scientifiques, une arithmétique en-ligne à précision variable

étendue, a été introduite par A. Tenca dans sa thèse, soutenue en 1998 [88].

2.5 Opérateurs à virgule flottante

2.5.1 Opérateurs parallèles à virgule flottante

Les concepteurs ont commencé au milieu des années 90 à adapter des unités arithmétiques à virgule flottante à l'architecture des FPGA. Les travaux portant sur l'arithmétique à virgule flottante se sont surtout attardés à créer des opérateurs optimisés pour l'économie d'espace afin de profiter au maximum des capacités des FPGA. Avec la densité croissante de ces puces, quelques unités arithmétiques à virgule flottante complètes ont pu, à ce jour, être réalisées sur FPGA, notamment celles de P. Belanović [74, 75] et de J. Detrey [33].

Les modules matériels développés par Belanović sont paramétrisés et peuvent opérér sur n'importe quelle représentation à virgule flottante personnalisée. Ils supportent la normalisation, la dénormalisation et l'arrondi, et incluent des modules de conversion entre les représentations à virgule fixe et à virgule flottante.

En 2004, K. Underwood a présenté une étude de la performance d'opérateurs à virgule flottante sur FPGA avec des résultats de mise en œuvre [93].

Si la virgule flottante est de loin le système de représentation des nombres réels le plus couramment utilisé, le système de représentation dit *logarithmique* offre une alternative intéressante pour certaines applications. Matoušek et al. (2002) [65] ont démontré la pertinence de la mise en œuvre de l'arithmétique en représentation logarithmique (LNS) sur FPGA. Les circuits arithmétiques LNS peuvent effectuer de façon simple et rapide les opérations arithmétiques de multiplication, de division et d'extraction de la racine carrée. Cependant, la réalisation des opérations d'addition et de soustraction

est complexe et fait croître de manière exponentielle la surface de ces circuits selon la précision souhaitée [65] [33].

Dans le cadre du projet français Arénaire (Arithmétique des ordinateurs), Detrey a effectué une étude comparative des systèmes de représentation des nombres réels à virgule flottante et en représentation logarithmique (LNS). Dans le cadre de cette étude, il a aussi développé, en VHDL, une bibliothèque d'opérateurs parallèles paramétrisables pour FPGA, supportant ces deux systèmes de représentation.

Les fonctions d'arrondi sont complexes en termes d'exactitude, et coûteuses en matériel, en particulier pour les opérations à virgule flottante. Réaliser en matériel un arrondi optimal et correct représente le plus grand défi, auquel ont été consacrés plusieurs travaux, notamment au sein du projet Arénaire.

Dans le cas des nombres représentés en virgule fixe, tel que mentionné précédemment, pour effectuer un arrondi, il faut ajouter à un nombre à virgule fixe un facteur correctif avant que la troncature ne se produise. De plus, les opérateurs de multiplication et de division en virgule fixe doivent avoir une précision interne suffisante pour mémoriser les résultats intermédiaires.

2.5.2 Opérateurs sériels à virgule flottante

L'arithmétique à virgule flottante sérielle a été étudiée dans le mode de fonctionnement dit *en-ligne*, utilisant un format numérique redondant (à chiffres signés d'Avizienis) [10], et dont les opérations se font chiffre de poids fort en tête.

Des algorithmes en-ligne à virgule flottante pour l'arithmétique sur les nombres complexes ont été présentés par McIlhenny et Ercegovac (1998) [67].

Dans sa thèse de doctorat, McIlhenny [66] a développé une bibliothèque d'arithmétique en-ligne à virgule flottante pour les nombres complexes et pour les nombres réels.

Ayant introduit un format numérique traitant la partie réelle et la partie imaginaire d'un nombre complexe comme un nombre unifié, il a réalisé, en VHDL, des opérateurs en-ligne fonctionnant dans cette représentation, incluant : l'addition, la multiplication, l'élévation au carré, la division et la racine carrée. Cette bibliothèque ne comporte, cependant, ni circuits de normalisation, ni de dénormalisation, ni d'arrondi. Seules les définitions théoriques s'en trouvent dans la thèse. Les opérandes d'entrée sont censés avoir été quasi-normalisés au préalable, et la bibliothèque n'inclut pas de circuit de quasi-normalisation en-ligne. L'auteur n'a pas non plus présenté une analyse d'erreur des fonctions supportées par sa bibliothèque.

Un algorithme en-ligne de délai 1 pour l'extraction de la racine carrée en représentation à virgule flottante et les caratéristiques de base de sa réalisation matérielle ont été présentés en [73].

À notre connaissance, aucun auteur n'a présenté de réalisation sérielle classique (en représentation signe, mantisse binaire, exposant) de l'arithmétique à virgule flottante. Nous avons investigué cette possibilité au cours de la présente recherche, tant en mode bit de poids faible en tête qu'en mode bit de poids fort en tête. Dans le cadre du Projet, nous avons développé des opérateurs à virgule flottante fonctionnant bit de poids faible en tête, et les économies d'espace obtenues sont intéressantes, malgré une latence élevée. Les résultats de mise en œuvre seront présentés dans le compte-rendu de projet, au prochain chapitre. De plus, nous avons découvert une solution pour réaliser des opérateurs à virgule flottante sériels en représentation signe, mantisse binaire, exposant, fonctionnant bit de poids fort en tête, sans avoir recours à une représentation numérique redondante [15].

2.6 Calcul des fonctions élémentaires

Les fonctions dites *élémentaires* sont les fonctions telles que l'extraction de la racine carrée, l'exponentielle, le logarithme, le sinus et le cosinus. Peu de chercheurs ont proposé des architectures matérielles dédiées à l'évaluation des fonctions élémentaires. Ces dernières jouent cependant un rôle important en calcul scientifique et dans certains algorithmes de commande, et la mise en œuvre d'opérateurs matériels les réalisant efficacement est indispensable.

L'évaluation des fonctions élémentaires peut se faire à l'aide de différents types de méthodes, dont on compte essentiellement trois grands classes : les approximations polynomiales ou rationnelles [22,23,51], les algorithmes à base d'additions et de décalages (CORDIC) [58, 86] et les méthodes à base de tables [87]. Il existe aussi des mélanges de ces trois principaux types de méthodes. On peut trouver un traitement détaillé de toutes ces méthodes dans [71].

Dans sa thèse de doctorat, Kla [51] a étudié le calcul de fonctions élémentaires en parallèle et en série. Il a présenté, pour chacune des deux approches, des algorithmes profitant de systèmes de représentation particuliers et a mis au point des opérateurs en ligne de délai très près du minimum pour plusieurs fonctions mathématiques.

C'est la méthode CORDIC qui a été retenue pour réaliser les fonctions élémentaires en psC, dans le cadre du Projet. De par sa simplicité de mise en œuvre, sa flexibilité et son efficacité, l'algorithme CORDIC, qui tire son nom du calculateur numérique à rotations de coordonnées (COordinate Rotation DIgital Computer) conçu à la fin des années 1950, est l'un des plus utilisés pour le calcul tant par logiciel que par matériel des fonctions élémentaires. C'est un algorithme de la classe des méthodes de convergence linéaire, qui a été décrit pour la première fois par Jack E. Volder en 1959. Il permettait, à l'origine, de calculer des fonctions trigonométriques telles que le sinus, le cosinus et

la fonction tangente inverse, ainsi que la fonction $\sqrt{a^2 + b^2}$. Il a plus tard été généralisé aux fonctions hyperboliques, à la multiplication et à la division.

La formulation simple de l'algorithme CORDIC est basée sur l'observation que si on fait tourner d'un angle z un vecteur unitaire ayant son sommet à $(x, y) = (1, 0)$, son nouveau sommet se trouvera à la position $(x, y) = (\cos z, \sin z)$. Par conséquent, $\cos z$ et $\sin z$ peuvent être calculés en trouvant la nouvelle position du sommet du vecteur après rotation par z.

Une première revue des mises en œuvres de l'algorithme CORDIC sur FPGA a été présentée en [8].

Parmi les nouvelles méthodes qu'il serait intéressant d'évaluer dans le futur, citons le travail de Beuchat et Tisserand [22,23,25], qui ont étudié une architecture matérielle pour l'évaluation polynomiale de fonctions élémentaires en arithmétique en-ligne sur FPGA et sa génération automatique en VHDL. Cette architecture est réutilisable pour le calcul de différentes fonctions et permet d'évaluer rapidement quelques fonctions élémentaires avec une surface de circuit modérée et un bon débit.

En collaboration avec Tisserand, nous avons commencé un travail sur une nouvelle approche d'évaluation polynomiale en-ligne pour FPGA utilisant des coefficients épars [16]. Cette méthode semble très intéressante pour l'évaluation de fonctions multiples.

Chapitre 3

Mises en œuvre

Dans ce chapitre, nous présentons des résultats de mise en œuvre pour la plupart des opérateurs que nous avons développé, tant en psC, dans le cadre du Projet, qu'en VHDL, pour d'autres expériences. Le code des principaux composants développés en psC est présenté en annexe. Quelques résultats sont tirés de la littérature.

3.1 Opérateurs arithmétiques

Les opérateurs arithmétiques développés dans le cadre du Projet sont à représentation entière, pour les uns, et à représentation à virgule flottante, pour les autres. Les opérateurs arithmétiques à virgule fixe utilisent, à la base, les mêmes circuits que leurs équivalents entiers, pourvu que la position de la virgule des deux opérandes soit préalablement alignée, et requièrent en plus, dans la plupart des cas, des circuits additionnels pour gérer les décalages et les arrondis, tâche relativement simple qu'il revient au compilateur psC d'optimiser selon la précision requise par une application donnée.

Le langage psC entend éventuellement permettre au programmeur de définir la position de la virgule d'une variable de type `fix`. Cependant, cette fonctionnalité n'était pas réalisée durant le Projet, et la virgule était considérée, par Novakod Studio, en plein centre d'un mot de donnée, ce qui empêchait d'optimiser l'étendue dynamique d'un type selon les besoins de l'application.

Le décalage d'un nombre à virgule fixe se fait directement à l'aide d'un circuit de décalage standard. Les décalages ne sont pas définis sur les nombres à virgule flottante, ce qui nécessite une conversion entière ou à virgule fixe préalable.

3.1.1 Additionneurs et soustracteurs

Dans la discussion précédente sur les additionneurs et les soustracteurs, nous avons vu que, pour des opérandes d'une largeur allant jusqu'à environ 32 bits, les additionneurs à retenue propagée standards (Fig. 2.1) offrent le meilleur compromis espace–temps sur FPGA.

La table 3.1 présente des résultats de synthèse pour la taille en cellules logiques (CL), le délai combinatoire en nanosecondes et la fréquence maximale en mégahertz, de différentes largeurs d'additionneurs standards. Les dispositifs ciblés pour la synthèse sont le Virtex xcv1000-4, le Virtex II xc2v2000-4 et le Spartan-3 xc3s5000-4.

Les résultats de synthèse des additionneurs et soustracteurs codés en psC sont présentés en Table 3.2.

TAB. 3.1 – Comparaison de différentes largeurs d'additionneurs standards sur FPGA (résultats de synthèse)

		xcv1000-4		xc2v2000-4		xc3s5000-4	
Largeur (bits)	CL	Délai (ns)	Fréq. (MHz)	Délai (ns)	Fréq. (MHz)	Délai (ns)	Fréq. (MHz)
1	2	4,140	241	1,895	527	1,714	583
2	4	4,349	229	2,079	481	1,939	515
3	6	5,577	179	3,058	327	2,842	351
4	4	5,634	177	3,081	324	2,894	345
5	6	5,691	175	3,625	275	3,310	302
6	6	5,748	173	3,678	271	3,374	296
7	8	5,805	172	3,731	267	3,438	290
8	8	5,862	170	3,784	264	3,502	285
16	16	6,318	158	4,208	237	4,014	249
18	18	6,432	155	4,314	231	4,142	241
24	24	6,774	147	4,632	215	4,526	220
32	32	7,230	138	5,056	197	5,038	198

TAB. 3.2 – Résultats de synthèse des additionneurs et soustracteurs codés en psC

Dispositif ciblé : Virtex v1000bg560-5					
Composant	Largeur des opérandes	Tranches	FF	4-LUT	Fréq. max. (MHz)
I_AddSub_SS	toutes	3	3	5	260
F_Add_SS	ser_float_t	285	385	270	86
F_Sub_SS	ser_float_t	285	385	271	86

Pour le mode parallèle, psC utilise des additionneurs et des soustracteurs à retenue propagée standards, générés automatiquement par les outils de synthèse matérielle commerciaux à partir d'une description comportementale. Le développement d'additionneurs pour opérandes larges ne figurait pas au cahier des charges du Projet.

Pour le mode sériel, un composant I_AddSub_SS a été codé en psC, réalisant un additionneur–soustracteur sériel conventionnel (Fig. 2.3). La latence entre l'entrée du premier bit des opérandes et la sortie du premier bit de la somme est d'un seul cycle d'horloge.

La mise en œuvre de l'addition et de la soustraction à virgule fixe utilise des additionneurs et des soustracteurs entiers. La position de la virgule est implicite, et il suffit d'aligner au préalable les opérandes de manière à ce que les bits de même poids soient vis-à-vis. La virgule du résultat est à la même position que celle des opérandes. La gestion de la virgule, notamment pour l'alignement et pour l'affichage des données, revient au compilateur et à l'environnement de développement psC.

Les additionneurs et soustracteurs à virgule flottante nécessitent un alignement préalable des mantisses et sont suivies d'une étape de normalisation. Dans le cadre du Projet, un additionneur et un soustracteur à virgule flottante opérant chiffre de poids faible en tête ont été réalisés en psC. Le schéma de conception des composants (F_Add_SS et

F_Sub_SS) est montré en Fig. 3.1.

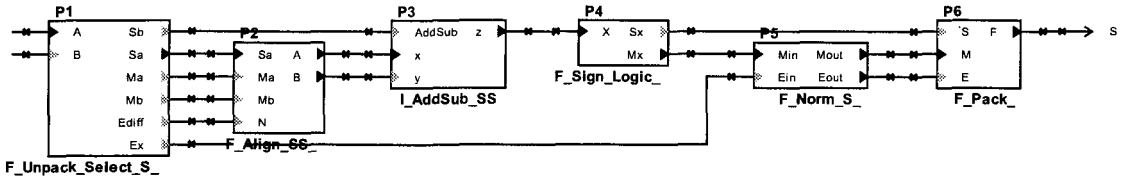

FIG. 3.1 – Schéma de conception des composants F_Add_SS et F_Sub_SS

Le type de données choisi pour véhiculer un nombre à virgule flottante simple précision de manière sérielle, ser_float_t, est codé sur 10 bits. La mantisse, de 24 bits, est transmise en série, et l'exposant, de 8 bits, est transmis en parallèle, ce qui minimise les coûts et les délais liés à sa manipulation.

3.1.2 Multiplieurs

Nous présentons des résultats de synthèse pour deux architectures de multiplieurs 32 bits parallèles du fabricant, sur différents FPGA. La première architecture (Tab. 3.3) utilise les blocs multiplieurs intégrés (MULT18x18), et chaque multiplieur requiert 4 blocs MULT18x18. La seconde architecture (Tab. 3.4) utilise exclusivement les LUT. Les caractéristiques présentées sont : le nombre de tranches, le nombre de LUT, la période minimale, τ, en nanosecondes et la fréquence maximale en mégahertz. Les tables 3.5 et 3.6 présentent des résultats de synthèse pour différentes largeurs d'opérandes de versions pipelinées des mêmes architectures de multiplieurs sur Spartan-3 xc3s5000-4.

TAB. 3.3 – Multiplieur 32 bits à base de blocs MULT18x18 (sortie enregistrée)

FPGA	Tranches	LUT	τ	Fréq.
Virtex-2P xc2vp70-5	48	92	11,947	83,7
Virtex-2 xc2v6000-4	48	92	18,123	55,1
Spartan3 xc3s5000-4	48	92	12,679	78,8

TAB. 3.4 – Multiplieur 32 bits à base de LUT (sortie enregistrée)

FPGA	Tranches	LUT	τ	Fréq.
Virtex-2P xc2vp70-5	544	1057	14,936	66,9
Virtex-2 xc2v6000-4	544	1057	17,292	57,8
Spartan3 xc3s5000-4	544	1057	16,717	59,8
Spartan2 xc2s200-5	544	1057	24,330	41,1
Virtex xcv1000-4	544	1057	26,703	37,4

TAB. 3.5 – Multiplieurs pipelinés à base de blocs MULT18x18 (Spartan-3 xc3s5000-4)

N-bits	Tranches	Bascules	LUT	τ	Fréq.
18	49	72	36	1,163	859,8
24	69	96	60	1,163	859,8
32	95	128	92	1,163	859,8

TAB. 3.6 – Multiplieurs pipelinés à base de LUT (Spartan-3 xc3s5000-4)

N-bits	Tranches	Bascules	LUT	τ	Fréq.
18	196	72	333	1,163	859,8
24	336	96	593	1,163	859,8
32	581	128	1057	1,163	859,8

Nous avons réalisé plusieurs multiplieurs en psC, dont plusieurs largeurs de multiplieurs parallèles sous forme d'arbre d'additionneurs profitant des lignes de propagation rapides des retenues (composants I_Mul_PP_8, I_Mul_PP_16 et I_Mul_PP_32), tel que montré à la Fig. 2.4, un multiplieur série–parallèle (I_MUL_SP), réalisant le circuit de la Fig. 2.7, et deux multiplieurs bit-sériels, l'un, I_MUL_SS, réalisant le circuit de la Fig. 2.8, et l'autre, celui de la Fig. 2.9.

Les résultats de synthèse des multiplieurs codés en psC sont présentés en Table 3.7.

TAB. 3.7 – Résultats de synthèse des multiplieurs codés en psC

Dispositif ciblé : Virtex v1000bg560-5					
Composant	Largeur des opérandes	Tranches	FF	4-LUT	Fréq. max. (MHz)
I_Mul_PP	8	30	9	41	80
I_Mul_PP	16	129	17	177	61
I_Mul_PP	32	403	33	737	50
I_Mul_Seq_PP	8	19	33	16	146
I_Mul_Seq_PP	16	37	65	32	124
I_Mul_Seq_PP	32	73	130	64	113
csas_mul_32uns	32	54	64	94	157
csas_mult	8	9	16	16	215
csas_mult	16	18	32	32	215
csas_mult	32	37	64	64	215
I_Mul_SP	8	7	12	8	170
I_Mul_SP	16	11	20	16	159
I_Mul_SP	32	20	36	32	141
I_Mul_SS	8	29	49	18	134
I_Mul_SS	16	51	90	34	126
I_Mul_SS	32	96	172	66	114
F_Mul_SS	ser_float_t	142	206	115	103

Le multiplieur à virgule flottante développé fonctionne bit de poids faible en tête et est basé sur multiplieur sériel non signé. La normalisation est problématique, car elle doit attendre le bit le plus significatif. La multiplication, la division et la racine carrée requièrent un décalage pour ajuster la position de la virgule après l'opération. Le schéma de conception du composant F_Mul_SS est présenté en Fig. 3.2.

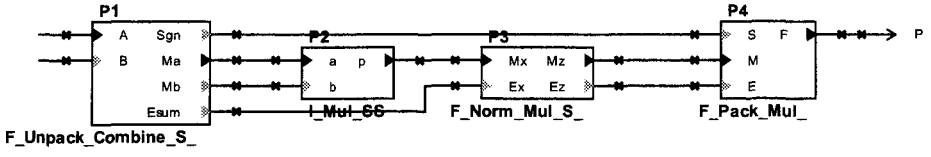

FIG. 3.2 – Schéma de conception du composant F_Mul_SS

3.1.3 Diviseurs et opérateur de racine carrée

Nous présentons des chiffres tirés de la littérature pour des mises en œuvre de diviseurs–tableaux et de diviseurs itératifs (Tab. 3.8 et 3.9). Les diviseurs itératifs non restaurants sériel et parallèle du deuxième tableau sont ceux que nous avons développé, et nous obtenons de meilleurs résultats que ceux trouvés pour les mêmes opérateurs dans la littérature.

TAB. 3.8 – Mises en œuvre de diviseurs–tableaux 32 bits (Virtex II xc2v1000-6) [83]

Mise en œuvre	N-pip.	Tranches	Bascules	Fréq.
non restorante	1	656	128	8,0
SRT base 2	1	627	128	8,2
SRT base 4	1	849	226	8,9
non restorante	8	943	688	55,4
SRT base 2	8	971	794	51,4
SRT base 4	8	988	631	50,5
non restorante	33	2000	2705	182,8
SRT base 2	33	2182	3202	182,2

TAB. 3.9 – Mises en œuvre de diviseurs itératifs 32 bits (Virtex II xc2v1000-6)

Mise en œuvre	Tranches	Bascules	LUT	Fréq.
non restor. sériel	36	35	65	227
non restor. paral.	59	105	100	205
SRT base 2 [83]	127	237	–	204
SRT base 4 [83]	184	219	–	127

I_DIV_PP est un diviseur–tableau (Fig. 2.23) restaurant. La version non restaurante et SRT restent à développer. I_Div_Seq_PP et I_Div_Seq_PP_NR sont des mises en œuvre séquentielles, respectivement de l'algorithme restaurant Fig. 2.15 et de l'algorithme non restaurant Fig. 2.16. I_Div_SP et I_Div_SS sont des diviseurs non restaurants sériels (Fig. 2.19) ; le premier, série–parallèle, et le second, série–série.

TAB. 3.10 – Résultats de synthèse des diviseurs codés en psC

Dispositif ciblé : Virtex v1000bg560-5					
Composant	Largeur des opérandes	Tranches	FF	4-LUT	Fréq. max. (MHz)
I_Div_PP	8	91	17	126	18
I_Div_PP	16	390	33	510	6
I_Div_PP	32	1621	65	2046	2
I_Div_Seq_PP	8	23	33	24	121
I_Div_Seq_PP	16	45	65	48	106
I_Div_Seq_PP	32	90	129	96	89
I_Div_Seq_PP_NR	8	14	25	16	130
I_Div_Seq_PP_NR	16	27	49	32	112
I_Div_Seq_PP_NR	32	55	98	65	102
I_Div_SP	8	12	14	20	120
I_Div_SP	16	21	22	36	105
I_Div_SP	32	38	38	68	89
I_Div_SS	8	33	43	38	118
I_Div_SS	16	52	75	54	87
I_Div_SS	32	98	140	103	81
I_Sqrt_P	32	69	85	89	109
F_Div_SS	ser_float_t	123	172	126	86

Nous avons vu qu'un diviseur fonctionne naturellement chiffre de poids fort en tête. Le diviseur à virgule flottante sériel développé est basé sur une réalisation pour FPGA de l'algorithme non restaurant. La normalisation est simplifiée par le mode d'opération chiffre de poids fort en tête. Latence est de n+K (K étant une petite constante). Le schéma de conception du composant F_Div_SS est présenté en Fig. 3.3.

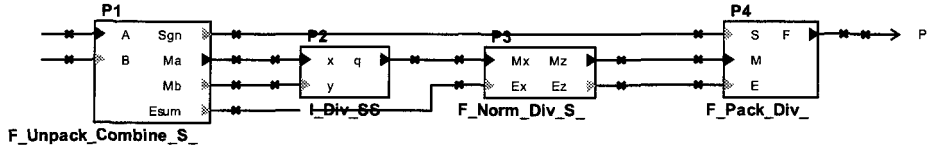

FIG. 3.3 – Schéma de conception du composant F_Div_SS

3.2 Processeur CORDIC

Les fonctions sin, cos et atan ont été réalisées en psC au moyen d'un processeur CORDIC circulaire. L'algorithme CORDIC n'utilise que des décalages, des additions et une lecture de valeurs dans une table.

Dans un grand nombre de FPGA modernes, la table peut être stockée économiquement dans l'un des blocs RAM intégrés. On peut aussi la stocker dans des LUT en mode RAM.

L'interface du composant X_CORDIC_Circ_P est montrée en Fig. 3.4.

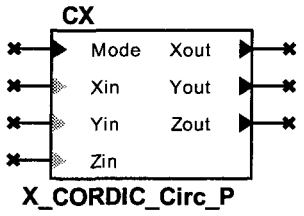

FIG. 3.4 – Interface du composant psC X_CORDIC_Circ_p

Pour obtenir sin(z) et cos(z) :

```
Entrées:

  Xin = 0.607241872
  Yin = 0.0
  Zin = un angle entre 0 et 90 degrés

Sorties:

  Xout = cos(z)
  Yout = sin(z)
  Zout = approximativement 0.0
```

Pour obtenir atan :

```
Entrées:
```

```
  Xin = 1.0
  Yin = un angle entre 0 et 90 degrés
  Zin = 0.0

Sorties:

  Xout = K sqrt(x^2 + y^2)
  Yout = approximativement 0.0
  Zout = z + atan(x/y)
```

TAB. 3.11 – Résultats de synthèse du processeur CORDIC codé en psC

Dispositif ciblé : Virtex v1000bg560-5					
Composant	Largeur des opérandes	Tranches	FF	4-LUT	Fréq. max. (MHz)
X_CORDIC_Circ_P	Fix32	523	169	989	52
X64_CORDIC_Circ_P	Fix64	1393	342	2283	41

Le processeur CORDIC développé occuperait beaucoup moins d'espace si ont trouvait une solution pour éliminer les décalages variables.

Le choix d'utiliser le type fix de psC plutôt qu'un type entier a été demandé pour rendre l'affichage des valeurs plus conviviales. Cependant, dans la version de psC utilisée durant le Projet, la virgule décimale des variables de type fix était définitivement fixée à la position médiane d'un mot, sans aucune possibilité pour le concepteur de changer sa position. Cela limitait considérablement la précision de l'opérateur dans la partie fractionnaire, une virgule en plein centre du mot attribuant inutilement un trop grand nombre de bits à la partie entière.

3.3 Étude d'opérateurs arithmétiques sériels MSBF non redondants

Effectuer des calculs massivement parallèles en matériel nécessite des modules arithmétiques très économiques en espace, qui puissent être interconnectés de manière uniforme en tableaux systoliques. Après la fin du Projet, nous nous sommes penché sur l'objectif de minimiser autant que possible l'espace occupé par des opérateurs arithmétiques opérant chiffre de poids fort en tête, ce qui demande une représentation numérique binaire non redondante. En plus de minimiser l'espace requis, nous avons aussi tenté de préserver une fréquence de fonctionnement d'au-moins 100 MHz.

3.3.1 Discussion des solutions envisageables

Notre revue en profondeur de la littérature nous a conduit à l'observation qu'il n'y a eu jusqu'à présent aucune tentative sérieuse en vue de concevoir des opérateurs arithmétiques de taille vraiment minimale, qui soient utilisables dans un grand tableau systolique, en particulier de tels opérateurs pour FPGA. Nous avons néanmoins pu identifier des pistes de solutions envisageables provenant d'autres perspectives de recherche et nous avons investigué une piste de solution pour la réalisation d'opérateurs non redondants opérant bit de poids fort en tête [15].

Il est clair que seul un mode d'opération bit de poids fort en tête (MSBF — *most significant bit first*) peut fournir l'uniformité voulue tout en minimisant l'espace d'une manière satisfaisante. Cependant, à cause de la contrainte imposée par l'utilisation d'une représentation numérique redondante, l'arithmétique en-ligne ne permet pas d'obtenir la plus grande économie d'espace possible.

Pour peu qu'il soit possible de trouver un compromis permettant d'effectuer l'addition et la soustraction bit de poids fort en tête en représentation binaire standard,

on obtiendrait une économie d'espace supérieure à celle obtenue avec l'arithmétique en-ligne. Bien qu'il y ait peu de contributions à cet effet dans la littérature, ce résultat est néanmoins possible. Dans leur article original de 1977 [91], les fondateurs de l'arithmétique en-ligne, Trivedi et Ercegovac, ont clairement énoncé que (traduction)

> *Si nous utilisions un système de numération non redondant, alors, même pour de simples opérations, telles que l'addition et la soustraction, il y aurait un délai en ligne $\delta = m$ dû à la propagation de la retenue. Pour peu que nous autorisions une redondance dans la représentation numérique, alors, il devient possible de limiter la propagation de la retenue à une seule position numérique.*

Il en découle que les opérations MSBF non redondantes sont possibles au prix d'une latence de m et d'une augmentation proportionnelle du coût en espace du circuit d'addition/soustraction.

Un premier multiplieur série–parallèle non redondant fonctionnant en mode MSBF a été introduit en 1995 par Lu et Kenney [61]. D'autres multiplieurs semblables ont été présentés en 1998 par Larsson-Edefors et Marnane [54], incluant des multiplieurs acceptant les deux opérandes en série. L'addition et la multiplication MSBF non redondante sont donc possibles, à un coût raisonnable, et cette architecture, malgré son inconvénient d'avoir une latence élevée, conduit à des réalisations économiques tout en conservant une fréquence de fonctionnement raisonnable et convient bien à certaines applications de traitement numérique de signaux.

La figure 3.5 montre une mise en œuvre d'additionneur MSBF non redondant sur FPGA.

De l'algorithme direct de multiplication série–parallèle non signée, plusieurs structures de multiplieurs MSBF non redondants peuvent être dérivés. La multiplication de

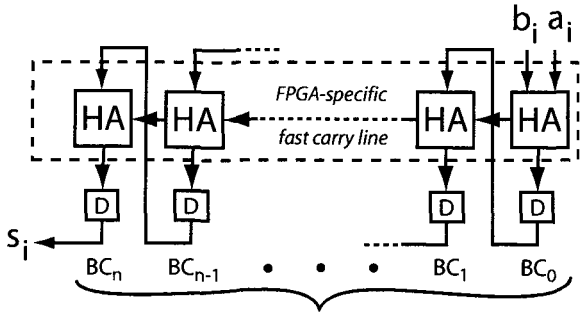

FIG. 3.5 – Additionneur MSBF non redondant utilisant la logique de propagation rapide de retenues

deux entiers non signés x et y, respectivement de m bits et n bits, donne un produit p de $(m+n)$ bits. Avec x comme opérande sériel (x_0 étant le MSB et x_{n-1} le LSB) et y comme opérande parallèle, une multiplication sérial–parallèle non signée peut s'exprimer sous la forme :

$$p = x \cdot y = \left(\sum_{j=0}^{m-1} x_j \cdot 2^{-j} \right) \cdot y = \sum_{j=0}^{m-1} (x_j \cdot y) \cdot 2^{-j}$$

La mise en œuvre de base de cet algorithme se caractérise par un chemin de propagation de retenue combinatoire (fig. 3.6). Son coût en espace est peu élevé, mais son fonctionnement est considéré relativement lent. Cette mise en œuvre est relativement simple et offre le potentiel de supporter des fréquences d'horloges supérieures à celles supportées par un multiplieur LSBF équivalent, grâce au fait que l'architecture MSBF ne comporte pas de boucle de rétroaction. En plus d'augmenter la fréquence d'horloge maximale possible en utilisant un chemin de propagation de retenues combinatoire, cette absence de boucle de rétroaction permet un pipelinage profond afin d'atteindre une fréquence élevée, rend l'architecture robuste aux problèmes de compétition et simplifie l'initialisation du multiplieur.

Tel que montré en figure 3.6, la mise en œuvre se compose de deux sections : un

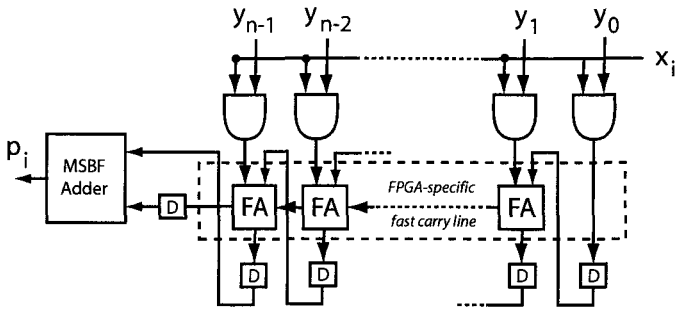

FIG. 3.6 – Multiplieur série–parallèle MSBF non redondant

additionneur de produits partiels de n tranches de bit et une additionneur final à retenue propagée de m tranches de bit. Le coût total de cette mise en œuvre est donc de $n+m$ tranches de bit, permettant de calculer le produit en double précision. La section additionneur final MSBF est requise pour extentionner la propagation de retenue afin de compléter le produit jusquà la double précision.

Selon notre expérience des architectures FPGA, ce sont généralement les structures les plus simples et directes qui atteignent les meilleures performances, tant en termes d'espace que de débit.

Considérant la réalisation directe d'un multiplieur série–parallèle ayant un chemin de propagation de retenues complètement combinatoire, on peut aisément constater comme sa structure correspond bien aux lignes de propagation de retenues rapides présente dans la plupart des FPGA. Comme nous l'avons vu dans la discussion sur les additionneurs, lorsque l'on n'utilise pas les lignes dédiées, les semi-additionneurs et les additionneurs complets coûtent chacun une tranche complète (deux LUT) : comme une LUT ne comporte qu'une seule sortie, il faut une LUT pour générer le bit de somme, et une autre pour générer le bit de retenue. Lorsqu'on utilise les lignes de propagation rapides, cependant, la génération du bit de retenue ne nécessite pas une LUT supplémentaire, pusique la retenue est prise en charge par des circuits dédiés, qui

demeureraient autrement inutilisés. Utiliser les lignes de propagation rapides permet ainsi de diminuer de moitié le coût en espace d'un additionneur dans un FPGA, ce qui peut donner une économie considérable à l'échelle d'un système.

3.3.2 Mise en œuvre et résultats

Nous avons réalisé, en VHDL, les architectures d'additionneurs et de mutliplieurs MSBF non redondants utilisant les lignes rapides de propagation de retenues afin d'évaluer leur performance sur FPGA. Les résultats, présentés en tables 3.12 et 3.13, montrent que non seulement cette architecture conduit à des réalisations très compactes, mais, tel qu'attendu, la fréquence obtenue est de plus de 100 MHz, même avec des opérandes d'une largeur atteignant 64 bits.

TAB. 3.12 – Résultats de synthèse pour l'additionneur MSBF non redondant

	N=8		N=16		N=32		N=64	
	Freq. (MHz)	**Area** (Slices)	**Freq.** (MHz)	**Area** (Slices)	**Freq.** (MHz)	**Area** (Slices)	**Freq.** (MHz)	**Area** (Slices)
SPARTAN 2 XC2S200-6	187.829	7	176.678	11	157.928	19	130.276	43
SPARTAN 3 XC3S5000-4	249.314	7	211.193	11	161.734	20	110.144	46
VIRTEX XCV1000-6	191.608	7	179.501	11	159.363	19	130.157	44
VIRTEX 2 XC2V8000-5	271.739	6	247.036	10	209.030	18	159.847	39
VIRTEX 2Pro XC2VP125-6	324.254	6	292.740	10	245.098	18	184.911	37

TAB. 3.13 – Résultats de synthèse pour le multiplieur MSBF série–parallèle non redondant

	N=8		N=16		N=32		N=64	
	Freq. (MHz)	**Area** (Slices)	**Freq.** (MHz)	**Area** (Slices)	**Freq.** (MHz)	**Area** (Slices)	**Freq.** (MHz)	**Area** (Slices)
SPARTAN 2 XC2S200-6	187.829	11	176.678	20	157.928	36	130.276	80
SPARTAN 3 XC3S5000-4	238.550	12	203.417	20	157.928	40	107.991	90
VIRTEX XCV1000-6	187.829	11	179.501	20	157.928	36	130.276	80
VIRTEX 2 XC2V8000-5	191.608	11	176.678	20	159.363	36	130.157	81
VIRTEX 2Pro XC2VP125-6	324.254	10	292.740	18	245.098	36	184.911	74

TAB. 3.14 – Résultats de synthèse pour le multiplieur MSBF non redondant entièrement sériel

	N=8		N=16		N=32		N=64	
	Freq. (MHz)	**Area** (Slices)	**Freq.** (MHz)	**Area** (Slices)	**Freq.** (MHz)	**Area** (Slices)	**Freq.** (MHz)	**Area** (Slices)
SPARTAN 2 XC2S200-6	163.026	17	139.958	30	127.926	57	109.159	116
SPARTAN 3 XC3S5000-4	216.990	18	179.195	31	142.278	61	102.098	132
VIRTEX XCV1000-6	165.865	17	141.723	30	128.866	57	109.075	116
VIRTEX 2 XC2V8000-5	252.525	16	223.814	29	192.160	58	150.240	119
VIRTEX 2Pro XC2VP125-6	269.760	16	262.674	29	223.663	58	173.130	115

Pour fins de comparaison, nous reproduisons en table 3.15 les résulats rapportés en [42] pour des multiplieurs en-ligne sur FPGA. Le premier multiplieur accepte un opérande en série, en représentation numérique redondante, et l'autre opérande en parallèle en représentation binaire en complément à 2. Le second multiplieur accepte les deux opérandes en série, en représentation numérique redondante. Ce sont des résultats de synthèse rapportés en termes de CLB pour les architectures Xilinx XC4000. Les CLB des XC4000 contiennent deux cellules logiques, équivalant à une tranche dans les architectures Virtex et Spartan.

TAB. 3.15 – Résultats de synthèse des multiplieurs série–parallèle et en-ligne de [42]

	N=8		N=12		N=16		N=32 (Estimé par extapolation)	
	Fréq. (MHz)	**Espace** (CLB)	**Fréq.** (MHz)	**Espace** (CLB)	**Fréq.** (MHz)	**Espace** (CLB)	**Fréq.** (MHz)	**Espace** (CLB)
Multiplieur en-ligne série–parallèle								
XC4000E-1	91	10	80	12	72	14	–	22–32
XC4000XL-09	103	10	96	12	89	14	–	22–32
Multiplieur en-ligne série–série								
XC4000E-1	77	48	69	68	63	88	–	168–192
XC4000XL-09	89	48	85	68	79	88	–	168–192

Le multiplieur en-ligne série–parallèlle tire avantage des lignes rapides de propagation des retenues, réalisant ainsi des économies d'espace remarquables (50 % pour l'additionneur résiduel), et la surface totale de cet opérateur peut être considérée mini-

male pour un opérateur en-ligne redondant. Le multiplieur pleinement en-ligne, quant à lui, même en profitant autant que possible des ressources du FPGA, demeure encore assez gros. Considérant la fréquence rapportée pour ces deux multiplieurs, on observe une baisse significative à mesure que la largeur des opérandes augmente. La latence des opérateurs en-ligne est cependant minimale. En dépit de cet avantage, l'utilisation d'un multiplieur en-ligne impose une représentation numérique redondante à tous les autres modules au sein d'un système, ce qui entraîne inévitablement un surcoût général en espace. Il convient donc de considérer les avantages et les inconvénients de chacune des deux approches en fonction du compromis recherché dans une application spécifique.

Chapitre 4

Analyses de résultats en vue d'estimer la puissance de calcul

Un FPGA peut fournir une puissance de calcul phénoménale à un coût minime. Il est actuellement possible d'effectuer des milliards d'opérations par seconde avec un seul FPGA coûtant à peine une centaine de dollars. Le langage psC entend permettre d'exploiter cette puissance de calcul, qui ne cesse d'augmenter, pour l'accélération d'algorithmes, et ce, à une fraction infime du coût d'un superordinateur.

Nous présentons maintenant une stratégie permettant d'estimer, pour des calculs massivement parallèles, la puissance de calcul brute de diverses mises en œuvre d'opérateurs, indépendamment d'autres critères de performance. Cette stratégie consiste à choisir une proportion des différents opérateurs conforme à l'application visée, à remplir le FPGA ciblé d'opérateurs dans cette proportion, et à calculer le nombre d'opérations par seconde effectués en parallèle par le FPGA.

On suppose que tous les opérateurs du réseau fonctionnent à la même fréquence d'horloge : la fréquence maximale supportée par l'opérateur le plus lent du réseau. Toutes les opérations ont avantage à être pipelinées, de manière à ce que la fréquence d'horloge ne soit pas le facteur limitant.

Le temps de calcul, exprimé en nanosecondes, est le temps mis par un opérateur pour calculer un nouveau résultat après une latence initiale de remplissage du pipeline. Soit $\tau = 1/frequence$, la période d'horloge, et N, le nombre de cycles d'horloge mis (après le temps de latence initial) pour calculer un nouveau résultat. Le temps de calcul, T_{Calc}, est donné par :

$$T_{Calc} = N \cdot \tau$$

Le nombre d'opérations par seconde effectuées par un opérateur est l'inverse de son temps de calcul. Le nombre d'opérations qu'effectue un FPGA par seconde — sa puissance de calcul — est la somme des opérations effectuées par seconde par tous les opérateurs qu'il contient.

Dans ce qui suit, nous présentons des exemples d'utilisation de cette stratégie d'estimation pour l'analyse des mises en œuvre d'opérateurs arithmétiques les plus fondamentales. Pour les fins de ces exemples, nous avons choisi des mises en œuvre d'opérateurs acceptent des opérandes de 32 bits. Il est à savoir que le synthétiseur utilise 4 blocs multiplieurs MULT18x18 pour réaliser un multiplieur 32 bits.

Nous présentons des résultats d'analyse pour les FPGA Virtex-II xc2v1000-6 et Spartan-3 xc3s5000-4. Le xc2v1000, de taille relativement petite, contient 5120 tranches et 40 blocs multiplieurs. Il existe, bien entendu, des modèles de Virtex-II plus costauds. Le xc3s5000, quant à lui, avec 33280 tranches et 104 blocs multiplieurs, est le plus grand FPGA de la famille Spartan-3. Ces deux choix permettent de comparer la puissance obtenue avec les mêmes opérateurs sur une surface réduite, avec un petit nombre de blocs multiplieurs, et sur une surface importante, avec un plus grand nombre de blocs multiplieurs.

Voici la signification des titres de colonnes des tableaux :

E_{Unit} : Espace unitaire, additionneur(s) + 1 multiplieur (Tranches) ;

T_{Calc} : Temps de calcul unitaire (ns) ;

P_{Unit} : Puissance unitaire, en millions d'opérations par seconde (MOPS) ;

N : Nombre d'opérateurs par FPGA (total des additionneurs et des multiplieurs) ;

P_{Tot} : Puissance totale par FPGA, en milliards d'opérations par seconde (GOPS).

Voici la signification des abréviations :

LUT_Mul : Multiplieur à base de LUT ;

Mul_pipl : Multiplieur à base de LUT pipeliné ;

RCAD : Multiplieur sériel à retenue conservée, addition et décalage ;

RPAD : Multiplieur sériel à retenue propagée, addition et décalage ;

Div_NR_seq : Diviseur non restaurant séquentiel ;

Div_NR_pipl : Diviseur non restaurant pipeliné ;

addf, mulf, divf : Additionneur, multiplieur et diviseur à virgule flottante.

4.1 Classes de problèmes se limitant aux opérations de la famille de la multiplication et de l'addition entières et à virgule fixe

Cette classe de problèmes inclut l'addition, la soustraction, la multiplication, et l'élévation au carré. La plupart des algorithmes de traitement de signaux numériques, incluant, notamment, le filtrage, la convolution, la transformée de Fourier rapide (FFT) et la transformée en cosinus discrète (DCT), font partie de cette catégorie.

Hypothèse (Tab. 4.1) : les additionneurs et les multiplieurs sont en nombre égal.

TAB. 4.1 – Puissance de calcul : additionneurs et multiplieurs en nombre égal

Mise en œuvre	E_{Unit}	T_{Calc}	P_{Unit}	xc2v1000-6 N	xc2v1000-6 P_{Tot}	xc3s5000-4 N	xc3s5000-4 P_{Tot}
Bloc MULT18x18 pipeliné	64	5,1	197	20	3,94	52	10,2
Multiplieur sériel RCAD	40	69,5	14	256	3,68	1664	24,0
LUT-Mul pipeliné	597	5,1	198	16	3,17	110	21,8
Multiplieur sériel RPAD	23	166,3	6	464	2,79	3024	18,2
LUT-Mul enregistré	560	16,7	59	18	1,08	118	7,1
Bloc MULT18x18 enregistré	64	12,7	78	20	1,58	52	4,1

On remarque que les blocs multiplieurs sont légèrement désavantagés dans le xc3s5000. Cela est dû à l'abondance de tranches, dont le nombre est suffisamment élevé proportionnellement à la quantité de blocs multiplieurs pour fournir une plus grande puissance.

Hypothèse (Tab. 4.2) : il y a deux fois plus d'additionneurs que de multiplieurs.

TAB. 4.2 – Puissance de calcul : 2 additionneurs pour 1 multiplieur

				xc2v1000-6		xc3s5000-4	
Mise en œuvre	E_{Unit}	T_{Calc}	P_{Unit}	N	P_{Tot}	N	P_{Tot}
Bloc MULT18x18 pipeliné	127	5,1	197	30	5,91	78	15,4
Multiplieur sériel RCAD	43	69,5	14	357	5,14	2319	33,4
LUT-Mul pipeliné	613	5,1	198	24	4,75	162	32,1
Multiplieur sériel RPAD	25	166,3	6	612	3,68	3993	24,0
LUT-Mul enregistré	576	16,7	59	24	1,44	171	10,2
Bloc MULT18x18 enregistré	80	12,7	78	30	2,36	78	6,2

Hypothèse (Tab. 4.3) : il y a 10 additionneurs pour 1 multiplieur.

TAB. 4.3 – Puissance de calcul : 10 additionneurs pour 1 multiplieur

				xc2v1000-6		xc3s5000-4	
Mise en œuvre	E_{Unit}	T_{Calc}	P_{Unit}	N	P_{Tot}	N	P_{Tot}
Bloc MULT18x18 pipeliné	255	5,08	197	110	21,67	286	56,3
LUT-Mul pipeliné	741	5,05	198	66	13,07	484	95,8
Multiplieur sériel RCAD	67	69,47	14	836	12,03	5456	78,5
Multiplieur sériel RPAD	49	166,25	6	1144	6,88	7469	44,9
LUT-Mul enregistré	704	16,72	59	77	4,60	517	30,9
Bloc MULT18x18 enregistré	208	12,69	78	110	8,67	286	22,5

À nombre égal de multiplieurs, l'utilisation des blocs multiplieurs intégrés fournit, dans tous les cas, la plus grande puissance de calcul. Il faut considérer d'autres architectures de multiplieurs lorsque des blocs multiplieurs intégrés ne sont pas disponibles

ou sont en nombre insuffisant. Lorsqu'il y a peu d'additionneurs comparativement au nombre de multiplieurs, les multiplieurs sériels RCAD (à retenue conservée, addition et décalage) fournissent le plus de puissance, suivis de près par les multiplieurs parallèles à base de LUT pipelinés. Lorsque le nombre d'additionneurs augmente, cependant, les multiplieurs parallèles pipelinés à base de LUT prennent les devants. Lorsque seule l'addition et la multiplication sont requises, des opérateurs sériels opérant chiffre de poids fort en tête sont utiles uniquement pour des calculs nécessitant un nombre limité des chiffres les plus significatifs des résultats. De cette manière, les calculs n'ont pas à se poursuivre au-delà.

4.2 Classes de problèmes comportant des divisions entières et à virgule fixe

Cette classe de problèmes inclut toutes les opérations arithmétiques de la classe précédente plus la division. La commande en boucle, ainsi que divers algorithmes et calculs scientifiques, incluent des divisions.

Hypothèse (Tab. 4.4) : 5 additionneurs, 5 multiplieurs et 1 diviseur.

TAB. 4.4 – Puissance de calcul : 5 additionneurs, 5 multiplieurs et 1 diviseur

				xc2v1000-6		xc3s5000-4	
Mise en œuvre	E_{Unit}	T_{Calc}	P_{Unit}	N	P_{Tot}	N	P_{Tot}
Mul_pipl, DIV_NR_pipl	4985	5,47	182,8	11	2,01	66	12,1
RPAD_Mul, ser_div_nr	211	145,37	6,9	264	1,82	1727	11,9
RCAD_Mul, ser_div_nr	236	145,37	6,9	231	1,59	1551	10,7
Mul_pipl, Div_NR_seq	3044	19,02	52,6	11	0,58	110	5,8

Dans ce cas particulier, l'utilisation d'un multiplieur à base de LUT pipeliné fournit la plus grande puissance de calcul. On remarque, cependant, que l'utilisation de

multiplieurs et diviseurs sériels fournit une puissance comparable, bien que légèrement inférieure. Pour des problèmes demandant un grand nombre d'opérateurs, l'utilisation d'opérateurs sériels semble donc préférable.

4.3 Classes de problèmes utilisant des opérations à virgule flottante

Plusieurs types de problèmes, notamment des calculs scientifiques, nécessitent une représentation à virgule flottante pour profiter d'une vaste étendue dynamique. Nous avons estimé la puissance de calcul de trois mises en œuvre notoires d'opérateurs à virgule flottante, celles de Detrey, Underwood et McIlhenny. Les deux premières sont des mises en œuvre parallèles. McIlhenny a présenté des opérateurs en-ligne.

En arithmétique à virgule flottante, l'addition est l'opération la plus complexe. Il n'existe pas, en arithmétique à virgule flottante, de différence notoire entre des classes de problèmes se limitant à une famille d'opérateurs, comme, par exemple, à l'addition et à la multiplication. Dans les tableaux qui suivent, nous faisons, pour chaque mise en œuvre, plusieurs hypothèses, en supposant que tous les opérateurs fonctionnent à 100 MHz. Dans le cas des opérateurs en-ligne, compte tenu de leurs caractéristiques, nous avons jugé pertinent de doubler leur fréquence à 200 MHz, fréquence qu'ils sont capables de supporter. Les résultats justifient cette décision. Il est à remarquer qu'à fréquence égale et à temps de calcul égal, seule la taille des opérateurs influe sur la puissance de calcul dans un FPGA donné.

TAB. 4.5 – Puissance de calcul : opérateurs à virgule flottante de J. Detrey

Mise en œuvre	E_{Unit}	T_{Calc}	P_{Unit}	xc2v1000-6		xc3s5000-4	
				N	P_{Tot}	N	P_{Tot}
1addf, 1mulf	804	10	100	12	1,2	82	8,2
2addf, 1mulf	1220	10	100	12	1,2	81	8,1
5addf, 1mulf	2468	10	100	12	1,2	78	7,8
2addf, 2mulf, 1divf	2717	10	100	6	0,6	72	7,2
4addf, 4mulf, 1divf	4325	10	100	11	1,1	77	7,7

TAB. 4.6 – Puissance de calcul : opérateurs à virgule flottante de K. Underwood

Mise en œuvre	E_{Unit}	T_{Calc}	P_{Unit}	xc2v1000-6		xc3s5000-4	
				N	P_{Tot}	N	P_{Tot}
1addf, 1mulf	1094	10	100	8	0,8	60	6,0
2addf, 1mulf	1590	10	100	9	0,9	60	6,0
5addf, 1mulf	3078	10	100	6	0,6	60	6,0
2addf, 2mulf, 1 divf	4117	10	100	5	0,5	40	4,0

TAB. 4.7 – Puissance de calcul : opérateurs à virgule flottante en-ligne de R. McIlhenny

Mise en œuvre	E_{Unit}	T_{Calc}	P_{Unit}	xc2v1000-6		xc3s5000-4	
				N	P_{Tot}	N	P_{Tot}
1addf, 1mulf	256	170	5,88	40	0,24	260	1,53
2addf, 1mulf	339	170	5,88	45	0,26	294	1,73
5addf, 1mulf	588	170	5,88	48	0,28	336	1,98
2addf, 2mulf, 1 divf	688	170	5,88	35	0,21	240	1,41
5addf, 5mulf, 1 divf	1456	170	5,88	33	0,19	242	1,42

Des trois mises en œuvre, c'est celle de J. Detrey qui fournit la plus grande puissance de calcul. Cet avantage est dû au fait que les opérateurs de Detrey sont plus compacts comparativement aux autres opérateurs parallèles. Malgré le fait que les opérateurs en-ligne sont les plus compacts, ils ont une puissance de calcul réduite à cause de

leur temps de calcul plus long, et ce, même au double de la fréquence. Des opérateurs en-ligne peuvent néanmoins s'avérer utiles dans des calculs qui ne nécessitent qu'un certain nombre des chiffres les plus significatifs des résultats, ce qui évite d'avoir à calculer en pleine précision. Grâce à cet avantage, il serait possible d'utiliser de plus petits opérateurs, ayant un temps de calcul réduit, pour obtenir la précision voulue.

Chapitre 5

Conclusion

Cette étude des compromis espace–temps dans les systèmes reconfigurables, qui s'inscrivait dans le cadre du développement du langage psC — un nouveau langage supportant les applications temps réel, parallèles et reconfigurables, — dresse, pour la première fois, un panorama général des mises en œuvre d'opérateurs arithmétiques sur FPGA. Le support de l'arithmétique matérielle sur FPGA en psC vise à permettre l'accélération, au moyen de circuits reconfigurables, de calculs numériques habituellement réalisés en logiciel.

Au cours de cette étude, nous avons constaté que peu d'auteurs ont proposé des modules arithmétiques vraiment économiques pour les FPGA conventionnels à base de LUTs. Les mises en œuvre d'arithmétique parallèle, autrefois trop coûteuses, sont désormais abordables sur FPGA grâce à une surface de puce accrue et à la présence d'un grand nombre de multiplieurs intégrés. Les opérateurs sériels conventionnels sont économiques, mais ne supportent que les opérations de la famille de l'addition et de la multiplication ; seuls les opérateurs en-ligne possèdent l'uniformité nécessaire tout en offrant des gains d'espace substantiels. La nécessité d'utiliser une représentation numérique redondante dans les opérateurs en-ligne entraîne un surcoût en espace, et cette arithmétique présente, de plus, des difficultés et des conversions additionnelles rarement abordées dans la littérature, et dont seulement quelques auteurs ont fait mention.

Dans le cadre du Projet, une bibliothèque de composants arithmétiques et de fonctions élémentaires correspondant aux livrables spécifiés lors des réunions officielles a été développée, testée, évaluée et livrée. Il est à savoir que le développement des composants arithmétiques en langage psC a été considérablement ralenti dû au fait que l'environnement de développement psC était lui-même en constant développement, et donc, très instable, au cours du Projet. Certaines de nos suggestions visant, notamment, à accélérer le développement des composants arithmétiques, et ce, indépendamment du

développement du langage psC lui-même, comme, par exemple, de développer et de tester des générateurs automatiques d'opérateurs arithmétiques en langage C standard ANSI plutôt qu'en psC, ou de développer et tester ces opérateurs arithmétiques en VHDL, puis de les traduire plus tard en psC, n'ont pas été acceptées par la compagnie, sous prétexte que tout développement devait être fait exclusivement en psC. La participation au déboguage et à l'amélioration des outils psC s'est donc avérée une part importante de notre travail, ainsi qu'une contribution substantielle.

Dans un travail effectué après la fin du Projet, les résultats de notre recherche nous ont permis de comprendre que l'addition et la soustraction pouvaient être effectuées chiffre de poids fort en tête dans une représentation binaire non redondante, au prix d'une latence de n. C'est un résultat qui semble méconnu, et cette découverte rend possible la réalisation d'un multiplieur de fonctionnement identique à l'additionneur. La division, quant à elle, opère naturellement chiffre de poids fort en tête. Cette famille d'opérateurs permettrait de remplacer, pour des économies d'espace accrues, l'approche en-ligne dans des applications de calcul massivement parallèle. Par ailleurs, l'utilisation de cette architecture au sein d'opérateurs à virgule flottante simplifierait considérablement les étapes d'alignement et de normalisation. Nous avons réalisé, dans cette architecture, en VHDL, des modules ayant une taille vraiment minimale et pouvant être interconnectés entre eux et avec un diviseur sériel de manière parfaitement uniforme. La latence additionnelle, d'ordre N, à payer, peut être considérée comme un compromis nécessaire, et, pour peu qu'une application donnée puisse s'en accomoder, cette architecture s'avère simple et économique.

Nous avons proposé une stratégie d'analyse de résultats en vue d'estimer la puissance de calcul brute d'opérateurs au sein d'un réseau de calcul massivement parallèle, et en avons présenté des exemples pour plusieurs classes de problèmes fondamentales. Il

ressort de ces analyses que l'arithmétique parallèle offre la plus grande puissance de calcul, mais ne permet de réaliser qu'un nombre relativement limité d'opérateurs dans un même FPGA. Dans la plupart des cas, l'arithmétique sérielle offre une puissance de calcul comparable, bien que légèrement inférieure, tout en permettant de réaliser un nombre important d'opérateurs. Globalement, on déduit que la puissance de calcul d'un FPGA est une constante et que les choix d'architectures d'opérateurs, indépendamment d'autres critères de performance, servent essentiellement à doser le compromis entre l'obtention d'une puissance de calcul maximale (arithmétique parallèle) et la possibilité de réaliser un plus grand nombre d'opérateurs (arithmétique sérielle).

Au point de vue des axes de recherche qui pourraient découler de la présente étude, comme celle-ci couvrait une bonne partie du domaine de l'arithmétique des ordinateurs ciblant les FPGA, les possibilités de recherche ainsi que les applications sont innombrables, indépendamment du contexte dans lequel l'étude a été réalisée. En ce qui nous concerne de plus près, des travaux ultérieurs pourraient chercher à concevoir une nouvelle architecture d'opérateurs à virgule flottante optimisés pour FPGA, réaliser un ensemble de réseaux systoliques en matériel pour la manipulation des matrices et des vecteurs, avec application à des problèmes concrets, tels que la mise en œuvre matérielle de la méthode de simulation par éléments finis, étudier la génération automatique de réseaux d'arithmétique matérielle optimisés à partir d'une description de haut niveau sous forme d'équations, approfondir la recherche sur les diverses mises en œuvre possibles des fonctions élémentaires sur FPGA et réaliser des réseaux d'arithmétique sur mesure pour diverses applications, notamment pour la commande en temps réel et l'accélération de calculs scientifiques.

Une quantité considérable de contributions restent encore à faire, et, au-delà du présent mémoire, nous continuons nos recherches sur l'arithmétique des ordinateurs

et l'optimisation des compromis espace–temps dans les systèmes reconfigurables. Nous voulons désormais réaliser des systèmes sur puce pour des applications spécifiques. Dans le cadre d'un doctorat sous la direction de Rachid Beguenane, de l'UQAC, et la codirection d'Arnaud Tisserand, du LIRMM, à Montpellier, nous avons entrepris, depuis janvier 2006, d'utiliser les résultats de cette recherche dans une application de commande en temps réel qui consiste à concevoir un contrôleur vectoriel de moteurs AC à induction intégré sur une puce, ayant un temps de calcul de l'ordre d'au plus 5 microsecondes, pour les modules d'entraînement sans capteur de vitesse des véhicules électriques. Nous travaillons également sur une méthode d'évaluation en-ligne des fonctions élémentaires sur FPGA par approximations polynomiales à coefficients épars et à la conception d'opérateurs à virgule flottante optimisés pour FPGA.

Annexe A

Code psC des principaux composants développés

A.1 Description des composants

Additionneurs et soustracteurs

I_AddSub_SS	Additionneur/soustracteur bit-sériel

Multiplieurs

I_MUL_PP_8	Multiplieur parallèle en arbre 8 x 8 bits = 16 bits
I_MUL_PP_16	Multiplieur parallèle en arbre 16 x 16 bits = 32 bits
I_MUL_PP_32	Multiplieur parallèle en arbre 32 x 32 bits = 64 bits
I_MUL_Seq_PP	Multiplieur parallèle séquentiel
I_MUL_SP	Multiplieur série–parallèle RPAD
I_MUL_SS	Multiplieur série–série

Diviseurs

I_DIV_PP	Diviseur–tableau restorant
I_DIV_Seq_PP	Diviseur parallèle séquentiel restaurant
I_DIV_Seq_PP_NR	Diviseur parallèle séquentiel non restaurant
I_DIV_SP	Diviseur série–parallèle non restaurant
I_DIV_SS	Diviseur série–série non restaurant

Racine carrée

I_Sqrt_P	Racine carrée parallèle non restaurante

Processeurs CORDIC

X_CORDIC_Circ_P	Processeur CORDIC en mode circulaire, type fix32
X64_CORDIC_Circ_P	Processeur CORDIC en mode circulaire, type fix64

Opérateurs à virgule flottante sériels

Ces opérateurs fonctionnent bit de poids faible en tête.

F_Add_SS	Additionneur sériel à virgule flottante
F_Mul_SS	Multiplieur sériel à virgule flottante
F_Div_SS	Diviseur sériel à virgule flottante

A.2 Additionneur–soustracteur bit-sériel

```
component I_AddSub_SS (
    in passive bit AddSub,
    in active bit x,
    in passive bit y,
    out active bit z)
{
    typedef uint:2 ui2_t;
    bit c;
    temp ui2_t ha = (ui2_t)x + (y ^ AddSub);
    temp ui2_t fa = ha + c;
    temp ui2_t fa1 = ha + AddSub;

fct(0) on x
{
        z: = bit(fa1, 0ub);
        c = bit(fa1, 1ub);
}

    always()
    {
        z = bit(fa, 0ub);
        c = bit(fa, 1ub);
    }

};
```

A.3 Multiplieurs

Différentes largeurs de multiplieurs parallèles en arbre

```
/* 8-bit Parallel Combinatorial Multiplier

   (In the current implementation, the type T can be either byte or ubyte)
*/
template <identifier T>
component I_Mul_PP_8 (
    in active T A,
in passive T B,
out active T P)
{
    const int N = 8;
    const int MSB = N - 1;
    const int seq0_MSB[] = 0 to (N - 1);
    const ubyte seq0_MSB_ub[] = 0 to (N - 1);

    /* generate partial products */
    #foreach pp_n in <seq0_MSB> bit_n in <seq0_MSB_ub>
```

```
        temp uint:N ss3_ ## pp_n = ~bit(B, bit_n) ? (uint:N)0 : (uint:N)A;
    #endforeach

    // level 3
    temp uint:10 ss2_0 = (uint:10)ss3_0 + ((uint:10)ss3_1 << 1ub);
    temp uint:10 ss2_1 = (uint:10)ss3_2 + ((uint:10)ss3_3 << 1ub);
    temp uint:10 ss2_2 = (uint:10)ss3_4 + ((uint:10)ss3_5 << 1ub);
    temp uint:10 ss2_3 = (uint:10)ss3_6 + ((uint:10)ss3_7 << 1ub);
    // level 2
    temp uint:12 ss1_0 = (uint:12)ss2_0 + ((uint:12)ss2_1 << 2ub);
    temp uint:12 ss1_1 = (uint:12)ss2_2 + ((uint:12)ss2_3 << 2ub);
    // level 1
    temp uint:16 ss0_0 = (uint:16)ss1_0 + ((uint:16)ss1_1 << 4ub);

    fct(0) on A
    {
        P: ;
    }

    always()
    {
        P = (T)ss0_0;
    }

};

/* 16-bit Parallel Combinatorial Multiplier

   (In the current implementation, the type T can be either short or ushort)
*/
template <identifier T>
component I_Mul_PP_16 (
    in active T A,
in passive T B,
out active T P)
{
    const int N = 16;
    const int MSB = N - 1;
    const int seq0_MSB[] = 0 to (N - 1);
    const ubyte seq0_MSB_ub[] = 0 to (N - 1);

    /* generate partial products */
    #foreach pp_n in <seq0_MSB> bit_n in <seq0_MSB_ub>
        temp uint:N ss4_ ## pp_n = ~bit(B, bit_n) ? (uint:N)0 : (uint:N)A;
    #endforeach

    // level 4
    temp uint:18 ss3_0 = (uint:18)ss4_0 + ((uint:18)ss4_1 << 1ub);
    temp uint:18 ss3_1 = (uint:18)ss4_2 + ((uint:18)ss4_3 << 1ub);
    temp uint:18 ss3_2 = (uint:18)ss4_4 + ((uint:18)ss4_5 << 1ub);
    temp uint:18 ss3_3 = (uint:18)ss4_6 + ((uint:18)ss4_7 << 1ub);
```

```
    temp uint:18 ss3_4 = (uint:18)ss4_8 + ((uint:18)ss4_9 << 1ub);
    temp uint:18 ss3_5 = (uint:18)ss4_10 + ((uint:18)ss4_11 << 1ub);
    temp uint:18 ss3_6 = (uint:18)ss4_12 + ((uint:18)ss4_13 << 1ub);
    temp uint:18 ss3_7 = (uint:18)ss4_14 + ((uint:18)ss4_15 << 1ub);
    // level 3
    temp uint:20 ss2_0 = (uint:20)ss3_0 + ((uint:20)ss3_1 << 2ub);
    temp uint:20 ss2_1 = (uint:20)ss3_2 + ((uint:20)ss3_3 << 2ub);
    temp uint:20 ss2_2 = (uint:20)ss3_4 + ((uint:20)ss3_5 << 2ub);
    temp uint:20 ss2_3 = (uint:20)ss3_6 + ((uint:20)ss3_7 << 2ub);
    // level 2
    temp uint:24 ss1_0 = (uint:24)ss2_0 + ((uint:24)ss2_1 << 4ub);
    temp uint:24 ss1_1 = (uint:24)ss2_2 + ((uint:24)ss2_3 << 4ub);
    // level 1
    temp uint:32 ss0_0 = (uint:32)ss1_0 + ((uint:32)ss1_1 << 8ub);

    fct(0) on A
    {
        P: ;
    }

    always()
    {
        P = (T)ss0_0;
    }

};

/* 32-bit Parallel Combinatorial Multiplier

   (In the current implementation, the type T can be either int or uint)
*/

template <identifier T>
component I_Mul_PP_32 (
    in active T A,
in passive T B,
out active T P)
{
    const int N = 32;
    const int MSB = N - 1;
    const int seq0_MSB[] = 0 to (N - 1);
    const ubyte seq0_MSB_ub[] = 0 to (N - 1);

    /* generate partial products */
    #foreach pp_n in <seq0_MSB> bit_n in <seq0_MSB_ub>
        temp uint:N ss5_ ## pp_n = bit(B, bit_n) ? (uint:N)A : (uint:N)0;
    #endforeach

    // level 5
    temp uint:34 ss4_0 = (uint:34)ss5_0 + ((uint:34)ss5_1 << 1ub);
    temp uint:34 ss4_1 = (uint:34)ss5_2 + ((uint:34)ss5_3 << 1ub);
```

```
    temp uint:34 ss4_2 = (uint:34)ss5_4 + ((uint:34)ss5_5 << 1ub);
    temp uint:34 ss4_3 = (uint:34)ss5_6 + ((uint:34)ss5_7 << 1ub);
    temp uint:34 ss4_4 = (uint:34)ss5_8 + ((uint:34)ss5_9 << 1ub);
    temp uint:34 ss4_5 = (uint:34)ss5_10 + ((uint:34)ss5_11 << 1ub);
    temp uint:34 ss4_6 = (uint:34)ss5_12 + ((uint:34)ss5_13 << 1ub);
    temp uint:34 ss4_7 = (uint:34)ss5_14 + ((uint:34)ss5_15 << 1ub);
    temp uint:34 ss4_8 = (uint:34)ss5_16 + ((uint:34)ss5_17 << 1ub);
    temp uint:34 ss4_9 = (uint:34)ss5_18 + ((uint:34)ss5_19 << 1ub);
    temp uint:34 ss4_10 = (uint:34)ss5_20 + ((uint:34)ss5_21 << 1ub);
    temp uint:34 ss4_11 = (uint:34)ss5_22 + ((uint:34)ss5_23 << 1ub);
    temp uint:34 ss4_12 = (uint:34)ss5_24 + ((uint:34)ss5_25 << 1ub);
    temp uint:34 ss4_13 = (uint:34)ss5_26 + ((uint:34)ss5_27 << 1ub);
    temp uint:34 ss4_14 = (uint:34)ss5_28 + ((uint:34)ss5_29 << 1ub);
    temp uint:34 ss4_15 = (uint:34)ss5_30 + ((uint:34)ss5_31 << 1ub);
    // level 4
    temp uint:36 ss3_0 = (uint:36)ss4_0 + ((uint:36)ss4_1 << 2ub);
    temp uint:36 ss3_1 = (uint:36)ss4_2 + ((uint:36)ss4_3 << 2ub);
    temp uint:36 ss3_2 = (uint:36)ss4_4 + ((uint:36)ss4_5 << 2ub);
    temp uint:36 ss3_3 = (uint:36)ss4_6 + ((uint:36)ss4_7 << 2ub);
    temp uint:36 ss3_4 = (uint:36)ss4_8 + ((uint:36)ss4_9 << 2ub);
    temp uint:36 ss3_5 = (uint:36)ss4_10 + ((uint:36)ss4_11 << 2ub);
    temp uint:36 ss3_6 = (uint:36)ss4_12 + ((uint:36)ss4_13 << 2ub);
    temp uint:36 ss3_7 = (uint:36)ss4_14 + ((uint:36)ss4_15 << 2ub);
    // level 3
    temp uint:40 ss2_0 = (uint:40)ss3_0 + ((uint:40)ss3_1 << 4ub);
    temp uint:40 ss2_1 = (uint:40)ss3_2 + ((uint:40)ss3_3 << 4ub);
    temp uint:40 ss2_2 = (uint:40)ss3_4 + ((uint:40)ss3_5 << 4ub);
    temp uint:40 ss2_3 = (uint:40)ss3_6 + ((uint:40)ss3_7 << 4ub);
    // level 2
    temp uint:48 ss1_0 = (uint:48)ss2_0 + ((uint:48)ss2_1 << 8ub);
    temp uint:48 ss1_1 = (uint:48)ss2_2 + ((uint:48)ss2_3 << 8ub);
    // level 1
    temp uint:64 ss0_0 = (uint:64)ss1_0 + ((uint:64)ss1_1 << 16ub);

    fct(0) on A
    {
        P: ;
    }

    always()
    {
        P = (T)ss0_0;
    }

};
```

Multiplieur parallèle séquentiel

```
/* Word-Parallel Iterative Multiplier
*/
```

```
template <ubyte S, identifier T>
component I_Mul_Seq_PP (
    in active T A,
in passive T B,
out active T P
)
{
    const ubyte Sm1 = S - 1ub;
    const ubyte Sp1 = S + 1ub;

    uint:S PP;
    uint:S M;
    uint:S Count;

fct(0) on A
{
        PP = (uint:S)0;
        P = B;
        M = (uint:S)A;
        Count = (uint:S)1t;
}

    temp uint:Sp1 Sum = (uint:Sp1)PP + (uint:Sp1)(bit(P, 0ub) ? (M) : (0));

    always()
    {
        PP = (uint:S)(Sum >> 1ub);
        P = (T)(((uint:S)bit(Sum, 0ub) << Sm1) | ((uint:S)P >> 1ub));
        if (bit(Count, Sm1)) {
            P: ;
        };
        Count <<= 1ub;
    }

};
```

Multiplieur série–parallèle

```
template <ubyte S, identifier Ser_T, identifier Par_T>
component I_Mul_SP (
    in active Ser_T x,
    in passive Par_T Y,
    out active Ser_T p)
{
    const ubyte Sp1 = S + 1ub;
    typedef uint:S ppadd_t;
    typedef uint:Sp1 reg_t;

    bit ev;
    reg_t R;
```

```
    temp ppadd_t PP = (ppadd_t)(x ? ((int:S)-1) : ((int:S)0)) & (ppadd_t)Y;

    always()
    {
        R = (reg_t)bits(R, S, 1ub) + PP;
        p = bit(R, 0ub);
        if (ev) {
            p: ;
        };
        ev = 0t;
    }

fct(0) on x
{
        R = (reg_t)PP;
        ev = 1t;
}

};
```

Multiplieur bit-sériel

```
template <ubyte S, identifier Ser_T>
component I_Mul_SS(
    in active Ser_T a,

in passive Ser_T b,

out active Ser_T p)

{
    const int Sm1 = S - 1;

    typedef uint:S   xPar_T;
    typedef uint:Sm1 yPar_T;


    template <int S, identifier Ser_T, identifier xPar_T, identifier yPar_T>
    component SP2_Loader_ (
        in active Ser_T x,
     in passive Ser_T y,
     out active Ser_T x_ser,
     out passive xPar_T x_par,
     out active Ser_T y_ser,
     out passive yPar_T y_par)
    {
        xPar_T ldMask;

        temp xPar_T xMask = ldMask & (int:S)((bit)x ? (-1) : (0));
```

```
    temp xPar_T yMask = ldMask & (int:S)((bit)x_ser ? (-1) : (0));

    fct(0) on x
    {
        x_ser: ;
        x_par  = (xPar_T)x;
        y_ser: ;
        y_par  = (yPar_T)0;
        ldMask = (xPar_T)1;
    }

    always()
    {
        x_ser = (Ser_T)y;
        x_par = (xPar_T)((~(ldMask << 1ub) & x_par) | (xMask << 1ub));
        y_ser = (Ser_T)x;
        y_par = (yPar_T)((~ldMask & y_par) | yMask);
        ldMask <<= 1ub;
    }

};

template <int S, identifier Ser_T, identifier Par_T>
component I_Mul_SP (
    in active Ser_T x,
    in passive Par_T Y,
    out active Ser_T p)
{
    const ubyte Sp1 = (ubyte)S + 1ub;
    typedef uint:S ppadd_t;
    typedef uint:Sp1 reg_t;

    bit ev;
    reg_t R;

    temp ppadd_t PP = (ppadd_t)(x ? ((int:S)-1) : ((int:S)0)) & (ppadd_t)Y;


 fct(0) on x
 {
        //R = (reg_t)PP;
        ev = 1t;
 }

    always()
    {
        R = (reg_t)bits(R, (ubyte)S, 1ub) + PP;
        p = (Ser_T)bit(R, 0ub);
        if (ev) {
            p: ;
```

```
            };
            ev = 0t;
        }

    };

    template <int S, identifier Ser_T>
    component I_Add_SS (
        in active Ser_T x,
        in passive Ser_T y,
        out active Ser_T z)
    {
        typedef uint:2 ui2_t;
        bit c;
        temp ui2_t ha = (ui2_t)(bit)x + (bit)y;
        temp ui2_t fa = ha + c;

     fct(0) on x
     {
            z: = (Ser_T)bit(ha, 0ub);
            c = bit(ha, 1ub);
     }

        always()
        {
            z = (Ser_T)bit(fa, 0ub);
            c = bit(fa, 1ub);
        }

    };

    SP2_Loader_ < S, Ser_T, xPar_T, yPar_T > P1;
    I_Mul_SP < S, Ser_T, xPar_T > P2;
    I_Mul_SP < Sm1, Ser_T, yPar_T > P3;
    I_Add_SS < S, Ser_T > P4;

    bit signal0 = {a, P1.x};
    bit signal1 = {b, P1.y};
    bit signal2 = {P4.z, p};
    bit signal3 = {P1.x_ser, P2.x};
    xPar_T signal4 = {P1.x_par, P2.Y};
    bit signal5 = {P1.y_ser, P3.x};
    yPar_T signal6 = {P1.y_par, P3.Y};
    bit signal7 = {P2.p, P4.x};
    bit signal8 = {P3.p, P4.y};
};
```

A.4 Diviseurs et racine carrée

Diviseur–tableau restorant

```
/* Combinatorial Divider
*/
template <ubyte S, identifier T>
component I_Div_PP (
    in active T Z,
in passive T D,
out active T Q,
out passive T R
)
{
    const int Last = (int)S;
    const ubyte Sm1 = S - 1ub;
    const ubyte Sp1 = S + 1ub;
    const int Loop0[] = 1 to (int)Sm1 step 1;
    const int Loop1[] = 2 to (int)S step 1;

    temp uint:S Rsh0 = (uint:S)bit(Z, Sm1);
    temp int:Sp1 W1 = ((int:Sp1)Rsh0) - (int:Sp1)D;
    temp bit qb1 = ~bit(W1, Sm1);
    temp uint:S R1 = (qb1 ? ((uint:S)W1) : (Rsh0));
    temp uint:S Q1 = ((uint:S)Z << 1ub) | (uint:S)qb1;

    #foreach Prev in <Loop0> Cur in <Loop1>
        temp uint:S Rsh##Prev = (uint:S)((R##Prev << 1ub) | bit(Q##Prev, Sm1));
        temp int:Sp1 W##Cur = (int:Sp1)Rsh##Prev - (int:Sp1)D;
        temp bit qb##Cur = bit(R##Prev, Sm1) | ~bit(W##Cur, Sm1);
        temp uint:S R##Cur = (qb##Cur ? ((uint:S)W##Cur) : (Rsh##Prev));
        temp uint:S Q##Cur = (Q##Prev << 1ub) | (uint:S)qb##Cur;
    #endforeach

fct(0) on Z
{
        Q: ;
}

    always()
    {
        Q = (T)(uint:S)Q##Last;
        R = (T)(uint:S)R##Last;
    }

};
```

Diviseurs parallèles séquentiels restorant et non restaurant

```
/* Word-Parallel Iterative Divider
```

```
*/
template <ubyte S, identifier T>
component I_Div_Seq_PP (
    in active T Z,
in passive T D,
out active T Q,
out passive T R
)
{
    const ubyte msb = S - 1ub;
    const ubyte Sp1 = S + 1ub;
    uint:S Rd;
    uint:S Count;
    temp bit cf = bit(R, msb);
    temp uint:S Rshl = (uint:S)((R << 1ub) | bit(Q, msb));
    temp int:Sp1 Diff = (int:Sp1)(uint:Sp1)Rshl - Rd;
    temp bit q_bit = cf | ~bit(Diff, msb);

fct(0) on Z
{
        R = (T)0;
        Q = Z;
        Rd = (uint:S)D;
        Count = (uint:S)1t;
}

    always()
    {
        R = q_bit ? ((T)Diff) : ((T)Rshl);
        Q = (T)((Q << 1ub) | (T)(uint:S)q_bit);
        if (bit(Count, msb)) {
            Q: ;
        };
        Count <<= 1ub;
    }

};
```

Diviseurs série–parallèle non restaurant

```
template <int S, identifier Ser_T, identifier Par_T>
component I_Div_SP (
    in active Ser_T x,
in passive Par_T Y,
out active Ser_T q)
{
    const ubyte Sm2 = (ubyte)(S - 2);
    const ubyte Sm1 = (ubyte)(S - 1);
    const ubyte Sp1 = (ubyte)(S + 1);
    const ubyte Sp2 = (ubyte)(S + 2);
```

```
    bit rx = 0t;
    bit mb = 0t;
    uint:Sp1 R = 0;
    uint:Sp1 rct = 0;

    temp uint:Sp1 Rshl = (uint:Sp1)((R << 1ub) | rx);
    temp int:Sp2 Diff = (int:Sp2)(uint:Sp2)Rshl - Y;
    temp bit q_tmp = mb | ~bit(Diff, (ubyte)Sp1);

    fct (0) on x
    {
        R = (uint:Sp1)0;
        rct = (uint:Sp1)1;
        q = 0t;
    }

    always ()
    {
        rx = x;
        mb = bit(R, (ubyte)S);
        R = (uint:Sp1)(q_tmp ? (Diff) : (Rshl));

        q = q_tmp;
        if (bit(rct, 0ub))
        {
            q: ;
        };
        rct <<= 1ub;
    }

};
```

Diviseurs série–série non restaurant

```
template <int S, identifier Ser_T>
component I_Div_SS (
    in active Ser_T x,
in passive Ser_T y,
out active Ser_T q,
    out passive ushort test)
{
    const ubyte Sm3 = (ubyte)(S - 3);
    const ubyte Sm2 = (ubyte)(S - 2);
    const ubyte Sm1 = (ubyte)(S - 1);
    const ubyte Sp1 = (ubyte)(S + 1);
    const ubyte Sp2 = (ubyte)(S + 2);

    bit ld = 0t;
    bit mb = 0t;
```

```
bit dx = 0t;
bit dy = 0t;
uint:Sp1 Rx = 0;
uint:Sp1 Ry = 0;
uint:Sp1 R = 0;
uint:Sp2 rcnt = 0;

temp uint:Sp1 Rshl = (uint:Sp1)((R << 1ub) | bit(Rx, (ubyte)S));
temp int:Sp2 Diff = (int:Sp2)(uint:Sp2)Rshl - Ry;
temp bit q_tmp = mb | ~bit(Diff, (ubyte)Sp1);

fct (0) on x
{
    ld = 0t;
    q = 0t;
    Rx = (uint:Sp1)0; //(Rx << 1ub) | 0t;
    Ry = (uint:Sp1)0; //(Ry << 1ub) | 0t;
    R = (uint:Sp1)0;
    rcnt = (rcnt << 1ub) | 1t;
}

always ()
{
    dx = x;
    dy = y;
    mb = bit(R, (ubyte)S);
    ld |= bit(rcnt, (ubyte)Sm1);
    R = (uint:Sp1)((q_tmp & ld) ? (Diff) : (Rshl));

    if (!ld)
    {

        Rx = (Rx << 1ub) | dx;
        Ry = (Ry << 1ub) | dy;
    }
    else
    {
        Rx = (Rx << 1ub) | 0t;
        Ry = Ry;
    };

    q = q_tmp;
    if (bit(rcnt, (ubyte)Sp1))
    {
        q: ;
    };
    rcnt <<= 1ub;

    test = (ushort)Rx;
}
```

```
};
```

Racine carrée non restaurante

```
/* 32-bit Nonrestoring Square Root Operator
*/
component I_Sqrt_P (
    in active uint D,
    out active uint Q,
    out active uint R)
{
    uint Dreg;
    ushort count;

    temp uint:18 Atmp = ((uint:18)R << 2ub)
     | (uint:18)bits(Dreg, 31ub, 30ub);
    temp uint:18 Btmp = ((uint:18)Q << 2ub)
     | (((uint:18)bit(R, 18ub)) << 1ub) | (uint:18)1t;
    temp uint:19 Stmp = bit(R, 18ub) ?
     ((uint:19)Atmp + Btmp) : ((uint:19)Atmp - Btmp);

    fct(0) on D
    {
        Dreg = D;
        Q = 0ui;
        R = 0ui;
        count = (ushort)1;
    }

    always()
    {
        Dreg <<= 2ub;
        Q = (uint)(((ushort)Q << 1ub) | ~bit(Stmp, 18ub));
        R = (uint)Stmp;
        if (bit(count, 15ub))
        {
            Q: ;
        };
        count <<= 1ub;
    }

};
```

A.5 Processeurs CORDIC

```
/* A First Prototype of a Fix32 Circular CORDIC Operator

   16 bits of fractional precision, giving three decimal digits of accuracy
```

```
   To obtain sin(z) and cos(z):

      Inputs:

        Xin = 0.607241872
        Yin = 0.0
        Zin = angle between 0 and 90 degrees

      Outputs:

        Xout = cos(z)
        Yout = sin(z)
        Zout = approximately 0.0

*/
component X_CORDIC_Circ_P (
    in active bit Mode,
in passive fix32 Xin,
in passive fix32 Yin,
in passive fix32 Zin,
out active fix32 Xout,
out active fix32 Yout,
out active fix32 Zout)
{
    fix32 lookup[17];

    start()
    {
        // arctan(2^-j)
        lookup[0] = 45.0x32;
        lookup[1] = 26.56505118x32;
        lookup[2] = 14.03624347x32;
        lookup[3] = 7.125016349x32;
        lookup[4] = 3.576334375x32;
        lookup[5] = 1.789910608x32;
        lookup[6] = 0.895173710x32;
        lookup[7] = 0.447614171x32;
        lookup[8] = 0.223810500x32;
        lookup[9] = 0.111905677x32;
        lookup[10] = 0.055952892x32;
        lookup[11] = 0.027976453x32;
        lookup[12] = 0.013988227x32;
        lookup[13] = 0.006994114x32;
        lookup[14] = 0.003497057x32;
        lookup[15] = 0.001748528x32;
        lookup[16] = 0.000874264x32;
    }

    ubyte j;
```

```
    temp bit di = Zout < 0.0x32;
    temp fix32 Xsr = Xout >> j;
    temp fix32 Ysr = Yout >> j;

    fct(0) on Mode
    {
        Xout = Xin;
        Yout = Yin;
        Zout = Zin;
        j = 0ub;
    }

    always()
    {
        if (di)
        {
            Xout = Xout + Ysr;
            Yout = Yout - Xsr;
            Zout = Zout + lookup[j];
        }
        else
        {
            Xout = Xout - Ysr;
            Yout = Yout + Xsr;
            Zout = Zout - lookup[j];
        };

        if (j == 15ub)
        {
            Xout: ;
            Yout: ;
            Zout: ;
        };

        if (j < 16ub)
        {
            j++;
        };

    }

};

/* A Prototype Fix64 Circular CORDIC Operator

   32 bits of fractional precision, giving eigth decimal digits of accuracy
   (The last digit may bear an error of plus or minus 1)

   To obtain sin(z) and cos(z):
```

```
      Inputs:

        Xin = 0.607252935
        Yin = 0.0
        Zin = angle between 0 and 90 degrees

      Outputs:

        Xout = cos(z)
        Yout = sin(z)
        Zout = approximately 0.0

*/
component X64_CORDIC_Circ_P (
in passive fix64 Xin,
in passive fix64 Yin,
in active fix64 Zin,
out active fix64 Xout,
out active fix64 Yout,
out passive fix64 Zout)
{
    fix64 lookup[33];

    start()
    {
        // arctan(2^-j)
        lookup[0] = 45.0x64;
        lookup[1] = 26.56505118x64;
        lookup[2] = 14.03624347x64;
        lookup[3] = 7.125016349x64;
        lookup[4] = 3.576334375x64;
        lookup[5] = 1.789910608x64;
        lookup[6] = 0.895173710x64;
        lookup[7] = 0.447614171x64;
        lookup[8] = 0.223810500x64;
        lookup[9] = 0.111905677x64;
        lookup[10] = 0.055952892x64;
        lookup[11] = 0.027976453x64;
        lookup[12] = 0.013988227x64;
        lookup[13] = 0.006994114x64;
        lookup[14] = 0.003497057x64;
        lookup[15] = 0.001748528x64;
        lookup[16] = 0.000874264x64;
        lookup[17] = 0.000437132x64;
        lookup[18] = 0.000218566x64;
        lookup[19] = 0.000109283x64;
        lookup[20] = 0.000054642x64;
        lookup[21] = 0.000027321x64;
        lookup[22] = 0.000013660x64;
        lookup[23] = 0.000006830x64;
```

```
    lookup[24] = 0.000003415x64;
    lookup[25] = 0.000001708x64;
    lookup[26] = 0.000000854x64;
    lookup[27] = 0.000000427x64;
    lookup[28] = 0.000000213x64;
    lookup[29] = 0.000000107x64;
    lookup[30] = 0.000000053x64;
    lookup[31] = 0.000000027x64;
    lookup[32] = 0.000000000x64;
}

ubyte j;

temp bit di = Zout < 0.0x64;
temp fix64 Xsr = Xout >> j;
temp fix64 Ysr = Yout >> j;

fct(0) on Zin
{
    Xout = Xin;
    Yout = Yin;
    Zout = Zin;
    j = 0ub;
}

always()
{
    if (di)
    {
        Xout = Xout + Ysr;
        Yout = Yout - Xsr;
        Zout = Zout + lookup[j];
    }
    else
    {
        Xout = Xout - Ysr;
        Yout = Yout + Xsr;
        Zout = Zout - lookup[j];
    };

    if (j == 31ub)
    {
        Xout: ;
        Yout: ;
    };

    if (j < 32ub)
    {
        j++;
    };
```

```
    }
};
```

A.6 Opérateurs à virgule flottante sériels

Additionneur

```
component F_Add_SS(in active uint:10 A,

in passive uint:10 B,

out active uint:10 S)

{


    /* WARNING: ROUNDING AND HANDLING OF SPECIAL CASES NOT IMPLEMENTED
    */
    component F_Unpack_Select_S_ (
        in active uint:10 A,
     in passive uint:10 B,
        out active bit Sb,
     out active bit Sa,
     out active bit Ma,
     out passive bit Mb,
     out passive ubyte Ediff,
        out active ubyte Ex)
    {
        // unpack
        temp bit Sa_tmp = bit(A, 9ub);
        temp bit Sb_tmp = bit(B, 9ub);
        temp bit Ma_tmp = bit(A, 8ub);
        temp bit Mb_tmp = bit(B, 8ub);
        temp ubyte Ea_tmp = (ubyte)bits(A, 7ub, 0ub);
        temp ubyte Eb_tmp = (ubyte)bits(B, 7ub, 0ub);

        // subtract exponents
        temp int:9 diff1 =  (int:9)(uint:9)Ea_tmp - (int:9)(uint:9)Eb_tmp;
        temp int:9 diff2 =  (int:9)(uint:9)Eb_tmp - (int:9)(uint:9)Ea_tmp;
        temp bit is_neg = bit(diff1, 8ub);

        fct(0) on A
        {
            // conditional swap
            if (is_neg)
            {
                Sb: = Sa_tmp;
```

```
            Sa = Sb_tmp;
            Ex = Eb_tmp;
            Ediff = (ubyte)diff2;
        }
        else
        {
            Sb: = Sb_tmp;
            Sa = Sa_tmp;
            Ex = Ea_tmp;
            Ediff = (ubyte)diff1;
        };

        Ma: ;
    }

    always()
    {
        // conditional swap
        if (is_neg)
        {
            Ma = Mb_tmp;
            Mb = Ma_tmp;
        }
        else
        {
            Ma = Ma_tmp;
            Mb = Mb_tmp;
        };
    }

};


template <ubyte S, identifier T>
component I_AddSub_SS (
    in passive bit AddSub,
    in active bit x,
    in passive bit y,
    out active bit z)
{
    typedef uint:2 ui2_t;
    bit c;
    temp ui2_t ha = (ui2_t)x + (y ^ AddSub);
    temp ui2_t fa = ha + c;
    temp ui2_t fa1 = ha + AddSub;

 fct(0) on x
 {
        z: = bit(fa1, 0ub);
        c = bit(fa1, 1ub);
```

```
 }

    always()
    {
        z = bit(fa, 0ub);
        c = bit(fa, 1ub);
    }

};


component F_Align_SS_ (
    in passive bit Sa,
    in active bit Ma,
    in passive bit Mb,
    in passive ubyte N,
    out active bit A,
    out passive bit B)
{
    ubyte Count;
    uint:49 Areg;
    uint:25 Ev;
    uint:48 Breg;
    uint:48 Bload;
    bit done;

    bit c;
    temp uint:2 ha0 = (uint:2)(Ma ^ Sa) + Sa;
    temp uint:2 haN = (uint:2)(Ma ^ Sa) + c;

    temp uint:48 Btmp = (done ? (Breg) : ((Breg & ~Bload)
     | ((uint:48)(Mb ? (-1) : (0)) & Bload)));

    fct(0) on Ma
    {
        A = 0t;
        Areg = (uint:48)bit(ha0, 0ub);
        c = bit(ha0, 1ub);

        if (N > 48)
        {
            Bload = (uint:48)0;
            Count = 0ub;
        }
        else
        {
            B = 0t;
            Breg = (uint:48)Mb;
            Bload = (uint:48)2;
            Count = 49ub - N;
```

```
        };

        Ev = (uint:25)1;
        done = 0t;
    }

    always()
    {
        A = bit(Areg, 48ub);
        Areg = (Areg << 1ub) | (done ? (Sa) : (bit(haN, 0ub)));
        c = bit(haN, 1ub);
        if (bit(Ev, 24ub))
        {
            A: ;
        };
        if (Count > 1ub)
        {
            Breg = Btmp;
            B = 0t;
            Bload <<= 1ub;
            Count--;
        }
        else
        {
            B = bit(Breg, 0ub);
            Breg = Btmp >> 1ub;
        };
        if (bit(Ev, 22ub))
        {
            done = 1t;
        };
        Ev <<= 1ub;
    }

};


component F_Pack_ (
    in passive bit S,
 in active bit M,
 in passive ubyte E,
 out active uint:10 F)
{
    uint:3 D_ev;

    fct(0) on M
    {
        D_ev = (uint:3)1;
    }
```

```
    always()
    {
        setbits(F, 7ub, 0ub, (uint:33)E);
        setbit(F, 8ub, M);
        setbit(F, 9ub, S);
        if (bit(D_ev, 2ub))
        {
            F: ;
        };
        D_ev <<= 1ub;
    }
};


component F_Norm_S_ (
    in active bit Min,
    in passive ubyte Ein,
    out active bit Mout,
    out passive ubyte Eout)
{
    bit z_flag;
    bit Done;
    uint:6 Count;
    uint:24 Ev;
    uint:49 Mbuf;

    fct(0) on Min
    {
        Mbuf = (uint:49)Min;
        Mout = 0t;
        Eout = Ein - 70ub;
        Count = (uint:6)48;
        z_flag = 1t;
        Done = 0t;
    }

    always()
    {
        Mout = bit(Mbuf, 48ub);
        if (!Done)
        {
            Eout++;
        };
        if (Count)
        {
            Mbuf = (Mbuf << 1ub) | Min;
            if (Min)
            {
                Ev = (uint:24)1t;
                if (Count < 24)
```

```
                {
                    z_flag = 0t;
                };
            }
            else
            {
                Ev <<= 1ub;
            };
            Count--;
        }
        else
        {
            if (z_flag || bit(Ev, 22ub))
            {
                z_flag = 0t;
                Mout: ;
                Done = 1t;
            };
            Mbuf = (Mbuf << 1ub);
            Ev <<= 1ub;
        };
    }

};



component F_Sign_Logic_ (
    in active bit X,
 out passive bit Sx,
 out active bit Mx)
{
    uint:6 count;
    uint:50 buf;

    bit c;
    bit busy;
    bit init;
    temp uint:2 ha0 = (uint:2)(bit(buf, 49ub)
     ^ bit(buf, 0ub)) + bit(buf, 0ub);
    temp uint:2 haN = (uint:2)(bit(buf, 49ub) ^ Sx) + c;

    fct(0) on X
    {
        count = (uint:6)48;
        init = 0t;
        busy = 1t;
    }

    always()
```

```
    {
        buf = (buf << 1ub) | X;

        if (busy)
        {
            if (count)
            {
                count--;
            }
            else
            {
                busy = 0t;
                init = 1t;
            };
        };

        if (init)
        {
            Sx = bit(buf, 0ub);
            Mx: = bit(ha0, 0ub);
            c = bit(ha0, 1ub);
            init = 0t;
        }
        else
        {
            Mx = bit(haN, 0ub);
            c = bit(haN, 1ub);
        };

    }

};


F_Unpack_Select_S_  P1;
F_Align_SS_  P2;
I_AddSub_SS < 50ub, bit > P3;
F_Norm_S_  P5;
F_Pack_  P6;
F_Sign_Logic_  P4;

uint:10 signal0 = {A, P1.A};
uint:10 signal1 = {B, P1.B};
bit signal2 = {P2.B, P3.y};
uint:10 signal3 = {P6.F, S};
ubyte signal4 = {P5.Eout, P6.E};
bit signal5 = {P5.Mout, P6.M};
bit signal6 = {P2.A, P3.x};
bit signal7 = {P1.Mb, P2.Mb};
ubyte signal8 = {P1.Ediff, P2.N};
```

```
    ubyte signal9 = {P1.Ex, P5.Ein};
    bit signal10 = {P1.Sb, P3.AddSub};
    bit signal11 = {P1.Sa, P2.Sa};
    bit signal12 = {P1.Ma, P2.Ma};
    bit signal13 = {P3.z, P4.X};
    bit signal14 = {P4.Mx, P5.Min};
    bit signal15 = {P4.Sx, P6.S};
};
```

Multiplieur

```
component F_Mul_SS(in active uint:10 A,

in passive uint:10 B,

out active uint:10 P)

{


    /* WARNING: ROUNDING AND HANDLING OF SPECIAL CASES NOT IMPLEMENTED
    */
    template <ubyte S, identifier Ser_T>
    component I_Mul_SS(
        in active Ser_T a,

     in passive Ser_T b,

     out active Ser_T p)

    {
        const int Sm1 = S - 1;

        typedef uint:S   xPar_T;
        typedef uint:Sm1 yPar_T;


        template <int S, identifier Ser_T, identifier xPar_T, identifier yPar_T>
        component SP2_Loader_ (
            in active Ser_T x,
         in passive Ser_T y,
         out active Ser_T x_ser,
         out passive xPar_T x_par,
         out active Ser_T y_ser,
         out passive yPar_T y_par)
        {
            xPar_T ldMask;

            temp xPar_T xMask = ldMask & (int:S)((bit)x ? (-1) : (0));
            temp xPar_T yMask = ldMask & (int:S)((bit)x_ser ? (-1) : (0));
```

```
    fct(0) on x
    {
        x_ser: ;
        x_par  = (xPar_T)x;
        y_ser: ;
        y_par  = (yPar_T)0;
        ldMask = (xPar_T)1;
    }

    always()
    {
        x_ser = (Ser_T)y;
        x_par = (xPar_T)((~(ldMask << 1ub) & x_par) | (xMask << 1ub));
        y_ser = (Ser_T)x;
        y_par = (yPar_T)((~ldMask & y_par) | yMask);
        ldMask <<= 1ub;
    }

};

template <int S, identifier Ser_T, identifier Par_T>
component I_Mul_SP (
    in active Ser_T x,
    in passive Par_T Y,
    out active Ser_T p)
{
    const ubyte Sp1 = (ubyte)S + 1ub;
    typedef uint:S ppadd_t;
    typedef uint:Sp1 reg_t;

    bit ev;
    reg_t R;

    temp ppadd_t PP = (ppadd_t)(x ? ((int:S)-1) : ((int:S)0)) & (ppadd_t)Y;


 fct(0) on x
 {
        //R = (reg_t)PP;
        ev = 1t;
 }

   always()
   {
        R = (reg_t)bits(R, (ubyte)S, 1ub) + PP;
        p = (Ser_T)bit(R, 0ub);
        if (ev) {
            p: ;
        };
```

```
            ev = 0t;
        }

    };

    template <int S, identifier Ser_T>
    component I_Add_SS (
        in active Ser_T x,
        in passive Ser_T y,
        out active Ser_T z)
    {
        typedef uint:2 ui2_t;
        bit c;
        temp ui2_t ha = (ui2_t)(bit)x + (bit)y;
        temp ui2_t fa = ha + c;

     fct(0) on x
     {
            z: = (Ser_T)bit(ha, 0ub);
            c = bit(ha, 1ub);
     }

        always()
        {
            z = (Ser_T)bit(fa, 0ub);
            c = bit(fa, 1ub);
        }

    };

    SP2_Loader_ < S, Ser_T, xPar_T, yPar_T > P1;
    I_Mul_SP < S, Ser_T, xPar_T > P2;
    I_Mul_SP < Sm1, Ser_T, yPar_T > P3;
    I_Add_SS < S, Ser_T > P4;

    bit signal0 = {a, P1.x};
    bit signal1 = {b, P1.y};
    bit signal2 = {P4.z, p};
    bit signal3 = {P1.x_ser, P2.x};
    xPar_T signal4 = {P1.x_par, P2.Y};
    bit signal5 = {P1.y_ser, P3.x};
    yPar_T signal6 = {P1.y_par, P3.Y};
    bit signal7 = {P2.p, P4.x};
    bit signal8 = {P3.p, P4.y};
};


component F_Unpack_Combine_S_ (
    in active uint:10 A,
```

```
 in passive uint:10 B,
    out active bit Sgn,
 out active bit Ma,
 out passive bit Mb,
 out passive ubyte Esum)
{
    uint:5 cnt;

    fct(0) on A
    {
        cnt = (uint:5)23;
        Sgn = bit(A, 9ub) ^ bit(B, 9ub);
        Esum = (ubyte)((uint:9)bits(A, 7ub, 0ub)
        + (uint:9)bits(B, 7ub, 0ub) + 1t) ^ 128ub;
        Ma: ;
    }

    always()
    {
        if (cnt > 0)
        {
            Ma = bit(A, 8ub);
            Mb = bit(B, 8ub);
            cnt--;
        }
        else
        {
            Ma = 0t;
            Mb = 0t;
        };
    }

};


component F_Norm_Mul_S_ (
    in active bit Mx,
 in passive ubyte Ex,
 out active bit Mz,
    out passive ubyte Ez)
{
    bit d_ev;
    bit enabled;
    uint:26 buf = 0;
    uint:6 cnt;

    fct(0) on Mx
    {
        enabled = 1t;
        cnt = (uint:6)46;
```

```
    }

    always()
    {
        buf = (buf << 1ub) | Mx;
        Mz = bit(buf, 25ub);
        if (cnt > 0)
        {
            cnt--;
        }
        else
        {
            if (enabled)
            {
                enabled = 0t;
                if (Mx)
                {
                    d_ev = 1t;
                }
                else
                {
                    Ez = Ex;
                    Mz: ;
                };
            };
        };
        if (d_ev)
        {
            d_ev = 0t;
            Ez = Ex + 1;
            Mz: ;
        };
    }

};


component F_Pack_Mul_ (
    in passive bit S,
 in active bit M,
 in passive ubyte E,
 out active uint:10 F)
{
    uint:2 D_ev;

    fct(0) on M
    {
        D_ev = (uint:2)1;
    }
```

```
        always()
        {
            setbits(F, 7ub, 0ub, (uint:33)E);
            setbit(F, 8ub, M);
            setbit(F, 9ub, S);
            if (bit(D_ev, 1ub))
            {
                F: ;
            };
            D_ev <<= 1ub;
        }
    };


    I_Mul_SS < 24ub, bit > P2;
    F_Unpack_Combine_S_  P1;
    F_Norm_Mul_S_  P3;
    F_Pack_Mul_  P4;

    uint:10 signal0 = {A, P1.A};
    uint:10 signal1 = {B, P1.B};
    bit signal2 = {P1.Ma, P2.a};
    bit signal3 = {P1.Mb, P2.b};
    bit signal4 = {P2.p, P3.Mx};
    ubyte signal5 = {P1.Esum, P3.Ex};
    bit signal6 = {P3.Mz, P4.M};
    ubyte signal7 = {P3.Ez, P4.E};
    bit signal8 = {P1.Sgn, P4.S};
    uint:10 signal9 = {P, P4.F};
};
```

Diviseur

```
/* WARNING: ROUNDING AND HANDLING OF SPECIAL CASES NOT IMPLEMENTED
*/
component F_Div_SS(out active uint:10 P,

in active uint:10 A,

in passive uint:10 B)

{


    component F_Unpack_Combine_S_ (
        in active uint:10 A,
     in passive uint:10 B,
        out active bit Sgn,
     out active bit Ma,
     out passive bit Mb,
```

```
 out passive ubyte Esum)
{
    bit de;
    bit da;
    bit db;
    uint:5 cnt;

    fct(0) on A
    {
        Ma = 0t;
        Mb = 0t;
        cnt = (uint:5)23;
        Sgn = bit(A, 9ub) ^ bit(B, 9ub);
        Esum = (ubyte)((uint:9)bits(A, 7ub, 0ub)
         + ((uint:9)bits(B, 7ub, 0ub) ^ (uint:9)-1)) ^ 128ub;
        de = 1t;
    }

    always()
    {
        da = bit(A, 8ub);
        db = bit(B, 8ub);
        if (cnt > 0)
        {
            Ma = da;
            Mb = db;
            cnt--;
        }
        else
        {
            Ma = 0t;
            Mb = 0t;
        };
        if (de)
        {
            Ma: ;
            de = 0t;
        };
    }

};


component F_Norm_Div_S_ (
    in active bit Mx,
 in passive ubyte Ex,
 out active bit Mz,
    out passive ubyte Ez)
{
    bit d_ev = 0t;
```

```
    uint:24 rcnt = 0;

    fct(0) on Mx
    {
        rcnt = (rcnt << 1ub) | 1t;
    }

    always()
    {
        Mz = Mx;

        if (bit(rcnt, 21ub))
        {
            if (Mx)
            {
                Ez = Ex - 1ub;
                Mz: ;
            }
            else
            {
                d_ev = 1t;
            };
        };

        if (d_ev)
        {
            Mz: ;
            Ez = Ex;
            d_ev = 0t;
        };

        rcnt <<= 1ub;
    }

};


component F_Pack_Div_ (
    in passive bit S,
 in active bit M,
 in passive ubyte E,
 out active uint:10 F)
{
    fct(0) on M
    {
        F:;
    }

    always()
    {
```

```
        setbits(F, 7ub, 0ub, (uint:33)E);
        setbit(F, 8ub, M);
        setbit(F, 9ub, S);
    }
};


template <int S, identifier Ser_T>
component I_Div_SS (
    in active Ser_T x,
 in passive Ser_T y,
 out active Ser_T q)
{
    const ubyte Sm3 = (ubyte)(S - 3);
    const ubyte Sm2 = (ubyte)(S - 2);
    const ubyte Sm1 = (ubyte)(S - 1);
    const ubyte Sp1 = (ubyte)(S + 1);
    const ubyte Sp2 = (ubyte)(S + 2);

    bit ld = 0t;
    bit mb = 0t;
    bit dx = 0t;
    bit dy = 0t;
    uint:Sp1 Rx = 0;
    uint:Sp1 Ry = 0;
    uint:Sp1 R = 0;
    uint:Sp2 rcnt = 0;

    temp uint:Sp1 Rshl = (uint:Sp1)((R << 1ub) | bit(Rx, (ubyte)S));
    temp int:Sp2 Diff = (int:Sp2)(uint:Sp2)Rshl - Ry;
    temp bit q_tmp = mb | ~bit(Diff, (ubyte)Sp1);

    fct (0) on x
    {
        ld = 0t;
        q = 0t;
        Rx = (uint:Sp1)0; //(Rx << 1ub) | 0t;
        Ry = (uint:Sp1)0; //(Ry << 1ub) | 0t;
        R = (uint:Sp1)0;
        rcnt = (rcnt << 1ub) | 1t;
    }

    always ()
    {
        dx = x;
        dy = y;
        mb = bit(R, (ubyte)S);
        ld |= bit(rcnt, (ubyte)Sm1);
        R = (uint:Sp1)((q_tmp & ld) ? (Diff) : (Rshl));
        if (!ld)
```

```
            {

                Rx = (Rx << 1ub) | dx;
                Ry = (Ry << 1ub) | dy;
            }
            else
            {
                Rx = (Rx << 1ub) | 0t;
                Ry = Ry;
            };
            q = q_tmp;
            if (bit(rcnt, (ubyte)Sp1))
            {
                q: ;
            };
            rcnt <<= 1ub;
        }

    };



    F_Unpack_Combine_S_  P1;
    F_Norm_Div_S_  P3;
    F_Pack_Div_  P4;
    I_Div_SS < 24, bit > P2;

    uint:10 Link13 = {A, P1.A};
    uint:10 Link1 = {B, P1.B};
    bit Link2 = {P1.Ma, P2.x};
    bit Link3 = {P1.Mb, P2.y};
    bit Link4 = {P2.q, P3.Mx};
    ubyte Link5 = {P1.Esum, P3.Ex};
    bit Link6 = {P3.Mz, P4.M};
    ubyte Link7 = {P3.Ez, P4.E};
    bit Link8 = {P1.Sgn, P4.S};
    uint:10 Link9 = {P, P4.F};
};
```

Bibliographie

[1] Altera. *Apex 20K Programmable Logic Device Family Data Sheet, ver. 2.05*. Altera Corp., San Jose, CA, Nov. 1999.

[2] Altera. *Cyclone II Device Handbook, Volume 1*. Altera Corp., San Jose, CA, June 2004.

[3] Altera. Implementing Multipliers in FPGA Devices. Application Note, July 2004. Altera Corp., San Jose, CA.

[4] Altera. *Stratix Device Handbook, Volume 1*. Altera Corp., San Jose, CA, Sept. 2004.

[5] Altera. Stratix GX FPGA Family. Data Sheet, Feb. 2004. Altera Corp., San Jose, CA.

[6] Altera. *Stratix II Device Handbook, Volume 1*. Altera Corp., San Jose, CA, July 2004.

[7] D. N. Amanor, C. Paar, J. Pelzl, V. Bunimov, and M. Schimmler. Efficient hardware architectures for modular multiplication on FPGAs. In *Proc. of the 15th International Conference on Field Programmable Logic and Application, FPL 2005*, Lecture Notes in Computer Science, pages 539–542, Leuven, Belgium, 2005. Springer.

[8] R. Andraka. A survey of CORDIC algorithms for FPGA based computers. In *Proc. 6th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 98)*, pages 191–200, Los Alamitos, CA, Feb. 1998. IEEE Computer Society Press.

[9] M. A. Ashour and H. I. Saleh. An FPGA implementation guide for some different types of serial-parallel multiplier structures. *Microelectronics Journal*, 31(3) :161–168, Mar. 2000.

[10] A. Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, EC-10 :389–400, September 1961. Reprinted in [84].

[11] J.-C. Bajard, J. Duprat, S. Kla, and J.-M. Muller. Some operators for on-line radix 2 computations. *Journal of Parallel and Distributed Computing*, 22(2) :336–345, Aug. 1994.

[12] A. E. Bashagha. Pipelined area-efficient digit serial divider. *Signal Processing*, 83(9) :2011–2020, 2003.

[13] A. E. Bashagha and M. Ibrahim. A new digit-serial divider architecture. *International Journal of Electronics*, 75(1) :133–140, 1993.

[14] R. Beguenane, J.-L. Beuchat, J.-M. Muller, and S. Simard. Modular Multiplication of Large Integers on FPGA. In *Proc. 39th Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, Nov. 2005. IEEE Press.

[15] R. Beguenane and S. Simard. Towards the design of area-minimal operators for massively-parallel computing on FPGAs. In *Proc. 1st International Computer Systems and Information Technology Conference (ICSIT'05), July 19-22, 2005*, pages 684–687, Algiers, Algeria, July 2005.

[16] 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 (IEEE-NEWCAS 2006)*, Gatineau, Canada, June 2006. IEEE.

[17] S. J. Bellis, W. P. Marnane, and P. Fish. Parametric Spectral Estimation on a Single FPGA. In *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP-99*, volume 4, Mar. 1999.

[18] P. Bertin, D. Roncin, and J. Vuillemin. Programmable Active Memories : A Performance Assesment. Technical Report 24, DEC-Paris Research Laboratory, March 1993.

[19] J.-L. Beuchat. *Etude et conception d'opérateurs arithmétiques optimisés pour circuits programmables.* PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 2001.

[20] J.-L. Beuchat. Some modular adders and multipliers for field programmable gate arrays. In *17th International Parallel and Distributed Processing Symposium (IPDPS-2003)*, pages 190–197, Los Alamitos, CA, Apr. 22–26 2003. IEEE Computer Society.

[21] J.-L. Beuchat and J.-M. Muller. Modulo M multiplication-addition : Algorithms and FPGA implementation. *Electronics Letters*, 40(11) :654–655, 2004.

[22] J.-L. Beuchat and A. Tisserand. Évaluation polynomiale en-ligne de fonctions élémentaires sur FPGA. Technical Report 4557, Institut national de recherche en informatique et en automatique (INRIA), Sept. 2002.

[23] J.-L. Beuchat and A. Tisserand. Opérateur en-ligne sur FPGA pour l'implantation de quelques fonctions élémentaires. In *Actes de la conférence Sympa'8 - Symposium en architectures de nouvelles machines*, pages 267–274, Hamamet, Tunisie, Avril 2002.

[24] J.-L. Beuchat and A. Tisserand. Small multiplier-based multiplication and division operators for Virtex-II devices. *Lecture Notes in Computer Science*, 2438 :513–522, 2002.

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

[26] J.-L. Beuchat and A. Tisserand. Opérateurs arithmétiques sur circuits FPGA. In J.-C. Bajard and J.-M. Muller, editors, *Calcul et arithmétique des ordinateurs*, Traité IC2, pages 109–152. Lavoisier, 2004.

[27] G. Cappuccino, G. Cocorullo, P. Corsonello, and S. Perri. Educational design of high-performance arithmetic circuits on FPGA. *IEEE Transactions on Education*, 42(4), Nov. 1999.

[28] K. D. Chapman. Fast integer multiplier fits in FPGAs. *Electronic Design News*, 39(10) :79–80, May 12, 1993. Design Idea Winner, EDN.

[29] K. D. Chapman. Fast integer multipliers using FPGAs. *Xilinx Xcell Journal*, 14(Q3) :28–31, 1994.

[30] K. D. Chapman. Constant coefficient multipliers for the XC4000E. Application Note XAPP 054, Dec. 1996. Xilinx, inc., San Jose, CA.

[31] M. Daumas, J.-M. Muller, and J. Vuillemin. Implementing on-line arithmetic on PAM. In R. W. Hartenstein and M. Servít, editors, *4th International Workshop on Field-Programmable Logic and Applications (FPL'94)*, volume 849, pages 196–207, Prague, Czech Republic, Sept. 1994. Springer.

[32] P. B. Denyer and D. Renshaw. *VLSI Signal Processing; A Bit-Serial Approach.* Addison-Wesley Longman Publishing Co., inc., Boston, MA, USA, 1985.

[33] J. Detrey. Bibliothèque d'opérateurs paramétrables pour l'arithmétique "réelle" sur FPGA. DEA, École Normale Supérieure de Lyon, July 2003.

[34] F. Dittmann, B. Kleinjohann, and A. Rettberg. Efficient bit-serial constant multiplication for FPGAs. In *Proceedings of the 11th NASA Symposium VLSI Design*, May 2003.

[35] W. P. du Plessis. Optimal MAC structures in an FPGA. In *Proceedings of the 6th IEEE Africon 2002 Conference*, volume 1, pages 333–336, Oct. 2002.

[36] M. D. Ercegovac. On-line arithmetic : An overview. In *SPIE, Real Time Signal Processing VII*, pages 86–93. SPIE-The International Society for Optical Engeneering, Bellingham, Washington, 1984.

[37] M. D. Ercegovac and T. Lang. On-line arithmetic : a design methodology and applications in digital signal processing. In *VLSI Signal Processing*, volume III, pages 252–263, 1988. Reprinted in [85].

[38] M. D. Ercegovac and T. Lang. Radix-4 square root without initial PLA. In *Proceedings of the 9th IEEE Symposium on Computer Arithmetic*, pages 162–168, Santa Monica, USA, 1989. IEEE Computer Society Press, Los Alamitos, CA.

[39] M. D. Ercegovac and T. Lang. Radix-4 Square Root Without Initial PLA. *IEEE Transactions on Computers*, C-39(9) :1016–1024, August 1990.

[40] M. D. Ercegovac and T. Lang. *Digital Arithmetic.* Morgan Kaufmann Publishers, 2004.

[41] R. Galli. Design and Evaluation of On-line Arithmetic Modules and Networks for Signal Processing Applications on FPGAs. Master's Thesis, Oregon State University, June 2001.

[42] R. Galli and A. F. Tenca. Design and evaluation of online arithmetic for signal processing applications on FPGAs. In F. T. Luk, editor, *Proc. of SPIE, Advanced Signal Processing Algorithms, Architectures, and Implementations XI*, volume 4474, pages 134–144. Society of Photo-Optical Instrumentation Engineers, Aug. 2001.

[43] R. S. Grover, W. Shang, and Q. Li. A faster distributed arithmetic architecture for FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA 2002)*, pages 31–39, Monterey, CA, Feb. 24–26 2002. ACM.

[44] S. D. Haynes and P. Y. Cheung. A reconfigurable multiplier array for video image processing tasks, suitable for embedding in an FPGA structure. In *Proceedings of the 6th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2003)*, pages 226–236. IEEE Computer Society Press, Apr. 1998.

[45] T. Isshiki and W. W.-M. Dai. High-Level Bit-Serial Datapath Synthesis for Multi-FPGA Systems. In *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pages 167–173, New York, NY, USA, Feb. 1995. ACM Press.

[46] E. Jamro and K. Wiatr. Convolution operation implemented in FPGA structures for real time image processing. In *Proc. of the IEEE Int. Symposium on Image Processing and Analysis*, June 2001.

[47] E. Jamro and K. Wiatr. FPGA implementation of addition as a part of the convolution. In *IEEE Euromicro Symposium on Digital Systems Design (DSD'01)*, pages 458–465. IEEE Computer Society, Sept. 2001.

[48] E. Jamro and K. Wiatr. Constant coefficient convolution implemented in FPGAs. In *Proceedings of Euromicro Symposium on Digital System Design (DSD'02)*, pages 291–298, Sept. 2002.

[49] V. Kantabutra, P. Corsonello, S. Perri, and M. A. Iachino. Efficient, practical adders for FPGAs. *Circuit Cellar Magazine*, 148 :28–38, Nov. 2002.

[50] T. Kean, B. New, and R. Slous. A fast constant coefficient multiplier for the xc6200. In *6th International Conference on Field Programmable Logic and Application, FPL 1996*, pages 230–236, Darmstadt, Germany, Sept. 1996.

[51] S. Kla. *Calcul parallèle et en ligne des fonctions arithmétiques*. PhD thesis, École Normale Supérieure de Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France, Feb. 1993.

[52] S. K. Knapp. Constant-coefficient multipliers save FPGA space, time. *Personal Engineering and Instrumentation News*, pages 45–48, July 1998.

[53] F. Kobayashi, T. Tsujino, and H. Saitoh. Efficient FPGA Implementation of Multiplier-Adder – Quotient-Remainder Approach. In *32nd Asilomar Conference on Signals, Systems, and Computers*, Asilomar, California, November 1998. IEEE Press.

[54] P. Larsson-Edefors and W. P. Marnane. Most-significant-bit-first serial/parallel multipliers. *IEE Proc. on Circuits, Devices and Systems*, 145(4) :278–284, Aug. 1998.

[55] D. Lau, A. Schneider, M. D. Ercegovac, and J. Villasenor. A FPGA-based library for on-line signal processing. *Journal of VLSI Signal Processing*, 28 :129–143, 1999.

[56] B. R. Lee and N. Burgess. Improved small multiplier based multiplication, squaring and division. In *Proceedings of the 11th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2003)*, pages 91–97. IEEE Computer Society Press, Apr. 2003.

[57] J. W. Lewis. Coding a 40x40 pipelined multiplier. In *5th International Conference on Military and Aerospace Applications of Programmable Devices and Technologies (MAPLD 2002)*. NASA Office of Logic Design, Sept. 2002.

[58] H. Lin and H. Sips. On-Line CORDIC Algorithms. *IEEE Transactions on Computers*, C-39(9) :1038–1052, August 1990.

[59] M. E. Louie and M. D. Ercegovac. A digit-recurrence square root implementation for field programmable gate arrays. In D. A. Buell and K. L. Pocek, editors, *Proceedings of the 1st IEEE Workshop on FPGAs for Custom Computing Machines (FCCM 1993)*, pages 178–183. IEEE Computer Society Press, Apr. 1993.

[60] M. E. Louie and M. D. Ercegovac. On digit-recurrence division implementations for field programmable gate arrays. In E. Swartzlander, M. J. Irwin, and G. Jullien, editors, *Proc. 11th IEEE Symposium on Computer Arithmetic*, pages 202–209. IEEE Computer Society Press, 1993.

[61] S.-L. Lu and J. Kenney. Design of most-significant-bit-first serial multiplier. *Electronic Letters*, 31(14) :1133–1135, July 1995.

[62] Z. Luo and M. Martonosi. Using delayed addition techniques to accelerate integer and floating-point calculations in configurable hardware. In J. Schewel, editor, *Configurable Computing : Technology and Applications, Proc. SPIE 3526*, pages 202–211, Bellingham, WA, November 1998. SPIE – The International Society for Optical Engineering.

[63] W. P. Marnane. Optimized bit serial modular multiplier for implementation on field programmable gate arrays. *IEE Electronic Letters*, 34(8) :738–739, Apr. 1998.

[64] W. P. Marnane, S. J. Bellis, and P. Larsson-Edefors. Bit-serial interleaved high speed division. *Electronics Letters*, 33(13) :1124–1125, Sept. 1997.

[65] R. Matousek, M. Tichý, Z. Pohl, J. Kadlec, C. Softley, and N. Coleman. Logarithmic number system and floating-point arithmetics on FPGA. In *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, pages 627–636. Springer-Verlag, 2002.

[66] R. D. McIlhenny. *Complex Number On-line Arithmetic for Reconfigurable Hardware : Algorithms, Implementations, and Applications*. PhD thesis, University of California, Los Angeles, 2002.

[67] R. D. McIlhenny and M. D. Ercegovac. On-line algorithms for complex number arithmetic. In *32th Asilomar Conference on Signals, Systems, and Computers*, pages 172–176, Pacific Grove, California, Oct. 1998. IEEE Press.

[68] L. Mintzer. FIR filters with the Xilinx FPGA. In *Proc. of the First International ACM/SIGDA Workshop on Field Programmable Gate Arrays (FPGA'92)*, University of California, Berkeley, CA, Feb. 1992. ACM.

[69] L. Mintzer. Soft Radios and Modems on FPGAs. *Communication Systems Design Magazine*, 6(2), Feb. 2000.

[70] T. Miomo, K. Yasuoka, and M. Kanazawa. The fastest multiplier on FPGAs with redundant binary representation. In R. W. Hartenstein and H. Grünbacher, editors, *10th International Workshop on Field Programmable Logic and Applications, FPL 2000*, volume 1896 of *Lecture Notes in Computer Science*, pages 515–524, Aug. 2000.

[71] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhäuser, Boston, 1997.

[72] Novakod. *psC Version 0.4 Programmer's Guide - Programming real-time and reconfigurable systems.* Chicoutimi (QC), Canada, Sept. 2004.

[73] V. G. Oklobdzija and M. D. Ercegovac. An on-line square root algorithm. *IEEE Transactions on Computers*, C-31(1) :70–75, Jan. 1982.

[74] P. Belanović. Library of parameterized hardware modules for floating-point arithmetic with an example application. Master's thesis, Northeastern University, Boston, USA, May 2002.

[75] P. Belanović and M. Leeser. A library of parameterized floating point modules and their use. In *12th International Conference on Field Programmable Logic and Application, FPL 2002*, pages 657–666, Montpellier, France, September 2002.

[76] A. Panato, S. Silva, F. Wagner, M. Johann, R. Reis, and S. Bampi. Design of very deep pipelined multipliers for FPGAs. In *Proceedings of the Design, Automation*

and Test in Europe Conference and Exhibition Designers' Forum (DATE'04). IEEE Computer Society, Feb. 2004.

[77] B. Parhami. *Computer Arithmetic, Algorithms and Hardware Designs*. Oxford University Press, New-York, 2000.

[78] S. Perri, M. A. Iachino, and P. Corsonello. Speed-efficient wide adders for Virtex FPGAs. In *9th International Conference on Electronics, Circuits and Systems (ICECS 2002)*, volume 2, pages 599–602. IEEE Circuits and Systems Society, Sept. 2002.

[79] R. J. Petersen and B. L. Hutchings. An assessment of the suitability of FPGA-based systems for use in DSPs. In W. Moore and W. Luk, editors, *5th International Conference on Field Programmable Logic and Application, FPL 1995*, volume 975 of *Lecture Notes in Computer Science*, pages 293–302, Berlin, Germany, Aug. 1995.

[80] K. Piromsopa, C. Aporntewan, and P. Chongstitvatana. An FPGA implementation of a fixed-point square root operation. In *International Symposium on Communications and Information Technologies, ISCIT 2001*, Nov. 2001.

[81] K. Rajagopalan and P. Sutton. A flexible multiplication unit for an FPGA logic block. In *Proceedings of 2001 IEEE International Symposium on Circuits and Systems vol IV (ISCAS 2001)*, pages 546–549, May 2001.

[82] A. Saha and R. Krishnamurthy. Design and FPGA implementation of efficient integer arithmetic algorithms. In *Proc. of IEEE Southeastcon '93*. IEEE, Apr. 1993.

[83] G. Sutter, G. Bioul, and J.-P. Deschamps. Comparative Study of SRT-Dividers in FPGA. In J. Becker, M. Platzner, and S. Vernalde, editors, *14th International Conference on Field Programmable Logic and Application, FPL 2004*, volume 3203 of *Lecture Notes in Computer Science*, pages 209–220, Leuven, Belgium, Sept. 2004. Springer.

[84] E. E. Swartzlander, editor. *Computer Arithmetic*, volume 1. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[85] E. E. Swartzlander, editor. *Computer Arithmetic*, volume 2. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[86] N. Takagi, T. Asada, and S. Yajima. Redundant CORDIC Methods with a Constant Scale Factor for Sine and Cosine Computation. *IEEE Transactions on Computers*, 40(9) :989–995, Sept. 1991.

[87] P. T. P. Tang. Table lookup algorithms for elementary functions and their error analysis. In P. Kornerup and D. W. Matula, editors, *Proc. 10th IEEE Symposium on Computer Arithmetic*, pages 232–236. IEEE Computer Society, June 1991.

[88] A. Tenca. *Variable Long-Precision Arithmetic (VLPA) for Reconfigurable Coprocessor Architectures*. PhD thesis, University of California, Los Angeles, 1998.

[89] A. F. Tenca and S. U. Hussaini. A Design of Radix-2 On-line Division Using LSA Organization. In *Proc. 15th IEEE Symposium on Computer Arithmetic*, pages 266–273. IEEE Computer Society, June 2001.

[90] A. Tisserand. *Adéquation arithmétique architecture — problèmes et étude de cas*. PhD thesis, École Normale Supérieure de Lyon, France, 1997.

[91] K. Trivedi and M. D. Ercegovac. On-line algorithms for division and multiplication. *IEEE Transactions on Computers*, C-26(7) :681–687, July 1977. Reprinted in [85].

[92] P. K.-G. Tu and M. D. Ercegovac. Design of On-Line Division Unit. In M. D. Ercegovac and E. E. Swartzlander, editors, *Proc. 9th IEEE Symposium on Computer Arithmetic*, pages 42–49, Santa Monica, USA, September 1989. IEEE Computer Society, IEEE Computer Society Press, Los Alamitos, CA.

[93] K. Underwood. FPGAs vs. CPUs : Trends in Peak Floating-Point Performance. In *Proc. 2004 ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays (FPGA 2004)*, Feb. 2004.

[94] J. Valls and E. Boemo. Efficient FPGA-implementation of two's complement digit-serial/parallel multipliers. *IEEE Transactions on Circuits and Systems - II*, 50(6) :317–322, June 2003.

[95] J. Valls, M. M. Peiro, T. Sansaloni, and E. Boemo. A study about FPGA-based digital filters. In *1998 IEEE Workshop on VLSI Signal Processing : Design and Implementation (SiPS'98)*, pages 192–201, Boston, MA, 1998.

[96] J. Valls, T. Sansaloni, M. M. Peiro, and E. Boemo. Fast FPGA-based pipelined digit-serial/parallel multipliers. In *1999 IEEE ISCAS International Symposium on Circuits and Systems*, volume 1, pages 482–485, 1999.

[97] C. Visavakul, P. Y. Cheung, and W. Luk. A digit-serial structure for reconfigurable multipliers. In G. J. Brebner and R. Woods, editors, *11th International Conference on Field Programmable Logic and Application, FPL 2001*, volume 2147 of *Lecture Notes in Computer Science*, pages 565–573, Aug. 2001.

[98] K. Wiatr and E. Jamro. Implementation of multipliers in FPGA structures. In *2nd International Symposium on Quality of Electronic Design (ISQED 2001), 26-28 March 2001, San Jose, CA, USA*, pages 415–421. IEEE Computer Society, 2001.

[99] M. Wojko. Pipelined multipliers and FPGA architectures. In P. Lysaght, J. Irvine, and R. W. Hartenstein, editors, *9th International Workshop on Field Programmable Logic and Applications, FPL'99*, volume 1673 of *Lecture Notes in Computer Science*, pages 347–352, Aug. 1999.

[100] M. Wojko and H. ElGindy. Self configuring binary multipliers for LUT addressable FPGAs. In Hawick and James, editors, *Proc. of the 5th Australasian Conference on Parallel and Real-Time Systems (PART'98)*, pages 201–212. Springer, 1998.

[101] Xilinx. *The Programmable Gate Array Data Book.* Xilinx, inc., San Jose, CA, 1999.

[102] Xilinx. Using Embedded Multipliers in Spartan-3 FPGAs. Application Note, May 2003. Xilinx, inc., San Jose, CA.

[103] Xilinx. Using Look-Up Tables as Shift Registers (SRL16) in Spartan-3 Devices. Application Note XAPP465 (v1.0), Apr. 2003. Xilinx, inc., San Jose, CA.

[104] Xilinx. Spartan-3 FPGA Family : Complete Data Sheet, Aug. 2004. Xilinx, inc., San Jose, CA.

[105] Xilinx. Virtex-4 Family Overview. Advance Product Specification, Sept. 2004. Xilinx, inc., San Jose, CA.

[106] Xilinx. Virtex-II Platform FPGAs : Complete Data Sheet. Product Specification, June 2004. Xilinx, inc., San Jose, CA.

[107] Xilinx. Virtex-II Pro and Virtex-II Pro X Platform FPGAs : Complete Data Sheet. Product Specification, June 2004. Xilinx, inc., San Jose, CA.

[108] S. Xing and W. Yu. FPGA adders : Performance evaluation and optimal design. *IEEE Design & Test of Computers*, 47(5) :587–602, January-March 1998.

[109] R. Zimmermann. VHDL Library of Arithmetic Units. www.iis.ee.ethz.ch/~zimmi, 1998.