POLYPUBLIE

Une méthode de dérivation de modèles de processeurs embarqués dédiés à une application

Olivier Hébert

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

Date	2001
Type	Mémoire ou thèse / Dissertation or Thesis Mémoire de maîtrise/Master's thesis
URL de PolyPublie PolyPublie URL	https://publications.polymtl.ca/6957/
Conditions d'utilisation Terms of use	© Tous droits réservés / All rights reserved
Référence Citation	Hébert, O. (2001). Une méthode de dérivation de modèles de processeurs embarqués dédiés à une application [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/6957/

Détails de la thèse
Thesis details

Directeurs de recherche Advisors	Yvon Savaria
Programme Program	Non spécifié

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal
https://publications.polymtl.ca

UNIVERSITÉ DE MONTRÉAL

UNE MÉTHODE DE DÉRIVATION DE MODÈLES DE PROCESSEURS EMBARQUÉS DÉDIÉS À UNE APPLICATION

OLIVIER HÉBERT
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE ET DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
MARS 2001

© Olivier Hébert, 2001.

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

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

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-65580-6

Canada

Identification des membres du jury

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

UNE MÉTHODE DE DÉRIVATION DE MODÈLES DE PROCESSEURS EMBARQUÉS DÉDIÉS À UNE APPLICATION

présenté par: HÉBERT Olivier

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment acceptée par le jury d'examen constitué de:

M. BOIS Guy, Ph.D., président

M. SAVARIA Yvon, Ph.D., membre et directeur de recherche

M. Aboulhamid El Mostapha, Ph.D., membre

Remerciements

J'aimerais tout d'abord remercier les organismes subventionnaires qui m'ont soutenu financièrement au cours de mes études de maîtrise et de la réalisation du projet décrit dans le présent mémoire. Ceux-ci sont le Conseil de recherche en sciences naturelles et en génie du Canada (CRSNG) ainsi que la compagnie MiroTech Microsystems.

J'aimerais remercier les gens qui m'ont guidé tout au long de ce projet de recherche, soient mon directeur de recherche, le professeur Yvon Savaria de l'École Polytechnique de Montréal, ainsi que M. Ivan C. Kraljic, ingénieur chez MiroTech Microsystems. Également, mes professeurs qui m'ont éclairé sous différents sujets, soient les professeurs Guy Bois de l'École Polytechnique de Montréal et El Mostapha Aboulhamid de l'Université de Montréal.

J'aimerais aussi remercier les gens avec qui j'ai travaillé ou que j'ai côtoyé tout au long de mes travaux. Ceux-ci sont M. Alexandre Fortin (avec qui j'ai directement travaillé sur le projet), M. Frédéric Doucet, Mme Geneviève Cyr, M. Jean-Marc Tremblay et M. Patrice Vado.

Finalement, j'aimerais remercier mes parents et amis qui m'ont supporté pendant toute la durée de mes études.

Résumé

Le temps de développement de systèmes embarqués étant de plus en plus court, et ceux-ci étant d'une complexité toujours grandissante, les concepteurs doivent maintenant utiliser des méthodologies nouvelles pour arriver à réaliser à temps leurs projets. Parmi les méthodes utilisées, la réutilisation de composantes disponibles sous la forme de blocs de propriété intellectuelle (*IP*) est très populaire. Toutefois, une implantation directe de composantes sans tenir compte de l'application qui est ciblée va produire des circuits qui ne seront pas optimaux. Ainsi, plusieurs parties des composantes pourraient ne pas être utilisées par l'application et être quand même implantées. On assisterait alors à un gaspillage des ressources disponibles.

Ce mémoire décrit une méthode de dérivation de processeurs dédiés permettant d'optimiser les circuits synthétisés en fonction de l'application que l'on cible. Ainsi, en introduisant une étape d'optimisation avant la synthèse finale du circuit, on peut extraire l'information contenue dans le microcode de l'application et optimiser le processeur pour l'exécution de cette application particulière. On décrit également le développement d'un modèle de processeur de traitement de signal parallèle destiné à implanter la méthode de dérivation. Celui-ci a été développé de façon à être fortement configurable et optimisable et ce, de façon automatique. Des expérimentations avec la méthode et le modèle de processeur sont également discutées. Elles permettent de prouver la validité de la méthode de dérivation de modèles de processeurs dédiés à une application.

Abstract

With the time to market constraints for the development of embedded systems getting more rigid and shorter, and those systems getting increasingly complex, the designers must now use new methodologies in order to deliver their projects on time. Among the new methods now used, the reuse of components, available in the form of intellectual property (IP) blocs is very popular. However, the direct implementation of components, without any regards to the target application will lead to circuits that are sub-optimal. That is, many parts of the components may not be used at all by the application, but are implemented nonetheless. This would lead to a waste of available physical resources.

This master thesis presents a method to derive application-specific embedded processors that optimizes the circuits synthesized according to the target application. By introducing an extra step of optimization prior to final synthesis of the circuit, we can extract information from the application microcode, and optimize the processor for the execution of this specific application. We also describe the development of a parallel digital signal processor core designed to implement the derivation method. It has been developed to be highly configurable and optimizable, in an automatic way. Experiments with the method and the processor core are also presented, which prove the validity of the method for the derivation of application-specific embedded cores.

Table des matières

IDENTIFICATION DES MEMBRES DU JURY ..... **III**

REMERCIEMENTS ..... **IV**

RÉSUMÉ ..... **V**

ABSTRACT ..... **VI**

TABLE DES MATIÈRES ..... **VII**

LISTE DES TABLEAUX ..... **XI**

LISTE DES FIGURES ..... **XII**

INTRODUCTION **1**

CHAPITRE 1 ..... **6**

REVUE DE LITTÉRATURE ET RÉALISATIONS INDUSTRIELLES **6**

1.1 PROCESSEUR XTENSA 7

1.2 ARC 11

1.3 IMPROV SYSTEMS 13

1.4 DÉRIVATION DE PROCESSEURS POUR LA TOLÉRANCE AUX FAUTES 16

1.5 DÉRIVATION DE CIRCUITS À PARTIR D'UN ÉMULATEUR 19

CHAPITRE 2 ..... **24**

LA MÉTHODE DE DÉRIVATION DE MODÈLES DE PROCESSEURS EMBARQUÉS DÉDIÉS À UNE APPLICATION **24**

2.1 INTRODUCTION 25

2.2 MOTIVATIONS 27

2.3 MÉTHODES DE DESIGN TRADITIONNELLES 29

2.4 OUTIL D'ANALYSE DU MICROCODE 32

2.4.1 Décodage des instructions 32

2.4.2 Statistiques d'utilisation 34

2.5 ÉTUDE DE CAS : LE PROCESSEUR PULSE V1 35

2.5.1 Suite d'applications de test 36

2.5.2 Résultats d'analyses de la suite de test 36

2.5.3 Optimisations manuelles initiales 39

2.6 OPTIMISATIONS DU MODÈLE 40

2.6.1 Élimination de ressources 43

2.6.2 Propagation de signaux constants 44

2.6.3 Tables de constantes locales 45

2.6.4 Recodage de champs 46

2.6.5 Autres optimisations 46

CHAPITRE 3 48

ARCHITECTURE PRODSP 48

3.1 ARCHITECTURE PULSE 49

3.1.1 Description 49

3.1.2 Réalisations 51

3.2 PRODSP 51

3.2.1 Description *52*

3.2.2 Jeu d'instructions et architecture du microcode *53*

3.2.3 Contrôleur *55*

3.2.4 Configuration et État *59*

3.2.5 Communication et Entrées/Sorties *60*

3.2.6 Élément de calcul (PE) *63*

3.2.7 Générateurs d'adresses externes *67*

3.3 MÉTHODOLOGIE ET STYLE DE CODAGE 70

3.3.1 Paramètres et modules génériques *71*

3.3.2 Types et configuration *73*

3.4 VALIDATION 79

3.5 OUTILS LOGICIELS 82

3.5.1 Simulation *82*

3.5.2 Synthèse *83*

3.5.3 Problèmes rencontrés *84*

CHAPITRE 4 ..... **85**

OPTIMISATION DU MODÈLE **85**

4.1 CONFIGURABILITÉ DU MODÈLE 86

4.1.1 Options de configuration *87*

4.2 MODIFICATIONS DU MODÈLE 88

4.2.1 Modifications du contrôleur *89*

4.2.2 Modifications du chemin de données *90*

4.3 OUTIL D'OPTIMISATION 92

4.3.1 Analyse du microcode *93*

4.3.2 Génération du modèle *93*

4.3.2.1 Constructions en langage HDL 94

4.3.2.2 Générateur de code HDL 96

4.3.2.3 Générateur de modèle ProDSP 97

4.3.3 Recodage du microcode *98*

4.4 INTERFACE-USAGER ET EXEMPLE D'UTILISATION 99

4.5 ÉTUDES DE CAS 101

4.5.1 Applications de test *101*

4.5.2 Configurations de test *102*

4.5.3 Synthèse des modèles de test *103*

4.5.4 Résultats des optimisations *104*

CONCLUSION **108**

RÉFÉRENCES **114**

Liste des tableaux

Tableau 1.1 Résultats quantitatifs de synthèse du contrôleur de processeur (Pflanz et al., 1998) 19

Tableau 1.2 Utilisation globale des ressources pour différents algorithmes (Kraljic et al., 1996) 23

Tableau 2.1 Utilisation des signaux de contrôle pour la suite d'applications de test 37

Tableau 3.1 Modes d'adressage de la mémoire de l'architecture ProDSP 77

Tableau 3.2 Modes d'adressage de la mémoire pour un exemple d'application 77

Tableau 4.1 Options de configuration du processeur ProDSP 87

Tableau 4.2 Configuration du modèle ProDSP pour les différentes applications 103

Tableau 4.3 Résultats de synthèse pour les différentes applications 105

Liste des figures

Figure I.1 Complexité des designs et productivité des concepteurs (SEMATECH, 1999) 3

Figure 1.1 Flot de conception d'un processeur Xtensa (Tensilica, 2000) 8

Figure 1.2 Flot d'information pour la conception d'un processeur avec la technologie Tensilica (Tensilica, 2000) 10

Figure 1.3 Architecture du processeur Jazz (Improv, 2000) 14

Figure 1.4 Architecture de la plate-forme de Improv (Improv, 2000) 15

Figure 1.5 Flot de design pour la synthèse spécifique à une application (Pflanz et al., 1998) 18

Figure 1.6 Architecture du processeur de flot de données (Kraljic et al., 1996) 20

Figure 1.7 Portion du processeur active pour l'opérateur d'addition (Kraljic et al., 1996) 21

Figure 1.8 Flot de design pour la dérivation à partir des résultats d'émulation (Kraljic et al.,1996) 22

Figure 2.1 Méthode traditionnelle de synthèse d'un processeur embarqué pour une implantation SOC 31

Figure 2.2 Dérivation de modèles de processeurs embarqués dédiés à une application spécifique 42

Figure 3.1 Architecture PULSE simplifiée (*Marriott et al., 1998*) 50

Figure 3.2 Diagramme bloc de l'architecture ProDSP 52

Figure 3.3 Diagramme bloc du Contrôleur ProDSP 55

Figure 3.4 Diagramme bloc du Séquenceur 56

Figure 3.5 Diagramme bloc des canaux de communication 61

Figure 3.6 Pipeline du chemin de données de ProDSP 64

Figure 3.7 Exemple de génération d'adresses 68

Figure 3.8 Diagramme bloc du générateur d'adresses linéaires à *n* dimensions 69

Figure 3.9 Passage de paramètres par clauses génériques 72

Figure 3.10 Génération et propagation des types 78

Figure 3.11 Banc de test du modèle ProDSP 81

Figure 4.1 Exemple de code sans modification pour granularité fine 91

Figure 4.2 Exemple de code avec modifications pour granularité fine 92

Figure 4.3 Interface-usager de l'optimiseur 100

Introduction

Au cours des dernières années, la technologie de fabrication de dispositifs électroniques a connu une évolution fulgurante. De ce fait, on a vu apparaître des systèmes électroniques d'une très grande complexité dans les domaines traditionnels d'utilisation de tels systèmes, tels l'informatique et les télécommunications. On a également assisté à une prolifération de systèmes électroniques embarqués, dans un grand nombre d'appareils fort diversifiés, allant de la voiture au grille-pain.

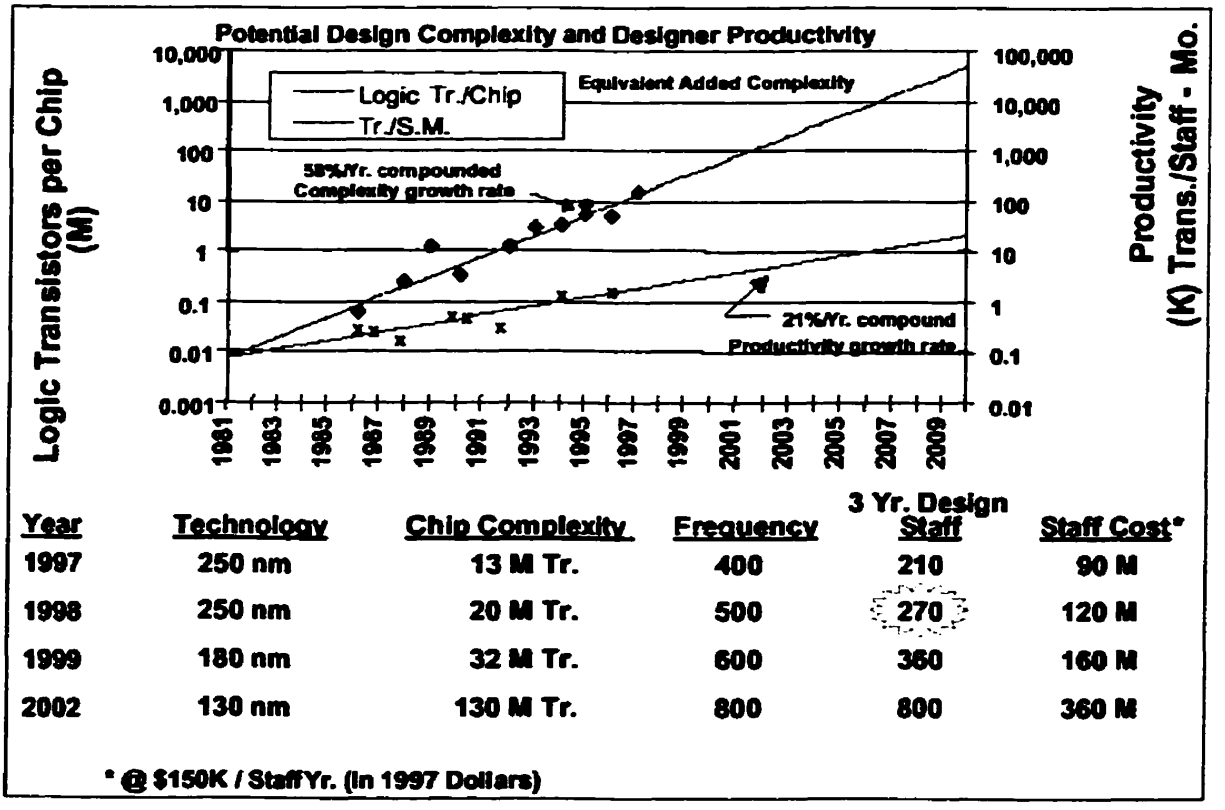
D'autres facteurs ont contribué à la popularité rapidement grandissante de ces systèmes embarqués. Tout d'abord, la démocratisation des réseaux de communication, en particulier Internet, ont favorisé une adoption rapide d'items reliés aux technologies de l'information. Par ailleurs, on assiste présentement à un fort développement de l'industrie des communications sans fil, notamment par les réseaux cellulaires numériques (SCP : Système de communication personnel).

La tendance actuelle est à la convergence de toutes ces technologies, notamment par l'intégration de l'accès réseau aux divers appareils. On a d'ailleurs commencé à voir apparaître des téléphones cellulaires permettant d'accéder a Internet, ainsi que des appareils domestiques se connectant sur le réseau pour effectuer leurs fonctions (*internet applicances*.)

Toute cette intégration de service requiert des systèmes électroniques particulièrement puissants et complexes. Toutefois, comme ce sont des systèmes embarqués, ils sont soumis à une multitude de contraintes immuables. Celles-ci sont notamment la performance du système, la dissipation de puissance, le coût de production et finalement l'intégration des composantes qui forment le système.

Avec la disponibilité de plate-formes cibles disposant de plusieurs millions de portes logiques équivalentes, tant sous la forme de logique programmable (FPGA : *Field Programmable Gate Arrays*) que de circuits intégrés à application spécifique (ASIC : Application Specific Integrated Circuit), on a vu apparaître une nouvelle façon de réaliser des systèmes embarqués : des systèmes sur une puce (SOC : *System-on-a-chip*). Toutefois, bien que la technologie soit capable de supporter des systèmes de plus en plus complexes, le temps et les ressources nécessaires pour réaliser de tels systèmes embarqués deviennent de plus en plus importants également. Ainsi, la productivité des concepteurs avec les méthodologies et les outils actuels ne suit pas la complexité possible des circuits intégrés. Ainsi, d'après (SEMATECH, 2000), un design de circuit de 32 millions de transistors en 1999 coûte environ 160 millions de dollars, et requiert 360 personnes pendant trois ans. Toutefois, un design de 130 millions de transistors (en suivant la capacité de fabrication) en 2002, coûterait 360 millions de dollars, et demanderait un personnel de 800 personnes. Ainsi, la complexité des designs augmente de 58% par année, alors que la productivité des concepteurs n'augmente que de 21% par année. Ceci est illustré à la figure I.1.

Pour tenter de vaincre cette disparité entre la productivité et la capacité de production, de nouvelles méthodologies de design sont apparues et continuent d'être développées. Ainsi, on tente d'introduire la réutilisation de composantes sous forme de blocs de propriété intellectuelle, tel que décrit en détails dans (Keating et al., 1999). Une autre méthodologie actuellement en développement est le codesign simultané des composantes logicielles et matérielles. Toutefois, ceci n'est pas encore suffisamment mature pour une utilisation industrielle courante.



Year	Technology	Chip Complexity	Frequency	3 Yr. Design Staff	Staff Cost*
1997	250 nm	13 M Tr.	400	210	90 M
1998	250 nm	20 M Tr.	500	270	120 M
1999	180 nm	32 M Tr.	600	360	160 M
2002	130 nm	130 M Tr.	800	800	360 M

* @ $150K / StaffYr. (In 1997 Dollars)

Figure I.1 Complexité des designs et productivité des concepteurs (SEMATECH, 1999)

La dérivation de modèles est une autre méthode, qui est essentiellement une forme de réutilisation de composantes, mais qui optimise la composante en fonction de l'utilisation que l'on va en faire. Dans le cas d'un processeur, en connaissant à l'avance l'application qui devra être exécutée sur celui-ci, on peut optimiser les circuits qui forment le processeur de façon à obtenir un processeur optimal pour l'application spécifique qui est visée.

Ce mémoire expose les travaux qui ont été effectués dans le cadre du développement d'une méthodologie de dérivation de modèles processeurs embarqués dédiés à une application spécifique. On élaborera également sur le développement d'un modèle de processeur capable d'implanter la méthodologie de dérivation.

Ainsi, le premier chapitre de ce mémoire est essentiellement une revue des réalisations commerciales et de la littérature en ce qui a trait au domaine des processeurs configurables, ainsi que des méthodes de dérivation de processeurs. Il existe encore bien peu de modèles de processeurs configurables disponibles commercialement. En fait, seules deux compagnies sont des joueurs majeurs dans cette industrie pour l'instant. Pour ce qui est des méthodologies de dérivation, bien peu de recherche a été effectuée, et on retrouve que deux méthodes dans la littérature : une ayant pour but la minimisation de la complexité des circuits dans le but de réaliser des systèmes redondants, alors que l'autre est basée sur une méthodologie de dérivation à partir de résultats d'émulation de circuits.

Le deuxième chapitre présente la méthodologie de dérivation. À partir des flots de conception actuels, on élaborera sur le contexte d'utilisation de la méthode. Puis, on expliquera la justification du développement d'une telle méthode. On présentera par la suite un outil d'analyse de microcode permettant d'effectuer les premières analyses pour valider l'utilité de la méthode. On présentera ensuite l'étude de cas qui a été réalisée avec l'analyseur de microcode. Finalement, on détaillera la méthode automatique d'optimisation que l'on propose.

Le troisième chapitre présente le modèle de processeur qui a été développé pour implanter la méthode de dérivation. Ainsi, on expliquera d'abord l'architecture PULSE qui était l'architecture d'origine. Puis les parties importantes du processeur seront décrites. La méthodologie et le style de codage du modèle de processeur seront détaillés. Finalement, on discutera de la validation, ainsi que des outils utilisés tout au long de la réalisation de ce modèle de processeur.

Le quatrième et dernier chapitre présente le travail qui a été réalisé pour automatiser l'optimisation du modèle de processeur en fonction d'une application spécifique. On commencera par présenter les modifications que l'on a faites au modèle de processeur et la configurabilité de celui-ci. Puis, on présentera l'outil d'optimisation lui-même en discutant des options et des compromis. Un exemple d'utilisation, avec l'interface-usager et les différentes options sera présenté. Finalement, les résultats de quelques études de cas seront détaillés.

Chapitre 1

Revue de littérature et réalisations industrielles

Ce chapitre a pour but de présenter les diverses réalisations qui ont été effectuées dans des champs d'intérêts qui s'apparentent aux travaux décrits dans le présent mémoire. En effet, il existe bien peu de recherche et de travaux qui ont été réalisés sur la dérivation de processeurs ou l'optimisation de processeurs en fonction d'applications spécifiques.

On commencera par présenter les modèles de processeurs configurables les plus populaires, soient ceux des compagnies Tensilica, ARC Cores et Improv Systems. Le modèle de cette dernière compagnie est particulièrement intéressant, puisqu'il accompagne une méthode de développement de systèmes à partir d'un concept de plate-forme.

Finalement, on présentera une méthode de dérivation de processeur qui a été proposée pour générer des processeurs tolérants aux fautes, ainsi qu'une autre qui permettait de générer des automates de vision numérique à l'aide d'un émulateur matériel.

1.1 Processeur Xtensa

Une des compagnies majeures dans le domaine des modèles de processeurs configurables est Tensilica. Celle-ci est un fournisseur de blocs de propriété intellectuelle, principalement des processeurs de type RISC (*Reduced Instruction Set Computing*). L'architecture du processeur Xtensa, le principal produit de Tensilica, est une architecture de type 32 bits, entièrement nouvelle, qui correspond davantage aux besoins des systèmes embarqués d'aujourd'hui, comparativement aux modèles de processeurs similaires, qui avaient été conçus à la base pour une utilisation dans des stations de travail. Les caractéristiques de base du processeur sont les suivantes :

- Complexité d'environ 25 000 portes logiques, ce qui donne une utilisation d'environ 0.7 mm^2 de surface de silicium, pour une implantation en technologie 0.18µm.
- Haute performance : Plus de 220 MIPS (Millions d'instructions par seconde) pour une fréquence d'horloge de 200 MHz.
- Faible consommation de puissance : 0.4 mW/MHz pour une configuration typique, avec une implantation en technologie 0.18 µm.

Une des particularités intéressantes de la technologie de Tensilica est la façon dont sont produits les modèles de processeurs. En effet, le modèle de processeur est configurable et peut être adapté en fonction d'une application spécifique. Ainsi, le concepteur dispose

d'un grand nombre d'options pour générer un processeur dédié à application. Tout d'abord, on peut spécifier certains paramètres plus généraux, comme par exemple la fréquence d'opération ciblée, la complexité globale du processeur, le compromis entre la performance et la faible consommation de puissance, etc. Puis, une série d'options plus spécifiques, comme par exemple l'inclusion d'instructions spécifiques (i.e.: *min, max*) ou d'éléments de calcul (multiplicateur-additionneur). De plus, on doit choisir la taille des mémoires caches, ainsi que la fonctionnalité des interruptions. Finalement, on peut également rajouter des instructions au processeurs, en les décrivant dans un langage appelé TIE (*Tensilica Instruction Extension*). Tout ceci correspond à la première étape du flot de conception, illustré à la figure 1.1.

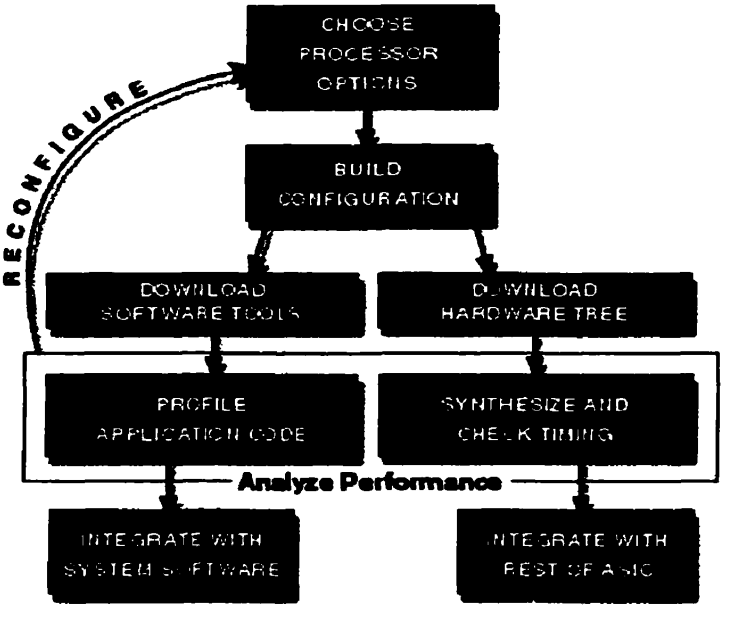


Figure 1.1 Flot de conception d'un processeur Xtensa (Tensilica, 2000)

Une fois les options de configuration choisies, on peut procéder à générer le modèle en langage HDL (*Hardware Description Language*), ainsi que les outils logiciels. Puis, on peut procéder à effectuer le profilage du code de l'application pour déterminer si le modèle est bien adapté à l'application que l'on vise. On peut également procéder à la synthèse du processeur et vérifier si celui-ci répond aux spécifications. Si ce n'est pas le cas, on peut retravailler la configuration et rajouter des instructions spécifiques de façon à améliorer le temps d'exécution de l'application. Finalement, on procède à intégrer le logiciel et le processeur avec le reste du système.

Une des particularités intéressantes de la technologie de Tensilica est le fait que les outils logiciels spécifiques au modèle de processeur dédié soient générés automatiquement. La figure 1.2 illustre le flot complet de l'information lors de la réalisation d'un modèle de processeur. On voit qu'en plus du modèle de processeur HDL et les estimations de *timing*, surface et consommation de puissance, on obtient la suite d'outils GNU pour le développement logiciel.

Ainsi, le compilateur C/C++, l'assembleur/désassembleur ainsi que le simulateur au niveau instruction sont générés automatiquement. Ceux-ci sont adaptés à l'architecture spécifique de la configuration du processeur que l'on conçoit. Ceci facilite grandement le développement du logiciel. De plus, le processeur est supporté par les systèmes d'exploitation temps-réel, ainsi que les outils de simulation de codesign logiciel/matériel les plus populaires.

Finalement, la compagnie propose maintenant un coprocesseur appelé Vectra, qui est construit de la même façon. Celui-ci a été conçu de façon à s'intégrer avec le modèle de processeur Xtensa. Il permet de réaliser de façon efficace des opérations de traitement de signal qui requièrent une puissance de calcul importante. L'architecture du coprocesseur est un croisement entre une architecture SIMD (*Single Instruction Multiple Data*) et un processeur vectoriel standard.

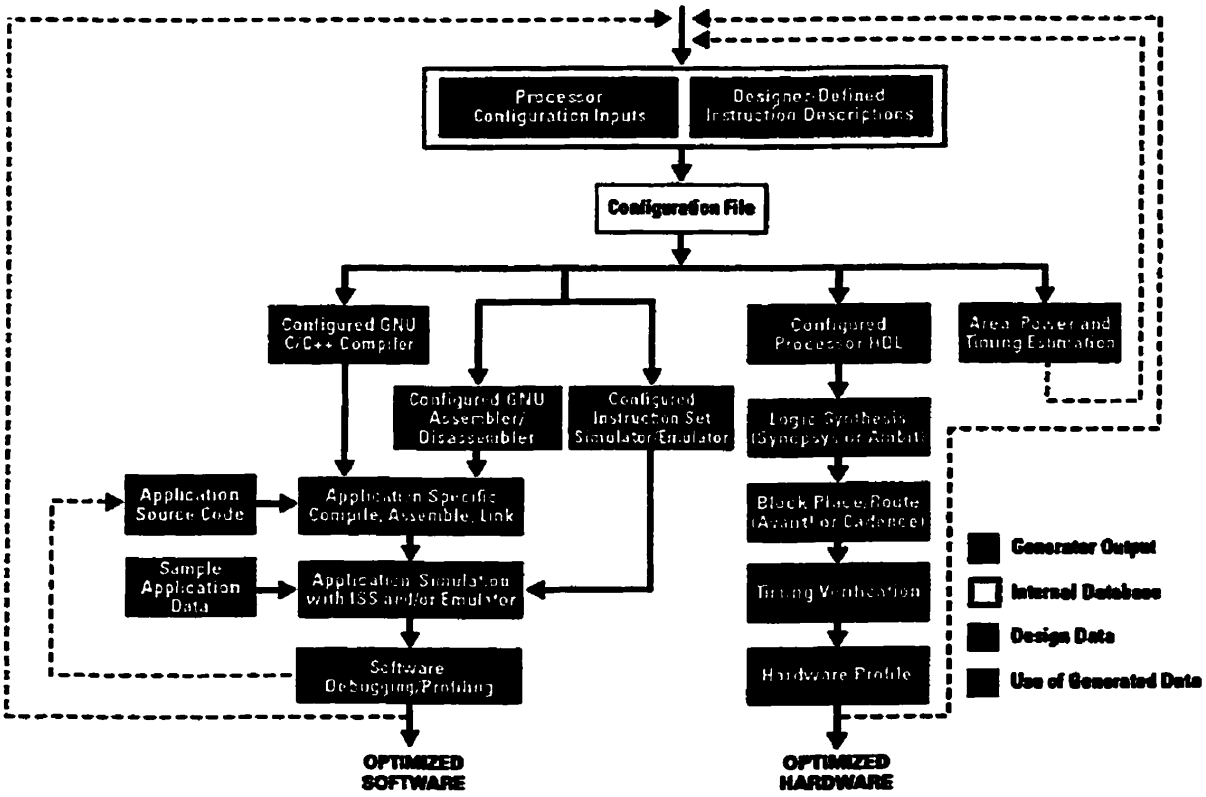


Figure 1.2 Flot d'information pour la conception d'un processeur avec la technologie Tensilica (Tensilica, 2000)

1.2 ARC

La seconde compagnie importante dans le domaine des modèles de processeurs configurables est en fait celle qui fut la première à lancer son produit sur le marché, soit la compagnie ARC Cores Ltd. en 1997.

L'architecture proposée par la compagnie est de type RISC à 32 bits. Celle-ci a été optimisée de façon à pouvoir être fortement configurable. Les performances sont relativement similaires à celles du processeur Xtensa de la compagnie Tensilica. Ainsi, le processeur peut atteindre une fréquence d'opération supérieure à 200 MHz pour une implantation en technologie CMOS de 0.18 µm. La complexité dépend quant à elle des différentes options de configuration. En effet, une configuration minimale ne requiert qu'environ 8000 portes logiques, alors qu'une configuration maximale, avec les extensions pour le traitement de signal numérique, puisse demander plus de 100 000 portes logiques. Pour ce qui est de la consommation de puissance, l'architecture peut être optimisée de façon à minimiser celle-ci, notamment en utilisant des portes pour contrôler le passage de l'horloge dans certaines parties du processeur (*gated clocks*).

Le processus de réalisation d'un modèle de processeur ARC est relativement similaire à celui d'un processeur Xtensa. Toutefois, le nombre d'options de configuration disponibles est plus élevé. Ainsi, le processeur ARC est plus facilement configurable et optimisable que le processeur Xtensa. De plus, de la même façon que pour le processeur Xtensa, on effectue le choix de la configuration que l'on va utiliser pour une application

en trois étapes : on commence par sélectionner les options de configuration. Puis, on génère les fichiers de configuration des différents outils, ainsi que le modèle de processeur. On effectue ensuite un profilage de l'application, de façon à déterminer si le logiciel peut s'exécuter tout en respectant les spécifications, et si le modèle de processeur se synthétise selon les spécifications. Si ce n'est pas le cas, on devra alors procéder à modifier le logiciel ou les choix de configuration du processeur et on recommencera le processus.

Les options de configuration sont relativement similaires à celles du processeur Xtensa, mais dans le cas de ARC, on offre davantage de possibilités et de flexibilité. Tout le travail de la sélection de la configuration est effectué à l'aide d'un environnement graphique intitulé ARChitect. Celui-ci sert à générer le modèle de processeur ainsi que les fichiers de configuration pour les outils et la documentation appropriée pour le processeur dédié.

Contrairement à la technologie de Tensilica, les outils de développement logiciels ne sont pas inclus gratuitement. Ceux-ci doivent être achetés de la compagnie MetaWare. Celle-ci offre une suite complète d'outils de développement, qui permettent de supporter les différentes configurations, ainsi que les instructions que le concepteur pourrait décider de rajouter à son processeur.

1.3 Improv Systems

Ce que la compagnie Improv Systems propose comme produit n'est pas un modèle de processeur configurable, mais plutôt une plate-forme configurable, à partir de laquelle on construit un système complet. En effet, la technologie est disponible sous deux formes : soit sous des blocs d'IP que l'on peut synthétiser et intégrer dans un système (comme par exemple le processeur Jazz) ou bien sous la forme d'un circuit intégré contenant toute la plate-forme configurable.

C'est cette dernière version qui distingue la technologie de cette compagnie des autres. En effet, la plate-forme configurable est une nouvelle approche pour la réalisation de systèmes embarqués. Une plate-forme configurable comprend une série de processeurs, de blocs de mémoires et de ports d'entrées/sorties configurable, un peu de la même façon qu'un FPGA (*Field Programmable Gate Array*) consiste en une série de tables de vérités et d'éléments de mémoire. La différence avec la plate-forme configurable est que le niveau d'abstraction est beaucoup plus élevé.

Le bloc principal de l'architecture de Improv est le processeur Jazz. Celui-ci est un processeur avec un mot d'instruction de type VLIW (*Very Long Instruction Word*) qui permet d'exploiter le parallélisme au niveau des instructions. Le processeur dispose de plusieurs unités de calcul lui permettant d'effectuer un grand nombre d'opérations en parallèle. Celui-ci est illustré à la figure 1.3.

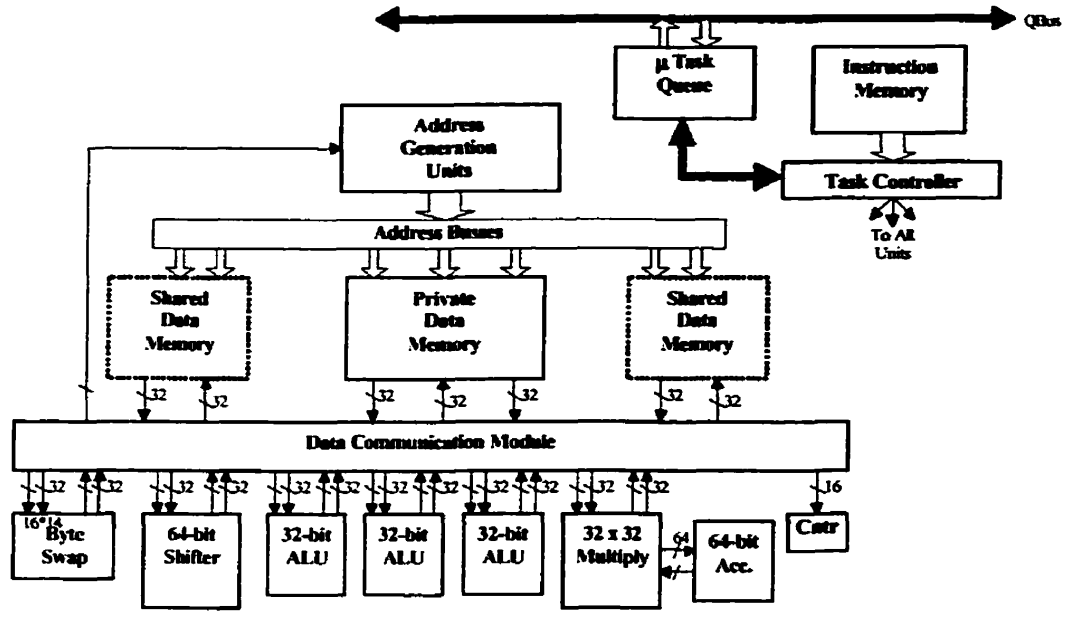


Figure 1.3 Architecture du processeur Jazz (Improv, 2000)

Le processeur peut fonctionner à une fréquence de 150 MHz pour une implantation en 0.18 µm. Comme il peut exécuter entre 7 et 14 instructions par cycle d'horloge, on obtient une performance d'environ 1.5 milliards d'opérations par seconde (BOPS : *Billions of operations per second).*

La plate-forme proposée par Improv consiste en une série de processeurs Jazz, interconnectés entre eux par des mémoires partagées, ainsi qu'un bus. De plus, on dispose de mémoires globales, ainsi que d'un module d'entrées/sorties. L'architecture est illustrée à la figure 1.4 :

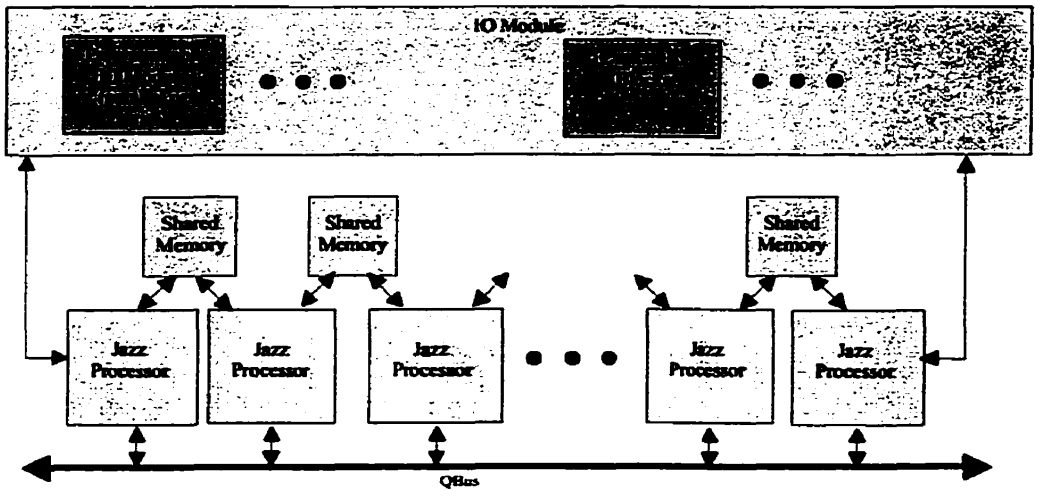


Figure 1.4 Architecture de la plate-forme de Improv (Improv, 2000)

Ainsi, on peut voir que chaque processeur est connecté à son voisin immédiat par une mémoire partagée. En plus, chaque processeur peut communiquer avec n'importe quel autre processeur via un bus partagé. Selon la compagnie, ce type d'architecture, qui ressemble à une architecture à flot de données (les données sont traitées en file par chacun des processeurs) se prête bien à des applications en télécommunication ou en électronique de consommation.

Ce qui est intéressant chez Improv est la façon dont sont développées les applications et donc le système. En effet, celles-ci sont décrites dans un langage de haut niveau : le Java. Une suite d'outils permet le développement et la vérification du code Java. Une fois l'application fonctionnelle, on utilise un compilateur qui va effectuer le partitionnement de l'application pour la distribuer sur les différents processeurs. De plus, il va gérer les communications, tant inter-processeurs, qu'avec les ports d'entrées/sorties. Ceci permet

un développement très rapide d'un système, puisque le concepteur n'a qu'à le décrire dans un langage de haut-niveau. En aucun cas le concepteur ne doit construire des modules en matériel ou écrire des routines en assembleur.

Les avantages de l'utilisation de telles plate-formes sont nombreux. Parmi ceux-ci on note :

- Le volume de production de circuits intégrés n'est pas dépendant du nombre de projets différents, puisqu'on utilise toujours la même plate-forme.
- Les outils de compilation sophistiqués permettent au concepteur de demeurer à un niveau d'abstraction élevé.
- Un environnement de développement unifié pour l'ensemble du système embarqué.
- La possibilité de configurer l'ensemble des composantes de la plate-forme.
- Aucun système d'exploitation n'est nécessaire.

1.4 Dérivation de processeurs pour la tolérance aux fautes

Une méthode de dérivation de processeurs dans le cadre de la réalisation de systèmes tolérants aux fautes a été proposée par un groupe de recherche de l'université de Cottbus en Allemagne (Pflanz et al., 1998).

L'approche proposée pour réaliser des systèmes tolérants aux pannes consiste à dupliquer les structures effectuant les opérations de calcul (comme les processeurs) et à utiliser un système de votation pour s'assurer qu'une majorité des structures donnent le même résultat.

Comme on désirait implanter un tel système de redondance en utilisant des circuits de logique programmable, on devait réduire la complexité des circuits, de façon à pouvoir générer plusieurs instances de chaque circuit dans un seul circuit programmable. Pour ce faire, les auteurs ont développé une méthodologie permettant d'effectuer la synthèse de processeurs spécifiques à une application.

Ainsi, pour effectuer la réduction de la complexité du processeur à synthétiser, on commence par analyser les instructions qui sont utilisées par l'application. Chaque instruction requiert une certaine quantité de circuits qui doivent être implantés pour pouvoir supporter l'instruction. Ainsi, lorsque l'on a déterminé l'ensemble des instructions qui sont utilisées, on peut déterminer les circuits que l'on devra synthétiser pour pouvoir exécuter le code de l'application. Le modèle de processeur est décrit dans une forme structurelle en langage HDL. De cette façon, on peut facilement rajouter ou éliminer certains circuits du processeur. Le flot de design utilisé est illustré à la figure 1.5.

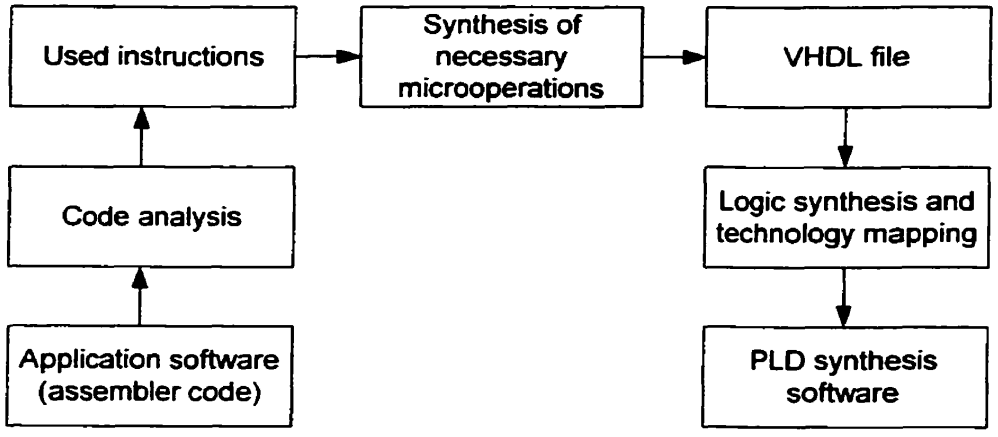


Figure 1.5 Flot de design pour la synthèse spécifique à une application (Pflanz et al., 1998)

Pflanz et al. ont implanté un processeur de type 8085, capable d'exécuter environ 60 microinstructions. L'ensemble du processeur, en excluant la mémoire interne, était d'une complexité d'environ 2500 portes logiques.

Ils ont effectué la synthèse du *contrôleur* de leur processeur pour différents programmes qui servaient d'exemples. Les programmes sont en fait des fonctions fort simples qui permettent d'effectuer des opérations simples avec le processeur. Les résultats qu'ils ont obtenus sont illustrés dans le tableau 1.1. Comme on peut le constater, les résultats de synthèse pour le *contrôleur* permettent de voir une réduction importante des circuits lorsque l'on élimine certaines instructions. De plus, il faut également tenir compte des structures opératives qui vont également être éliminées lorsque l'on retire des instructions.

Tableau 1.1 Résultats quantitatifs de synthèse du contrôleur de processeur (Pflanz et al., 1998)

Fonction	Nombre de portes logiques	Nombre de nœuds
Logique de contrôle entière	258	332
ADD	42	150
LOOP	24	135
ADDSUB	30	138
MULT	43	152
LOGIC	34	139
CONTR1	52	155
CONTR2	53	156

Toute cette méthode de synthèse spécifique de processeurs permet de générer des circuits plus simples, qui vont pouvoir être implantés de façon redondante dans des circuits logiques programmables.

1.5 Dérivation de circuits à partir d'un émulateur

Une dernière méthode de dérivation de processeurs a été proposée dans le cadre de la génération d'automates pour la vision numérique (Kraljic et al., 1996). Le principe est basé sur l'utilisation d'émulateurs matériels, qui permettent de simuler plus rapidement le fonctionnement d'un circuit en développement. Les émulateurs sont utilisés puisqu'ils permettent de diminuer les coûts reliés à la validation et à la simulation d'un circuit.

Ainsi, dans ce projet de recherche, la dérivation est faite à partir d'un processeur de flot de données (*data-flow processor)*. Ainsi, l'émulateur consiste en fait en un vecteur tri-dimensionnel de processeurs de flot de donnés, chacun implantant un opérateur effectuant un traitement sur les données. L'architecture d'un tel processeur est donné à la figure 1.6. La configuration de chacun de ces processeurs permet de déterminer l'application qui va être implanté par l'ensemble du système.

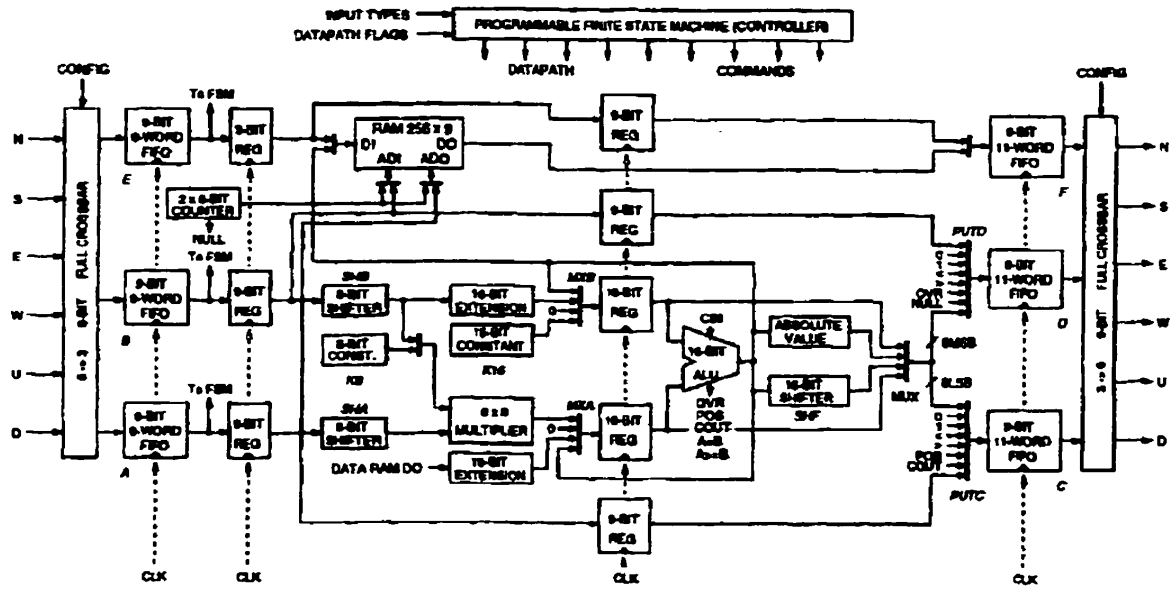


Figure 1.6 Architecture du processeur de flot de données (Kraljic et al., 1996)

Une fois l'application décrite, on peut l'exécuter sur l'émulateur, de façon à valider le comportement de celle-ci. Lorsque le fonctionnement de l'application a été validé, on peut procéder à optimiser le circuit pour la synthèse finale du ASIC (*Application Specific*

Integrated Circuit) qui va implanter l'application. En effet, comme les ressources de chaque processeur ne seront sûrement pas toutes utilisées pour l'implantation d'un opérateur, on peut procéder à effectuer une série d'optimisations de façon à réduire la complexité finale du circuit, tout en conservant la fonctionnalité. Par exemple, les éléments actifs dans le cas d'une opération d'addition sont illustrés à la figure 1.7 (éléments en gris).

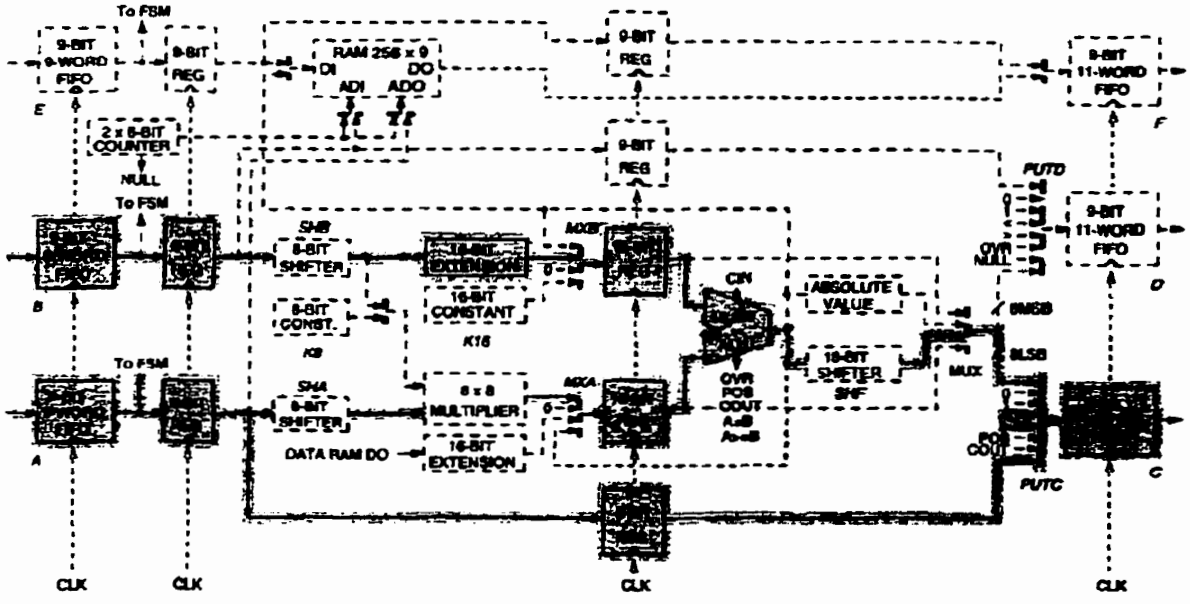


Figure 1.7 Portion du processeur active pour l'opérateur d'addition (Kraljic et al., 1996)

Une série d'optimisations possibles, tant de haut niveau (en retravaillant le graphe de flot de données) que de bas niveau sont décrites dans l'article. Celles-ci permettent de réduire la complexité du circuit qui va être synthétisé. Le flot complet de design, incluant ces optimisations, est illustré à la figure 1.8.

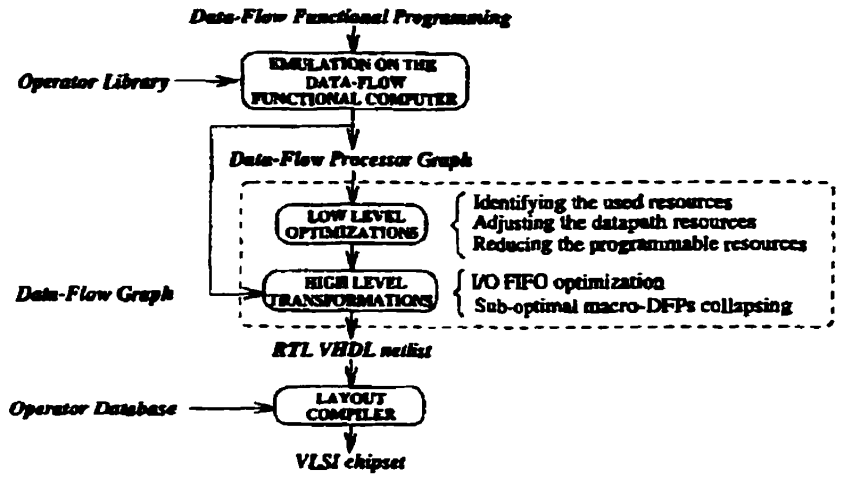


Figure 1.8 Flot de design pour la dérivation à partir des résultats d'émulation (Kraljic et al.,1996)

Des études quantitatives ont été effectuées pour vérifier l'utilité de la méthode. Ainsi, les auteurs ont développé six applications et ont utilisé la méthode de dérivation pour obtenir des circuits optimisés dédiés pour les applications spécifiques. Le tableau 1.2 donne les résultats d'utilisation des ressources globales des processeurs en fonction des différentes applications qui ont été implantées. Comme on peut le constater, l'utilisation des ressources excède rarement utilisées plus de 50% et ce, pour l'ensemble des applications. Ceci fait en sorte que l'on peut facilement optimiser les circuits pour obtenir des circuits synthétisés beaucoup plus petits. Ces résultats vont être particulièrement utiles dans le cadre du projet décrit dans le présent mémoire.

Tableau 1.2 Utilisation globale des ressources pour différents algorithmes (Kraljic et al., 1996)

Name	Edge detector	3 · 3 convolver	2048 word LUT	Nagao-like filter	Defect detector	Erosion
Operating DFPs	13	22	25	86	94	110
I/O ports %	41	43	51	50	45	47
Input FIFOs %	46	56	70	63	56	62
Output FIFOs %	35	31	32	36	34	33
Multiplier %	0	41	0	0	8	0
True ALU op %	23	36	0	37	39	43
Data RAM %	15	18	32	13	12	10
Counter %	30	54	0	41	9	54
Prog RAM size %	3	8	0	9	5	10

Chapitre 2

La méthode de dérivation de modèles de processeurs embarqués dédiés à une application

Ce chapitre présente la méthode de dérivation de processeurs embarqués spécifiques à une application, qui constitue la motivation principale du projet décrit dans le présent mémoire. On présentera dans le prochain chapitre l'architecture du processeur qui a été développé pour permettre d'utiliser la méthode décrite dans ce chapitre. Le dernier chapitre, quant à lui, présentera l'outil d'optimisation utilisé et les détails d'implantation pour appliquer la méthode sur le modèle de processeur développé.

On commencera par présenter le contexte dans lequel la méthode peut s'appliquer. Ainsi, certaines particularités dans le flot de conception actuel de systèmes ont permis de justifier une telle méthode. Puis, la motivation derrière le développement d'une telle méthode sera élaborée et on établira un lien avec des expérimentations antérieures. Ensuite, le flot traditionnel de design utilisé en industrie sera présenté, pour ensuite pouvoir le comparer avec notre méthode.

Un outil d'analyse de microcode sera ensuite décrit. Celui-ci nous a permis d'effectuer les premières analyses pour valider l'utilité de la méthode. On présentera ensuite l'étude de cas qui a été réalisée avec l'analyseur de microcode. Celle-ci est basée sur l'architecture de processeur PULSE qui va être présentée à la section 3.1. On détaillera le résultat des études préliminaires d'optimisations manuelles sur l'architecture PULSE, avant d'élaborer sur la méthode automatique d'optimisation que l'on propose. Celle-ci sera expliquée en détails, et on énumérera les différentes optimisations possibles.

2.1 Introduction

La vaste majorité des systèmes embarqués modernes disposent d'au moins un microprocesseur ou, d'un processeur de traitement de signal numérique (*DSP : Digital Signal Processor).* De plus, il est maintenant courant de retrouver plusieurs microprocesseurs ou DSP dans un système embarqué moyen. Par ailleurs, des plate-formes cibles disposant de plusieurs millions de portes logiques équivalentes sont maintenant couramment disponibles et ce, tant sous la forme de logique programmable (FPGA : *Field Programmable Gate Arrays*) ou de circuits intégrés à application spécifique (ASIC : Application Specific Integrated Circuit). Grâce à ces technologies, la possibilité de créer des systèmes sur une puce (SOC : *System-on-a-chip*) est maintenant une réalité. De plus, le temps disponible pour lancer un produit sur le marché (*time to market*) étant de plus en plus court, les concepteurs n'ont d'autre choix que d'effectuer de

la réutilisation de blocs déjà conçus. Ces blocs sont souvent disponibles sous la forme de blocs de propriété intellectuelle (IP : *Intellectual property*). Ainsi, un grand nombre de processeurs utilisés dans les systèmes embarqués, qui sont souvent utilisés pour exécuter de petits programmes à répétition sont, la majorité du temps, implanté à partir de blocs IP obtenus de différents fournisseurs.

Toutefois, même si de nos jours le nombre de portes logiques disponibles pour l'implantation des circuits est considéré comme étant plus qu'abondant, les exigences des systèmes embarqués font en sorte que le design de tels systèmes est toujours un défi. En effet, ceux-ci ont des besoins particuliers en ce qui a trait à la performance, la dissipation de puissance, l'intégration globale du système et surtout, les coûts de production. Ceci est encore le cas avec les SOC, où les techniques de design traditionnelles avec des blocs d'IP rigides (*hard IP cores*) n'utilisent pas de façon optimale les ressources disponibles. En effet, la plupart des applications n'utilisent pas toutes les composantes et fonctionnalités disponibles pour un modèle de processeur particulier. Ceci se produit notamment parce que le procédé de synthèse du processeur n'est pas lié à l'application spécifique qui va être exécuté sur le processeur dans le système en question.

C'est pourquoi, au cours des deux dernières années, on a vu apparaître de nouveaux types de blocs IP, comme les modèles configurables de processeurs (*soft cores*), tel que présenté dans le chapitre 1. Ces nouveaux types d'IP tentent d'exploiter davantage la flexibilité de la plate-forme SOC, et d'éliminer certains problèmes rencontrés avec les

blocs d'IP rigides, en permettant aux concepteurs de systèmes de configurer leur modèle de processeur pour les besoins de leur application-cible. Mais, encore une fois, l'optimisation réalisée est encore trop peu liée à l'application, et le modèle de processeur ainsi généré ne sera encore une fois pas optimal pour l'utilisation qu'on en fera.

C'est pourquoi une étape d'optimisation pourrait être introduite juste avant l'étape de synthèse. Cette optimisation permettra d'exploiter les particularités de l'application cible pour générer un processeur dédié à cette application. On aurait alors une méthode de dérivation de modèles de processeurs embarqués dédiés à une application.

2.2 Motivations

Un ensemble de facteurs a motivé le développement de ce projet. Tout d'abord, le temps de développement est maintenant un des obstacles majeurs auxquels sont confrontés les concepteurs. Les périodes de développement raccourcissent et la complexité des projets augmente constamment. Il devient alors impératif d'effectuer la réutilisation de certains blocs qui composent un système. Toutefois, il est toujours plus long de concevoir un module particulier à une application à partir de zéro ou en utilisant des bibliothèques de composants, que de dériver automatiquement un modèle spécifique à partir d'un modèle à utilisation générale.

Le temps nécessaire à la validation est particulièrement coûteux. En effet, celui-ci peut facilement accaparer plus des deux tiers du temps nécessaire pour la réalisation d'un système. La dérivation d'un modèle spécifique à partir d'un modèle générique permet de construire un modèle qui est correct par construction. Ainsi, un modèle ne comportant qu'un sous-ensemble de modules d'un modèle complet préalablement validé devrait normalement être fonctionnel, et ne requérir que très peu de validation pour s'assurer de son bon fonctionnement.

Finalement, les modèles dédiés à une application sont plus simples que les modèles à usage général. Ceci est dû au fait que bien souvent une bonne partie de la fonctionnalité non-utilisée est éliminée. À cause de cela, les circuits synthétisés vont être plus simples. Ces circuits plus simples vont être plus performants, dissiper moins de puissance et nécessiter moins de portes logiques (et donc moins de ressources) à l'implantation.

Un projet de recherche antérieur avait prouvé la validité de la méthode de dérivation dans le design d'automates de vision numérique à flot de donnés (Kraljic et al., 1996). Ainsi, dans ce projet de recherche, une application de traitement d'image était en premier lieu implantée et validée en temps-réel sur un émulateur disposant de 1024 processeurs. Ensuite, une description RTL (*Register Transfer Level*) qui représentait l'architecture minimale nécessaire à l'implantation de l'application était extraite en langage VHDL, à partir du graphe de processeurs de l'application.

La méthode de dérivation décrite dans le présent mémoire améliore celle proposée par Kraljic (1996), puisque les modèles de processeurs dédiés seront dérivés directement à partir d'un modèle VHDL validé. De plus, aucune architecture spécifique n'est vraiment visée, bien que dans le cadre de notre projet, un processeur de traitement de signal numérique de type SIMD (*Single Instruction Multiple Data*) sera notre cible initiale. Ainsi, le champ d'application visé dépasse les applications de vision numérique. Finalement, aucun émulateur matériel, coûteux et rapidement obsolète, n'est nécessaire. Seulement les outils traditionnels de design et de simulation, en plus de quelques outils spécifiques simples à développer.

2.3 Méthodes de design traditionnelles

Les méthodes de design traditionnelles de systèmes embarqués, qui impliquent l'utilisation de blocs de propriété intellectuelle (IP) pour implantation sur un système-sur-une-puce, comportent habituellement deux phases distinctes et pratiquement indépendantes, soit le design du matériel et le développement du logiciel. Habituellement, le développement du matériel n'implique que le choix du modèle de processeur qui va disposer du jeu d'instruction, de la puissance de calcul et des périphériques nécessaires pour l'application que l'on désire implanter. Le modèle de processeur, généralement disponible sous la forme d'un modèle rigide (*hard core*), est habituellement décrit avec un sous-ensemble synthétisable d'un langage de description de matériel (HDL : *Hardware Description Language*), comme le VHDL ou le Verilog, ou bien sous la forme

d'une liste de portes logiques et d'interconnexions (*netlist*). Le développement du logiciel est effectué normalement après le choix du modèle de processeur, de façon à pouvoir exploiter les particularités du modèle de processeur choisi. Cet échange d'information (développement du logiciel en fonction du type de processeur) est le seul qui aura lieu entre les flots de design matériel et logiciel et ce, pour toute la durée de la conception du système.

Ainsi, lorsque les parties matérielles et logicielles du système sont complétées, et que les deux ont été simulées et validées en profondeur, le système est alors synthétisé et intégré. Ceci est habituellement réalisé avec des outils de synthèse et de placement et routage dans le cas de la partie matérielle, et un compilateur et un assembleur (et possiblement un éditeur de liens (*linker*)) pour la partie logicielle. Les deux processus sont effectués indépendamment l'un de l'autre, tel qu'illustré à la figure 2.1.

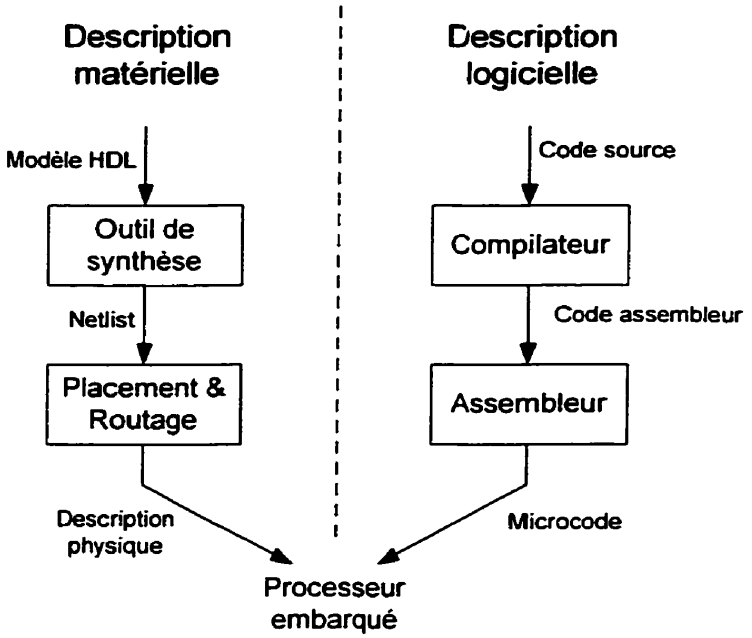


Figure 2.1 Méthode traditionnelle de synthèse d'un processeur embarqué pour une implantation SOC

Cette division du design entre les parties logicielles et matérielles va produire un modèle de processeur embarqué qui va être sous-optimal en fonction de l'application ciblée. En effet, l'outil de synthèse ne peut exploiter aucune information concernant l'application qui va être exécutée sur le processeur de façon à pouvoir éliminer certaines composantes du circuit et en optimiser d'autres.

Les outils de synthèses modernes sont maintenant très puissants, et ils peuvent éliminer ou simplifier efficacement les circuits qui ne sont pas utilisés, en tout ou en partie. Toutefois, ces outils ne peuvent optimiser les circuits pour lesquels ils ne connaissent pas à l'avance les valeurs qui vont en alimenter les entrées. Ainsi, en analysant ce qui va être

donné en entrée au modèle de processeur, on pourra déterminer quels signaux vont permettre l'optimisation de quels circuits, et si ces optimisations sont valables et utiles.

2.4 Outil d'analyse du microcode

Pour déterminer la pertinence d'un couplage serré entre la synthèse d'un modèle de processeur et l'application logicielle qui va être exécutée, un outil d'analyse était nécessaire. Cet outil, une fois complété, peut également servir de base pour l'outil d'optimisation qui est nécessaire pour supporter la méthode.

L'outil d'analyse va utiliser la prédictibilité des signaux d'entrée au modèle de processeur pour réaliser son analyse. Il exploite le fait qu'il connaît à l'avance le microcode qui va être exécuté sur le processeur, puisque le développement de l'application est réalisé avant la synthèse finale du processeur. Ainsi, quand le code de l'application est terminé (que l'on va désigner comme du *firmware*, puisqu'il ne change pas aussi souvent que du logiciel ou *software*), il peut être analysé pour déterminer quelles composantes du processeur vont être utilisées. Le processeur peut donc être optimisé en fonction du *firmware* qu'il va avoir à exécuter et ce, avant la synthèse finale.

2.4.1 Décodage des instructions

On a conçu l'outil d'analyse de microcode pour identifier les ressources non-utilisées par une application spécifique. L'outil en question a été développé pour une architecture en

particulier, soit le processeur PULSE, décrit à la section 3.1. Toutefois, il peut être adapté à n'importe quelle architecture de décodage similaire à celle illustrée à la figure 2.2. L'outil décode une instruction du microcode de la même façon que cela se fait à l'intérieur du processeur visé. Ainsi, les différentes instructions sont décodées en des listes de champs de bits, en fonction du patron d'encodage des différents types d'instructions. Ensuite, certains champs sont passés en entrée à un autre ensemble de décodeurs. Ceux-ci émulent les nombreuses ROM (*Read Only Memory*) de décodage qui sont imbriquées à l'intérieur du processeur. La majorité des signaux qui vont alimenter les autres sections du processeur sont les sorties de ces émulateurs de ROM de décodage. Il est à noter que les optimisations proposées par la méthode décrite dans ce mémoire est également applicable pour les processeurs qui utilisent de la logique câblée (*hardwired logic*) plutôt que des ROM de décodage, même si dans ce cas-ci on ne vas considérer qu'une implantation à base de ROM.

L'analyseur a été conçu pour être aussi générique et indépendant de l'architecture ciblée que possible. Toute l'information concernant la structure des décodeurs d'instructions, des ROMs de décodage et des différents champs de bits sont définis dans des tables contenues dans des fichiers textes. Pour ce qui est des ROMs de décodage, elles sont extraites directement du code VHDL synthétisable, de façon à assurer un décodage authentique. Ainsi, pour porter l'outil d'une architecture de processeur à une autre, on n'a qu'à réécrire les différentes tables décrivant l'architecture, et possiblement modifier quelques portions du code source.

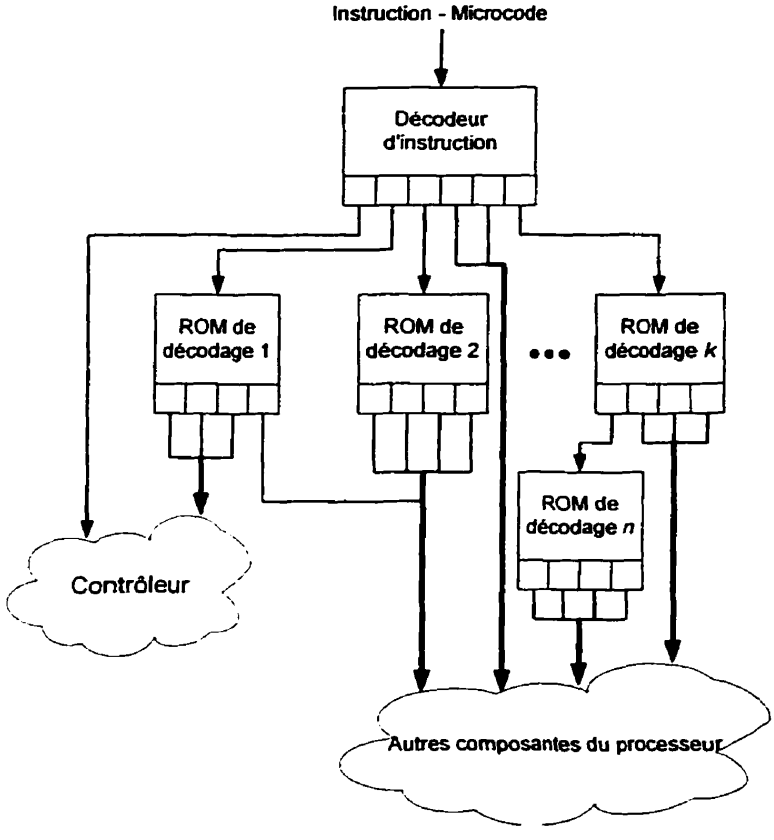


Figure 2.2 Décodage du microcode par l'outil d'analyse

2.4.2 Statistiques d'utilisation

La seconde fonction de l'outil d'analyse est d'accumuler les différentes valeurs que vont prendre les signaux décodes pour toutes les instructions d'un programme en particulier. Lorsqu'un programme en question a été décodé et traité, l'analyseur peut déterminer les valeurs de chaque signal qui sont effectivement utilisées par l'application. Ainsi, pour chaque champ (ou signal) décodé, l'analyseur va fournir le nombre de valeurs réellement utilisées, les valeurs elles-mêmes, ainsi que le nombre de fois qu'elles sont utilisées. Il

calcule également le nombre de bits minimal pour coder un champ, en fonction du nombre de valeurs utilisées.

L'outil permet d'effectuer l'analyse de groupes de signaux. Ainsi, des statistiques relatives aux signaux contrôlant une composante en particulier, comme par exemple une unité arithmétique et logique, peuvent être générées. Ceci permet alors au concepteur d'avoir une bonne idée des ressources qui sont requises par son application. Ces statistiques pourront ensuite être utilisées de façon à optimiser la structure du modèle de processeur, tel qu'expliqué à la section 2.6.

2.5 Étude de cas : Le processeur PULSE V1

Pour réaliser des mesures sur un processeur réel, l'analyseur de microcode a été adapté pour le processeur PULSE (*Parallel Ultra Large Scale Engine*) V1. Celui-ci est un processeur de type SIMD dédié à des applications de traitement d'images numériques en temps réel. Ce processeur a été choisi parce qu'on disposait de son modèle VHDL synthétisable et validé. De plus, un prototype avait été construit avec succès à partir de ce même modèle. Le design du processeur PULSE était d'une complexité d'environ 250 000 portes logiques. De plus, on disposait d'une suite d'applications qui avaient été écrites pour ce processeur.

2.5.1 Suite d'applications de test

Tel que mentionné, une série d'applications a été écrite pendant le développement du processeur PULSE V1. Un ensemble représentatif de programmes typiques pour cette architecture a été choisi pour servir de données de tests pour effectuer des mesures d'utilisations des différentes composantes du processeur. La suite d'application était la suivante :

- CONV3X3 : Une convolution bidimensionnelle 3×3 générique.
- CONVMED : Une convolution bidimensionnelle 3×3 générique suivie d'un filtre médian.
- EDGE : Un filtre de détection de contours.
- HISTO : Une analyse d'image en histogramme.
- IDEA : L'*International Data Encryption Algorithm* (IDEA), utilisé notamment par *Pretty Good Privacy* (PGP), tel que décrit dans Schneier (1995).
- MEDIAN3X3 : Un filtre d'image médian 3×3.
- RSA : Encryptage et décryptage RSA.
- SHRINK : Morphologie mathématique : réduction binaire d'une image.

2.5.2 Résultats d'analyses de la suite de test

Deux ensembles de signaux ont été mesurés pour nos applications de test : le microcode, ainsi que les signaux de contrôle de différents modules de l'élément de calcul de

l'architecture. Les résultats fournis au tableau 2.1 indiquent la taille minimale du microcode nécessaire pour encoder toutes les informations requises à l'exécution de l'application. Ceux-ci incluent l'impact recodage des différents champs, où les indices et les adresses sont optimisés pour minimiser le nombre de bits nécessaires. Ceci est expliqué plus en détails dans la section 2.6 qui décrit les différentes optimisations.

Pour les autres signaux de contrôle, il n'y a ni adresse, ni indexe dans les listes. Ces signaux contrôlent le comportement des différents modules. Les nombres indiqués entre parenthèses dans l'entête correspondent à la largeur des champs dans la structure de processeur originale alors que ceux contenus dans le tableau correspondent aux nombres de signaux de contrôle qui changent durant l'exécution de l'application. Un signal qui ne change pas indique qu'une ou plusieurs fonctionnalités ou mode d'opération d'un module ne sont pas utilisés.

Tableau 2.1 Utilisation des signaux de contrôle pour la suite d'applications de test

Application	Microcode (66)	Contrôle de l'accumulateur (5)	Contrôle de l'ALU (27)	Contrôle du décaleur-barrillet (8)
CONV3X3	17 (26 %)	2 (40 %)	7 (26 %)	3 (38 %)
CONVMED	24 (37 %)	2 (40 %)	20 (75 %)	7 (88 %)
EDGE	16 (25 %)	0 (0 %)	9 (33 %)	0 (0 %)
HISTO	14 (22 %)	0 (0 %)	7 (26 %)	0 (0 %)
IDEA	20 (31 %)	0 (0 %)	15 (66 %)	6 (75 %)
MEDIAN3X3	22 (33 %)	2 (40 %)	19 (71 %)	5 (63 %)
RSA	20 (31 %)	0 (0 %)	12 (45 %)	5 (63 %)
SHRINK	22 (33 %)	0 (0 %)	6 (23 %)	0 (0 %)

Tableau 2.1 Utilisation des signaux de contrôle pour la suite d'applications de test (suite)

Application	Contrôle des canaux d E/S (14)	Contrôle du multiplicateur-additionneur(6)	Contrôle des mémoires (17)	Contrôle des registres (11)
CONV3X3	6 (43 %)	2 (40 %)	2 (12 %)	3 (28 %)
CONVMED	6 (43 %)	4 (67 %)	11 (65 %)	11 (100 %)
EDGE	4 (29 %)	0 (0 %)	2 (12 %)	10 (91 %)
HISTO	3 (22 %)	0 (0 %)	5 (30 %)	0 (0 %)
IDEA	6 (43 %)	4 (67 %)	14 (83 %)	10 (91 %)
MEDIAN3X3	6 (43 %)	4 (67 %)	11 (65 %)	11 (100 %)
RSA	4 (29 %)	5 (84 %)	10 (59 %)	10 (91 %)
SHRINK	4 (29 %)	0 (0 %)	14 (83 %)	0 (0 %)

La première chose que l'on peut observer est que le microcode peut être recodé de façon à être beaucoup plus petit, une fois que l'on applique les optimisations proposées. Ceci nous donne en moyenne un microcode d'une largeur équivalente à environ un tiers de la taille du microcode original.

Pour ce qui est des signaux de contrôle des différents modules de l'élément de calcul, on peut déterminer que ce ne sont pas toutes les applications qui utilisent l'ensemble de ces modules, puisque plusieurs (tous, dans certains cas) signaux demeurent constants durant toute l'exécution de l'application. Par exemple, l'accumulateur n'est utilisé que dans trois des neuf applications de la suite de test. Et pour les modules qui sont effectivement utilisés par les applications, il est rare que la fonctionnalité d'un module soit effectivement entièrement exploitée. Ainsi, pour la suite de test, on peut déterminer que

plusieurs sections des modules de l'élément de calcul ne sont pas utilisées. Ceci nous indique que l'on pourrait facilement éliminer certaines sections sans modifier le comportement du processeur pour une application ciblée. Dans le cas du processeur PULSE V1, ceci pourrait être particulièrement intéressant puisqu'il y a quatre éléments de calcul dans l'architecture, l'élimination d'un composant est en fait répétée quatre fois.

2.5.3 Optimisations manuelles initiales

Une adaptation de l'architecture PULSE V1 pour une implantation sur un FPGA Virtex (Xilinx Inc., 2000) de la compagnie Xilinx a été effectuée. Le *Contrôleur* occupait 3934 *slices*[1] et le chemin de données 6772 *slices*. Des optimisations manuelles ont été effectuées de façon à réduire la complexité de l'architecture du processeur, dans le cas où celui-ci n'est utilisé que comme une machine dédiée à des opérations de multiplication-accumulation (MAC : multiply-accumulate) sur 8-bits. Par exemple, ce type de machine pourrait être utilisé pour implanter des opérations de traitement de signal tels des filtres à réponse impulsionnelle finie (FIR : Finite Impulse Response) ou des convolutions. Des optimisations plutôt simples ont été effectuées, comme par exemple l'ajustement de la largeur du chemin de données à la largeur des données traitées, l'élimination de composantes non-utilisées dans le chemin de données et la réduction de la profondeur des ROM de décodage. Même si ces optimisations étaient plutôt simples, le *Contrôleur* a été

---

[1] *Slice* : Structure élémentaire d'un FPGA Virtex, comprenant 2 tables de vérité (*LUT : Look up table*), deux bascules et de la logique de retenue et de contrôle.

réduit considérablement de taille (1330 *slices*). Même chose dans le cas du chemin de données, qui a été réduit à une taille de 1066 *slices*. L'optimisation globale du processeur a permis de réduire d'environ 75% sa taille. Ces résultats nous ont indiqué que par des optimisations automatiques, on pourrait obtenir de meilleurs résultats en appliquant des optimisations complexes, difficiles à réaliser manuellement.

2.6 Optimisations du modèle

Avec l'utilisation de systèmes-sur-une-puce pour intégrer et optimiser la performance et la dissipation de puissance des systèmes embarqués, ainsi que la disponibilité de plate-formes cibles disposant de plusieurs millions de portes logiques équivalentes, de nouvelles options sont maintenant disponibles pour le concepteur de systèmes embarqués. Notamment, on a vu apparaître des systèmes-sur-une-puce configurables, qui sont une combinaison d'un microprocesseur traditionnel, de logique programmable, de bus dédiés et de mémoire sur une même puce. Ce type de circuit permet d'implanter un système complet facilement, puisque l'ensemble des composantes de base sont déjà implantées (Triscend Inc., 2000)

Un autre type de modèle de processeur est apparu ces dernières années : le modèle de processeur configurable (*soft processor cores*). Ces modèles de processeurs permettent au concepteur de configurer leur processeur en choisissant le jeu d'instructions, les tailles des mémoires caches, et des registres. Ils peuvent aussi permettre l'addition de nouvelles

instructions, registres spéciaux et extensions multimédia. (Levy, 1999), (Burksy, 1999), (Xtensa Application Specific Microprocessor Solutions Overview Handbook. Tensilica Inc.)

Toutefois, bien que ces nouveaux types de modèles de processeurs et les outils logiciels qui les accompagnent offrent davantage de flexibilité au concepteur, ils ne permettent pas une optimisation du modèle de processeur en fonction de l'application ciblée dans le système. En utilisant les statistiques concernant une application cible, comme celles produites par l'outil présenté à la section 2.4, on peut réaliser des optimisations sur le modèle HDL du processeur, ainsi que sur le microcode et ce, avant la synthèse finale. Ceci va donc nous donner la dérivation de modèles de processeurs embarqués, dédiés à une application spécifique. La figure 2.3 illustre cette nouvelle étape dans le flot de design.

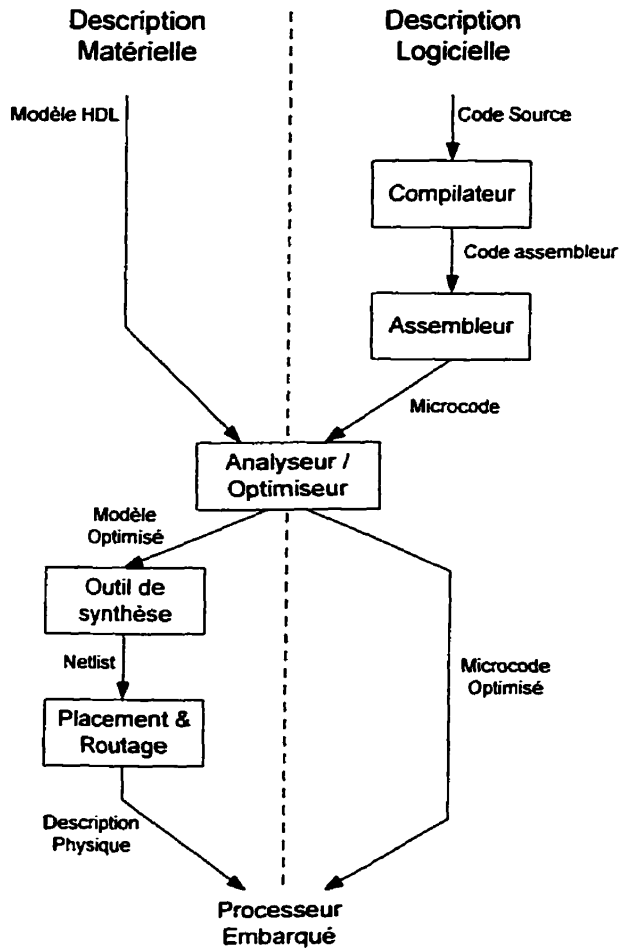


Figure 2.2 Dérivation de modèles de processeurs embarqués dédiés à une application spécifique

Ainsi, dans notre approche, l'application est décrite en écrivant le logiciel qui cible un processeur complet. Puis, durant la phase d'optimisation avant synthèse, le sous-ensemble minimal requis du modèle de processeur est extrait et optimisé, ce qui nous donne une synthèse dédiée à une application. Comme mentionné à la section 1.4, une telle forme de synthèse dédiée a été proposée dans le contexte de processeurs tolérants aux pannes (Pflanz et al., 1998). Ainsi, comme on synthétisait des processeurs plus simples, on pouvait en produire davantage, et ainsi introduire de la redondance.

Les statistiques recueillies sur une application par l'outil d'analyse permettent de réaliser certaines optimisations simples, dont certaines seront décrites ci-bas.

2.6.1 Élimination de ressources

Quand on a pu déterminer qu'un composant en particulier n'est pas utilisé tout au long d'une application spécifique, on peut l'éliminer de la description du modèle de processeur avant de procéder à la synthèse de celui-ci. Par exemple, si le signal indiquant au module de multiplication-addition d'écrire son résultat sur le bus de destination de l'unité arithmétique et logique n'est jamais activé, alors le module de multiplication-addition peut être éliminé, économisant ainsi sur les ressources requises.

Il y a deux façons d'effectuer cela : la première est décrite dans la prochaine section, et implique simplement d'utiliser les capacités de l'outil de synthèse pour éliminer ces ressources inutilisées. La seconde stratégie est utilisée quand l'outil de synthèse ne peut effectuer automatiquement cette optimisation. Ceci peut se produire sous différentes conditions, une d'entre elles est la présence d'un tampon à trois-états qui est toujours en haute-impédance. Certains outils de synthèse n'éliminent pas le circuit qui a comme entrée un tampon à trois-états toujours en haute-impédance. Il faut donc procéder à éliminer ces sections de code avant d'envoyer le modèle à l'outil de synthèse. Ceci peut être facilement réalisé avec des clauses d'inclusion dans le modèle HDL (par exemple, en utilisant des constructions `if [...] generate` en VHDL), qui peuvent être sélectionnées par l'optimiseur. Ceci requiert un modèle HDL codé dans ce but : ainsi, il

faut pouvoir éliminer certains circuits sans que cela n'ait un impact sur la fonctionnalité de l'ensemble du reste du processeur.

2.6.2 Propagation de signaux constants

Tel que spécifié précédemment, les outils de synthèse modernes sont très efficaces quand vient le temps d'optimiser des circuits logiques ou d'éliminer ces circuits quand des valeurs constantes sont données en entrées. Ainsi, quand on peut déterminer que des signaux demeurent constants pour toute l'exécution d'une application durant la phase d'analyse du microcode, on peut ainsi propager ces constantes dans le code, aux entrées des modules appropriés. L'outil de synthèse va alors procéder à remplacer tous les modules inutilisés par des valeurs constantes à leurs sorties. Ceci permet une économie de ressources, tel que présenté à la section précédente. Toutefois, ceci ne requiert pas un modèle codé spécifiquement pour ce type d'optimisation, puisque l'outil de synthèse se charge d'effectuer tout le travail. On n'a qu'à lui donner des valeurs constantes aux entrées.

De plus, cette méthode permet non seulement d'éliminer des circuits inutilisés, mais également de simplifier ceux dont toute la fonctionnalité n'est pas utilisée. Ainsi, si par exemple dans une unité arithmétique et logique on n'utilise qu'une partie des opérations arithmétiques, alors certains signaux de contrôle vont demeurer constants, et l'outil de synthèse va pouvoir optimiser le circuit pour ne conserver que les opérations réellement utilisées.

2.6.3 Tables de constantes locales

Des valeurs constantes sont souvent utilisées dans le microcode, par exemple comme adresses et valeurs de comparaison. La largeur de ces constantes est habituellement grande (i.e. 16 ou 32 bits) alors que le nombre de valeurs réellement utilisé peut être très souvent minime. Ainsi, ceci implique que des bus relativement larges soient routés entre la source des constantes et les modules qui utilisent ces constantes, même si le nombre de valeurs qui va circuler sur ces bus est petit. Quoique ceci ne soit pas un problème énorme pour la majorité des modèles de processeurs, ceci pourrait devenir problématique dans le cas où l'on aurait un processeur SIMD et que l'on doive router ces bus propageant des constantes vers un grand nombre d'éléments de calcul. Ce problème est encore plus sérieux dans le cas où l'on ciblerait des plates-forme réalisées en logique configurable comme par exemple des FPGAs, où les ressources de routage sont bien souvent limitées. Pour éliminer ce problème, les valeurs de constantes peuvent être recodées en des valeurs d'indices dans des tables de constantes. Ainsi, le module qui transmet les valeurs de constantes n'a qu'à transmettre l'indice (qui ne requiert que quelques bits) aux modules destinataires de la valeur. Dans les modules qui reçoivent les constantes, l'indice est utilisé pour retrouver la valeur de constante dans la table de constantes locale. Le reste du traitement est effectué normalement. Bien que cette implantation demande davantage de ressources logiques (pour l'implantation des tables de constantes locales), les besoins en ressources de routage sont diminués, ce qui pourrait être un compromis intéressant. Évidemment, le microcode doit être adapté pour supporter cette modification du modèle.

2.6.4 Recodage de champs

Une autre optimisation implique le recodage des champs de bits qui ne sont pas totalement utilisés. Par exemple, si un processeur a un mot d'instruction de 8 bits, et qu'une application spécifique ne fait usage que de 30 instructions différentes, le code d'instruction pourrait être recodé en un mot de seulement 5 bits. Cette optimisation peut être appliquée à un vaste nombre de types de signaux tels que les mots d'instruction, les modes d'adressage ou les codes de contrôle des unités arithmétiques et logiques. Cela est facile à réaliser si on dispose d'un modèle HDL codé en fonction de cette optimisation, avec des types de signaux fortement hiérarchisés. De plus, on doit utiliser des constantes littérales tout au long du code, et non pas des valeurs numériques. Ainsi, modifier un champ spécifique n'implique que la modification de la largeur du type et des tables de constantes associées. Encore une fois, le microcode doit être adapté pour fonctionner en conjonction avec cette optimisation.

2.6.5 Autres optimisations

L'élimination d'un étage de pipeline pourrait être possible si aucune opération utile n'est effectuée dans cet étage de pipeline. Par exemple, si on n'utilise pas la deuxième partie de l'unité arithmétique et logique, qui s'exécute dans un étage de pipeline différent, on peut l'éliminer puisqu'il ne sert plus qu'à passer des valeurs d'un étage à un autre.

Une dernière optimisation, très simple mais également très efficace, consiste à ajuster la largeur du chemin de données avec celle des données que l'on va effectivement traiter. Ainsi, on pourrait facilement adapter un chemin de données pour cibler des données de 12 bits, 8 bits ou même des données binaires. Un chemin de données ajusté à la largeur des données que l'on va traiter va grandement réduire les besoins en ressources logiques nécessaires pour implanter le chemin de données.

Chapitre 3

Architecture ProDSP

Ce chapitre présente de manière générale l'architecture du processeur de traitement de signaux ProDSP qui a été conçu pour valider la méthode de dérivation proposée dans ce mémoire. Un aperçu des différentes composantes formant le processeur est nécessaire à la compréhension des nombreuses optimisations qui seront proposées ultérieurement. Ainsi, on présentera d'abord l'architecture PULSE qui était l'architecture d'origine du processeur ProDSP. Puis les parties importantes du processeur seront décrites suffisamment en détail pour en apprécier leur complexité ainsi que le choix des optimisations qu'il sera possible d'effectuer sur le modèle de processeur. La méthodologie et le style de codage du modèle de processeur étant particulièrement importants dans le cadre de ce projet, une section y sera consacrée. La validation étant une étape cruciale de la réalisation d'un modèle de processeur, on y présentera une brève description de la méthodologie que l'on a adoptée. Finalement, les outils utilisés tout au long de la réalisation de ce modèle de processeur seront décrits, ainsi que les différents problèmes rencontrés à leur utilisation.

3.1 Architecture PULSE

L'architecture du processeur PULSE (*Parallel Ultra Large Scale Engine*) a été développée au Groupe de Recherche en Microélectronique de l'École Polytechnique de Montréal de 1995 à 1997. Le processeur a été conçu pour effectuer le traitement de signal en temps-réel de signaux vidéo numériques. Des applications typiques dans ce champ d'activité nécessitent la capacité d'effectuer plusieurs millions d'opérations par seconde. Dans le cas d'applications plus complexes, on parle même de plusieurs milliards d'opérations par seconde, sans compter le haut débit nécessaire pour les entrées et sorties de données.

3.1.1 Description

Ainsi, pour palier à ces contraintes, les concepteurs ont conçu une architecture de type SIMD (*Single Instruction Multiple Data*) dont les opérations d'entrées et sorties de données s'effectuent en parallèle avec les opérations de traitement des données. Ce type d'architecture est particulièrement intéressant pour ce genre d'application, puisque chaque élément de calcul (*PE : Processing Element*) peut travailler sur des pixels différents. Chaque PE peut se comparer à un processeur de type C40 (16/32 bits) de la compagnie Texas Instruments (Texas Instruments, Inc.) auquel on a retranché les unités

opératives pour les données en virgule flottante et avait rajouté des fonctionnalités supplémentaires.

La figure 3.1 présente une vue simplifiée de l'architecture PULSE originale. On y voit clairement les principales composantes : les deux canaux de communication (*Nord* et *Sud*), les différents PEs, le *contrôleur*, les générateurs d'adresses ainsi que les mémoires internes et les interfaces. Le rôle de ces différentes composantes étant essentiellement identique à celles que l'on retrouve dans l'architecture ProDSP, leur description sera omise ici puisque l'on couvrira en détails cette nouvelle architecture.

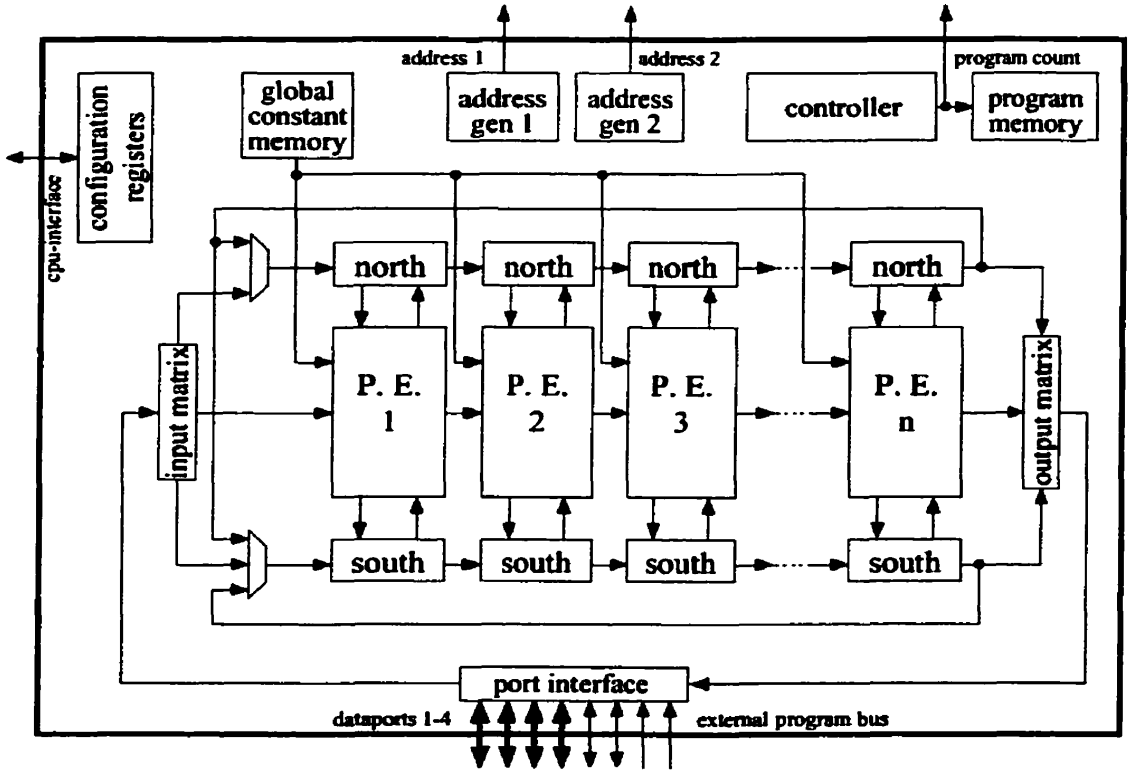


Figure 3.1 Architecture PULSE simplifiée (*Marriott et al., 1998*)

3.1.2 Réalisations

Une version entièrement synthétisable du modèle de l'architecture PULSE a été réalisée. La complexité de ce modèle était d'environ 32 000 lignes de code VHDL. Celui-ci a été validé et, par la suite, un prototype physique a été fabriqué et testé avec succès. Le tout donnait un circuit d'une complexité équivalente à environ 250 000 portes logiques.

Le travail pour le développement d'une deuxième version de l'architecture PULSE (que l'on a nommée V2), qui permettait d'éliminer des limitations de l'architecture initiale a été entamé. Toutefois, seulement quelques spécifications ainsi que certains morceaux du nouveau *contrôleur* ont été réalisés avant la fin du projet.

3.2 ProDSP

L'architecture ProDSP qui a été réalisée est une version dérivée de l'architecture PULSE originale (nommée V1) et de la version V2 qui était en développement. Ainsi, on a récupéré essentiellement l'architecture du chemin de données de V1, auquel on a rajouté un *contrôleur* inspiré des spécifications préliminaires de V2. De plus, on a rajouté certaines fonctionnalités intéressantes, en plus d'éliminer les éléments jugés inutiles.

Le design de la nouvelle architecture a été réalisé à partir du document de spécifications de l'architecture PULSE V1 (Marriott, 1996), du document décrivant le jeu d'instructions de V1 (Bonnello, 1997), du document de spécifications de l'architecture PULSE V2 (Marriott, 1997) en plus du code VHDL du modèle de V1. Un important travail

d'ingénierie inverse (*reverse engineering*) a dû être réalisé sur les 32 000 lignes de code VHDL décrivant le modèle synthétisable de V1, pour réussir à comprendre le fonctionnement interne de l'architecture, puisque la documentation disponible décrivant celle-ci ne faisait qu'une cinquantaine de pages environ.

3.2.1 Description

L'architecture globale de ProDSP est essentiellement la même que celle de PULSE. En effet, il s'agit encore une fois d'un processeur de type SIMD dans lequel les communications s'effectuent en parallèle avec le traitement des données. La figure 3.2 illustre les principales composantes de l'architecture ProDSP : le *Contrôleur*, qui gère le séquençage des instructions à être exécutées ainsi que le contrôle des autres modules; le module de *Configuration et d'État*, qui emmagasine la configuration et l'état des autres modules; les *Éléments de calcul*, qui effectuent le traitement sur les données et les communications; et les *Générateurs d'adresses*, qui produisent les adresses nécessaires pour accéder aux mémoires de données externes.

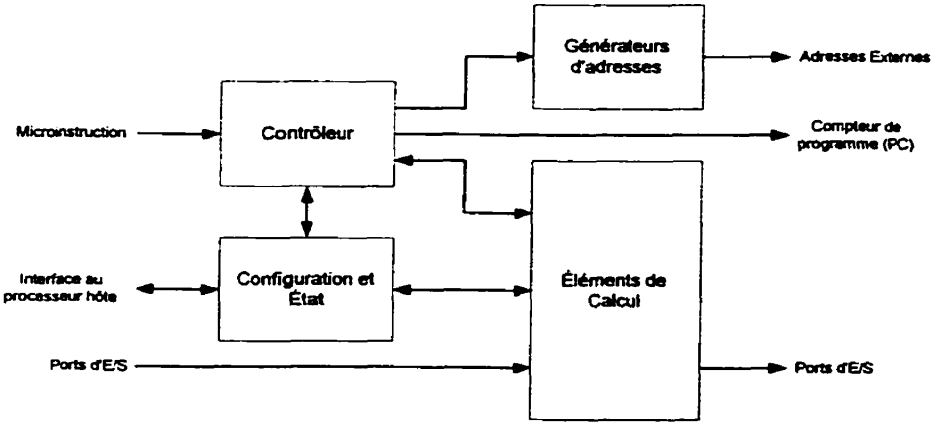


Figure 3.2 Diagramme bloc de l'architecture ProDSP

3.2.2 Jeu d'instructions et architecture du microcode

ProDSP est une architecture à mot d'instruction très long (*VLIW* : *Very Large Instruction Word*). Les sous-systèmes de contrôle/calcul, de communication et d'entrée/sortie disposent tous d'une section distincte dans le microcode. Ainsi, les trois sous-systèmes sont indépendants et peuvent opérer en parallèle avec les autres, sans aucune restriction.

Les sous-systèmes de communication et d'entrée/sortie sont toujours codés de la même façon pour toutes les instructions. Les champs qui gèrent la section de contrôle/calcul requièrent 9 types d'encodage différents, en fonction des instructions qui y sont codés. Huit de ces encodages servent aux différentes instructions de séquençage et de contrôle, l'autre étant commun pour toutes les instructions de calcul.

Il existe également deux autres champs qui sont communs à tous les types d'encodage. Le premier sert à encoder directement dans le microcode le nombre d'instructions nulles (*NOP : No Instruction*) à exécuter séquentiellement après l'exécution de l'instruction codée. Des études réalisées sur le code qui tournait sur l'architecture PULSE ont démontré qu'il était courant d'avoir des instructions nulles dans le code, pour pouvoir synchroniser les entrées/sorties avec le calcul. Avec le nombre de NOPs à effectuer directement encodés dans le microcode, on peut économiser substantiellement sur la taille du programme. Par exemple, pour justifier le fait que l'on utilise 3 bits pour le codage des NOPs dans chaque instruction, il faut qu'au moins un NOP soit exécuté à toutes les 122 (taille du microcode de ProDSP) / 3 bits $\approx$ 40 instructions. L'expérience avec PULSE a

montré que des NOPs apparaissaient beaucoup plus souvent qu'à toutes les 40 instructions.

L'autre champ qui est commun à toutes les instructions est celui des instructions vectorielles. On peut coder directement dans le microcode le nombre de fois que l'on veut répéter une instruction. Ainsi, cette instruction va être répétée un certain nombre de fois, et les adresses de source et de destinations des opérandes peuvent être auto-incrémentées à chaque itération (en fonction des modes d'adressage utilisés.)

Le microcode permet d'utiliser n'importe quelle combinaison de sources comme opérandes pour les opérations de calcul. Plusieurs destinations sont également possibles pour le résultat d'une seule opération.

Toutes ces possibilités font en sorte que la syntaxe assembleur pour cette architecture est longue et passablement complexe. Un exemple complet serait :

```
#8 add r1, r2+ -> memA[0]+, acc+, nport || nrr || ssl || write_a || readnorth_b || #4 nop
```

Dans cet exemple, nous avons une instruction vectorielle qui va être exécutée 8 fois (d'où le #8 en avant de l'instruction). L'instruction à exécuter est une addition, et la valeur du registre r1 va être additionnée aux registres r2 à r9 (séquentiellement, à cause du + après le r2) et les 8 résultats vont être écrits dans la mémoire A aux adresses 0 à 7. L'accumulateur va sommer les résultats et ceux-ci vont également être écrits dans le port

d'entrée/sortie *Nord*. En parallèle à ceci, le canal *Nord* va effectuer une rotation vers la droite et le canal *Sud* va effectuer un décalage vers la gauche. Une opération d'écriture sera effectuée à chaque itération sur le port A et une lecture sur le port B. Finalement, lorsque les 8 itérations de l'instruction vectorielle seront complétées, 4 NOPs seront exécutés.

Comme on a pu le voir dans l'exemple précédent, le nombre de combinaisons possibles pour les instructions, les sources et les destinations est considérable. De plus, un grand nombre de combinaisons d'opérations et d'opérandes vont donner des résultats invalides. Ainsi, il est relativement difficile de programmer correctement cette architecture sans des outils logiciels appropriés.

3.2.3 Contrôleur

Le *Contrôleur* est, avec les PEs, la partie la plus complexe de toute l'architecture du processeur ProDSP. Il est responsable de plusieurs choses, notamment le décodage et le séquençage des instructions des PEs, ainsi que la génération du compteur de programme.

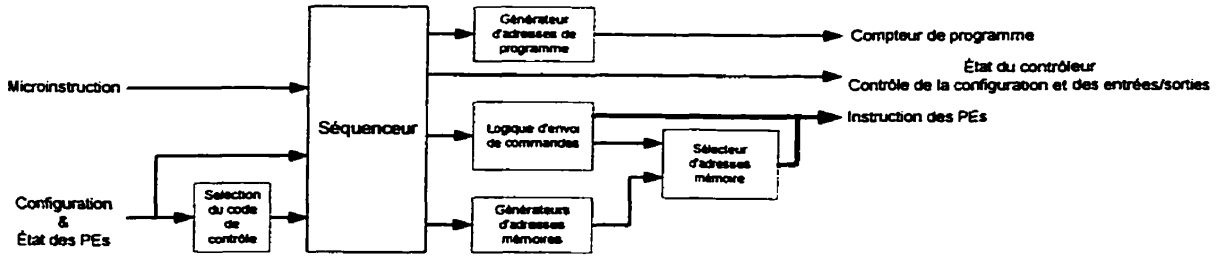


Figure 3.3 Diagramme bloc du Contrôleur ProDSP

Comme on peut le voir à la figure 3.3, le *Contrôleur* est composé de plusieurs parties, la principale étant le *Séquenceur*. Ce module est également le plus complexe de toute l'architecture ProDSP. Il est responsable du décodage et du séquençage de toutes les instructions. La machine à états finis qui détermine l'état d'activité du processeur fait partie du *Séquenceur*. Le processeur peut être en état d'attente, en exécution normale, en exécution vectorielle ou bien en exécution de NOPs.

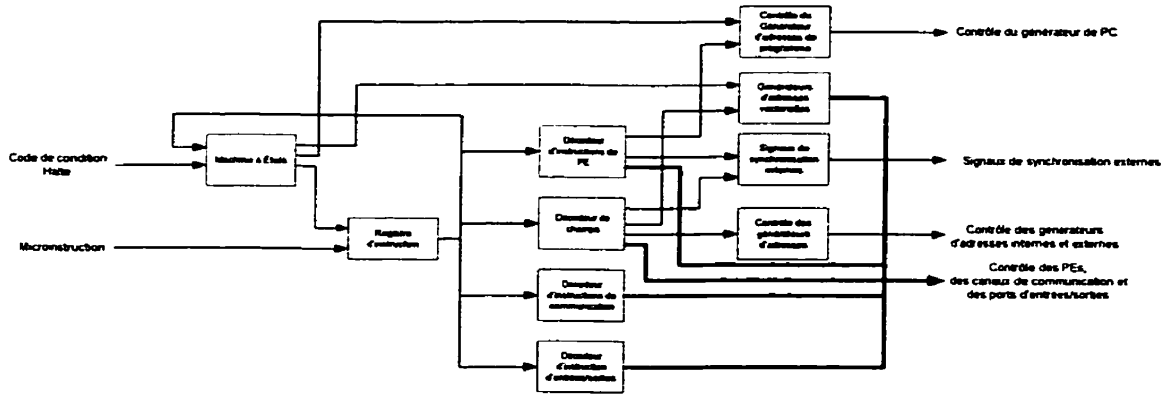


Figure 3.4 Diagramme bloc du Séquenceur

Comme on peut le voir sur la figure 3.4, l'instruction à être exécutée est conservée dans un registre d'instruction pour les cas où l'on doit effectuer des instructions vectorielles pendant plusieurs cycles. Cette instruction est décodée par une série de modules qui sont essentiellement des mémoires de décodage (ROM). Ces décodeurs incluent ceux pour les instructions des PEs, des canaux de communication ainsi que des ports d'entrées/sorties. On retrouve également les générateurs de signaux de contrôle pour le *Générateur d'adresse de programme* ainsi que les *Générateurs d'adresses internes et externes*. C'est également dans le *Séquenceur* que se trouve le *Générateur d'adresses vectorielles*, qui

permet d'incrémenter les adresses utilisées lors de l'exécution des instructions vectorielles. Finalement, la dernière composante du *Séquenceur* est le générateur de signaux de synchronisation externes. Ces signaux (quatre par défaut) permettent de signaler un état à des composantes externes au processeur. Des instructions permettent de les mettre à un état actif (*Set*), inactif (*Reset*) ou d'inverser leur état (*Toggle*.)

À la sortie du *Séquenceur* se trouve la *Logique d'envoi de commandes* (*Command Issue Logic*). Ce module est essentiellement quatre séries de bascules permettant de retarder les signaux décodés par le *Séquenceur* pour qu'ils arrivent lors du bon cycle d'exécution de l'instruction dans le pipeline. Ainsi, les signaux allant dans le premier étage de pipeline (Lecture d'opérande) seront retardés d'un seul cycle, alors que ceux contrôlant les modules du dernier étage (Écriture) seront retardés de quatre cycles.

Le Générateur d'adresses de programme permet de déterminer la valeur du compteur de programme (PC) pour la prochaine instruction à exécuter. Ce module comprend deux piles, un registre, un incrémenteur et un multiplexeur. La première pile permet de stocker les adresses de retour lors des appels de sous-routines, alors que la seconde permet de stocker les valeurs des compteurs pour les boucles itératives. L'incrémenteur permet de déterminer la valeur du prochain PC lors d'une exécution séquentielle, et le multiplexeur permet de choisir parmi toutes les valeurs disponibles celle qui va être utilisée comme prochain compteur de programme.

Le *Sélecteur de code de contrôle* n'est en fait qu'un multiplexeur qui permet de choisir la source du signal qui va être utilisée comme condition dans le cas des opérations de branchement conditionnel. Les signaux possibles sont ceux décrivant l'état des PEs, des piles du *Générateur d'adresses de programme*, des *Générateurs d'adresses externes*, ainsi que les signaux de synchronisation qui sont disponibles en lecture.

Quatre *Générateurs d'adresses mémoires* sont disponibles. Deux sont assignés à la banque de mémoire A des PEs, et les deux autres sont assignés à la banque B. Pour chacun de ces groupes de deux, un des générateurs est utilisé pour calculer les adresses de lecture des mémoires, alors que l'autre est utilisé pour les adresses d'écritures. Ils sont implantés comme des compteurs modulo, et sont configurables pendant l'exécution, avec des paramètres pour les valeurs minimales et maximales. Ils peuvent ainsi être facilement utilisés pour implanter des tampons circulaires (*circular buffer*) ou des bandes avec recouvrement.

Le *Sélecteur d'adresses mémoires* est essentiellement un multiplexeur complexe, avec des éléments mémoires pour conserver de l'information relative aux états précédents. Son utilité est de déterminer quelles adresses mémoires vont être utilisées pour accéder aux mémoires durant le cycle d'opération. En effet, comme les mémoires sont à simple port, on ne peut les utiliser que dans un seul étage de pipeline à chaque cycle (aléas de données). Ainsi, on doit déterminer si l'on doit utiliser les adresses provenant des générateurs ou bien celles fournies par le *séquenceur*.

3.2.4 Configuration et État

Ce sous-système de l'architecture ProDSP est divisé en trois parties. La première est l'interface au processeur hôte, qui permet de configurer le processeur ProDSP en plaçant les valeurs appropriées dans les registres de configuration. On peut également lire l'état du processeur en effectuant la lecture des registres d'état. L'implantation actuelle permet d'interface le processeur ProDSP avec un processeur de type 68HC11, 8051 ou un équivalent. Toutefois, il serait très facile de modifier ce module pour l'adapter à un autre type d'interface de communication avec le monde extérieur.

Le second module de ce sous-système est celui qui contient les registres qui permettent de configurer le processeur. Parmi les différents éléments que l'on peut configurer, on retrouve l'adresse de départ du programme, la possibilité de geler le processeur par un signal externe, le comportement de l'accumulateur dans le cas des débordements, ainsi que la possibilité d'écrire dans la mémoire de constantes. Les valeurs de ces registres peuvent être écrites et lues par l'interface au processeur hôte ou directement par le *Contrôleur*, en utilisant les instructions appropriées dans l'application.

Finalement, le dernier module de ce sous-système est celui qui contient les registres d'état du processeur. On retrouve deux types de registres : les registres simples et ceux qui peuvent déclencher un signal d'alarme. Ces derniers peuvent conserver la valeur d'un état particulier jusqu'à ce que celui-ci soit lu par le processeur hôte. Les signaux qui

peuvent déclencher des alarmes peuvent être choisis par un masque pour chaque registre. Ainsi, on peut implanter un système d'interruptions avec le processeur hôte.

3.2.5 Communication et Entrées/Sorties

Le sous-système de communication et d'entrées/sorties de l'architecture ProDSP fait partie du vecteur SIMD et est composé de deux parties : les canaux de communication et les ports d'entrées/sorties. Chaque PE est connecté à deux éléments de registres à décalages, sur lesquels il peut effectuer des opérations de lecture et d'écriture de données. Les éléments de ces registres à décalages sont connectés ensemble pour former deux chaînes de registres à décalages : les canaux de communication *Nord* et *Sud*. Ces deux canaux sont connectés ensemble à chacune des extrémités par deux modules d'interface de canaux. Ces interfaces, avec les canaux, les générateurs d'adresses externes (décrits à la section 3.2.7) et quelques modules supplémentaires dans le *Contrôleur* forment le sous-système de communication. On a également deux ports d'entrées/sorties, que l'on nomme port A (ou gauche) et port B (ou droit.) Les canaux de communication et les interfaces sont illustrés à la figure 3.5.

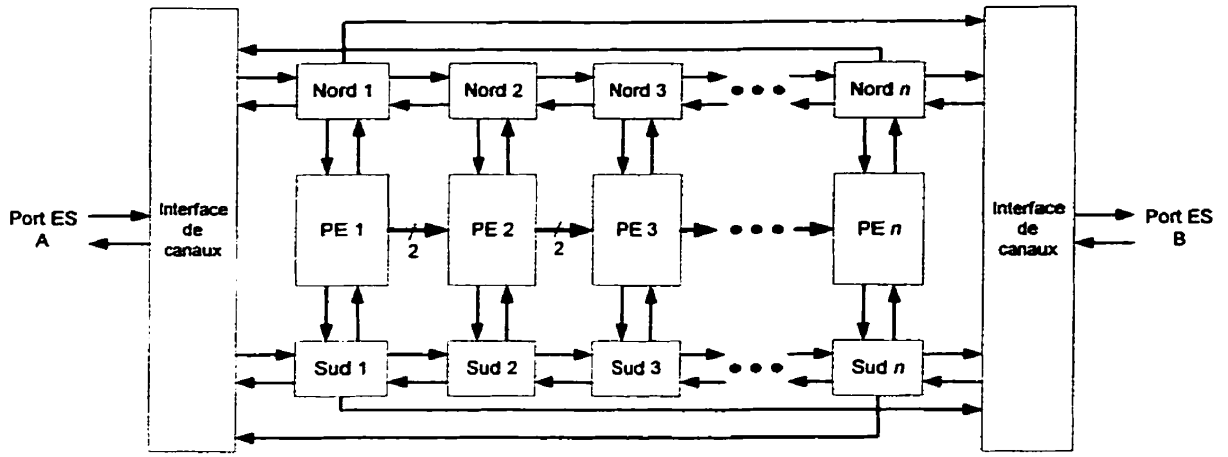


Figure 3.5 Diagramme bloc des canaux de communication

Comme on peut le voir sur la figure 3.5, chaque élément des registres à décalages peut recevoir ses données de trois sources différentes : l'élément directement à sa gauche, celui à sa droite ou bien le PE qui lui est associé. Dans le cas des éléments qui sont situés aux extrémités des canaux, ils peuvent recevoir leurs données de leur voisin, du PE associé ou de l'interface de canaux. Les interfaces de canaux reçoivent leurs données des éléments avoisinants des deux canaux, des éléments opposés des deux canaux ainsi que des ports d'entrées/sorties. Ainsi, les éléments situés aux extrémités peuvent recevoir leur données de toutes ces sources.

L'ensemble du sous-système de communication permet l'entrée et la sortie simultanée de données (via les ports) et peut également recevoir ou transférer des données aux PEs. Cette architecture permet de réaliser facilement un grand nombre d'opérations de communication de données entre les PEs. Ainsi, on peut effectuer des rotations et des décalages sur les deux canaux de façon indépendante. De plus, on peut effectuer des

opérations encore plus complexes avec les interfaces, en utilisant les deux canaux simultanément. En tout, plus de 30 opérations différentes sont possibles.

Un exemple possible de cette architecture de communication est une application de traitement d'images. Dans ce cas, on peut connecter le port A et le canal *Nord* à la mémoire contenant l'image source, et le port B et le canal *Sud* à la mémoire qui va recevoir l'image traitée. Ainsi, lorsque tous les PEs auront terminé le traitement sur leur pixel respectif, ils vont placer le pixel résultant sur le canal *Sud*. Après, on va effectuer *n* décalages vers la droite sur le canal *Sud* pour sortir les résultats vers la mémoire destination. Pendant le temps de traitement sur les pixels, *n* nouveaux pixels sont chargés via le canal *Nord*, pour permettre d'avoir de nouvelles données une fois que le traitement sera terminé. Ainsi, les PEs ne seront jamais en attente lors de l'entrée ou la sortie de données.

Il existe un autre moyen de communication entre les PEs : la chaîne d'accumulation. Elle est utilisée pour transférer des données entre les PEs et ainsi former une chaîne d'accumulation. Elle est utilisée par le module de multiplication-addition, dans le cas d'une opération MADDL (*Multiply and Add from Left PE*). La chaîne transfère le résultat de la multiplication-addition d'un PE vers le PE sur sa droite.

3.2.6 Élément de calcul (PE)

L'élément de calcul de l'architecture ProDSP est essentiellement un chemin de données complet, capable de réaliser un grand nombre d'opérations. Il comprend des sources et destinations de données, trois éléments fonctionnels, ainsi qu'un accumulateur. L'architecture du pipeline d'exécution pour le PE de ProDSP est donnée à la figure 3.6. Le pipeline est divisé en quatre étages : le premier étant la lecture des opérandes, les deux suivants sont l'exécution des opérations et finalement le dernier est l'écriture des résultats.

Parmi les sources possibles pour les opérandes, on retrouve deux mémoires : la première est locale à chaque PE, et est implantée comme deux banques, relativement profondes (chacune ayant la largeur d'un mot de donnée complet), de mémoire à simple-port. Ces

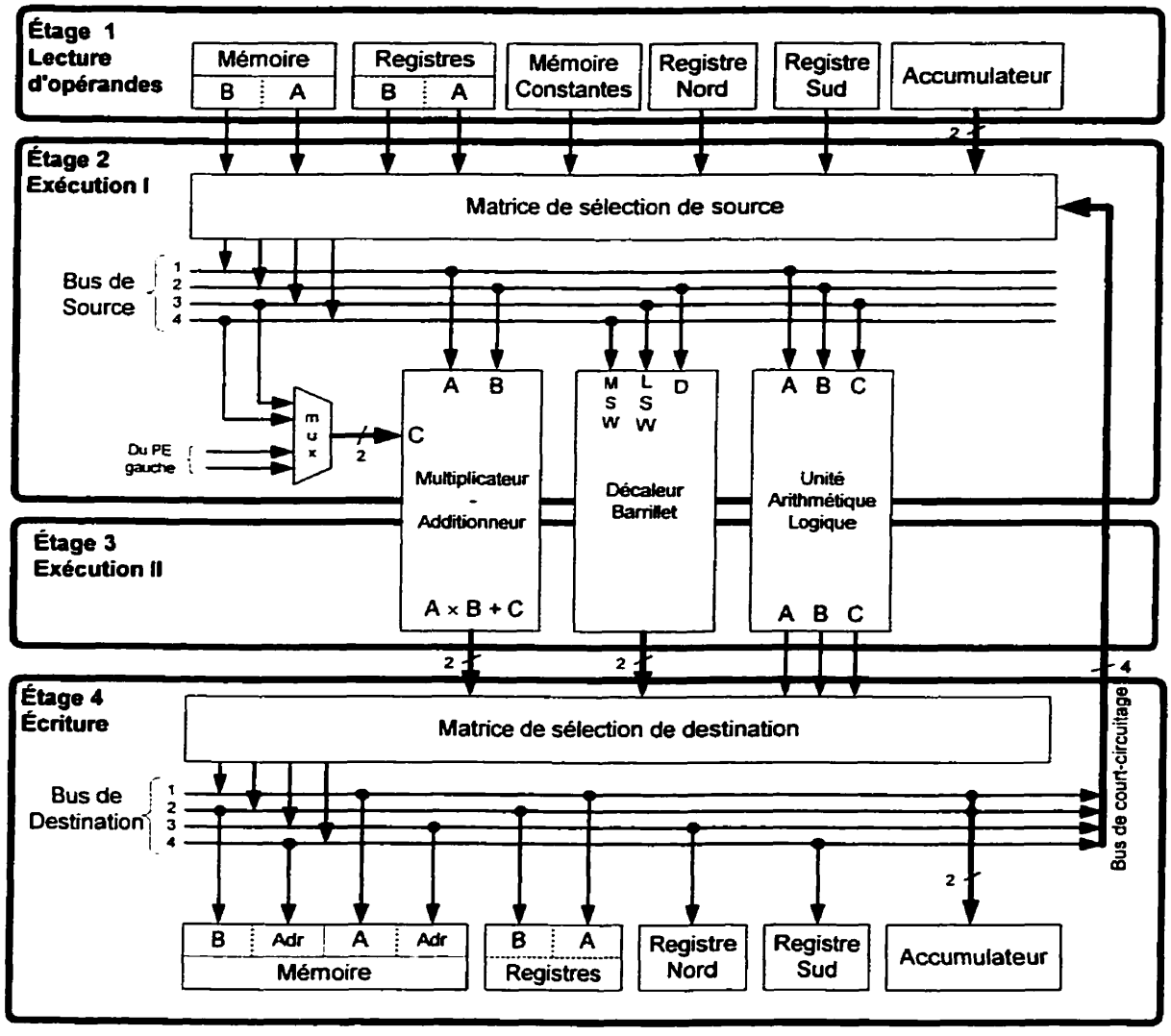


Figure 3.6 Pipeline du chemin de données de ProDSP

mémoires peuvent être adressées par l'adresse fournie par le *Contrôleur*, ou bien par le registre d'adresse indirecte, pour un accès par pointeur. Ces deux banques de mémoires sont utilisées pour le stockage de données à moyen et long termes. La seconde mémoire qui est disponible est commune à tous les PEs, et elle sert à distribuer les constantes qui sont utilisées dans les algorithmes. Les PEs n'ont accès à cette mémoire qu'en lecture

seulement. Il est toutefois possible d'écrire des valeurs dans la mémoire lors de l'exécution mais pour ce faire, il faut passer par un mécanisme différent qui implique les ports d'entrées/sorties.

Deux banques de registres sont disponibles pour le stockage de données à court terme. Ces registres sont implantés sous la forme de mémoires à double-ports, de plus faible profondeur que les mémoires, mais de même largeur (un mot de données.) Les registres peuvent être utilisés dans un même cycle comme source et comme destination, contrairement aux mémoires, puisqu'ils sont à double-ports.

Les registres *Nord* et *Sud*, qui correspondent aux valeurs comprises dans les éléments des registres à décalages de communication respectifs, peuvent également être utilisés comme source pour les opérandes. Finalement, l'accumulateur peut également être utilisé comme source d'opérandes.

Les trois unités fonctionnelles sont actives pendant les deux cycles d'exécution. La première, le multiplicateur-additionneur permet d'effectuer des opérations de multiplication et des opérations multiplication-addition. Tel que présenté dans la section 3.2.5 sur les communications, une chaîne d'accumulation permet de court-circuiter le chemin normal des données dans les cas où l'on voudrait additionner le résultat du PE précédent au produit calculé dans un PE. Ceci est utile pour accélérer l'exécution de certains algorithmes.

Le décaleur-barrillet est complet, et permet de réaliser toute la gamme des opérations normalement disponibles pour une telle unité de traitement. Ainsi, on peut réaliser des rotations et des décalages, tant arithmétiques que logiques. Il opère sur des doubles-mots.

L'unité arithmétique et logique comprend essentiellement trois parties : un module permettant de réaliser les opérations arithmétiques, un autre pour les opérations logiques et finalement un autre pour les opérations à trois points. Les opérations arithmétiques disponibles sont l'addition et la soustraction, avec ou sans retenue ou emprunt. Les opérations logiques disponibles sont celles que l'on retrouve normalement dans une unité de ce type (AND, OR, NAND, NOR, XOR, XNOR, NOT.) Finalement, l'unité pour les opérations à trois points permet d'effectuer des opérations non-linéaires qui normalement requièrent plusieurs cycles. Par exemple, des opérations permettant de déterminer le minimum, le maximum, la médiane ou de mettre en ordre croissant trois opérandes. Ces instructions sont utiles dans le cas de l'implantation de certains algorithmes, comme des filtres médians.

Le dernier étage de pipeline est celui d'écriture des résultats. Il accède essentiellement aux même modules que l'étage de lecture, à l'exception de la mémoire de constantes (qui ne peut être écrite de cette façon.) De plus, on peut écrire dans les registres d'adresses mémoire pour un accès indirect. Il est à noter également que l'accumulateur a une précision interne d'un double mot plus un bit. Par ailleurs, il dispose d'une unité configurable permettant de gérer les débordements de différentes façons.

Une autre unité importante du PE ne figure pas sur le diagramme de pipeline. Il s'agit du *Contrôleur d'activité*. Chaque PE dispose de ce module qui lui permet de déterminer s'il doit être actif ou non lors de l'exécution d'une instruction. Une pile conservant les états d'activité permet d'implanter des instructions *si-alors-sinon* (*If-Then-Else*). Par exemple, sur une opération *If*, tous les PEs qui satisfont la condition de test vont être actifs, les autres inactifs. Puis, lorsque l'opération *else* est exécutée, ceux qui étaient inactifs deviennent actifs et vice-versa.

3.2.7 Générateurs d'adresses externes

Deux générateurs d'adresses externes sont disponibles pour permettre l'accès aux mémoires qui permettent le stockage des données d'entrée et de sortie. Ces deux générateurs sont des générateurs d'adresses linéaires à n dimensions. Ces générateurs permettent d'accéder à un espace mémoire linéaire comme si celui-ci était en fait une mémoire à n dimensions. Un exemple de ce type d'accès serait une séquence d'images : chaque image est en fait un espace à deux dimensions, et la suite d'image fait en sorte que l'on a un espace à trois dimensions. Dans plusieurs applications en traitement d'image, des séquences complexes d'adresses sont nécessaires. Ce type de générateur simplifie grandement la complexité d'accès aux mémoires.

L'algorithme utilisé (Bélanger et al., 1997) est en fait assez simple : pour chaque dimension, on a deux variables : le déplacement (*stride*) et la forme *(shape*). Le déplacement spécifie la valeur à ajouter à chaque étape, et la forme indique le nombre

d'étapes pour cette dimension. Lorsqu'une dimension est épuisée (le nombre d'étapes effectuées est égal à la forme), on remet à zéro le compteur d'étape pour cette dimension et on active la dimension supérieure.

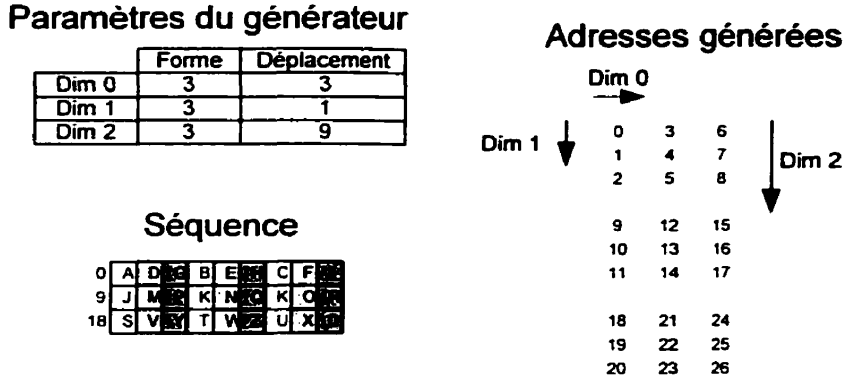


Figure 3.7 Exemple de génération d'adresses

Dans l'exemple simple donné à la figure 3.7, on veut accéder aux données de la matrice à deux dimensions en colonnes, une ligne à la fois. La séquence d'accès des données est donné par les lettres A à Z (et un 0 final). La configuration des paramètres du générateur (déplacement et forme) est donnée. À droite, on voit la séquence d'adresses générées par cette configuration.

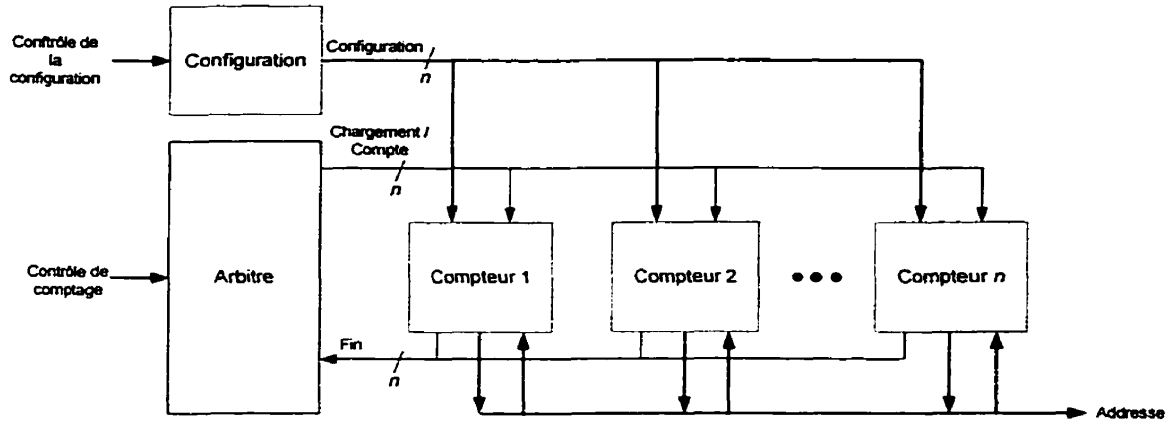


Figure 3.8 Diagramme bloc du générateur d'adresses linéaires à *n* dimensions

Le diagramme bloc de l'implantation est donné à la figure 3.8. Comme on peut le voir sur la figure, trois modules servent à construire un générateur d'adresses. Le premier est celui qui stocke la configuration de chacune des dimensions : la forme et le déplacement. Pour chaque dimension du générateur, on a une instance d'un compteur spécial. Celui-ci effectue deux opérations lorsqu'il est activé. Tout d'abord, il compte le nombre de fois qu'il a été activé et compare cette valeur avec le paramètre de forme qui lui est associé. Si le compteur a été activé le même nombre de fois que la forme, le compteur signale à l'arbitre (présenté plus loin) qu'il a terminé sa séquence, et il est remis à zéro. La seconde opération qu'un compteur doit effectuer lorsqu'il est activé est d'incrémenter l'adresse qui se trouve sur le bus de sa valeur de déplacement appropriée. Comme on peut le voir sur la figure, l'adresse est transmise en même temps à tous les compteurs, mais seulement un de ceux-ci va l'incrémenter. La sélection du compteur qui va être activé durant un cycle est effectuée par l'arbitre. Celui-ci effectue son choix en se basant sur les signaux de fin de séquence de chacun des compteurs. Le compteur qui est activé va émettre sa

valeur sur le bus, alors que tous les autres vont être désactivés. C'est cette valeur qui va être l'adresse générée pour ce cycle.

Cette implantation est très simple, et offre d'excellentes performances. En effet, une adresse peut être générée à chaque coup d'horloge et ce sans utiliser de pipelinage. De plus, le chemin critique de ce circuit ne passe que par l'arbitre et un seul compteur, peu importe le nombre de dimensions dans le générateur. Ainsi, la croissance du délai n'est pas linéaire avec le nombre de dimensions, puisqu'elle ne dépend pas du nombre de compteurs.

3.3 Méthodologie et style de codage

Dans le cadre d'un tel projet, il est important de bien définir la méthodologie ainsi que le style utilisés pour décrire l'architecture ProDSP. En effet, les contraintes de notre projet étaient doubles : tout d'abord, le processeur PULSE comportait environ 35 000 lignes de code en VHDL pour sa description synthétisable. Ceci laissait présager que le processeur ProDSP, bien que plus simple, allait être d'une complexité se situant dans le même ordre de grandeur que PULSE. Ainsi, il fallait s'assurer d'uniformiser l'ensemble du code pour en faciliter le codage, le déboguage et la compréhension. De plus, notre projet visait l'optimisation d'un modèle de processeur en fonction d'une application spécifique. Il fallait donc s'assurer qu'il serait facile de modifier le modèle de façon à effectuer les optimisations.

Après réflexion, il a été décidé de mettre la complexité de l'ensemble du procédé d'optimisation dans la description du modèle de processeur en code VHDL. Ce choix, et les compromis qu'il implique, seront traités dans le chapitre décrivant l'optimiseur (section 4.1).

3.3.1 Paramètres et modules génériques

Étant donné que l'on désire paramétriser l'ensemble du processeur ProDSP de façon à pouvoir l'optimiser pour une application spécifique, il faut donc pouvoir configurer l'ensemble des modules. Le langage de description de matériel que l'on utilise, le VHDL, permet de configurer de plusieurs façons des modules paramétrables.

La première façon, qui est la plus évidente, est d'utiliser des clauses génériques dans la description de l'entité. Celles-ci permettent de spécifier les paramètres du module à chaque instanciation de celui-ci. Ceci est particulièrement avantageux lorsque l'on a des modules qui vont être instanciés plusieurs fois, avec des paramètres différents. Toutefois, le désavantage majeur apparaît lorsque l'on a une hiérarchie profonde de modules. En effet, on doit passer les paramètres à partir du module supérieur et propager les constantes aux modules inférieurs. Dans le cas où l'on a une hiérarchie complexe, on a un grand nombre de paramètres pour le module le plus haut dans la hiérarchie. À chaque niveau où l'on descend, on doit passer les paramètres qui ne sont pas utilisés aux niveaux inférieurs. Un exemple est donné par le diagramme de la figure 3.9, où les instances sont

représentées par des rectangles, les paramètres par des points noirs et les clauses génériques par des zones ombragées. Ainsi, on voit bien toute la complexité que cela peut entraîner lorsque l'on a plusieurs niveaux de hiérarchie, et un grand nombre de paramètres. Par ailleurs, si on devait rajouter un paramètre en cours de développement, on devra le rajouter à tous les niveaux de la hiérarchie, de façon à ce qu'il se rende jusqu'au niveau supérieur.

L'autre façon de propager les valeurs des paramètres est d'utiliser les paquets (*package*) d'informations qui sont disponibles dans le langage VHDL. Ces paquets permettent de stocker notamment des types de données, des constantes et des fonctions. On n'a qu'à

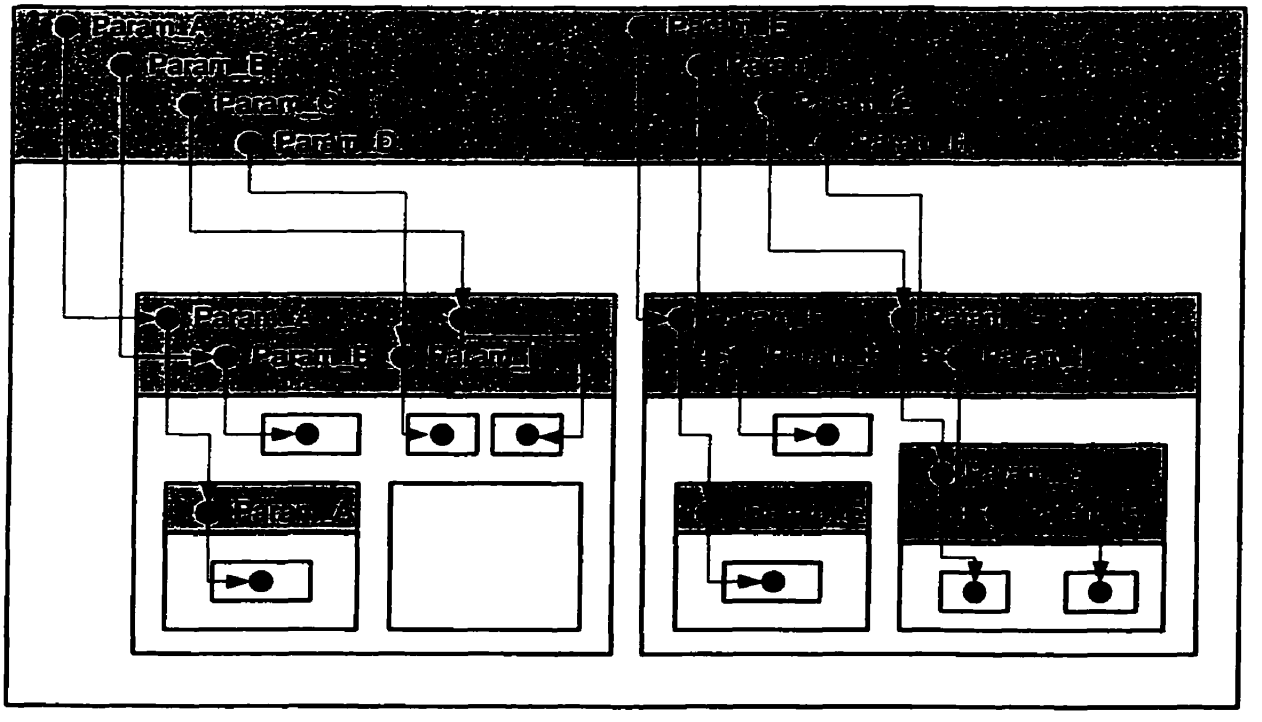


Figure 3.9 Passage de paramètres par clauses génériques

compiler le paquet dans une librairie, et inclure cette librairie dans un module pour que celui-ci puisse utiliser tout ce que le paquet contient. Ainsi, on n'a qu'à déclarer une table de constantes et chaque module ayant besoin de paramètres n'à qu'à déclarer l'utilisation de la table de constantes (qui se trouve dans un *package*). Comme le modèle de processeur ProDSP avait besoin d'un très grand nombre de paramètres, c'est cette option que l'on a privilégiée.

Une troisième alternative consiste à utiliser des clauses génériques lorsque l'on déclare les entités, mais de ne pas passer les valeurs des paramètres d'un niveau de hiérarchie à un autre. À la place, on déclare l'utilisation d'une table de constantes et ce sont ces constantes que l'on passe comme paramètres aux clauses génériques lors de l'instanciation. On évite alors toute la complexité du passage de paramètres hiérarchique.

3.3.2 Types et configuration

Vu la complexité de l'ensemble du modèle de processeur de l'architecture ProDSP, il existe un grand nombre de types de signaux de contrôle qui sont nécessaires pour permettre aux différents modules de communiquer entre eux.

Dans le modèle du processeur PULSE, on avait essentiellement utilisé des signaux de type `std_logic_vector` ou bien des types énumérés. L'emploi de types énumérés est correct, mais n'était pas répandu à l'ensemble du modèle. En fait, il y a avait de grandes différences entre les façons de représenter l'information dans les différentes sections du processeur. Ceci rendait difficile l'interaction entre les différentes sections,

puisqu'il fallait convertir les signaux d'un type à un autre pour qu'ils puissent communiquer ensemble.

On a donc décidé de déclarer des types surdéfinis pour chacun des types de signaux qui étaient utilisés. Bien souvent, ceux-ci étaient en fait des `std_logic_vector` mais l'utilisation d'un type nommé rendait beaucoup plus clair le code résultant. Ainsi, il est plus facile de comprendre le type de données qui est reçu par un port de type `Data_type` qu'un type `std_logic_vector( 15 downto 0 )`. Les types sont déclarés dans un paquet qui est disponible pour tous les modules qui composent le processeur. Ainsi, une modification sur un type dans le fichier de types a des répercussions automatiques sur tous les modules qui utilisent ce type de données. On peut donc effectuer facilement une modification à toute la hiérarchie de modules en ne changeant qu'un seul type. Par exemple, si l'on désire que la largeur du chemin de données soit différente de celle prévue par défaut, on n'a qu'à modifier le type `Data_type` pour que la taille de tous les modules soient automatiquement ajustée.

Par ailleurs, on a eu souvent recours au type `record` qui permet de regrouper plusieurs signaux dans un seul type. On effectue alors un regroupement logique des signaux de contrôle d'un module. On divise alors en une série de regroupements de signaux ceux nécessaires au contrôle d'un module. Ainsi, on peut passer directement tous les signaux nécessaires au contrôle de ce module en passant beaucoup moins de signaux. Ceci fait en sorte que les déclarations d'entités sont beaucoup moins engorgées. De ce fait, lorsqu'on

va instancier ces entités, et que l'on va connecter ensemble ces différents modules, le nombre de signaux s'en verra réduit, et cela permettra une plus grande compréhension du code. Ainsi, on a vu des entités passer de plus de 30 ports d'entrée/sortie à moins de 10. La réduction de déclaration de signaux nécessaires dans les architectures était également considérable.

De plus, on a utilisé uniquement des constantes littérales pour toutes les valeurs numériques que pouvaient prendre les signaux. Ces constantes étaient également placées dans un paquet qui était accessible à tous les modules. Ainsi, il est nettement plus facile de modifier toutes les apparitions d'une constante spécifique. En effet, on n'a qu'à modifier la valeur qui se trouve dans la table globale et les modifications sur tous les modules qui utilisent cette constante sont ainsi effectuées. De plus, la compréhension du code est grandement facilitée encore une fois.

La construction de la hiérarchie de types se fait selon la configuration que l'on va avoir pour le processeur ProDSP. Ceci est fait de la façon suivante : on a une librairie, appelée `ProDSP_Lib` qui contient trois paquets (`packages`). Le premier contient une série de fonctions qui permettent de déterminer certaines valeurs. Par exemple, on retrouve une fonction permettant de déterminer le plus petit nombre de bits nécessaires pour coder une valeur. On a aussi un fichier de configuration. C'est dans ce fichier que l'on va déterminer la configuration de l'ensemble des composantes du processeur ProDSP. On retrouve un grand nombre de constantes, comme par exemple le nombre d'éléments de

calcul (PE), les paramètres des générateurs d'adresses, l'utilisation de différents modules etc. On a également des tables de valeur booléennes pour l'utilisation des différents mots de code (*opcodes*) et des modes d'adressages. Toutes les valeurs de ces constantes peuvent être déterminées par le concepteur ou, mieux encore, par un outil d'optimisation automatique.

À partir de ces paramètres de configuration, on peut déclarer les types qui seront utilisés. Ainsi, on va pouvoir déclarer des types de taille minimale en fonction de l'utilisation qui est faite des différents modules. Un exemple concret de ceci serait de diminuer la taille de l'*opcode* en fonction du nombre d'*opcodes* qui sont réellement utilisé dans l'application. La difficulté rencontrée ici réside dans le fait que l'on désire effectuer ces opérations de façon quasi-automatique. En effet, on désire que seul le fichier de configuration soit modifiable et que tous les types soient dérivés automatiquement. On a dû avoir recours à l'utilisation de plusieurs fonctions dans la déclaration des types pour réussir cela. Notamment, une grande difficulté résidait dans l'assignation des valeurs des constantes. Par exemple, on a comme modes d'adressage de la mémoire dans l'architectures ProDSP les modes suivants :

Tableau 3.1 Modes d'adressage de la mémoire de l'architecture ProDSP

Mode	Valeur
Direct – Source	0
Indirect – Source	1
Vectoriel – Source	2
Modulo – Source	3
Direct – Destination	4
Indirect – Destination	5
Vectoriel – Destination	6
Modulo – Destination	7

Toutefois, dans une application particulière, on n'utilise que trois modes, soit les modes vectoriels en source et destination, ainsi que le mode modulo en source. De ce fait, les modes d'adressages qui étaient codés sur 3 bits, peuvent maintenant l'être sur 2 bits. Par ailleurs, les valeurs que doivent prendre les modes d'adressages sont les suivantes :

Tableau 3.2 Modes d'adressage de la mémoire pour un exemple d'application

Mode	Valeur
Vectoriel – Source	0
Modulo – Source	1
Vectoriel – Destination	2

Ainsi, il faut que le fichier de types soit capable de générer les bonnes valeurs de constantes à partir des paramètres d'utilisation de chacun des modes d'adressage. Ceci est relativement difficile et requière des déclarations assez particulières en VHDL.

La figure 3.10 illustre le flot d'informations nécessaires à la génération et à la propagation des types. Ainsi, on voit que c'est la librairie qui génère et propage les types. Ceux-ci sont déterminés à partir de la configuration, qui est le seul élément modifiable pour la génération des types spécifiques à une application. Le reste de la description de l'architecture ne change pas en fonction de l'application.

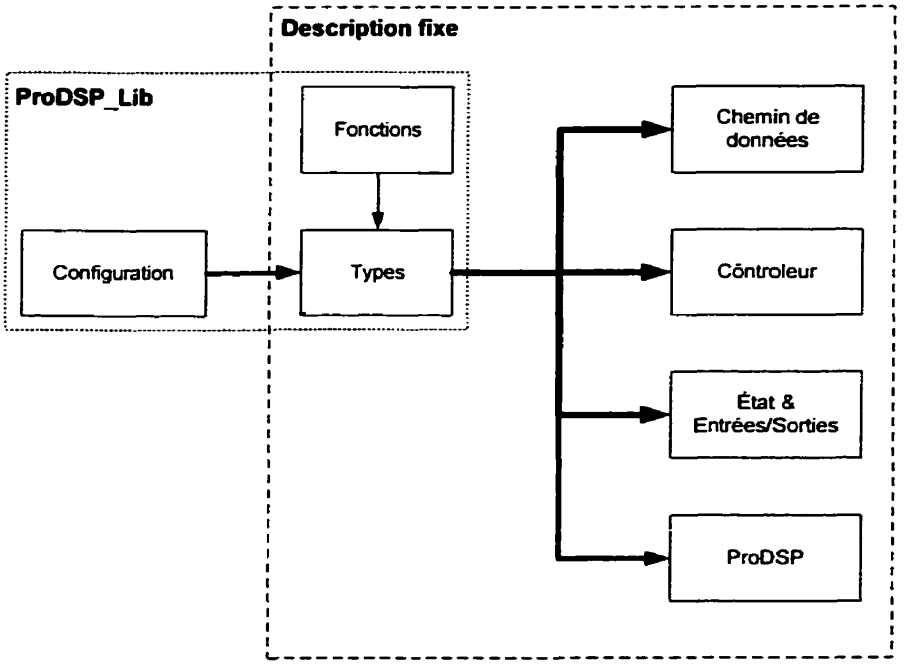


Figure 3.10 Génération et propagation des types

3.4 Validation

Dans tout projet de design, une étape de validation est cruciale pour s'assurer que l'implantation rencontre bien les spécifications du projet. Cette étape permet de déceler les erreurs de conception avant que le produit entre en fabrication. Il existe plusieurs façons de réaliser la validation d'un design numérique tel que celui qui a été réalisé dans le cadre de ce projet. Ainsi, la vérification formelle et la simulation sont deux méthodes employées. Toutefois, la seconde est beaucoup plus mature et utilisée de façon prédominante en industrie. C'est pourquoi c'est cette méthode que l'on a décidé d'utiliser.

On a donc divisé le procédé de validation en trois parties distinctes : la validation du *Chemin de données*, la validation du *Contrôleur* et des autres composantes, ainsi que la validation du modèle de processeur dans son ensemble.

La presque totalité de la validation du C*hemin de données* a été effectuée par Alexandre Fortin, dans le cadre de son projet de maîtrise. Le tout est décrit dans (Fortin, 2000).

Pour ce qui est du *Contrôleur*, celui-ci a été validé par une suite de tests de régression (*regression tests*). Ainsi, on a commencé par valider séparément différents modules simples. Puis, on a rassemblé les différents modules de base ensemble de façon à simuler un module de plus en plus complexe. On a ainsi procédé progressivement jusqu'à ce que des sections complètes du *Contrôleur* soient fonctionnelles. Finalement, on a procédé à

simuler l'ensemble du *Contrôleur*. Les bancs de tests utilisés étaient généralement assez simples : on utilisait un fichier pour générer les suites d'entrées (qui correspondaient normalement aux instructions d'un programme que le *Contrôleur* devrait séquencer), et les sorties des modules étaient observées.

Pour les modules autres que ceux du chemin de données et du *Contrôleur*, très peu ont été simulés de façon exhaustive. En effet, ceux-ci sont pour la plupart relativement similaires à ceux de l'architecture PULSE. On a plutôt procédé à examiner le résultat de synthèse des modules de ProDSP et de PULSE. Ainsi, en comparant le circuit logique résultant, on peut arriver à déterminer l'exactitude du code. En effet, les deux devaient être identiques ou, au moins fonctionnellement équivalents. Ainsi, si les portes logiques générées pour un circuit étaient différentes de celles du circuit de référence, mais que les fonctions logiques finalement implantées étaient les mêmes, on considérait le circuit comme valide. La plupart des modules ainsi comparés étaient assez simples.

Pour la simulation du modèle complet, on a utilisé un banc de test comportant quatre composantes. On avait deux mémoires de données : une pour celles en entrée (branchée sur le port d'entrées/sorties A), une pour celles en sortie (branchée sur le port d'entrées/sorties B). On avait également une mémoire de programme, permettant de fournir au modèle ProDSP une série d'instructions de microcode à exécuter. Finalement, on avait également une instance du modèle de processeur complet ProDSP. Le tout est illustré à la figure 3.11.

Il est important de noter que bien que des simulations ont été effectuées pour s'assurer du bon fonctionnement de base du processeur, le processus de validation n'a pas été effectué de façon complètement exhaustive. En effet, le manque de temps, de ressources et de personnel, combinés avec l'importante complexité de l'architecture ProDSP, n'ont pas permis de réaliser une validation que l'on pourrait considérer comme complète, selon des normes industrielles. Ainsi, le processus permettant de valider un modèle complètement comporte un grand nombre d'étapes que l'on n'a pas pu implanter. Ainsi, d'après Mednick (2000), le procédé de validation qui est normalement utilisé en industrie peut comporter des étapes de tests dirigés (ce qu'on a effectué par nos simulations), de tests pseudo-aléatoires, de covérification logicielle/matérielle et des tests sur des planches d'évaluations avec des applications réelles.

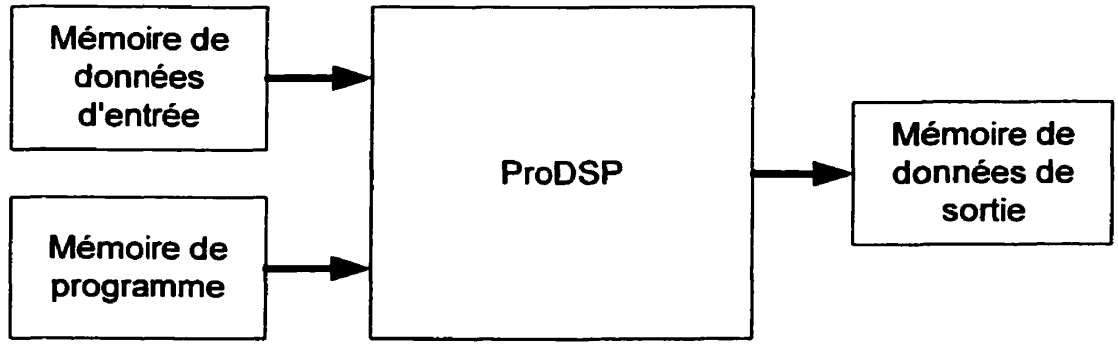


Figure 3.11 Banc de test du modèle ProDSP

3.5 Outils logiciels

Dans tout projet de microélectronique, un nombre important d'outils sont nécessaires pour sa réalisation et ce projet ne fait pas exception. Dans le cadre de ce projet, on a procédé aux étapes de simulation, de synthèse et de placement routage (quoique cette dernière étape a été réalisée par Fortin (2000) et non par l'auteur).

3.5.1 Simulation

Tel que présenté à la section précédente, la simulation est absolument nécessaire pour s'assurer de la validité du bon fonctionnement du circuit que l'on crée. Ainsi, le concepteur peut simuler son modèle pour s'assurer que celui-ci respecte l'ensemble des spécifications établies préalablement à l'implantation.

Deux outils ont été utilisés pour effectuer les simulations du modèle ProDSP. Tout d'abord, on a utilisé ModelSim de la compagnie ModelTech (HDL Simulation Products. Model Technology, Inc.) On a utilisé les version 4.7i, 5.2(a à e), 5.3 et 5.4. De nombreux problèmes ont été rencontrés lors des simulations, et on a dû signaler un grand nombre de bogues à la compagnie. ModelSim est un outil de simulation très puissant, mais dont l'interface-usager laisse à désirer. Ceci est d'ailleurs pire avec la version PC, qui était une conversion directe de la version Unix.

L'autre outil utilisé est Active HDL de la compagnie Aldec (Active-HDL Standard Edition Datasheet. Aldec, Inc.) Les versions utilisées furent les version 3.2, 3.3, 3.6 et 4.1 (beta build 791). L'environnement est nettement plus convivial, notamment parce qu'il permet d'établir une hiérarchie de modules pour la compilation, mais l'engin de simulation souffre de certaines limitations. Encore une fois, on a dû interagir avec la compagnie pour régler certains bogues majeurs qui empêchaient la simulation de notre modèle.

3.5.2 Synthèse

La synthèse est la deuxième étape de développement, souvent effectuée en parallèle avec la simulation, pour s'assurer que le circuit simulé va être synthétisable. Elle consiste à produire un circuit logique équivalent à la description VHDL qui a été analysée et compilée.

L'outil utilisé pour la synthèse était Synplify de la compagnie Synplicity (Product Literature. Synplify, Inc.). On a utilisé toutes les versions à partir de la version 5.0.9 jusqu'à la version 6.0. Il effectue une compilation, synthèse et l'étape de correspondance entre les ressources disponibles sur la technologie ciblée (un FPGA de la famille Virtex de Xilinx dans notre cas) et les ressources synthétisées (*technology mapping*). Il est très simple à utiliser, et les performances de l'outil de synthèse sont excellentes. Toutefois, l'interface-usager laissait encore une fois à désirer (avant la version 6.0).

Un module particulièrement utile de Synplify est le HDL Analyst. Ce module permet de visualiser graphiquement les circuits synthétisés. Ainsi, on peut observer une vue schématique des circuits qui sont générés. La hiérarchie y est représentée, et on peut observer tous les modules et signaux. Ceci est particulièrement utile pour vérifier le résultat de la synthèse et pour observer certaines anomalies dans nos circuits. C'est d'ailleurs ce module qui a été utilisé pour effectuer les comparaisons de circuits synthétisés décrits à la section 3.4.

3.5.3 Problèmes rencontrés

Comme on l'a mentionné dans les sections qui précèdent, on a rencontré un grand nombre de problèmes avec les outils utilisés. En effet, on a très souvent atteint les limites des outils quant au support du langage VHDL. Ainsi, notre description du modèle de processeur ProDSP, avec toute sa configurabilité et ses possibilités d'optimisations, est très complexe. Pas mal plus complexe en fait qu'un modèle de processeur décrit sans ces fonctionnalités. On a crée un modèle qui utilise des constructions plus complexes et élaborées que ce qui est normalement produit en industrie. C'est ce qui a fait que l'on a poussé les outils à leurs limites, et que très souvent on les a brisés. De ce fait, on a eu beaucoup de difficulté à effectuer le développement, avec la qualité des outils actuels.

Chapitre 4

Optimisation du modèle

Ce dernier chapitre présente le travail qui a été réalisé pour automatiser l'optimisation du modèle de processeur en fonction d'une application spécifique. Ce processus correspond en fait à l'étape d'optimisation, qui est réalisée avant la synthèse finale du processeur, dans la méthode dérivation de modèles de processeurs spécifiques à une application, telle que présentée au chapitre 2.

On commencera par présenter les modifications que l'on a faites au modèle de processeur ProDSP présenté au chapitre 3 pour pouvoir accommoder le processus d'optimisation. Plus spécifiquement, on traitera de la configurabilité générale du modèle, des modifications effectuées au chemin de données ainsi qu'au *contrôleur* qui ont été effectuées pour permettre cette configurabilité.

Puis, on présentera l'outil d'optimisation lui-même. On y expliquera les différentes options qui ont été explorées durant la conception de l'outil, et les différents compromis de design. Un exemple d'utilisation, avec l'interface-usager et les différentes options sera présenté.

Finalement, on présentera quelques études de cas qui ont été réalisées lors d'expérimentations avec l'outil. Ainsi, on présentera les différentes applications, ainsi que les résultats de l'optimisation.

4.1 Configurabilité du modèle

La configurabilité du modèle est la capacité de celui-ci de pouvoir être adapté à une architecture particulière. Comme notre modèle devait être conçu pour être utilisé dans un contexte de dérivation de modèles, celui-ci devait être particulièrement configurable. On distingue deux niveaux de configurabilité, soient la paramétrisation et la modularité. Le premier implique que l'on peut adapter les paramètres des différents modules du processeur à une application particulière, alors que le deuxième implique que l'on puisse enlever certains modules sans affecter le reste du fonctionnement du processeur.

Dans notre modèle ProDSP, on dispose d'un très grand nombre d'options de configuration qui permettent de configurer le modèle. La plupart de ces options de configuration impliquent les deux niveaux de configurabilité. En effet, bien souvent on peut déterminer si l'on désire inclure un module, et quels en seront les paramètres. Par exemple, on peut vouloir déterminer si les générateurs d'adresses externes sont utilisés, et quels en seront leurs paramètres (i.e. largeur des adresses, nombre de dimensions etc.) On a donc poussé au maximum la configurabilité de notre modèle de processeur, de façon à pouvoir en tirer le maximum de flexibilité.

4.1.1 Options de configuration

Comme on l'a mentionné précédemment, on dispose d'un très grand nombre d'options de configuration. Le tableau 4.1 présente brièvement la majorité des différentes options, de façon à comprendre la flexibilité de configuration du processeur.

Tableau 4.1 Options de configuration du processeur ProDSP

Contrôleur
Largeur du compteur de programme (PC)
Largeur des compteurs de boucles
Profondeur des piles de sous-routines et de compteurs de boucles
Utilisation des instructions vectorielles
Longueur maximale des instructions vectorielles
Utilisation des NOPs directement encodés dans le microcode
Longueur maximale des NOPs directement encodes
Utilisation des signaux de synchronisation externe
Nombre de signaux de synchronisation externe
Chemin de données
Largeur des données
Nombre d'éléments de calcul (PEs)
Utilisation des registres
Utilisation de la deuxième banque de registres
Profondeur des banques de registres
Utilisation des mémoires locales
Utilisation de la deuxième banque de mémoire
Profondeur des banques de mémoire
Utilisation de l'accumulateur
Profondeur de la pile d'activité

Tableau 4.1 Options de configuration du processeur ProDSP (suite)

Mémoire de constantes
Utilisation de la mémoire de constantes
Utilisation vectorielle de la mémoire de constantes
Profondeur de la mémoire de constantes
Générateurs d'adresses externes
Utilisation du générateur A et/ou du générateur B
Nombre de dimensions
Largeur des adresses externes
Largeur des valeurs de forme
Largeur des valeurs de déplacement
Registres de configuration et d'état
Utilisation des registres de configuration
Utilisation des registres d'état

En plus de ces options, un grand nombre d'options permettent d'indiquer les différents modes d'adressages, les différentes instructions etc. qui vont être utilisées par l'application. Ces options sont utilisées pour éliminer ou adapter la fonctionnalité de certains modules.

4.2 Modifications du modèle

Plusieurs modifications ont dû être réalisées de façon à adapter le modèle pour qu'il puisse être adaptable en fonction des options de configuration. La première modification majeure se rapporte à la façon dont les types et les structures de données sont codées. Ces modifications ont été décrites dans la section 3.3.2. Elles sont importantes, puisqu'elles

permettent d'adapter automatiquement les différents types de signaux en fonction de l'usage qui est fait des différents modes d'adressages, instructions etc.

4.2.1 Modifications du contrôleur

L'ensemble du *Contrôleur* ayant déjà été prévu de façon à être paramétrable, seuls certaines modifications ont dû être réalisées de façon à pouvoir accommoder la modularité. Ainsi, on a dû s'arranger pour pouvoir éliminer certains modules et s'assurer que tous les signaux allaient se transmettre correctement, et que le fonctionnement du *Contrôleur* n'allait pas en être altéré. Par exemple, dans la section du *séquenceur* qui décode les différents champs du microcode, certaines sections deviennent invalides si les modules qu'ils alimentent ne sont plus utilisés. Une autre section qui a été particulièrement modifiée est l'ensemble de générateurs d'adresses pour les instructions vectorielles. Ainsi, si on n'utilise pas un mode d'adressage vectoriel pour une source, on doit pouvoir éliminer ce générateur d'adresses sans causer de problème.

L'ensemble de ces modifications ont été réalisées principalement en utilisant des clauses `if [...] generate` du langage VHDL. Ceci fonctionne bien, mais a le désavantage d'alourdir passablement le code. Ainsi, quelques lignes de code peuvent facilement devenir plusieurs dizaines de lignes de code si on a plusieurs conditions à vérifier pour s'assurer de générer la bonne structure. Ceci a le désavantage de rendre le code plus difficile à vérifier (puisqu'il existe alors plusieurs possibilités d'instanciation) et réduit énormément la lisibilité du code.

4.2.2 Modifications du chemin de données

Une partie des modifications a été réalisée par Fortin (2000) dans le cadre de son projet de maîtrise, et sont décrites dans son mémoire. Les modifications qu'il a réalisées portaient principalement sur la modularité du chemin de données. Ainsi, comme le PE est constitué d'un ensemble de modules fonctionnels essentiellement indépendants les uns des autres, on peut normalement en éliminer un sans affecter le fonctionnement des autres.

De plus, on a procédé à modifier davantage le chemin de données, de façon à exploiter une granularité plus fine de la fonctionnalité. Ainsi, les modifications de Fortin (2000) permettaient d'éliminer des modules ou des sous-modules entiers. Toutefois, on désire pouvoir éliminer seulement une partie de la fonctionnalité de la plupart des modules entiers. Ceci est particulièrement évident dans le cas des modules permettant d'exécuter un grand nombre d'opérations différentes : si on n'utilise qu'un sous-ensemble de toutes les opérations disponibles, on peut donc en éliminer un certain nombre. Toutefois, ceci peut amener certaines complications, notamment dans la complexité du code. Ainsi, on a à la figure 4.1 un exemple de code sans modification pour exploiter une granularité plus fine.

```
Sub_s   <= '1' when Sub_Enable_p else '0';

-- Adjust B in the case of a substraction
Inv_s   <= ( Inv_s'range => Sub_s );
B_Mod_s <= (B_p xor Inv_s) + Sub_s;

-- The carry is either 0 (no carry in), 0001 (carry in, add) or 1111 = -1 (carry in, sub)
Carry_Borrow_s <= ( PE_Data_Width_c downto 1 => Carry_In_p and Sub_s ) & Carry_In_p;

-- Perform the addition
Sum_Var_s      <= A_p + B_Mod_s + Carry_Borrow_s;

-- Assign outputs
Sum_p          <= Sum_Var_s(PE_Data_Width_c-1 downto 0);
Carry_Out_p <= Sum_Var_s(PE_Data_Width_c);
```

Figure 4.1 Exemple de code sans modification pour granularité fine

À la figure 4.2, on a le même exemple de code, mais cette fois avec les modifications pour exploiter une granularité plus fine. Comme on peut le voir, cet exemple, bien que simple (c'est le corps d'un additionneur/soustracteur signé) devient passablement compliqué lorsque l'on tente d'exploiter une granularité plus fine. Dans cet exemple, on n'avait pourtant que quatre instructions de l'application qui pouvaient être utilisées et exécutées par ce module.

```
Sub_s   <= '1' when Sub_Enable_p else '0';

Sub: if PE_Instruction_SUB or PE_Instruction_SUBB generate
begin
        -- Adjust B in the case of a substraction
        Inv_s   <= ( Inv_s'range => Sub_s );
        B_Mod_s <= (B_p xor Inv_s) + Sub_s;
end generate;

Add: if not( PE_Instruction_SUB or PE_Instruction_SUBB ) generate
        B_Mod_s <= B_p;
end generate;

Carry: If PE_Instruction_ADDC or PE_Instruction_SUBB generate
        -- The carry is either 0 (no carry in), 0001 (carry in, add) or 1111 = -1
        -- (carry in, sub)
        Carry_Borrow_s <= ( PE_Data_Width_c downto 1 => Carry_In_p and Sub_s ) &
                          Carry_In_p;
end generate;

No_Carry: If not( PE_Instruction_ADDC or PE_Instruction_SUBB ) generate
        Carry_Borrow_s <= ( others => '0' );
end generate;

-- Perform the addition
Sum_Var_s       <= A_p + B_Mod_s + Carry_Borrow_s;

-- Assign outputs
Sum_p           <= Sum_Var_s(PE_Data_Width_c-1 downto 0);
Carry_Out_p <= Sum_Var_s(PE_Data_Width_c);
```

Figure 4.2 Exemple de code avec modifications pour granularité fine

Par ailleurs, il arrive souvent que la granularité fine ne puisse être exploitée en n'utilisant que les constructions qui sont permises par le VHDL. Ainsi, on devra avoir recours à d'autres moyens pour optimiser les structures de différents modules. Ceci est discuté à la section 4.3.2.2.

4.3 Outil d'optimisation

L'outil d'optimisation que l'on décrit dans cette section correspond à l'outil qui effectue l'étape supplémentaire dans le flot d'implantation décrite à la section 2.6. Celui-ci doit effectuer trois opérations distinctes : l'analyse du microcode de l'application, la

génération du modèle optimisé et finalement le recodage du microcode pour refléter le nouveau modèle optimisé.

4.3.1 Analyse du microcode

La première étape du processus d'optimisation correspond à l'analyse du microcode de l'application pour laquelle on désire optimiser le modèle de microcode. Cette étape est essentiellement la même que celle réalisée par l'outil d'analyse de microcode décrit à la section 2.4. Toutefois, on a reprogrammé une nouvelle version de cet analyseur de façon à l'adapter uniquement à l'architecture ProDSP, puisque seule celle-ci sera visée par l'outil d'optimisation. Ceci nous a permis de simplifier l'écriture du code, vu que seule cette architecture devait être visée.

4.3.2 Génération du modèle

La deuxième étape d'optimisation est la plus importante : c'est celle qui permet de générer le modèle VHDL du processeur optimisé pour l'application. Ceci est réalisé en fonction des résultats d'analyse du microcode qui a été effectuée à l'étape précédente.

Il y a deux façons de générer le modèle VHDL dédié à une application, chacune avec ses avantages et ses inconvénients. On va tenter de détailler ceux-ci et d'expliquer les choix qui ont été faits dans le cadre de ce projet.

4.3.2.1 Constructions en langage HDL

La première façon de générer le modèle est d'utiliser les capacités du langage de description de matériel pour produire des structures génériques. Dans le cas qui nous intéresse, le VHDL permet plusieurs constructions possibles, notamment avec les `for [...] generate` et `if [...] generate`. Ainsi, on peut construire des modules fortement génériques et configurables en n'utilisant que les structures du langage. Ainsi, pour configurer le modèle, on n'a qu'à décrire quelques constantes qui vont être utilisées dans les paramètres des clauses `if` et `for`.

Ceci a plusieurs avantages : tout d'abord, comme le code couvrant toutes les possibilités de configuration est visible, il est très facile d'en vérifier la syntaxe. De plus, ce code est normalement portable, puisqu'il n'utilise que des constructions VHDL standards; ainsi, n'importe quel outil de synthèse ou de simulation devrait pouvoir le lire et le traiter. Finalement, celui-ci est complet en soit : on n'a pas réellement besoin d'outil ou de support extérieur pour avoir un modèle entièrement configurable et paramétrable.

Toutefois, ceci a également plusieurs désavantages importants. Tout d'abord, la complexité du code que l'on doit produire est nettement supérieure pour un modèle générique que pour un modèle qui ne l'est pas (ou pas entièrement). L'exemple donné aux figures figure 4.1 et figure 4.2 illustre bien la complexité d'écriture d'un module entièrement générique et optimisable. De plus, il arrive que dans certains cas, la

complexité soit particulièrement importante, comme par exemple dans le cas du multiplicateur pipeliné générique décrit dans Fortin (2000). Dans cet exemple, le code pour un multiplicateur pipeliné fixe est passé de quelques dizaines de lignes de VHDL, à 500 lignes (dont certaines de 300 caractères de large) pour une version générique. Dans des cas comme ceux-ci, il vaut mieux voir si un générateur de code (discuté plus loin) ne serait pas avantageux, plutôt que de tenter de construire des modèles excessivement complexes en n'utilisant que les constructions possibles pour le VHDL. Par ailleurs, la complexité du code fait en sorte que celui-ci sera moins lisible, et plus difficile à comprendre. De ce fait, il sera particulièrement difficile à déboguer.

Finalement, le code produit sera d'une telle complexité que bien des outils ne pourront le supporter. Tel que présenté à la section 3.5.3, lors du développement du modèle avec des constructions génériques complexes, on a rencontré un très grand nombre de problèmes avec les outils logiciels. En fait, aucun de ceux-ci n'a été capable de supporter le modèle sans avoir recours à des révisions, soit du modèle, soit des outils eux-mêmes.

En fait, il arrive bien souvent que l'effort nécessaire pour créer un module générique soit beaucoup plus important que pour la création d'un module dédié, de la même façon que le design et l'implantation d'un module réutilisable demande plus d'effort qu'un module utilisé une seule fois.

4.3.2.2 Générateur de code HDL

La seconde option pour générer le modèle consiste à utiliser un outil externe pour produire du code HDL dédié. Ainsi, un générateur de code va produire le code adapté pour notre configuration.

Ceci a plusieurs avantages : tout d'abord, le code généré va être beaucoup plus simple que du code VHDL générique, parce que celui-ci est dédié et ne doit pas permettre une multitude de possibilités. Ceci est nettement avantageux dans les cas où l'on a des constructions complexes, comme dans le cas de structures répétées selon un algorithme précis. Un bon exemple de ceci est le multiplicateur générique décrit par Fortin (2000) : celui-ci est composé d'un certain nombre de modules qui sont répétés dans une structure en arbre selon un algorithme précis. Bien que l'algorithme en soi ne soit pas excessivement complexe, il est particulièrement complexe à implanter en VHDL structurel. Ceci ne serait pas le cas avec un générateur de code, implanté dans un autre langage et dont la sortie serait du code VHDL.

De plus, on va pouvoir effectuer certaines modifications au code qui ne sont pas réellement possibles en n'utilisant que des constructions HDL normales. Ainsi, pour certains cas, il ne sera pas possible de construire des structures génériques; il faudra donc avoir recours à un outil externe pour optimiser certaines constructions. C'est le cas notamment avec des multiplexeurs dont on désire modifier le nombre d'entrées lors de la compilation.

Toutefois, il y a également des désavantages à utiliser un générateur de code : tout d'abord, celui-ci n'est pas réellement portable. En effet, on n'a pas que la description en langage HDL, mais plutôt un outil qui va générer celui-ci, en tout ou en partie. Cet outil n'est possiblement pas facilement portable. De plus, il peut être ardu d'effectuer des modifications au modèle de base du processeur une fois que celui-ci est arrangé pour être produit par le générateur. Ainsi, on devra modifier le code du générateur qui va produire le code HDL.

Aussi, il peut être relativement difficile de produire du code HDL avec une syntaxe correcte, pour toutes les possibilités. Ainsi, on pourrait se retrouver avec des cas limites qui ne génèrent pas une syntaxe correcte. Les erreurs seront alors détectées à la compilation du code. De ce fait, il peut être excessivement difficile de valider le générateur de code si celui-ci peut produire un grand nombre de configurations différentes.

4.3.2.3 Générateur de modèle ProDSP

Dans le cadre du projet ProDSP, on a utilisé les deux approches de façon à contourner certains problèmes. Toutefois, on a tenté au maximum de produire un code optimisable et configurable en VHDL, de façon à minimiser le travail requis par le générateur de code VHDL. Ainsi, la majorité de la configuration se fait par le fichier de configuration VHDL. Celui-ci est produit par le générateur de code de l'optimiseur. Il contient les

paramètres de configuration des différents modules. C'est à partir de celui-ci que tous les types de signaux seront construits, et que les différents modules seront structurés. Le processus de configuration du processeur est illustré à la figure 3.10.

On a également dû produire un générateur de code pour certains modules qui ne pouvaient être construits entièrement avec le langage VHDL. Ceux-ci sont notamment les ROMs de décodage, les modules fonctionnels de l'unité arithmétique et logique ainsi que les décodeurs de bus. On a ainsi minimisé le nombre de modules qui doivent être générés par l'optimiseur.

4.3.3 Recodage du microcode

La dernière étape du processus d'optimisation que doit réaliser l'optimiseur est l'adaptation du microcode à la nouvelle architecture dédiée. En effet, le microcode original, qui devait être exécuté sur un processeur complet, ne peut fort probablement plus être exécuté sur le modèle optimisé généré à l'étape précédente.

Ainsi, pour chaque instruction, il faut recoder les différents champs en fonction des nouveaux codages des signaux. Il faut également éliminer les champs qui ne sont plus utilisés et ajuster la taille des constantes et adresses. Ceci va donc nous donner un microcode compacté et adapté à la nouvelle architecture.

4.4 Interface-usager et exemple d'utilisation

L'optimiseur réalisé dans le cadre de ce projet a été implanté en langage C++ sur une plate-forme Windows NT. L'outil est relativement simple à utiliser : une fois le microcode produit, on exécute l'optimiseur. On sélectionne alors le fichier contenant le microcode ainsi que les options que l'on désire utiliser. Finalement, on n'a qu'à accepter le tout et, si la configuration est valide, le modèle optimisé va être produit et le microcode recodé en fonction du nouveau modèle de processeur.

La figure 4.3 illustre l'interface-usager de l'optimiseur. Comme on peut le voir, on peut modifier la grande majorité des paramètres que l'optimiseur ne peut pas déterminer à partir du microcode de l'application.

Pour illustrer le processus d'utilisation, on va détailler l'usage que ferait un utilisateur de l'outil pour une application. Tout d'abord, l'application est écrite, soit en assembleur, soit dans un langage compilé. Si un simulateur de jeu d'instructions (*instruction set simulator*) est disponible, celui-ci peut être utilisé pour valider le code. Comme celui-ci n'existe pas pour l'architecture ProDSP, on devra simuler à l'aide d'un simulateur VHDL. Puis, on procède à obtenir le microcode, soit en compilant et assemblant le code, soit en assemblant le code tout simplement. Dans notre cas, il n'existe pas d'assembleur pour l'architecture ProDSP alors on fait la transcription de l'assembleur au microcode manuellement.

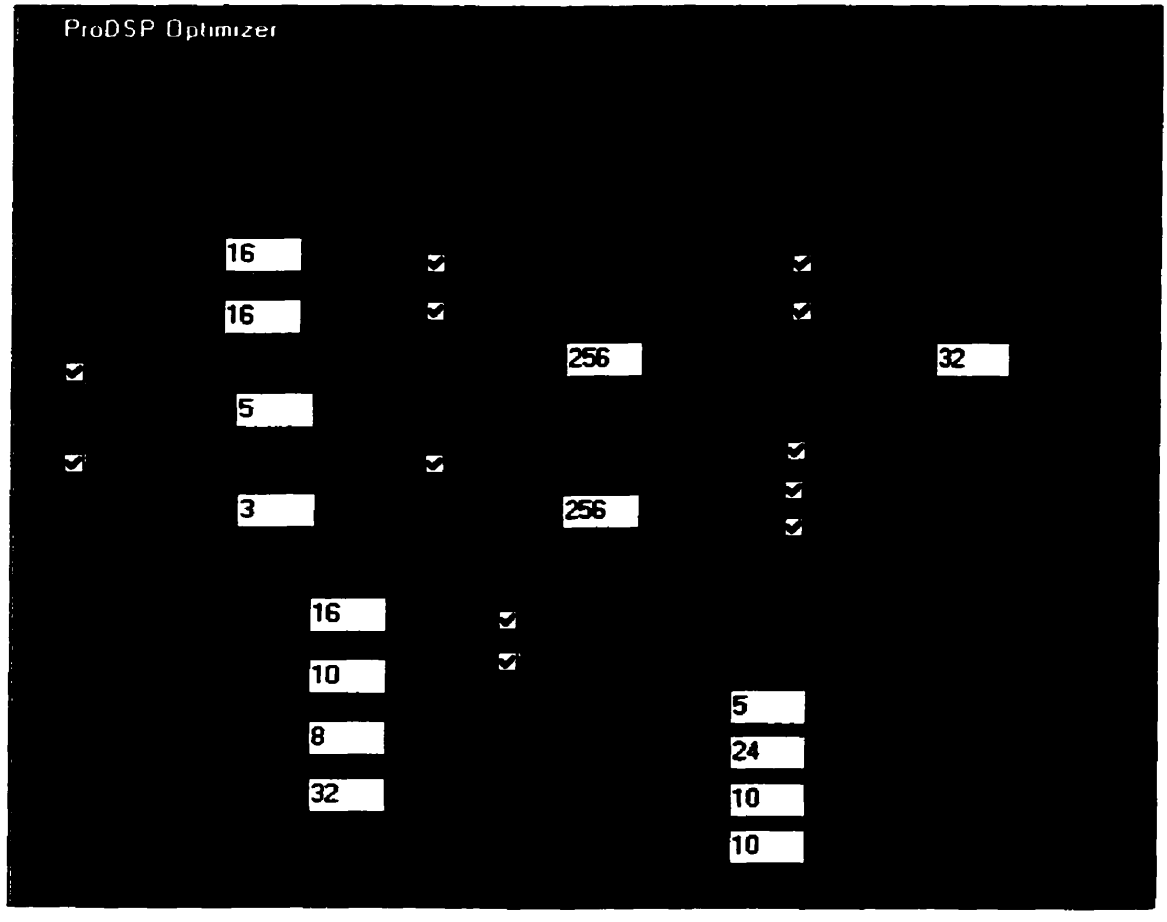


Figure 4.3 Interface-usager de l'optimiseur

Puis, on exécute l'optimiseur. On sélectionne les options appropriées pour l'application et on part l'optimisation. En sortie, l'outil nous donne un modèle VHDL synthétisable du processeur optimisé pour l'application, ainsi que le microcode de l'application recodé en fonction de la nouvelle architecture.

4.5 Études de cas

Quelques études de cas ont été réalisées de façon à déterminer l'efficacité de la méthode de dérivation de modèles de processeurs embarqués dédiés, ainsi que la capacité du modèle de l'architecture ProDSP à être optimisé.

4.5.1 Applications de test

Toutefois, on n'a pu réaliser un grand nombre d'expérimentations pour différentes raisons. Tout d'abord, il n'existe aucune application déjà écrite pour l'architecture ProDSP. Comme celle-ci est assez complexe et différente de ce que l'on retrouve comme type de processeur sur le marché, il est particulièrement ardu d'écrire des applications pour celle-ci sans outils appropriés. De plus, il n'existe aucun outil de support logiciel pour l'écriture d'application. Ainsi, on ne dispose ni d'un débogueur, ni de compilateur, ni d'assembleur. Il faut donc écrire le microcode à la main, ce qui n'est pas nécessairement chose facile, vue la largeur du microcode de l'architecture ProDSP (122 bits). Par ailleurs, des problèmes majeurs avec les outils de synthèse et de simulation dus à la complexité du code VHDL du modèle ont fait en sorte que le temps consacré à la conception d'applications et à l'utilisation de l'outil a été passablement réduit.

On a néanmoins réalisé quatre applications, et on a utilisé l'outil pour générer une version optimisée du modèle, de façon à évaluer la performance de la méthode. La première application est un simple filtre à réponse impulsionnelle finie (FIR : *Finite Impulse*

Response) qui utilise la chaîne d'accumulation décrite à la section 3.2.5 pour simplifier l'implantation.

Les autres applications sont des adaptations des applications utilisées sur le processeur PULSE (section 2.5.1), soient le détecteur de contours (EDGE), le convolueur bidimensionnel 3×3 (CONV3X3) et l'analyse d'image en histogramme (HISTO).

4.5.2 Configurations de test

Le tableau 4.1 indique la configuration qui a été utilisée pour chacun des paramètres de configuration lors de l'optimisation du modèle en fonction des différentes applications. Ces paramètres correspondent à ce que l'usager doit spécifier à l'aide de l'interface usager de l'outil d'optimisation. La première colonne des applications indique les paramètres pour la synthèse d'un modèle ProDSP complet.

Tableau 4.2 Configuration du modèle ProDSP pour les différentes applications

Paramètre	ProDSP Complet	FIR	EDGE	CONV3X3	HISTO
Nombre de PEs	4	4	4	4	4
Largeur des données	16	16	16	16	16
Vecteurs utilisés	Oui	Non	Non	Non	Non
Largeur des vecteurs (bits)	5	-	-	-	-
NOPs encodés utilisés	Oui	Non	Non	Non	Oui
Largeur des NOPs (bits)	3	-	-	-	3
Mémoire des PEs utilisée	Oui	Non	Oui	Oui	Oui
Deux banques de mémoire	Oui	-	Non	Non	Non
Profondeur des mémoires	256	-	256	256	256
Registres des PEs utilisés	Oui	Oui	Oui	Non	Non
Deux banques de registres	Oui	Non	Oui	-	-
Profondeur des registres	32	2	8	-	-
Mémoire de constantes utilisée	Oui	Non	Non	Oui	Non
Profondeur des constantes	256	-	-	2	-
Largeur du PC	16	8	8	8	8
Largeur des boucles	10	10	12	5	8
Profondeur des piles	8	2	2	8	5
Profondeur de la pile d'activité	32	2	2	2	2
Générateur d'adresses externes A	Oui	Non	Oui	Oui	Oui
Générateur d'adresses externes B	Oui	Non	Oui	Oui	Oui
Nombre de dimensions	5	-	5	5	5
Largeur des adresses	24	-	10	10	10
Largeur des formes	10	-	10	10	10
Largeur des déplacements	10	-	10	10	10
Signaux de synchronisation	Oui	Non	Non	Non	Non
Registres de configuration	Oui	Non	Non	Non	Non
Registres d'état	Oui	Non	Non	Non	Non

4.5.3 Synthèse des modèles de test

On a effectué la synthèse des modèles optimisés à l'aide de l'outil de synthèse Synplify, version 6.0.0 (Product Literature. Synplify, Inc.). La plate-forme ciblée était un FPGA de la famille Virtex-E de la compagnie Xilinx (Virtex product specifications. Xilinx, Inc.)

Plus spécifiquement, on a visé un FPGA comportant environ 1 million de portes logiques équivalentes (selon les spécifications de la compagnie), le XCV1000E-6.

Pour vérifier la performance des optimisations, on a vérifié un certain nombre de métriques, ayant comme point commun qu'elles se rapportent à la grosseur du circuit résultant. Ainsi, on a utilisé comme principe qu'un circuit plus simple devrait en principe être plus performant et devrait également dissiper moins de puissance. On a donc mesuré la taille des circuits, notamment parce que c'était la procédure la plus simple à réaliser. Si on avait voulu avoir des mesures de performances, il aurait fallu effectuer le placement et le routage des circuits, et ceci aurait demande un temps considérable, puisque les circuits sont quand même complexes, et que les outils sont très exigeants. Le manque de temps et de ressources matérielles nous ont empêchés d'obtenir ces estimés de performance.

4.5.4 Résultats des optimisations

Les résultats de synthèse nous ont permis d'obtenir des mesures quantitatives des optimisations effectuées sur le modèle de processeur ProDSP en fonction des différentes applications. On a mesuré différents paramètres pour chacun des modèles de processeurs synthétisés. Ainsi, on peut diviser en trois catégories ces métriques : la taille du microcode, le nombre de ressources générales, ainsi que les ressources mémoires utilisées. Dans le cas des ressources générales, on retrouve le nombre de LUTs (*Look Up Tables*) soit l'unité de base pour mesurer la complexité combinatoire d'un circuit implanté dans la technologie FPGA Xilinx. On a également le nombre de bits de

registres, qui permet de mesurer la demande pour les éléments séquentiels. Finalement, on a également le nombre de tampons à trois états, qui permet d'évaluer la demande sur les ressources de routage nécessaires à la réalisation du circuit.

Pour ce qui est des ressources mémoires, on a trois types de mémoires : deux à simple port et une à double port. Ces ressources permettent d'évaluer les besoins en bloc de mémoire pour un circuit en particulier.

Le Tableau 4.3 présente les résultats pour les différentes métriques utilisées. La deuxième colonne correspond à une configuration de modèle ProDSP complet. C'est ce modèle qui va servir de référence pour évaluer les optimisations.

Tableau 4.3 Résultats de synthèse pour les différentes applications

Métrique	ProDSP Complet	FIR	EDGE	CONV3X3	HISTO
Largeur du microcode (bits)	122	25 (20%)	50 (41%)	55 (45%)	35 (29%)
Nombre de LUTs	11 471	986 (9%)	3342 (29%)	3127 (27%)	2451 (21%)
Nombre de bits de registres	4344	441 (10%)	1329 (31%)	1897 (44%)	1111 (26%)
Nombre de tampons à trois états	7240	480 (7%)	2020 (28%)	3132 (43%)	1188 (16%)
Mémoire simple port (16×1)	36	18 (50%)	20 (56%)	29 (81%)	16 (44%)
Mémoire simple port (32×1)	1152	0 (0%)	640 (56%)	512 (44%)	512 (44%)
Mémoire double port (16×1)	256	0 (0%)	128 (50%)	0 (0%)	0 (0%)

Les colonnes suivantes présentent le nombre d'éléments nécessaires pour chaque catégorie. Le nombre entre parenthèses correspond au pourcentage de ressources utilisées comparativement au modèle complet. Comme on peut le voir dans le tableau, les optimisations sont relativement importantes. Ainsi, la largeur de microcode est réduite à environ le tiers de la taille originale en moyenne. Pour ce qui est des ressources générales, l'optimisation va de plus de 90% à environ 50%, dépendamment de la complexité de l'application. Ceci est particulièrement important du côté du nombre de LUTs, qui est la métrique la plus importante, puisqu'elle correspond à la ressource la plus en demande pour cette famille de FPGAs.

Pour ce qui est des mémoires, celles-ci sont moins éliminées que la moyenne des autres composantes. Ainsi, seules les mémoires à double port sont vraiment radicalement éliminées. Ceci est dû au fait que les registres sont rarement utilisés dans les applications de test que l'on a choisi.

Finalement, on peut conclure que ces optimisations sont relativement importantes pour le jeu d'application de test dont on disposait. Toutefois, il serait intéressant d'évaluer la performance de ces applications lorsque l'on désire utiliser des applications de plus en plus complexes. Toutefois, ces résultats sont encourageants, et permettent de déterminer que l'on pourrait obtenir une grande performance des processeurs optimisés, puisque comme on diminue la complexité totale des circuits, on va pouvoir placer davantage d'éléments de calcul pour un même FPGA. Par exemple, plutôt que d'implanter un filtre

FIR à 4 points sur un modèle ProDSP complet, on pourrait implanter un filtre FIR à 16 points en utilisant le même nombre de ressources.

Conclusion

Le but du travail exposé dans ce mémoire était le développement d'un modèle de processeur de traitement de signal qui pouvait être optimisé en fonction de l'application qui devait être exécutée sur celui-ci. Une méthodologie de dérivation de processeurs en fonction de l'application a également dû être développée en parallèle, de façon à pouvoir exploiter la configurabilité du modèle de processeur. Le point de départ du projet était un modèle de processeur de traitement de signal parallèle, originalement ciblé vers une implantation ASIC. Celui-ci n'était pas facilement utilisable pour un processus de dérivation et c'est pourquoi nous avons dû le retravailler. Dans le présent mémoire, on a exploré le travail qui a été nécessaire au développement de la méthode de dérivation, ainsi que le développement du modèle de processeur qui permettait d'exploiter cette méthode de dérivation.

Le premier chapitre constitue une revue des travaux qui ont été réalisés dans le domaine de la dérivation et de la synthèse spécifique de processeurs, ainsi que des processeurs configurables disponibles commercialement. Ainsi, il existe présentement sur le marché deux sociétés, qui offrent des modèles de processeurs configurables, soient Tensilica et ARC. Ces sociétés permettent au concepteur d'adapter l'architecture de leurs processeurs de façon à l'optimiser en fonction de l'application ciblée. Chacune de ces sociétés offre une suite d'outils de développement complets, dont ceux pour le développement du logiciel qui va être exécuté sur le processeur. Ceux-ci s'adaptent aux différentes

configurations possibles, ainsi qu'aux instructions que le concepteur peut rajouter à son architecture de processeur. De plus, on a exploré l'utilisation de plate-formes configurables pour la conception de systèmes embarqués. La compagnie Improv Systems propose une telle plate-forme, ainsi que les outils de développement pour concevoir des systèmes. L'environnement de développement est particulièrement intéressant, puisqu'il permet de décrire l'ensemble du système en langage Java. Un compilateur permet ensuite d'effectuer le partitionnement entre différents processeurs et s'occupe de gérer les communications. Finalement, on a étudié deux méthodologies de dérivation de processeurs. La première permet de réduire la complexité de structures telles que des processeurs de façon à pouvoir implanter plusieurs instances de chaque structure, pour ainsi obtenir de la redondance. Ceci est utile pour les architectures qui doivent être tolérantes aux fautes. La deuxième méthode se base sur des résultats d'émulations de circuits pour dériver un circuit plus simple. Ainsi, on a un circuit complet que l'on valide sur un émulateur matériel. Puis, une fois que le circuit est fonctionnel, on extrait les composantes qui sont vraiment utilisées pour l'application et on synthétise ainsi le circuit minimal pour réaliser l'application. C'est à partir de ces résultats que l'on a basé le point de départ du projet décrit dans ce mémoire.

Le deuxième chapitre présente la méthodologie de dérivation qui a été développée au court de ce projet. À partir des flots de conceptions actuels de systèmes embarqués, on a exposé les problèmes reliés à une implantation directe des composantes disponibles sous

forme de blocs de propriété intellectuelle. Ainsi, on effectue un gaspillage de ressources, puisque les circuits ne sont pas optimisés en fonction de l'application qui est ciblée. On a également exposé les trois éléments principaux qui ont motivé le développement de cette méthode de dérivation : le temps de développement, le temps de validation et la simplicité finale des circuits. Par la suite, on a présenté un outil d'analyse de microcode qui a permis de valider l'approche que l'on désirait employer. En effet, on a pu démontrer par une étude de cas que dans un grand nombre d'applications, plusieurs composantes d'un processeur ne sont pas utilisées, et peuvent donc être supprimées ou optimisées. On a ensuite présenté des essais d'optimisations simples, réalisées manuellement. Celles-ci ont montré que des optimisations automatiques, plus complexes, pourraient donner de très bons résultats. Finalement, on a décrit la méthode proprement dite, qui est en fait une étape d'optimisation avant synthèse, à partir de l'information contenue dans le microcode. Le résultat est un modèle de processeur optimisé et une application en microcode recodé en fonction du modèle de processeur. On a décrit également les différentes optimisations que l'on peut réaliser.

Le troisième chapitre traite du modèle de processeur de traitement de signaux ProDSP qui a été développé pour implanter la méthode de dérivation décrite au chapitre précédent. Comme un aperçu des différentes composantes qui forment le processeur est nécessaire à la compréhension du processus d'optimisation, celles-ci sont décrites sommairement. Ainsi, on a commencé par décrire l'architecture PULSE, à partir de laquelle a été conçue l'architecture ProDSP. Puis, on a décrit les différentes composantes

qui forment le processeur. On a également discuté de la méthodologie et du style de codage, puisque ceux-ci sont fort importants dans le cadre d'un processeur fortement configurable et optimisable comme celui que l'on développait. Une brève section traitant de la validation de l'ensemble du processeur a été présentée puisque ceci est important, quoique nos méthodes étaient relativement simples et tout à fait standards. Finalement, on a discuté des outils que l'on a utilisés au cours du développement, ainsi que des nombreux problèmes auxquels on a dû faire face lors du développement d'un tel processeur. Ainsi, avec une façon de décrire le processeur d'une complexité supérieure à la moyenne processeur, on a souvent rencontré des difficultés avec les outils.

Le quatrième et dernier chapitre présente le travail qui a été réalisé pour automatiser l'optimisation du modèle de processeur en fonction d'une application spécifique. On a commencé par décrire les différentes modifications que l'on a dû faire au modèle de processeur que l'on a décrit au chapitre précédent pour pouvoir réaliser les optimisations. Ainsi, on a discuté de la configurabilité générale du modèle et des modifications réalisées au *contrôleur* et au chemin de données. Puis, on a présenté l'outil d'optimisation en y expliquant les différentes options qui ont été explorées durant la conception de l'outil. On a par la suite détaillé un exemple d'utilisation avec l'interface-usager et les différentes options. Finalement, on a terminé ce chapitre en présentant des études de cas qui ont été réalisées lors des expérimentations. Ainsi, on a pu montrer que les optimisations étaient tout à fait valables pour les applications que l'on ciblait.

Toutefois, il serait maintenant intéressant de vérifier si l'ensemble du processus de dérivation est encore valable dans les cas où l'on vise des applications nettement plus complexes que celles qui étaient ciblées dans nos expérimentations. En effet, on a surtout visé des applications de traitement d'images, avec un processeur de type DSP à architecture parallèle SIMD. Il se peut que le processus de dérivation ne soit pas aussi performant pour d'autres types d'architectures de processeurs, ou encore des applications d'un autre domaine.

Par ailleurs, il faut également vérifier si le temps nécessaire au développement d'un processeur fortement configurable et optimisable est justifié. En effet, le temps de développement de composants génériques est habituellement plus élevé. Cette augmentation de complexité peut, dans certains cas, devenir assez importante. Dans le cas de composants aussi génériques et configurables, le temps nécessaire est encore plus grand. Ceci est d'ailleurs accentué par le fait que les outils ne sont habituellement pas utilisés pour développer des structures aussi génériques et configurables, et ceux-ci brisent souvent.

En conclusion, ce mémoire a montré les expérimentations qui ont été réalisées dans le cadre du développement d'une méthodologie de dérivation de processeurs. On a montré que le processus de dérivation pouvait donner d'excellent résultats d'optimisations dans un domaine d'application. Ceci est particulièrement intéressant dans le contexte de

développement actuel, ou on tente d'effectuer le plus de réutilisation possible de composantes.

Références

1. ARC Cores Ltd. (Page consultée le 7 juillet 2000) *Site de la compagnie Arc Cores Ltd.*, [En ligne]. http://www.arccores.com

2. ASHENDEN, P.J. (1996). *The Designer's Guide to VHDL*, Morgan Kaufmann Publishers, Inc., San Francisco, 688 p.

3. BÉLANGER, N. ANTAKI, B. et SAVARIA, Y. (1997). *An Algorithm for Fast Array Transfers*, High-Performance Computing Symposium, Winnipeg, July 1997, pp. 117-126.

4. BURSKY, D. "Tool Suite Enables Designers To Craft Customized Embedded Processors". *Electronic Design*, volume 47, numéro 3, 8 février 1999.

5. BONNELLO, C. et KHOUMSI, K. (1997). *PULSE Technical Report : PULSE Instruction Set V1.3*, Document interne du Projet PULSE, Groupe de Recherche en Microélectronique de l'École Polytechnique de Montréal, 313 p.

6. FORTIN, A. (2000). *Paramétrisation et reconfiguration automatique du chemin de données d'un processeur DSP synthétisable, ciblé vers des applications spécifiques*, Mémoire de maîtrise, École Polytechnique de Montréal, Canada.

7. KEATING, M. et BRICAUD, P. (1999). *Reuse Methodology Manual for System-On-A-Chip Designs 2nd edition*, Kluwer Academic Publishers, Dordrecht, Pays-Bas, 312 p.

8. KRALJIC, I.C., QUÉNOT, G.M., ZAVIDOVIQUE, B. (1996). From Real-Time Emulation to ASIC Integration for Image Processing Applications. *IEEE Transactions on VLSI Systems*, Septembre 1996, pp.391-404.

9. The Jazz PSA Platform. Improv Systems Inc. (Page consultée le 7 juillet 2000). *Site de la compagnie Improv Systems, Inc.* [En ligne]. http://www.improvsys.com/Products/Jazz/JazzPSAPlatform.PDF

10. LEVY, M. (1999). "Customized processors : have it your way". *EDN*, 7 Janvier 1999, pp. 97-104.

11. MARRIOTT, P. (1996). *PULSE Technical Report : V1 Processing Element and Chip Architecture*, Document interne du Projet PULSE, Groupe de Recherche en Microélectronique de l'École Polytechnique de Montréal, 37 p.

12. MARRIOTT, P. (1997). *PULSE Technical Report : V2 Chip Specification*, Document interne du Projet PULSE, Groupe de Recherche en Microélectronique de l'École Polytechnique de Montréal, 22 p.

13. MARRIOTT, P., KRALJIC, I.C., et SAVARIA, Y. (1998). Parallel Ultra Large Scale Engine SIMD Architecture For Real-Time Digital Signal Processing Applications. *International Conference on Computer Design*, Austin 1998, 482-487.

14. MEDNICK, E., LIND, C. et HOOKER, F. (2000). *Verification of Lexra Processor Cores*, DesignCon 2000, Santa Clara.

15. HDL Simulation Products. Model Technology, Inc. (Page consultée le 6 juin 2000). *Site de la compagnie Model Technology, Inc.* [En ligne]. http://www.model.com/products/index.html

16. PFLANZ, M. et VIERHAUS, H.T. (1998). "Generating Reliable Embedded Processors". *IEEE Micro*, Septembre-Octobre 1998, pp.33-41

17. Product Literature. Synplicity, Inc. (Page consultée le 6 juin 2000). *Site de la compagnie Synplicity, Inc.* [En ligne]. http://www.synplicity.com/literature/index.html

18. SCHNEIER, B. (1995). *Applied Cryptography : Protocols, Algorithms, and Source Code in C, 2nd Edition* , Wiley & Sons, New York, 784 p.

19. SEMATECH (1999), *International Technology Roadmap for Semiconductors 1999 Edition*, (Page consultée le 11 juillet 2000) [En ligne]. http://www.itrs.net/1999_SIA_Roadmap/Home.htm

20. Triscend E5 Configurable System-on-chip. Triscend, Inc. (Page consultée le 9 juin 2000) *Site de la compagnie Triscend Corporation.* [En ligne]. http://www.triscend.com/products/E5_Prod_Brief_Sep99.pdf

21. Xtensa Application Specific Microprocessor Solutions Overview Handbook. Tensilica Inc. (Page consultée le 9 juin 2000) *Site de la compagnie Tensilica, Inc.* [En ligne]. http://www.tensilica.com/dl/handbook.pdf

22. Texas Instruments, Inc (1991). *TMS320C4x User`s Guide.*

23. Virtex product specifications. Xilinx, Inc. (Page consultée le 8 juin 2000). *Site de la compagnie Xilinx, Inc.* [En ligne]. http://www.xilinx.com/partinfo/ds003.pdf