Etude d’un coeur de processeur pour l’arithmétique exacte

V. Coissard

▶ To cite this version:

V. Coissard. Etude d’un coeur de processeur pour l’arithmétique exacte. Micro et nanotechnologies/Microélectronique. Institut National Polytechnique de Grenoble - INPG, 2000. Français. NNT : . tel-00002997

HAL Id: tel-00002997
https://theses.hal.science/tel-00002997

Submitted on 13 Jun 2003

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

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

THÈSE

présentée par

Vincent COISSARD

pour obtenir le grade de **DOCTEUR**

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(arrêté ministériel du 30 mars 1992)

Spécialité : **Microélectronique**

ETUDE D'UN CŒUR DE PROCESSEUR POUR L'ARITHMETIQUE EXACTE

Date de Soutenance : 2 septembre 1998

Composition du jury :

Messieurs :	Guy MAZARE	*Président*
	Daniel ETIEMBLE	*Rapporteur*
	Habib MEHREZ	*Rapporteur*
	Daniel AUVERGNE	*Examinateur*
	Marc RENAUDIN	*Examinateur*
	Alain GUYOT	*Examinateur*

Thèse préparée au sein du Laboratoire TIMA-INPG à Grenoble

Résumé

L'arithmétique virgule flottante utilisée en machine pour le calcul scientifique introduit des erreurs dans le résultat des opérations. Le calcul sur ordinateur porte en effet sur des opérandes qui possèdent un nombre limité de chiffres significatifs, lesquels ne représentent qu'une approximation de la valeur exacte. Au fur et à mesure du déroulement des programmes, on assiste à une dégradation progressive de la précision des nombres manipulés. Ces accumulations d'erreurs peuvent conduire à des résultats invalides sans que l'utilisateur en soit averti. Parmi les solutions développées pour maîtriser les erreurs du calcul en machine, seule l'utilisation d'une arithmétique exacte conduit à un résultat dont on est sûr qu'il est correct. Malheureusement cette solution est obtenue par logiciel au prix d'un temps de calcul extrêmement long. Une des principales raisons de la lenteur de ce type de logiciel provient du fait qu'ils s'exécutent sur des processeurs qui ne disposent pas d'une arithmétique adaptée au calcul exact. Il faut donc faire une émulation de chaque opération élémentaire de l'arithmétique exacte en faisant appel à des routines logicielles utilisant les instructions disponibles sur le processeur. Cette émulation entraîne alors une dégradation des performances de l'arithmétique, et donc des logiciels, utilisés pour le calcul exact.

On propose de développer un circuit qui réalisera au niveau matériel toutes les opérations élémentaires de l'arithmétique exacte. L'architecture du circuit sera optimisée pour répondre aux spécificités de cette arithmétique et plus particulièrement pour calculer sur des nombres de grande taille. Afin d'augmenter encore les performances des logiciels, on intégrera en matériel certaines fonctions usuelles du calcul exact.

Abstract

The floating point arithmetics used within machines for scientific calculations introduce errors within the results of such processings. Indeed computer processings consider a limited number of significant figures for operands, thereby corresponding to only an approximation of the exact value. As programs get executed, there is a gradual loss in the precision of the numbers handled. These accumulations of errors may lead to incorrect results without the user being aware of it. Among the solutions developed in order to take into account these calculation errors, only exact arithmetics lead to a result on which one may thrust. It is thus necessary to emulate each elementary operation of exact arithmetics using software routines according to the statements available on the processor. Such an emulation gives rise to a loss in the arithmetic precision and thus in that of the software used for the exact calculation.

We propose to develop a processor that would execute all the elementary operations of exact arithmetics in hardware. The architecture of the circuit will be optimised in order to handle the characteristics of such arithmetics and to allow the processing of large numbers. Moreover to increase the performances of software, some common functions of exact calculations will be implemented in hardware.

Remerciements

Je tiens à remercier Monsieur Guy Mazare, Professeur à l'ENSIMAG, qui a bien voulu me faire l'honneur de présider mon jury de thèse.

Que Messieurs Daniel Etiemble, Professeur à l'Université de Paris Sud Orsay, et Habib Merhez, Maître de Conférences à l'Université Pierre et Marie Curie, Paris 6, trouvent ici l'expression de ma reconnaissance pour avoir accepté d'être rapporteurs de mon travail.

Je remercie aussi Messieurs Daniel Auvergne, Professeur à l'Université de Montpellier II, et Marc Renaudin, Professeur à l'ENSERG, qui ont accepté de faire partie de mon jury de thèse ainsi que Monsieur Alain Guyot pour m'avoir accueilli dans son équipe.

Je tiens à témoigner ma reconnaissance à tous ceux qui m'ont aidé durant ces trois années, aussi bien dans l'équipe Integrated Systems Design qu'au sein du laboratoire TIMA. Ces remerciements vont tout particulièrement à Jean-Marc Daveau, François Naçabal, Philippe Guillaume, Damien Veychard, Polen Kission ainsi qu'à Alexandre, Selim, Ricardo, Pascal, Patricia.

Je remercie enfin toute ma famille pour m'avoir toujours soutenu au cours de mes études et je souhaite dédier cette thèse à Valérie et Manon.

Table des matières

1 Introduction **19**

2 La précision des calculs **23**
2.1 L'arithmétique des ordinateurs 23
2.1.1 Les arithmétiques entières et flottantes 23
2.1.2 La norme IEEE 754 25
2.1.3 Les limites de l'arithmétique flottante 26
2.2 Solutions aux erreurs de calcul 27
2.2.1 Validation des résultats 28
2.2.2 Atténuation des erreurs 28
2.2.3 Arithmétique exacte 29
2.3 Une alternative pour l'arithmétique exacte 31
2.4 Conclusions 32

3 Le cœur du processeur **33**
3.1 Le format des données 33
3.1.1 Les nombres en grande précision 34
3.1.2 Les nombres signés 34
3.2 Jeu d'instructions 35
3.2.1 Instructions sur les blocs 35
3.2.2 Accès au banc de registres 39
3.2.3 Instructions sur les tableaux 41
3.2.4 Gestion dynamique des débordements 42
3.3 Architecture générale du cœur du processeur 44
3.3.1 Les principales architectures 44
3.3.1.1 CISC 44
3.3.1.2 RISC 45
3.3.1.3 Superscalaire 46
3.3.1.4 Vectorielle 47
3.3.1.5 VLIW 47
3.3.2 Architecture retenue 48
3.3.3 Hiérarchie mémoire 50
3.3.4 La structure des pipelines 51
3.4 Conclusions 51

4 Lancement d'une instruction 53
4.1 Chargement d'une instruction 53
4.2 Décodage d'une instruction 54
4.2.1 Types des instructions 55
4.2.2 Détection des dépendances 58
4.2.3 Allocation des ressources 60
4.2.4 Table des longueurs 61
4.2.5 Schéma général du décodage 65
4.3 Conclusions 65

5 Unités fonctionnelles 67
5.1 Architecture générale d'une unité 67
5.1.1 Opérateurs pipelinés 68
5.1.2 Notation redondante 69
5.1.3 Reconversion en CC2 70
5.2 Unité d'addition 71
5.2.1 Caractéristiques générales 71
5.2.2 Opérateur d'addition/soustraction 71
5.2.2.1 Taille d'un résultat 73
5.2.2.2 Additionneur/soustracteur 74
5.2.3 Synthèse et placement-routage 76
5.3 Unité de multiplication 79
5.3.1 Caractéristiques générales 79
5.3.2 Opérateur de multiplication-accumulation 80
5.3.2.1 Architecture du multiplieur 81
5.3.2.2 Multiplieur signé 81
5.3.2.3 Opération de multiplication-accumulation 86
5.3.3 Synthèse et placement-routage 86
5.3.4 Optimisations 88
5.3.4.1 Multiplieur régulier 88
5.3.4.2 "Tranche" de multiplieur 89
5.3.4.3 Réducteur 4/2 89
5.4 Unité de division 91
5.4.1 Caractéristiques générales 91
5.4.1.1 Quotient et reste exacts 91
5.4.2 Opérateur de division 92
5.4.2.1 Reconversion du quotient 93
5.4.2.2 Reconversion du reste 93
5.4.2.3 Signe du reste 95
5.4.3 Synthèse et placement-routage 98
5.5 Unité de décalage 98
5.5.1 Caractéristiques générales 98
5.5.2 L'opérateur de décalage/normalisation 98
5.5.2.1 Le décaleur 100
5.5.2.2 Le codage des commandes 101
5.5.2.3 La fonction normalisation 101
5.5.3 Synthèse et placement-routage 103

5.5.4 Optimisations 104
5.5.4.1 Décaleur 8 positions 104
5.5.4.2 Décaleur 16 positions 106
5.6 Unité de chargement/rangement 108
5.6.1 Caractéristiques générales 108
5.6.2 Architecture de l'unité 108
5.6.3 Synthèse et placement-routage 109
5.7 Conclusions 109

6 Optimisations de l'architecture 113
6.1 Gestion dynamique de la longueur 113
6.1.1 Principe 114
6.1.2 Détection des 3 types de résultats 114
6.1.3 Calculs des longueurs 117
6.2 Recouvrement des opérations 122
6.2.1 Principe 122
6.2.2 Détection du recouvrement 124
6.2.3 Estimation de la longueur 125
6.3 Conclusions 129

7 Coûts et performances 131
7.1 Coût matériel 131
7.2 Performances 132
7.2.1 Réduction de la durée des opérations élémentaires 133
7.2.2 Réduction de la durée des fonctions complexes 135

8 Conclusions et perspectives 139

A Architecture du multiplieur 147

B Programmation VHDL 153

Table des figures

2.1 Codage d'un nombre flottant dans la norme IEEE 754. 25
2.2 La structure de Givaro. 31

3.1 Format des nombres dans le processeur. 35
3.2 Principe d'une addition/soustraction sur un tableau. 36
3.3 Principe d'une multiplication sur un tableau. 37
3.4 Principe d'une division sur un tableau. 37
3.5 Principe d'un décalage sur un tableau. 38
3.6 Gestion dynamique des débordements pour l'addition. 44
3.7 Principe du pipeline. 46
3.8 Architecture superscalaire. 46
3.9 Architecture vectorielle. 47
3.10 Architecture VLIW. 48
3.11 Architecture générale du cœur du processeur. 49
3.12 Structure des pipelines dans le processeur. 52

4.1 Détermination de la valeur du compteur de programme CP. 54
4.2 Adressage sur 8 bits des blocs et des tableaux. 55
4.3 Codage de type I.a 56
4.4 Codage de type I.b 56
4.5 Codage de type I.bi 56
4.6 Codage de type I.c 57
4.7 Codage de type I.ci 57
4.8 Codage de type II 57
4.9 Codage de type III 58
4.10 Réduction des délais de l'étage de décodage. 59
4.11 Détection d'une dépendance entre deux adresses. 60
4.12 Détection des bus de lecture libres. 62
4.13 Détection des bus d'écriture libres. 62
4.14 Allocation des bus de lecture et d'écriture. 63
4.15 Lecture de la table des longueurs. 64
4.16 Schéma général du décodage. 65

5.1 Architecture générale d'une unité. 68
5.2 Propagation de la retenue pour les opérations d'addition. 69
5.3 Accès aux registres effectués par l'unité d'addition. 72
5.4 Schéma général de l'opérateur d'addition/soustraction. 72
5.5 Réducteur 4/2. 75

5.6 Additionneur redondant. 76
5.7 Accès aux registres effectués par l'unité de multiplication. 79
5.8 Schéma général de l'opérateur de multiplication-accumulation. 80
5.9 Arbre de réducteurs 4/2. 82
5.10 Disposition des produits partiels. 83
5.11 Principe de l'accumulation et de la reconversion en CC2. 83
5.12 Réducteur 4/2 acceptant une entrée signée ou non signée. 84
5.13 Positions des bits négatifs en sortie du multiplieur. 85
5.14 Câblage de l'accumulateur. 87
5.15 La multiplication-accumulation avec une troisième opérande. 87
5.16 Distribution des bus d'entrée dans les cellules du multiplieur. 89
5.17 Vue partielle du câblage entre deux colonnes de réducteurs 4/2 90
5.18 Dessins des masques des réducteurs 4/2 en *full-custom*. 90
5.19 Accès aux registres effectués par l'unité de division. 91
5.20 Schéma général de l'opérateur de division. 94
5.21 Schéma de la reconversion "à la volée". 94
5.22 Reconversion "à la volée" qui calcule Q_n, Q_n^{-n} et Q_n^{+n}. 96
5.23 Correction du quotient. 97
5.24 Accès aux registres effectués par l'unité de décalage. 99
5.25 Schéma général de l'opérateur de décalage/normalisation. 99
5.26 Réalisation de décalages à droite. 100
5.27 Réalisation matérielle de la commande des décaleurs. 102
5.28 Détection d'un changement de valeur entre deux bits consécutifs. . . 102
5.29 Fonction priorité sur 16 bits. 103
5.30 Principe de la normalisation. 103
5.31 Décaleur à barillet. 104
5.32 Décaleur à barillet avec cellule de base à deux transistors. 105
5.33 Réalisation *full-custom* du dessin des masques de la cellule de base des décaleurs. 105
5.34 Décaleurs 8 positions réalisés à partir de 16 décaleurs élémentaires. . 106
5.35 Dessin des masques en *full-custom* de la cellule de base du décaleur 8 positions. 107
5.36 Dessin des masques en *full-custom* de la cellule de base du décaleur 16 positions. 108
5.37 Unité de chargement/rangement. 109

6.1 Longueur d'un résultat selon que le nombre est signé ou non signé. . 114
6.2 Détection des résultats intermédiaires de type 2. 117
6.3 Schéma pour la réalisation des signaux *d0* et *d1*. 118
6.4 Calcul de la longueur pour l'unité d'addition. 119
6.5 Calcul de la longueur pour l'unité de multiplication. 121
6.6 Mise à jour des signaux NS/S lors du calcul des longueurs. 121
6.7 Principe du recouvrement de deux opérations. 123
6.8 Schéma général du module de détection des dépendances. 125
6.9 Longueur exacte supérieure à la longueur estimée. 127
6.10 Longueur exacte inférieure à la longueur estimée. 128

7.1 Plan de masse du circuit. 132

A.1 Câblage de la zone de queue du multiplieur. 150
A.2 Câblage de la zone de tête du multiplieur. 151
A.3 Câblage d'une "tranche" de réducteurs 4/2 de la zone centrale. 152

Liste des tableaux

2.1 Probabilité p_N d'avoir la moitié des bits de la mantisse 27

3.1 Nombre d'accès au banc de registre pour les opérations élémentaires. 40
3.2 Nombre d'accès au banc de registre dans le cas d'opérandes de N blocs. 41

4.1 Comparaisons des adresses selon le type des données. 58

5.1 Codage des chiffres en notation *Borrow Save*. 70
5.2 Comparaisons de quatre architectures d'additionneurs sur 128 bits. . 71
5.3 Retenues entrantes des opérateurs de l'unité d'addition. 74
5.4 Codage de l'opérande d'entrée A. 74
5.5 Codage de l'opérande d'entrée B. 75
5.6 Détection des conditions en fonctions des signaux $A = B$ et $A > B$. . 76
5.7 Produits partiels négatifs. 85
5.8 Exemples de codage pour la commande des décaleurs. 101

6.1 Possibilités de recouvrement entre deux opérations. 123
6.2 Caractéristiques des opérations à partir des signaux I/P et R/L. . . 124

7.1 Caractéristiques des blocs. 131
7.2 Temps de traversée des cellules en technologie 0,7 μm et 0,35 μm. . . 133
7.3 Caractéristiques des processeurs. 134
7.4 Cycles nécessaires par mot de 128 bits pour effectuer des opérations élémentaires en arithmétique exacte. 134
7.5 Durée des opérations élémentaires par mot de 128 bits. 135
7.6 Durée des fonctions d'addition et de multiplication-accumulation. . . 138
7.7 Facteur d'accélération apporté par le processeur par rapport à des processeurs utilisant une technologie similaire. 138
7.8 Facteur d'accélération apporté par le processeur2 par rapport à des processeurs utilisant une technologie similaire. 138

A.1 Disposition des cellules du multiplieur en fonction du poids des entrées. 148
A.2 Disposition des cellules du multiplieur en fonction du poids des entrées. 149

Chapitre 1

Introduction

La puissance de calcul des ordinateurs n'a cessé de croître de façon importante depuis plus de trente ans. Une simple machine sur un bureau est maintenant capable d'exécuter plusieurs millions d'instructions par seconde. Cette augmentation ininterrompue de la puissance des ordinateurs a alors permis aux scientifiques de traiter des problèmes toujours plus gros et plus complexes. Il est aujourd'hui possible de résoudre des problèmes auxquels l'homme n'aurait pas pu s'attaquer, pour des raisons de temps ou de complexité, si il n'avait pas disposé de machines offrant de telles possibilités de calcul.

Le revers de la médaille à la puissance des ordinateurs est qu'elle fait perdre à l'utilisateur tout contact avec ses calculs. Il ne lui est plus possible en effet de suivre l'évolution d'un programme et de valider l'ensemble des opérations qui sont effectuées. Il est alors contraint de faire entièrement confiance aux résultats qui sont retournés par la machine.

Or les ordinateurs ne calculent pas toujours très juste. L'arithmétique virgule flottante utilisée dans les processeurs modernes pour le calcul scientifique introduit des erreurs d'arrondi. Celles-ci sont liées à l'utilisation d'opérandes possédant un nombre limité de chiffres significatifs. Les nombres codés en machine ne représentent alors qu'une approximation de la valeur exacte. La propagation de ces arrondis à travers les calculs fait que les résultats sont toujours entachés d'une erreur. La question qui se pose alors est de savoir, lorsque l'on effectue une opération sur un ordinateur, qu'elle est l'importance de cette erreur dans le résultat obtenu. La perte de contrôle de l'utilisateur sur les calculs effectués rend malheureusement très difficile, voir impossible, toute réponse à cette question. Il n'est donc pas possible lorsque l'on effectue un calcul en virgule flottante de connaître avec certitude la précision de son résultat.

Si par le passé, les applications ont montré que bien souvent la précision obtenue était correcte, ceci est de moins en moins vrai aujourd'hui. La raison principale provient de la taille toujours plus importante des problèmes à résoudre qui font appel à toujours plus de calculs. Cette augmentation entraîne une accumulation des erreurs d'arrondi. On assiste alors à une dégradation progressive de la précision des résultats obtenus au fur et à mesure du déroulement du programme. Cette dégradation peut même conduire à des résultats totalement faux sans que l'utilisateur en soit averti.

De nombreuses solutions ont été développées pour tenter de maîtriser ces problèmes d'arrondi et de perte de précision. Bien souvent elles permettent de diminuer

les erreurs introduites par l'arithmétique virgule flottante des processeurs mais sans pour autant assurer la validité des résultats. Seul l'emploi d'une arithmétique exacte peut permettre à l'utilisateur d'obtenir des résultats dont il soit sûr. Cette solution est obtenue par l'intermédiaire de logiciels qui offrent des fonctions pour réaliser toutes les opérations de base du calcul exact. Bien que ces logiciels représentent actuellement la seule solution réellement efficace pour effectuer des calculs sans erreur, ils ne sont pas très utilisés car ils nécessitent des temps de calcul extrêmement longs. Les processeurs actuels, sur lesquels ils s'exécutent, ne possèdent pas une architecture adaptée à ce type de calcul.

L'objectif de cette thèse est de faire l'étude et le développement d'un circuit qui soit dédié au calcul exact afin d'augmenter les performances de ces logiciels. On propose de réaliser une architecture qui implémentera au niveau matériel toutes les opérations élémentaires de l'arithmétique exacte. Pour réaliser efficacement ces opérations, l'architecture doit tenir compte de certaines caractéristiques propres à l'arithmétique exacte comme la taille importante des nombres générés lors des calculs ou bien le format variable des résultats en sortie des opérateurs. Ce sont ces caractéristiques qui pénalisent les logiciels de calcul exact car elles ne peuvent pas être traitées correctement dans les architectures des processeurs modernes. Ces dernières utilisent en effet un format limité à 32 ou 64 bits qui est insuffisant pour représenter correctement les nombres avec toute leur précision. L'arithmétique exacte a la particularité de conserver tous les bits d'un résultat, sans faire de troncature ou d'arrondi, ce qui entraîne une augmentation de la taille des nombres au fur et à mesure des calculs. Pour avoir des performances élevées, l'architecture doit donc pouvoir manipuler des nombres de plusieurs milliers de bits. Le temps passé pour accéder en lecture ou en écriture à de tels nombres peut être particulièrement important par rapport au temps de calcul. Il faut donc optimiser l'utilisation des opérandes au sein du circuit de façon à diminuer le coût des transferts de données. De même l'utilisation d'opérateurs arithmétiques permettant de gérer efficacement le problème lié au format variable des résultats peut s'avérer une source d'accélération importante pour les calculs. Nous verrons aussi au cours de cette thèse que la détermination dynamique de la longueur d'un résultat peut permettre une meilleure utilisation des ressources matérielles disponibles et augmenter ainsi les performances de l'architecture en diminuant les cycles d'attente.

Toutes les caractéristiques de l'architecture du circuit seront développées dans cette thèse selon le plan suivant :

Dans le chapitre 2, on introduit l'arithmétique utilisée dans les ordinateurs et plus particulièrement l'arithmétique virgule flottante. On présente les erreurs de calculs qu'elle peut engendrer ainsi qu'un certain nombre de solutions qui ont été proposées pour essayer de les maîtriser. On expose enfin l'avantage que peut représenter une solution matérielle pour le calcul exact.

Le chapitre 3 présente les grandes lignes de l'architecture du circuit. Il décrit plus particulièrement certains points importants comme le format des nombres, le jeu d'instruction ou la structure des pipelines.

Le chapitre 4 décrit les deux étapes qui interviennent dans le lancement d'une instruction, à savoir le chargement de l'instruction depuis le cache et son décodage. On présentera les mécanismes de détection dynamique des dépendances et d'allocation

dynamique des ressources.

Dans le chapitre 5, on présente les caractéristiques des unités fonctionnelles. On décrit l'architecture des opérateurs arithmétiques qui composent chaque unité et on présente les caractéristiques principales de leur réalisation matérielle.

Le chapitre 6 introduit des solutions qui ont été développées au niveau matériel pour augmenter les performances de l'architecture. Il s'agit de la gestion de la longueur et du recouvrement de deux opérations.

Le chapitre 7 présente le coût en surface de ce circuit ainsi que les performances que l'on peut en attendre.

Le dernier chapitre est la conclusion. Il présente les perspectives attachées aux travaux menés dans le cadre de cette thèse.

Chapitre 2

La précision des calculs

Ce chapitre présente les principaux aspects du calcul en machine. Il introduit en particulier les caractéristiques des arithmétiques implantées dans les processeurs et leurs influences sur les calculs. Pour des raisons d'implantation matérielles, ces derniers sont effectués avec une précision limitée ce qui peut s'avérer être une source d'erreur importante.

Le calcul scientifique ne pouvant pas toujours se satisfaire de résultats imprécis, des solutions ont vu le jour pour tenter de maîtriser ces erreurs. Parmi celles-ci, seule l'utilisation d'une arithmétique exacte permet d'obtenir un résultat qui soit toujours correct. Malheureusement cette solution ne peut être obtenue que par logiciel avec des temps de calcul extrêmement importants.

On propose à la fin de ce chapitre une nouvelle solution qui consiste à développer un processeur pour l'arithmétique exacte. L'architecture de ce circuit permettra d'optimiser les opérations spécifiques à cette arithmétique afin de diminuer les temps d'exécution des logiciels de calcul exact.

2.1 L'arithmétique des ordinateurs

Les ordinateurs utilisent des arithmétiques qui sont différentes de celle que l'on utilise lorsque l'on fait un calcul “à la main”. L'emploi de ces arithmétiques particulières est imposé par des contraintes matérielles et en particulier le caractère “non infini” des bus, mémoires et autres éléments constituant un circuit. On verra cependant dans le paragraphe 2.1.3 que ces arithmétiques, qui permettent pourtant de réaliser efficacement des calculs sur les machines modernes, peuvent être la source de résultats erronés, voir totalement faux.

2.1.1 Les arithmétiques entières et flottantes

Les ordinateurs utilisent deux types d'arithmétiques : l'arithmétique entière et l'arithmétique flottante. Bien que toutes les deux manipulent des nombres codés sur 32 ou 64 bits, elles possèdent cependant des chemins de données différents avec leurs registres et leurs unités spécifiques car le codage des nombres et les calculs à effectuer ne sont pas les mêmes.

En arithmétique entière, les nombres sont représentés sous la forme binaire classique :

– Arithmétique entière non signée

$$A_n = a_{n-1}\, a_{n-2} \ldots a_1\, a_0 = \sum_{i=0}^{n-1} a_i \times 2^i$$

– Arithmétique entière signée

$$A_n = a_{n-1}\, a_{n-2} \ldots a_1\, a_0 = -a_{n-1} \times 2^{n-1} + \left(\sum_{i=0}^{n-2} a_i \times 2^i \right)$$

L'arithmétique entière correspond à celle que l'on utilise lorsque l'on fait un calcul “à la main”. Elle possède cependant un inconvénient majeur pour le calcul scientifique qui est d'avoir un domaine de définition des nombres trop restreint.

En arithmétique virgule flottante, les nombres possèdent un codage spécifique qui se décompose généralement sous la forme de deux champs : la mantisse (m) et l'exposant (e). Un nombre s'exprime donc par la formule :

$$Nombre = m \times 2^e$$

A l'apparition du format flottant, chaque constructeur implantait son propre codage ce qui empêchait toute portabilité d'une machine à une autre. Parmi les principaux codages développés, on peut citer ceux de l'IBM/370, du VAX de DEC ou du CRAY.

L'IBM/370 avait un format simple précision[1] composé d'un exposant codé sur 7 bits et d'une mantisse sur 24 bits. Les nombres étaient stockés en hexadécimal pour réduire le nombre de décalages lors de l'alignement des mantisses. L'IBM/370 n'avait pas de mode d'arrondi et tronquait systématiquement les nombres. Par ailleurs, toutes les opérations retournaient un nombre valide. Il n'y avait pas de codage particulier pour l'écriture des exceptions. Il fallait donc fixer des conventions au niveau des routines de calcul pour déterminer les résultats qui n'étaient en fait pas corrects.

Le format simple précision du VAX était constitué d'un exposant codé sur 8 bits et d'une mantisse codée sur 23 bits. Les nombres étaient normalisés et s'écrivaient tous sous la forme $1, f$ où f représentait la partie fractionnaire du nombre flottant. Le VAX ne stockait donc dans les 23 bits de la mantisse que les bits de f. Le premier bit, toujours à 1, n'était jamais mémorisé dans cette représentation. Cette machine ne possédait pas de codage des valeurs particulières. Elle avait cependant un mode de fonctionnement qui permettait de traiter les résultats qui n'étaient pas des nombres (division par zéro, racine carrée d'un nombre négatif,...).

Le CRAY utilisait quant à lui un format constitué d'un exposant de 15 bits et d'une mantisse de 48 bits qui contenait la partie fractionnaire du nombre. Ce format permettait d'avoir un domaine de définition des nombres machine très grand.

Afin d'homogénéiser les formats flottants utilisés par les différentes machines et d'assurer ainsi la portabilité entre divers matériels, les scientifiques et les industriels

[1]Correspond à un nombre flottant codé sur 32 bits Le format double précision est codé sur 64 bits.

ont développé la norme IEEE 754 qui fixe précisement les caractéristiques de l'arithmétique flottante.

2.1.2 La norme IEEE 754

La norme IEEE 754 [34] définit précisément le format des nombres flottants. Ceux-ci sont composés de trois champs (Figure 2.1), dont les fonctions et les caractéristiques sont totalement fixées par la norme :

- Le premier champ contient le signe (s). Ce champ est codé sur 1 bit.
- Le deuxième champ contient l'exposant (e). Celui-ci est codé sur 8 ou 11 bits respectivement pour la simple ou la double précision. L'exposant peut alors prendre les valeurs comprises dans les intervalles [-126, +127] lorsqu'il est codé sur 8 bits et [-1022, +1023] lorsqu'il est codé sur 11 bits.
- Le troisième champ contient la partie fractionnaire de la mantisse (f). En effet, la norme impose que les valeurs contenues dans la mantisse soient normalisées, c'est-à-dire que l'on ait :

  $$1 \leq m < 2$$

  Cette normalisation nécessite d'avoir un codage particulier pour le zéro. La valeur contenue dans la mantisse peut alors s'écrire sous la forme :

  $m = 1,f$ où f représente la partie fractionnaire de la mantisse.

  On ne stocke dans ce troisième champ que les bits servant à coder f. Le 1 en début de mantisse est implicite et ne sera jamais stocké. Nous verrons dans la suite de ce paragraphe qu'il existe cependant des valeurs spéciales, non normalisées, qui peuvent être écrites dans ce troisième champ. Ce dernier comprend 23 bits pour la simple précision et 52 bits pour la double précision.

FIG. 2.1: Codage d'un nombre flottant dans la norme IEEE 754.

Dans la norme IEEE 754, un nombre s'exprime alors par la formule :

$$Nombre = (-1)^s \times 1,f \times 2^e$$

Cette norme possède aussi les caractéristiques suivantes :

- Le codage du zéro s'écrit de façon particulière. On a de plus deux valeurs possibles, -0 et +0.
- Il existe des valeurs spéciales : $+\infty$, $-\infty$ et NaN (*Not a number*). Cette dernière sert à caractériser les résultats qui ne sont pas des nombres (résultat d'une division par zéro, racine carrée d'un nombre négatif).
- Il existe des nombres dénormalisés pour représenter les valeurs comprises entre $1,0 \times 2^{E_{min}}$ et zéro. Ces nombres dénormalisés permettent de limiter l'écart entre le plus petit nombre représentable par la norme et zéro.

- Il existe quatre modes d'arrondi : au plus proche (en cas d'égalité, on arrondit vers le nombre dont le chiffre de poids faible est pair), vers zéro, vers $+\infty$ et vers $-\infty$.
- Toutes les fonctions retournent une valeur définie par la norme. Ainsi tous les calculs peuvent s'effectuer sans interruption et retourner un résultat cohérent même lors de calculs "particuliers".

Ces caractéristiques font de l'arithmétique flottante un outil très adapté au calcul scientifique. Cette norme est d'ailleurs implantée sur la quasi totalité des ordinateurs construits après 1985.

2.1.3 Les limites de l'arithmétique flottante

L'arithmétique virgule flottante, bien que très adaptée au calcul scientifique, possède cependant des limites. Elle peut en effet introduire des erreurs dans les résultats des calculs. Ces erreurs sont liées essentiellement à la perte de précision entraînée par l'utilisation d'une mantisse de taille fixe pour stocker les nombres. On présente dans la suite de ce paragraphe deux types d'erreurs qui peuvent apparaître dans les calculs :

1. **L'absorption**
   Ce phénomène se produit lorsque l'on additionne deux nombres qui ont des ordres de grandeurs différents. La mantisse du plus petit nombre est décalée afin de rendre son exposant identique à celui du plus grand nombre. Ce décalage fait perdre une partie de l'information concernant le plus petit des deux nombres comme le montre l'exemple suivant. Nous utiliserons dans cet exemple des nombres ayant une mantisse et un exposant codés respectivement sur 6 et 3 bits.

   $$\begin{array}{llll} \text{Nombre A :} & A_m = 110000 & \text{et} & A_e = 110 \\ \text{Nombre B :} & B_m = 100110 & \text{et} & B_e = 010 \end{array}$$

   Lorsque l'on additionne les deux nombres A et B, on décale la mantisse de B de façon à écrire ce dernier avec la même valeur d'exposant que A :

   $$\begin{array}{rl} 110000 & 110 \\ + \; 0 \longrightarrow 10\,(0110) & 110 \\ \hline 110010\,(0110) & 110 \end{array}$$

   Le résultat machine sera donc $R_m = 110010$ et $R_e = 110$.
   La mantisse ayant un nombre fixe de bits, toute l'information provenant de ce calcul ne peut pas être stockée. Dans cet exemple, on perd ainsi l'information codée sur les 4 derniers bits entre parenthèses. Ce phénomène, qui correspond à une perte de précision par rapport au résultat exact, survient aussi à chaque multiplication puisque l'on ne conserve que la moitié des bits (en poids fort) par rapport au résultat exact.

2. **La cancellation**
   Ce second type d'erreur survient lorsque l'on fait la soustraction de deux nombres de même signe ayant des valeurs voisines. Dans ce cas, la mantisse

du résultat contient un nombre important de zéros en poids forts qui proviennent du fait que les mantisses des nombres soustraits étaient identiques dans leurs parties poids fort. Or les nombres étant normalisés, il faut ensuite effectuer un décalage vers la gauche en poussant avec des zéros non significatifs qui vont faire partie intégrante du résultat comme on peut le voir sur l'exemple suivant :

$$\begin{array}{lrcl}\text{Nombre A :} & A_m = 110111 & \text{et} & A_e = 110 \\ \text{Nombre B :} & B_m = 110010 & \text{et} & B_e = 110\end{array}$$

Soustraction de ces deux nombres, puis normalisation du résultat :

	110111	110
	– 110010	110
	000101	110
Normalisation de la mantisse	101000 ◄── 0	011

Dans cet exemple, on introduit trois zéros non significatifs dans la mantisse du résultat. On n'a donc plus que les trois bits de poids fort qui contiennent réellement une information correcte. La précision des nombres, qui était initialement de 6 bits, est maintenant ramenée à trois pour ce résultat, mais aussi pour tous les autres calculs qui utiliseront ce résultat comme opérande.

Ces erreurs ont un impact limité sur le résultat lorsque l'on effectue une opération. Elles peuvent cependant devenir critiques lorsque l'on enchaîne un grand nombre d'opérations. Le tableau 2.1 provenant de [23], donne la probabilité p_N d'avoir la moitié des bits de la mantisse (27 bits) non significatifs lorsque l'on effectue N soustractions.

N	5	15	20	100
p_N	0,21	0,51	0,91	0,99

TAB. 2.1: Probabilité p_N d'avoir la moitié des bits de la mantisse (27 bits) non significatifs après N soustractions.

La puissance de calcul toujours croissante des ordinateurs permet aux utilisateurs de traiter des problèmes sans cesse plus gros et plus complexes et qui font appel à une quantité toujours plus importante de calculs. Or plus le nombre de calculs augmente, plus l'erreur sur le résultat final augmente aussi, à tel point que bien souvent, la validité même du résultat peut être remise en cause sans que l'utilisateur en soit averti. Il existe dans la littérature de nombreux exemples de calculs qui ne donnent pas le bon résultat à cause des erreurs introduites par l'arithmétique virgule flottante [55][77][65].

2.2 Solutions aux erreurs de calcul

De nombreuses solutions ont été développées pour essayer de résoudre les problèmes de précision que peuvent rencontrer les personnes qui réalisent des calculs

sur ordinateurs. Cependant ces solutions ne sont souvent adaptées qu'à un type bien précis de calcul et ne concernent donc qu'un nombre restreint de problèmes. Ces solutions touchent plusieurs domaines (arithmétique, logiciel, matériel...) et peuvent être classées en 3 catégories selon le degré “d'amélioration” qu'elles proposent. La première catégorie correspond aux solutions qui permettent de valider un résultat en renseignant l'utilisateur sur l'erreur commise, la seconde concerne les solutions qui améliorent la précision du résultat et enfin la troisième correspond à celles qui permettent d'obtenir le résultat exact. Ces différentes solutions sont présentées plus en détails dans les paragraphes suivants.

2.2.1 Validation des résultats

Le concept de ces solutions est d'informer l'utilisateur sur la justesse de ses résultats afin qu'il puisse prendre les décisions appropriées lorsqu'il jugera la précision trop faible.

La première approche est de représenter les nombres sous forme d'un intervalle qui contient toujours le résultat exact [53]. Cette approche, qui a été particulièrement étudiée par U.W. Kulisch [45], a permis de développer des logiciels pour calculer en arithmétique d'intervalle [40][21]. M. Daumas [17] a présenté une arithmétique d'intervalle qui utilise une partie des bits les moins significatifs de la mantisse pour coder la valeur de l'intervalle. Cette solution permet de passer facilement de l'arithmétique virgule flottante à l'arithmétique d'intervalle.

La seconde approche, dite probabiliste, consiste d'abord à réaliser plusieurs fois les calculs en les perturbant aléatoirement avec un ordre de grandeur identique à celui des erreurs d'arrondi, puis ensuite à étudier les effets sur les résultats [63][79]. Ce principe est celui utilisé par la méthode CESTAC pour évaluer les erreurs d'arrondi [13].

Une troisième approche consiste à faire l'étude de la fiabilité arithmétique des logiciels numériques pour déterminer si ceux-ci sont sensibles aux perturbations engendrées par l'arithmétique virgule flottante [59][11].

Ces différentes approches ont cependant l'inconvénient de ne pas donner l'ordre de grandeur de l'erreur. La première par exemple retourne un intervalle majoré afin de toujours inclure le résultat exact, ce qui donne une estimation pessimiste de l'erreur commise. La méthode probabiliste quant à elle s'appuie sur des statistiques et ne peut pas donner une valeur sûre.

2.2.2 Atténuation des erreurs

La deuxième catégorie de solutions correspond à celles pour lesquelles un des paramètres qui entre en compte dans la réalisation des calculs a été modifié afin d'augmenter la précision.

L'utilisation de nouveaux algorithmes permet, à partir de calculs en virgule flottante, d'obtenir la même valeur que celle à laquelle on aurait arrondi le résultat exact [62][6][69]. Des algorithmes ont aussi été développés afin d'offrir à l'utilisateur la possibilité de fixer la précision qu'il désire en fonction de la nature de ses calculs [64].

D'autres recherches ont été faites dans le domaine matériel. Certaines architectures permettent de réaliser les opérations avec une précision arbitraire en utilisant

une mantisse de taille variable [70]. D'autres architectures plus spécialisées pour les opérations sur les matrices ou les vecteurs exécutent, à partir de l'arithmétique flottante, les calculs avec un maximum de précision [51][39][41].

Une autre approche consiste à développer de nouvelles arithmétiques puisque l'arithmétique virgule flottante utilisée dans les processeurs actuels est à la base des problèmes de précision dans les calculs. D. W. Matula et P. Kornerup ont présenté une arithmétique pour le calcul sur les rationnels [48] qui n'utilise qu'un seul champ pour le codage des deux composantes d'une fraction rationnelle. Ces nouvelles arithmétiques doivent être un bon compromis entre la précision qu'elles offrent, les fonctions qu'elles permettent de calculer, leur complexité de mise en œuvre et la rapidité des calculs. Une autre arithmétique, appelée "Arithmétique Paresseuse" [49], est un mélange d'arithmétiques flottante et entière. Cette dernière étant utilisée uniquement lorsque les calculs l'imposent afin de gagner du temps [35][4]. Milos Ercegovac a quant à lui ouvert la voie à un nouveau type d'arithmétique : l'arithmétique en ligne [75]. Elle présente de gros avantages au niveau de la précision des calculs. D'autres arithmétiques, basées sur celle-ci, ont ensuite fait l'objet de recherches [44][58].

Les solutions citées précédemment ne permettent bien souvent de réaliser les calculs qu'avec une précision finie. Seule l'arithmétique paresseuse et les arithmétiques en ligne permettent d'avoir une précision infinie mais cela est obtenu au détriment de la simplicité d'utilisation ce qui rend concrètement leur emploi difficile. La réalisation de calculs avec une grande précision, voire une précision infinie, nécessite alors d'utiliser une arithmétique exacte. Des travaux ont montré que certains types de calculs ne peuvent même conduire à un résultat correct que si l'on emploie une arithmétique exacte [60][66]. Cette dernière arithmétique est présentée plus en détail dans le paragraphe suivant.

2.2.3 Arithmétique exacte

Parmis les différentes solutions qui existent pour réaliser des calculs sans erreur, l'arithmétique exacte est la plus sûre et la plus simple à utiliser. Elle permet d'effectuer les calculs sans faire ni arrondi ni troncature. Un nombre conserve tous les bits significatifs qui proviennent de l'opération. Cela entraîne donc un format variable pour les résultats qui dépend de la taille des opérandes d'entrée et du type d'opération effectuée.

Cette propriété génère une différence importante entre l'arithmétique exacte et les arithmétiques implantées sur les processeurs actuels car ces dernières utilisent un format unique pour tous les nombres. Les architectures internes des processeurs ayant été optimisées pour tirer parti au maximum d'une représentation unique, cela rend incompatible l'implantation à faible coût d'une arithmétique exacte au niveau matériel sur une architecture de processeur moderne.

Il faut donc utiliser une solution logicielle si l'on veut réaliser des calculs en arithmétique exacte. Ces logiciels sont souvent constitués de plusieurs couches dont la plus basse émule l'arithmétique exacte à partir des arithmétiques dont disposent les processeurs. Les couches supérieures offrant pour leur part des opérations plus complexes pour aider l'utilisateur à résoudre des problèmes toujours plus importants. Parmi les logiciels existants, on peut citer MATHEMATICA [82], MAPLE [9][10], AXIOM-XL [37] ou PARI [2]. Tous ces systèmes sont construits sur un même modèle : un noyau réduit

qui comprend les fonctions de base optimisées, lesquelles sont ensuite appelées par un ensemble de fonctions de plus haut niveau. Ces systèmes ont cependant un très gros inconvénient qui est leur lenteur. Karasick présente des exemples relativement simples où le temps de calcul en arithmétique exacte est 10 000 fois plus grand que celui obtenu avec des nombres flottants [38].

Pour réduire ce problème, PARI utilise un noyau avec des fonctions codées en assembleur pour différentes machines. Afin de ne pas trop réduire la portabilité du système, il existe une version écrite en C mais ses performances sont alors éloignées des versions en assembleur. Une des solutions intéressantes pour réduire le temps de calcul est d'utiliser des machines parallèles. En effet, en théorie, exécuter un programme sur une machine à N processeurs peut réduire jusqu'à N fois le temps de calcul. En pratique ce n'est pas exact car il se pose le délicat problème de la distribution du travail (charge) sur les différents processeurs [3][78]. Cependant le gain que peut apporter ce type de machine a incité au développement de systèmes de calcul parallèles. MAPLE [2] par exemple possède aussi une version parallèle mais celle-ci n'est pas très performante car il s'agit en fait du logiciel classique qui fait appel à une bibliothèque d'expression du parallélisme ce qui rend les communications très coûteuses car les données doivent être converties pour passer d'un processus à un autre.

Pour être plus efficaces, les systèmes utilisent généralement le schéma inverse à savoir des primitives d'expression du parallélisme couplées à une bibliothèque pour le calcul exact. La majorité des systèmes de calcul parallèles utilisent cette structure[2]. Parmis ceux-ci, on peut citer GIVARO [25] développé par le laboratoire LMC[3].

Malheureusement, le temps de calcul de ces systèmes, bien qu'il ait diminué avec l'apport du parallélisme, reste encore important. Il y a deux raisons principales à cela :

1. La première provient du fait que l'arithmétique exacte génère des nombres dont les tailles augmentent lors de l'exécution des programmes. Cela entraîne un accroissement de la durée des opérations qui est proportionnel à l'augmentation de la taille des nombres manipulés. Cette particularité étant propre à l'arithmétique exacte, il n'est pas possible d'en diminuer le coût de façon significative.

2. La seconde raison est liée au fait que les logiciels de calcul exact doivent faire une émulation de l'arithmétique à partir des machines sur lesquelles ils s'exécutent. Cela ralentit fortement les fonctions de base du noyau et donc l'ensemble des primitives du système car les arithmétiques dont disposent les processeurs ne sont pas du tout adaptées au calcul exact. On verra dans le paragraphe suivant que le coût engendré par ce deuxième cas peut être réduit puisque sa cause n'est pas inhérente à l'arithmétique exacte mais au matériel sur lequel elle est exécutée.

[2]Le site Web http ://www.can.nl contient une liste importante de systèmes de calcul séquentiels et parallèles.

[3]Laboratoire de Modélisation et de Calcul, Institut National Polytechnique de Grenoble, France.

2.3 Une alternative pour l'arithmétique exacte

Une des raisons de la lenteur des logiciels de calcul exact provient du fait qu'ils ne disposent pas directement sur les machines d'une arithmétique exacte adaptée et qu'ils doivent donc en faire l'émulation. Ce sont en particulier les fonctions de bas niveau des logiciels de calcul exact qui sont coûteuses en temps car elles sont réalisées sur des processeurs inadaptés à ce type de calcul.

On propose alors de développer un processeur pour le calcul exact [15][16]. Celui-ci réalisera directement au niveau matériel toutes les opérations élémentaires utilisées en arithmétique exacte. Il intégrera ainsi dans son jeu d'instructions les opérations de bas niveau des logiciels ce qui permettra de les exécuter plus rapidement. L'architecture du circuit sera aussi optimisée pour prendre en compte toutes les spécificités liées à ce type de calcul.

Le processeur sera développé en utilisant comme support logiciel la bibliothèque de calcul exact Givaro. On implantera dans le processeur les opérations élémentaires de Givaro ce qui permettra de simplifier le compilateur ainsi que la réécriture des fonctions de Givaro qui doivent s'exécuter sur le processeur. On intégrera aussi en matériel certaines fonctions usuelles de l'arithmétique exacte comme la gestion des signes, le traitement des dépassements de capacité ou bien le calcul de la longueur d'un résultat.

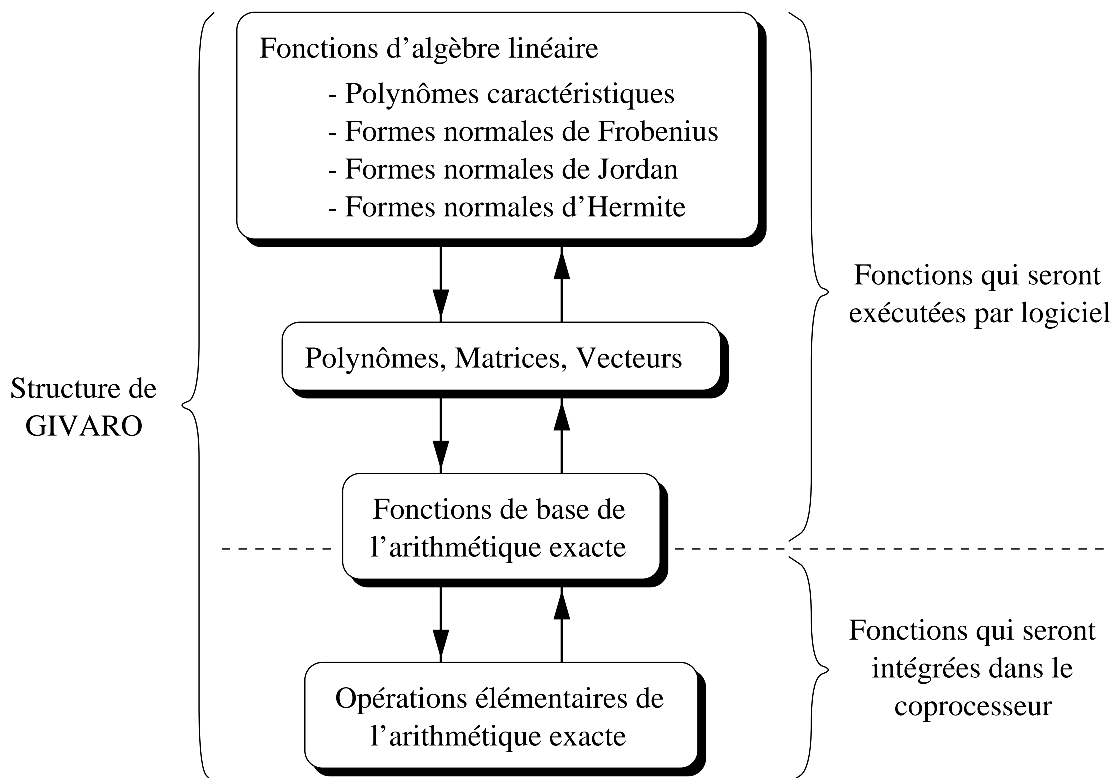


Fig. 2.2: La structure de Givaro.

L'intérêt de prendre comme support logiciel Givaro provient du fait qu'il possède une structure performante pour le calcul exact. Il utilise en effet des algorithmes efficaces pour coder les fonctions de haut niveau et possède en Athapascan [14][28]

un répartiteur dynamique de charge qui permet d'obtenir de très bons résultats pour la gestion du parallélisme sur plusieurs processeurs. La réalisation en matériel des opérations élémentaires et de certaines fonctions de base de l'arithmétique exacte peut permettre d'en faire un ensemble logiciel-matériel très performant.

La structure de GIVARO correspond à une évolution de celle développée dans PAC++ [67][26]. Elle se compose de plusieurs couches logicielles qui réalisent des fonctions de plus en plus complexes lorsque l'on va vers le niveau le plus haut (Voir figure 2.2). Chaque couche fait appel à des fonctions des couches inférieures. Il est donc essentiel dans une telle structure d'avoir les fonctions du niveau le plus bas qui soient très performantes car sinon c'est l'ensemble du système qui est pénalisé.

2.4 Conclusions

Ce chapitre, qui est une introduction aux problèmes liés à l'arithmétique des ordinateurs, présente différents types d'erreurs qui peuvent être introduites par l'arithmétique virgule flottante. Pour pallier ces erreurs, des solutions diverses ont été développées. Celles-ci apportent différents niveaux "d'amélioration" qui sont obtenus aux dépens de la complexité d'utilisation ou du temps d'exécution. Le choix d'une de ces solutions implique donc un compromis entre l'amélioration souhaitée et le coût qu'il faudra payer. Pour les domaines scientifiques où les calculs doivent être effectués sans erreur, le choix se restreint à l'emploi d'une arithmétique exacte. Dans ce cas, le coût en temps de calcul est tellement important qu'il limite l'accès à cette arithmétique. Beaucoup d'utilisateurs préfèrent en effet se détourner vers d'autres solutions certes moins précises mais bien plus rapides.

La principale raison de la lenteur des logiciels de calcul exact provient du fait qu'ils doivent faire l'émulation d'une arithmétique exacte à partir de l'arithmétique disponible sur les processeurs. Pour supprimer cette émulation coûteuse en temps, on propose de développer un processeur optimisé pour le calcul exact. Il permettra de réaliser efficacement toutes les opérations élémentaires de l'arithmétique exacte. On intégrera en matériel les fonctions de base de la bibliothèque GIVARO. Celle-ci possède en effet une structure de haut niveau qui est optimisée pour le calcul exact mais qui reste tributaire des performances de ses opérations arithmétiques de base. L'utilisation d'un processeur spécialisé a pour but de diminuer les temps d'exécution des fonctions élémentaires du calcul exact afin de rendre les logiciels comme GIVARO plus rapides et donc plus accessibles.

Chapitre 3

Le cœur du processeur

En arithmétique exacte, plus un algorithme calcule et plus la taille des objets manipulés augmente. L'architecture interne du processeur doit donc non seulement permettre d'effectuer des calculs en arithmétique exacte, mais elle doit aussi de pouvoir manipuler efficacement des nombres de plusieurs milliers de bits. Cette dernière caractéristique influe directement sur l'architecture générale du cœur du processeur. Il faut en effet minimiser les transferts de données au sein de l'architecture. Cela est d'autant plus critique que l'arithmétique exacte génère un volume de données intermédiaires très important dont la manipulation est coûteuse en temps. Un exemple cité dans [25] montre que l'inversion d'une matrice de dimension 126, dans laquelle la moitié des coefficients sont nuls, et qui utilise 200 Ko[1] pour écrire la matrice de départ, nécessite 9 Mo[2] pour écrire le résultat et plusieurs Go[3] pour stocker les résultats intermédiaires. Il faut donc chercher à diminuer les accès à la mémoire ainsi que les accès des unités fonctionnelles au banc de registres.

Une autre particularité des nombres de l'arithmétique exacte provient du fait qu'ils ont une taille variable qu'il est impossible de déterminer avant d'avoir effectué le calcul. Cela signifie que dans une architecture où la taille des bus, des registres, etc... est fixe, il peut exister des "dépassements" de capacité qui doivent être traités dynamiquement au niveau matériel.

Ce chapitre présente dans un premier temps les caractéristiques essentielles qui devront être intégrées au niveau de l'architecture générale du cœur du processeur afin qu'il puisse manipuler efficacement les nombres générés par l'arithmétique exacte. Dans la suite du chapitre on présente l'architecture qui a été retenue. Celle-ci possède des performances élevées en terme de calculs mathématiques et permet une intégration aisée des caractéristiques définies précédemment.

3.1 Le format des données

Le choix d'un format pour les données qui soit bien adapté au type des calculs à effectuer est essentiel dans l'obtention de bonnes performances. L'effet désastreux de l'utilisation du format virgule flottante pour le calcul exact en est la meilleure

[1] Ko (kilo-octets) : vaut 1024 octets, un octet étant codé sur 8 bits.
[2] Mo (méga-octets) : vaut 1024 kilo-octets.
[3] Go (giga-octets) : vaut 1024 méga-octets.

illustration. Le format des données dans le processeur doit permettre à une architecture ayant un mot machine de taille fixe de traiter efficacement des nombres de taille beaucoup plus grande. Il faut aussi que ce format permette aux opérateurs arithmétiques de retourner directement le résultat définitif. Il est en effet trop coûteux en temps de reprendre un résultat pour lui faire subir une opération telle que ajouter 1 ou complémenter le résultat car cela peut nécessiter de relire et de réécrire l'ensemble des bits du résultat.

3.1.1 Les nombres en grande précision

Dans GIVARO, les nombres sont représentés sous la forme d'un tableau. Un nombre en grande précision est donc "découpé" en blocs qui sont rangés consécutivement dans un tableau. Le bloc est un élément de taille fixe qui correspond généralement à l'élément de base (mot machine) manipulé par le processeur. Dans GIVARO, la taille d'un bloc est cependant paramétrable afin d'adapter le logiciel à l'évolution de l'architecture sur laquelle il s'exécute.

Au niveau matériel, les possibilités d'intégration dans un circuit sont limitées ce qui ne permet pas d'effectuer directement des calculs sur des nombres de taille importante. On utilise donc au niveau matériel une représentation des nombres identique à celle de GIVARO, ce qui a aussi l'avantage de simplifier le compilateur. La taille du bloc, qui correspond aux nombres de bits qui sont traités en parallèle dans une architecture, est un facteur d'accélération important. Dans le processeur, le format du bloc est de 128 bits. L'utilisation d'architectures qui calculent en temps constant (par l'intermédiaire d'une notation redondante) ou en temps $O(\log_2 N)$, avec N correspondant à la taille des opérandes, permet de travailler sur des blocs de tailles importantes pour un surcoût en temps faible. C'est la surface du circuit qui limite l'utilisation de formats encore plus grands pour les blocs.

3.1.2 Les nombres signés

Dans GIVARO, les nombres signés sont représentés en ajoutant un bit de signe à la valeur absolue du nombre qui est contenue dans le tableau. Cette notation avait été choisie car c'est celle qui s'adaptait le mieux, au niveau logiciel, aux architectures des processeurs actuels. Elle n'est cependant pas optimale pour certaines opérations comme l'addition ou la soustraction.

Le calcul s'effectue sur les valeurs absolues des nombres, la détermination du signe est faite à part. Le résultat d'une opération est donc toujours positif car il représente une valeur absolue. Pour les opérations de multiplication et de division, cela n'implique aucune complication contrairement aux additions et aux soustractions signées. Pour ces dernières, il faut s'assurer que le résultat du calcul sera bien une valeur absolue, donc positif, c'est pourquoi il faut comparer au préalable les deux opérandes et déterminer laquelle des deux a la plus grande valeur. Cela peut entraîner, avant de lancer l'opération, la comparaison séquentielle de tous les blocs de chacune des opérandes ce qui dégrade fortement les performances du système car les opérations d'addition et de soustraction sont les plus utilisées.

Pour supprimer cet inconvénient, on va utiliser la notation complément à 2 (CC2) pour l'écriture des nombres dans le processeur [55]. Cette notation, qui inclut directe-

ment le signe dans le codage du nombre, permet d'effectuer directement des additions ou des soustractions sur des nombres signés. En revanche, elle entraîne un surcoût en temps pour les multiplications et les divisions mais celui-ci peut-être fortement diminué par l'utilisation d'opérateurs signés qui effectuent directement les calculs en notation CC2.

En notation complément à 2, c'est le bit de poids fort du nombre qui contient le signe. Dans la représentation utilisée par le processeur, cela signifie que seul le bloc de poids fort d'un nombre est signé, les autres blocs du tableau sont non signés. Les opérateurs arithmétiques, qui effectuent les opérations successivement sur tous les blocs du tableau, doivent donc avoir une architecture qui leur permette d'effectuer les opérations dans les deux notations. Pour simplifier la tâche du programmeur, la gestion des signes est effectuée dynamiquement par le matériel. Pour cela, on associe à chaque bloc un bit (NS/S) qui sert à indiquer son type (signé ou non signé) et qui configure les opérateurs à chaque calcul sur un nouveau bloc. Un même opérateur arithmétique peut donc effectuer des opérations sur des nombres qui utilisent deux formats différents.

La représentation des nombres dans le processeur est celle donnée figure 3.1. L'utilisation de la notation CC2 permet d'effectuer les opérations d'addition et de soustraction plus rapidement que la notation "signe plus valeur absolue" utilisée par GIVARO. Pour les opérations de multiplication et de division, on obtient des performances sensiblement identiques mais au prix d'une augmentation de la complexité des opérateurs arithmétiques.

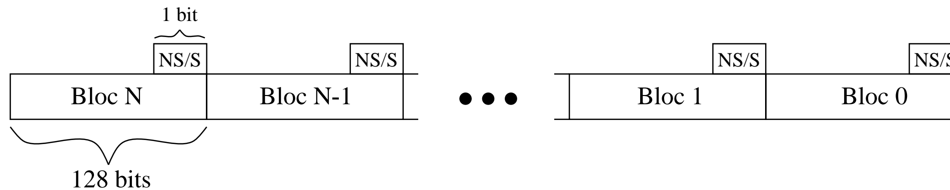


FIG. 3.1: Format des nombres dans le processeur.

3.2 Jeu d'instructions

Le jeu d'instruction du processeur comprend l'ensemble des opérations de base de l'arithmétique exacte ainsi que les opérations de contrôle comme les branchements conditionnels ou les sauts. Des opérations sur les tableaux ont été ajoutées afin de minimiser les accès au banc de registre et ainsi augmenter les performances lors de calculs sur des nombres de taille importante. Les instructions qui ont été définies prennent aussi en compte au niveau matériel les problèmes de "dépassement" de capacité et de taille variable qui peuvent survenir au cours d'un calcul de façon à les rendre transparents vis-à-vis du programmeur.

3.2.1 Instructions sur les blocs

Dans la représentation des nombres décrite paragraphe 3.1, l'élément de base est le bloc. Les instructions du processeur manipulent donc des opérandes d'entrée et de

sortie qui ont la taille d'un bloc. Au niveau de l'architecture, cette taille correspond à la dimension physique des registres et des bus de données.

Le jeu d'instruction du processeur peut être décomposé en 7 catégories :

- Addition, soustraction
- Multiplication, multiplication-accumulation
- Division
- Décalage, normalisation
- Chargement, rangement
- Instructions de contrôle
- Instructions de transfert.

Ces opérations s'appliquent à des blocs signés ou non signés. Une seule instruction est nécessaire pour les deux types car c'est le matériel qui gère le signe automatiquement.

1. **Addition, soustraction**
   - Addition de deux blocs. Cette opération retourne un résultat sur un bloc.
   - Addition de deux blocs et d'une retenue entrante *Cin* contenue dans un troisième bloc. Cette opération retourne un résultat sur un bloc et une retenue sortante *Cout* sur un bloc aussi. C'est cette instruction qui est utilisée pour réaliser l'opération d'addition sur les tableaux (Figure 3.2).
   - Soustraction de deux blocs. Cette opération retourne un résultat sur un bloc.
   - Soustraction de deux blocs et d'un bloc pour la retenue entrante *Cin*. Cette opération retourne un résultat sur un bloc et une retenue sortante *Cout* sur un bloc. C'est cette instruction qui est utilisée pour réaliser l'opération de soustraction sur les tableaux (Figure 3.2).

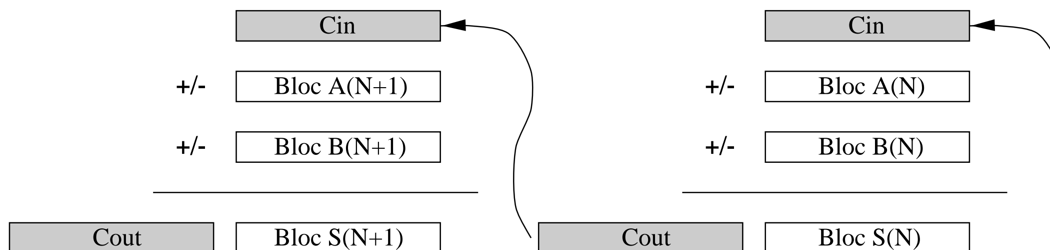


FIG. 3.2: Principe d'une addition/soustraction sur un tableau.

2. **Multiplication, multiplication-accumulation**
   - Multiplication de deux blocs. Cette opération retourne un résultat sur deux blocs.
   - Multiplication de deux blocs et accumulation avec un troisième bloc. Cette opération retourne un résultat sur deux blocs. C'est cette instruction qui est utilisée pour réaliser l'opération de multiplication sur les tableaux car elle permet de faire l'addition entre le bloc de poids fort d'un résultat et le bloc de poids faible du résultat suivant (Figure 3.3).

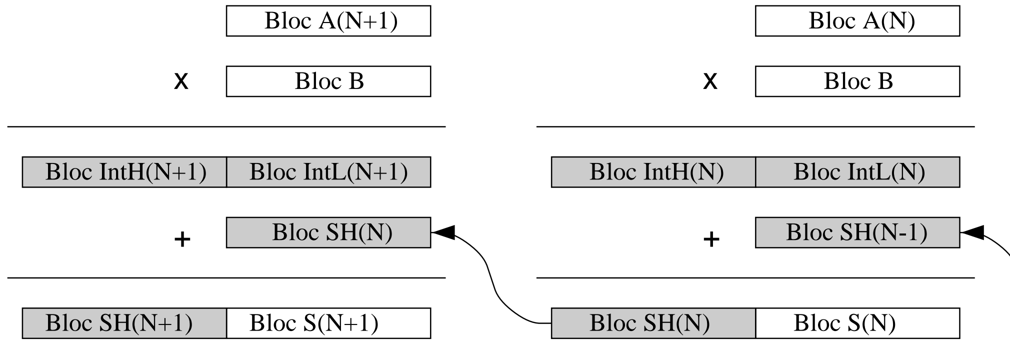


FIG. 3.3: Principe d'une multiplication sur un tableau.

3. **Division**

   – Division sans ajustement du reste. Cette opération effectue la division entre un bloc double (le dividende) et un bloc (le diviseur). Le bloc de poids fort du dividende doit être inférieur au diviseur. Cela permet d'utiliser cette instruction pour les divisions sur les tableaux (Figure 3.4). Elle retourne un quotient sur un bloc et un reste sur un bloc.

   – Division avec ajustement du reste. Cette opération est identique à la précédente mais elle lit une information supplémentaire qui est utilisée comme référence pour ajuster le signe du reste. Cette instruction sert pour ajuster le signe du reste d'une division sur un tableau avec le signe du bloc de poids fort du tableau. Elle retourne un quotient sur un bloc et un reste sur un bloc.

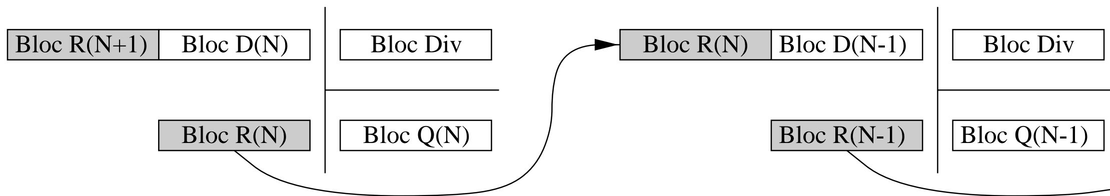


FIG. 3.4: Principe d'une division sur un tableau.

4. **Décalage, normalisation**

   – Décalage à gauche d'un bloc.

   – Décalage à droite d'un bloc.

   Les opérations de décalage à gauche ou à droite nécessitent de lire trois blocs : celui qui doit être décalé, celui qui contient la valeur du décalage et celui qui contient la valeur avec laquelle on pousse. Ce troisième bloc permet de réaliser les opérations de décalage sur les tableaux car on peut ainsi décaler avec la valeur du bloc adjacent et pas seulement avec des 0 ou des 1 (Figure 3.5). Ces opérations retournent un résultat sur un bloc qui correspond à la valeur décalée du premier bloc.

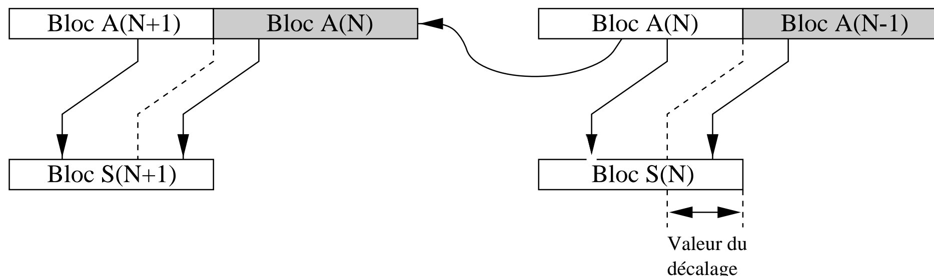


FIG. 3.5: Principe d'un décalage sur un tableau.

- Décalage arithmétique d'un bloc. Cette instruction, qui effectue un décalage à droite, utilise le bit de signe du bloc à décaler comme valeur pour les bits qui poussent. L'instruction n'effectue donc pas la lecture du troisième bloc. Elle retourne un résultat sur un bloc.
- Normalisation d'un bloc. Cette instruction effectue le décalage à gauche d'un bloc de manière à supprimer tous les bits de poids fort qui ne correspondent qu'à une extension de son bit de signe. La valeur du décalage est déterminé automatiquement par le matériel. Cette instruction lit le bloc à normaliser ainsi que le bloc qui pousse. Elle retourne comme résultat : le bloc normalisé et le bloc qui contient la valeur du décalage réalisé.

5. **Chargement, rangement**

   - Chargement d'un bloc. Cette instruction permet de charger dans un registre un bloc provenant de la mémoire. On utilise un mode d'adressage indirect par registre avec déplacement signé.
   - Chargement étendu. Cette instruction permet de charger un mot de 32 bits depuis la mémoire dans un bloc (128 bits) en faisant l'extension de son bit de signe (bit 31) dans le reste du bloc (bits 32 à 127). Le mot de 32 bits doit être aligné dans la mémoire avec les blocs (128 bits).
   - Chargements d'un immédiat. Dans ces instructions, l'immédiat qui provient du décodage de l'instruction a une taille maximum de 32 bits. Il y a quatre instructions qui permettent de charger cet immédiat dans un bloc en l'alignant sur les poids 0, 32, 64 ou 96 du bloc. Une cinquième instruction permet de charger l'immédiat dans les poids faibles du bloc (bits 0 à 31) et de faire l'extension de son bit de signe (bit 31) dans le reste du bloc (bits 32 à 127).
   - Rangement d'un bloc. Cette instruction permet de ranger dans la mémoire un bloc provenant d'un registre. On utilise un mode d'adressage indirect par registre avec déplacement signé.

6. **Instructions de contrôle**

   - Branchements conditionnels avec immédiat. Ces instructions testent la condition entre deux blocs et effectuent ou non le branchement. Le déplacement est relatif au compteur de programme. La valeur du déplacement est signée et correspond à l'immédiat. Il y a six conditions qui peuvent

être détectées : égalité, différence, supérieur, supérieur ou égal, inférieur, inférieur ou égal.
   - Branchements conditionnels avec registre. Ces instructions sont identiques aux précédentes mais avec une valeur de déplacement contenue dans un registre.
   - Saut avec immédiat. Cette instruction effectue un déplacement relatif du compteur de programme avec la valeur de l'immédiat.
   - Saut avec registre. Cette instruction effectue un déplacement relatif du compteur de programme avec une valeur contenue dans un registre.
   - Saut avec immédiat et lien. Cette instruction effectue un déplacement relatif du compteur de programme avec la valeur de l'immédiat. L'adresse de retour est mémorisée dans un registre.
   - Saut avec registre et lien. Cette instruction effectue un déplacement relatif du compteur de programme avec une valeur contenue dans un registre. L'adresse de retour est mémorisée dans un registre.

7. **Instructions de transfert**
   - Transfert bloc - bloc. Cette instruction déplace un bloc d'un registre dans un autre registre.
   - Dans les chapitres suivants, on verra que des informations utilisées par les opérateurs pour accélérer les calculs sont stockées dans des registres de contrôle et des registres de longueur. Les instructions suivantes permettent de transférer des informations entre ces registres et les registres contenant les blocs.
   - Transfert bloc - registre de contrôle.
   - Transfert bloc - registre de longueur.
   - Transfert registre de contrôle - bloc.
   - Transfert registre de longueur - bloc.

3.2.2 Accès au banc de registres

Pour réaliser efficacement des calculs sur les tableaux, les opérations élémentaires sur les blocs doivent effectuer un nombre important de lectures et d'écritures. Celles-ci sont nécessaires pour conserver la continuité des calculs lorsque l'on passe d'un bloc au suivant dans un tableau. Le tableau 3.1 donne le nombre de lectures et d'écritures utilisées pour effectuer différentes opérations élémentaires ainsi que pour effectuer ces opérations sur des opérandes de N blocs à partir de l'opération élémentaire sur un bloc.

Par rapport aux processeurs actuels, ce nombre élevé de lectures et d'écritures a des conséquences sur l'architecture et les performances du processeur. En particulier, il accroît considérablement la taille du banc de registre car il nécessite plus d'accès simultanés. Il augmente aussi la complexité du contrôle qui doit tester un nombre plus important de dépendance lors du décodage. Cela augmente le temps, généralement critique, de ce module.

	Opération élémentaire		Opération sur N blocs	
	Lectures	Ecritures	Lectures	Ecritures
Addition/Soustraction	3	2	3 N	2 N
Multiplication	2	2	2 N	2 N
Multiplication-accumul.	3	2	3 N	2 N
Division	3	2	3 N	2 N
Décalage	3	1	3 N	N
Normalisation	2	2	2 N	2 N

Tab. 3.1: Nombre d'accès au banc de registre pour les opérations élémentaires.

Pour éviter tout accroissement important de la taille du banc de registre, il faut limiter le nombre d'accès simultanés disponibles. Cela se traduit part une augmentation du nombre de cycle pour les opérations qui devront alors effectuer certains accès séquentiellement.

On remarque cependant que pour chaque opération, lorsque l'on effectue des calculs sur des blocs consécutifs, des paramètres peuvent être transmis directement d'une opération à la suivante sans être stockés dans le banc de registres ce qui en diminue les accès.

- Pour l'addition et la soustraction, la retenue sortante peut être réinjectée directement en tant que retenue entrante de l'opération sur le bloc suivant.
- Pour la multiplication et la multiplication-accumulation, le bloc de poids fort peut être additionné directement avec le bloc de poids faible du résultat de la multiplication sur le bloc suivant.
- Pour la division, le reste peut être réinjecté en tant que bloc de poids fort du dividende de la division sur le bloc inférieur.
- Pour le décalage, le bloc qui sert à pousser peut être réutilisé directement car c'est le bloc qui doit être décalé dans l'opération suivante.

On propose d'ajouter dans le jeu d'instruction du processeur des instructions sur les tableaux. Elles permettent de diminuer :

- Les accès en lecture et en écriture car elles transférent directement les paramètres d'une opération à la suivante.
- Les accès en lecture car elles suppriment des lectures identiques. C'est le cas par exemple pour les multiplications. Lorsque l'on effectue la multiplication de tous les blocs d'un tableau par un bloc B, ce bloc B est lu à chaque opération élémentaire sur un nouveau bloc du tableau. Les instructions sur les tableaux n'effectuent la lecture du bloc B qu'une seule fois au début car ensuite il reste identique pendant toute l'exécution sur le tableau. Ceci est aussi valable pour la division avec le diviseur et le décalage avec la valeur du décalage.

Le tableau 3.2 donne le nombre d'accès en lecture et en écriture utilisés pour effectuer différentes opérations sur des opérandes de N blocs dans le cas de l'enchaînement de N opérations élémentaires sur les blocs et dans le cas d'une opération sur des tableaux.

Ces instructions sur les tableaux, présentées dans le paragraphe 3.2.3, permettent d'augmenter la performance des opérations enchaînées sur des blocs consécutifs, en diminuant les accès au banc de registre et en simplifiant le contrôle.

	Opérations élémentaires		Opération sur les tableaux	
	Lectures	Ecritures	Lectures	Ecritures
Addition/Soustraction	3 N	2 N	2 N + 1	N + 1
Multiplication	2 N	2 N	N + 1	N + 1
Multiplication-accumul.	3 N	2 N	2 N + 1	N + 1
Division	3 N	2 N	N + 2	N + 1
Décalage	3 N	N	N + 2	N
Normalisation	2 N	2 N	N + 1	N + 1

Tab. 3.2: Nombre d'accès au banc de registre dans le cas d'opérandes de N blocs.

3.2.3 Instructions sur les tableaux

Une instruction sur un tableau va effectuer l'enchaînement sur tous les blocs du tableau, de l'opération élémentaire sur un bloc. Pour réaliser efficacement les enchaînements, le matériel possède les caractéristiques suivantes :

- Un banc de registres vectoriels qui permet d'accéder rapidement à des registres consécutifs. Ce banc est implémenté sous forme d'une matrice où les lignes représentent les tableaux. La taille de ces lignes est un compromis entre le coût matériel et les performances. En effet, plus la taille est importante, plus le coût matériel augmente mais plus le nombre de cycle moyen, pour calculer un bloc du tableau, diminue. Cette dernière remarque doit cependant être nuancée car une taille trop élevée accroît la complexité du contrôle, lequel augmente la durée du cycle d'horloge. Le banc de registre que l'on utilise est donc constitué de 16 registres vectoriels de 16 blocs de 128 bits chacun. Les instructions sur les tableaux étant réalisées à partir des instructions élémentaires, il est donc possible d'enchaîner facilement des instructions sur les tableaux pour traiter des nombres plus grands.
- Un transfert d'informations d'une opération sur un bloc à l'opération sur le bloc suivant. Cette fonction est câblée directement au niveau des opérateurs et permet de diminuer les accès au banc de registre.
- Une exécution des instructions sur des tableaux de taille variable. Cette propriété permet d'obtenir une efficacité maximale pour les opérations sur les tableaux. A chaque tableau du banc de registre est associé un registre de longueur de 4 bits qui contient la taille réelle, en nombre de blocs, du tableau. Cela permet d'exécuter une instruction uniquement sur les blocs valides du tableau et non pas systématiquement sur sa longueur maximale.

Au niveau des instructions, la taille du tableau est transmise comme paramètre. Elle indique au matériel le nombre de blocs sur lesquels doit être enchaînée l'opération

élémentaire. Il y a deux modes d'adressage pour cette longueur : soit avec un immédiat, soit avec un registre de longueur. Cela permet au programmeur d'avoir plus de liberté pour l'écriture de ses programmes. Les instructions sur les tableaux, qui ont été ajoutées au jeu d'instructions du processeur, sont présentées après. Il convient de noter que chacune de ces instructions existe en double puisqu'elle possède deux types d'adressage pour la longueur des tableaux.

- Addition de deux tableaux. Cette instruction retourne un résultat sur un tableau.
- Addition de deux tableaux et d'un bloc pour la retenue entrante. Le résultat est un tableau et un bloc supplémentaire pour la retenue sortante.
- Soustraction de deux tableaux. Cette instruction retourne un résultat sur un tableau.
- Soustraction de deux tableaux et d'un bloc pour la retenue entrante. Le résultat est un tableau et un bloc supplémentaire pour la retenue sortante.
- Multiplication d'un tableau par un bloc. Le résultat est un tableau et un bloc supplémentaire pour le dernier bloc poids fort du résultat.
- Multiplication d'un tableau par un bloc et accumulation avec un deuxième tableau. Le résultat est un tableau et un bloc supplémentaire pour le dernier bloc poids fort du résultat.
- Décalage à gauche d'un tableau. Le résultat est un tableau.
- Décalage à droite d'un tableau. Le résultat est un tableau.
- Décalage arithmétique d'un tableau. Le résultat est un tableau.
- Normalisation d'un tableau. Le résultat est un tableau.
- Chargement d'un tableau depuis la mémoire.
- Rangement d'un tableau dans la mémoire.
- Transfert d'un tableau dans un autre tableau.

L'instruction de division sur un tableau n'a pas été implémentée car elle n'apportait qu'un gain en temps très faible pour une augmentation importante de sa complexité. Cela provient du fait que la division est une opération dont le temps d'exécution est long et pour laquelle le nombre de cycles nécessaires pour les accès en lecture et en écriture est négligeable par rapport au nombre de cycles du calcul.

3.2.4 Gestion dynamique des débordements

En arithmétique virgule flottante, il n'y a qu'un seul format pour coder les nombres. Ceux-ci sont tronqués de façon à être écrit dans un format identique pour tous. En arithmétique exacte, toutes les opérations ne retournent pas des résultats dans le même format puisque l'intégralité des bits est conservée. On considère généralement qu'il y a un débordement lorsqu'une opération retourne un résultat dans un format qui est différent de celui de ses entrées. Si on se réfère au tableau 3.2, on constate que c'est le cas pour deux types d'opérations : les additions/soustractions et les multiplications/multiplications-accumulations qui retournent des résultats sur N+1 blocs alors que leurs entrées ont N blocs. Les divisions et les normalisations

écrivent aussi N+1 blocs mais il n'y a pas de débordement car elles écrivent deux résultats, un sur N blocs (qui ne déborde pas) et un autre sur 1 bloc.

Le temps d'exécution d'une instruction sur un tableau dépend directement du nombre de lectures ou d'écritures qu'elle effectue. Pour les additions et les multiplications, qui ont des débordements, c'est le nombre d'écritures qui impose la durée de l'opération puisqu'il y a plus d'écritures que de lectures.

On remarque cependant que pour l'addition, le dernier bloc écrit ne contient pas toujours de l'information utile pour le codage du résultat mais seulement une extension du bit de signe. Dans ce cas, l'écriture de ce dernier bloc rallonge inutilement la durée de l'opération.

On propose d'utiliser une solution matérielle pour effectuer la gestion des débordements dans le cas de l'addition. Cette solution permet d'économiser l'écriture d'un bloc à chaque opération d'addition, y compris lorsqu'il y a réellement un débordement (retenue sortante) dans le dernier bloc.

Cette solution est basée sur le report de l'écriture du dernier bloc. Seule la longueur exacte du résultat sera mise à jour. En effet lors d'une opération, l'information devant être écrite dans un bloc du résultat n'a besoin d'être réellement mise à jour que lorsque l'on réutilise ce résultat, soit dans une nouvelle opération, soit pour l'écrire en mémoire. La solution matérielle n'écrira donc l'information de débordement, si elle existe, dans le dernier bloc que lors de la prochaine utilisation de ce bloc.

Pour cela, on associe à chaque bloc de 128 bits un registre de contrôle de 5 bits. Les informations contenues dans ces registres de contrôle sont utilisées par différentes opérations afin de s'exécuter plus efficacement. Dans le cas de la gestion des débordements, on écrit à chaque nouveau bloc du résultat, deux bits dans le registre de contrôle du bloc suivant. Le premier de ces deux bits (D) indique si il y a ou non un débordement sur le bloc suivant, le second bit (Vd) contient la valeur du débordement car les nombres étant signés, on peut avoir un 0 (nombre positif) ou un 1 (nombre négatif) comme retenue sortante. Lors de la prochaine utilisation de ce résultat, la longueur transmise à la nouvelle opération étant exacte, tous les blocs du tableau vont être lus. Pour le dernier bloc, le bit D du registre de contrôle associé commande des multiplexeurs qui aiguillent sur l'entrée de l'opérateur, soit la valeur du bloc qui a été lu (pas de débordement), soit la valeur du bit Vd (débordement). En effet, dans le cas d'un débordement, le dernier bloc lu contient n'importe quoi puisque la valeur de la retenue sortante n'avait pas été écrite dans ce bloc. Lorsque D indique un débordement, la valeur du bit Vd qui correspond au bit de signe, sera étendue à tous les bits du bloc en entrée de l'opérateur (Figure 3.6).

Pour la multiplication, la probabilité que le résultat soit codé sur N+1 blocs est très importante. Dans ce cas, on réalisera l'écriture systématique du bloc N+1 car elle ne coûte pas de cycles supplémentaires. De plus, l'utilisation de la solution retenue pour l'addition serait beaucoup trop coûteuse en place car il faudrait dans ce cas mémoriser non pas un bit mais un bloc, puisque la multiplication écrit dans tous les bits du bloc N+1.

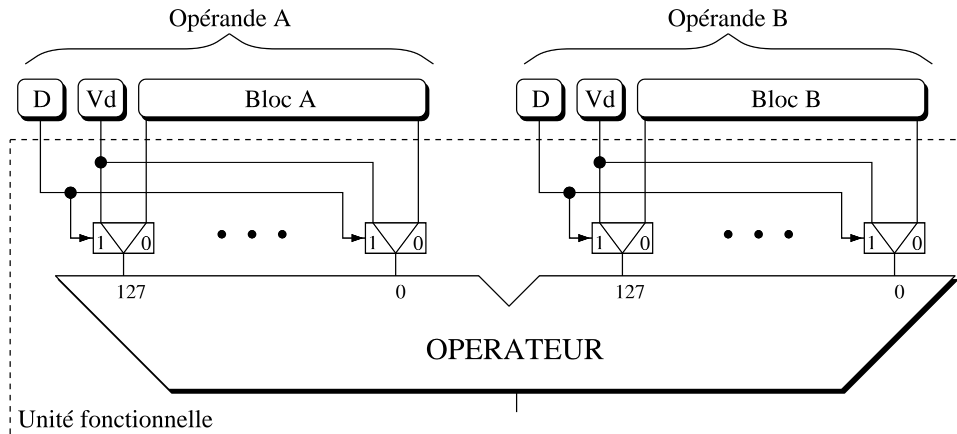


FIG. 3.6: Gestion dynamique des débordements pour l'addition.

3.3 Architecture générale du cœur du processeur

L'architecture générale du cœur du processeur a été développée pour enchaîner efficacement les calculs sur les blocs afin de diminuer la durée des opérations sur des grands nombres.

Après une étude des principales architectures existantes, on présente la solution que l'on a implantée pour le cœur du processeur. Toutes les caractéristiques ou propriétés décrites dans les paragraphes précédants ont été intégrées dans cette architecture car elles sont essentielles pour l'obtention de bonnes performances en calcul exact.

On présentera enfin plus en détail deux caractéristiques importantes de l'architecture du circuit qui sont la hierarchie mémoire et la structure des pipelines.

3.3.1 Les principales architectures

Dans cette section, on présente les différentes architectures qui existent dans les processeurs actuels. Cette liste n'est pas exhaustive, il s'agit seulement de présenter les principales caractéristiques architecturales[4] ainsi que les avantages et les inconvénients qui en découlent. On peut noter aussi que ces caractéristiques ne sont pas toutes exclusives et que certaines peuvent être associées pour donner de nouvelles architectures.

3.3.1.1 CISC

Les processeurs CISC (Complex Instruction Set Computer) [19][76] sont caractérisés par un grand jeu d'instructions qui comprend de 120 à 350 instructions pour les processeurs les plus courants. Ces instructions sont complexes et nécessitent entre 2 et 15 cycles pour être exécutées. Le jeu d'instructions utilise plusieurs formats d'instructions et un nombre élevé de mode d'adressage. Les instructions sont codées sur

[4]Pour plus d'informations concernant les architectures récentes, on peut se référer au site http ://www.irisa.fr/caps/PROJECTS/TechnologicalSurvey

des formats qui vont de 16 bits à 64 bits. Le nombre de mode d'adressage est généralement compris entre 12 et 24. Ces architectures n'intègrent qu'un faible nombre de registres (de 8 à 24) car elles autorisent les références mémoires au niveau des opérations.

Il en résulte un décodage et un contrôle très complexes. Dans les premières générations de processeurs CISC, le contrôle était microprogrammé (ROM) ce qui avait un effet désastreux sur le temps de cycle. Dans les processeurs plus récents, c'est un contrôle câblé qui est utilisé. Cela ne permet cependant pas de résoudre totalement le principal inconvénient de ce type d'architecture qui est un contrôle trop complexe, lequel impose une fréquence de fonctionnement faible. Les instructions étant coûteuses en cycles, les performances de ces architectures restent faibles.

Elles ont cependant l'avantage de simplifier les compilateurs car elles offrent un large choix d'adressage ainsi que des instructions complexes qui effectuent plusieurs taches et réduisent les lignes de code.

Les architectures CISC ont été développées jusqu'a la fin des années 80 mais semblent aujourd'hui condamnées à disparaître des prochaines générations de processeurs.

3.3.1.2 RISC

Les processeurs RISC (Reduced Instruction Set Computer) [50][46] ont été développés dans le but de réduire le temps de cycle en diminuant la complexité de l'architecture. En effet, l'étude des traces des programmes ont montré que durant 95% du temps d'exécution, seulement 25% des instructions disponibles dans une architecture CISC étaient utilisées [33]. Cela a conduit les concepteurs à réaliser des architectures plus simples de manière à accélérer les opérations couramment utilisées au dépend de celles qui l'étaient peu.

Le jeu d'instruction d'un processeur RISC comprend moins de 150 instructions. Celles-ci utilisent toutes le même format (sur 32 bits) et effectuent seulement des taches élémentaires. Les opérations plus complexes sont réalisées au niveau logiciel au moyen de plusieurs instructions. Toutes les opérations sont de type registre-registre ce qui nécessite un banc de registre plus important que dans les CISC. Il comprend entre 32 et 150 registres. Seules deux instructions (load et store) permettent les échanges avec la mémoire.

Ces architectures utilisent seulement entre 3 et 5 modes d'adressage. Le contrôle est alors simplifié et peut être facilement intégré dans le circuit. De plus, l'uniformité des opérations a permis d'utiliser efficacement la technique du pipeline ce qui permet d'obtenir un nombre de cycle par instruction (CPI) proche de un (Figure 3.7).

Les fréquences de fonctionnement des RISC sont plus élevées que celles des CISC du fait d'un contrôle simplifié et de l'utilisation d'unités pipelinées. C'est ce type d'architecture qui est implanté dans les processeurs actuels et qui a progressivement remplacé les CISC.

Les RISC possèdent néanmoins un inconvénient qui est le goulot d'étranglement que constitue les transferts de données avec la mémoire, puisque ceux-ci ne peuvent être effectués que par deux instructions spécialisées.

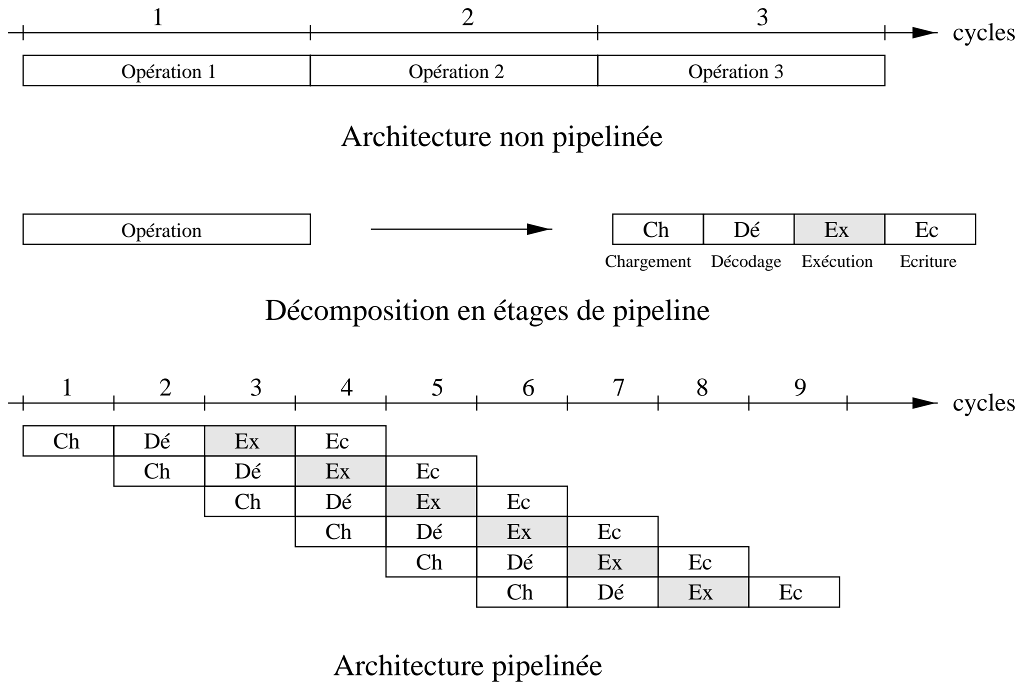


FIG. 3.7: Principe du pipeline.

3.3.1.3 Superscalaire

Le terme superscalaire correspond plus à une caractéristique qui peut s'appliquer aussi bien aux CISC qu'aux RISC, plutôt qu'à une architecture à part entière.

Dans les architectures CISC ou RISC , une seule instruction pouvait être lue, décodée puis exécutée par cycle. On obtenait donc au maximum qu'un seul résultat par cycle. Dans les architectures superscalaires, plusieurs instructions peuvent être décodées et lancées à chaque cycle (Figure 3.8). C'est le matériel qui détecte dynamiquement quelles sont les instructions qu'il va pouvoir exécuter simultanément. Les architectures RISC récentes peuvent lancer entre 2 et 5 instructions par cycle ce qui permet d'avoir un nombre de cycle par instruction (CPI) inférieur à un.

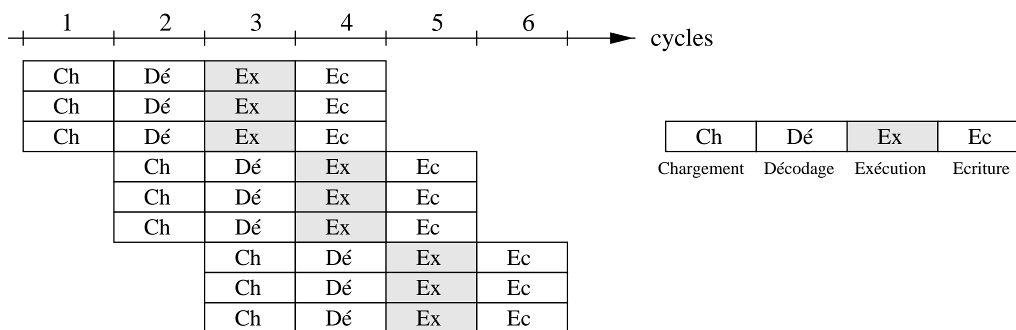


FIG. 3.8: Architecture superscalaire.

Le goulot d'étranglement pour les transferts avec la mémoire est encore plus critique dans ce type d'architecture. Pour diminuer ce problème, le banc de registre contient encore plus de registres que celui d'une architecture classique. L'utilisation

de mémoires caches pour les instructions et les données permet aussi de diminuer le coût des transferts avec la mémoire.

Les processeurs RISC superscalaires [57][18][20] font partie des plus performants qui existent. Cependant cette architecture n'est pas indéfiniment extensible car lancer un grand nombre d'instructions en parallèle complique le contrôle et donc ralentit la fréquence de fonctionnement. De plus, ces architectures exploitent le parallélisme au niveau des instructions. Il faut cependant qu'il n'y ait pas de dépendances entre les instructions qui sont lancées simultanément car sinon certaines ne pourront pas être lancées. Ces suspensions annulent l'avantage que peut procurer une architecture fortement superscalaire car bien souvent elle ne sera pas utilisée à 100% de ses capacitées.

3.3.1.4 Vectorielle

Les architectures vectorielles [8][5] permettent d'enchaîner des opérations identiques sur tous les éléments d'un vecteur (ou tableau). Ces architectures utilisent au maximum le parallélisme au niveau des données. L'indépendance entre les éléments d'un vecteur et la répétition des opérations sur tous les éléments de ce vecteur permettent d'utiliser les pipelines avec une grande efficacité (Figure 3.9). On obtient alors un CPI très proche de un. Dans le cas d'architectures vectorielles superscalaires, plusieurs unités vectorielles peuvent alors travailler simultanément ce qui diminue considérablement le CPI.

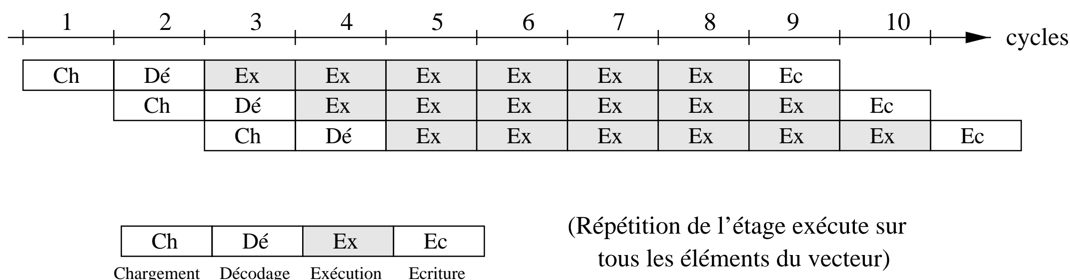


FIG. 3.9: Architecture vectorielle.

L'avantage de ces architectures est leur rapidité lorsque les vecteurs sont grands. Elles réduisent aussi le goulot d'étranglement au niveau de la mémoire car une instruction correspond à plusieurs cycles d'exécution. Leur inconvénient provient du fait qu'il faut trouver des problèmes à résoudre où l'on a beaucoup de parallélisme afin d'avoir des vecteurs de grande taille.

Ces architectures, si elles ont fait leurs preuves au niveau des performances, sont cependant complexes et coûteuses à développer pour un marché relativement restreint. Elles ont tendance à être remplacées par des architectures RISC superscalaires qui offrent des performances élevées pour un champ d'applications plus vaste.

3.3.1.5 VLIW

Une architecture VLIW (Very Long Instruction Word) [24][47] utilise un mot d'instructions de plusieurs centaines de bits qui contient le codage de plusieurs instructions qui seront décodées puis exécutées simultanément. Contrairement aux ar-

chitectures superscalaires, la détection du parallélisme au niveau des instructions se fait de façon statique lors de la compilation.

Une architecture VLIW est composée de plusieurs unités fonctionnelles qui travaillent en parallèle. Un mot d'instruction, codé sur un nombre de bits généralement compris entre 256 et 1024, contient plusieurs champs qui correspondent chacun à une unité fonctionnelle. A chaque champ est associé un bit de validité qui indiquera si une opération a pu ou non être placée dans ce champ lors de la compilation. La détection du parallélisme n'étant pas faite dynamiquement, le contrôle est alors extrêmement simple. Cela permet de mettre aisément un grand nombre d'unités fonctionnelles pipelinées en parallèle (au moins 6 ou 7) et d'obtenir autant de résultats à chaque cycle, réduisant fortement le CPI (Figure 3.10).

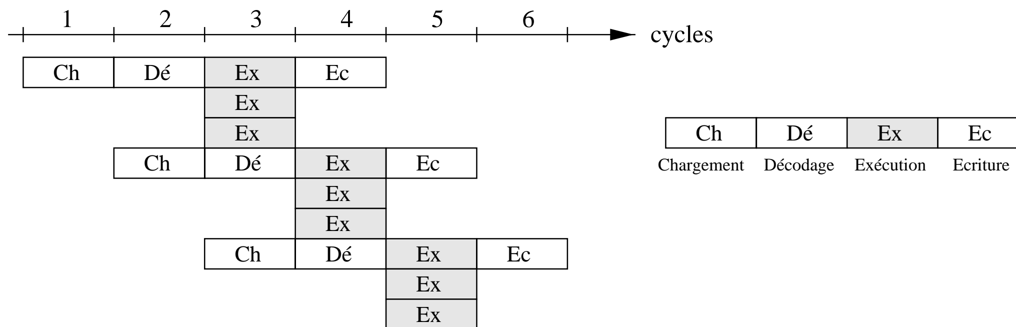


FIG. 3.10: Architecture VLIW.

L'avantage de ces architectures, par rapport à celles superscalaires, vient du fait que l'on peut mettre un nombre élevé d'unités en parallèle sans augmenter la complexité du contrôle. Elles possèdent en revanche deux inconvénients majeurs :

- elles sont très dépendantes des performances du compilateur et des stratégies concernant les branchements. Les architectures VLIW ne seront vraiment performantes que dans l'exécution de programmes où les branchements seront prévisibles.
- Le compilateur est très lié à l'architecture puisqu'il doit tenir compte en particulier de la profondeur des pipelines afin de synchroniser, dans la même instruction VLIW, les différentes instructions du programme. Cela signifie qu'un compilateur n'est pas compatible avec toutes les réalisations matérielles d'une même architecture VLIW puisqu'il doit tenir compte de particularités propres à la machine sur laquelle il travaille.

3.3.2 Architecture retenue

L'architecture générale du cœur du processeur est présentée figure 3.11. Elle se décompose en deux parties : la première concerne les instructions, la seconde les données. La partie instruction est constituée principalement par :

- Un cache instructions.
- Un module de décodage dans lequel est réalisé le décodage de l'instruction, la détection des dépendances et l'allocation des ressources, l'estimation de longueur du résultat, la détection des chaînages.

– Un module de génération d'adresse pour le compteur de programme.

La partie donnée est pour sa part principalement constituée par :

– Un banc de registre vectoriel constitué de 16 tableaux de 16 blocs de 128 bits.
– Cinq unités vectorielles qui peuvent enchaîner des opérations sur les blocs consécutifs des tableaux du banc de registre.
– Un banc de registre pour les registres de contrôle.
– Une table des longueurs qui contient la taille de chaque tableau du banc de registre vectoriel.

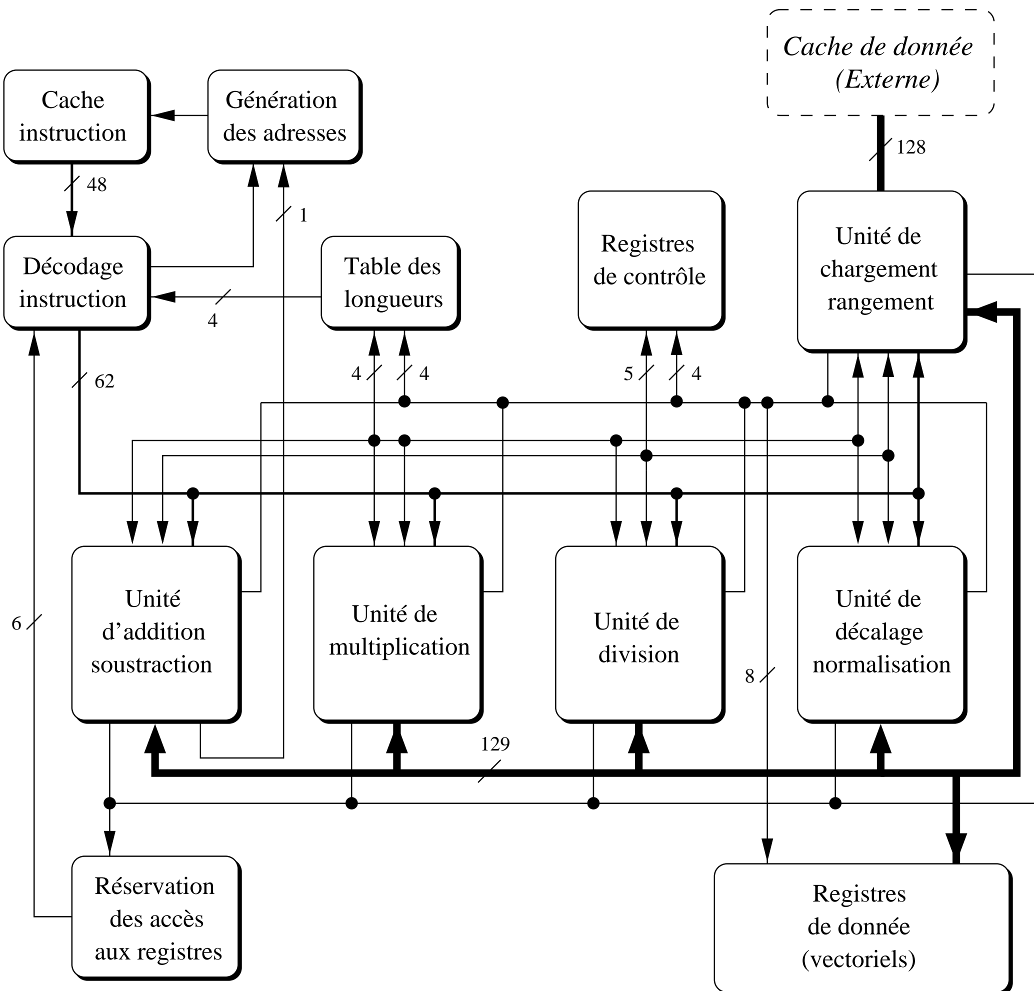


FIG. 3.11: Architecture générale du cœur du processeur.

L'architecture du processeur est de type RISC. Elle utilise des instructions qui sont codées sur 48 bits. Contrairement aux processeurs récents qui cherchent à lancer plusieurs instructions simultanément pour augmenter leurs performances [74][32][57], l'architecture présentée ne peut transférer qu'une seule instruction par cycle depuis le cache instruction vers le décodage. Dans le cas de notre processeur, une architecture superscalaire n'apporte qu'une faible augmentation des performances pour un accroissement important de la complexité du système. En effet, la majorité des instructions étant multi-cycles, avec des durées d'exécution variables d'une instruction

à l'autre, cela entraîne une libération des unités de façon quelconque dans le temps. Il est donc rare de pouvoir synchroniser le lancement de deux nouvelles instructions dans le même cycle. Malgré le lancement d'une seule instruction par cycle, l'architecture permet un fonctionnement parallèle de plusieurs unités fonctionnelles car des opérations multi-cycles lancées successivement (une par cycle) peuvent continuer de s'exécuter simultanément pendant plusieurs cycles dans des unités différentes. On obtient alors une architecture dont le CPI est similaire à celui des processeurs superscalaires.

Le module qui effectue le décodage des instructions détecte ensuite si l'instruction peut être réellement exécutée (unité disponible, ressources matérielles suffisantes, pas de conflit d'adresse,...) ou si elle doit être suspendue. Une description plus précise de ces étapes est faite dans le chapitre 4. Lorsqu'une instruction est lancée, tous les paramètres nécessaires à son exécution sont transmis à l'unité fonctionnelle correspondante. L'architecture générale est de type RISC avec cinq unités fonctionnelles pipelinées pouvant travailler en parallèle. Ces unités, qui sont présentées plus en détail dans le chapitre 5, réalisent les opérations suivantes :

- Addition, soustraction.
- Multiplication, multiplication-accumulation.
- Division.
- Décalage, normalisation.
- Chargement, rangement.

Toutes ces unités ont une architecture vectorielle qui permet d'exécuter efficacement les opérations sur les tableaux. Le banc de registre vectoriel, qui contient les données, autorise 4 lectures et 2 écritures simultanément sur des données de la taille d'un bloc. Des bus de données de 128 bits relient le banc de registre vectoriel aux différentes unités. Celles-ci possèdent chacune un contrôleur d'unité spécifique qui leur permet de réaliser les opérations de façon autonome. A partir des informations transmises par le décodage, ce contrôleur gère les commandes des opérateurs et génère toutes les adresses dont l'unité a besoin pour accéder aux registres vectoriels, aux registres de contrôle et à la table des longueurs. C'est l'unité qui, lorsqu'elle a fini d'exécuter une instruction, transmet un signal de “fin d'opération” au module décodage pour lui indiquer qu'elle est de nouveau libre.

3.3.3 Hiérarchie mémoire

La hiérarchie mémoire du processeur [31] est composée de trois niveaux :

1. Le premier niveau correspond au banc de registres de donnée. Ce banc est composé de 256 registres de 128 bits chacun agencés selon une matrice de 16 tableaux de 16 registres. Le nombre élevé de registres pour ce banc est imposé non seulement par l'architecture qui exécute plusieurs opérations simultanément, mais aussi par le fait que la taille importante pour les données ne permet pas d'intégrer un cache de donnée sur le circuit. On utilise alors le banc de registres de donnée comme un premier niveau de mémorisation.

2. Le second niveau est composé d'un cache d'instructions à correspondance directe de 3 kilo-octets intégré sur le circuit. On verra dans le paragraphe 4.2.1 que les instructions sont codées sur 48 bits. Elles sont cependant alignées en mémoire sur des mots de 64 bits ce qui permet une manipulation plus facile par le processeur principal. La ligne de cache transférée lors des défauts de cache est de 12 octets. Lorsqu'une ligne de cache est chargée depuis la mémoire, on lit 16 octets en mémoire mais seulement 12 sont transférés puisque pour chaque instruction, 2 octets sont inutilisés.

   Le cache de donnée se situe en dehors du circuit car l'utilisation de blocs de 128 bits nécessite une dimension trop importante pour le cache. En effet, intégrer un cache ayant une taille trop faible entraînerait un nombre élevé d'échecs liés à la capacité ce qui diminuerait fortement les performances du système [73][27]. On utilise donc un cache externe de 256 kilo-octets, associatif par ensemble de 4 positions qui utilise l'algorithme *LRU* (Least Recently Used) pour le remplacement des lignes. La ligne de cache transférée lors des défauts de cache est de 512 octets ce qui permet de charger un tableau entier.

3. Le troisième niveau est composé de la mémoire implantée sur la carte. Elle contient les instructions qui sont alignées sur des mots de 64 bits ainsi que les données qui sont alignées sur des mots de 128 bits. La capacité maximum de cette mémoire, qui est imposée par l'adressage sur 32 bits du processeur, est de 4 giga-octets.

3.3.4 La structure des pipelines

L'architecture interne du processeur est composée de plusieurs pipelines [42][61] qui sont présentés figure 3.12. Les deux premiers étages sont commun à tous les pipelines. Ils correspondent aux étapes de chargement et de décodage des instructions. Lorsque l'instruction est autorisée à s'exécuter, elle est envoyée dans l'unité concernée. Chaque unité possède son propre pipeline. Ceux-ci sont tous construits sur le même modèle : ils possèdent un premier étage de lecture durant lequel l'unité lit les registres de donnée, de contrôle, et la table des longueurs. Le pipeline est ensuite composé de plusieurs étages d'exécution, entre 2 et 4 selon l'unité, qui correspondent aux opérateurs arithmétiques. Le dernier étage des pipelines correspond à l'écriture des résultats dans les registres. Tous les pipelines des unités sont indépendants et peuvent s'exécuter en parallèle.

Il faut cependant noter que la division est une opération séquentielle qui ne peut pas être pipelinée. Seuls l'ajustement du reste puis son recodage sont pipelinés dans le but de diminuer le temps d'exécution total de ces deux fonctions et permettre ainsi d'avoir un temps de cycle plus faible.

3.4 Conclusions

Ce chapitre a abordé les principales caractéristiques de l'architecture générale du processeur. Bien que celle-ci ne lance qu'une nouvelle instruction par cycle, elle autorise malgré tout l'éxécution simultanée de plusieurs opérations grâce à l'utilisation d'unités vectorielles autonomes pouvant travailler en parallèle. La structure

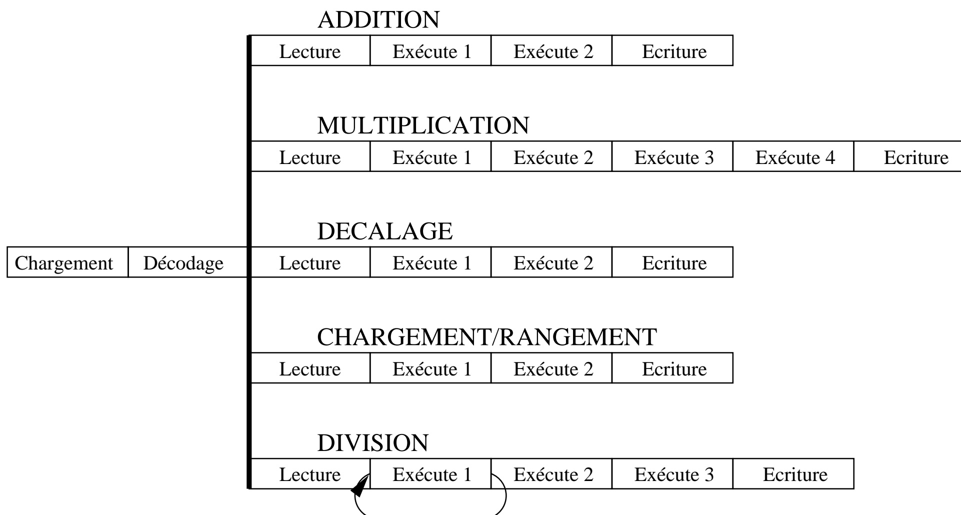


FIG. 3.12: Structure des pipelines dans le processeur.

vectorielle de ces unités et du banc de registres est particulièrement adaptée au traitement d'objets constitués de blocs consécutifs comme le sont les nombres dans le processeur. Cela permet d'effectuer ainsi efficacement des calculs sur des nombres de taille importante.

L'architecture du cœur du processeur intègre au niveau matériel la gestion dynamique des signes. Celle-ci est obtenue par l'intermédiaire d'opérateurs arithmétiques configurables qui peuvent exécuter des opérations sur des nombres au format non signé ou en notation complément à 2. Cette gestion dynamique des signes présente l'avantage, outre le fait qu'elle élimine une routine logicielle destinée à cet effet, de supprimer les comparaisons entre les deux nombres lors d'une addition ou d'une soustraction comme cela est le cas dans GIVARO.

Les "dépassements" éventuels de capacité qui peuvent survenir lors des additions, soustractions ou multiplications sont eux aussi traités dynamiquement. Ils passent ainsi inaperçu pour le programmeur. La solution utilisée pour les additions et les soustractions est basée sur le report de l'écriture du "dépassement" à la prochaine utilisation de ce résultat ce qui permet d'économiser un cycle à chaque opération, y compris lorsque le "dépassement" existe.

Le jeu d'instructions a aussi été présenté dans ce chapitre. Aux opérations élémentaires sur les blocs ont été ajoutées des opérations sur les tableaux qui permettent d'enchaîner les calculs sur des blocs consécutifs. Ces opérations possèdent des performances supérieures à celles obtenues si on avait utilisé successivement sur tous les éléments d'un tableau l'opération élémentaire sur les blocs, et ceci sans augmentation des ressources matérielles nécessaires (opérateurs arithmétiques, accès au banc de registre, etc...). L'implantation d'une table des longueurs qui contient la taille exacte de chaque tableau permet de minimiser la durée des opérations sur les tableaux. Cette longueur peut être transmise comme paramètre d'une instruction, soit sous forme d'immédiat, soit par l'intermédiaire des registres de longueur. Cela permet d'exécuter une instruction uniquement sur les blocs valides du tableau et non pas systématiquement sur sa longueur maximale.

Chapitre 4

Lancement d'une instruction

Ce chapitre décrit les mécanismes de lancement d'une instruction. Le lancement se décompose en deux étapes qui correspondent aux deux premiers étages communs dans les pipelines de la figure 3.12. La première étape correspond au chargement de l'instruction depuis le cache instruction en parallèle avec la mise à jour du compteur de programme (CP). Dans la seconde étape, il faut effectuer le décodage de l'instruction puis vérifier que l'ensemble des conditions nécessaires à l'exécution correcte de cette instruction sont validées.

4.1 Chargement d'une instruction

Le chargement d'une instruction, depuis le cache d'instructions, s'exécute dans le premier étage du pipeline. A chaque nouveau cycle d'horloge, on charge l'instruction sur laquelle pointe le compteur de programme (CP). On calcule en parallèle l'adresse de la prochaine instruction en incrémentant la valeur du compteur de programme. Les instructions comme les branchements conditionnels ou les sauts (branchements inconditionnels) effectuent des déplacements signés relatifs au compteur de programme. Pour ces instructions, la valeur du déplacement est soit un immédiat codé sur 20 bits, soit le contenu d'un registre de donnée. Dans ce dernier cas, la valeur du déplacement est codée sur 32 bits ce qui permet de balayer tout l'espace adressable. On utilise un additionneur sur 32 bits pour ajouter la valeur signée du déplacement à la valeur du compteur de programme. Pour les branchements conditionnels et les sauts, la nouvelle adresse ne peut pas être calculée en parallèle avec le chargement de l'instruction, ce qui entraîne des cycles de suspension dans le pipeline. Le nombre de cycles de suspension varie selon les instructions :

- Saut avec immédiat : La valeur de l'immédiat n'est disponible que lors du cycle de décodage. On a donc une suspension de 1 cycle.
- Saut avec registre : Le registre est lu dans le $3^{ème}$ cycle du pipeline. La nouvelle valeur du compteur de programme n'est calculée que durant le $4^{ème}$ cycle. On a donc dans ce cas 3 cycles de suspension.
- Branchements conditionnels : Le résultat de la comparaison est obtenu à la fin du $5^{ème}$ étage du pipeline de l'addition. La nouvelle valeur du compteur de programme, qui dépend si le branchement est pris ou pas pris, est alors calculée

durant le 6ème cycle. Cependant on peut anticiper le calcul des deux adresses de destination du branchement. Cela permet de pouvoir sélectionner la bonne valeur du compteur de programme dès que la comparaison est valide. On gagne ainsi un cycle de suspension. Les branchements conditionnels n'entraînent alors que 4 cycles de suspension.

Pour réaliser les cycles de suspension, on utilise une instruction NOP (instruction qui ne fait aucune opération) placée après chaque instruction de saut ou de branchement. Les instructions de ce type, lorsqu'elles sont décodées, commandent un module de "contrôle du CP". Celui-ci fait pointer le compteur de programme sur l'opération NOP autant de cycle que le nécessite l'instruction en cours, puis charge la bonne valeur du compteur de programme. C'est ce module qui permet aussi d'anticiper le calcul des deux adresses de destination possible lors d'un branchement (selon qu'il sera pris ou non) en attendant de connaître le résultat de la comparaison. Le schéma de l'ensemble du système qui permet de mettre à jour le compteur de programme est donné figure 4.1.

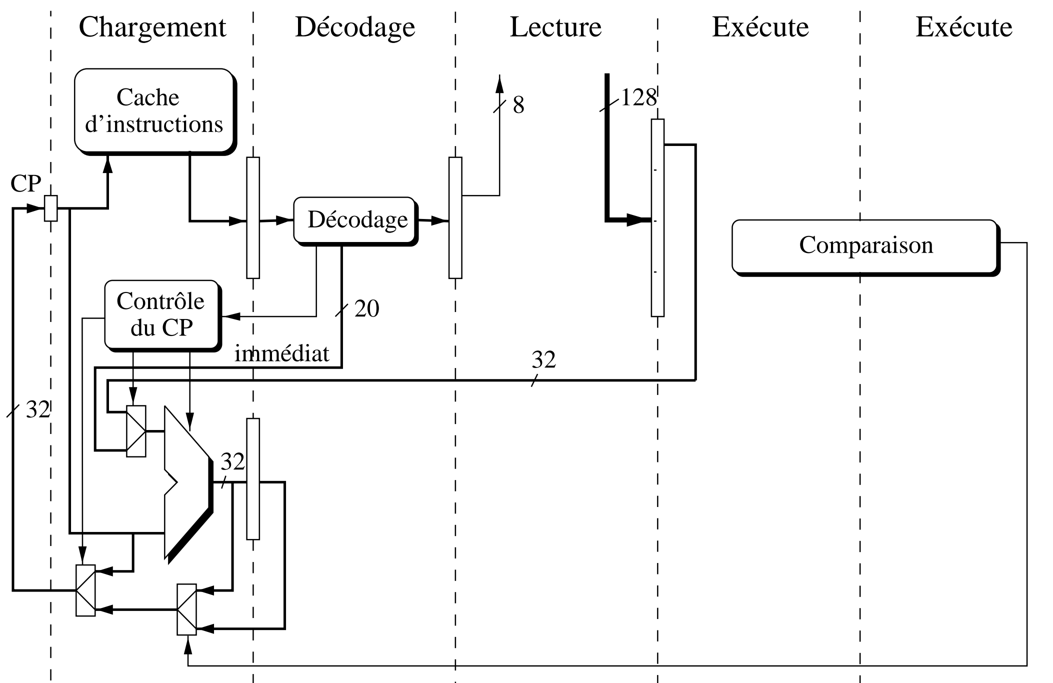


FIG. 4.1: Détermination de la valeur du compteur de programme CP.

4.2 Décodage d'une instruction

Le décodage d'une instruction correspond au deuxième étage du pipeline. Cet étage réalise non seulement le décodage du code opération et des différents champs d'adressage contenus dans une instruction, mais il effectue aussi la détection des dépendances de données et l'allocation des ressources matérielles qui déterminent si une opération peut être lancée ou non. C'est durant cette étape que s'effectue la mise à jour des informations qui doivent être transmises à l'unité fonctionnelle lorsque la nouvelle opération est autorisée à être lancée.

Cet étage est souvent critique en temps dans les processeurs c'est pourquoi il faut chercher à le minimiser. C'est principalement le décodage des différents champs d'adressage dans l'instruction, suivi de la détection des dépendances de données, qui génèrent un délai important. Pour le minimiser, il faut simplifier le décodage des champs en utilisant le plus souvent possible le même type de codage pour plusieurs instructions.

4.2.1 Types des instructions

Le banc de registre pour les données est organisé sous forme d'une matrice de 16 tableaux (lignes) de 16 blocs chacun (colonnes). Pour réaliser l'adressage d'un bloc, on utilise donc un champ d'adresse de 8 bits, qui est décomposé en deux adresses de 4 bits : les 4 bits de poids forts pour adresser le numéro du tableau, les 4 bits de poids faibles pour adresser la position du bloc à l'intérieur du tableau.

Pour l'adressage d'un tableau, il faut transmettre deux informations : l'adresse du tableau et sa longueur. On utilise le même champ d'adresse de 8 bits que pour l'adressage des blocs ce qui simplifie fortement le décodage puisque tous les champs restent alignés. Les 4 bits de poids forts de ce champ contiennent le numéro du tableau, les 4 bits de poids faibles contiennent la longueur du tableau. L'adressage de la longueur peut être de deux types : soit un immédiat, soit l'adresse d'un registre de longueur (Figure 4.2).

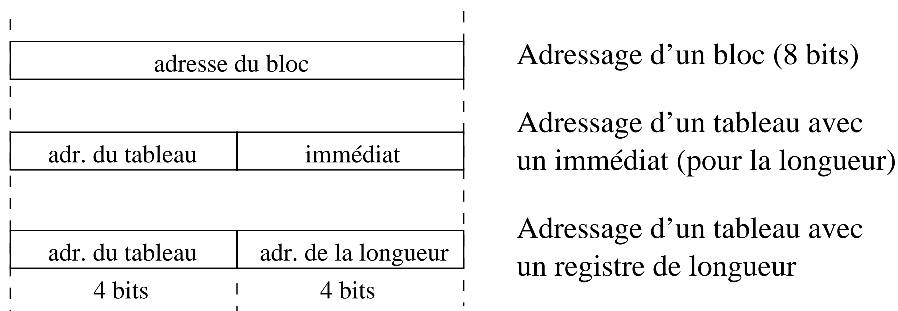


FIG. 4.2: Adressage sur 8 bits des blocs et des tableaux.

On a vu dans le paragraphe 3.2.2 que les instructions effectuent un nombre élevé de lectures et d'écritures. Cela se traduit au niveau du codage des instructions par autant de champs d'adressage. La taille d'un champ étant de 8 bits pour un bloc ou un tableau, le format classique de 32 bits pour les instructions ne permet de coder que 3 champs en plus du code opération ce qui le rend insuffisant pour la majorité des instructions.

Le format utilisé pour le codage des instructions dans le processeur est donc de 48 bits ce qui permet d'intégrer au moins 4 champs dans une même instruction. Ce format permet aussi d'écrire directement des immédiats de 32 bits dans une instruction ce qui est un avantage lorsque l'on travaille avec des blocs de 128 bits. Le code opération est contenu dans les 12 premiers bits de l'instruction, les 36 bits suivants contiennent les différents champs d'adresse.

Afin de minimiser le temps de décodage des instructions, il faut minimiser le nombre de formats différents pour le codage en associant plusieurs instructions à

chaque format. Il existe trois principaux types de codage :

- Le type I qui correspond aux opérations arithmétiques, aux transferts, aux branchements conditionnels et sauts avec registres.
- Le type II qui correspond aux branchements conditionnels et sauts avec immédiat, aux chargements et aux rangements.
- Le type III qui correspond aux chargements des immédiats sur 32 bits.

1. **Le type I**

   Le type I peut se décomposer en 5 sous-types qui ne présentent entre eux que peu de différences. Celles-ci sont dues principalement aux 3 possibilités d'adressage à l'intérieur d'un champ de 8 bits.

Type I.a

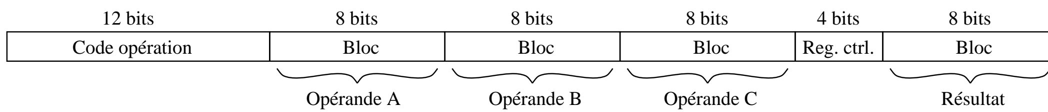


Fig. 4.3: Codage de type I.a

Le type I.a (Figure 4.3) est utilisé pour toutes les opérations arithmétiques sur les blocs (addition, addition avec retenue entrante, soustraction, soustraction avec retenue entrante, multiplication, multiplication-accumulation, division, division avec reste ajusté, décalages et normalisation) ainsi que pour les transferts, les branchements conditionnels avec registre et les sauts avec registre.

Type I.b

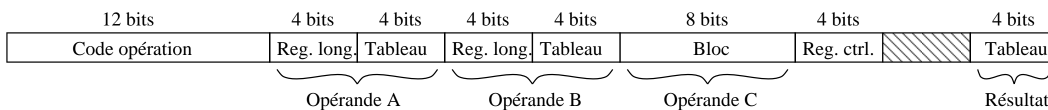


Fig. 4.4: Codage de type I.b

Le type I.b (Figure 4.4) est utilisé pour les opérations d'addition, d'addition avec retenue, de soustraction et de soustraction avec retenue pour lesquelles les deux opérandes sont des tableaux avec adressage de la longueur par registre.

Type I.bi

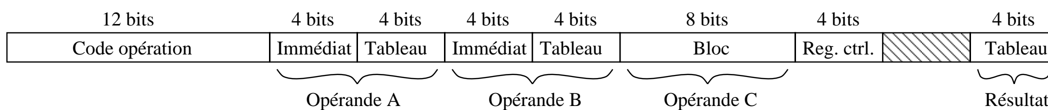


Fig. 4.5: Codage de type I.bi

Le type I.bi (Figure 4.5) correspond aux opérations d'addition, d'addition avec retenue, de soustraction et de soustraction avec retenue. Les deux opérandes de ces opérations sont des tableaux avec un adressage de leurs longueurs par immédiats.

Type I.c

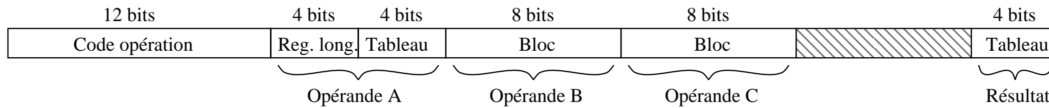


FIG. 4.6: Codage de type I.c

Le type I.c (Figure 4.6) est utilisé pour les opérations de multiplication, multiplication-accumulation, décalages et normalisation pour lesquelles une opérande est un tableau dont l'adressage de la longueur se fait par registre.

Type I.ci

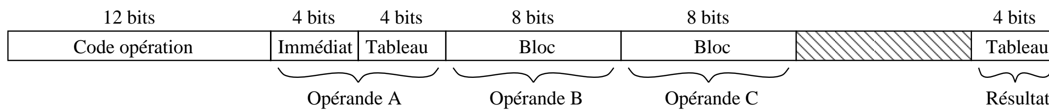


FIG. 4.7: Codage de type I.ci

Le type I.ci (Figure 4.7) est utilisé pour les opérations de multiplication, multiplication-accumulation, décalages et normalisation pour lesquelles une opérande est un tableau avec adressage de la longueur par immédiat.

2. **Le type II**

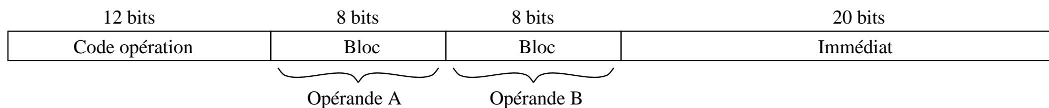


FIG. 4.8: Codage de type II

Le type II (Figure 4.8) contient un champ de 20 bits pour écrire un immédiat. Il est utilisé pour les branchements conditionnels et les sauts avec immédiat, les chargements, les chargements étendus et les rangements.

3. **Le type III**

Le type III (Figure 4.9) contient un champ de 32 bits pour écrire un immédiat. Il ne reste dans ce cas que 4 bits pour coder l'adresse du registre de destination. Les 4 bits du champ restant contiennent les poids faibles de l'adresse du registre destination, les 4 bits de poids forts étant fixés à 0000.

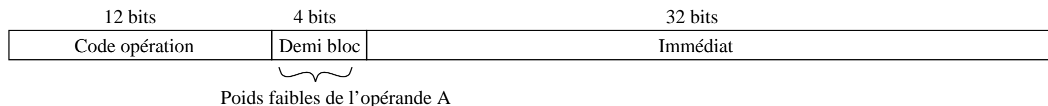


FIG. 4.9: Codage de type III

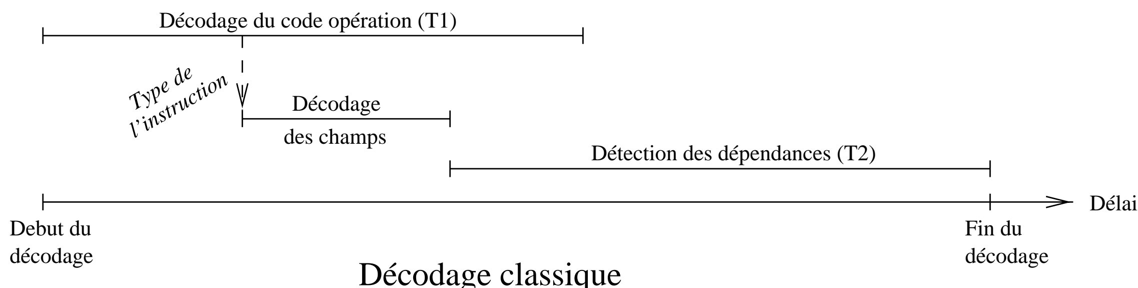
Ces différents codages peuvent cependant se ramener, dans un premier temps (c'est à dire au début du décodage), à seulement trois types distincts. En effet, on peut considérer que les types II et III sont en fait des types I dans lesquels on a moins d'adressages de registres puisqu'une partie de ces champs est occupée par un immédiat. De même à l'intérieur du type I, les types I.bi et I.ci peuvent être considérés respectivement comme des types I.b et I.c dans lesquels on a aussi moins d'adressages car la longueur des tableaux est un immédiat. Il suffit donc de détecter si le codage de l'opération est de type I.a, I.b ou I.c, c'est à dire si les opérandes sont des blocs ou des tableaux, et de commencer la détection des dépendances sur ce type. Dans un deuxième temps, lorsque le décodage du type exact de l'instruction est fini, il suffit alors d'annuler, si le type n'est pas I.a, I.b ou I.c, les dépendances supplémentaires que l'on a détectées et qui n'existent pas, puisque dans ces champs il y a des immédiats. Ce principe anticipe le type exact du codage ce qui permet de réduire le délai total constitué par le décodage et la détection des dépendances (Figure 4.10). Dans ce cas, le décodage doit retourner, pour chaque champ d'adresse de l'instruction, deux bits qui seront ensuite utilisés par le module de détection des dépendances. Le décodage génère, dans un premier temps, les bits *bloc/tab* qui indiquent le type de l'opérande (bloc ou tableau) contenu dans chaque champ selon que l'opération s'apparente à un type I.a, I.b ou I.c, puis dans un second temps, lorsque le type exact est connu, les bits I/U qui signalent si les champs sont utilisés ou non.

4.2.2 Détection des dépendances

Opération à lancer		Opération en cours		Adresses à comparer
Donnée	Adresse	Donnée	Adresse	
Bloc	Ah[3..0] , Al[3..0]	Bloc	Bh[3..0] , Bl[3..0]	Ah et Bh , Al et Bl
Bloc	Ah[3..0] , Al[3..0]	Tableau	B[3..0]	Ah et B
Tableau	A[3..0]	Bloc	Bh[3..0] , Bl[3..0]	A et Bh
Tableau	A[3..0]	Tableau	B[3..0]	A et B

TAB. 4.1: Comparaisons des adresses selon le type des données.

Les dépendances de données sont détectées entre les données qui doivent être utilisées par l'opération à lancer et celles qui sont utilisées par les opérations en cours. Pour cela on compare les adresses des blocs qui contiennent les données. Cependant, pour les opérations sur les tableaux, il ne faut pas seulement comparer l'adresse du bloc utilisé au moment de la comparaison, mais l'ensemble des blocs du tableau. Les comparaisons à effectuer ne sont donc pas les mêmes selon que les données soient des blocs ou des tableaux, comme le montre le tableau 4.1. La figure 4.11 présente



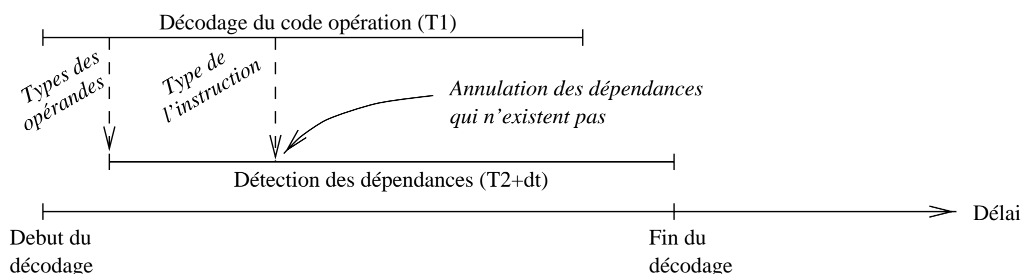


Fig. 4.10: Réduction des délais de l'étage de décodage.

le module qui permet d'effectuer la détection des dépendances entre deux adresses en fonctions des deux signaux ($bloc/tab$ et I/U) qui proviennent du décodage et qui sont associés à chaque champ d'adresse.

Il y a trois types de dépendances qui doivent être détectées :

1. **Ecriture après lecture**

   Cette dépendance n'existe que si les données lues par les opérations en cours sont des tableaux. On a au maximum 5 adresses correspondant à des tableaux en lecture dans le processeur : deux pour l'unité d'addition, une pour les unités de multiplication, décalage et chargement/rangement. Une opération peut écrire au maximum 2 données (un tableau et un bloc ou deux blocs). On a donc au total pour cette dépendance 10 comparaisons d'adresses à effectuer.

2. **Lecture après écriture**

   On compare les adresses qui seront écrites par les opérations en cours dans les unités avec les adresses des données qui doivent être lues par l'opération à lancer. Les unités étant pipelinées, plusieurs instructions peuvent être en cours d'exécution et ne pas avoir encore écrit leurs résultats. Il faut donc rechercher les adresses des données devant être écrites dans tous les étages des pipelines. Pour le processeur, on peut ainsi avoir au maximum 13 adresses en écriture au sein des différents pipelines. Le nombre maximum de lectures dans une instruction est 3 ce qui nécessite donc de faire 39 comparaisons d'adresses pour cette dépendance.

3. **Ecriture après écriture**

   On compare les adresses qui seront écrites par les opérations en cours dans les unités avec les adresses des données qui doivent être écrites par l'opération à

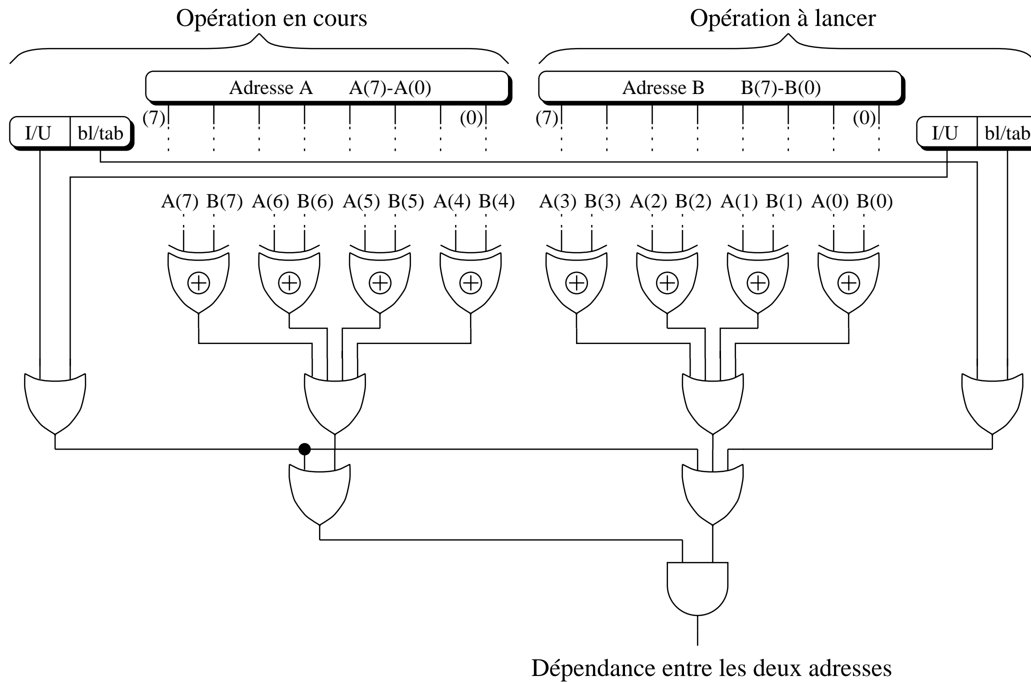


FIG. 4.11: Détection d'une dépendance entre deux adresses.

lancer. Le nombre d'écritures dans les unités est de 13, le nombre maximum d'écritures effectuées dans une instruction est 2, ce qui implique 26 comparaisons d'adresses pour cette dépendance.

Des dépendances de données existent aussi avec les registres de contrôle. Lorsque l'on fait un accès (en lecture ou en écriture) à un registre de donnée, on fait aussi un accès au registre de contrôle associé à cette adresse ce qui ne rajoute pas de nouvelles dépendances par rapport à celles sur les données. Par contre, les opérations d'addition et de soustraction permettent d'adresser directement des registres de contrôle pour sélectionner les retenues entrantes ou sortantes. Il faut donc aussi rechercher les dépendances avec ces registres adressés directement. On obtient au total pour les trois types de dépendances sur les registres de contrôle 24 comparaisons d'adresses supplémentaires.

Les registres de longueur quant à eux ne génèrent pas de dépendances puisqu'ils sont toujours associés avec les adresses des tableaux.

On a donc, au total, 99 dépendances qui doivent être testées avec le module de la figure 4.11. Si une seule de ces comparaisons signale une dépendance, alors le lancement de l'instruction est suspendu et la valeur du compteur de programme n'est pas modifiée.

4.2.3 Allocation des ressources

En parallèle avec la détection des dépendances de données, on vérifie si l'allocation des ressources matérielles nécessaires à la nouvelle opération est possible. Si cette allocation n'est pas possible, alors le lancement de la nouvelle instruction sera suspendu. Il faut donc vérifier :

1. **L'occupation des unités fonctionnelles**

   Chaque unité fonctionnelle possède un bit d'occupation qu'elle tient à jour et qui est consulté par le décodage pour le lancement d'une nouvelle instruction. Le décalage d'un cycle qui existe entre les étages successifs dans un pipeline fait que l'étage de décodage teste ce bit un cycle avant que l'on ait réellement besoin que l'unité soit libre. Il faut donc que le bit d'occupation corresponde au signal qui indique que l'unité a atteint son dernier cycle. Ce signal, généré par le module "dernier coup", dépend de l'opération et de la longueur des opérandes.

2. **L'allocation des bus libres**

   Dans le paragraphe 3.3.2, on a vu que le banc de registres autorisait 4 lectures et 2 écritures simultanément. Ce nombre d'accès étant limité, on doit donc s'assurer avant de lancer une nouvelle opération qu'elle pourra s'exécuter jusqu'à son terme sans qu'il y ait de conflits pour les accès au banc. Pour cela on alloue les bus aux unités, selon les besoins de l'opération, parmis les 4 bus de lecture et les 2 bus d'écriture qui sont disponibles. Les bus sont alloués à une unité pour toute la durée de l'opération. Chaque bus possède un bit d'état qui indique si il est occupé ou non. C'est l'unité qui, lorsqu'elle a terminé son opération, modifie le bit d'état du bus pour indiquer qu'il est de nouveau libre.

   Pour les bus de lecture, il faut faire une ou deux allocations de bus selon l'opération qui va être lancée. Le module qui fait cette allocation (Figure 4.12) est donc basé sur le principe suivant :

   - Recherche du premier bus libre parmis les 4 bus : résultat intermédiaire utilisé lorsque l'on a besoin d'un seul bus.
   - Recherche à nouveau du premier bus libre à partir d'un masque qui élimine le bus alloué précédemment : utilisé lorsque l'on a besoin d'un deuxième bus.

Pour le bus d'écriture, on ne fait qu'une seule allocation par opération. Il s'agit dans ce cas de rechercher un bus libre parmis 2 bus et de signaler le cas échéant, l'impossibilité de faire l'allocation (Figure 4.13).

Le schéma général pour les allocations des bus de lecture et d'écriture est donné figure 4.14. Les modules "fin de lecture" et "fin d'écriture" génèrent des signaux qui permettent de libérer les bus respectivement lors de la dernière lecture et de la dernière écriture. Cela signifie aussi que l'affectation d'un bus pour un seul accès ne modifie pas son bit d'état puisque le début de l'affectation coïncide avec la libération. C'est pourquoi, hormis l'unité d'addition, les autres unités n'utilisent pas de matériel pour modifier le bit d'état du bus de lecture associé à leur deuxième opérande car celle-ci n'effectue jamais plus d'une lecture. Cela permet de réduire de 40% le matériel nécessaire à l'allocation des bus de lecture.

4.2.4 Table des longueurs

La table des longueurs contient la taille réelle des tableaux qui sont stockés dans les registres de donnée. Cette table est lue à chaque cycle par les unités fonctionnelles. Ces dernières utilisent l'information de longueur dans le premier étage de leur pipeline pour générer des adresses ou des signaux de contrôle.

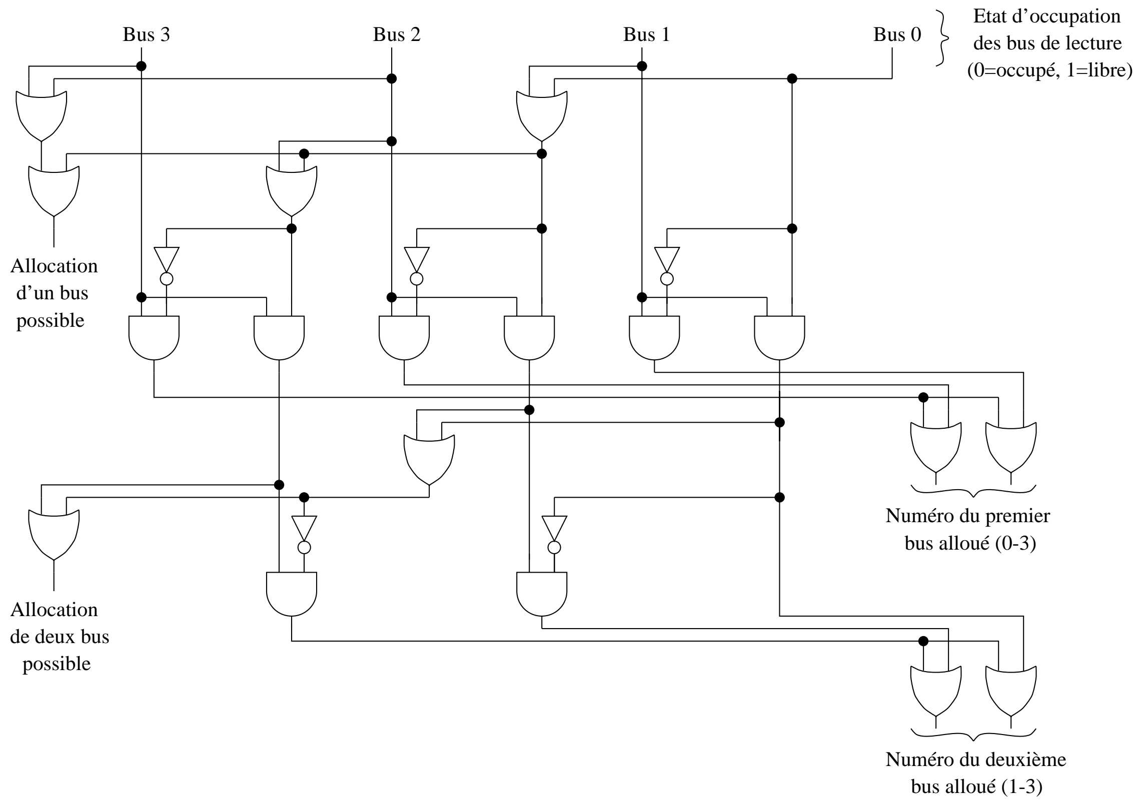


FIG. 4.12: Détection des bus de lecture libres.

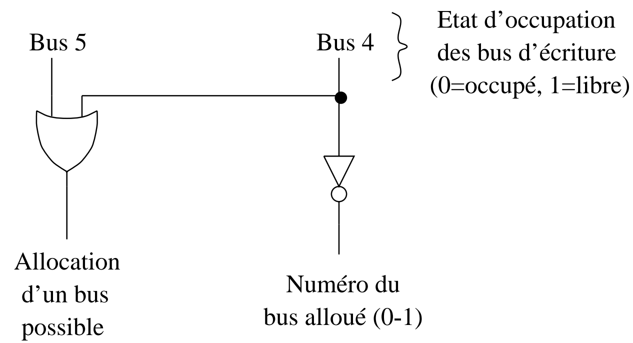


FIG. 4.13: Détection des bus d'écriture libres.

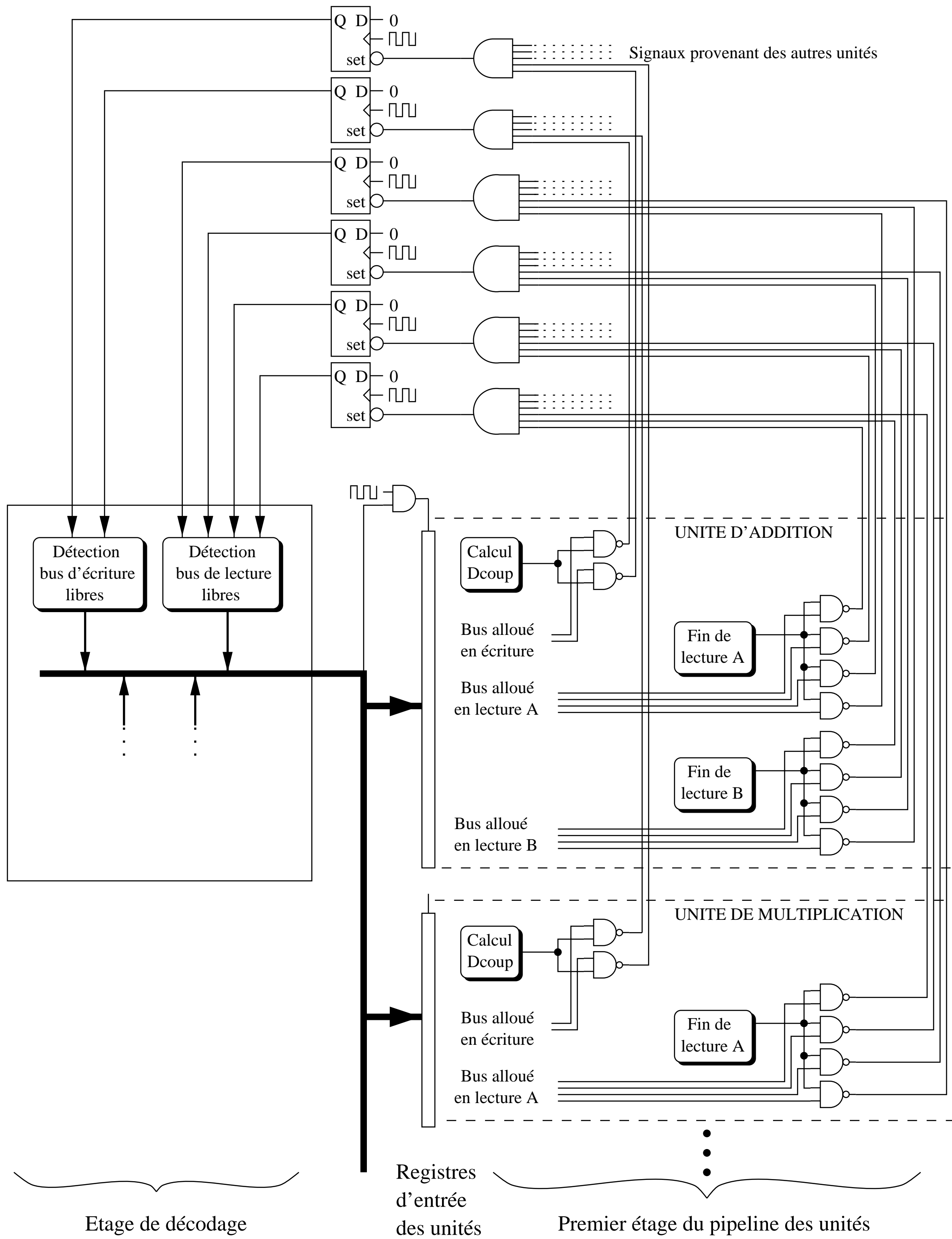


FIG. 4.14: Allocation des bus de lecture et d'écriture.

Or la lecture de la table des longueurs suivi de la génération des adresses puis de la lecture des registres de donnée entraînent un chemin critique important pour le premier étage des pipelines. Afin de diminuer le temps de cycle, la lecture de la table des longueurs est anticipée et s'effectue à la fin du cycle précédent. La première lecture de la table des longueurs doit donc être effectuée par l'étage de décodage. Les autres lectures sont ensuite réalisées par le premier étage des pipelines pour le cycle suivant. Cela permet de transmettre la longueur au début de chaque cycle et donc de commencer à générer les adresses et les signaux de contrôle très tôt dans le cycle. L'écriture de la table des longueurs ne s'effectue qu'une seule fois par opération, lors de l'écriture du dernier bloc du résultat.

Pour éviter des dépendances de donnée liées à l'information de longueur, il faut s'assurer que lorsque l'on anticipe la lecture de la table, l'information de longueur soit déjà disponible. Une telle dépendance survient lorsqu'on réalise une écriture suivie d'une lecture durant le même cycle d'horloge. Pour supprimer cette dépendance, on effectue chaque accès sur un demi-cycle ce qui permet de faire la mise à jour (écriture) de la longueur dans la première moitié du cycle puis de lire cette même information de longueur durant la seconde partie du cycle. Deux accès consécutifs sont en effet possibles pendant un cycle car la table des longueurs ne possède que 16 éléments ce qui permet des lectures et des écritures rapides.

On peut ainsi enchaîner une opération dès que la précédente est finie, sans cycles de suspension supplémentaires liés à la longueur, comme le montre la figure 4.15. La dépendance de donnée sur l'opérande C impose une suspension de la seconde opération jusqu'au cycle 7. On voit alors que la longueur écrite par $Op1$ dans la première partie du cycle 8 peut être lue par le décodage de $Op2$ dans la seconde partie de ce même cycle sans provoquer de suspensions supplémentaires.

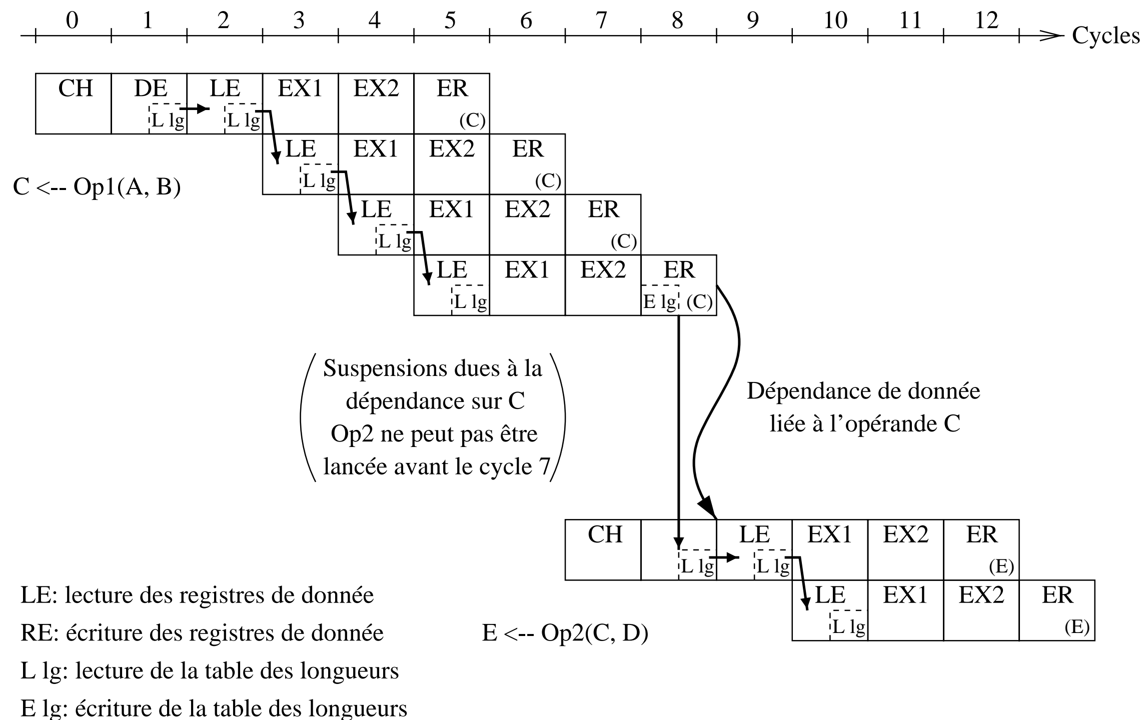


FIG. 4.15: Lecture de la table des longueurs.

4.2.5 Schéma général du décodage

Une instruction peut être lancée si aucune des conditions bloquantes n'est valide, c'est à dire si il n'y a pas de dépendance de données, si l'unité destination est libre et si on peut allouer les bus de lecture et d'écriture nécessaires à l'exécution de cette instruction. Si toutes ces conditions sont vérifiées, alors au prochain cycle, le décodage transmettra à l'unité sélectionnée tous les paramètres qui lui serviront à exécuter l'instruction de façon autonome. Ces paramètres sont principalement le type des opérandes (bloc ou tableau), la longueur dans le cas d'un tableau, les adresses des différents registres à lire ou écrire et les bits de contrôle qui permettent entre autre de sélectionner une opération précise parmis celles réalisables par l'unité. Le schéma complet du décodage est présenté figure 4.16.

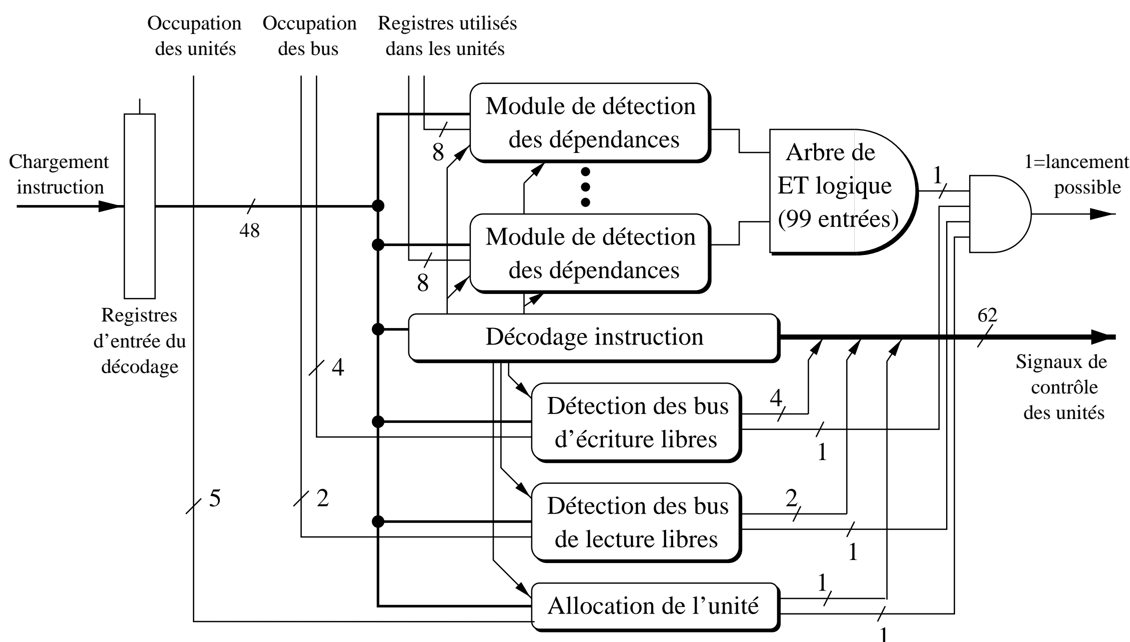


FIG. 4.16: Schéma général du décodage.

4.3 Conclusions

On a présenté dans ce chapitre des solutions pour minimiser les temps d'exécution des étapes de chargement et de décodage. Dans le premier étage du pipeline, on utilise un module de "contrôle du CP" qui permet de gagner un cycle lors des branchements conditionnels en anticipant le calcul des deux adresses de destination possibles (correspondantes aux branchements pris et non pris). Ce module est aussi utilisé pour maintenir la valeur du compteur de programme à l'adresse de l'instruction *NOP* qui suit toutes les instructions de saut ou de branchement tant qu'il y a une suspension. Cette solution réduit la taille du code en supprimant les instructions *NOP* supplémentaires lorsqu'il y a plusieurs cycles de suspension.

Le principal effort d'optimisation a cependant été réalisé dans l'étage de décodage car celui-ci était critique au niveau du temps. La définition d'un nombre restreint

de formats d'instruction, avec une recherche de similitudes entre eux, simplifie le décodage. Une technique d'anticipation du format exact d'une instruction permet de détecter plus tôt les champs de l'instruction et ainsi de commencer plus rapidement la détection des dépendances de données. Lorsque le format exact, et donc les champs réellement utilisés, sont décodés, on peut alors ajuster la détection des dépendances en fonction du type (bloc, tableau ou inutilisé) de chaque champ. Cette solution réduit le chemin critique de l'étage car elle autorise un recouvrement du décodage de l'instruction et de la détection des dépendances de données.

L'allocation dynamique des ressources matérielles comme les bus d'accès aux bancs de registres évite les conflits entre des instructions en parallèle et permet d'optimiser leur occupation.

Enfin l'utilisation d'une table des longueurs est essentielle pour ramener uniquement aux blocs valides l'exécution des instructions sur les tableaux. La solution d'une lecture en fin de cycle précédent, avec des accès à cette table sur des demi-cycles, élimine toutes les dépendances de données liées à l'information de longueur.

Chapitre 5

Unités fonctionnelles

L'architecture générale du cœur du processeur est de type RISC et possède cinq unités fonctionnelles pipelinées qui peuvent travailler en parallèle. Ces unités réalisent les opérations suivantes :

- Addition, soustraction.
- Multiplication, multiplication-accumulation.
- Division.
- Décalage, normalisation.
- Chargement, rangement.

Les pipelines de chacune des unités sont présentés figure 3.12, page 52.

Ces unités réalisent directement les opérations de base de l'arithmétique exacte. Elles possèdent pour cela une architecture optimisée pour ce type d'opération et tout particulièrement des opérateurs arithmétiques qui prennent en compte toutes les spécificités liées au calcul en arithmétique exacte.

Dans les paragraphes suivants, on présente en détail la structure générale utilisée par l'ensemble des unités, puis on décrit ensuite chaque unité séparément.

5.1 Architecture générale d'une unité

Les unités fonctionnelles possèdent une même architecture qui est présentée figure 5.1. Elle se compose principalement d'un contrôleur d'unité, d'un opérateur arithmétique pipeliné en notation redondante et d'un module de reconversion en notation CC2.

Le contrôleur d'unité a plusieurs fonctions :

- Il synchronise les transferts d'informations entre l'unité, qui est autonome, et le reste du circuit.
- Il génère à chaque cycle les nouvelles adresses pour les lectures et les écritures et commande les accès aux différents bancs de registre.
- Il contrôle le fonctionnement du pipeline de l'unité.

Lorsqu'une nouvelle opération est lancée, l'étage de décodage d'une instruction transmet au contrôleur d'unité (de l'unité sélectionnée lors du décodage) tous les paramètres qui lui sont nécessaires pour exécuter l'opération de façon indépendante. Une

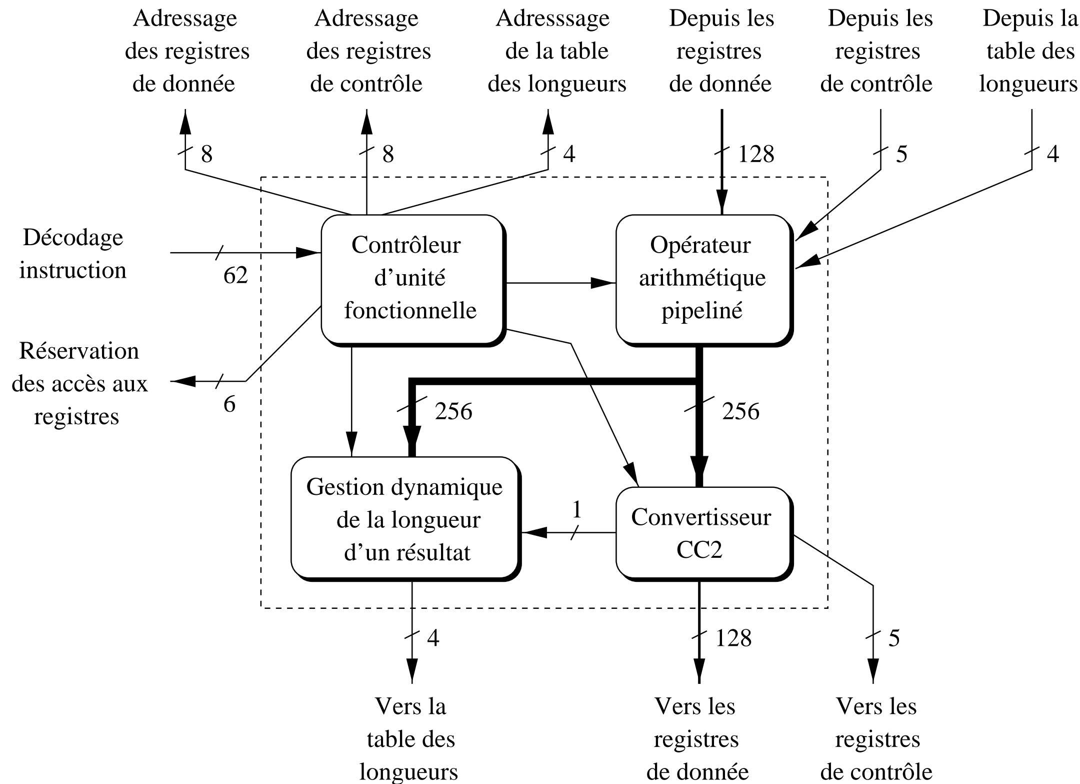


FIG. 5.1: Architecture générale d'une unité.

fois ces paramètres transmis, plus aucun échange d'information n'est réalisé entre le décodage et l'unité. L'opération s'exécute alors jusqu'à son terme sans interruption.

Les caractéristiques générales concernant le pipeline des opérateurs, la notation redondante et la reconversion sont présentées dans les paragraphes suivants. Les modules utilisés pour la gestion dynamique de la longueur des résultats sont décrit dans le paragraphe 6.1.

5.1.1 Opérateurs pipelinés

Les opérations sur les tableaux, présentées paragraphe 3.2.3, transmettent des paramètres directement d'une opération sur un bloc à l'opération sur le bloc suivant. L'utilisation d'opérateurs pipelinés peut générer des dépendances de données au niveau de ces paramètres. En effet, le pipeline permettant de commencer une nouvelle opération avant que la précédente ne soit terminée, une opération peut vouloir lire le paramètre qui doit lui être transmis directement avant que la précédente ne l'ait émis. Ces dépendances ne peuvent pas être détectées par l'étage de décodage car les paramètres ne passent pas par des registres. Pour que ces dépendances ne soient pas bloquantes, ce qui annulerait l'accélération apportée par les opérations sur les tableaux, les pipelines des opérateurs doivent respecter la condition suivante :

> Le temps maximum entre la réception d'un paramètre provenant de l'opération précédente et le transfert du nouveau paramètre à l'opération suivante doit être de un cycle d'horloge.

Cette condition est illustrée figure 5.2 à partir de l'exemple de l'addition. Au cycle t1, l'étage E1 de la première opération lit la retenue entrante Cin1. Au cycle suivant, le même étage de la deuxième opération lit la retenue entrante Cin2. Or cette dernière dépend du calcul précédent puisqu'elle correspond à la retenue sortante Cout générée par la première opération. Il faut donc que la première instruction ait renvoyé l'information Cout dans un intervalle de temps de un cycle après la lecture de la retenue entrante Cin.

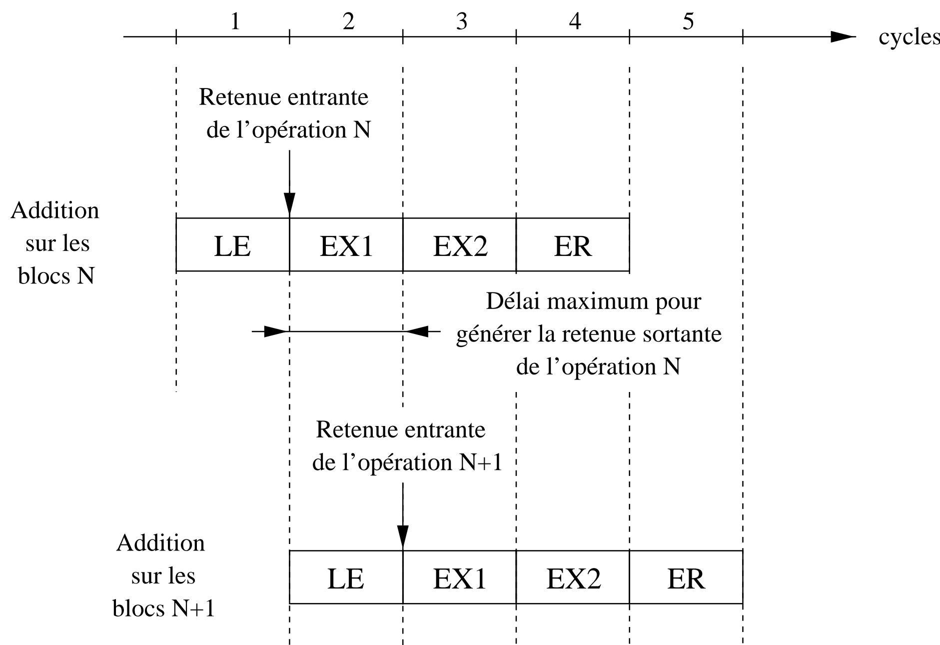


FIG. 5.2: Propagation de la retenue pour les opérations d'addition.

Cette condition, qui est valable pour toutes les opérations sur les tableaux, implique donc un "découpage" particulier des pipelines puisqu'il faut que la lecture et la transmition des paramètres par le matériel s'effectue dans le même cycle. Afin que cette particularité n'impose pas un temps de cycle trop grand, on utilisera dans les unités des opérateurs qui permettent d'obtenir un délai faible entre la lecture et la transmition de ces paramètres.

5.1.2 Notation redondante

Les opérateurs arithmétiques utilisent une notation redondante qui supprime les propagations de retenue et permet de réaliser les calculs en un temps constant [1]. Cette notation apporte un gain en temps important lorsque l'on travaille sur des données de 128 bits. Seuls les résultats définitifs, lorsqu'ils sont reconvertis en CC2, entraînent une propagation de retenue. L'exécution en temps constant permet aussi de transférer plus rapidement les paramètres d'une opération à la suivante et donc de réduire la durée du cycle.

Dans cette notation, chaque nombre est représenté comme étant la différence de deux nombres positifs en notation binaire classique :

$S = \sum s_i 2^i$ s'écrit $S = S^+ - S^- = \sum (s_i^+ - s_i^-) 2^i$

avec $S^+ = \sum s_i^+ 2^i$ et $S^- = \sum s_i^- 2^i$

Cette notation, appelée *Borrow Save*, permet de coder des chiffres appartenant à l'ensemble {-1, 0, +1} comme le montre le tableau 5.1.

s_i^+	s_i^-	s_i
0	0	0
0	1	-1
1	0	+1
1	1	0

TAB. 5.1: Codage des chiffres en notation *Borrow Save*.

Inverser le signe d'un nombre en notation *Borrow Save* s'effectue aisément par simple permutation des bits s_i^+ et s_i^-. En revanche, la détection du signe d'un nombre est complexe puisqu'il faut rechercher, à partir des poids forts, le premier indice pour lequel $s_i^+ \neq s_i^-$.

Il existe aussi la notation *Carry Save* dans laquelle chaque nombre est représenté comme étant la somme de deux nombres positifs en notation binaire classique :

$S = \sum s_i 2^i$ s'écrit $S = S_1 + S_2 = \sum (s_{1i} + s_{2i}) 2^i$

avec $S_1 = \sum s_{1i} 2^i$ et $S_2 = \sum s_{2i} 2^i$

Cette notation permet de coder des chiffres appartenant à l'ensemble {0, 1, 2}.

5.1.3 Reconversion en CC2

La reconversion en CC2 d'un nombre S codé en *Borrow Save* s'obtient en faisant la soustraction de S^+ et S^-. On utilise une architecture en temps $O(\log_2 N)$ afin de diminuer le temps de calcul de cette reconversion car elle se situe sur le chemin critique des opérateurs. La soustraction est obtenue à partir de l'addition en inversant les bits de l'opérande à soustraire. Pour simplifier l'écriture des équations, et améliorer la compréhension, on ne présentera dans ce paragraphe que des architectures d'additionneurs.

Il existe plusieurs architectures qui ont des temps de traversée en $O(\log_2 N)$ [72][43][7][30]. Elles peuvent toutes être décrites à partir des équations suivantes qui ont été proposées par Brent et Kung [7].

- Génération d'une retenue à la position i
$$G_{i,i} = s_i^+ \wedge \overline{s_i^-}$$
- Propagation d'une retenue à la position i
$$P_{i,i} = s_i^+ \oplus \overline{s_i^-}$$
- Génération d'une retenue entre les positions i et k
$$G_{k,i} = G_{k,j} \vee P_{k,j} \wedge G_{j-1,i} \qquad (k \geq j > i)$$
- Propagation d'une retenue entre les positions i et k
$$P_{k,i} = P_{k,j} \wedge P_{j-1,i} \qquad (k \geq j > i)$$
- Retenue à la position $i + 1$
$$c_{i+1} = G_{i,0} \vee P_{i,0} \wedge \overline{c_0}$$

– Résultat à la position i

$$S_i = P_{i,i} \oplus c_i$$

Une synthèse de ces différentes architectures d'additionneurs est présentée dans [29]. Elle permet en particulier de comparer le nombres de cellules élémentaires sur le chemin critique, le nombres total de cellules élémentaires utilisées ainsi que la charge maximale en sortie des cellules élémentaires. Le tableau 5.2 permet de comparer ces paramètres pour quatre architectures lorsque la taille des opérandes est de 128 bits.

type d'architecture	Nombre total de cellules	Cellules sur le chemin critique	Charge maximale (en nombre de cellules)
Brent et Kung	249	12	7
Sklansky	448	7	64
Kogge et Stone	769	7	2
Han et Carlson	448	8	2

TAB. 5.2: Comparaisons de quatre architectures d'additionneurs sur 128 bits.

On utilisera une architecture de type Han et Carlson car elle offre le meilleur compromis délai$\times$surface. Le délai est une fonction qui dépend du nombre de cellules sur le chemin critique et de la charge en sortie des cellules. Dans l'architecture de Sklansky, cette dernière augmente considérablement le délai c'est pourquoi l'architecture n'a pas été retenue.

5.2 Unité d'addition

5.2.1 Caractéristiques générales

L'unité d'addition effectue les opérations d'addition, de soustraction, avec ou sans retenues, sur des blocs ou des tableaux, ainsi que les comparaisons pour les branchements conditionnels. Cette unité est pipelinée sur 4 étages. Le premier correspond principalement au contrôleur d'unité. Les deuxièmes et troisièmes étages contiennent l'opérateur d'addition/soustraction et le quatrième correspond à la phase d'écriture des résultats dans les bancs de registres. Cette unité a un débit de 1 bloc par cycle, avec une latence de 4 cycles.

Le contrôleur d'unité effectue les accès aux registres comme indiqué figure 5.3. Les retenues en entrée et en sortie de l'opérateur sont stockées dans les registres de contrôle. La retenue entrante est lue dans le registre de contrôle de même adresse que le bloc résultat, la retenue sortante est écrite dans le registre de contrôle d'adresse suivante.

5.2.2 Opérateur d'addition/soustraction

Un schéma général de l'opérateur d'addition/soustraction est donné figure 5.4. Les différents modules qui le composent sont développés dans les paragraphes suivants. Les signaux $Add/Sous$ et $Cinb/Cin$ proviennent du décodage et indiquent respectivement une opération d'addition ou de soustraction et une opération avec ou sans retenue.

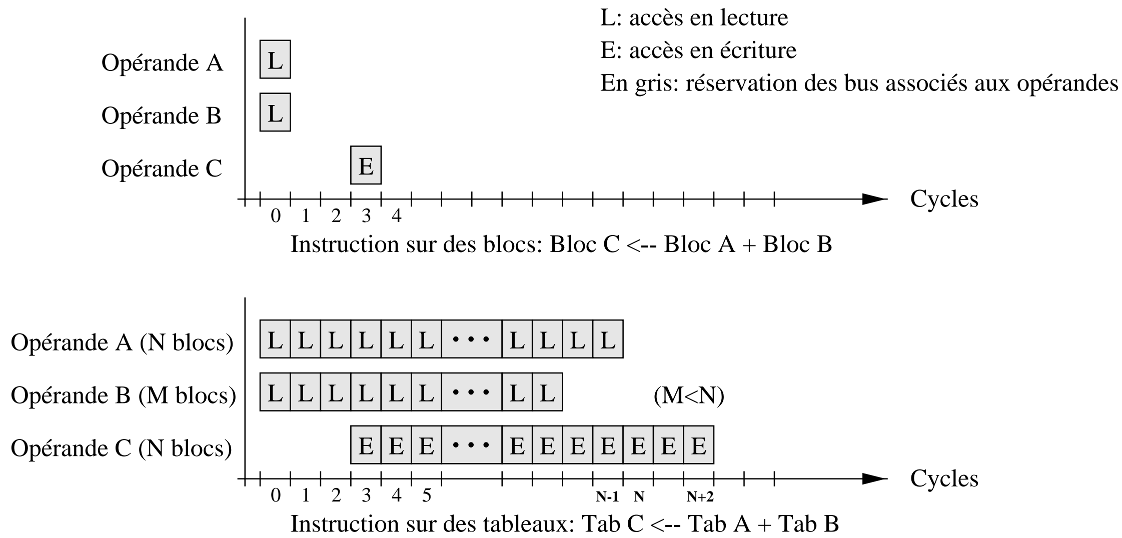


FIG. 5.3: Accès aux registres effectués par l'unité d'addition.

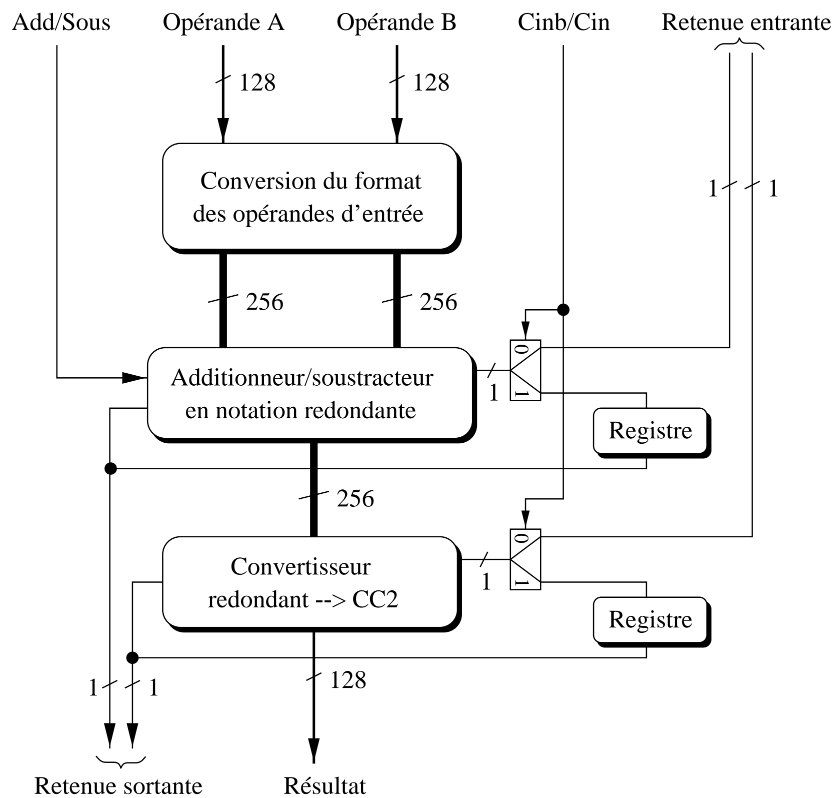


FIG. 5.4: Schéma général de l'opérateur d'addition/soustraction.

5.2.2.1 Taille d'un résultat

Les opérandes d'entrées, qui ont un format de 128 bits, peuvent être de différents types :

- *Non signé*, c'est à dire que tous les bits de poids 2^0 à 2^{127} sont positifs. Cette représentation est équivalente à celle d'un nombre codé en CC2 sur 129 bits avec le bit de poids fort toujours à 0 pour indiquer un nombre positif. La valeur contenue dans ces blocs appartient à l'ensemble $\{0, 2^{128}\text{-}1\}$.
- *Signé*, c'est à dire codés en CC2 sur 128 bits, les bits de poids 2^0 à 2^{126} étant positifs, le bit de poids 2^{127} négatif. La valeur contenue dans ces blocs appartient à l'ensemble $\{\text{-}2^{127}, 2^{127}\text{-}1\}$.

Lorsque l'on fait l'addition ou la soustraction de deux blocs de 128 bits, on peut avoir des résultats qui nécessitent 130 bits pour être codés. Cela provient du type *Non signé* qui en réalité se comporte vis à vis de la taille du résultat comme un nombre signé de 129 bits. Il y a trois catégories de résultats qui dépendent de la taille et du signe :

- Les résultats signés qui appartiennent à l'ensemble $\{\text{-}2^{128}, 2^{128}\text{-}1\}$. Le bit 2^{128} correspond à une retenue sortante classique et sera injecté en tant que retenue entrante lors du calcul sur les blocs suivants.
- Les résultats signés qui appartiennent à l'ensemble $\{2^{128}, 2^{129}\text{-}1\}$. Cette catégorie de résultats correspond à des nombres positifs qui nécessitent 130 bits pour être codés mais dont le bit 2^{129} (bit de signe) vaut toujours zéro. Dans ce cas, le bit 2^{128} est positif et pourra donc être additionné en tant que retenue entrante positive lors du calcul sur les blocs suivants. Le bit 2^{129} étant nul, il n'intervient pas dans le calcul sur les blocs suivants.
- Les résultats signés qui appartiennent à l'ensemble $\{\text{-}2^{129}, \text{-}2^{128}\text{-}1\}$. Cette catégorie de résultats correspond à des nombres négatifs qui nécessitent 130 bits pour être codés et dont le bit 2^{129} (bit de signe) vaut 1. Dans ce cas, la retenue sortante tient sur 2 bits (les bits 2^{128} et 2^{129}) et on ne peut pas l'injecter dans le calcul suivant car le bit 2^{129} devrait être soustrait des poids 2^1 ce qui n'est pas possible dans une architecture d'additionneur classique puisque la retenue entrante ne peut être que de poids 2^0.

Pour résoudre ce troisième cas, on décompose le bit de poids 2^1 en deux bits de poids 2^0 qui correspondront aux retenues entrantes de deux additionneurs en série. Pour ne pas trop augmenter la durée de l'opération, nous utilisons un premier additionneur en notation redondante qui effectue les opérations en un temps constant sans propagation de retenue interne. Cette solution ne fait en réalité que réutiliser le matériel nécessaire pour la gestion dynamique des longueurs qui, elle, impose d'avoir un additionneur/soustracteur redondant suivi d'un soustracteur classique pour la reconversion en CC2 (voir paragraphe 6.1). Le tableau 5.3 montre les retenues entrantes qui doivent leur être appliquées en fonction des retenues obtenues lors du calcul précédent, en remarquant que dans le cas d'une addition, la retenue ne peut se propager que sur le bit de poids supérieur alors que dans le cas de la soustraction, cette propagation peut se faire sur les deux bits de poids supérieurs. .

Opération	Retenue sortante de l'étage précédent	Retenues entrantes	
		Additionneur/soustracteur redondant	Soustracteur de Han et Carlson
Addition	0	0	0
Addition	+1	1	0
Soustraction	0	0	0
Soustraction	-1	1	0
Soustraction	-2	1	1

TAB. 5.3: Retenues entrantes des opérateurs de l'unité d'addition.

5.2.2.2 Additionneur/soustracteur

L'opérateur d'addition/soustraction est composé de 3 parties : la première correspond au codage des entrées, la seconde à l'additionneur/soustracteur en notation redondante et la troisième partie à la reconversion du résultat redondant en notation CC2 avec un soustracteur de type Han et Carlson.

La première partie réalise le codage des opérandes A et B en fonction du signal *add/sous*, provenant du décodage, et des signaux NS/S, D, Vd et NF/F associés à chaque opérande . Ce dernier signal, généré par le contrôleur d'unité, indique que l'opérande a atteint sa longueur maximale et qu'il faut forcer ses entrées à zéro lors des prochains cycles. Il est utilisé dans le cas d'opérations sur les tableaux lorsque les deux opérandes n'ont pas la même longueur. On rappelle que le signal NS/S est associé à chaque bloc et indique si celui-ci est signé et que les signaux D et Vd proviennent des registres de contrôle et indiquent si le bloc associé contient un débordement.

C'est ce module de codage des entrées qui permet d'effectuer des opérations sur deux formats différents (signé ou non signé) avec le même opérateur. Les valeurs que cette interface doit appliquer sur les entrées A^+, A^- et B^+, B^- de l'opérateur redondant sont données dans les tableaux 5.4 et 5.5.

Signaux de commande				Entrées de l'opérateur redondant					
add/sous	NF/F_A	D_A	NS/S_A	A^+_{127}	A^-_{127}	A^+_x	A^-_x	A^+_0	A^-_0
x	0	0	0	A_{127}	0	A_x	0	A_0	0
x	0	0	1	0	A_{127}	A_x	0	A_0	0
x	0	1	x	0	0	0	0	0	Vd_A
x	1	x	x	0	0	0	0	0	0

TAB. 5.4: Codage de l'opérande d'entrée A.

L'opérateur redondant est constitué de réducteurs 4/2. Ceux-ci sont présentés figure 5.5. Dans ce schéma, la fonction *Majorité* vérifie l'équation :

$$h = bc + cd + bd$$

Les réducteurs 4/2 vérifient l'équation :

$$2h + 2g + f = a + b + c + d + e$$

Ces cellules permettent d'effectuer l'addition sans propagation de retenue. Le schéma complet de l'additionneur redondant est donné figure 5.6.

Signaux de commande				Entrées de l'opérateur redondant					
$add/sous$	NF/F_B	D_B	NS/S_B	B_{127}^+	B_{127}^-	B_x^+	B_x^-	B_0^+	B_0^-
0	0	0	0	B_{127}	0	B_x	0	B_0	0
0	0	0	1	0	B_{127}	B_x	0	B_0	0
0	0	1	x	0	0	0	0	0	Vd_B
0	1	x	x	0	0	0	0	0	0
1	0	0	0	0	B_{127}	0	B_x	0	B_0
1	0	0	1	B_{127}	0	0	B_x	0	B_0
1	0	1	x	0	0	0	0	Vd_B	0
1	1	x	x	0	0	0	0	0	0

TAB. 5.5: Codage de l'opérande d'entrée B.

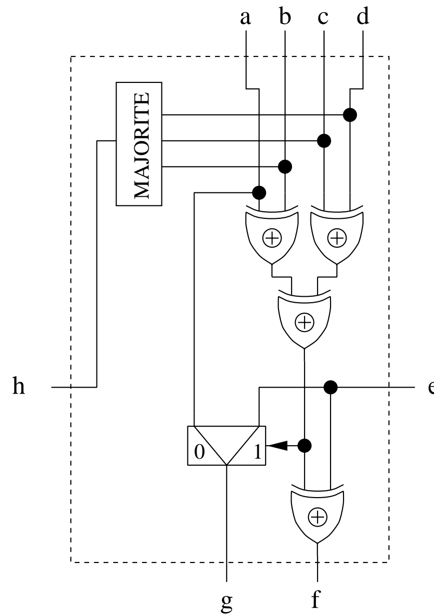


FIG. 5.5: Réducteur 4/2.

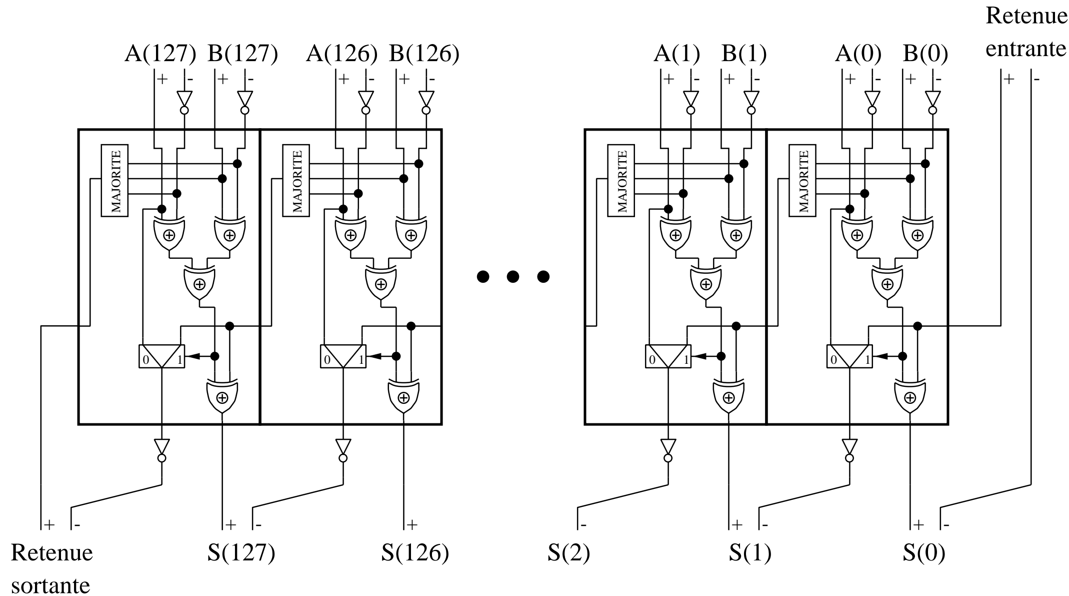


FIG. 5.6: Additionneur redondant.

La reconversion du résultat redondant en notation CC2 s'effectue avec un soustracteur de type Han et Carlson (Voir paragraphe 5.1.3). Cet opérateur retourne en même temps que le résultat deux signaux : $A = B$ et $A > B$. Ceux-ci seront utilisés lors des branchements conditionnels pour détecter si la condition est vraie ou fausse (Tableau 5.6).

Condition à tester	Réalisation en fonction des signaux $A = B$ et $A > B$
$A = B$	$A = B$
$A \neq B$	$\overline{A = B}$
$A > B$	$A > B$
$A \geq B$	$(A > B) + (A = B)$
$A < B$	$\overline{(A > B) + (A = B)}$
$A \leq B$	$\overline{A > B}$

TAB. 5.6: Détection des conditions en fonctions des signaux $A = B$ et $A > B$.

5.2.3 Synthèse et placement-routage

Le flot de conception qui a été utilisé pour la réalisation des blocs principaux du circuit est décrit dans ce paragraphe. La méthodologie qui a été respectée pour la conception de l'unité d'addition est la même que celle qui sera employé pour la réalisation des autres unités.

Le circuit est d'abord décrit en langage VHDL. La description d'un circuit en VHDL présente l'avantage d'être beaucoup plus rapide que la description de ce même

circuit à partir d'un éditeur graphique de schémas. De même, les corrections et les modifications sont plus aisées sur un programme en VHDL que sur un schéma. On utilise une description mixte qui fait appel à des descriptions structurelles et comportementales. Ces dernières étant utilisées lorsque la partie décrite n'influe pas sur le chemin critique ou la surface de l'unité. De plus, dans l'optique d'une synthèse de notre description, celle-ci doit se restreindre à un sous ensemble du langage VHDL afin de ne pas comporter d'éléments comme les délais ou les retards. Ceux-ci ne sont pas synthétisables car ils ne correspondent à aucune représentation matérielle.

La description VHDL fait appel à plusieurs fichiers. Il y a ceux qui sont utilisés par l'ensemble des blocs décrit :

- `elem_composants.vhd` qui contient des composants élémentaires.
- `elem_fonctions.vhd` qui contient des composants élémentaires représenté sous forme de fonction pour des raisons de lisibilité des programmes et de simplicité d'emploi.
- `elem_unité.vhd` qui contient des composants utilisés par la partie contrôle des unités.

Ces fichiers sont présentés en annexe B. Il y a ensuite les fichiers qui sont spécifiques à chaque bloc. Pour les unités, il y a trois fichiers qui sont :

- `nomunité_fct.vhd` qui contient des composants spécifiques à cette unité.
- `nomunité_op.vhd` qui est la description VHDL de l'opérateur principal constituant l'unité.
- `nomunité_ctrl.vhd` qui correspond à la description VHDL de l'unité incluant l'opérateur décrit dans le fichier précédent.

`Nomunité` correspond à la dénomination de l'unité soit `add`, `dec`, `mult`, `div` ou `ls` respectivement pour les unités d'addition, de décalage, de multiplication, de division ou de chargement/rangement. Les fichiers concernant l'unité d'addition sont donnés en annexe B.

On effectue ensuite des simulations logiques fonctionnelles à l'aide du simulateur VHDL de SYNOPSYS pour s'assurer que le fonctionnement du circuit décrit en VHDL est correct. Lorsque ce n'est pas le cas, on remonte corriger la description VHDL.

Une fois le fonctionnement validé, on fait une synthèse RTL de notre circuit à l'aide de l'outil de synthèse de SYNOPSYS. Cette opération va transformer notre description VHDL en une représentation constituée de portes interconnectées par des fils (*Netlist*). L'optimisation (avec le même outil) de cette *Netlist* est ensuite effectuée en tenant compte des caractéristiques des portes de la bibliothèque avec laquelle on va fabriquer le circuit. Dans notre cas, nous avons utilisé la bibliothèque en technologie CMOS 0,7μm de ATMEL-ES2. Chaque composant de cette bibliothèque est caractérisé par des informations sur son temps de traversée, ses capacités, sa sortance (*Fan-out*). A partir de ces paramètres, l'outil peut calculer l'ensemble des temps de traversée entre deux nœuds du circuit. En particulier, il lui est possible de déterminer le chemin critique d'un bloc et d'en préciser les bornes à l'utilisateur. On utilise cette information pour effectuer l'optimisation de la *Netlist* en spécifiant des contraintes de temps pour ce chemin critique. L'outil de SYNOPSYS réalisera alors une nouvelle optimisation de façon à satisfaire le critère de temps fixé. On recherche

ensuite le nouveau chemin critique et on recommence les opérations précédentes jusqu'à ce qu'il n'y ait plus aucun chemin dont le temps soit supérieur à la contrainte que l'on a fixé. On peut remarquer que si l'on connaît initialement les bornes des chemins susceptibles d'être au-delà de notre contrainte de temps, il est alors possible d'appliquer cette dernière dès la première optimisation à l'ensemble des chemins "longs". On diminue ainsi le nombre d'optimisations à effectuer et donc le temps de conception. Cette remarque impose de bien connaître l'architecture du bloc optimisé et donc d'avoir peu d'éléments utilisant une description comportementale. Dans le cas du processeur, l'utilisation d'une architecture pipelinée pour les unités fonctionnelles nécessite de rechercher le chemin critique de chaque étage. Ces derniers doivent être considérés comme des modules indépendants s'exécutant en parallèle. Il faut donc appliquer à chacun la contrainte de temps.

Lorsque toutes les optimisations ont été réalisées, on fait une première analyse du résultat retourné par l'outil. Il faut vérifier en particulier si la contrainte de temps fixée a pu être respectée pour tous les chemins. Lorsque ce n'est pas le cas, on a trois solutions possibles :

- On augmente la contrainte de temps. Dans le cas du processeur, cela correspond à une diminution de sa fréquence de fonctionnement puisque l'on augmente la durée du cycle de pipeline.
- On modifie l'architecture de façon à réduire le ou les chemins trop longs. Pour les architectures pipelinées, une meilleure répartition des éléments dans les différents étages peut parfois suffire pour réduire les chemins critiques. Les modifications sont effectuées au niveau de la description VHDL et l'ensemble des étapes précédentes (Simulations fonctionnelles, synthèse, optimisations) doivent être de nouveau effectuées.
- On remplace certains éléments du chemin critique par des modules équivalents réalisés en conception optimisée (*full-custom*). Les temps de ces derniers sont plus faibles que ceux des modules en cellules précaractérisées (cellules standards) ce qui permet de réduire le chemin critique. Si on réalise en *full-custom* un module qui présente les mêmes caractéristiques à ses bornes que celui qu'il remplace, il suffit alors de refaire une seule fois la synthèse et l'optimisation sur la description VHDL amputée de tous les éléments réalisés en *full-custom*.

L'autre point important que l'on peut vérifier à ce stade de la conception concerne la réalisation en cellules standards des parties provenant d'une description comportementale. Certains modules peuvent en effet présenter des régularités ou des propriétés qui sont mal exploitées ce qui entraîne un nombre important de portes et donc une surface élevée. Dans ce cas, une réalisation *full-custom* peut permettre de tirer parti des caractéristiques du module pour diminuer considérablement la surface. On applique dans ce cas la même stratégie de conception que celle citée précédemment.

Pour l'unité d'addition, l'analyse des temps obtenus a nécessité un rééquilibrage des étages de l'opérateur d'addition. Les temps obtenus après une nouvelle optimisation sont de 16,31 ns pour le premier étage de l'additionneur et de 15,83 ns pour le second.

Lorsque toutes les décisions qui découlent de l'analyse des résultats sont prises, on effectue alors le placement-routage du module optimisé. Cette opération est réalisée

par l'outil CELL-ENSEMBLE de CADENCE. Cependant, pour passer de SYNOPSYS à CADENCE, il est nécessaire de convertir la *Netlist* générée par SYNOPSYS dans un format accepté par CADENCE. C'est le format EDIF qui est utilisé et bien qu'il soit normalisé, il existe des différences entre les deux logiciels. On a du réaliser un programme d'interface qui permette de convertir le format EDIF de SYNOPSYS dans celui de CADENCE.

Une fois la *Netlist* transférée sous CADENCE, on effectue le placement-routage. Celui-ci doit être fait généralement deux fois. La première permet de connaître la surface utilisée par le module afin d'esquisser un plan de masse du circuit. La seconde correspond à la même opération mais dans laquelle on a fixé des paramètres pour la géométrie du module afin qu'il respecte la place allouée sur le plan de masse définitif.

Pour l'unité d'addition, on a obtenu une surface totale de 12,11 mm^2. La géométrie de cette unité est définie dans le chapitre 7.

5.3 Unité de multiplication

5.3.1 Caractéristiques générales

Cette unité réalise les opérations de multiplication de deux blocs ou d'un bloc par un tableau ainsi que les opérations de multiplication-accumulation avec un bloc ou un tableau.

Cette unité est pipelinée sur 6 étages. Le premier étage est composé principalement du contrôleur d'unité. Les étages deux à cinq correspondent à l'opérateur de multiplication-accumulation et le sixième à l'écriture des résultats. Le débit de l'unité est de 1 bloc tous les 4 cycles, avec une latence de 6 cycles.

Le contrôleur d'unité génère les adresses et effectue les accès aux registres de donnée selon des séquences détaillées dans la figure 5.7.

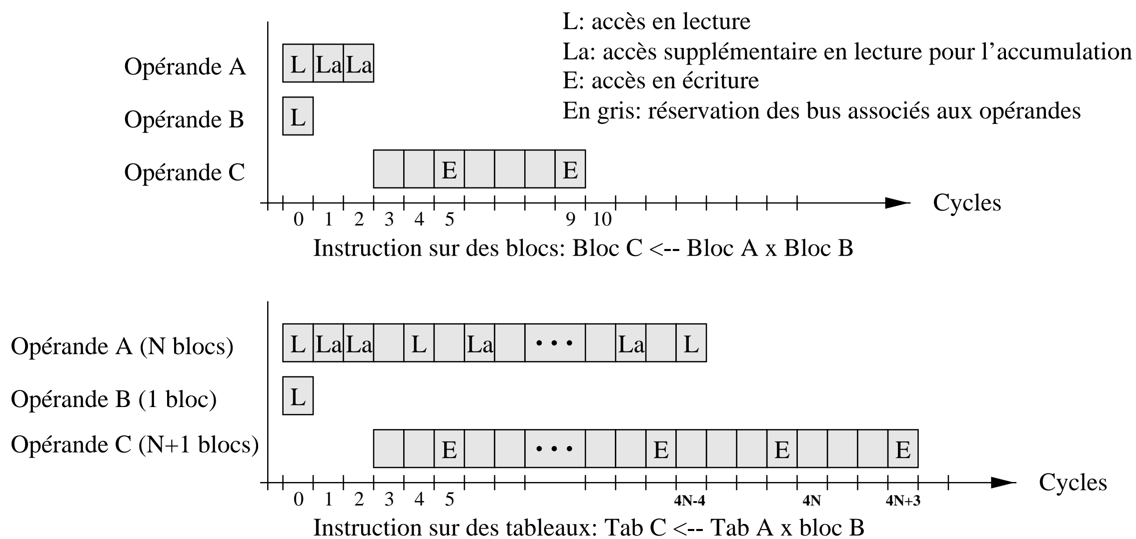


FIG. 5.7: Accès aux registres effectués par l'unité de multiplication.

5.3.2 Opérateur de multiplication-accumulation

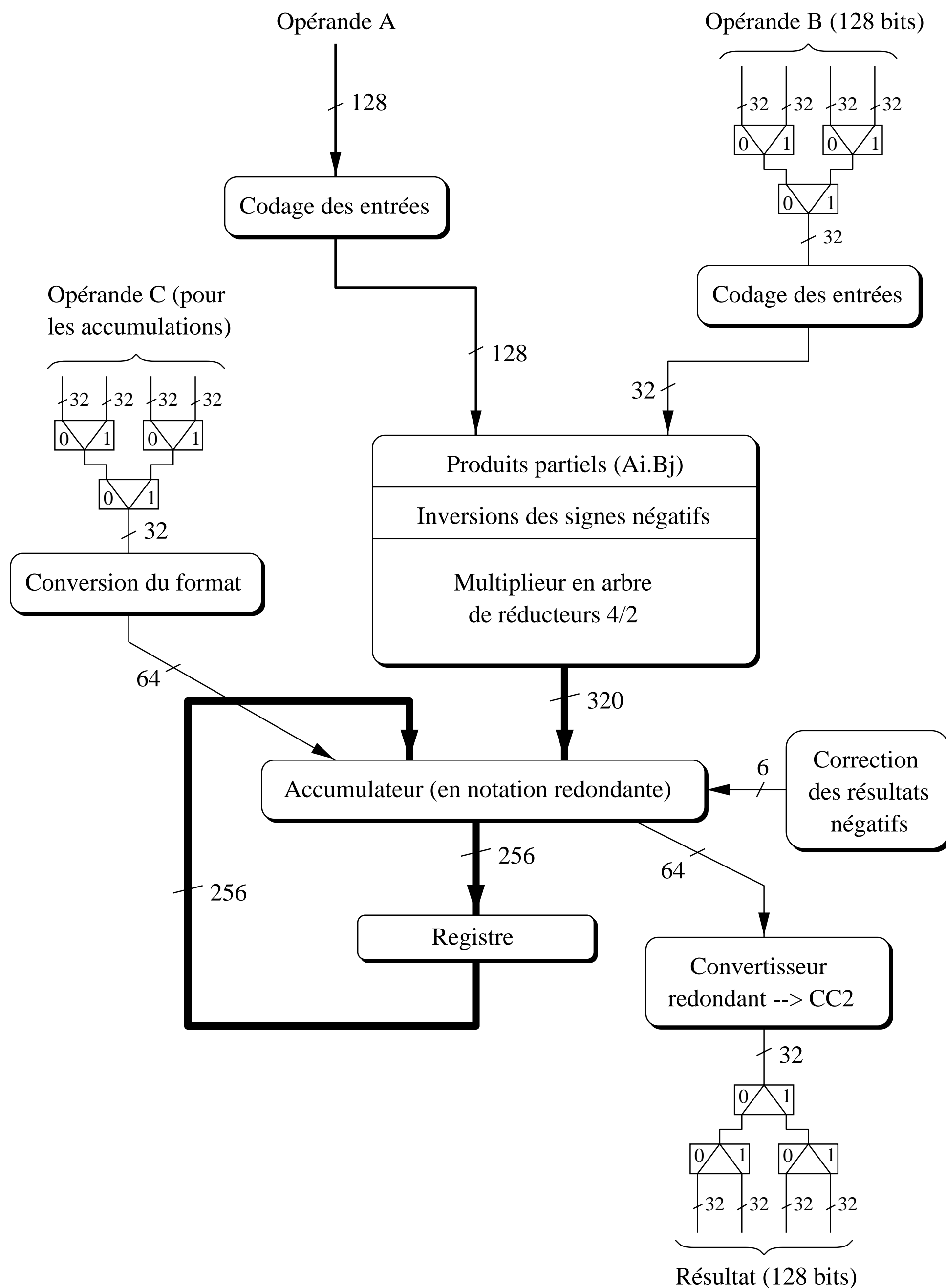


FIG. 5.8: Schéma général de l'opérateur de multiplication-accumulation.

On utilise pour le multiplieur une architecture régulière en arbre de réducteurs 4/2. Cette architecture utilise en entrée des opérandes en binaire classique (non signées) et retourne en sortie un résultat en notation redondante *Carry Save*. La réalisation d'un multiplieur 128×128 bits étant trop coûteuse en surface, on ne va implanter en matériel qu'un multiplieur 128×32 bits. La multiplication 128×128 bits est alors obtenue en effectuant quatre opérations successives et en accumulant les résultats intermédiaires. On utilise un accumulateur redondant pour ne pas avoir un temps

de cycle trop long car le résultat de l'accumulation, qui doit être réinjecté en entrée avec le résultat de la multiplication 128×32 bits, correspond au paramètre transmis lors des opérations sur les tableaux. Il doit donc être rebouclé en moins d'un cycle (Voir paragraphe 5.1.1).

La multiplication 128×32 bits suivie d'une accumulation génère à chaque cycle 32 bits du résultat définitif en notation redondante. On peut donc effectuer une reconversion en CC2 à chaque cycle en utilisant une architecture de type Han et Carlson sur 32 bits seulement. Cette solution est moins coûteuse en surface et plus rapide que de réaliser la reconversion sur 128 bits tous les quatre cycles. Le schéma général du multiplieur est donné figure 5.8. Les opérations de multiplication-accumulation utilisent l'accumulateur déjà présent dans l'architecture pour effectuer l'accumulation avec une nouvelle opérande d'entrée. Ces opérations seront décrites plus en détail dans le paragraphe 5.3.2.3.

5.3.2.1 Architecture du multiplieur

Le multiplieur utilise une architecture régulière en arbres de réducteurs 4/2 [68]. Dans une telle architecture, un arbre permet de faire l'addition de tous les produits partiels de même poids. Un réducteur 4/2 permettant d'additionner 4 bits, le multiplieur a donc des arbres de profondeur $(\log_2 \frac{N}{4})+1$ réducteurs 4/2, avec N correspondant à la taille de l'opérande B. Dans le cas du multiplieur 128×32 bits, on a des arbres constitués de 4 couches de réducteurs 4/2 comme le montre la figure 5.9.

La disposition des produits partiels sur les entrées du multiplieur est donnée dans la figure 5.10. On voit sur cette figure que l'architecture du multiplieur se décompose en trois parties : La partie centrale, totalement régulière, et les deux extrémités, la tête côté poids forts et la queue côté poids faibles qui présentent une moins grande régularité. Les câblages de la tête et de la queue sont présentés en annexe A.

Ce multiplieur retourne un résultat, en notation redondante, qui est accumulé avec celui de la multiplication précédente décalé de 32 bits (Figure 5.11). L'additionneur redondant qui effectue l'opération d'accumulation a la même architecture que celui utilisé dans l'unité d'addition (Figure 5.6). Dans ce cas il est constitué de 160 réducteurs 4/2 car la multiplication 128×32 bits retourne un résultat sur 160 bits.

Les 32 bits de poids faibles qui ont été décalés représentent une partie du résultat définitif de la multiplication 128×128 bits (Figure 5.11). Ils peuvent donc être reconvertis en CC2 dès qu'ils sont obtenus. On voit sur la figure 5.11 que la zone *S_l1* correspond aux 32 bits décalés de la première multiplication peuvent être reconvertis durant le cycle 2. Cette reconversion ne nécessite alors qu'un soustracteur sur 32 bits. Pour ne pas modifier le fonctionnement du pipeline, la reconversion du bloc de poids fort se fera aussi en 4 cycles par “paquets” de 32 bits. Il faut pour cela forcer la sortie de l'opérateur de multiplication à zéro tout en continuant d'effectuer les accumulations ce qui permet de vider l'accumulateur en 4 cycles.

5.3.2.2 Multiplieur signé

Pour effectuer des multiplications avec des opérandes codées dans les formats signés et non signés, il y a deux solutions :

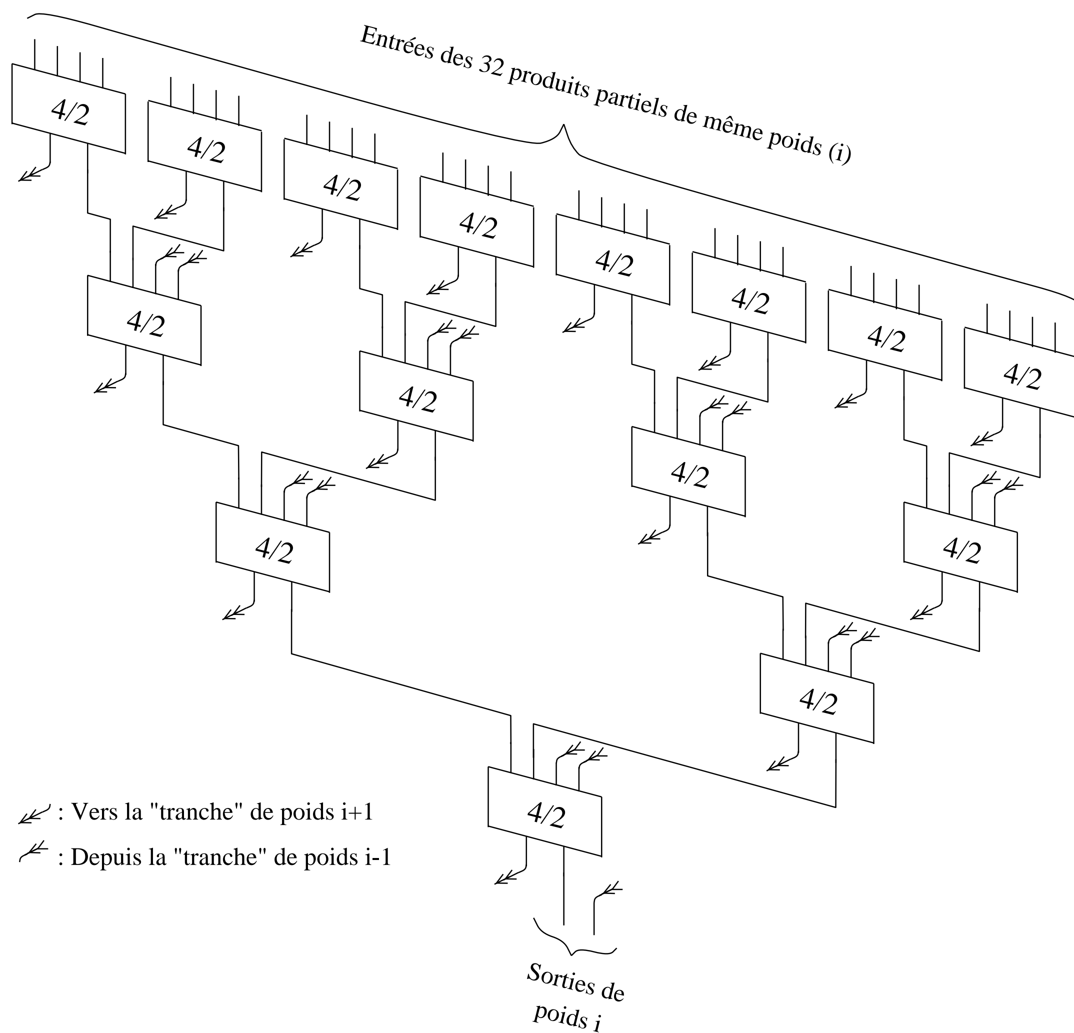


FIG. 5.9: Arbre de réducteurs 4/2.

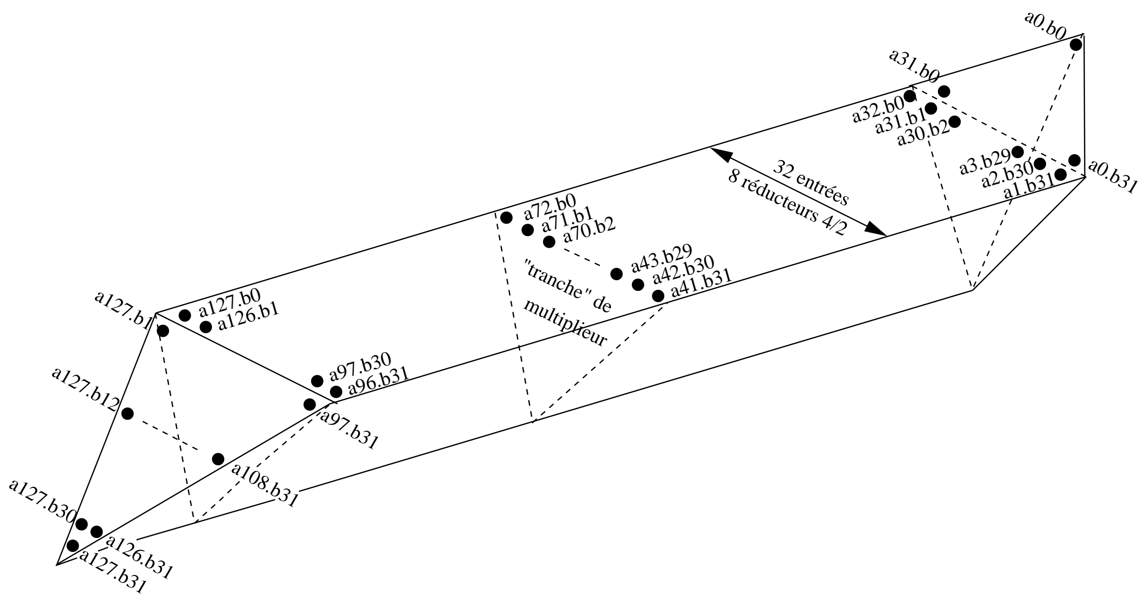


FIG. 5.10: Disposition des produits partiels.

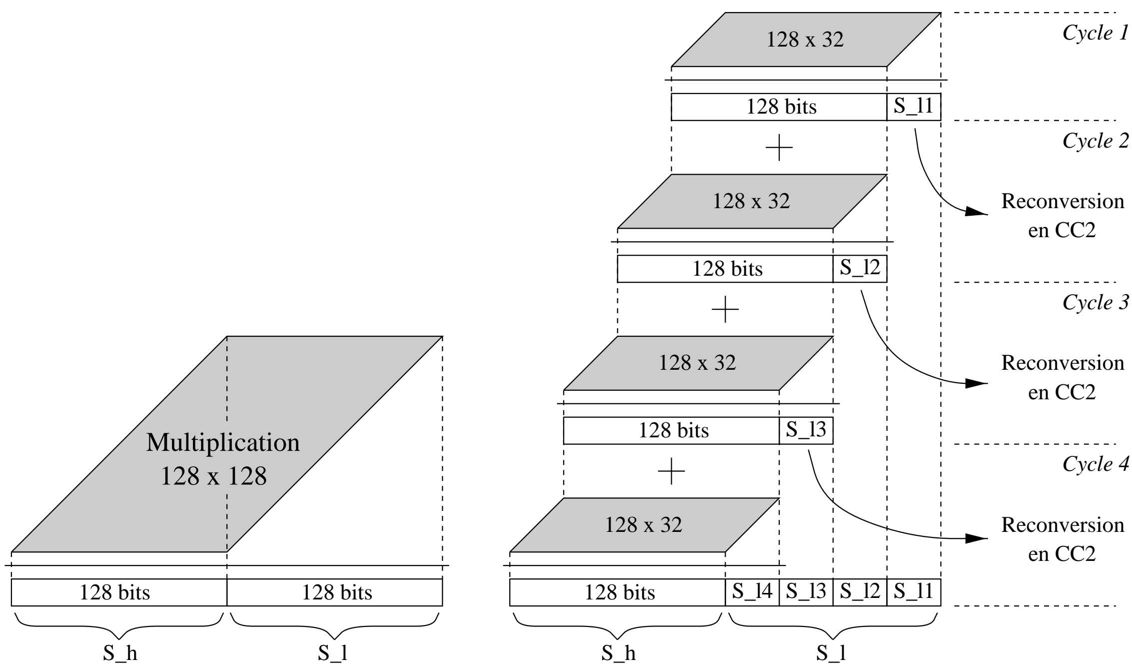


FIG. 5.11: Principe de l'accumulation et de la reconversion en CC2.

- Soit on code les deux formats en notation redondante puis on utilise un multiplieur redondant qui réalise les calculs indépendamment du format et du signe des opérandes.
- Soit on réalise un multiplieur non signé en notation binaire classique auquel on ajoute des portes logiques pour inverser la valeur des bits lorsqu'ils sont négatifs et permettre ainsi à cette architecture de traiter aussi les nombres signés.

La première solution est trop coûteuse en surface par rapport à la seconde pour être implémentée. En effet, par rapport à la notation binaire classique, la notation redondante double la taille des opérandes ce qui quadruple le nombre de produits partiels et nécessite un multiplieur quatre fois plus gros.

On réalise donc un multiplieur non signé avec des opérandes d'entrée en notation binaire classique. Cette architecture doit permettre de traiter aussi bien des nombres signés que non signés. Dans le cas d'opérandes signées, il faut inverser la valeur des bits qui ont des poids négatifs.

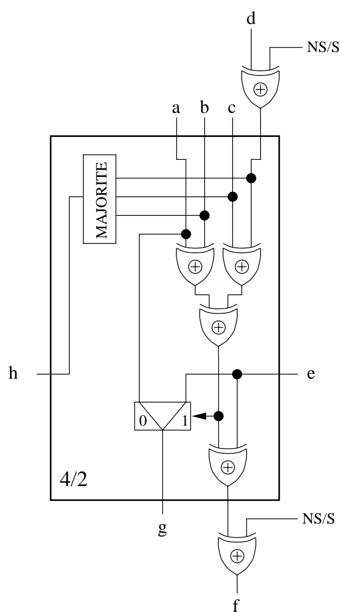


FIG. 5.12: Réducteur 4/2 acceptant une entrée signée ou non signée.

La figure 5.12 présente un réducteur 4/2 qui peut effectuer des opérations avec une entrée signée ou non signée, en fonction du signal NS/S qui indique le type de l'opérande. On remarque lorsque l'on réalise une "tranche" de multiplieur avec ces réducteurs, que seules les valeurs en entrée et en sortie de cette tranche doivent être inversées. A l'intérieur de l'architecture, l'inversion en sortie d'un réducteur s'annule avec celle en entrée du suivant.

Le tableau 5.7 présente, en fonction du format des d'entrées, les produits partiels qui sont négatifs et qui devront être inversés.

On notera $|A| = \sum_{i=0}^{127} |a_i| \times 2^i$ et $|B| = \sum_{i=0}^{31} |b_i| \times 2^i$.

Opérande A	Opérande B	Produits partiels négatifs
Non signée	Non signée	aucun
Non signée	Signée	$a_i \times b_{31} \quad i \in [0,127]$
Signée	Non signée	$a_{127} \times b_j \quad j \in [0,31]$
Signée	Signée	$a_i \times b_{31}$ *et* $a_{127} \times b_j \quad i \in [0,126] \quad j \in [0,30]$

TAB. 5.7: Produits partiels négatifs.

Les inversions des sorties du multiplieur qui résultent des quatre combinaisons d'entrée sont données figure 5.13. La disposition irrégulière des bits négatifs en sortie du multiplieur nécessite de pouvoir commander aussi l'inversion des entrées et des sorties de l'accumulateur. Cela demande un nombre élevé de portes OU-exclusif (inverseur commandé) et crée des lignes de commande très chargées. De plus cette solution ne permet pas de traiter les nombres négatifs car de nouvelles irrégularités apparaissent lors des accumulations successives et génèrent à chaque cycle une écriture différente pour les résultats qui n'est alors pas interprétable par le module de reconversion en CC2.

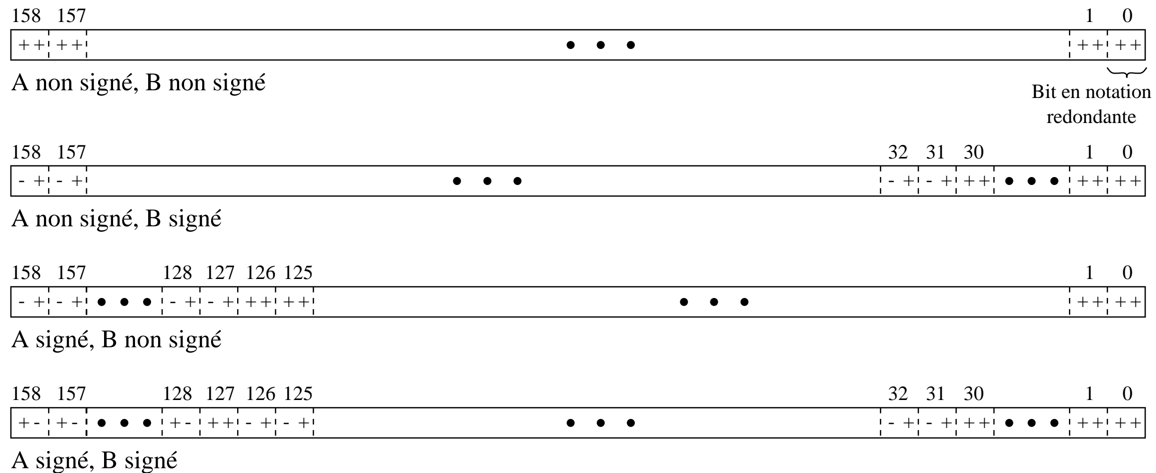


FIG. 5.13: Positions des bits négatifs en sortie du multiplieur.

On propose une solution qui peut réaliser des calculs sur des nombres signés et non signés pour un coût matériel faible. Elle est basée sur un nouveau câblage de l'accumulateur et sur la correction de certaines irrégularités dans la disposition des signes négatifs, permettant ainsi de retourner à chaque cycle des résultats en notation *Borrow Save* qui peuvent alors être recodés facilement en notation CC2.

Cette solution n'effectue pas de correction des signes en sortie du multiplieur ce qui réduit le coût en portes OU-exclusif et supprime les lignes de commande très chargées. On laisse ainsi des "erreurs" se propager, à chaque cycle, à travers l'accumulateur. Ce dernier utilise un nouveau câblage qui permet de limiter la propagation et la dispersion de ces "erreurs" lors des rebouclages. Cependant certaines "erreurs" subsistent et doivent être corrigées, soit parce qu'elles entraînent trop de perturbations, soit parce que l'on est à la fin du calcul. Ces corrections, qui s'appliquent seulement à 6 bits sur les 640 entrées de l'accumulateur, dépendent de 3 paramètres :

– Le signe des opérandes du multiplieur : NS/S_A et NS/S_B ainsi que NS/S_A^{ret4}

et NS/S_B^{ret4} qui correspondent aux signaux précédents retardés de 4 cycles.
- La position des 32 bits d'entrée du multiplieur dans l'opérande B. Cette position ($Pos0$ à $Pos3$) correspond à la commande des multiplexeurs de l'entrée B.
- Les signaux de début et de fin de l'instruction de multiplication : $Pcoup$ correspond à la première multiplication (4 cycles), $Dcoup$ à la dernière (4 cycles), et $Scoup$ aux 4 cycles supplémentaires pour générer le (N+1)ème bloc du résultat. Si la multiplication s'effectue sur un bloc ou un tableau de longueur 1, alors $Pcoup$ et $Dcoup$ sont valident simultanément.

Hormis NS/S_A^{ret4} et NS/S_B^{ret4} qui doivent être retardés de 4 cycles, tous les autres signaux sont déjà générés par le contrôleur d'unité et ne nécessitent pas de matériel supplémentaire. Les équations des 6 commandes qui permettent de faire les corrections des signes de façon à retourner un résultat en notation *Borrow Save* sont :

- $Cmd0 = NS/S_A^{ret4} \wedge \overline{NS/S_B^{ret4}} \wedge Scoup \wedge Pos0$
- $Cmd1 = NS/S_A \wedge \overline{NS/S_B} \wedge Dcoup \wedge Pos3$
- $Cmd2 = NS/S_A^{ret4} \wedge \overline{NS/S_B^{ret4}} \wedge Scoup \wedge Pos3$
- $Cmd3 = NS/S_B \wedge Pcoup \wedge Pos3$
- $Cmd4 = NS/S_B \wedge Dcoup \wedge Pos3$
- $Cmd5 = NS/S_A \wedge NS/S_B \wedge Dcoup \wedge Pos0$

Le schéma complet de l'accumulateur et des corrections est présenté figure 5.14.

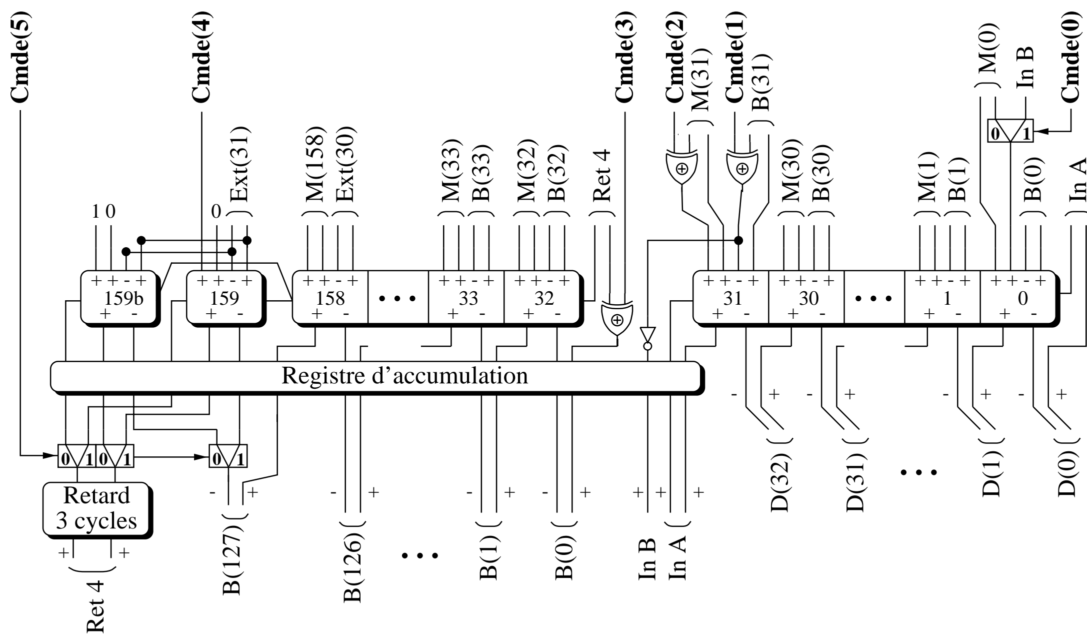
Cette solution permet donc de réaliser un multiplieur qui effectue des opérations sur des nombres signés ou non signés pour un coût supplémentaire faible car il faut seulement des portes OU-exclusif pour les produits partiels comportant un bit de signe et quelques portes logiques pour effectuer les corrections en sortie. Cette solution est générale et peut s'adapter à tous les multiplieurs qui utilisent une boucle avec accumulation pour effectuer l'opération en plusieurs cycles, et ceci quelle que soit la taille du multiplieur ou le nombre de boucles.

5.3.2.3 Opération de multiplication-accumulation

La multiplication-accumulation peut être réalisée avec le même matériel que celui utilisé pour la multiplication. Bien que l'accumulateur soit déjà utilisé pour les résultats partiels des multiplications 128×32 bits, on peut malgré tout l'utiliser pour accumuler les résultats des multiplications 128×128 bits avec une troisième opérande d'entrée. Il faut pour cela initialiser l'accumulateur avec le premier bloc de cette opérande, puis envoyer les blocs suivants, par "paquets" de 32 bits dans les poids forts de l'entrée rebouclée de l'accumulateur. En effet, à chaque cycle on décale de 32 bits vers la droite le précédent résultat de l'accumulation ce qui libère les 32 bits de poids forts qui peuvent alors être utilisés pour une accumulation avec une nouvelle donnée (Figure 5.15).

5.3.3 Synthèse et placement-routage

Pour la réalisation de cette unité, on utilise le même flot de conception que celui décrit pour l'unité d'addition (Paragraphe 5.2.3). L'unité de multiplication est



Cmde(5..0): Commandes permettant de réaliser une architecture soit signée, soit non signée
B(127..0) : Sorties de l'accumulateur rebouclées sur les entrées à travers un registre
M(158..0): Sortie du multiplieur
D(31..0) : Vers le convertisseur redondant-CC2
Ext(31..0): Entrées utilisées pour l'accumulation avec une troisième opérande

FIG. 5.14: Câblage de l'accumulateur.

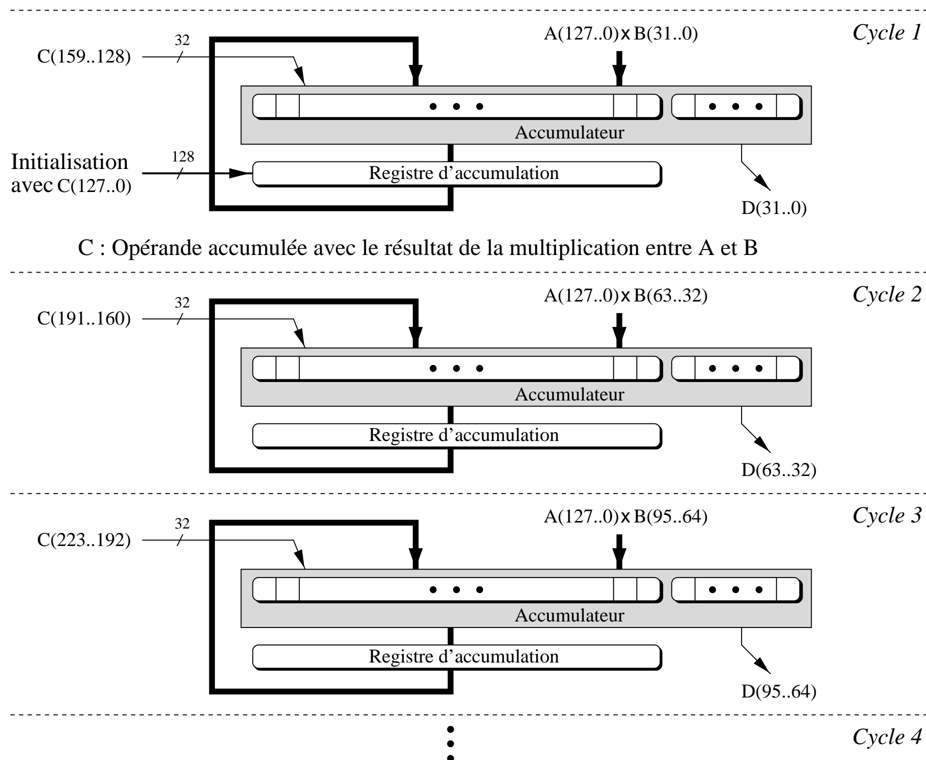


FIG. 5.15: La multiplication-accumulation avec une troisième opérande.

constituée de 6 étages de pipeline : le premier correspond à la lecture des registres, les quatre suivants à l'opérateur de multiplication (Figure 5.8) et le dernier à l'écriture des résultats dans le banc de registres. L'équilibrage des quatre étages de l'opérateur de multiplication (parmis lesquels se trouve le chemin critique de l'unité) a permis d'obtenir les résultats suivants : 9,10 ns pour le $2^{ème}$ étage, 18,90 ns pour le $3^{ème}$, 16,71 ns pour le $4^{ème}$ et 12,37 ns pour le $5^{ème}$. En anticipant sur les résultats des autres modules du circuit, on remarque que le $3^{ème}$ étage de cette unité, qui contient le multiplieur 128×32 bits, correspond au chemin critique du cœur du processeur. Pour diminuer ce temps, on va réaliser cet opérateur en *full-custom*. On obtient ainsi 6,7 ns pour ce $3^{ème}$ étage. Le multiplieur 128×32 bits en *full-custom* est décrit plus en détail dans les paragraphes suivants.

5.3.4 Optimisations

On verra dans le chapitre 7 que le chemin critique du processeur correspond au troisième étage de l'unité de multiplication. Cet étage contient le multiplieur 128×32 bits. La réalisation *full-custom* de cet opérateur permet de diminuer son temps de calcul et donc d'augmenter la fréquence de fonctionnement de l'ensemble. L'autre avantage de la réalisation *full-custom* du multiplieur est de diminuer fortement sa surface du fait de la régularité de son architecture.

On obtient alors un temps de traversée pour le multiplieur de 6,7 ns contre 18,9 ns pour la solution en cellules standards, soit un gain de 64 %. Concernant la taille de ces deux implantations, la solution *full-custom* occupe une surface de 14,54 mm^2 alors que la solution en cellules standards nécessite 23,49 mm^2. Le gain apporté au niveau de la surface par la réalisation *full-custom* en donc de 38,1 %.

L'utilisation d'une technologie possédant au moins trois niveaux de métallisation permettrait de diminuer d'environ 31 % la surface obtenue par la solution *full-custom* à deux niveaux de métal. L'architecture du multiplieur possède beaucoup de connexions qui doivent être insérées entre les cellules lorsque la technologie ne comporte pas un nombre élevé de niveaux de métallisation. Lorsque ce nombre est égal (ou supérieur) à trois, toutes les connexions entre les cellules peuvent alors être réalisées au dessus des cellules de base du multiplieur. Une telle technologie permettrait ainsi de diminuer fortement la surface du circuit sans modifier ni la structure ni les cellules de base du multiplieur.

5.3.4.1 Multiplieur régulier

Le multiplieur est implanté sous la forme d'une matrice de réducteurs 4/2, de 15 lignes par 159 colonnes. Suivant les lignes, on dispose les réducteurs 4/2 de poids consécutifs. Suivant les colonnes, on dispose les "tranches de multiplieur", c'est à dire les réducteurs 4/2 de même poids. Les portes *Nand* utilisées pour le calcul des produits partiels sont intégrées dans l'architecture ce qui permet de ne pas distribuer les 4096 produits partiels sur l'opérateur de multiplication mais seulement les 160 signaux correspondant aux 128 bits de l'opérande A et aux 32 bits l'opérande B.

On propose figure 5.16 un câblage régulier pour la propagation des bus de données. Cette régularité permet d'intégrer le câblage des bus au sein des réducteurs 4/2 ce qui réduit la place nécessaire pour la distribution des bus. Chaque cellule est ainsi

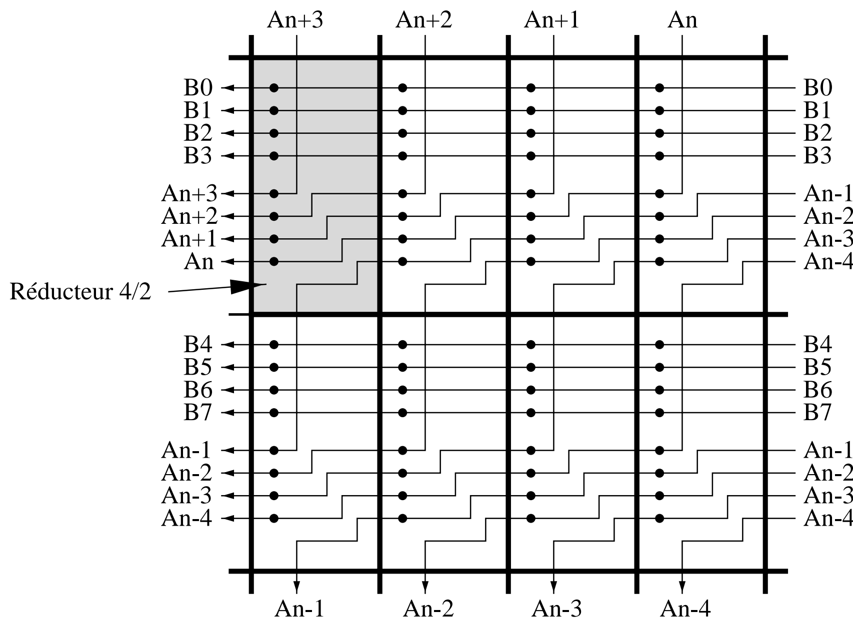


FIG. 5.16: Distribution des bus d'entrée dans les cellules du multiplieur.

traversée uniquement par les données dont elle a besoin pour calculer les produits partiels qui doivent être présents sur ses entrées (D'après figure 5.10). Cette solution réduit en particulier la taille du bus A qui n'utilise par ligne de réducteurs 4/2 qu'une surface équivalente à 4 fils .

5.3.4.2 "Tranche" de multiplieur

Une "tranche" de multiplieur est constituée d'une colonne de 15 réducteurs 4/2 qui réalisent l'addition des 32 produits partiels de même poids. La figure 5.17 présente le détail du câblage entre quelques réducteurs 4/2 appartenant à deux colonnes consécutives du multiplieur. Le câblage complet entre deux colonnes est donné en annexe A.

Le câblage utilisé est celui qui minimise la largeur du canal de routage associé à chaque "tranche". Les connexions des 45 sorties des réducteurs 4/2 nécessitent une largeur de canal qui permette de faire passer un maximum de 9 fils côte à côte.

5.3.4.3 Réducteur 4/2

Il existe deux types différents de réducteurs 4/2 pour la réalisation *full-custom* :

- Ceux dont les entrées proviennent de portes Nand. Ces cellules intègrent les 4 portes Nand.
- Ceux dont les entrées proviennent de réducteurs 4/2 des couches supérieures.

La réalisation *full-custom* du dessin des masques de ces deux types de cellules est donnée figure 5.18.

Pour diminuer la largeur du canal de routage d'une "tranche" de multiplieur, on a réalisé des réducteurs 4/2 en *full-custom* qui permettent de faire passer 4 lignes de métal (transparences) au dessus des cellules. On peut ainsi faire passer 4 fils du

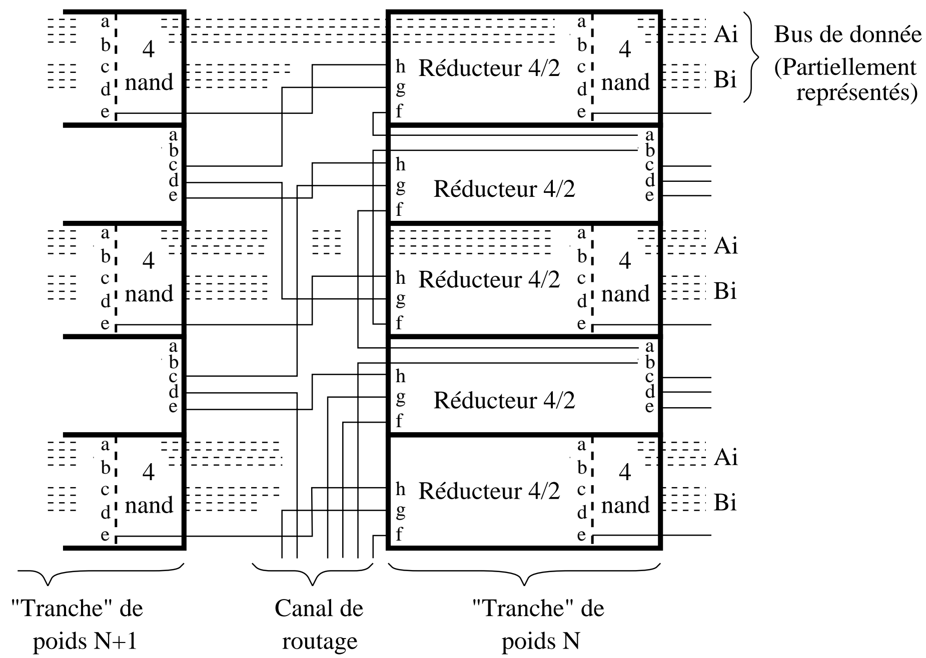
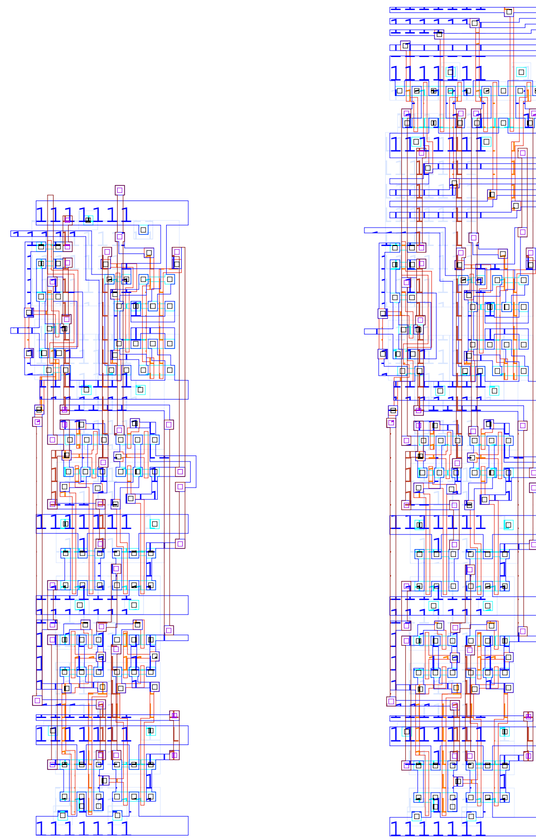


FIG. 5.17: Vue partielle du câblage entre deux colonnes de réducteurs 4/2 .



a) Sans portes Nand b) Avec 4 portes Nand

FIG. 5.18: Dessins des masques des réducteurs 4/2 en *full-custom*.

routage, parmis les 9 que contient le canal de routage, au dessus des réducteurs 4/2 de chaque colonne. Ceci réduit de près de 45% la largeur du canal de routage de chaque colonne. L'utilisation d'une technologie ayant au moins trois niveaux de métallisation, comme cela a été précisé au début du paragraphe 5.3.4, permettrait de supprimer totalement la surface du canal de routage entre chaque "tranche" en le faisant passer au dessus des cellules de réducteurs 4/2.

5.4 Unité de division

5.4.1 Caractéristiques générales

Cette unité réalise les opérations de division entre un double bloc pour le dividende et un bloc pour le diviseur. L'opérande d'entrée qui correspond au diviseur doit être normalisée afin d'assurer la convergence du calcul. L'utilisation d'un double bloc pour le dividende permet d'enchaîner facilement, au niveau logiciel, des instructions de divisions sur les blocs pour réaliser la division d'éléments plus importants. Les opérations de division sur les tableaux n'ayant pas été implantées en matériel, la division de deux grands nombres [36] ou le calcul du PGCD[1] [80], qui sont couramment employés en arithmétique exacte, doivent être réalisés en logiciel à partir de l'opération élémentaire de division sur les blocs.

L'opération de division est réalisée séquentiellement en effectuant une boucle à travers un étage de diviseur. Cet étage permet d'obtenir un bit du quotient par cycle. L'utilisation d'une boucle pour la division ne permet pas de pipeliner cette unité. Cependant, pour réduire le temps de cycle, les fonctions en dehors de la boucle peuvent être "découpées" en plusieurs étages. L'unité est donc constituée de quatre étages, avec la boucle au niveau du deuxième étage. Cette unité fournit un quotient et un reste sur 128 bits tous les 132 cycles.

Le contrôleur d'unité effectue les accès aux registres comme indiqué figure 5.19.

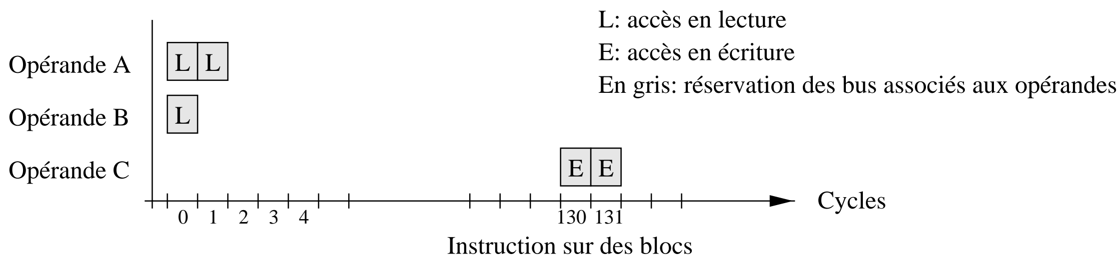


FIG. 5.19: Accès aux registres effectués par l'unité de division.

5.4.1.1 Quotient et reste exacts

La normalisation du diviseur entraîne des modifications au niveau du quotient et du reste par rapport aux résultats exacts que l'on aurait du obtenir. Il faut donc corriger ces résultats une fois le calcul terminé.

[1]Plus Grand Commun Diviseur.

Lorsque l'on réalise la division de A par B sans normaliser ce dernier, on obtient un quotient Q et un reste R tels que :

$$A = B \times Q + R \qquad \text{avec } -B < R < +B$$

La normalisation du diviseur, qui correspond à un décalage à gauche, génère un nouveau diviseur B' tel que :

$$B' = K \times B$$

La division avec le diviseur normalisé vérifie donc :

$$A = B' \times Q' + R' \qquad \text{avec } -B' < R' < +B'$$

$$\text{soit } -K \times B < R' < +K \times B$$

Cela signifie que dans certains cas on peut avoir $|R'| > |B|$. De plus, pour "dé-normaliser" Q', on le décale vers la gauche en poussant avec des zéros. La précision alors obtenue sur la valeur de Q' n'est pas maximale. On a donc dans ce cas là un calcul qui n'est pas fini. Plusieurs solutions existent pour résoudre ce problème mais à des coûts différents :

- On réinjecte le reste R' en poids forts du dividende, les poids faibles étant mis à zéro, et on exécute une nouvelle division. Le décalage étant inférieur à 128 bits, on est sûr en rajoutant un bloc supplémentaire d'avoir toute la précision sur Q' et donc sur R'. Cette solution nécessite une division supplémentaire soit 132 cycles.
- On peut réaliser un diviseur qui continue à effectuer des boucles dans son deuxième étage. Le nombre de boucles supplémentaires sera égal à la valeur de la normalisation qui a été effectuée sur B. On ne réalise ainsi que juste le nombre de cycles supplémentaires nécessaires. Cette solution est cependant complexe à implémenter en matériel car elle demande de transmettre des paramètres supplémentaires et retourne des résultats qui ne sont pas alignés sur des blocs.
- La troisième solution consiste à décaler à gauche le dividende de la même valeur que l'on a décalé le diviseur. On obtient ainsi directement Q et R. Dans ce cas l'algorithme de division doit donc normaliser le diviseur, décaler à gauche le dividende de la même valeur que pour la normalisation puis effectuer la division. Le décalage à gauche du dividende doit tenir compte des débordements éventuels sur un bloc supplémentaire. Cette solution est cependant la moins coûteuse à mettre en œuvre.

5.4.2 Opérateur de division

L'opérateur de division doit pouvoir réaliser les opérations sur des opérandes dans les formats signés ou non signés. L'architecture utilisée a un dividende codé en notation redondante et un diviseur normalisé codé en CC2. Pour le dividende, une interface logique permet de coder facilement les deux formats en notation redondante. Pour le diviseur, c'est la normalisation qui génère un codage identique pour les deux formats (Voir paragraphe 5.5.2.3).

L'opérateur de division utilisé [54] est basé sur des additions/soustractions et des décalages. Il est constitué de quatre parties :

- Une interface qui code les différents formats du dividende en notation redondante avant de le transmettre à l'opérateur.
- Une "tranche de 1 bit" qui se compose d'une cellule de tête et de 128 cellules de queue. La cellule de tête détermine, à partir des premiers bits du reste partiel précédent, l'opération à effectuer sur ce reste, à savoir additionner ou soustraire le diviseur ou "ne rien faire". Le choix effectué par la cellule de tête est transmis aux 128 cellules de queue qui calculent le nouveau reste partiel. Ces cellules sont décrites plus en détail dans [71]. Cette "tranche", constituée d'une cellule de tête et de 128 cellules de queue, permet de générer 1 bit du quotient. Un diviseur purement combinatoire, constitué de 128 "tranches de 1 bit", est cependant trop coûteux en surface pour pouvoir être implanté. La division sur 128 bits est alors réalisée en utilisant une seule "tranche de 1 bit" sur laquelle on effectue une boucle pour obtenir, au bout de 128 itérations, tous les bits du quotient. Pour cela, on réinjecte à chaque cycle le reste partiel dans le dividende. Lors de cette boucle, on décale le reste partiel d'un bit vers la gauche et on ajoute en poids faible un nouveau bit provenant du bloc de poids faible du dividende initial.
- Un module de reconversion "à la volée" du quotient. Les bits en notation redondante obtenus à chaque cycle sont reconvertis en CC2 au fur et à mesure qu'ils arrivent.
- Un module de reconversion du reste. Tous les bits (en notation redondante) du reste étant obtenus en parallèle dans le dernier cycle, on utilise un soustracteur de type Han et Carlson pour effectuer la reconversion en CC2.

Un schéma général de l'opérateur de division est donné figure 5.20.

5.4.2.1 Reconversion du quotient

Les bits du quotient qui sont retournés à chaque cycle sont en notation redondante. Pour la reconversion en CC2, les bits du quotient étant obtenus en commençant par les poids forts, on réalise une reconversion "à la volée", c'est à dire au fur et à mesure que le nouveau bit en redondant arrive [22]. Cette reconversion calcule en parallèle deux résultats Q_n et $Q_n^{-n} = Q_n - 2^{-n}$ car lorsqu'un nouveau bit en redondant vaut -1, cela peut entraîner une propagation de retenue et modifier l'ensemble des bits en CC2 déjà calculés. A chaque nouveau bit redondant q_{n+1} on a donc les relations suivantes :

$$
\begin{array}{llll}
\text{Si } q_{n+1} = +1 & \text{alors} & Q_{n+1} = Q_n + 2^{-(n+1)} & \text{et} \quad Q_{n+1}^{-n} = Q_n \\
\text{Si } q_{n+1} = 0 & \text{alors} & Q_{n+1} = Q_n & \text{et} \quad Q_{n+1}^{-n} = Q_n^{-n} + 2^{-(n+1)} \\
\text{Si } q_{n+1} = -1 & \text{alors} & Q_{n+1} = Q_n^{-n} + 2^{-(n+1)} & \text{et} \quad Q_{n+1}^{-n} = Q_n^{-n}
\end{array}
$$

La figure 5.21 présente la réalisation matérielle de cette reconversion. Le bit redondant q_{n+1} est représenté sous la forme de deux signaux q_{n+1}^+ et q_{n+1}^- tels que $q_{n+1} = q_{n+1}^+ - q_{n+1}^-$.

5.4.2.2 Reconversion du reste

Le reste retourné par les cellules de queue du diviseur est aussi en notation redondante. Tous les bits de ce reste étant calculés en parallèle, on utilise pour la

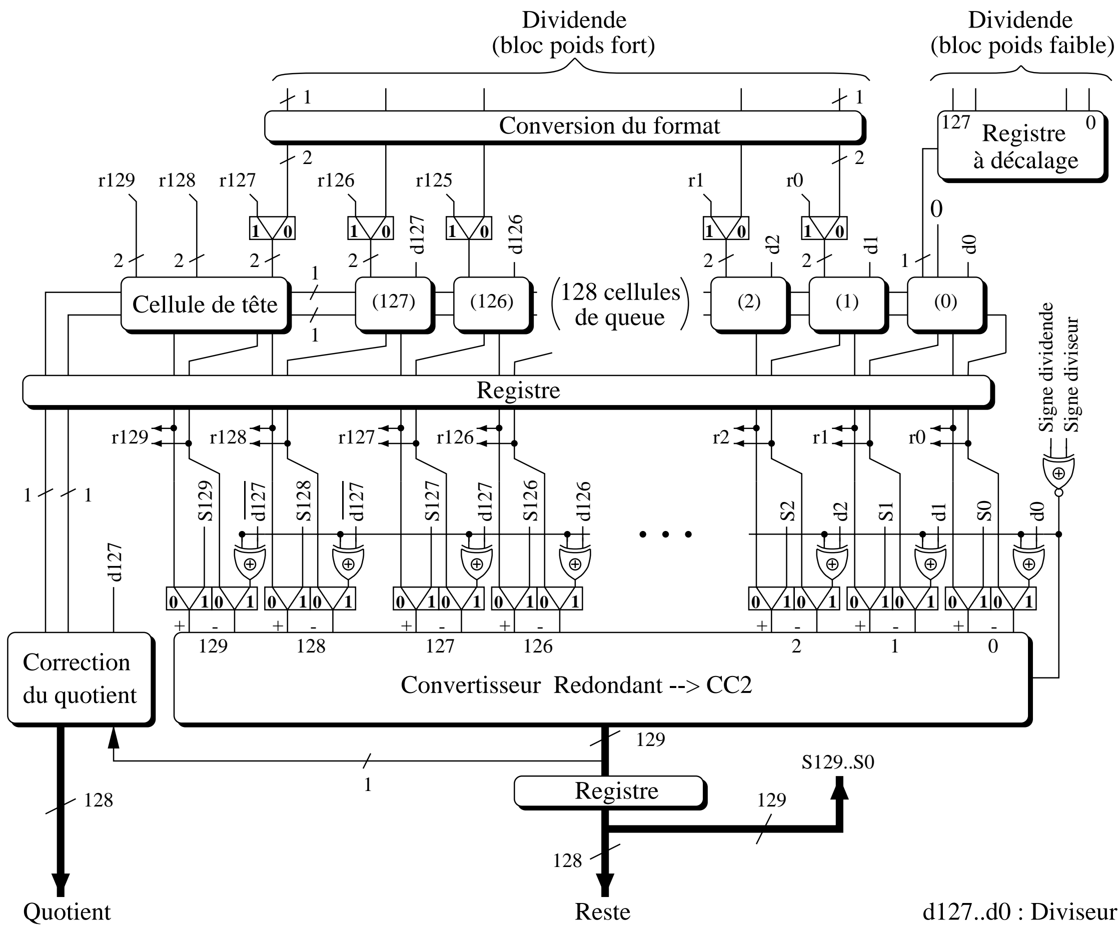


FIG. 5.20: Schéma général de l'opérateur de division.

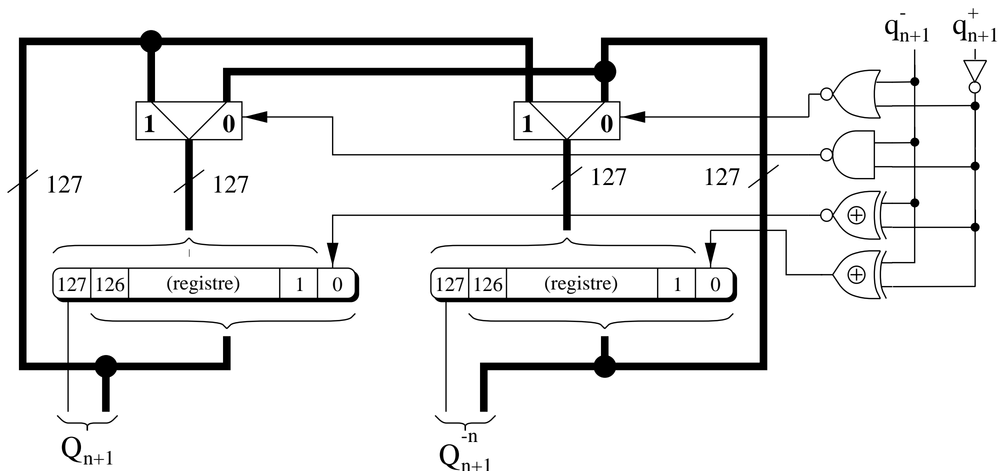


FIG. 5.21: Schéma de la reconversion "à la volée".

reconversion un soustracteur de type Han et Carlson. Ce soustracteur est “découpé” en deux étages de manière à maintenir un temps de cycle faible.

5.4.2.3 Signe du reste

Par convention, le signe du reste doit être le même que le signe du dividende. Or la structure de la “tranche de 1 bit” permet au reste partiel d’être négatif, c’est à dire que l’on peut soustraire le diviseur au dividende même si la valeur du dividende est plus faible que celle du diviseur. Le codage du quotient en notation redondante avec des chiffres signés permet de “compenser” au cycle suivant la valeur du bit précédent. Si ce cas où se produit lors de la dernière itération, la “compensation” n’a pas lieu. Le signe du reste est alors différent de celui du dividende et la valeur absolue du quotient trop grande d’une unité. Il faut donc faire une correction du reste et du quotient qui sont retournés.

1. **Correction du reste :**

   Lorsque le signe du reste est différent de celui du diviseur, alors le reste doit être corrigé. Deux cas se présentent selon que les signes du dividende et du diviseur soient égaux ou non :

   Si Signe dividende = Signe diviseur alors Reste ⇐ Reste + Diviseur

   Si Signe dividende ≠ Signe diviseur alors Reste ⇐ Reste - Diviseur

   Pour effectuer les opérations d’addition ou de soustraction, on réutilise le soustracteur de type Han et Carlson qui a servi à convertir le reste.

   - Pour la soustraction, on reboucle le reste sur une de ses entrées, sur l’autre on applique la valeur du diviseur.
   - Pour l’addition, on utilise la relation :

   $$A + B = A - (-B) = A - \overline{B} - 2^{-n}$$

   On applique donc dans ce cas la valeur complémentée du diviseur sur la deuxième entrée du soustracteur et un 1 sur la retenue entrante.

   Il faut deux cycles supplémentaires pour obtenir le reste corrigé car le soustracteur est réalisé sur deux étages.

   Dans le cas ou le reste ne doit pas être corrigé (Signe reste = Signe dividende), alors on soustrait la valeur zéro au reste.

2. **Correction du quotient :**

   Une correction du reste s’accompagne automatiquement d’une correction du quotient. Lorsque le signe du reste est différent de celui du diviseur, il faut soustraire 1 au quotient lorsque ce dernier est positif et lui additionner 1 lorsqu’il est négatif. La reconversion “à la volée” du quotient, paragraphe 5.4.2.1, effectue déjà le calcul de $Q_n^{-n} = Q_n - 2^{-n}$. On va rajouter lors de cette reconversion une fonction qui calcule $Q_n^{+n} = Q_n + 2^{-n}$. A chaque nouveau bit redondant q_{n+1} on aura donc les relations suivantes :

$$\text{Si } q_{n+1} = +1 \quad \text{alors} \quad \begin{aligned} Q_{n+1}^{+n} &= Q_n^{+n} \\ Q_{n+1} &= Q_n + 2^{-(n+1)} \\ Q_{n+1}^{-n} &= Q_n \end{aligned}$$

$$\text{Si } q_{n+1} = 0 \quad \text{alors} \quad \begin{aligned} Q_{n+1}^{+n} &= Q_n + 2^{-(n+1)} \\ Q_{n+1} &= Q_n \\ Q_{n+1}^{-n} &= Q_n^{-n} + 2^{-(n+1)} \end{aligned}$$

$$\text{Si } q_{n+1} = -1 \quad \text{alors} \quad \begin{aligned} Q_{n+1}^{+n} &= Q_n \\ Q_{n+1} &= Q_n^{-n} + 2^{-(n+1)} \\ Q_{n+1}^{-n} &= Q_n^{-n} \end{aligned}$$

Le schéma du module de reconversion qui calcule Q_n, Q_n^{-n} et Q_n^{+n} est donné figure 5.22.

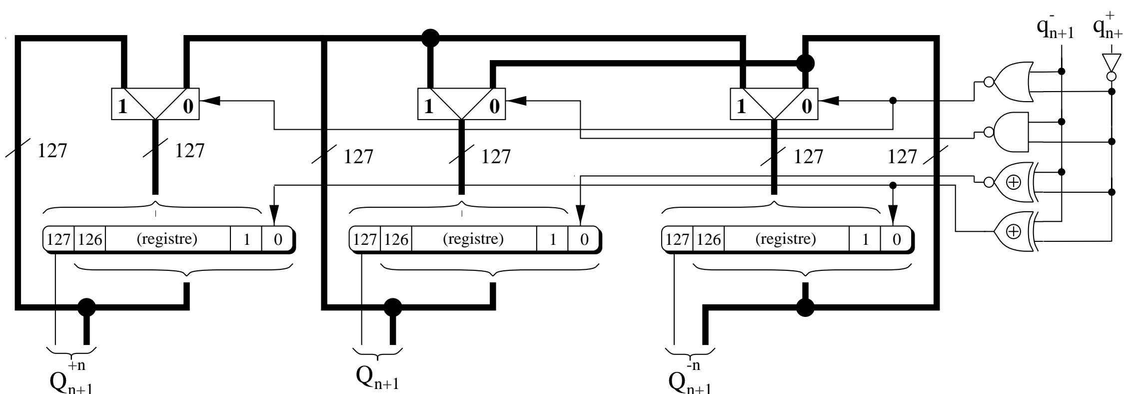


FIG. 5.22: Reconversion “à la volée” qui calcule Q_n, Q_n^{-n} et Q_n^{+n}.

Selon le signe du reste obtenu, il faut ensuite sélectionner parmi ces trois valeurs celle qui correspond à la bonne correction pour le quotient. On a trois possibilités :

- Cas N°1 : Signe reste = Signe dividende
  $\Rightarrow$ on doit sélectionner Q_n.
- Cas N°2 : (Signe reste $\neq$ Signe dividende) et (quotient positif)
  $\Rightarrow$ on doit sélectionner Q_n^{-n}.
- Cas N°3 : (Signe reste $\neq$ Signe dividende) et (quotient négatif)
  $\Rightarrow$ on doit sélectionner Q_n^{+n}.

Pour aiguiller en sortie la bonne valeur du quotient, on va se servir des multiplexeurs utilisés pour la reconversion “à la volée”. A chaque cycle, les signaux Q_n, Q_n^{-n} et Q_n^{+n} sont rebouclés sur eux mêmes à travers des multiplexeurs. En forçant la commande de ces multiplexeurs pendant les deux cycles supplémentaires dus à la correction du reste, il est possible de toujours aiguiller la bonne valeur du quotient sur le même bus de sortie (Figure 5.23). Cette solution permet de ne pas utiliser de matériel supplémentaire pour sélectionner la bonne valeur du quotient.

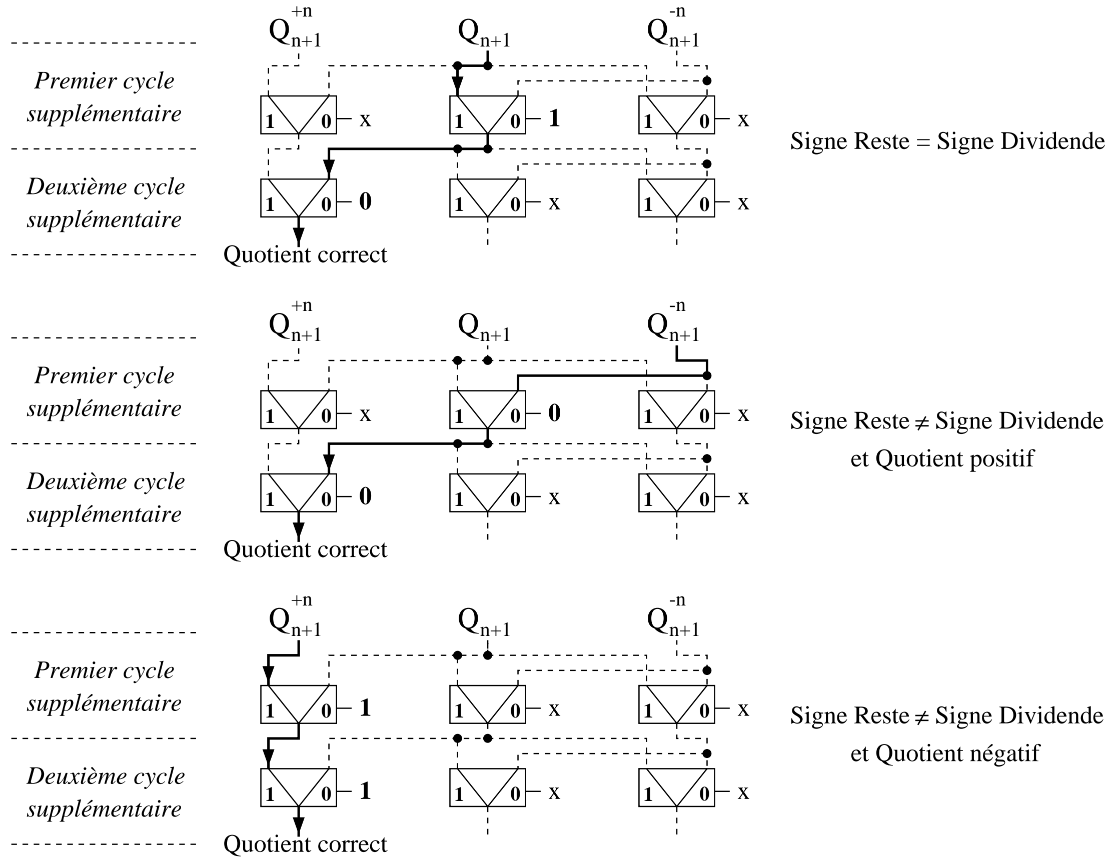


FIG. 5.23: Correction du quotient.

5.4.3 Synthèse et placement-routage

L'unité de division est constituée de cinq étages. Le premier correspond à la lecture des registres et le second, qui est rebouclé sur lui même, réalise la division. Les 3ème et 4ème étages effectuent la reconversion du reste et du quotient et le dernier écrit les résultats dans le banc de registres.

Après synthèse, on a obtenu 17,23 ns pour le 2ème étage, 17,25 ns pour le 3ème et enfin 16,63 ns pour le 4ème. Les 2ème et 3ème étages sont sensiblement équivalents et correspondent au nouveau chemin critique du cœur du processeur (le multiplieur ayant été réalisé en *full-custom*). Ces étages sont principalement composés de portes élémentaires et ne comportent pas de gros modules pour lesquels une solution *full-custom* permettrait de gagner du temps. De plus le 2ème étage est rebouclé sur lui même ce qui ne permet pas de le décomposer en deux étages moins longs. Cette unité a donc été entièrement réalisée en cellules standards. Sa surface obtenue après placement-routage est de 12,35 mm^2.

5.5 Unité de décalage

5.5.1 Caractéristiques générales

Cette unité réalise les opérations de décalage à gauche, à droite ou arithmétique sur des blocs ou des tableaux. La valeur du décalage est compris entre 0 et 127 bits. L'unité de décalage effectue aussi les opérations de normalisation d'un bloc ou d'un tableau. La normalisation consiste à faire un décalage vers la gauche pour éliminer les bits de poids forts qui ne représentent qu'une extension du bit de signe. La gestion des longueurs élimine les blocs qui ne contiennent que des bits identiques au bit de signe, de façon à minimiser la longueur des tableaux. Ainsi la normalisation ne correspond qu'à des décalages compris aussi entre 0 et 127 bits. La valeur du décalage pour les opérations de normalisation est détectée automatiquement par l'unité.

Cette unité est pipelinée sur 4 étages. Le premier étage est composé principalement du contrôleur d'unité. Les deuxièmes et troisièmes étages correspondent à l'opérateur de décalage/normalisation et le quatrième à l'écriture des résultats. Le débit de l'unité est de 1 bloc par cycle, avec une latence de 4 cycles.

Le contrôleur d'unité génère les adresses et effectue les accès aux registres de donnée selon des séquences qui varient en fonction de l'opération effectuée. Ces séquences sont détaillées dans la figure 5.24.

5.5.2 L'opérateur de décalage/normalisation

Un schéma général de l'opérateur de décalage/normalisation est donné figure 5.25. Les différents modules qui le composent sont développés dans les paragraphes suivants. Les signaux *dec/norm* et *gauche/droite* proviennent du décodage et indiquent respectivement une opération de décalage ou de normalisation et un décalage à gauche ou à droite.

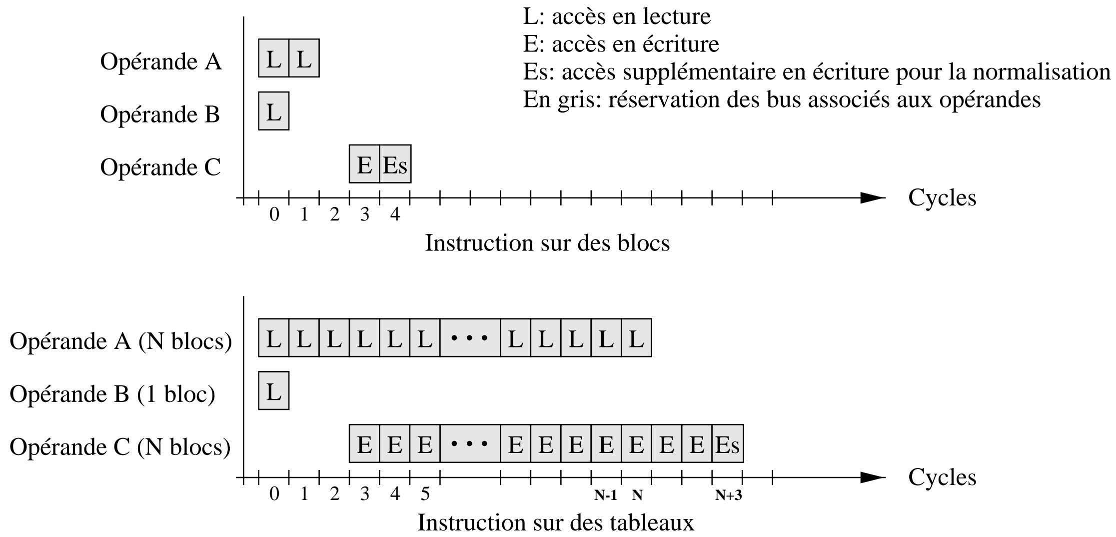


FIG. 5.24: Accès aux registres effectués par l'unité de décalage.

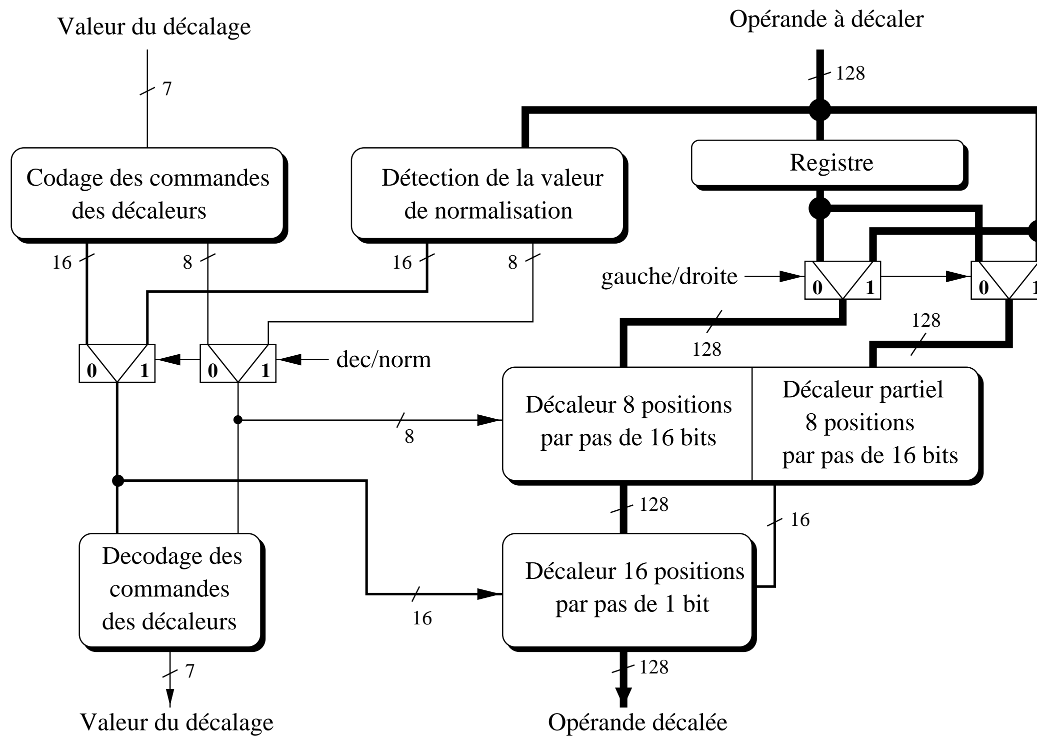


FIG. 5.25: Schéma général de l'opérateur de décalage/normalisation.

5.5.2.1 Le décaleur

Afin de diminuer la taille du décaleur, la solution matérielle implantée effectue seulement des décalages à gauche. Les décalages à droite sont alors réalisés en utilisant le complément par rapport à 128 de la commande du décaleur (Figure 5.26). La réalisation des décalages se fait donc de la façon suivante :

- Décalage à gauche : Lecture des blocs dans l'ordre décroissant des adresses (blocs poids forts en tête), lecture du bloc supplémentaire pour "pousser" à la fin. La commande du décalage est envoyée directement au décaleur.
- Décalage à droite : Lecture des blocs dans l'ordre croissant des adresses (blocs poids faibles en tête), lecture du bloc supplémentaire pour "pousser" à la fin. La commande du décalage est complémentée avant d'être envoyée au décaleur.

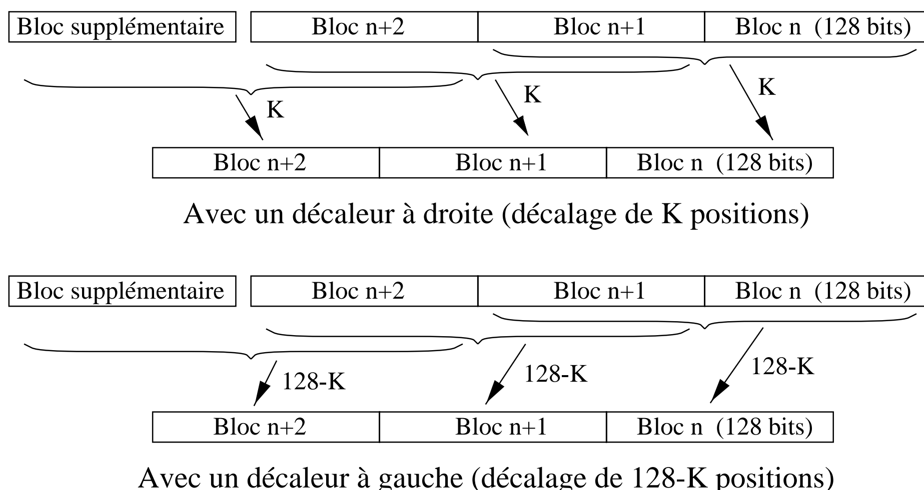


FIG. 5.26: Réalisation de décalages à droite.

L'utilisation d'un décaleur à gauche permet de réaliser directement les opérations de normalisation sans avoir à inverser la commande. Cette solution permet de ne pas augmenter le chemin critique des opérations de normalisation contrairement à celle d'un décaleur à droite.

La réalisation d'un décaleur à gauche de 128 bits par pas de 1 bit est cependant encore trop coûteuse en place. On va utiliser une solution avec deux décaleurs en série. Cette solution augmente le temps de l'opérateur de décalage mais permet de réduire de manière significative sa surface. Le premier décaleur va effectuer un décalage de M positions, le second un décalage de N positions tel que $M \times N = 128$. On considère que la surface d'un décaleur augmente linéairement en fonction du nombre de cellules élémentaires servant de décaler 1 bit d'un nombre fixe de positions. Dans le cas du décaleur de 128 bits par pas de 1 bit, la surface en nombre de cellules élémentaires est :

$$S1 = 128 \times 128$$

Pour la solution avec deux décaleurs en série, la surface en nombre de cellules élémentaires est :

$$S2 = (128 \times M) + (128 \times N)$$

$$S2 = (128 \times M) + \frac{128^2}{M}$$

La surface $S2$ est minimum pour $M = N = \sqrt{128}$.

On prend pour M et N les valeurs correspondant à des puissances de 2 les plus proches de $\sqrt{128}$ soit $M = 8$ et $N = 16$. La surface, en nombre de cellules élémentaires, de cette solution est donc :

$$S2 = (128 \times 8) + (128 \times 16)$$
$$S2 = 128 \times 24$$

Cette solution à deux décaleurs a une surface environ 5,3 fois plus faible que la solution avec un seul décaleur de 128 bits.

5.5.2.2 Le codage des commandes

L'utilisation d'une solution avec deux décaleurs nécessite de coder la valeur du décalage, comprise entre 0 et 127, en deux valeurs comprises entre 0 et 7 pour le premier décaleur et entre 0 et 15 pour le second. Il faut aussi prendre le complément par rapport à 128 de cette valeur lorsque l'on veut effectuer un décalage à droite. Les décaleurs utilisent des commandes en codage "1 parmi n" : le premier décaleur a donc une commande sur 8 bits (avec un seul bit à 1), le second une commande sur 16 bits (avec un seul bit à 1). Le tableau 5.8 présente quelques exemples de codage.

Décalage	Commande 1^{er}décaleur	Commande 2eme décaleur
Gauche, 0 position	0000 0001	0000 0000 0000 0001
Gauche, 1 position	0000 0001	0000 0000 0000 0010
Gauche, 32 positions	0000 0100	0000 0000 0000 0001
Gauche, 33 positions	0000 0100	0000 0000 0000 0010
Gauche, 127 positions	1000 0000	1000 0000 0000 0000
Droite, 0 position	0000 0001	0000 0000 0000 0001
Droite, 1 position	1000 0000	1000 0000 0000 0000
Droite, 32 positions	0100 0000	0000 0000 0000 0001
Droite, 33 positions	0010 0000	1000 0000 0000 0000
Droite, 127 positions	0000 0001	0000 0000 0000 0010

TAB. 5.8: Exemples de codage pour la commande des décaleurs.

On propose une solution qui se décompose en deux parties : la première génère le codage "1 parmi n" à partir des équations logiques des sorties provenant de la table de vérité. La seconde partie, qui effectue l'inversion de la commande par rapport à 128 en fonction du signal $gauche/droite$, est réalisée à base de multiplexeurs. Cette solution, présentée figure 5.27 est en effet plus rapide que la solution classique qui consiste à inverser chacun des bits puis à additionner 1.

5.5.2.3 La fonction normalisation

Cette fonction doit détecter la valeur du décalage à gauche qu'il faut effectuer sur un nombre pour éliminer en poids forts tous les bits qui ne correspondent qu'à une extension du bit de signe. Cela consiste donc à détecter le premier bit à 1 dans le cas

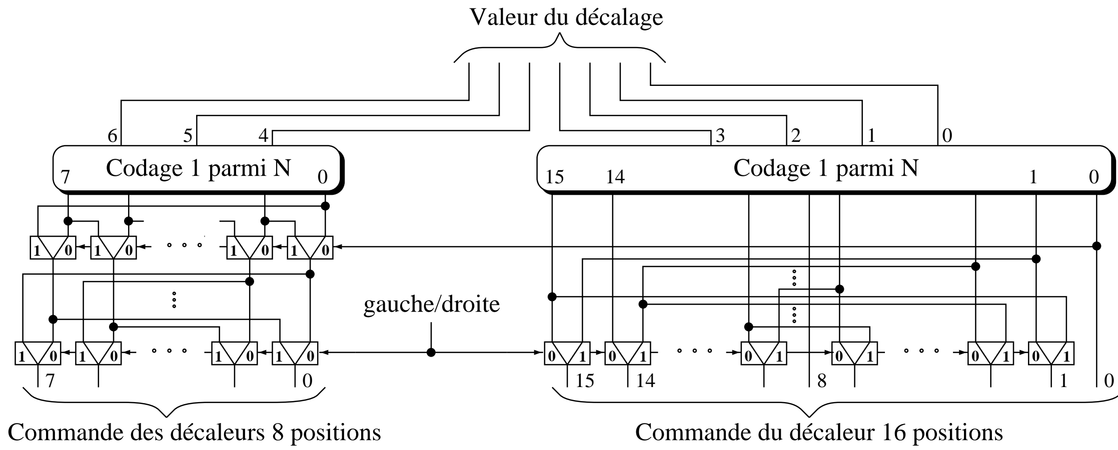


FIG. 5.27: Réalisation matérielle de la commande des décaleurs.

des nombres positifs ou non signés (d'après le bit NS/S associé à chaque bloc), et le premier bit à 0 dans le cas des nombres négatifs. La position de ce bit est ensuite codée puis envoyée sur les commandes des décaleurs qui effectuent le décalage du nombre à normaliser.

Afin d'utiliser le même matériel pour la normalisation des nombres positifs et négatifs, la détection pour ces derniers s'effectue sur le nombre complémenté bit à bit. C'est le bit de signe du nombre qui commande l'inversion ou non de tous les bits. L'inconvénient de cette solution est de générer une ligne très chargée (127 portes OU-exclusif) sur le chemin critique ce qui nécessite d'ajouter des amplificateurs.

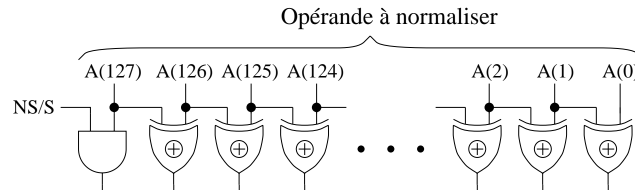


FIG. 5.28: Détection d'un changement de valeur entre deux bits consécutifs.

Pour supprimer ce problème, on propose de détecter non pas le premier bit à 1 ou à 0 mais la première variation (0 puis 1 ou 1 puis 0) entre deux bits consécutifs. Cette variation étant indépendante du signe, elle ne nécessite donc pas d'inversion pour les nombres négatifs. L'implémentation de cette solution est présentée figure 5.28. L'information rajoutée en poids fort permet de prendre en compte la première variation selon que le nombre est non signé, signé positif ou signé négatif.

La valeur de normalisation correspond donc à la première variation à partir des poids forts. Pour détecter sa position dans un bloc de 128 bits, on utilise deux étages de fonctions priorités (Figure 5.29). Cette fonction retourne un 1 à la position du premier bits non nul à partir des poids fort, et des zéros dans tous les autres bits. Le premier étage correspond à 8 fonctions priorités sur 16 bits qui détectent dans chaque groupe de 16 bits la position du premier bit à 1 lorsqu'il y en a un. Le deuxième étage détecte lequel de ces groupes est le premier non nul : son résultat sert de commande pour le premier décaleur de 8 positions. La commande du deuxième décaleur de

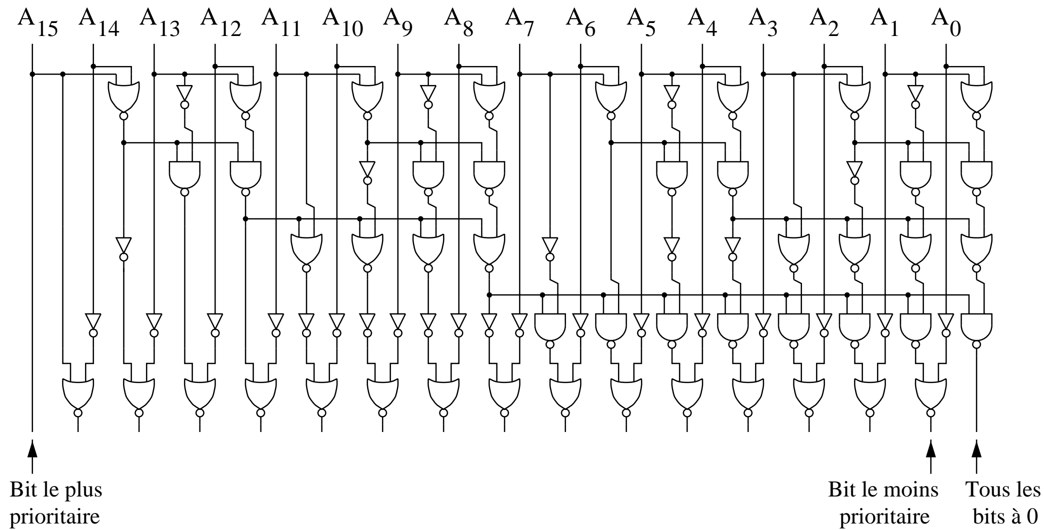


FIG. 5.29: Fonction priorité sur 16 bits.

16 positions correspond au résultat de la fonction priorité sur le premier groupe de 16 bits non nul (Figure 5.30).

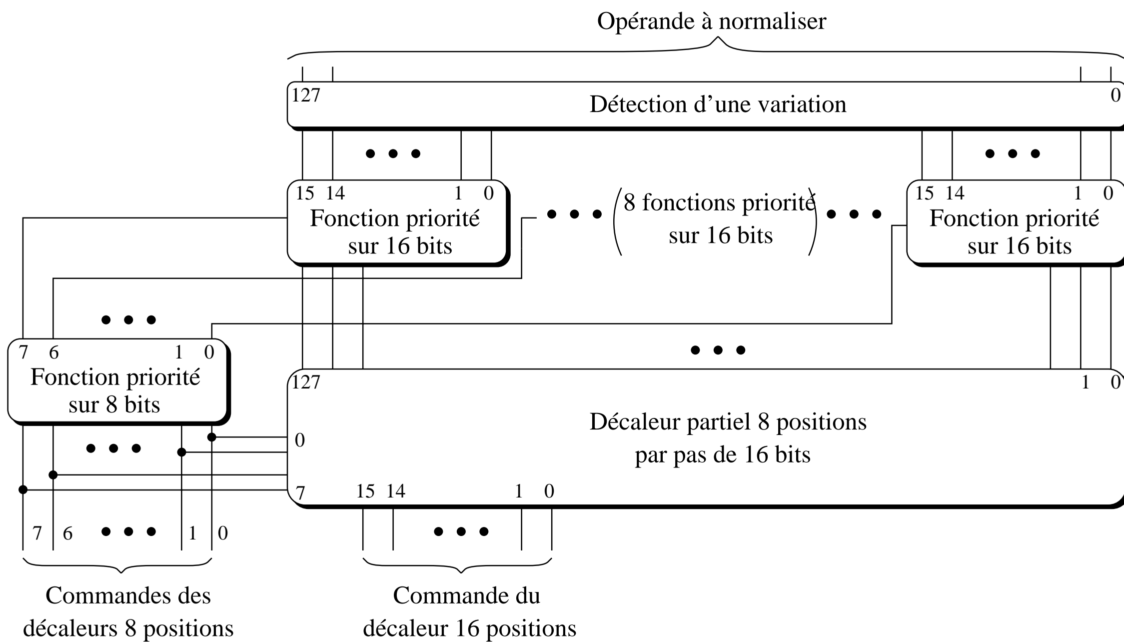


FIG. 5.30: Principe de la normalisation.

5.5.3 Synthèse et placement-routage

L'unité de décalage est constituée de quatre étages de pipeline. Le premier réalise la lecture des registres, le $2^{\text{ème}}$ et $3^{\text{ème}}$ étages effectuent les opérations de décalage et de normalisation et le dernier écrit les résultats dans les registres.

On obtient après synthèse un temps de 10,34 ns pour le $2^{\text{ème}}$ étage et de 16,54 ns pour le $3^{\text{ème}}$. Concernant ces deux étages, l'analyse de la synthèse a permis de voir

que la réalisation des décaleurs sur 128 bits était très coûteuse en portes et donc en surface. On a choisi de réaliser les décaleurs en *full-custom*. Les détails de leur architecture sont présentés dans les paragraphes suivants. Le résultat obtenu est alors de 0,30 mm^2 pour la surface du décaleur 8 positions contre 1,21 mm^2 lorqu'il est réalisé en cellules standards. Pour le décaleur 16 positions, la surface passe de 3,06 mm^2 en cellules standards à 0,46 mm^2 en *full-custom*.

La surface totale de l'unité est alors de 15,64 mm^2 contre 20,05 mm^2 pour la solution constituée uniquement de cellules standards.

5.5.4 Optimisations

La réalisation *full-custom* des décaleurs réduit leur surface par rapport à la solution en cellules standards. Pour ces décaleurs, on va implanter une architecture “à barillet” [81] qui est présentée figure 5.31. Celle-ci n'utilise que des transistors de même type (NMOS) ce qui permet d'avoir une grande densité d'intégration. Il faut cependant régénérer les signaux de sortie des décaleurs car les niveaux “1 logique” sont mal transmis.

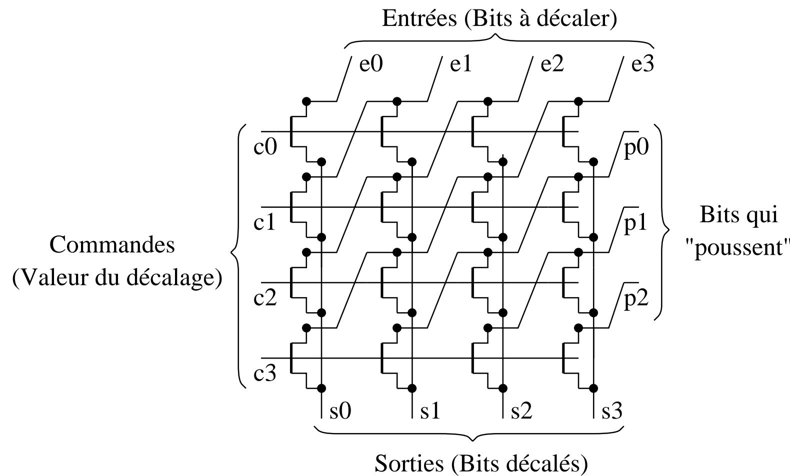


FIG. 5.31: Décaleur à barillet.

Pour augmenter encore la densité d'intégration, on utilise une cellule de base dans laquelle on fusionne 2 à 2 les sources des transistors appartenant à 2 rangées consécutives (Figure 5.32).

La réalisation *full-custom* du dessin des masques de cette cellule de base est présentée figure 5.33.

5.5.4.1 Décaleur 8 positions

Cet opérateur décale de 8 positions, par “paquets” de 16 bits, les 128 bits du bloc d'entrée. Les bits qui poussent proviennent d'un décaleur partiel, 8 fois plus petit que le décaleur complet, qui effectue la même opération mais qui ne récupère en sortie que les 16 bits de poids forts du 2$^{\text{ème}}$ bloc décalé. Ces deux opérateurs sont équivalents à 16 décaleurs élémentaires décalant de 8 positions par pas de 1 bit et qui acceptent chacun 9 bits d'entrée et 8 bits qui poussent (Figure 5.34).

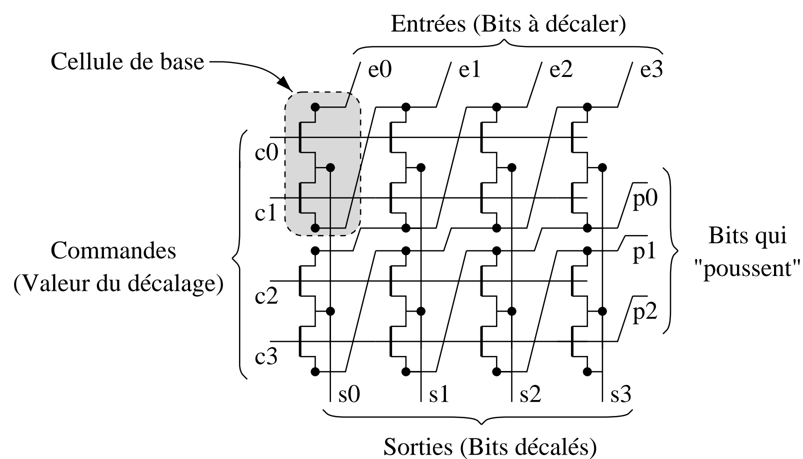


FIG. 5.32: Décaleur à barillet avec cellule de base à deux transistors.

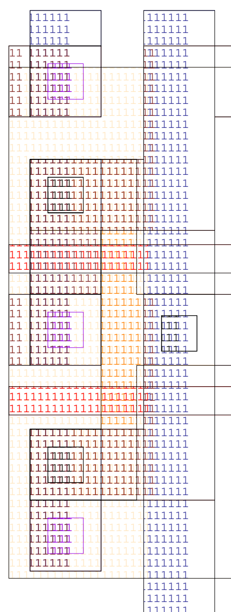


FIG. 5.33: Réalisation *full-custom* du dessin des masques de la cellule de base des décaleurs.

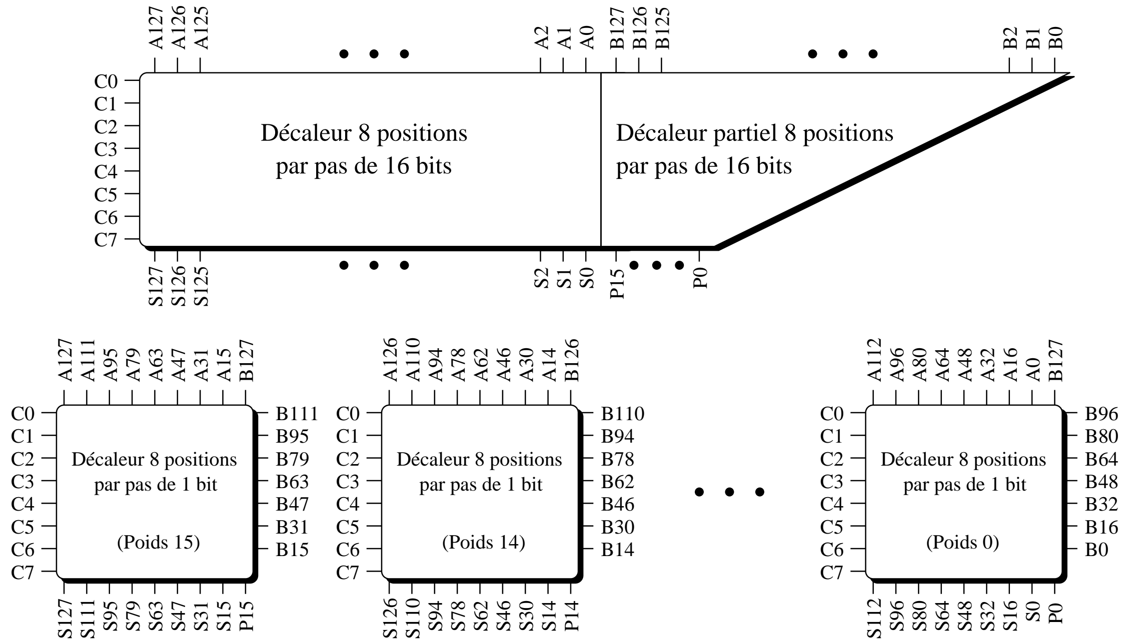


FIG. 5.34: Décaleurs 8 positions réalisés à partir de 16 décaleurs élémentaires.

Dans cette architecture, les lignes de commande sont très chargées (144 transistors) car elles sont communes à tous les décaleurs. On réalise alors une adaptation de charge en utilisant un arbre d'inverseurs pour distribuer les signaux de commande. Un inverseur commande 9 transistors ce qui permet d'insérer des inverseurs au début des lignes de commande de chaque décaleur élémentaire à 9 entrées. Le dimensionnement de ces inverseurs, ainsi que ceux de sortie qui servent à régénérer les signaux, est donné dans [56].

La réalisation *full-custom* de la cellule de base d'un décaleur élémentaire à 9 entrées est présentée figure 5.35. Cette cellule se compose de deux lignes de 9 transistors avec leurs sources fusionnées et des deux inverseurs de commande associés à chaque ligne.

Le décaleur 8 positions complet est alors réalisé à partir d'une matrice de 4 lignes et de 16 colonnes de la cellule de base présentée figure 5.35. Les dimensions du décaleur 8 positions complet sont de 1,68 mm$\times$0,18 mm soit 0,30 mm^2. La réalisation de ce décaleur en cellules standards occupe une surface de 1,21 mm^2. On a donc un gain en surface pour la solution *full-custom* de 75 %.

Les décaleurs à barillet étant des cellules qui comportent beaucoup de connexions, l'utilisation d'une technologie à trois niveaux de métal permettrait de réduire encore de 51 % la surface car les lignes de commande pourraient alors passer au dessus des cellules de base. On obtiendrait dans ce cas une surface de 0,15 mm^2 contre 0,30 mm^2 pour la réalisation actuelle avec une technologie à deux niveaux de métal .

5.5.4.2 Décaleur 16 positions

Cet opérateur effectue le décalage de 16 positions d'un bloc de 128 bits. Les 16 bits qui poussent proviennent de la sortie du décaleur partiel de 8 positions. La cellule de base de ce décaleur est celle donnée figure 5.32.

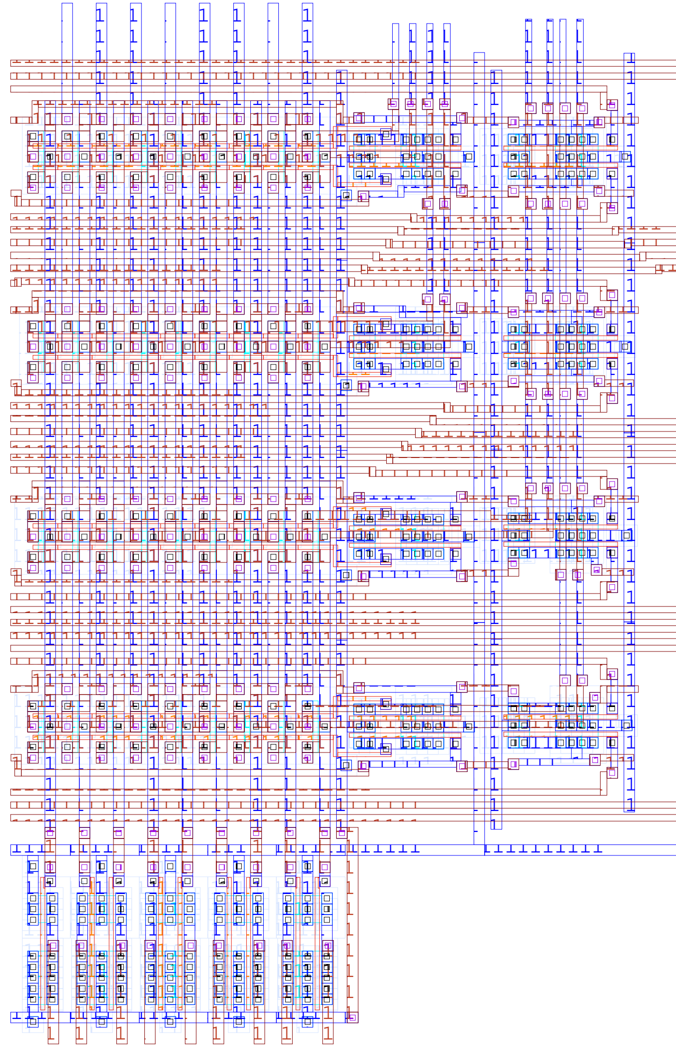


FIG. 5.35: *Dessin des masques en full-custom* de la cellule de base du décaleur 8 positions.

De même que pour le décaleur précédent, les lignes de commande sont très chargées (128 transistors). On utilise donc le même principe de distribution des signaux de commande. Pour ce décaleur, la cellule de base se compose de deux lignes de 8 transistors et de deux inverseurs de commande. La réalisation *full-custom* du dessin des masques de cette cellule de base est présentée figure 5.36.

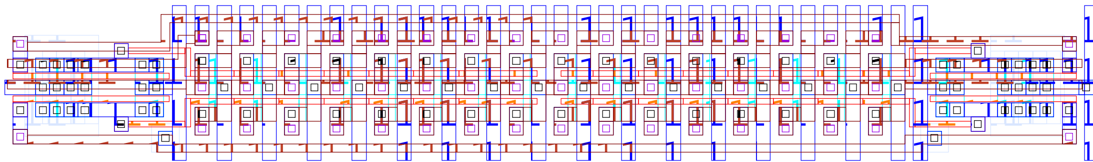


FIG. 5.36: Dessin des masques en *full-custom* de la cellule de base du décaleur 16 positions.

Le décaleur 16 positions par pas de 1 bit se compose de 8 lignes de 16 cellules de base. Les dimensions obtenues pour le décaleur complet sont 1,41 mm$\times$0,32 mm soit 0,46 mm^2. La surface occupée par ce décaleur en cellules standards est 3,06 mm^2. On a donc un gain en surface pour la solution *full-custom* de 85 %.

Avec une technologie à trois niveaux de métal, on pourrait obtenir une surface de 0,22 mm^2 soit une réduction supplémentaire de surface de 48 %.

5.6 Unité de chargement/rangement

5.6.1 Caractéristiques générales

Cette unité permet de réaliser les opérations de chargement et de rangement entre le processeur et la mémoire. Elle permet aussi d'effectuer les chargements d'immédiats ainsi que les transferts entre les registres de donnée, les registres de contrôle et la table des longueurs.

Cette unité est pipelinée sur 4 étages. Pour les transferts et les chargements d'immédiats, on obtient un débit d'une opération par cycle après une latence de 4 cycles. Pour les opérations avec la mémoire, le débit maximal qui peut être atteint est de 1 bloc transféré par cycle.

5.6.2 Architecture de l'unité

L'adressage de la mémoire est de type indirect par registre avec déplacement signé. La génération d'une adresse mémoire se fait sur trois cycles : le premier correspond à la lecture du registre contenant l'adresse de base codée sur 32 bits, le deuxième et le troisième à l'addition de cette adresse avec le déplacement signé codé sur 20 bits, avec extension du signe sur 32 bits.

Pour les opérations de rangement, qui écrivent des blocs dans la mémoire, l'unité prend en compte automatiquement les informations de débordement. En effet, lorsque l'unité doit écrire en mémoire un bloc qui comporte un débordement, elle écrit dans ce cas en mémoire une donnée qui ne contient que des bits égaux à la valeur du débordement Vd, car la valeur contenue dans le bloc est non significative.

Pour les opérations de chargement d'un immédiat, on doit effectuer l'écriture masquée d'un immédiat de 32 bits dans un registre de donnée de 128 bits. Or lorsque l'on fait une écriture dans un registre, celle-ci s'effectue sur l'ensemble des bits du registre. Pour les opérations de chargement d'un immédiat, il faut donc effectuer la lecture du registre destination lors du premier cycle de l'instruction, puis en utilisant des multiplexeurs, remplacer les 32 bits sélectionnés (selon l'instruction) par la valeur de l'immédiat. On réécrit ensuite ce bloc de 128 bits à l'adresse de destination.

Pour les opérations de transfert entre un registre de contrôle et un bloc, on effectue pour les mêmes raisons la lecture de tous les bits du registre et l'on écrit seulement ceux qui sont sélectionnés (selon l'instruction) dans le bloc destination.

Un schéma général de l'unité de chargement/rangement est donné figure 5.37.

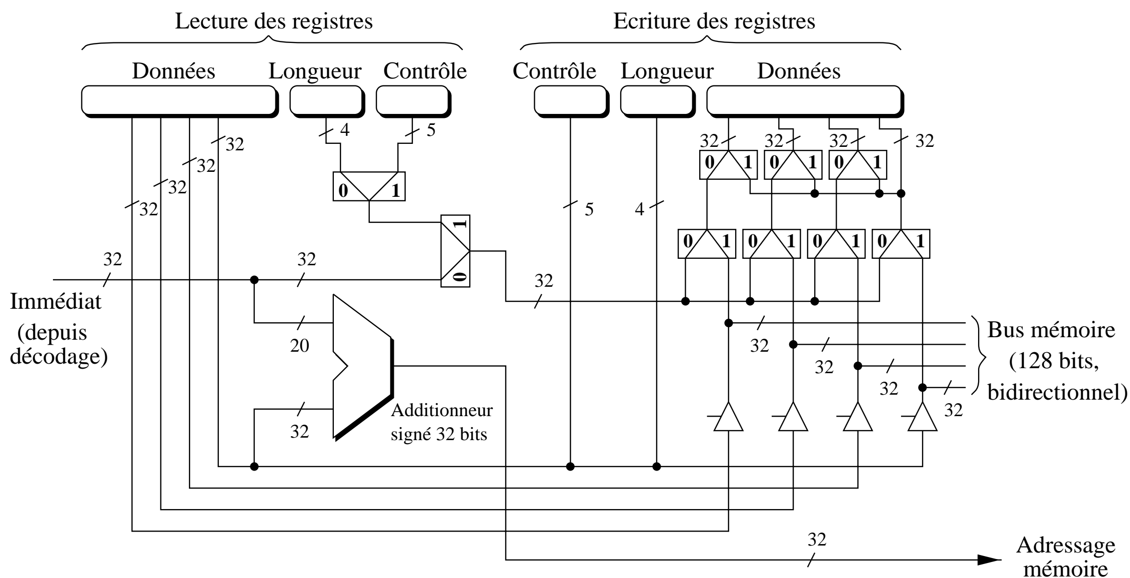


Fig. 5.37: Unité de chargement/rangement.

5.6.3 Synthèse et placement-routage

On a utilisé le flot de conception décrit pour l'unité d'addition (Paragraphe 5.2.3). L'unité de chargement/rangement est constituée de 4 étages de pipeline : le premier correspond à la lecture des registres, les deux suivants à la génération des adresses et des signaux de contrôle. Le dernier étage écrit les résultats dans le banc de registres. L'équilibrage des étages a permis d'obtenir les résultats suivants : 9,81 ns pour le 1er étage, 10,59 ns pour le $2^{ème}$ et 10,16 ns pour le $3^{ème}$. Les étages de cette unité ne sont pas critiques en temps et ne présentent pas de coûts élevés en surface. On réalise donc cette unité en cellules standards. On obtient ainsi une surface totale de 5,31 mm^2.

5.7 Conclusions

Les unités fonctionnelles qui ont été présentées dans ce chapitre ont une architecture pipelinée qui permet d'avoir un temps de cycle faible tout en maintenant

un débit de calcul élevé. Afin de diminuer la profondeur des pipelines, on réduit le temps de traversée des unités en utilisant pour les différents blocs des architectures en temps $O(\log_2 N)$ où N correspond au nombre de bits des entrées.

Chaque unité fonctionnelle possède un contrôleur d'unité qui permet aux instructions de s'exécuter de façon autonome. Le contrôleur effectue les accès aux bancs de registre, génère les adresses de lecture, d'écriture ainsi que les signaux de contrôle nécessaires au bon fonctionnement du pipeline. Les unités fonctionnelles comportent aussi des opérateurs arithmétiques qui utilisent en interne une notation redondante permettant de calculer en temps constant sans propagation de retenue. Cette notation procure un gain au niveau de la durée des opérations, d'autant plus important que la taille des données manipulées par le processeur est grande (128 bits).

Les opérateurs contenus dans les unités fonctionnelles effectuent directement les calculs en arithmétique exacte. Pour cela, les opérateurs doivent pouvoir traiter des opérandes d'entrée codées dans deux formats, non signé ou signé (complément à 2), selon qu'il s'agit du bloc de poids fort du nombre ou d'un bloc de poids inférieur. Dans le cas du multiplieur, on a développé une nouvelle architecture pour l'accumulateur afin de pouvoir intégrer cette particularité dans des temps corrects et pour un coût matériel raisonnable. Cette solution ne nécessite que quelques portes logiques supplémentaires par rapport à une solution ne traitant qu'un seul des deux formats. La solution proposée est générale et peut s'adapter à tous les multiplieurs qui utilisent une boucle avec accumulation pour effectuer l'opération en plusieurs cycles, et ceci quelle que soit la taille du multiplieur ou le nombre de boucles.

L'architecture des opérateurs a été optimisée pour le calcul sur les tableaux car les opérations utilisées en arithmétique exacte s'exécutent essentiellement avec pour opérandes des nombres de grande taille. Cette optimisation se traduit en particulier au niveau du transfert de paramètres d'un calcul à un autre lorsque l'on travaille sur des blocs consécutifs. L'utilisation d'opérateurs pipelinés impose en effet que ce transfert soit effectué en moins d'un cycle. Pour répondre à cette propriété, on a employé des architectures adaptées.

Pour l'opérateur d'addition, on a utilisé un additionneur redondant qui effectue l'opération en un temps inférieur au temps de cycle du circuit. Cela permet de transférer la retenue sortante moins d'un cycle après avoir reçu la retenue entrante. La retenue sortante ainsi transférée pourra alors être utilisée en tant que retenue entrante lors du calcul sur le prochain bloc.

Pour la multiplication, l'accumulateur qui a été développé utilise aussi une notation redondante ce qui permet d'additionner en moins d'un cycle le bloc de poids fort du résultat d'une multiplication avec le bloc de poids faible de la multiplication suivante.

L'opérateur de décalage a été réalisé à partir de deux décaleurs à barillet. Les décalages sont ainsi effectués rapidement puisque dans une structure à barillet, l'information que l'on veut décaler ne traverse qu'un seul transistor. Les bits qui sont décalés en dehors du bloc résultat peuvent alors être réinjectés au cycle suivant en tant que bits qui "poussent" dans le bloc suivant.

De même le module de reconversion de redondant en complément à 2 qui est présent dans toutes les unités doit respecter les mêmes règles. L'architecture de type Han et Carlson qui est utilisée possède un chemin critique entre les retenues entrante

et sortante qui est très faible et qui permet de propager la retenue entre des blocs consécutifs en moins d'un cycle.

On a aussi présenté dans ce chapitre des réalisations *full-custom* du dessin des masques qui permettent de réduire le chemin critique et/ou la surface des opérateurs. C'est le cas du multiplieur 128×32 bits dont le temps de calcul correspondait au chemin critique du circuit. Pour les décaleurs, leurs réalisations *full-custom* ont été faites dans le but de diminuer la surface car l'implantation de décaleurs à barillet peut apporter plus de 75% de réduction.

Chapitre 6

Optimisations de l'architecture

On présente dans ce chapitre plusieurs optimisations qui ont été appliquées au cœur du processeur afin de le rendre plus performante. Il s'agit en particulier d'optimiser la gestion des longueurs car cette fonction, lorsqu'elle est réalisée en logiciel, est coûteuse en temps. Elle pénalise l'enchaînement des opérations et diminue donc le remplissage des pipelines des unités fonctionnelles.

On présentera aussi une modification du mécanisme de détection des dépendances afin d'autoriser, lorsque certaines conditions sont vérifiées, le lancement d'instructions qui possèdent des dépendances sur des données de type tableau. Cette modification permet de commencer une nouvelle opération avant même que la précédente, avec laquelle il y a la dépendance de donnée sur un tableau, soit terminée.

6.1 Gestion dynamique de la longueur

On a vu dans le paragraphe 3.2.3 que les instructions sur les tableaux utilisaient un paramètre de longueur afin d'exécuter les opérations uniquement sur les blocs du tableau qui servent à coder le nombre. Chaque fois que le résultat d'une opération est un tableau, il faut donc mettre à jour la nouvelle longueur qui est associée à ce résultat de façon à pouvoir réutiliser le tableau dans une autre instruction. Pour les unités d'addition et de multiplication, cette mise à jour consiste à détecter quels sont les blocs de poids forts du résultat qui ne correspondent qu'à une extension du bit de signe afin de ne pas les prendre en compte dans la longueur du résultat. L'unité de décalage retournant en sortie des tableaux de même taille que ceux d'entrée, il n'est pas nécessaire dans ce cas de calculer la nouvelle longueur.

Dans les logiciels actuels, on fait appel à une fonction logicielle qui effectue cette gestion une fois le résultat obtenu ce qui augmente la durée réelle des opérations. De plus ce surcoût en temps peut être important car il faut de nouveau effectuer des accès aux registres de donnée pour relire les blocs de poids forts du résultat et ensuite calculer la nouvelle longueur.

Afin de supprimer le délai supplémentaire introduit par la fonction logicielle qui calcule la longueur, on propose une solution qui effectue dynamiquement la mise à jour de la longueur. Cette solution retourne la longueur exacte du nombre en même temps que le dernier bloc du résultat ce qui n'augmente pas la durée des opérations.

6.1.1 Principe

Pour éviter une relecture des blocs du résultat, on va effectuer un calcul de la longueur au fur et à mesure que ces blocs sont disponibles. Cependant ce calcul n'est pas le même selon que le résultat est un nombre signé ou non signé. Il faut donc calculer en parallèle deux longueurs, une qui correspond à un résultat signé, l'autre à un résultat non signé puisque l'information concernant le signe n'est connue qu'à la fin de l'opération.

Pour les nombres signés, une suite de blocs (en poids forts) n'intervient pas dans le calcul de la longueur lorsque chaque bloc vérifie les deux conditions suivantes :

- Tous les bits du bloc sont égaux au bit de signe du résultat.
- Le bit de poids fort du bloc précédent est égal au bit de signe du résultat.

Pour les nombres non signés, qui sont des nombres positifs, ce sont les suites de blocs (en poids forts) ne contenant que des zéros qui ne sont pas prises en compte dans le calcul de la longueur comme le montre l'exemple de la figure 6.1.

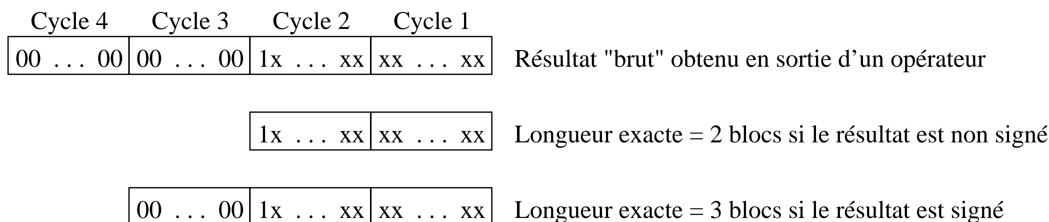


FIG. 6.1: Longueur d'un résultat selon que le nombre est signé ou non signé.

On doit donc détecter les blocs du résultat qui ne contiennent que des 0 (signal *d0*) et ceux qui ne contiennent que des 1 (signal *d1*) et ensuite effectuer le calcul des deux longueurs. Pour ne pas rallonger la durée des opérations, on va effectuer la détection à partir de l'écriture intermédiaire en notation redondante du résultat, en parallèle avec la reconversion en notation CC2. Le résultat en notation redondante diffère cependant du résultat exact en CC2 de la valeur de la retenue entrante (Cin) du soustracteur de type Han et Carlson utilisé pour la reconversion. Il faut donc faire la détection des résultats intermédiaires suivants :

- **Bloc du résultat qui ne contient que des 0 (signal *d0*) :**
  Si Cin = 0 alors détection des résultats intermédiaires 00...00 **(Type 1)**
  Si Cin = -1 alors détection des résultats intermédiaires 00...01 **(Type 2)**
- **Bloc du résultat qui ne contient que des 1 (signal *d1*) :**
  Si Cin = 0 alors détection des résultats intermédiaires 11...11 **(Type 3)**
  Si Cin = -1 alors détection des résultats intermédiaires 00...00 **(Type 1)**

La détection des trois types de résultats intermédiaires est présentée dans le paragraphe suivant.

6.1.2 Détection des 3 types de résultats

On rappelle ici les équations du soustracteur de type Han et Carlson présentées paragraphe 5.1.3 car la détection des différents résultats utilise des signaux déjà générés par le soustracteur.

$$S^{+}_{127,0} = \sum_{i=0}^{127} s^{+}_{i} 2^{i} \quad \text{et} \quad S^{-}_{127,0} = \sum_{i=0}^{127} s^{-}_{i} 2^{i}$$

– Génération d'une retenue à la position i

$$G_{i,i} = s^{+}_{i} \wedge \overline{s^{-}_{i}}$$

– Propagation d'une retenue à la position i

$$P_{i,i} = s^{+}_{i} \oplus \overline{s^{-}_{i}}$$

– Génération d'une retenue entre les positions i et k

$$G_{k,i} = G_{k,j} \vee P_{k,j} \wedge G_{j-1,i} \qquad (k \geq j > i)$$

– Propagation d'une retenue entre les positions i et k

$$P_{k,i} = P_{k,j} \wedge P_{j-1,i} \qquad (k \geq j > i)$$

Cette architecture étant en $O(\log_2 N)$, il faut donc aussi réaliser une détection des trois types de résultat en $O(\log_2 N)$ si on veut que la détection ne dure pas plus longtemps que la reconversion avec laquelle elle s'exécute en parallèle.

1. **Détection des résultats de type 1**

   La détection à partir de la notation redondante des résultats intermédiaires de type 1 est faite directement par le signal $P_{127,0}$ du soustracteur. On a en effet :

   $$P_{127,0} = (s^{+}_{127} \oplus \overline{s^{-}_{127}}) \wedge (s^{+}_{126} \oplus \overline{s^{-}_{126}}) \wedge ... \wedge (s^{+}_{1} \oplus \overline{s^{-}_{1}}) \wedge (s^{+}_{0} \oplus \overline{s^{-}_{0}})$$

2. **Détection des résultats de type 2**

   Les combinaisons des valeurs de $S^{+}_{127,0}$ et $S^{-}_{127,0}$ qui génèrent des résultats intermédiaires de type 2 sont dans le cas général :

   $$\begin{array}{rcccccccccc} S^{+}_{127,0} = & \substack{1\\0} & \substack{1\\0} & \substack{...\\...} & \substack{1\\0} & \substack{1\\0} & 1 & 0 & 0 & ... & 0 \quad 0 \\[2ex] S^{-}_{127,0} = & \multicolumn{5}{c}{\underbrace{\substack{1\\0} \;\; \substack{1\\0} \;\; \substack{...\\...} \;\; \substack{1\\0} \;\; \substack{1\\0}}_{\text{zone C } (s^{+}_{i} = s^{-}_{i})}} & \underbrace{0}_{\text{zone B}} & \multicolumn{4}{c}{\underbrace{1 \quad 1 \quad ... \quad 1 \quad 1}_{\text{zone A}}} \end{array}$$

   La zone A correspond à une zone où il n'y a ni génération ni propagation de retenue, la zone B correspond à une génération de retenue et la zone C à une zone de propagation de retenue. Il y a trois cas particuliers de combinaisons qui génèrent aussi des résultats intermédiaires de type 2 :

   – Pas de zone A

   $$\begin{array}{rcccccc} S^{+}_{127,0} = & \substack{1\\0} & \substack{1\\0} & \substack{...\\...} & \substack{1\\0} & \substack{1\\0} & 1 \\[2ex] S^{-}_{127,0} = & \multicolumn{5}{c}{\underbrace{\substack{1\\0} \;\; \substack{1\\0} \;\; \substack{...\\...} \;\; \substack{1\\0} \;\; \substack{1\\0}}_{\text{zone C } (s^{+}_{i} = s^{-}_{i})}} & \underbrace{0}_{\text{zone B}} \end{array}$$

– Pas de zone C

$$S^{+}_{127,0} = \quad 1 \quad 0 \quad 0 \; ... \; 0 \quad 0$$

$$S^{-}_{127,0} = \underbrace{0}_{\text{zone B}} \; \underbrace{1 \quad 1 \; ... \; 1 \quad 1}_{\text{zone A}}$$

– Pas de zones B et C

$$S^{+}_{127,0} = \quad 0 \quad 0 \; ... \; 0 \quad 0$$

$$S^{-}_{127,0} = \underbrace{1 \quad 1 \; ... \; 1 \quad 1}_{\text{zone A}}$$

On ne peut pas faire une détection de ces combinaisons à partir des bits s^{+}_{i} et s^{-}_{i} car la zone B a une position qui peut varier des indices 0 à 127, ce qui entraîne par conséquent des tailles variables pour les zones A et C.
On définit alors les trois zones à l'aide des signaux $P_{i,i} = s^{+}_{i} \oplus \overline{s^{-}_{i}}$ et $G^{'}_{i,i} = \overline{s^{+}_{i}} \wedge s^{-}_{i}$. On obtient pour chaque zone les résultats suivants :

$$\begin{array}{rcccc} & & \overbrace{\quad}^{\text{Zone C}} & \overbrace{\quad}^{\text{Zone B}} & \overbrace{\quad}^{\text{Zone A}} \\ P_{i,i} = s^{+}_{i} \oplus \overline{s^{-}_{i}} & = & 1\ 1\ ...\ 1\ 1 & 0 & 0\ 0\ ...\ 0\ 0 \\ G^{'}_{i,i} = \overline{s^{+}_{i}} \wedge s^{-}_{i} & = & 0\ 0\ ...\ 0\ 0 & 0 & 1\ 1\ ...\ 1\ 1 \end{array}$$

On remarque alors que quelque soit les combinaisons des zones A, B et C qui génèrent des résultats intermédiaires de type 2, les propriétés suivantes sont vérifiées :

– $P_{0,0} = 0$
– $P_{i,i} \neq G^{'}_{i-1,i-1}$ pour $i \in [1,\ 127]$ $\qquad (P_{i,i} = G^{'}_{i,i} = 1$ est impossible$)$

On peut, à partir de ces équations, réaliser une architecture matérielle en temps O($\log_2$N) qui détecte toutes les combinaisons des valeurs de $S^{+}_{127,0}$ et $S^{-}_{127,0}$ qui génèrent des résultats intermédiaires de type 2 (Figure 6.2).

3. **Détection des résultats de type 3**
Les combinaisons des valeurs de $S^{+}_{127,0}$ et $S^{-}_{127,0}$ qui génèrent des résultats intermédiaires de type 3 sont dans le cas général :

$$S^{+}_{127,0} = \begin{array}{c} 1\ 1\ ...\ 1\ 1 \\ 0\ 0\ ...\ 0\ 0 \end{array} \quad 0 \quad 1\ 1 ...\ 1\ 1$$

$$S^{-}_{127,0} = \underbrace{\begin{array}{c} 1\ 1\ ...\ 1\ 1 \\ 0\ 0\ ...\ 0\ 0 \end{array}}_{\substack{\text{zone C} \\ (s^{+}_{i} = s^{-}_{i})}} \quad \underbrace{1}_{\text{zone B}} \quad \underbrace{0\ 0\ ...\ 0\ 0}_{\text{zone A}}$$

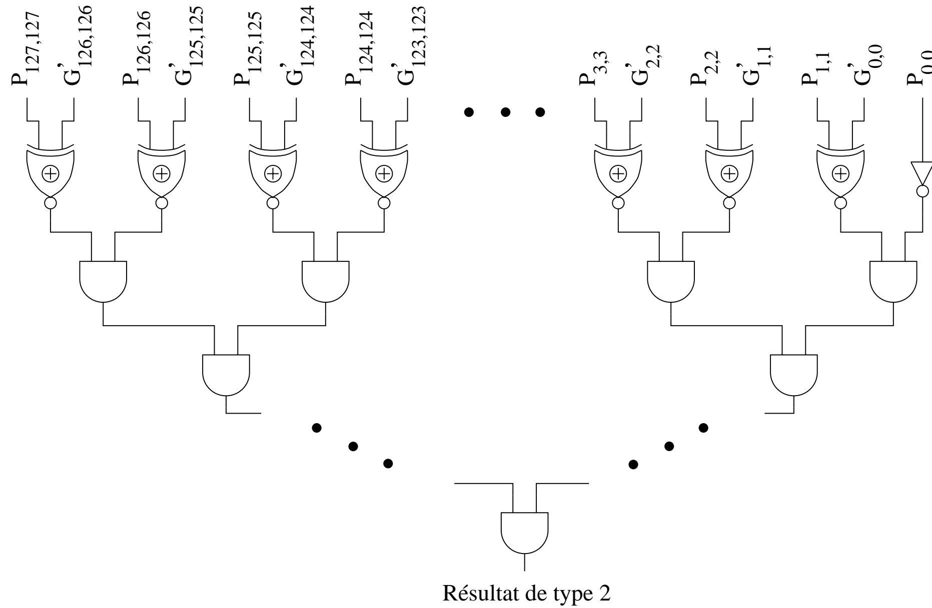


FIG. 6.2: Détection des résultats intermédiaires de type 2.

Il existe aussi trois cas particuliers qui correspondent aux mêmes combinaisons des zones A, B et C que pour la détection du type 2. On définit dans ce cas les trois zones à l'aide des signaux $P_{i,i} = s_i^+ \oplus \overline{s_i^-}$ et $G_{i,i} = s_i^+ \wedge \overline{s_i^-}$. On obtient pour chaque zone les résultats suivants :

$$\begin{array}{rcccc} & & \text{Zone C} & \text{Zone B} & \text{Zone A} \\ P_{i,i} = s_i^+ \oplus \overline{s_i^-} & = & \overbrace{1\ 1 \ldots 1\ 1} & \overbrace{0} & \overbrace{0\ 0 \ldots 0\ 0} \\ G_{i,i} = s_i^+ \wedge \overline{s_i^-} & = & 0\ 0 \ldots 0\ 0 & 0 & 1\ 1 \ldots 1\ 1 \end{array}$$

Les combinaisons des zones A, B et C qui génèrent des résultats intermédiaires de type 3 vérifient des propriétés similaires à celles du type 2, à savoir :

- $P_{0,0} = 0$
- $P_{i,i} \neq G_{i-1,i-1}$ pour $i \in [1,\ 127]$ $\quad (P_{i,i} = G_{i,i} = 1$ est impossible)

On peut donc utiliser une architecture identique à celle présentée figure 6.2 mais avec les signaux d'entrée $P_{i,i}$ et $G_{i,i}$.

A partir de la définition des trois types, faite paragraphe 6.1.1, on remarque que l'on ne détectera jamais simultanément les types 2 et 3. On peut donc utiliser le même matériel pour faire la détection de ces deux types. Il suffit de sélectionner, en fonction de la retenue entrante (Cin), le signal d'entrée $G_{i,i}$ ou $G'_{i,i}$, l'autre étant toujours $P_{i,i}$.

La figure 6.3 présente le schéma complet utilisé pour la réalisation des signaux *d0* et *d1* qui indiquent lorsqu'un bloc du résultat final ne contient respectivement que des 0 et que des 1.

6.1.3 Calculs des longueurs

Chaque unité possède son propre gestionnaire de longueur. Celui-ci calcule à chaque nouveau bloc du résultat, en fonction des signaux *d0* et *d1* définis précé-

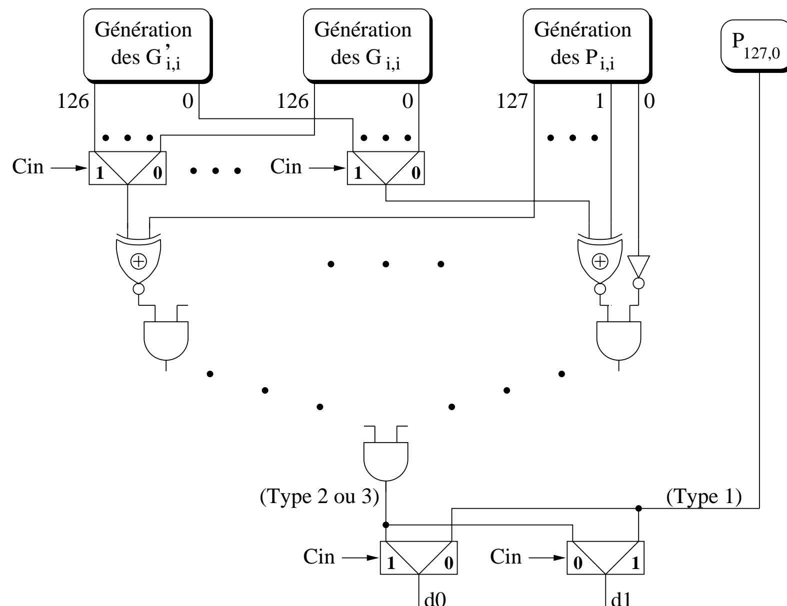


FIG. 6.3: Schéma pour la réalisation des signaux *d0* et *d1*.

demment, les longueurs dans le cas d'un résultat final signé ou non signé. Lorsque l'opération est terminée, et que le type du résultat est connu, le gestionnaire sélectionne la bonne longueur parmi les deux calculées. La longueur retournée correspond alors à la longueur exacte du résultat.

Pour les algorithmes présentés dans la suite de ce paragraphe, il est important de noter que la longueur est codée sur 4 bits et qu'elle varie entre 0 et 15 pour des tableaux qui ont respectivement entre 1 et 16 blocs. Cette information de longueur correspond à l'indice du bloc de poids fort.

– **Unité d'addition/soustraction**

  Le calcul de la longueur pour les opérations d'addition ou de soustraction doit tenir compte des débordements éventuels. Dans ce cas, le bloc supplémentaire qui n'est pas écrit immédiatement doit être pris en compte dans la longueur.

  On notera :

  Lg_s la longueur intermédiaire pour un résultat signé.

  Lg_{ns} la longueur intermédiaire pour un résultat non signé.

  Lg la longueur finale.

  D le bit qui indique un débordement.

  NS/S le bit qui indique le type (signé ou non signé) d'un bloc.

  Algorithme :

  $Lg_s \Leftarrow 0$;

  $Lg_{ns} \Leftarrow 0$;

  $compteur \Leftarrow 0$;

  $b_{127} \Leftarrow 0$;

```
Répéter
  Si (D=1 et compteur ≠15)
        alors compteur ⇐ compteur+1 ; {débordement}
  Si (d0=1 et D=0)
        alors Lg_ns ⇐ Lg_ns ; {pas d'augmentation}
        sinon Lg_ns ⇐ compteur ;
  Si (d0=1 et b_127=0) ou (d1=1 et b_127=1)
        alors Lg_s ⇐ Lg_s ; {pas d'augmentation}
        sinon Lg_s ⇐ compteur ;
  Si NS/S=0
        alors Lg ⇐ Lg_ns ; {n'est valide que lors
        sinon Lg ⇐ Lg_s ;     du dernier coup}
  b_127 ⇐bit de poids fort du bloc obtenu ;
  compteur ⇐ compteur+1 ;
Jusqu'au dernier bloc du résultat (Dcoup=1) ;
```

Lorsque le résultat possède un débordement, il faut prendre une longueur incrémentée de 1 sauf lorsque l'on est au 16ème bloc du résultat. Dans ce cas, il faut écrire Lg=15 pour le 16ème bloc et Lg=0 pour le bloc supplémentaire qui contiendra le débordement.

La réalisation matérielle du calcul de la longueur pour l'unité d'addition est présentée figure 6.4.

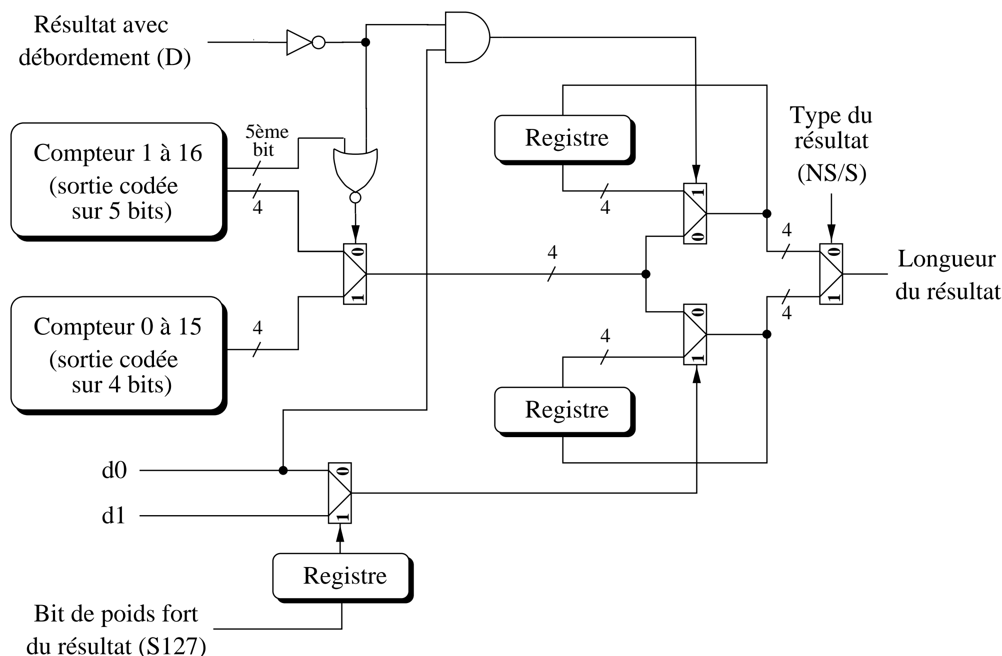


FIG. 6.4: Calcul de la longueur pour l'unité d'addition.

– **Unité de multiplication**

On notera :

Lg_s la longueur intermédiaire pour un résultat signé.
Lg_{ns} la longueur intermédiaire pour un résultat non signé.
Lg la longueur finale.
D le bit qui indique un débordement.
NS/S le bit qui indique le type (signé ou non signé) d'un bloc.

Algorithme :

```
Lg_s ⇐0 ;
Lg_ns ⇐0 ;
compteur ⇐0 ;
b_127 ⇐0 ;
Répéter
  Si d0=1
        alors Lg_ns ⇐ Lg_ns ; {pas d'augmentation}
        sinon Lg_ns ⇐ compteur ;
  Si (d0=1 et b_127=0) ou (d1=1 et b_127=1)
        alors Lg_s ⇐ Lg_s ; {pas d'augmentation}
        sinon Lg_s ⇐ compteur ;
  Si NS/S=0
        alors Lg ⇐ Lg_ns ; {n'est valide que lors
        sinon Lg ⇐ Lg_s ;    du dernier coup}
  b_127 ⇐bit de poids fort du bloc obtenu ;
  compteur ⇐ compteur+1 ;
Jusqu'au dernier bloc (bloc supplémentaire) du résultat (Scoup=1) ;
```

Lorsque le résultat tient sur 17 blocs, il faut écrire Lg=15 lors du 16ème bloc et ensuite Lg=0 pour le 17ème bloc qui correspond au premier élément d'un nouveau tableau.

La réalisation matérielle du calcul de la longueur pour l'unité de multiplication est présentée figure 6.5.

Il faut cependant ajouter une modification pour l'écriture des signaux NS/S. En effet chaque signal NS/S est écrit dans les registres en même temps que le bloc auquel il est associé. Pour un nombre signé, on a NS/S=0 pour tous ses blocs jusqu'à l'avant dernier et NS/S=1 pour le dernier bloc.

Lorsque le calcul de la longueur exacte génère une réduction de la longueur du nombre obtenue à la fin de l'opération (longueur maximum), il faut alors écrire le signal NS/S=1 à l'adresse du dernier bloc de la nouvelle longueur (longueur réduite) car sinon on va obtenir un résultat non signé. On verra dans le paragraphe 6.2 qu'il est nécessaire que le signal NS/S du dernier bloc soit écrit dans tous les blocs compris dans l'intervalle entre la longueur réduite et la longueur maximum. Pour cela, on code ces deux longueurs comme étant des suites de 1 comprises entre 1 et 16 bits. On effectue ensuite une comparaison bit à bit de ces deux longueurs codées, ce qui permet d'obtenir toutes les positions des blocs compris dans l'intervalle entre les deux longueurs. C'est ce signal qui est utilisé pour sélectionner, en parallèle, les blocs dans lesquels on doit écrire le signal NS/S (Figure 6.6).

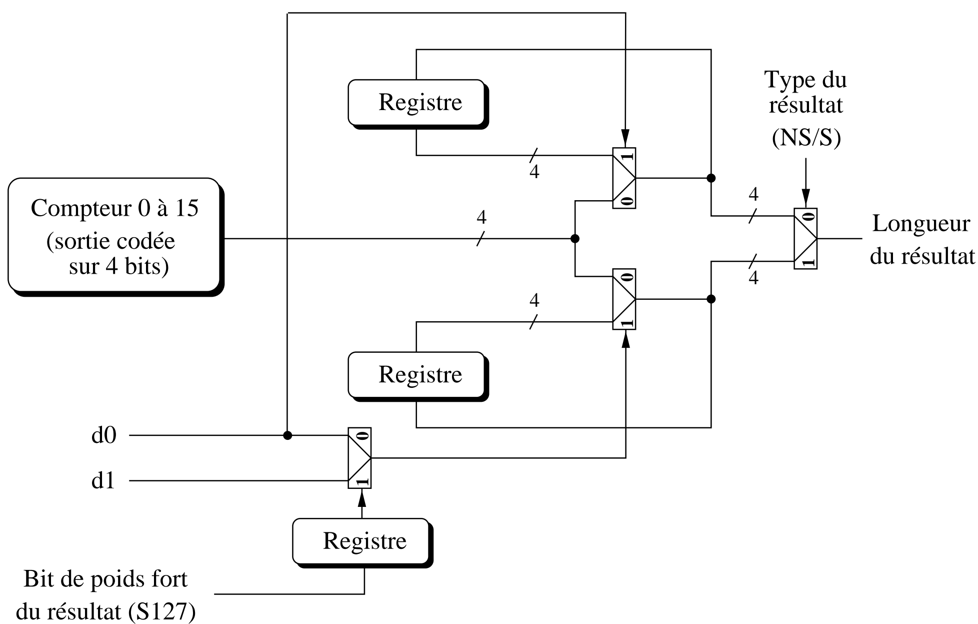


FIG. 6.5: Calcul de la longueur pour l'unité de multiplication.

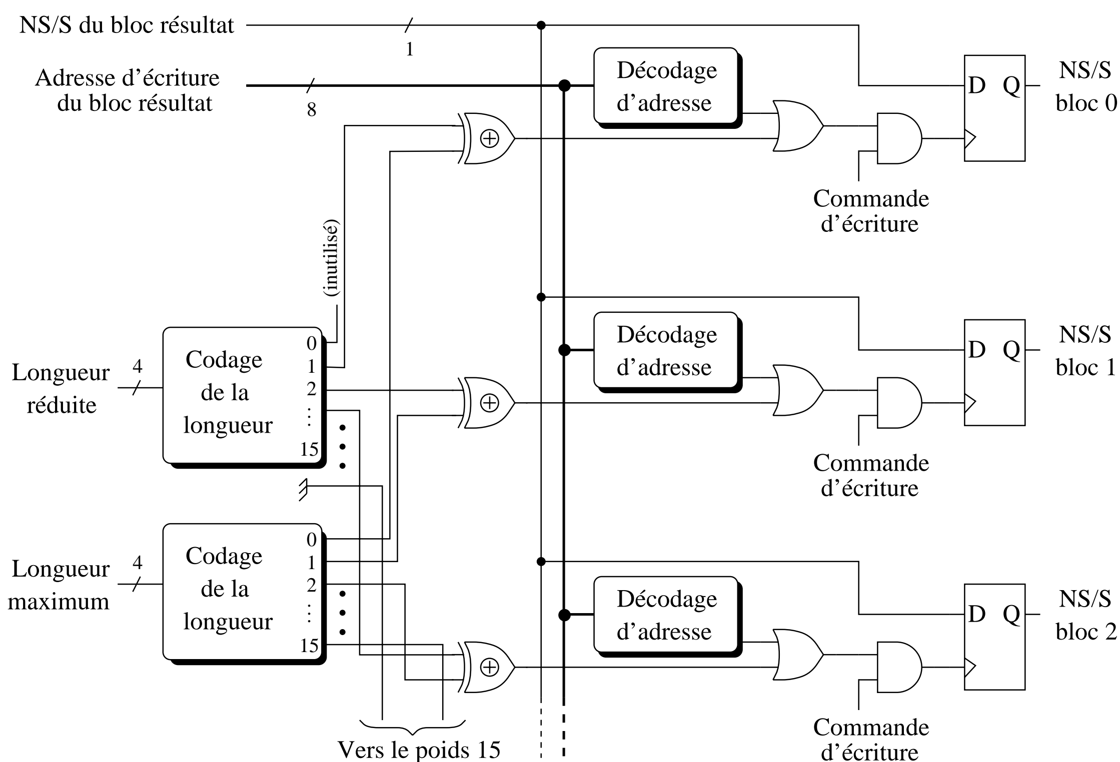


FIG. 6.6: Mise à jour des signaux NS/S lors du calcul des longueurs.

6.2 Recouvrement des opérations

Lorsqu'une opération est exécutée, ses opérandes (d'entrée et de sortie) deviennent invalides et ne peuvent pas être utilisées par les prochaines opérations. Ces dernières sont alors suspendues jusqu'à ce que les opérandes soient de nouveau valides. Seul le cas d'une lecture suivie d'une lecture, qui ne génère pas de dépendances, est autorisé lors du décodage.

Cependant, dans le cas d'opérations sur les tableaux, cet algorithme est trop restrictif et peut suspendre des instructions alors que la dépendance, détectée au niveau tableau, n'est en réalité pas bloquante au niveau des blocs manipulés séquentiellement par les opérateurs.

On présente dans la suite de cette section une solution pour modifier efficacement l'algorithme de détection des dépendances afin d'autoriser le lancement, sous certaines conditions, d'opérations possédant des dépendances sur des opérandes au format tableau.

6.2.1 Principe

Le recouvrement d'opération signifie qu'une opération va être lancée bien que la précédente (avec laquelle elle possède une dépendance sur une opérande de type tableau) n'ait pas fini de s'exécuter. C'est le cas par exemple lorsqu'une opération retourne un résultat de type tableau qui doit être lu en tant qu'opérande d'entrée par une prochaine opération. La seconde opération doit attendre que la première soit totalement finie avant de pouvoir lire les blocs du résultat. Le recouvrement de deux opérations permet de lancer la seconde opération avant que le résultat de la première ne soit terminé (Figure 6.7). En effet, les premiers blocs du résultat retournés par la première opération sont disponibles depuis plusieurs cycles et peuvent, sous certaines conditions, être utilisés par la seconde opération avant la fin de la première. Ce recouvrement permet d'obtenir un gain de cycles qui sera d'autant plus important que la taille du tableau réutilisé sera grande.

Pour que le recouvrement puisse être réalisé, il faut que les deux opérations vérifient les conditions suivantes :

- Les opérandes sont des tableaux.
- L'ordre de lecture des blocs (selon les adresses croissantes ou décroissantes) est le même pour les deux opérations.
- La fréquence des lectures ou des écritures de la deuxième opération n'est pas supérieure à celle des lectures ou des écritures de la première opération.

Le tableau 6.1 présente les différents recouvrements possibles entre deux opérations.

On peut remarquer que les opérations de chargement et de rangement (accès mémoire) ne permettent pas les recouvrements car ces deux opérations génèrent des ruptures de séquence lors du transfert de blocs entre le processeur et la mémoire. Les opérations sur les tableaux n'autorisant pas les "interruptions", il n'est pas possible d'effectuer des recouvrements car cela crée des conflits si une des opérations est interrompue et pas l'autre.

Ce recouvrement d'opérations est similaire dans son principe au chaînage vectoriel [52][12] mais il est ici généralisé à des tableaux de taille variable et parfois

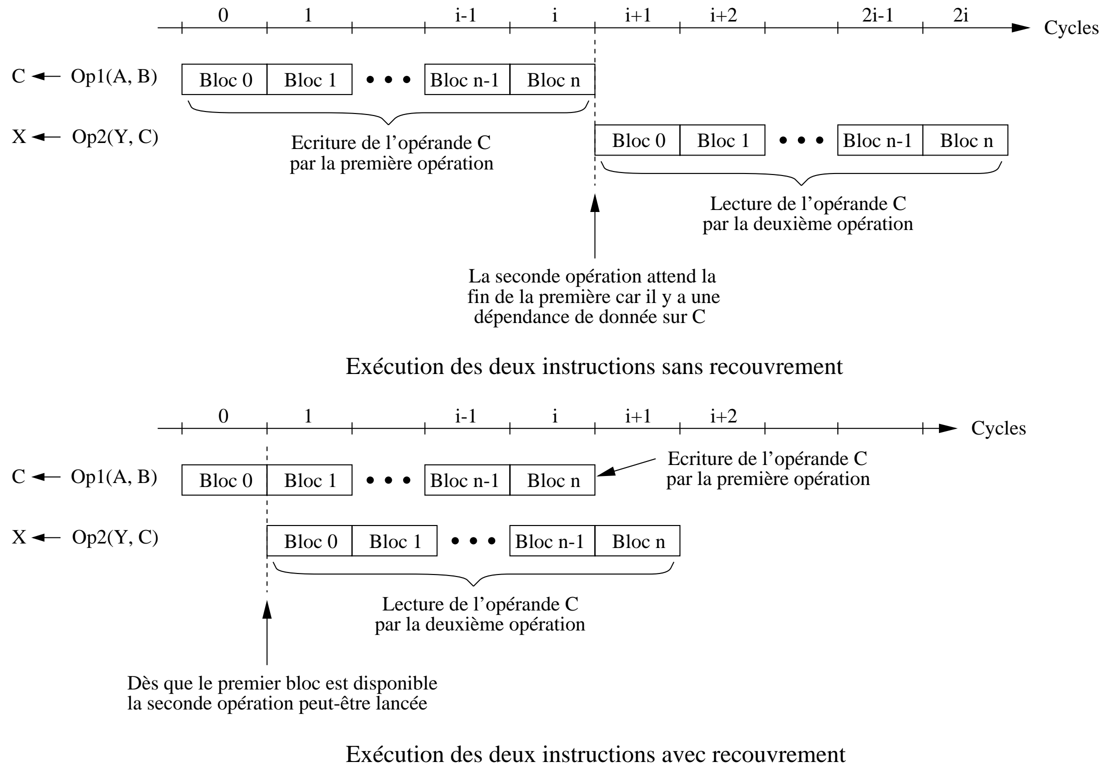


FIG. 6.7: Principe du recouvrement de deux opérations.

$2^{\text{ème}}$ opération → ↓ $1^{\text{ère}}$ opération	Add.	Sous.	Mult.	Mult-accu.	Dec G, Dec D, Norm.	Dec arith.	Tran.	Acc. mém.
Addition	Pos	Pos	Pos	Pos	Imp	Pos	Pos	Imp
Soustraction	Pos	Pos	Pos	Pos	Imp	Pos	Pos	Imp
Multiplication	Imp	Imp	Pos	Pos	Imp	Imp	Imp	Imp
Multiplication-accumulation	Imp	Imp	Pos	Pos	Imp	Imp	Imp	Imp
Décalages G, D, Normalisation	Imp	Imp	Imp	Imp	Imp	Imp	Imp	Imp
Décalage arithmétique	Pos	Pos	Pos	Pos	Imp	Pos	Pos	Imp
Transfert de blocs	Pos	Pos	Pos	Pos	Imp	Pos	Pos	Imp
Accès mémoire	Imp	Imp	Imp	Imp	Imp	Imp	Imp	Imp

Pos : recouvrement possible Imp : recouvrement impossible

TAB. 6.1: Possibilités de recouvrement entre deux opérations.

inconnue au moment de lancer l'opération comme nous le verrons dans le paragraphe 6.2.3.

6.2.2 Détection du recouvrement

Un recouvrement correspond au lancement d'une opération malgré une dépendance sur un tableau avec une opération en cours. Pour autoriser le recouvrement, il faut donc modifier la détection des dépendances sur les tableaux pour que celle qui existe entre les deux opérations ne soit plus bloquante.

Les conditions, énoncées paragraphe 6.2.1, qui autorisent les recouvrements dépendent uniquement de propriétés spécifiques aux opérations. On caractérise alors chaque opération par deux bits :

- Le bit I/P indique si l'opération permet ou non les recouvrements. Ce bit inclut dans les opérations qui n'autorisent pas les recouvrements ($I/P = 0$) aussi bien les opérations qui n'ont pas un ordre de lecture des blocs correct que celles comme les chargements ou les rangements qui créent des conflits.
- Le bit R/L indique la fréquence des lectures ou des écritures de l'opération. Il y a deux types d'opérations : celles qui lisent et écrivent tous les cycles ($R/L = 1$) et celles qui lisent et écrivent tous les quatre cycles ($R/L = 0$).

Le tableau 6.2 donne les caractéristiques de chaque opérations.

Opérations	I/P	R/L
Addition	1	0
Soustraction	1	0
Multiplication	1	1
Multiplication-accumulation	1	1
Décalages à gauche, à droite	0	0
Décalage arithmétique	1	0
Normalisation	0	0
Transfert de blocs	1	0
Chargement	0	0
Rangement	0	0

TAB. 6.2: Caractéristiques des opérations à partir des signaux I/P et R/L.

Le bit *bloc/tab*, qui est déjà associé à chaque opérande lors du décodage, permet de vérifier la troisième condition concernant le type des opérandes.

La détection d'un recouvrement entre une opération en cours ($bloc/tab_{EC}$, I/P_{EC}, R/L_{EC}) et une nouvelle opération ($bloc/tab_N$, I/P_N, R/L_N) est obtenue par la formule :

$$Recouvrement = bloc/tab_{EC} \wedge bloc/tab_N \wedge I/P_{EC} \wedge I/P_N \wedge (R/L_{EC} \vee \overline{R/L_N})$$

Pour que la seconde opération puisse être lancée lors d'un recouvrement, il faut modifier la détection des dépendances pour que celle qui est liée au tableau utilisé par les deux opérations ne soit plus bloquante. Lorsqu'un recouvrement est possible

entre deux opérations, la recherche d'une dépendance qui était au niveau tableau est alors ramenée à une recherche d'une dépendance entre le bloc actuel de la première opération et le premier bloc de la seconde opération. En effet, comme on peut le voir sur la figure 6.7, il suffit que le premier bloc du tableau lié à la première opération soit "disponible" (lecture ou écriture de ce bloc effectuée) pour que la seconde opération puisse commencer son exécution.

Le module de détection des dépendances (Figure 4.11) est modifié de façon à pouvoir détecter, lorsqu'un recouvrement est possible, la nouvelle dépendance entre les blocs. Le nouveau module obtenu s'applique à toutes les détections pour lesquelles les deux opérandes comparées peuvent être de type tableau. Le schéma général de ce nouveau module est présenté figure 6.8.

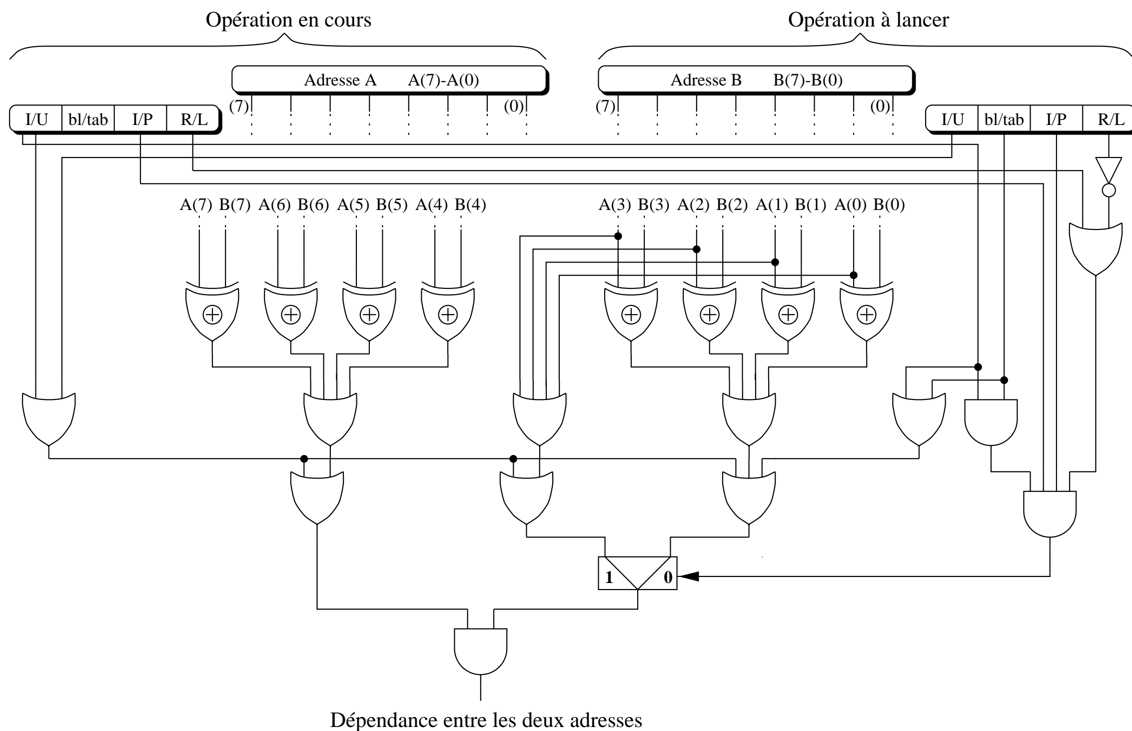


FIG. 6.8: Schéma général du module de détection des dépendances.

6.2.3 Estimation de la longueur

Lorsque l'on exécute une opération sur un tableau, il faut que le décodage transmette à l'unité fonctionnelle l'adresse et la longueur de ce tableau. Dans le cas d'un recouvrement, il faut que le décodage de la deuxième opération transmette la longueur du tableau qui est le résultat de la première opération. Or celle-ci n'est pas terminée et la longueur du résultat pas encore mis à jour. Il faut donc transmettre à la seconde opération une estimation de la longueur du tableau en attendant que la première opération retourne la valeur exacte de cette longueur.

Les unités fonctionnelles écrivent donc au début de chaque nouvelle opération la longueur estimée du résultat dans la table des longueurs. Lorsque l'on atteint le dernier bloc de la première opération, le gestionnaire de longueur écrit dans cette

table la longueur exacte du résultat. Cela permet de transmettre à la deuxième opération une longueur qui ne perturbe pas son fonctionnement en attendant la longueur exacte retournée par la première opération.

La longueur estimée d'un résultat (Lg_{est}) est calculée à partir du type de l'opération et de la longueur de ses opérandes d'entrée (Lg_A, Lg_B).

- Pour une addition ou une soustraction, on prend :

$$Lg_{est} = max\,(\,Lg_A\,,\,Lg_B\,)$$

- Pour une multiplication ou une multiplication-accumulation, on prend :

$$Lg_{est} = Lg_A + 1$$

- Pour un décalage, on prend :

$$Lg_{est} = Lg_A$$

Pour que la deuxième opération s'effectue correctement, il faut qu'elle lise la longueur exacte avant d'avoir atteint les derniers blocs de la longueur estimée, car sinon le signal *Dcoup* indiquant la fin de l'opération serait généré ce qui entraînerait un résultat faux. On présente dans la suite de ce paragraphe la synchronisation des signaux lors d'un recouvrement dans le cas d'une dépendance "lecture après écriture" :

$$\begin{array}{l} tableau\ C \leftarrow op1\,(\,tableau\ A\,,\ tableau\ B\,) \\ tableau\ E \leftarrow op2\,(\,tableau\ C\,,\ tableau\ D\,) \end{array}$$

C'est ce type de dépendance qui génère le plus de contraintes pour la réalisation d'un recouvrement.

Il y a trois cas possibles selon la valeur de la longueur exacte que l'on obtient :

1. La longueur exacte est supérieure à la longueur estimée.

   Si on n'a pas atteint la dernière valeur de Lg_{est}, alors la longueur exacte est lue et l'exécution se prolonge jusqu'à ce que l'on atteigne la dernière valeur de la longueur exacte. Le nombre de blocs restant à lire (ou à écrire) est obtenu par comparaison entre la valeur de la longueur et un compteur de telle façon que lorsque la longueur change, le nombre de blocs soit automatiquement corrigé comme le montre la figure 6.9.

2. La longueur exacte est égale à la longueur estimée.

   Dans ce cas, la longueur lue par l'unité est toujours la même ce qui ne génère aucune contrainte.

3. La longueur exacte est inférieure à la longueur estimée.

   Tous les blocs du résultat retournés par la première opération sont corrects même si ils ne sont pas tous utiles pour le codage de ce résultat. La deuxième opération peut donc continuer de s'exécuter sur des blocs qui ne feront plus partie du résultat lorsque la mise à jour de la première longueur sera effectuée. Les résultats retournés par la seconde opération restent cependant toujours corrects. Lorsque la longueur exacte du premier résultat est connue, si elle est inférieure à la longueur estimée, alors la deuxième opération se termine au

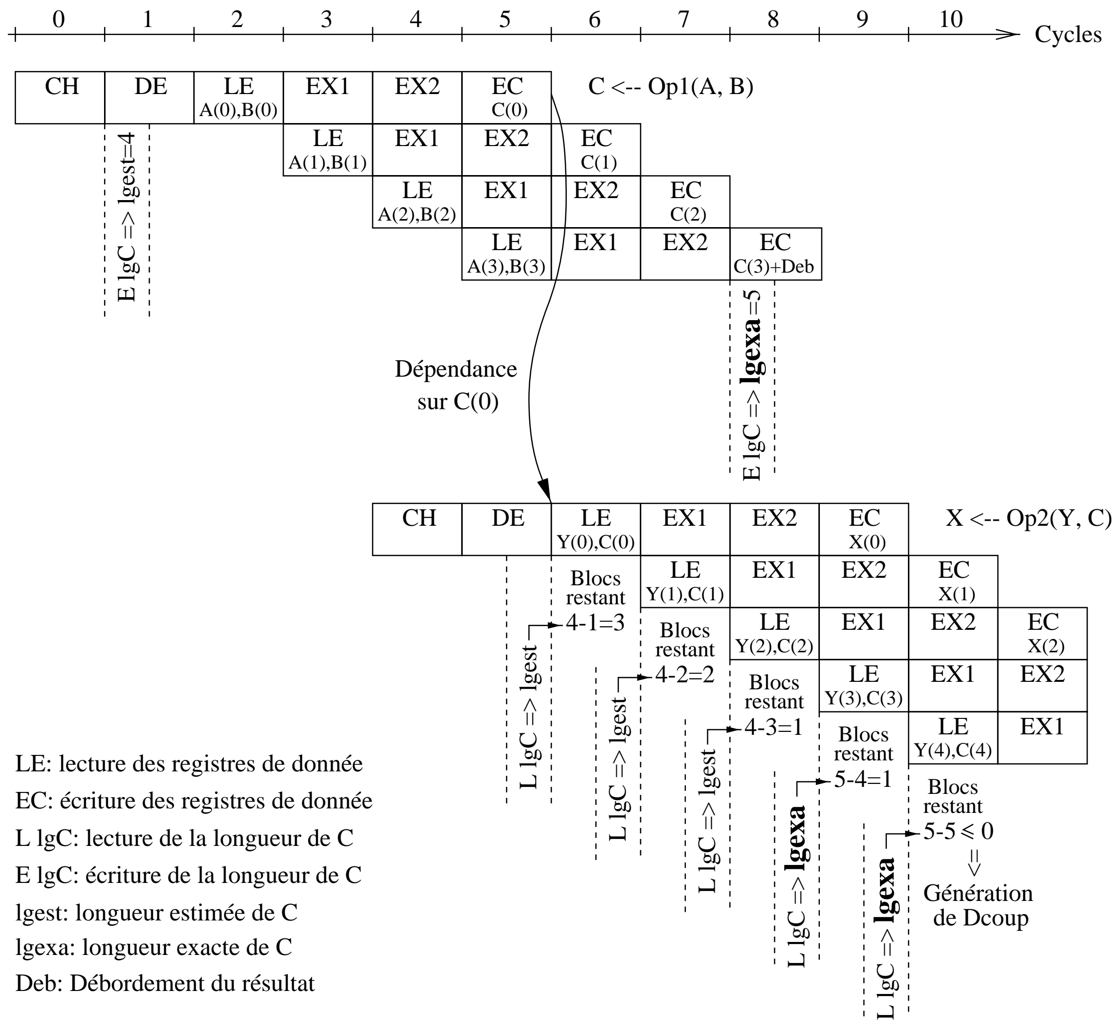


FIG. 6.9: Longueur exacte supérieure à la longueur estimée.

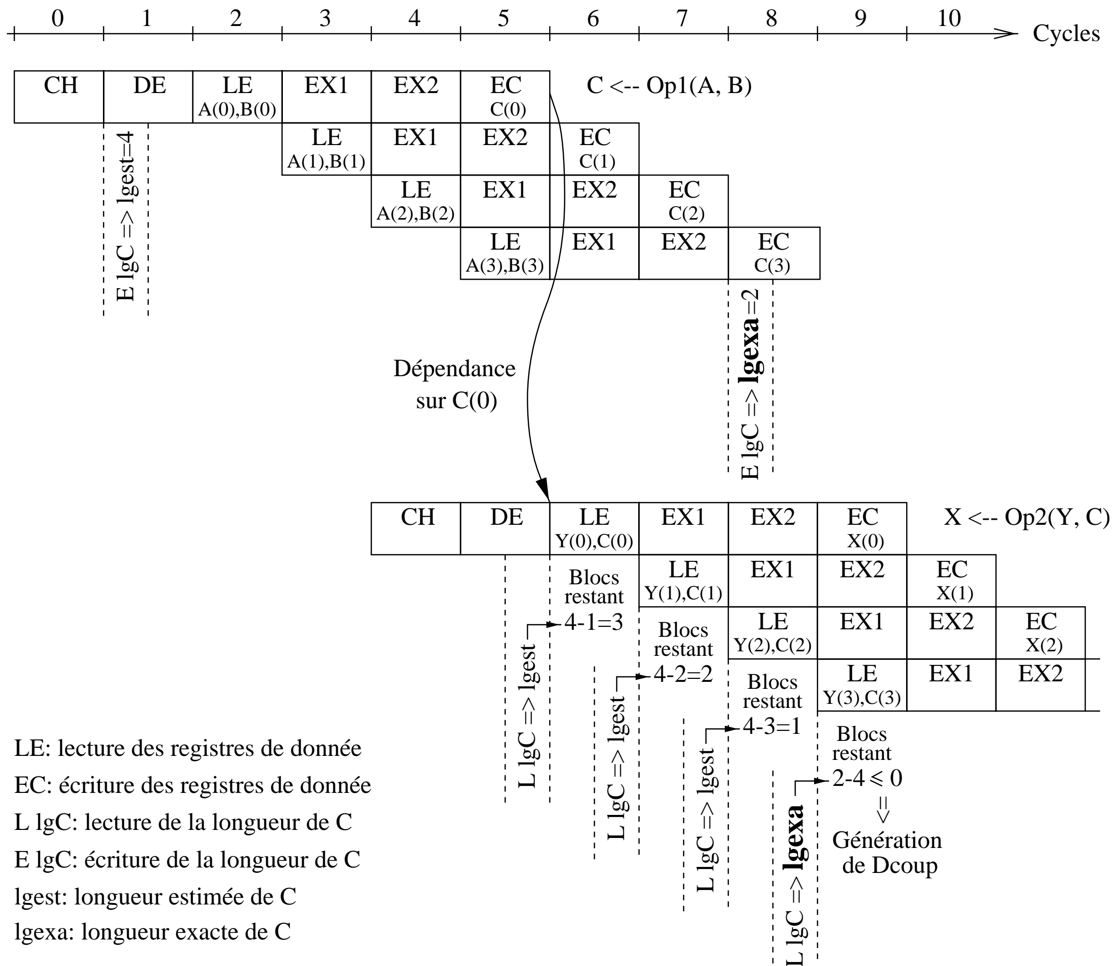


FIG. 6.10: Longueur exacte inférieure à la longueur estimée.

cycle suivant comme le montre la figure 6.10. C'est à cause de ce cas que l'on doit écrire la valeur du bit NS/S du dernier bloc du premier résultat dans tous les blocs compris entre la longueur exacte et la longueur estimée (voir paragraphe 6.1.3). Le dernier bloc lu par la deuxième opération doit représenter le dernier bloc du résultat et donc posséder en particulier la bonne valeur NS/S. Or la position de ce dernier bloc lu est quelconque (dans l'intervalle longueur exacte - longeur estimée) puisque l'on ne sait pas exactement où en est l'avancement de la deuxième opération. Afin d'être sûr que la deuxième opération lira la bonne valeur du bit NS/S, celle-ci est écrite simultanément dans tous les blocs situés après la position correspondant à la longueur exacte.

6.3 Conclusions

On a présenté dans ce chapitre deux optimisations importantes lorsque l'on veut faire du calcul exact car elles conduisent, non seulement à une réduction de la durée des opérations, mais surtout à un meilleur enchaînement de celles-ci.

La première optimisation concerne le calcul de la longueur lorsqu'une opération retourne comme résultat un tableau. Cette solution détecte les blocs ne contenant qu'une extension du bit de signe à partir de l'écriture en notation redondante du résultat. La détection à partir de cette écriture permet de réaliser le calcul de la longueur en parallèle avec la reconversion en complément à 2. Pour cela on a développé une solution matérielle en temps $O(\log_2 N)$ qui permet de détecter toutes les combinaisons de la notation redondante qui conduisent à des blocs en complément à 2 s'écrivant 00...00, 00...01 ou 11...11.

On a ensuite développé des algorithmes de calcul de la longueur propres à chaque unité. Ces algorithmes ont été implantés en matériel ce qui permet un calcul dynamique de la longueur. Avec ce système, on obtient alors l'information de longueur d'un tableau en même temps que le résultat du dernier bloc de ce tableau. Cette solution présente l'avantage de pouvoir réutiliser le résultat d'une opération dès qu'elle est terminée ce qui évite ainsi aux unités d'être inutilisées en attendant la mise à jour de la longueur.

Une autre optimisation portant sur le module de détection des dépendances a permis d'augmenter encore l'utilisation des unités fonctionnelles en autorisant le recouvrement d'opérations. Ce dernier permet de lancer une nouvelle opération bien qu'elle ait une dépendance de donnée sur un tableau avec une opération en cours d'exécution. Ce lancement n'est cependant possible que si les conditions concernant l'ordre de traitement des blocs du tableau et la fréquence de lecture et d'écriture des ces blocs sont vérifiées. Cette technique est très efficace car elle permet dans certains cas d'exécuter en parallèle des opérations qui possèdent pourtant des dépendances.

Cette solution est rendue possible par la gestion dynamique de la longueur des tableaux. La seconde opération est en effet lancée alors que la longueur d'une de ses opérandes d'entrée, provenant de la première opération, n'est pas encore connue. On utilise alors une estimation de la longueur afin de pouvoir commencer la deuxième opération. La longueur sera mise à jour dynamiquement, avant la fin de la seconde opération, permettant ainsi à cette dernière de se terminer correctement.

Chapitre 7

Coûts et performances

7.1 Coût matériel

Le placement-routage des principaux blocs du circuit a permis de déterminer leur surface (2ème colonne du tableau 7.1). Pour le cache d'instruction ou les bancs de registres, qui n'ont pas été réalisés, cette surface est estimée.

Le plan de masse du circuit a été établi en tenant compte de deux paramètres :

- La surface de chaque bloc.
- Les liaisons entre blocs.

La figure 7.1 présente le plan de masse du circuit. Les blocs qui ont été routés sont en gris alors que ceux qui ne l'ont pas été apparaissent en blanc.

Bloc	Surface (mm^2)	Géométrie (mm)	Transistors
Décodage	10,78	3,7 × 3,0	16 000
Génération des adresses	2,95	0,6 × 3,7 1,6 × 0,5	3 800
Unité d'addition	12,11	4,1 × 3,0	22 100
Unité de multiplication	26,83	3,5 × 3,4 6,5 × 2,3	127 000
Unité de division	12,35	3,9 × 3,2	28 600
Unité de décalage	15,64	4,3 × 3,5 0,4 × 1,9	34 300
Unité de charg/rang.	5,31	1,9 × 2,8	6 700
Table des longueurs	0.21	0,2 × 1,1	1 300
Cache d'instruction	12,90 (estimée)	4,4 × 3,0	170 000
Registres de données	2,50 (estimée)	0,6 × 4,2	23 000
Registres de contrôle	63,60 (estimée)	5,7 × 10,5 0,6 × 6,3	600 000
Cœur du processeur	166,40	12,8 × 13,0	1 200 000

TAB. 7.1: Caractéristiques des blocs.

Un nouveau placement-routage des blocs a été réalisé en tenant compte des contraintes géométriques du plan de masse. Ces contraintes sont données dans la

3$^{\text{ème}}$ colonne du tableau 7.1.

Le circuit total a une surface de 166,41 mm^2, soit 13 mm×12,8 mm en technologie CMOS 0,7μm, et contient environ 1.2 million de transistors.

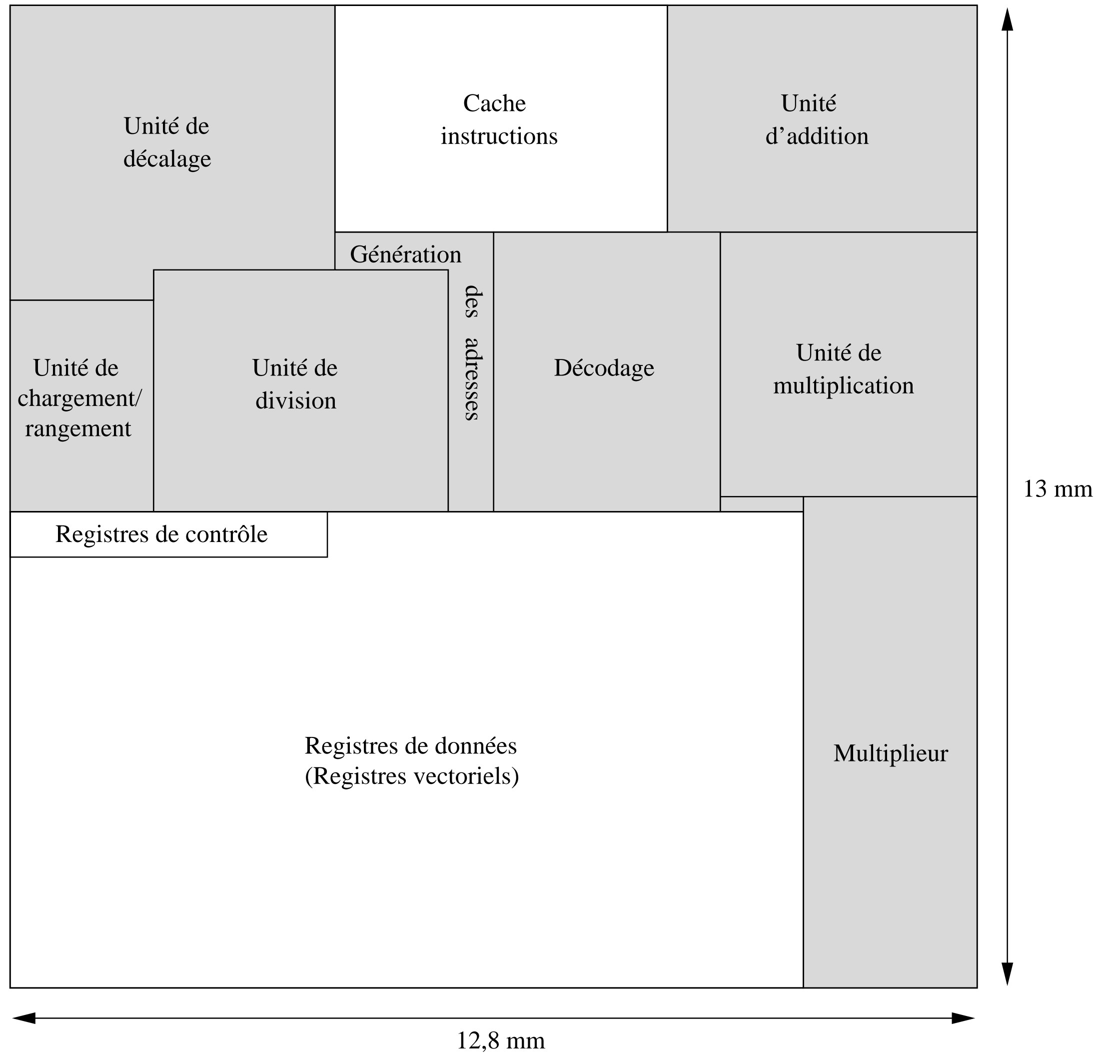


Fig. 7.1: Plan de masse du circuit.

7.2 Performances

L'utilisation d'un processeur spécialisé permet de diminuer le temps d'exécution des opérations par rapport aux solutions logicielles actuelles. Cette réduction peut être décomposée en deux parties :

- La réduction liée aux opérations élémentaires.
- La réduction liée à l'écriture de fonctions plus complexes.

7.2.1 Réduction de la durée des opérations élémentaires

Pour des données de taille fixée, la durée d'une opération élémentaire en arithmétique exacte est :

$$D_{elem} = Nb_cycles \times \frac{1}{F_{fonct}} \times \frac{1}{Format}$$

avec Nb_cycles : Nombre de cycles nécessaires pour réaliser cette opération.
F_{fonct} : Fréquence de fonctionnement du processeur.
$Format$: Format du mot machine du processeur.

Pour réduire la durée D_{elem} obtenue avec le processeur, on peut agir sur les trois paramètres en diminuant le nombre de cycles de l'opération élémentaire et en augmentant la fréquence de fonctionnement et le format du mot machine par rapport aux processeurs modernes.

Pour obtenir une fréquence de fonctionnement élevée pour le cœur du processeur, on a utilisé des unités pipelinées et des architectures dont le temps est en $O(\log_2 N)$. Cependant la fréquence est une grandeur qui est aussi fortement liée à la technologie employée pour la fabrication du circuit. Pour cette raison, il est important lorsque l'on veut faire la comparaison des performances obtenues par deux architectures d'effectuer une étude à technologie équivalente.

La technologie employée pour la réalisation du circuit n'offre pas des performances très élevées. L'utilisation d'une technologie plus récente permettrait d'effectuer des comparaisons avec les dernières générations de processeurs. Le tableau 7.2 compare les performances de la technologie utilisée (CMOS 0,7 μm de ATMEL-ES2) avec celles de la technologie CMOS 0,35 μm de Thomson. Les temps obtenus montrent que l'on peut avoir avec cette dernière un facteur d'accélération qui soit au moins égal à 2. Il peut être en effet plus élevé car ce facteur 2 ne tient pas compte du gain en surface provenant du nombre plus important de niveaux de métal. Or la réduction de la surface entraîne une diminution de la longueur moyenne des connexions et donc une augmentation de la vitesse. Le circuit qui correspond à la migration de l'architecture du processeur sur la technologie CMOS 0,35 μm de Thomson est appelé *Processeur2*. Il sera utilisé pour faire la comparaison de l'architecture de notre processeur avec celles des processeurs récents. Le tableau 7.3 présente les caractéristiques de plusieurs processeurs. La première partie (6 premières lignes) correspond aux processeurs réalisés avec des technologies "anciennes" et la seconde partie (5 dernières lignes) à ceux utilisant des technologies récentes.

	Inverseur	Nand2	Mux2 :1	Latch	Registre
Techno. 0,7 μm	0,17 ns	0,24 ns	0,74 ns	E->S 0,68 ns	Clk ->S 1.16 ns Setup 1,75 ns
Techno. 0,35 μm	0,12 ns	0,15 ns	0,37 ns	E->S 0,29 ns	Clk ->S 0,42 ns Setup 0,12 n

TAB. 7.2: Temps de traversée des cellules en technologie 0,7 μm et 0,35 μm.

Le deuxième paramètre sur lequel on peut agir pour diminuer la durée d'une opération élémentaire est la taille du mot machine. Le processeur utilise un format de 128 bits ce qui permet d'augmenter ses performances par rapport aux architectures

32 ou 64 bits. La dernière colonne du tableau 7.3 présente les formats des mots machines des différents processeurs.

	Technologie (μm)	Niveaux de métal	Fréquence (MHz)	Format (bits)
SuperSparc	0,6	3	60	32
PowerPC 601	0,6	4	66	32
486 DX 4	0,6	4	100	32
HP PA7100	0,8	3	100	32
MIPS R8000	0,7	3	75	64
Processeur	0,7	2	55	128
UltraSparc	0,42	4	200	64
PowerPC 604	0,5	5	166	32
Pentium Pro	0,35	4	200	32
HP PA8000	0,5	4	180	64
Processeur 2	0,35	5	$\geq$110	128

TAB. 7.3: Caractéristiques des processeurs.

Le troisième paramètre qui correspond au nombre de cycles a été réduit en définissant un jeu d'instruction comportant des opérations spécifiques à l'arithmétique exacte. Le tableau 7.4 permet de comparer le nombre de cycles nécessaires sur plusieurs processeurs pour exécuter différentes opérations élémentaires. Ce nombre de cycle correspond à la valeur minimum qui est obtenue pour effectuer uniquement l'opération arithmétique, sans tenir compte des fonctions annexes comme les accès mémoire ou la gestion des longueurs.

PERFORMANCES (Cycles / mot de 128 bits)	Déc. (1)	Add. (2)	Mult. (3)	Mult-Accu. (4)
SuperSparc	12	10	132	172
PowerPC 601	12	24	176	224
486 DX 4	38	36	272	320
HP PA7100	13	17	112	128
MIPS R8000	6	9	32	32
Processeur	1	1	4	4
UltraSparc	5	20	160	184
PowerPC 604	8	12	32	48
Pentium Pro	10	14	96	144
HP PA8000	6	8	28	28
Processeur 2	1	1	4	4

(1) : Décalages à gauche ou à droite. (3) : Multiplications.
(2) : Additions ou soustractions. (4) : Multiplications-Accumulations.

TAB. 7.4: Cycles nécessaires par mot de 128 bits pour effectuer des opérations élémentaires en arithmétique exacte.

Le tableau 7.5 présente la durée des opérations élémentaires sur plusieurs processeurs. Les performances obtenues pour *Processeur2* correspondent au pire cas. En effet, la fréquence de fonctionnement obtenue lors de la migration en technologie 0,35 μm pourrait être supérieure à celle de 110 MHz (utilisée pour les calculs) ce qui entraînerait une diminution de la durée des opérations.

Durée des opérations (ns / mot de 128 bits)	Déc. (1)	Add. (2)	Mult. (3)	Mult-Accu. (4)
SuperSparc	200	167	2200	2867
PowerPC 601	182	364	2667	3394
486 DX 4	380	360	2720	3200
HP PA7100	130	170	1120	1280
MIPS R8000	80	120	427	427
Processeur	18	18	72	72
UltraSparc	25	100	800	920
PowerPC 604	48	72	193	289
Pentium Pro	50	70	480	720
HP PA8000	33	44	156	156
Processeur 2	9	9	36	36

(1) : Décalages à gauche ou à droite. (3) : Multiplications.
(2) : Additions ou soustractions. (4) : Multiplications-Accumulations.

TAB. 7.5: Durée des opérations élémentaires par mot de 128 bits.

7.2.2 Réduction de la durée des fonctions complexes

Une fonction complexe correspond à un programme ou une partie de programme qui est constitué de plusieurs opérations élémentaires. Le cœur du processeur a été développée pour améliorer l'enchaînement des opérations et réduire le coût des fonctions de gestion qui sont exécutées après chaque opération élémentaire. La réduction du temps d'exécution qui en résulte est cependant extrêmement difficile à quantifier. Elle varie généralement de façon importante selon le type de calculs effectué. La comparaison avec les processeurs actuels augmente cette difficulté car leur complexité rend très difficile, voir impossible, le suivi d'un calcul au sein d'une architecture. La détection dynamique du parallélisme et des dépendances de données, les suspensions éventuelles des pipelines, les défauts de cache sont autant d'évènements difficiles à prévoir. Seule l'utilisation de programmes de test (*Benchmark*) qui s'exécutent sur le système complet permettent de tenir compte de tous ces facteurs lorsque l'on veut comparer les performances de plusieurs processeurs.

De même la mesure du gain de temps apporté par certaines solutions, comme les gestions dynamiques des débordements et des longueurs, ne peut être significative que si elle est effectuée en s'appuyant sur des exemples réels. Les caractéristiques de certains blocs du processeur n'ayant pas été définies, on ne peut pas connaitre les performances exactes de l'ensemble du processeur dans le cas de problèmes com-

plexes. On peut cependant effectuer quelques comparaisons à partir de cas simples afin d'avoir un ordre de grandeur de l'augmentation des performances.

Les opérations arithmétiques élémentaires du processeur se situent en fait à un niveau supérieur à celles utilisées par les logiciels car elles réalisent directement toutes les fonctions d'un algorithme. Pour le processeur, les fonctions supplémentaires qui apparaissent dans l'algorithme sont soit intégrées dans l'opérateur arithmétique, soit effectuées en parallèle avec l'opération, ce qui n'entraîne aucune augmentation de la durée d'exécution.

L'exemple qui est présenté ci-après correspond à l'algorithme d'addition utilisé dans le logiciel Givaro.

```
abs_lgA ⇐ VAL_ABSOLUE(lgA)                          } partie I
abs_lgB ⇐ VAL_ABSOLUE(lgB)                          }
Si abs_lgA < abs_lgB                                } partie II
alors temp ⇐ lgA
sinon lgA ⇐ lgB
      lgB ⇐ temp
      temp ⇐ abs_lgA
      abs_lgA ⇐ abs_lgB
      abs_lgB ⇐ temp
lgS ⇐ lgA + 1
Si mem_allouee_S < lgS
alors RE_ALLOUER(S, lgS)
Si signe(A) ≠ signe(B)                              } partie III
alors Si abs_lgA ≠ abs_lgB                          } partie IV
      alors SOUSTRACTION(S, A, abs_lgA, B, abs_lgB)
            lgS ⇐ abs_lgA
            NORMALISATION(S, lgS)
            Si lgA < 0
            alors lgS ⇐ −lgS
      sinon COMPARAISON(A, B, abs_lgA)
            Si A < B
            alors SOUSTRACTION(S, B, A, abs_lgA)
                  lgS ⇐ abs_lgA
                  NORMALISATION(S, lgS)
                  Si lgA ≥ 0
                  alors lgS ⇐ −lgS
            sinon SOUSTRACTION(S, A, B, abs_lgA)
                  lgS ⇐ abs_lgA
                  NORMALISATION(S, lgS)
                  Si lgA < 0
                  alors lgS ⇐ −lgS
sinon ADDITION(retenue, S, A, abs_lgA, B, abs_lgB)  } partie V
      lgS ⇐ abs_lgA + retenue
      Si lgA < 0
      alors lgS ⇐ −lgS
```

Cet algorithme effectue l'addition signée de deux nombres et détermine la longueur exacte du résultat obtenu. Les nombres A et B ne sont pas signés et ce sont les longueurs associées (lgA et lgB) qui comportent le signe.

- La partie I permet de récupérer la valeur positive de la longueur. Cette fonction n'existe pas dans l'instuction du processeur car celui ci peut travailler directement sur des nombres signés.
- La partie II inverse les deux opérandes d'entrée de façon à toujours avoir $lgA \geq lgB$. L'architecture de l'unité d'addition permet d'accepter indifféremment l'opérande la plus grande sur les entrées de l'opérateur.
- La troisième partie est un test qui a pour but de déterminer si les nombres sont de même signe pour effectuer soit l'addition soit la soustraction des deux opérandes de façon à toujours obtenir un résultat positif.
- La partie IV qui suit correspond aux différents cas possibles. Elle peut être très coûteuse en temps car elle contient plusieurs boucles. En particulier elle contient la fonction NORMALISATION qui peut nécessiter de relire tous les blocs du résultat et de tester si chacun de ces blocs est nul. De même la fonction COMPARAISON de deux nombres A et B est équivalente à une soustraction entre les deux nombres (en commençant par les poids forts). Chacune de ces deux fonctions peut être aussi coûteuse en temps que la fonction principale ADDITION.
- La cinquième partie correspond à l'addition des deux nombres (lorsqu'ils sont de même signe) et à la détermination de la longueur du résultat.

Par rapport à l'opération élémentaire d'addition non signée, qui correspond à celle réalisée directement par le processeur, l'algorithme comporte des fonctions supplémentaires qui sont coûteuses en temps. Elles peuvent multiplier la durée d'exécution de l'algorithme par trois par rapport au temps de l'addition non signée. Cette augmentation étant dépendante des données, on présente alors les deux cas limite. Le tableau 7.6 compare les performances obtenues sur plusieurs processeurs avec celles de notre processeur pour une même fonctionnalité qui correspond à l'addition signée et au calcul de la longueur exacte du résultat dans les deux cas extrêmes.

Le même problème se pose pour les opérations de multiplication-accumulation. Dans certains cas il faudra aussi faire la comparaison du résultat de la multiplication avec le nombre auquel on doit l'accumuler, puis ensuite normaliser le nouveau résultat. On se trouve alors dans le cas de l'addition décrit précédemment. La durée de l'accumulation dépend donc aussi des données. On présente dans le tableau 7.6 la durée de la fonction de multiplication-accumulation signée dans les deux cas extrêmes.

Pour les fonctions de décalage et de multiplication, l'intégration de l'algorithme complet ne modifie que très faiblement le gain obtenu au niveau de l'opération arithmétique seule. Les algorithmes pour ces deux fonctions ne comportent en effet que peu d'opérations annexes qui de plus s'exécutent en un temps constant qui peut être négligé devant les temps de calcul.

Les tableaux 7.7 et 7.8 résument les accélérations apportées par le processeur et le processeur2 pour différentes opérations. Les tableaux comparent ces deux circuits à plusieurs processeurs réalisés dans des technologies similaires.

Fonctions complexes (ns / mot de 128 bits)	Addition		Multiplication-accumulation	
	Meilleur cas	Pire cas	Meilleur cas	Pire cas
SuperSparc	167	501	2867	4201
PowerPC 601	364	1092	3394	4848
486 DX 4	360	1080	3200	4160
HP PA7100	170	510	1280	1600
MIPS R8000	120	360	427	667
Processeur	18	18	72	72
UltraSparc	100	300	920	1160
PowerPC 604	72	216	289	481
Pentium Pro	70	210	720	1200
HP PA8000	44	132	156	244
Processeur 2	9	9	36	36

TAB. 7.6: Durée des fonctions d'addition et de multiplication-accumulation.

Facteur d'accélération (par mot de 128 bits)	Déc. (1)	Add. (2)	Mult. (3)	Mult-Accu. (4)
SuperSparc	11,1	9,3 à 27,8	30,6	39,8 à 58,3
PowerPC 601	10,1	20,2 à 60,7	37,0	47,1 à 67,3
486 DX 4	21,1	20,0 à 60,0	37,8	44,4 à 57,8
HP PA7100	7,2	9,4 à 28,3	15,6	17,8 à 22,2
MIPS R8000	4,4	6,7 à 20,0	5,9	5,9 à 9,3

(1) : Décalages à gauche ou à droite. (3) : Multiplications.
(2) : Additions ou soustractions. (4) : Multiplications-Accumulations.

TAB. 7.7: Facteur d'accélération apporté par le processeur par rapport à des processeurs utilisant une technologie similaire.

Facteur d'accélération (par mot de 128 bits)	Déc. (1)	Add. (2)	Mult. (3)	Mult-Accu. (4)
UltraSparc	2,8	11,1 à 33,3	22,2	25,6 à 32,2
PowerPC 604	5,3	8,0 à 24,0	5,4	8,0 à 13,4
Pentium Pro	5,6	7,8 à 23,3	13,3	20,0 à 33,3
HP PA8000	3,7	4,9 à 14,7	4,3	4,3 à 6,8

(1) : Décalages à gauche ou à droite. (3) : Multiplications.
(2) : Additions ou soustractions. (4) : Multiplications-Accumulations.

TAB. 7.8: Facteur d'accélération apporté par le processeur2 par rapport à des processeurs utilisant une technologie similaire.

Chapitre 8

Conclusions et perspectives

La perte de précision dans les calculs en arithmétique virgule flottante nécessite d'utiliser des logiciels de calcul exact lorsque l'on veut des résultats fiables ou sans erreur. Or ces logiciels sont pénalisés en temps par le matériel inadapté sur lequel ils s'exécutent.

Nous avons présenté dans cette thèse un cœur de processeur dédié au calcul exact. Il possède un jeu d'instruction qui intègre toutes les opérations élémentaires associées à une telle arithmétique. De plus, il dispose d'instructions sur des tableaux de taille variable qui sont associées, au niveau matériel, à des unités fonctionnelles optimisées pour enchaîner des calculs sur les blocs consécutifs d'un nombre. Ces instructions réduisent le coût des calculs sur les nombres de grande taille générés en calcul exact par rapport aux instructions élémentaires sur les blocs.

Cette diminution du coût est aussi liée aux opérateurs arithmétiques qui composent chaque unité fonctionnelle. L'utilisation de la notation redondante ou d'architectures en temps $O(\log_2 N)$ leur permet de travailler sur des blocs de 128 bits tout en maintenant des temps de traversée faibles. De plus ces opérateurs peuvent effectuer dynamiquement la gestion des signes par l'intermédiaire d'architectures pouvant exécuter des calculs sur des opérandes signées ou non signées. Ils traitent aussi dynamiquement les dépassements de capacité générés lors de certaines opérations sans introduire de cycle d'horloge supplémentaire.

L'intégration au niveau matériel du calcul concernant la longueur d'un résultat permet de diminuer fortement la durée totale d'une opération. Le calcul de la longueur est effectué dynamiquement en parallèle avec l'exécution de l'opération qui doit retourner le résultat. Cette solution permet d'obtenir la longueur exacte d'un nombre en même temps que le résultat du dernier bloc de ce nombre. Elle autorise aussi sous certaines conditions le recouvrement d'opérations sur des tableaux qui possèdent des dépendances de données. Cette solution, qui nécessite une estimation de la longueur du calcul en cours, permet d'avoir une diminution des cycles de suspension liés aux dépendances de données entre des instructions. Le gain obtenu par l'implantation de cette solution sera d'autant plus grand que la taille des tableaux sera importante.

La réalisation à partir d'une bibliothèque de cellules standards des parties les plus importantes de ce circuit a montré qu'une solution matérielle pour le calcul exact permettait d'obtenir une accélération importante des calculs par rapport aux solutions logicielles. Cette réalisation a aussi permis de vérifier en particulier que

la surface occupée par ce circuit n'était pas un obstacle à sa fabrication avec les technologies dont on dispose aujourd'hui.

Le travail réalisé dans cette thèse montre que l'utilisation d'une architecture dédiée au calcul exact peut être une solution efficace face aux problèmes de précision rencontrés aujourd'hui avec les processeurs classiques. Jusqu'à maintenant, les problèmes de précision ont été résolus en augmentant le format utilisé pour le codage des nombres en virgule flottante. Cette solution ne fait cependant que retarder l'apparition des problèmes sans les traiter. De plus, elle est coûteuse car elle nécessite la réécriture de tous les logiciels. La nouvelle architecture qui a été développée apporte une réponse à long terme plus efficace et peut être moins coûteuse que celle proposée aujourd'hui.

Bibliographie

[1] A. Avizienis. “Signed Digit Number Representations for Fast Parallel Arithmetic”. *IRE Transactions on Electronic Computers*, EC-10 :389–400, Septembre 1961.

[2] C. Batut. “Aspects Algorithmiques du Système de Calcul Arithmétique en Multiprécision PARI”. Thèse, Université de Bordeaux 1, 1989.

[3] F. Baude and D. Skillicorn. “Vers de la Programmation Parallèle Structurée fondée sur la Théorie des Catégories”. *Technique et Science Informatiques*, 13(4) :525–538, 1994.

[4] M. O. Benouamer, P. Jaillon, D. Michelucci, and J. M. Moreau. “A Lazy Arithmetic Library”. In *Proceedings of the 11th Symposium on Computer Arithmetic*, pages 242–249, Juin 1993.

[5] D. Bhandarkar and R. Brunner. “VAX Vector Architecture”. In *Proceedings of the 17th Symposium on Computer Architecture*, pages 204–215, Mai 1990.

[6] G. Bohlender, W. Walter, P. Kornerup, and D. W. Matula. “Semantics for Exact Floating Point Operations”. In *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 22–27, Juin 1991.

[7] R. T. Brent and H. Kung. “A Regular Layout for Parallel Adders”. *IEEE Transactions on Computers*, C-31(3) :260–264, Mars 1982.

[8] W. Buchholz. “The IBM System/370 Vector Architecture”. *IBM System Journal*, 25(1) :51–62, 1986.

[9] B. W. Char, K. O. Geddes, and G. H. Gonnet. *Maple V : Language Reference Manual*. Springer, 1991.

[10] B. W. Char, K. O. Geddes, and G. H. Gonnet. *Maple V : Library Reference Manual*. Springer, 1991.

[11] F. Chatelin and V. Frayssé. “Analysis of Arithmetic Algorithms : A Statistical Study”. In *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 10–16, Juin 1991.

[12] H. Cheng. “Vector Pipelining, Chaining, and Speed on the IBM 3090 and Cray X-MP”. *IEEEăComputer*, pages 31–46, Septembre 1989.

[13] J. M. Chesnaux. “Etude Théorique et Implémentation en ADA de la Méthode CESTAC”. Thèse, Université Paris VI, 1988.

[14] M. Christaller. “Vers un support d’exécution portable pour applications parallèles irrégulières : Athapascan-0”. Thèse, Université Joseph Fourier, Grenoble, Novembre 1996.

[15] V. Coissard and A. Guyot. “Un Coprocesseur pour l’Arithmétique exacte”. In *Proceedings of the 4th Symposium sur les architectures nouvelles de machines”*, pages 173–182, Février 1996.

[16] V. Coissard and A. Guyot. “OCAPI : A Coprocessor for Infinite Precision Arithmetic”. In *Proceedings of SCAN’97*, pages VIII.5–VIII.9, Septembre 1997.

[17] M. Daumas. “Contribution à l’Arithmétique des Ordinateurs : Vers une Maîtrise de la Précision”. Thèse, Ecole Normale Supérieure de Lyon, Janvier 1996.

[18] T. A. Diep, C. Nelson, and J.P.Shen. “Performance Evaluation of the PowerPC 620 Microarchitecture”. In *Proceedings of the 22th Symposium on Computer Architecture*, pages 163–174, Mai 1995.

[19] R. Edenfield, M. Gallup, W. Ledbetter, et al. “The 68040 Processor, Part 1, Design and Implementation”. *IEEE Micro*, 10(1) :66–78, Février 1990.

[20] J. H. Edmondson, P. Rubinfeld, and R. Preston. “Superscalar Instruction Execution in the 21164 Alpha Microprocessor”. *IEEE Micro*, 15(4) :33–43, Avril 1995.

[21] J. E. Ely. “The VPI Software Package for Variable-Precision Interval Arithmetic”. *Interval Computations*, 2 :135–153, 1993.

[22] M. D. Ercegovac and T. Lang. “On-the-fly Conversion of Redundant into Conventional Representations”. *IEEE Transactions on Computers*, C-36(7) :895–897, Juillet 1987.

[23] A. Feldstein and P. Turner. “Overflow, Underflow and Severe Loss of Significance in Floating Point Addition and Subtraction”. *IMA Journal of Numerical Analysis*, 6 :241–251, 1986.

[24] J. A. Fisher. “Very Long Instruction Word Architectures and the ELI-512”. In *Proceedings of the 10th Symposium on Computer Architecture*, pages 140–150, Juin 1983.

[25] T. Gautier. “Calcul Formel et Parallélisme : Conception du Système GIVARO et Applications au Calcul dans les Extensions Algébriques”. Thèse, Institut National Polytechnique de Grenoble, Juin 1996.

[26] T. Gautier and J. L. Roch. “PAC++ System and Parallel Algebraic Numbers Computation”. In *Proceedings of the 1st International Symposium on Parallel Symbolic Computation PASCO’94*, pages 145–153, Septembre 1994.

[27] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith. “Cache Performance of the SPEC92 Benchmark Suite”. *IEEE Micro*, 13(4) :17–27, Avril 1993.

[28] I. Ginzburg. “Athapascan-0b : Integration efficace et portable de multiprogrammation légère et de communications”. Thèse, Institut National Polytechnique de Grenoble, Septembre 1997.

[29] A. Guyot, M. Belrhiti, and G. Bosco. “Adders Synthesis”. In *Proceedings of IFIP Workshop on Logic and Architecture Synthesis*, pages 280–286, Grenoble, Décembre 1994.

[30] T. Han and D. A. Carlson. “Fast Area-Efficient VLSI Adders”. In *Proceedings of the 8th Symposium on Computer Arithmetic*, pages 49–56, Mai 1987.

[31] J. L. Hennessy and D. A. Patterson. *Architecture des ordinateurs*. International Thomson Publishing, 1996.

[32] R. W. Horst, R. L. Harris, and R. L. Jardine. “Multiple Instruction Issue in the NonStop Cyclone Processor”. In *Proceedings of the 17th Symposium on Computer Architecture*, pages 216–226, Mai 1990.

[33] K. Hwang. *Advanced Computer Architecture : Parallelism, Scalability, Programmability*. Mc Graw Hill, 1993.

[34] IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*. IEEE Computer Society Press, 1985.

[35] P. Jaillon. “Proposition d’une Arithmétique Rationnelle Paresseuse et d’un Outil d’Aide à la Saisie d’Objets en Synthèse d’Images”. Thèse, Ecole des Mines de Saint-Etienne, Septembre 1993.

[36] T. Jebelean. “An Algorithm for Exact Division”. *Journal of Symbolic Computation*, 15(2) :169–180, Février 1993.

[37] R. D. Jenks and R. S. Sutor. *Axiom : The Scientific Computation System*. Springer, 1992.

[38] M. Karasick. “On the Representation and Manipulation of Rigid Solids”. Thèse, Cornell University, Ithaca (NY), US, Mars 1989.

[39] J. Kernhof, C. Baumhof, B. Höfflinger, et al. “A CMOS Floating-Point Processing Chip for Verified Exact Vector Arithmetic”. In *Proceedings of ESSCIRC’94*, pages 1–4, 1994.

[40] R. Klatte, U. Kulisch, C. Lawo, et al. *C-XSC : A C++ Class Library for Extended Scientific Computing*. Springer-Verlag, 1993.

[41] A. Knöfel. “Fast Hardware Units for the Computation of Accurate Dot Products”. In *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 70–74, Juin 1991.

[42] P. M. Kogge. *The Architecture of Pipelined Computers*. Mc Graw Hill, 1981.

[43] P. M. Kogge and H. S. Stone. “A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations”. *IEEE Transactions on Computers*, C-22(8) :786–792, Août 1973.

[44] P. Kornerup and D. W. Matula. “An On-line Arithmetic Unit for Bit-Pipelined Rational Arithmetic”. *Journal of Parallel and Distributed Computing*, 5(3) :310–330, Mai 1988.

[45] U. Kulisch and W. Miranker. *Computer Arithmetic in Theory and in Practice*. Academic Press, 1981.

[46] R. Lee. “Precision Architecture”. *Computer*, 22(1) :78–91, Janvier 1989.

[47] J. Lenell and N. Bagherzadeh. “A Performance Comparaison of Several Superscalar Processor Models with a VLIW Processor”. In *Proceedings of the 7th International Parallel Processing Symposium*, pages 44–48, 1993.

[48] D. W. Matula and P. Kornerup. “Finite Precision Rational Arithmetic : Slash Number Systems”. *IEEE transactions on Computers*, C-34(1) :3–18, Janvier 1985.

[49] D. Michelucci. "Lazy Arithmetic". *IEEE Transactions on Computers*, 46(9) :961–975, Septembre 1997.

[50] S. Mirapuri, M. Woodacre, and N. Vasseghi. "The MIPS R4000 Processor". *IEEE Micro*, 12(2) :10–22, Avril 1992.

[51] M. Müller, C. Rüb, and W. Rülling. "Exact Accumulation of Floating-Point Numbers". In *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 64–69, Juin 1991.

[52] B. Moore, A. Padegs, R. Smith, and W. Buchholz. "Concepts of the System/370 Vector Architecture". In *Proceedings of the 14th Symposium on Computer Architecture*, pages 282–288, Juin 1987.

[53] R. E. Moore. *Interval Analysis*. Prentice Hall, 1963.

[54] I. Moussa, A. Skaf, and A. Guyot. "Design of a GaAs Redundant Divider". In *Proceedings of VLSI'93*, pages 63–72, Septembre 1993.

[55] J. M. Muller. *Arithmétique des Ordinateurs*. Masson, 1989.

[56] M. A. Naal. "Synthèse et Optimisation d'Opérateurs Arithmétiques en Grande Précision". Rapport de DEA, Laboratoire TIMA, Université Joseph Fourier, Grenoble, Juin 1997.

[57] M. Nakajima, H. Nakano, Y. Nakakura, et al. "OHMEGA : A VLSI Superscalar Processor Architecture for Numerical Applications". In *Proceedings of the 18th Symposium on Computer Architecture*, pages 160–168, mai 1991.

[58] A. M. Nielsen and P. Kornerup. "MSB-First Digit Serial Arithmetic". In *Proceedings of Real Numbers and Computers*, pages 23–43, Saint-Etienne, Avril 1995.

[59] T. Ottman, G. Thiemt, and C. Ullrich. "Numerical Stability of Geometric Algorithms". In *Proceedings of the 3rd Annual ACM Symposium on Computational Geometry*, pages 119–125, 1987.

[60] P. Ozello. "Calcul Exact des Formes de Jordan et Frobenius d'une Matrice". Thèse, Institut National Polytechnique de Grenoble, Janvier 1987.

[61] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design :The Hardware/Software Interface*. Morgan Kaufmann, 1994.

[62] M. Pichat. "Contribution à l'Etude des Erreurs d'Arrondi en Arithmétique Virgule Flottante". Thèse, Université Scientifique et médicale de Grenoble, 1976.

[63] M. La Porte and J. Vignes. "Etude statistique des erreurs dans l'arithmétique des ordinateurs ; Application au contrôle des résultats d'algorithmes numériques". *Numerische Mathematik*, 23(1) :63–72, 1974.

[64] D. M. Priest. "Algorithms for Arbitrary Precision Floating Point Arithmetic". In *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 132–143, Juin 1991.

[65] D. Ratz. "The Effects of the Arithmetic of Vector Computers on Basic Numerical Algorithms". In *Proceedings of SCAN'89*, Octobre 1989.

[66] F. Roch. "Calcul Formel et Parallélisme : Formes Normales d'Hermite : Méthodes de Calcul et Parallélisation". Thèse, Institut National Polytechnique de Grenoble, Janvier 1990.

[67] J. L. Roch. "L'Architecture du Système PAC et son Arithmétique Rationnelle". Thèse, Institut National Polytechnique de Grenoble, Décembre 1989.

[68] T. Sato, M. Nakajima, T. Sukemura, and G. Goto. "A Regularly Structured 54-bit Modified-Wallace-Tree Multiplier". In *Proceedings of VLSI'91*, pages 1–9, Août 1991.

[69] M. Schulte and E. E. Schwartzlander. "Exact Rounding of Certain elementary Functions". In *Proceedings of the 11th Symposium on Computer Arithmetic*, Juin 1993.

[70] M. Schulte and E. E. Schwartzlander. "Hardware Design and Arithmetic Algorithms for a Variable-Precision, Interval Arithmetic Coprocessor". In *Proceedings of the 12th Symposium on Computer Arithmetic*, pages 222–229, Juillet 1995.

[71] A. Skaf. "Conception de Processeurs Arithmétiques Redondants et En-Ligne". Thèse, Institut National Polytechnique de Grenoble, Septembre 1995.

[72] J. Sklansky. "Conditionnal-Sum Addition Logic". *IRE transactions on Electronic Computers*, EC-9 :226–231, Juin 1960.

[73] K. So and V. Zecca. "Cache Performance of Vector Processors". In *Proceedings of the 15th Symposium on Computer Architecture"*, pages 261–268, Mai 1988.

[74] G. S. Sohi. "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers". *IEEE Transactions on Computers*, C-39(3) :349–359, Mars 1990.

[75] K. S. Trivedi and M. D. Ercegovac. "On-Line Algorithms for Division and Multiplication". *IEEE Transactions on Computers*, C-26(7) :681–687, Juillet 1977.

[76] M. Troiani, S. S. Ching, N. N. Quaynor, et al. "The VAX 8600 I Box, a Pipelined Implementation of the VAX Architecture". *Digital Technical Journal*, 1 :4–19, Août 1985.

[77] C. Ullrich. *Computer Arithmetic and Self-Validating Numerical Methods*. Academic Press, 1990.

[78] L. G. Valiant. "A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8) :103–111, Août 1990.

[79] J. Vignes. "Contrôle et Estimation Stochastique des Arrondis de Calcul". *AFCET Interfaces*, 54 :3–11, Avril 1987.

[80] K. Weber. "The Accelerated Integer GCD Algorithm". *ACM Transactions on Mathematical Software*, 21 :111–122, Mars 1995.

[81] N. H. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison-Wesley, 1993.

[82] S. Wolfram. *Mathematica : A System for doing Mathematics by Computer*. Addison-Wesley, 1988.

Annexe A

Architecture du multiplieur

Cette annexe contient une description plus précise de l'architecture d'un multiplieur en arbre de réducteurs 4/2. Le multiplieur peut se décomposer en trois parties :

- La tête qui correspond à la partie poids forts.
- Le corps qui correspond à la partie centrale.
- La queue qui correspond à la partie poids faibles.

Dans une architecture classique de multiplieur en arbre de réducteurs 4/2, la partie centrale est totalement régulière contrairement à la tête et à la queue. L'architecture proposée permet d'obtenir aussi une structure régulière, quelque soit la taille des opérandes à multiplier, pour la tête et la queue.

Les tableaux A.1 et A.2 présentent la disposition des cellules de réducteur 4/2 et de *Full-Adder* (pour les extrémités des rangées) respectivement pour la queue et la tête dans le cas du multiplieur 128×32 bits qui est implanté dans le processeur. Le câblage de la queue du multiplieur, qui correspond à la zone comprise entre les "tranches" de poids 0 et de poids 30, est présenté figure A.1. Celui de la tête, qui est comprise entre les "tranches" de poids 128 et de poids 158, est donné figure A.2. Pour ces deux zones, la distribution des produits partiels sur les cellules possède la même régularité que la partie centrale. Cela permet d'utiliser les mêmes cellules que celles de la partie centrale et de maintenir la continuïté dans la propagation des opérandes d'entrée au sein du multiplieur.

La partie centrale du multiplieur 128×32 bits, qui s'étend de la "tranche" de poids 31 à la "tranche" de poids 127, est totalement régulière. Le câblage d'une "tranche" de cette zone, constituée de 15 réducteurs 4/2, est présenté figure A.3.

(1)	(3)	(4)		(2)	(3)	(4)		(2)	(3)	(4)		(2)	(3)	(4)	
0												0	1	0	
1												0	2	0	F
2												0	3	0	F
3												0	4	0	R
4								0	2	0	F	1	3	1	R
5								0	4	0	R	2	2	1	R
6								0	6	1	R F	3	1	1	R
7								0	8	1	R R	4	0	1	R
8				0	2	0	F	1	7	2	R R	4	0	1	R
9				0	4	0	R	2	6	2	R R	4	0	1	R
10				0	6	1	R F	3	5	2	R R	4	0	1	R
11				0	8	1	R R	4	4	2	R R	4	0	1	R
12				0	10	2	R R F	5	3	2	R R	4	0	1	R
13				0	12	2	R R R	6	2	2	R R	4	0	1	R
14				0	14	3	R R R F	7	1	2	R R	4	0	1	R
15				0	16	3	R R R R	8	0	2	R R	4	0	1	R
16	2	0	F	1	15	4	R R R R	8	0	2	R R	4	0	1	R
17	4	0	R	2	14	4	R R R R	8	0	2	R R	4	0	1	R
18	6	1	R F	3	13	4	R R R R	8	0	2	R R	4	0	1	R
19	8	1	R R	4	12	4	R R R R	8	0	2	R R	4	0	1	R
⋮	⋮	⋮	⋮ ⋮ ⋮ ⋮ ⋮ ⋮	⋮	⋮	⋮	⋮ ⋮ ⋮ ⋮	⋮	⋮	⋮	⋮ ⋮	⋮	⋮	⋮	⋮
27	24	5	R R R R R R	12	4	4	R R R R	8	0	2	R R	4	0	1	R
28	26	6	R R R R R R F	13	3	4	R R R R	8	0	2	R R	4	0	1	R
29	28	6	R R R R R R R	14	2	4	R R R R	8	0	2	R R	4	0	1	R
30	30	7	R R R R R R R F	15	1	4	R R R R	8	0	2	R R	4	0	1	R
31	32	7	R R R R R R R R	16	0	4	R R R R	8	0	2	R R	4	0	1	R
32	32	8	R R R R R R R R	16	0	4	R R R R	8	0	2	R R	4	0	1	R
33	32	8	R R R R R R R R	16	0	4	R R R R	8	0	2	R R	4	0	1	R

(1) Poids des entrées. (2) Bits générés par l'étage supérieur.
(3) Produits partiels. (4) Retenues horizontales issues de la tranche précédente.
R : Réducteurs 4/2. F : *Full-Adder.*

TAB. A.1: Disposition des cellules du multiplieur en fonction du poids des entrées.

(1)	158	157	156	155	154	153	152	151	150	149	148	147	146	145	144	143	142	141	…	129	128	127
(3)																2	4	6	…	30	31	32
(4)																1	1	2	…	8	8	8
																				F	R	R
																			…	R	R	R
																			…	R	R	R
																			…	R	R	R
																			…	R	R	R
																			…	R	R	R
																		F	…	R	R	R
																F	R	R	…	R	R	R
(2)								0	0	0	0	0	0	0	1	2	3	4	…	16	16	16
(3)								2	4	6	8	10	12	14	15	14	13	12	…	0	0	0
(4)								1	1	2	2	3	3	4	4	4	4	4	…	4	4	4
														F	R	R	R	R	…	R	R	R
												F	R	R	R	R	R	R	…	R	R	R
										F	R	R	R	R	R	R	R	R	…	R	R	R
								F	R	R	R	R	R	R	R	R	R	R	…	R	R	R
(2)				0	0	0	1	2	3	4	5	6	7	8	8	8	8	8	…	8	8	8
(3)				2	4	6	7	6	5	4	3	2	1	0	0	0	0	0	…	0	0	0
(4)				1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	…	2	2	2
						F	R	…	R	R	R	R	R	R	R	R	R	R	…	R	R	R
				F	R	R	R	…	R	R	R	R	R	R	R	R	R	R	…	R	R	R
(2)	0	0	1	2	3	4	4	4	4	4	4	4	4	4	4	4	4	4	…	4	4	4
(3)	1	2	3	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	…	0	0	0
(4)	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…	1	1	1
		F	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	…	R	R	R

(1) Poids des entrées. (2) Bits générés par l'étage supérieur.
(3) Produits partiels. (4) Retenues horizontales issues de la tranche précédente.
R : Réducteurs 4/2. F : *Full-Adder*.

TAB. A.2: Disposition des cellules du multiplieur en fonction du poids des entrées.

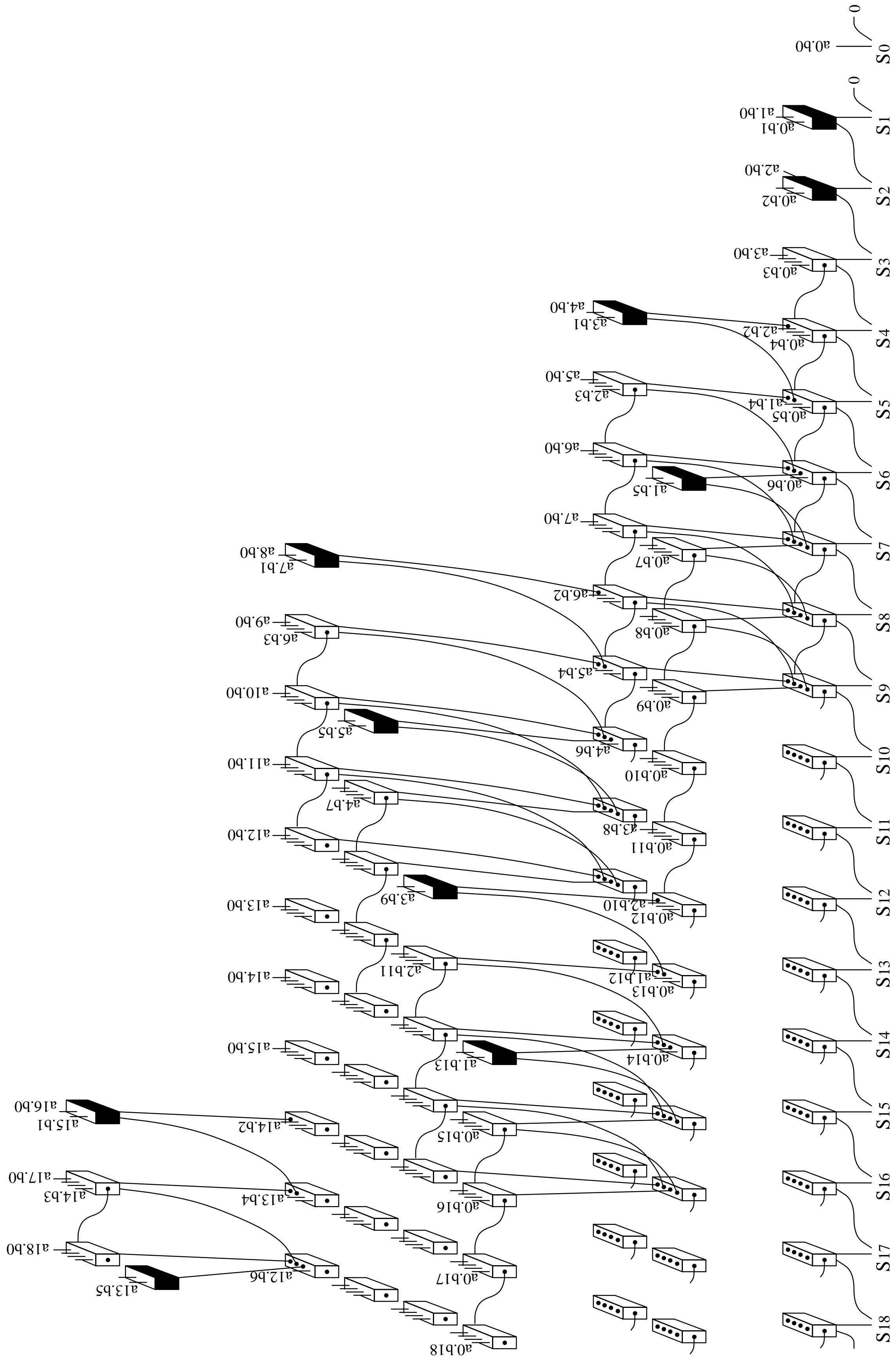


FIG. A.1: Câblage de la zone de queue du multiplieur.

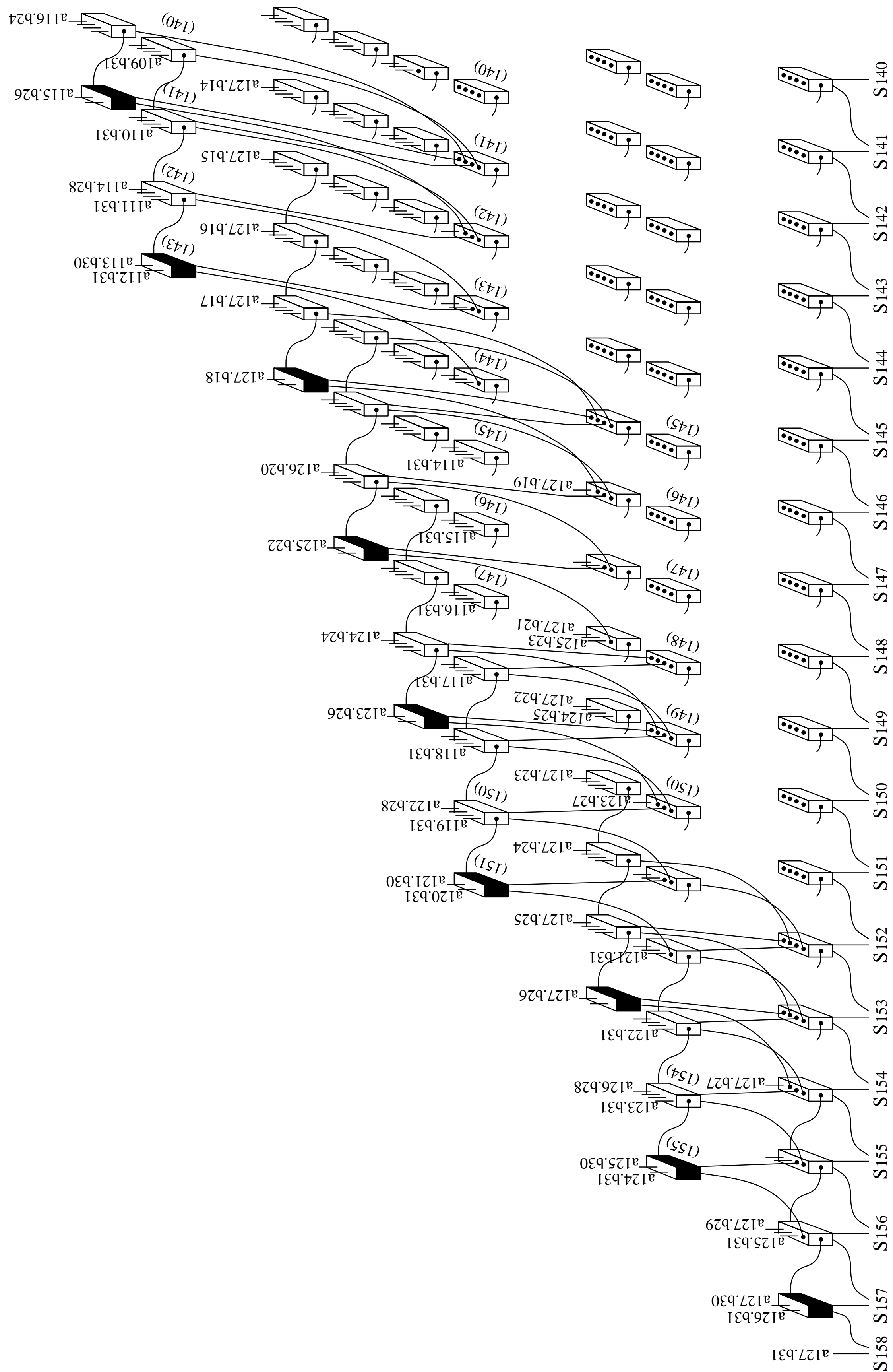


FIG. A.2: Câblage de la zone de tête du multiplieur.

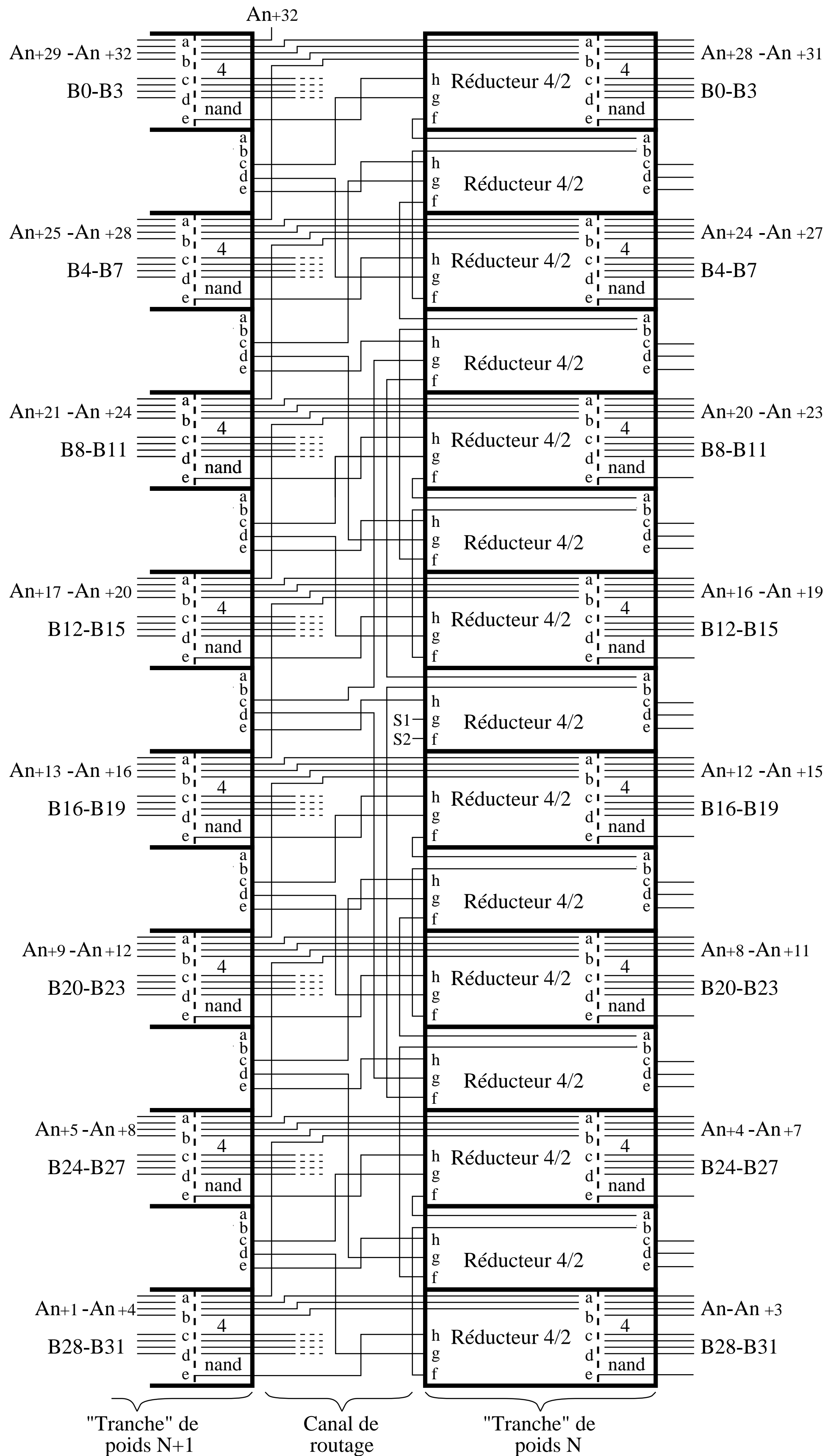


Fig. A.3: Câblage d'une "tranche" de réducteurs 4/2 de la zone centrale.

Annexe B

Programmation VHDL

Le fichier `elem_composants.vhd` contient la description VHDL des composants élémentaires utilisés par l'ensemble des blocs constituant le coprocessseur.

```
------------ Démultiplexeur -------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.elem_fonctions.all;

entity demux is
port( a, c   : in std_ulogic;
      s0, s1  : out std_ulogic
    );
end demux;

architecture arch_demux of demux is
begin
  s0 <= a when c='0'
          else 'Z';
  s1 <= a when c='1'
          else 'Z';
end arch_demux;

--------------- Full adder --------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.elem_fonctions.all;

entity FA32 is
port( a, b, cin : in std_ulogic;
      s, cout    : out std_ulogic
    );
end FA32;
```

```
architecture arch_FA32 of FA32 is
begin
  cout   <= majorite(a,b,cin) ;
  s      <= a xor b xor cin ;
end arch_FA32 ;

------------- Réducteur 4/2 -------------
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.elem_fonctions.all ;

entity add42 is
port( a, b, c, d, e : in std_ulogic ;
      f, g, h       : out std_ulogic
    ) ;
end add42 ;

architecture arch_add42 of add42 is
  signal tmp : std_ulogic ;
begin
  h   <= majorite(a,b,c) ;
  tmp <= modulo2(d,a,b,c) ;
  g   <= mux2(d,e,tmp) ;
  f   <= tmp xor e ;
end arch_add42 ;

------------- Registre 1 bit ------------
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;

entity registre is
port (clk, d  : in std_ulogic ;
      q       : out std_ulogic
     ) ;
end registre ;

architecture archi_reg of registre is
begin
  process(clk, d)
   begin
    if (clk'event and clk = '1')
      then q <= d ;
    end if ;
  end process ;
end archi_reg ;
```

```
------------ Registre 4 bits ------------
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;

entity registre4 is
port (clk     : in  std_ulogic ;
      d       : in  std_ulogic_vector(3 downto 0) ;
      q       : out std_ulogic_vector(3 downto 0)
     ) ;
end registre4 ;

architecture archi_4reg of registre4 is
begin
  process(clk, d)
   begin
    if (clk'event and clk = '1')
      then q <= d ;
    end if ;
  end process ;
end archi_4reg ;

-------- Registre avec mise à 1 ---------
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;

entity registreSET is
port (clk, init, d  : in std_ulogic ;
      q             : out std_ulogic
     ) ;
end registreSET ;

architecture archi_reg_set of registreSET is
begin
  process(clk, init, d)
   begin
    if init = '0'
           then q <= '1' ;
           elsif (clk'event and clk = '1')
                 then q <= d ;
    end if ;
  end process ;
end archi_reg_set ;

-------- Registre avec mise à 0 ---------
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
```

```
entity registreRESET is
port (clk, init, d  : in std_ulogic ;
      q             : out std_ulogic
     ) ;
end registreRESET ;

architecture archi_reg_reset of registreRESET is
begin
  process(clk, init, d)
   begin
    if init = '0'
           then q <= '0' ;
           elsif (clk'event and clk = '1')
                 then q <= d ;
    end if ;
  end process ;
end archi_reg_reset ;

------ Registre avec mise à 0 et 1 ------
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;

entity registreSET_RESET is
port (clk, init, val_init, d  : in std_ulogic ;
      q                       : out std_ulogic
     ) ;
end registreSET_RESET ;

architecture archi_reg_set_reset of registreSET_RESET is
begin
  process(clk, init, val_init, d)
   begin
    if init = '0'
           then if val_init = '0'
                          then q <= '0' ;
                          else q <= '1' ;
                end if ;
           elsif (clk'event and clk = '1')
                 then q <= d ;
    end if ;
  end process ;
end archi_reg_set_reset ;

---------- Retard de 2 cycles -----------
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
```

```
entity retard_2 is
port (clk, init, d : in std_ulogic ;
      q            : out std_ulogic
     ) ;
end retard_2 ;

architecture arch_retard_2 of retard_2 is
   signal tmp1 : std_ulogic ;
begin
  process(clk, init, d)
   begin
    if init = '0' then
                    tmp1 <= '0' ;
                    q    <= '0' ;
    elsif (clk'event and clk = '1') then
                                      q    <= tmp1 ;
                                      tmp1 <= d ;
    end if ;
  end process ;
end arch_retard_2 ;

---------- Retard de 3 cycles -----------
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;

entity retard_3 is
port (clk, init, d : in std_ulogic ;
      q            : out std_ulogic
     ) ;
end retard_3 ;

architecture arch_retard_3 of retard_3 is
   signal tmp1, tmp2 : std_ulogic ;
begin
  process(clk, init, d)
   begin
    if init = '0' then
                    tmp1 <= '0' ;
                    tmp2 <= '0' ;
                    q    <= '0' ;
    elsif (clk'event and clk = '1') then
                                      q    <= tmp2 ;
                                      tmp2 <= tmp1 ;
                                      tmp1 <= d ;
    end if ;
  end process ;
end arch_retard_3 ;
```

```
---------- Retard de 4 cycles -----------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;


entity retard_4 is
port (clk, init, d : in std_ulogic;
      q            : out std_ulogic
     );
end retard_4;

architecture arch_retard_4 of retard_4 is
   signal tmp1, tmp2, tmp3 : std_ulogic;
begin
  process(clk, init, d)
   begin
    if init = '0' then
                    tmp1 <= '0';
                    tmp2 <= '0';
                    tmp3 <= '0';
                    q    <= '0';
    elsif (clk'event and clk = '1') then
                                      q    <= tmp3;
                                      tmp3 <= tmp2;
                                      tmp2 <= tmp1;
                                      tmp1 <= d;
    end if;
  end process;
end arch_retard_4;

------- Cellule de Brent et Kung --------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity PG_c_b is
port( Pa, Ga, Pb, Gb : in  std_ulogic;
      Ps, Gs         : out std_ulogic
    );
end PG_c_b;

architecture arch_PG_c_b of PG_c_b is
begin
  Ps <= Pa nand Pb;
  Gs <= not(Ga or (Pa and Gb));
end arch_PG_c_b;
```

```
------- Cellule de Brent et Kung --------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity PG_b_c is
port( Pa, Ga, Pb, Gb : in  std_ulogic;
      Ps, Gs         : out std_ulogic
    );
end PG_b_c;

architecture arch_PG_b_c of PG_b_c is
begin
  Ps <= Pa nor Pb;
  Gs <= not(Ga and (Pa or Gb));
end arch_PG_b_c;

------- Cellule de Brent et Kung --------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity Piib is
port( a, b : in  std_ulogic;
      s    : out std_ulogic
    );
end Piib;

architecture arch_Piib of Piib is
begin
  s <= not(a xor b);
end arch_Piib;

------- Cellule de Brent et Kung --------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity Giib is
port( a, b : in  std_ulogic;
      s    : out std_ulogic
    );
end Giib;

architecture arch_Giib of Giib is
begin
  s <= a nand b;
end arch_Giib;
```

```
------- Cellule de Brent et Kung --------
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;

entity Giib_bis is
port( a, b : in  std_ulogic ;
      s    : out std_ulogic
    ) ;
end Giib_bis ;

architecture arch_Giib_bis of Giib_bis is
begin
  s <= not(a nor b) ;
end arch_Giib_bis ;

-----------------     Package     -----------------
----------------- elem_composants -----------------

library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;

package elem_composants is

component demux
port( a, c   : in std_ulogic ;
      s0, s1  : out std_ulogic
    ) ;
end component ;

component FA32
port( a, b, cin : in std_ulogic ;
      s, cout    : out std_ulogic
    ) ;
end component ;

component add42
port( a, b, c, d, e : in std_ulogic ;
      f, g, h       : out std_ulogic
    ) ;
end component ;

component registre
port (clk, d  : in std_ulogic ;
      q       : out std_ulogic
     ) ;
end component ;
```

```
component registre4
port (clk     : in  std_ulogic;
      d       : in  std_ulogic_vector(3 downto 0);
      q       : out std_ulogic_vector(3 downto 0)
     );
end component;

component registreSET
port (clk, init, d  : in std_ulogic;
      q             : out std_ulogic
     );
end component;

component registreRESET
port (clk, init, d  : in std_ulogic;
      q             : out std_ulogic
     );
end component;

component registreSET_RESET
port (clk, init, val_init, d  : in std_ulogic;
      q                       : out std_ulogic
     );
end component;

component retard_2
port (clk, init, d : in std_ulogic;
      q            : out std_ulogic
     );
end component;

component retard_3
port (clk, init, d : in std_ulogic;
      q            : out std_ulogic
     );
end component;

component retard_4
port (clk, init, d : in std_ulogic;
      q            : out std_ulogic
     );
end component;

component PG_c_b
port( Pa, Ga, Pb, Gb : in  std_ulogic;
      Ps, Gs         : out std_ulogic
    );
end component;
```

```
component PG_b_c
port( Pa, Ga, Pb, Gb : in  std_ulogic ;
      Ps, Gs         : out std_ulogic
    ) ;
end component ;

component Piib
port( a, b : in  std_ulogic ;
      s    : out std_ulogic
    ) ;
end component ;

component Giib
port( a, b : in  std_ulogic ;
      s    : out std_ulogic
    ) ;
end component ;

component Giib_bis
port( a, b : in  std_ulogic ;
      s    : out std_ulogic
    ) ;
end component ;

end elem_composants ;
```

Le fichier `elem_fonctions.vhd` contient la description VHDL de composants élémentaires représentés, pour des raisons de clarté dans l'écriture des programmes, sous forme de fonctions. Ces composants sont utilisés par l'ensemble des blocs du coprocesssseur.

```
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use ieee.std_logic_unsigned.all ;

------------------    Package    ------------------
------------------ elem_fonctions -----------------

package elem_fonctions is

 function majorite (a,b,c : std_ulogic) return std_ulogic ;
 function modulo2 (a,b,c,d : std_ulogic) return std_ulogic ;
 function mux2 (a0,a1,c : std_ulogic) return std_ulogic ;
 function vect128 ( Vd : std_ulogic) return std_logic_vector ;
 function vect129 ( Vd : std_ulogic) return std_logic_vector ;

end elem_fonctions ;

package body elem_fonctions is

--------- Majorité à 3 entrées ----------

 function majorite (a,b,c : std_ulogic) return std_ulogic is
 begin
   return ((a and b) or (a and c) or (b and c)) ;
 end majorite ;


--------- Modulo 2 à 4 entrées ----------

 function modulo2 (a,b,c,d : std_ulogic) return std_ulogic is
 begin
   return ((a xor b) xor (c xor d)) ;
 end modulo2 ;


------- Multiplexeur à 2 entrées --------

 function mux2 (a0,a1,c : std_ulogic) return std_ulogic is
 begin
   if c='1' then return a1 ;
   else return a0 ;
   end if ;
 end mux2 ;
```

```
------- Init. vecteur de 128 bits -------

 function vect128 ( Vd : std_ulogic) return std_logic_vector is
  variable vect : std_ulogic_vector(127 downto 0);
 begin
   vect(127 downto 0) := (others => Vd);
   return to_stdlogicvector(vect);
 end vect128;


------- Init. vecteur de 129 bits -------

 function vect129 ( Vd : std_ulogic) return std_logic_vector is
  variable vect : std_ulogic_vector(128 downto 0);
 begin
   vect(128 downto 0) := (others => Vd);
   return to_stdlogicvector(vect);
 end vect129;

end elem_fonctions;
```

Le fichier `elem_unités.vhd` contient la description VHDL de composants élémentaires utilisés par la partie contrôle des unités fonctionnelles.

```
--------- Incrémentation 4 bits ---------
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;

entity add_p1_5bits is
port(A3, A2, A1, A0         : in std_ulogic ;
     q4, q3, q2, q1, q0     : out std_ulogic
    ) ;
end add_p1_5bits ;

architecture arch_add_p1_5bits of add_p1_5bits is
  signal tmp10, tmp32, tmp20, tmp30    : std_ulogic ;

begin
 tmp10 <= A0 nand A1 ;
 tmp32 <= A3 nand A2 ;
 tmp20 <= tmp30 nor tmp10 ;
 tmp30 <= not(A2) ;
 q0 <= not(A0) ;
 q1 <= A1 xor A0 ;
 q2 <= A2 xor not(tmp10) ;
 q3 <= A3 xor tmp20 ;
 q4 <= tmp10 nor tmp32 ;
end arch_add_p1_5bits ;

---------- Soustracteur 4 bits ----------
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.elem_fonctions.all ;
use work.elem_composants.all ;

entity soust_4bits is
port(A3, B3, A2, B2, A1, B1, A0, B0  : in std_ulogic ;
     C5                              : out std_ulogic
    ) ;
end soust_4bits ;

architecture arch_soust_4bits of soust_4bits is

signal B4b, B3b, B2b, B1b, B0b,
       Lii_P3b, Lii_P2b, Lii_P1b, Lii_P0b, Lii_P0,
       Lii_G3b, Lii_G2b, Lii_G1b, Lii_G0b, Lii_G0,
       L1_P1, L1_P3, L1_G1, L1_G3, L2_P3b, L2_G3b    : std_ulogic ;
```

```
begin
 B3b <= not(B3) ;
 B2b <= not(B2) ;
 B1b <= not(B1) ;
 B0b <= not(B0) ;

 Lii_P_3 : Piib
  port map(A3, B3b, Lii_P3b) ;
 Lii_P_2 : Piib
  port map(A2, B2b, Lii_P2b) ;
 Lii_P_1 : Piib
  port map(A1, B1b, Lii_P1b) ;
 Lii_P_0 : Piib
  port map(A0, B0b, Lii_P0b) ;

 Lii_G_3 : Giib
  port map(A3, B3b, Lii_G3b) ;
 Lii_G_2 : Giib
  port map(A2, B2b, Lii_G2b) ;
 Lii_G_1 : Giib
  port map(A1, B1b, Lii_G1b) ;
 Lii_G_0 : Giib
  port map(A0, B0b, Lii_G0b) ;

 L1_1 : PG_b_c
  port map(Lii_P1b, Lii_G1b, Lii_P0b, Lii_G0b, L1_P1, L1_G1) ;
 L1_3 : PG_b_c
  port map(Lii_P3b, Lii_G3b, Lii_P2b, Lii_G2b, L1_P3, L1_G3) ;

 L2_3 : PG_c_b
  port map(L1_P3, L1_G3, L1_P1, L1_G1, L2_P3b, L2_G3b) ;
 C5 <= not(L2_G3b) ;
end arch_soust_4bits ;

---------  Registre à décalage  ---------
--------- 4 bits avec INIT=0001 ---------
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.elem_composants.all ;

entity compt_0to3 is
port (init, clk            : in std_ulogic ;
      s0, s1, s2, s3       : out std_ulogic
     ) ;
end compt_0to3 ;

architecture arch_compt_0to3 of compt_0to3 is
  signal initb, tmp0, tmp1, tmp2, tmp3 : std_ulogic ;
```

```
begin
 initb <= not(init);
 reg0 : registreSET
  port map(clk, initb, tmp3, tmp0);
 reg1 : registreRESET
  port map(clk, initb, tmp0, tmp1);
 reg2 : registreRESET
  port map(clk, initb, tmp1, tmp2);
 reg3 : registreRESET
  port map(clk, initb, tmp2, tmp3);
 s0 <= tmp0;
 s1 <= tmp1;
 s2 <= tmp2;
 s3 <= tmp3;
end  arch_compt_0to3;

----------- Compteur 5 bits -------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.elem_composants.all;

entity compteur_5bits is
port(init, clk             : in std_ulogic;
     q0, q1, q2, q3, q4     : out std_ulogic
    );
end compteur_5bits;

architecture arch_compteur_5bits of compteur_5bits is
  signal initb, tmp0, tmp1, tmp2, tmp3, tmp4,
         tmp0b, tmp1b, tmp2b, tmp3b, tmp4b     : std_ulogic;

begin
 initb <= not(init);
 tmp0b <= not(tmp0);
 tmp1b <= not(tmp1);
 tmp2b <= not(tmp2);
 tmp3b <= not(tmp3);
 tmp4b <= not(tmp4);
 reg0 :registreSET
  port map(clk , initb, tmp0b, tmp0);
 reg1 :registreSET
  port map(tmp0, initb, tmp1b, tmp1);
 reg2 :registreSET
  port map(tmp1, initb, tmp2b, tmp2);
 reg3 :registreSET
  port map(tmp2, initb, tmp3b, tmp3);
```

```
 reg4 :registreSET
  port map(tmp3, initb, tmp4b, tmp4);
 q0 <= tmp0b;
 q1 <= tmp1b;
 q2 <= tmp2b;
 q3 <= tmp3b;
 q4 <= tmp4b;
end arch_compteur_5bits;

----------- Compteur 5 bits -------------
-----------  avec INIT=0011 -------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.elem_composants.all;

entity compteur_5bits_init3 is
port(init, clk             : in std_ulogic;
     q0, q1, q2, q3, q4     : out std_ulogic
    );
end compteur_5bits_init3;

architecture arch_compteur_5bits_init3 of compteur_5bits_init3 is
  signal initb, tmp0, tmp1, tmp2, tmp3, tmp4,
         tmp0b, tmp1b, tmp2b, tmp3b, tmp4b     : std_ulogic;

begin
 initb <= not(init);
 tmp0b <= not(tmp0);
 tmp1b <= not(tmp1);
 tmp2b <= not(tmp2);
 tmp3b <= not(tmp3);
 tmp4b <= not(tmp4);
 reg0 :registreRESET
  port map(clk , initb, tmp0b, tmp0);
 reg1 :registreRESET
  port map(tmp0, initb, tmp1b, tmp1);
 reg2 :registreSET
  port map(tmp1, initb, tmp2b, tmp2);
 reg3 :registreSET
  port map(tmp2, initb, tmp3b, tmp3);
 reg4 :registreSET
  port map(tmp3, initb, tmp4b, tmp4);
 q0 <= tmp0b;
 q1 <= tmp1b;
 q2 <= tmp2b;
 q3 <= tmp3b;
 q4 <= tmp4b;
end arch_compteur_5bits_init3;
```

```
----------- Compteur 7 bits -------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.elem_composants.all;

entity compteur_7bits is
port(init, clk                       : in std_ulogic;
     q0, q1, q2, q3, q4, q5, q6      : out std_ulogic
    );
end compteur_7bits;

architecture arch_compteur_7bits of compteur_7bits is
  signal initb, tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6,
         tmp0b, tmp1b, tmp2b, tmp3b, tmp4b, tmp5b, tmp6b     : std_ulogic;

begin
 initb <= not(init);
 tmp0b <= not(tmp0);
 tmp1b <= not(tmp1);
 tmp2b <= not(tmp2);
 tmp3b <= not(tmp3);
 tmp4b <= not(tmp4);
 tmp5b <= not(tmp5);
 tmp6b <= not(tmp6);
 reg0 :registreSET
  port map(clk , initb, tmp0b, tmp0);
 reg1 :registreSET
  port map(tmp0, initb, tmp1b, tmp1);
 reg2 :registreSET
  port map(tmp1, initb, tmp2b, tmp2);
 reg3 :registreSET
  port map(tmp2, initb, tmp3b, tmp3);
 reg4 :registreSET
  port map(tmp3, initb, tmp4b, tmp4);
 reg5 :registreSET
  port map(tmp4, initb, tmp5b, tmp5);
 reg6 :registreSET
  port map(tmp5, initb, tmp6b, tmp6);
 q0 <= tmp0b;
 q1 <= tmp1b;
 q2 <= tmp2b;
 q3 <= tmp3b;
 q4 <= tmp4b;
 q5 <= tmp5b;
 q6 <= tmp6b;
end arch_compteur_7bits;
```

```
----------- Compteur 7 bits -------------
-----------  avec INIT=0010 -------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.elem_composants.all;

entity compteur_7bits_init2 is
port(init, clk                      : in std_ulogic;
     q0, q1, q2, q3, q4, q5, q6     : out std_ulogic
    );
end compteur_7bits_init2;

architecture arch_compteur_7bits_init2 of compteur_7bits_init2 is
  signal initb, tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6,
         tmp0b, tmp1b, tmp2b, tmp3b, tmp4b, tmp5b, tmp6b     : std_ulogic;

begin
 initb <= not(init);
 tmp0b <= not(tmp0);
 tmp1b <= not(tmp1);
 tmp2b <= not(tmp2);
 tmp3b <= not(tmp3);
 tmp4b <= not(tmp4);
 tmp5b <= not(tmp5);
 tmp6b <= not(tmp6);
 reg0 :registreSET
  port map(clk , initb, tmp0b, tmp0);
 reg1 :registreRESET
  port map(tmp0, initb, tmp1b, tmp1);
 reg2 :registreSET
  port map(tmp1, initb, tmp2b, tmp2);
 reg3 :registreSET
  port map(tmp2, initb, tmp3b, tmp3);
 reg4 :registreSET
  port map(tmp3, initb, tmp4b, tmp4);
 reg5 :registreSET
  port map(tmp4, initb, tmp5b, tmp5);
 reg6 :registreSET
  port map(tmp5, initb, tmp6b, tmp6);
 q0 <= tmp0b;
 q1 <= tmp1b;
 q2 <= tmp2b;
 q3 <= tmp3b;
 q4 <= tmp4b;
 q5 <= tmp5b;
 q6 <= tmp6b;
end arch_compteur_7bits_init2;
```

```
----------- Compteur 7 bits -------------
-----------  avec INIT=0110 -------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.elem_composants.all;

entity compteur_7bits_init6 is
port(init, clk                     : in std_ulogic;
     q0, q1, q2, q3, q4, q5, q6    : out std_ulogic
    );
end compteur_7bits_init6;

architecture arch_compteur_7bits_init6 of compteur_7bits_init6 is
  signal initb, tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6,
         tmp0b, tmp1b, tmp2b, tmp3b, tmp4b, tmp5b, tmp6b     : std_ulogic;

begin
 initb <= not(init);
 tmp0b <= not(tmp0);
 tmp1b <= not(tmp1);
 tmp2b <= not(tmp2);
 tmp3b <= not(tmp3);
 tmp4b <= not(tmp4);
 tmp5b <= not(tmp5);
 tmp6b <= not(tmp6);
 reg0 :registreSET
  port map(clk , initb, tmp0b, tmp0);
 reg1 :registreRESET
  port map(tmp0, initb, tmp1b, tmp1);
 reg2 :registreRESET
  port map(tmp1, initb, tmp2b, tmp2);
 reg3 :registreSET
  port map(tmp2, initb, tmp3b, tmp3);
 reg4 :registreSET
  port map(tmp3, initb, tmp4b, tmp4);
 reg5 :registreSET
  port map(tmp4, initb, tmp5b, tmp5);
 reg6 :registreSET
  port map(tmp5, initb, tmp6b, tmp6);
 q0 <= tmp0b;
 q1 <= tmp1b;
 q2 <= tmp2b;
 q3 <= tmp3b;
 q4 <= tmp4b;
 q5 <= tmp5b;
 q6 <= tmp6b;
end arch_compteur_7bits_init6;
```

```
----------- Compteur 8 bits -------------
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.elem_composants.all ;

entity compteur_8bits is
port(init, clk                       : in std_ulogic ;
     q0, q1, q2, q3, q4, q5, q6, q7  : out std_ulogic
    ) ;
end compteur_8bits ;

architecture arch_compteur_8bits of compteur_8bits is
  signal initb, tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7,
         tmp0b, tmp1b, tmp2b, tmp3b, tmp4b, tmp5b, tmp6b, tmp7b  : std_ulogic ;

begin
 initb <= not(init) ;
 tmp0b <= not(tmp0) ;
 tmp1b <= not(tmp1) ;
 tmp2b <= not(tmp2) ;
 tmp3b <= not(tmp3) ;
 tmp4b <= not(tmp4) ;
 tmp5b <= not(tmp5) ;
 tmp6b <= not(tmp6) ;
 tmp7b <= not(tmp7) ;
 reg0 :registreSET
  port map(clk , initb, tmp0b, tmp0) ;
 reg1 :registreSET
  port map(tmp0, initb, tmp1b, tmp1) ;
 reg2 :registreSET
  port map(tmp1, initb, tmp2b, tmp2) ;
 reg3 :registreSET
  port map(tmp2, initb, tmp3b, tmp3) ;
 reg4 :registreSET
  port map(tmp3, initb, tmp4b, tmp4) ;
 reg5 :registreSET
  port map(tmp4, initb, tmp5b, tmp5) ;
 reg6 :registreSET
  port map(tmp5, initb, tmp6b, tmp6) ;
 reg7 :registreSET
  port map(tmp6, initb, tmp7b, tmp7) ;
 q0 <= tmp0b ;
 q1 <= tmp1b ;
 q2 <= tmp2b ;
 q3 <= tmp3b ;
 q4 <= tmp4b ;
 q5 <= tmp5b ;
 q6 <= tmp6b ;
```

```
 q7 <= tmp7b ;
end arch_compteur_8bits ;

---------- Décompteur 4 bits ------------
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.elem_composants.all ;
entity decompteur_4bits is
port(lg_A                   : in std_ulogic_vector(3 downto 0) ;
     init, clk              : in std_ulogic ;
     q0, q1, q2, q3         : out std_ulogic
    ) ;
end decompteur_4bits ;

architecture arch_decompteur_4bits of decompteur_4bits is
  signal initb, tmp0, tmp1, tmp2, tmp3,
         tmp0b, tmp1b, tmp2b, tmp3b        : std_ulogic ;

begin
 initb <= not(init) ;
 reg0 :registreSET
  port map(clk , initb, tmp0b, tmp0) ;
 reg1 :registreSET
  port map(tmp0, initb, tmp1b, tmp1) ;
 reg2 :registreSET
  port map(tmp1, initb, tmp2b, tmp2) ;
 reg3 :registreSET
  port map(tmp2, initb, tmp3b, tmp3) ;
 q0 <= tmp0 ;
 q1 <= tmp1 ;
 q2 <= tmp2 ;
 q3 <= tmp3 ;
end arch_decompteur_4bits ;

--------- Additionneur 32 bits ----------
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.elem_fonctions.all ;
use work.elem_composants.all ;

entity add_32bits is
 port(tmpF, tmpG        : in  std_ulogic_vector(31 downto 0) ;
      Cin_HC            : in  std_ulogic ;
      tmp_S             : out std_ulogic_vector(31 downto 0) ;
      Cout              : out std_ulogic
      ) ;
end add_32bits ;
```

```
architecture arch_add_32bits of add_32bits is
  signal tmpP_Lb, tmpG_Lb       : std_ulogic_vector(31 downto 0);
  signal tmpP_L , tmpG_L        : std_ulogic_vector(31 downto 0);
  signal tmpP_L0, tmpG_L0       : std_ulogic_vector(31 downto 0);
  signal tmpP_L1, tmpG_L1       : std_ulogic_vector(31 downto 0);
  signal tmpP_L2, tmpG_L2       : std_ulogic_vector(31 downto 0);
  signal tmpP_L3, tmpG_L3       : std_ulogic_vector(31 downto 0);
  signal tmpP_L4, tmpG_L4       : std_ulogic_vector(31 downto 0);
  signal tmpP_L5, tmpG_L5       : std_ulogic_vector(31 downto 0);
  signal tmp_Cout               : std_ulogic_vector(32 downto 1);

begin
  Liib : for i in 31 downto 0 generate
         instPiib_n : Piib
          port map(tmpF(i), tmpG(i), tmpP_Lb(i));
         instGiib_n : Giib
          port map(tmpF(i), tmpG(i), tmpG_Lb(i));
        end generate Liib;
  Lii : for i in 31 downto 0 generate
        tmpP_L(i) <= not tmpP_Lb(i);
        tmpG_L(i) <= not tmpG_Lb(i);
       end generate Lii;

  ----------------------------------
  L0 : for i in 15 downto 0 generate
       instPG_n : PG_b_c
        port map(tmpP_Lb(2*i+1), tmpG_Lb(2*i+1),
                 tmpP_Lb(2*i),   tmpG_Lb(2*i),
                 tmpP_L0(2*i+1),tmpG_L0(2*i+1)
                );
      end generate L0;

  ----------------------------------
  L1 : for i in 15 downto 1 generate
       instPG_n : PG_c_b
        port map(tmpP_L0(2*i+1), tmpG_L0(2*i+1),
                 tmpP_L0(2*i-1), tmpG_L0(2*i-1),
                 tmpP_L1(2*i+1), tmpG_L1(2*i+1)
                );
      end generate L1;
  tmpP_L1(1) <= not(tmpP_L0(1));
  tmpG_L1(1) <= not(tmpG_L0(1));

  ----------------------------------
  L2 : for i in 15 downto 2 generate
       instPG_n : PG_b_c
        port map(tmpP_L1(2*i+1), tmpG_L1(2*i+1),
                 tmpP_L1(2*i-3), tmpG_L1(2*i-3),
```

```
               tmpP_L2(2*i+1), tmpG_L2(2*i+1)
              ) ;
    end generate L2 ;
L2_vide : for i in 1 downto 0 generate
          tmpP_L2(2*i+1) <= not(tmpP_L1(2*i+1)) ;
          tmpG_L2(2*i+1) <= not(tmpG_L1(2*i+1)) ;
         end generate L2_vide ;

----------------------------------
L3 : for i in 15 downto 4 generate
     instPG_n : PG_c_b
      port map(tmpP_L2(2*i+1), tmpG_L2(2*i+1),
               tmpP_L2(2*i-7), tmpG_L2(2*i-7),
               tmpP_L3(2*i+1), tmpG_L3(2*i+1)
              ) ;
    end generate L3 ;
L3_vide : for i in 3 downto 0 generate
          tmpP_L3(2*i+1) <= not(tmpP_L2(2*i+1)) ;
          tmpG_L3(2*i+1) <= not(tmpG_L2(2*i+1)) ;
         end generate L3_vide ;

----------------------------------
L4 : for i in 15 downto 8 generate
     instPG_n : PG_b_c
      port map(tmpP_L3(2*i+1), tmpG_L3(2*i+1),
               tmpP_L3(2*i-15), tmpG_L3(2*i-15),
               tmpP_L4(2*i+1), tmpG_L4(2*i+1)
              ) ;
    end generate L4 ;
L4_vide : for i in 7 downto 0 generate
          tmpP_L4(2*i+1) <= not(tmpP_L3(2*i+1)) ;
          tmpG_L4(2*i+1) <= not(tmpG_L3(2*i+1)) ;
         end generate L4_vide ;

----------------------------------
L5_pair : for i in 15 downto 1 generate
          instPG_n : PG_c_b
           port map(tmpP_L(2*i), tmpG_L(2*i),
                    tmpP_L4(2*i-1), tmpG_L4(2*i-1),
                    tmpP_L5(2*i), tmpG_L5(2*i)
                   ) ;
         end generate L5_pair ;
L5_impair : for i in 15 downto 0 generate
            tmpP_L5(2*i+1) <= not(tmpP_L4(2*i+1)) ;
            tmpG_L5(2*i+1) <= not(tmpG_L4(2*i+1)) ;
           end generate L5_impair ;
tmpP_L5(0) <= tmpP_Lb(0) ;
tmpG_L5(0) <= tmpG_Lb(0) ;
```

```
  ----------------------------------
  Lcout : for i in 31 downto 0 generate
          tmp_Cout(i+1) <= ( not(tmpP_L5(i)) and Cin_HC ) or not(tmpG_L5(i)) ;
         end generate Lcout ;
  tmp_S(0) <= tmpP_L(0) xor Cin_HC ;
  Lfin : for i in 31 downto 1 generate
             tmp_S(i)   <= tmpP_L(i) xor tmp_Cout(i) ;
            end generate Lfin ;
  Cout <= tmp_Cout(32) ;
end arch_add_32bits ;

-------------------   Package   -------------------
------------------- elem_unites -------------------

library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;

package elem_unites is

component add_p1_5bits
port(A3, A2, A1, A0         : in std_ulogic ;
     q4, q3, q2, q1, q0     : out std_ulogic
    ) ;
end component ;

component soust_4bits
port(A3, B3, A2, B2, A1, B1, A0, B0  : in std_ulogic ;
     C5                              : out std_ulogic
    ) ;
end component ;

component compt_0to3
port (init, clk             : in std_ulogic ;
      s0, s1, s2, s3        : out std_ulogic
     ) ;
end component ;

component compteur_5bits
port(init, clk              : in std_ulogic ;
     q0, q1, q2, q3, q4     : out std_ulogic
    ) ;
end component ;

component compteur_5bits_init3
port(init, clk              : in std_ulogic ;
     q0, q1, q2, q3, q4     : out std_ulogic
    ) ;
end component ;
```

```
component compteur_7bits
port(init, clk                      : in std_ulogic;
     q0, q1, q2, q3, q4, q5, q6     : out std_ulogic
    );
end component;

component compteur_7bits_init2
port(init, clk                      : in std_ulogic;
     q0, q1, q2, q3, q4, q5, q6     : out std_ulogic
    );
end component;

component compteur_7bits_init6
port(init, clk                      : in std_ulogic;
     q0, q1, q2, q3, q4, q5, q6     : out std_ulogic
    );
end component;

component compteur_8bits
port(init, clk                       : in std_ulogic;
     q0, q1, q2, q3, q4, q5, q6, q7  : out std_ulogic
    );
end component;

component decompteur_4bits
port(lg_A                  : in std_ulogic_vector(3 downto 0);
     init, clk             : in std_ulogic;
     q0, q1, q2, q3        : out std_ulogic
    );
end component;

component add_32bits
 port(tmpF, tmpG        : in  std_ulogic_vector(31 downto 0);
      Cin_HC            : in  std_ulogic;
      tmp_S             : out std_ulogic_vector(31 downto 0);
      Cout              : out std_ulogic
      );
end component;

end elem_unites;
```

Le fichier `add_fct.vhd` contient la description VHDL de composants élémentaires utilisés par l' unité d'addition.

```
----------- Compteur 4 bits -------------
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.elem_composants.all ;

entity compteur_4bits is
port(init, clk              : in std_ulogic ;
     q0, q1, q2, q3         : out std_ulogic
    ) ;
end compteur_4bits ;

architecture arch_compteur_4bits of compteur_4bits is
  signal initb, tmp0, tmp1, tmp2, tmp3,
         tmp0b, tmp1b, tmp2b, tmp3b       : std_ulogic ;
begin
 initb <= not(init) ;
 tmp0b <= not(tmp0) ;
 tmp1b <= not(tmp1) ;
 tmp2b <= not(tmp2) ;
 tmp3b <= not(tmp3) ;
 reg0 :registreSET
  port map(clk , initb, tmp0b, tmp0) ;
 reg1 :registreSET
  port map(tmp0, initb, tmp1b, tmp1) ;
 reg2 :registreSET
  port map(tmp1, initb, tmp2b, tmp2) ;
 reg3 :registreSET
  port map(tmp2, initb, tmp3b, tmp3) ;
 q0 <= tmp0b ;
 q1 <= tmp1b ;
 q2 <= tmp2b ;
 q3 <= tmp3b ;
end arch_compteur_4bits ;

----------- Compteur 5 bits -------------
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.elem_composants.all ;

entity compteur_5bits_init1 is
port(init, clk              : in std_ulogic ;
     q0, q1, q2, q3, q4     : out std_ulogic
    ) ;
end compteur_5bits_init1 ;
```

```
architecture arch_compteur_5bits_init1 of compteur_5bits_init1 is
  signal initb, tmp0, tmp1, tmp2, tmp3, tmp4,
         tmp0b, tmp1b, tmp2b, tmp3b, tmp4b     : std_ulogic;
begin
 initb <= not(init);
 tmp0b <= not(tmp0);
 tmp1b <= not(tmp1);
 tmp2b <= not(tmp2);
 tmp3b <= not(tmp3);
 tmp4b <= not(tmp4);
 reg0 :registreRESET
  port map(clk , initb, tmp0b, tmp0);
 reg1 :registreSET
  port map(tmp0, initb, tmp1b, tmp1);
 reg2 :registreSET
  port map(tmp1, initb, tmp2b, tmp2);
 reg3 :registreSET
  port map(tmp2, initb, tmp3b, tmp3);
 reg4 :registreSET
  port map(tmp3, initb, tmp4b, tmp4);
 q0 <= tmp0b;
 q1 <= tmp1b;
 q2 <= tmp2b;
 q3 <= tmp3b;
 q4 <= tmp4b;
end arch_compteur_5bits_init1;

------------------    Package     -----------------
------------------ add_composants -----------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

package add_composants is

component compteur_4bits
port(init, clk              : in std_ulogic;
     q0, q1, q2, q3         : out std_ulogic
    );
end component;

component compteur_5bits_init1
port(init, clk              : in std_ulogic;
     q0, q1, q2, q3, q4     : out std_ulogic
    );
end component;
end add_composants;
```

Le fichier `add_op.vhd` contient la description VHDL de l'opérateur d'addition/soustraction contenu dans l' unité d'addition. Cet opérateur correspond aux 2ème et 3ème étages du pipeline de cette unité.

```
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.elem_fonctions.all ;
use work.elem_composants.all ;
use work.add_composants.all ;

entity additionneur is
 port(pin_data_A, pin_data_B     : in  std_ulogic_vector(127 downto 0) ;
      pin_Cin2, pin_Cin1         : in  std_ulogic ;
      pin_add_sous               : in  std_ulogic ;
      pin_comparaison            : in  std_ulogic ;
      pin_NS_A, pin_NS_B         : in  std_ulogic ;
      pin_D_A, pin_D_B           : in  std_ulogic ;
      pin_Vd_A, pin_Vd_B         : in  std_ulogic ;
      pin_NF_A, pin_NF_B         : in  std_ulogic ;
      pin_CT1, pin_CT2, pin_CT3  : in  std_ulogic ;
      clk                        : in  std_ulogic ;
      pin_init                   : in  std_ulogic ;
      pin_init_S                 : in  std_ulogic ;
      pout_S                     : out std_ulogic_vector(127 downto 0) ;
      pout_NS_S                  : out std_ulogic ;
      pout_Cout2, pout_Cout1     : out std_ulogic ;
      pout_Vd_S                  : out std_ulogic ;
      pout_Dd_S                  : out std_ulogic ;
      pout_long_S                : out std_ulogic_vector(3 downto 0) ;
      pout_comparaison           : out std_ulogic ;
      pout_result_comp           : out std_ulogic
    ) ;
end additionneur ;

architecture arch_add of additionneur is
  signal in_A, in_B                     : std_ulogic_vector(127 downto 0) ;
  signal tmpA1_p, tmpB1_p               : std_ulogic_vector(127 downto 0) ;
  signal tmpA1_m, tmpB1_m               : std_ulogic_vector(127 downto 0) ;
  signal tmpF, tmpG, tmpH               : std_ulogic_vector(127 downto 0) ;
  signal tmpP_Lb, tmpG_Lb, tmpG_Lb_bis  : std_ulogic_vector(128 downto 0) ;
  signal tmpP_L, tmpG_L                 : std_ulogic_vector(128 downto 0) ;
  signal tmpP_L0, tmpG_L0               : std_ulogic_vector(127 downto 0) ;
  signal tmpP_L1, tmpG_L1               : std_ulogic_vector(127 downto 0) ;
  signal tmpP_L2, tmpG_L2               : std_ulogic_vector(127 downto 0) ;
  signal tmpP_L3, tmpG_L3               : std_ulogic_vector(127 downto 0) ;
  signal tmpP_L4, tmpG_L4               : std_ulogic_vector(127 downto 0) ;
  signal tmpP_L5, tmpG_L5               : std_ulogic_vector(127 downto 0) ;
  signal tmpP_L6, tmpG_L6               : std_ulogic_vector(127 downto 0) ;
```

```
  signal tmpP_L7, tmpG_L7            : std_ulogic_vector(128 downto 0);
  signal tmp_Coutb                   : std_ulogic_vector(128 downto 0);
  signal tmpP_reg, tmpG_reg          : std_ulogic_vector(128 downto 0);
  signal tmpPiib_reg, tmpGiib_reg    : std_ulogic_vector(128 downto 0);
  signal tmp_S                       : std_ulogic_vector(127 downto 0);

  signal d1_reg                      : std_ulogic_vector(63 downto 0);
  signal d1_S1                       : std_ulogic_vector(127 downto 0);
  signal d1_S1_bis                   : std_ulogic_vector(127 downto 0);
  signal d1_S2                       : std_ulogic_vector(63 downto 0);
  signal d1_S2_bis                   : std_ulogic_vector(63 downto 0);
  signal d1_mux                      : std_ulogic_vector(63 downto 0);
  signal d1_S3                       : std_ulogic_vector(31 downto 0);
  signal d1_S4                       : std_ulogic_vector(15 downto 0);
  signal d1_S5                       : std_ulogic_vector(7 downto 0);
  signal d1_S6                       : std_ulogic_vector(3 downto 0);
  signal d1_S7                       : std_ulogic_vector(1 downto 0);

  signal Cpt_p1                      : std_ulogic_vector(4 downto 0);

  signal longueur, Cpt,
         in_reg_NS, out_reg_NS,
         in_reg_S, out_reg_S,
         out_mux_cpt                 : std_ulogic_vector(3 downto 0);

  signal Gnd, Vss, add_sous,
         NS_A, NS_B, NF_A, NF_B, Vd_A, Vd_B, D_A, D_B,
         Cin1, Cin2, Cin_p_reg, Cin_m_reg, Cin_p, Cin_m,
         Cin_HC_inputreg, Cin_HC,
         Cout_p_fin, Cout_m_fin, tmp_Cout1, tmp_Cout2,
         init1, init2, comp1, comp2, tmpG127b,
         CT1, CT1_reg, CT2, CT2_reg, CT3, CT3_reg,
         Signe_S, Signe_S_reg, tmp_Ddb, cmde_NS, Cmde_S,
         d1_S8, d0, d1, EGAL, SUP, s127_ret,
         tmp_NS_S, tmp_Dd_S, tmp_Vd_S,
         tmp_d0_S, tmp_d1_S, tmp_result_comp           : std_ulogic;

begin

  Gnd <= '0';
  Vss <= '1';

  -------------------------------------------------------------
  ---                 REGISTRES D'ENTREE                    ---
  -------------------------------------------------------------

  reg_init1 : registre
   port map(clk, pin_init, init1);
```

```
reg_in_A : for i in 0 to 127 generate
           reg : registre
            port map(clk, pin_data_A(i), in_A(i)) ;
end generate reg_in_A ;

reg_in_B : for i in 0 to 127 generate
           reg : registre
            port map(clk, pin_data_B(i), in_B(i)) ;
end generate reg_in_B ;

reg_add_sous : registre
 port map(clk, pin_add_sous, add_sous) ;

reg_comp1 : registre
 port map(clk, pin_comparaison, comp1) ;

reg_NS_A : registre
 port map(clk, pin_NS_A, NS_A) ;

reg_NS_B : registre
 port map(clk, pin_NS_B, NS_B) ;

reg_D_A : registre
 port map(clk, pin_D_A, D_A) ;

reg_D_B : registre
 port map(clk, pin_D_B, D_B) ;

reg_Vd_A : registre
 port map(clk, pin_Vd_A, Vd_A) ;

reg_Vd_B : registre
 port map(clk, pin_Vd_B, Vd_B) ;

reg_NF_A : registre
 port map(clk, pin_NF_A, NF_A) ;

reg_NF_B : registre
 port map(clk, pin_NF_B, NF_B) ;

reg_CT1 : registre
 port map(clk, pin_CT1, CT1) ;

reg_CT2 : registre
 port map(clk, pin_CT2, CT2) ;

reg_CT3 : registre
 port map(clk, pin_CT3, CT3) ;
```

```
reg_Cin1 : registre
 port map(clk, pin_Cin1, Cin1);

reg_Cin2 : registre
 port map(clk, pin_Cin2, Cin2);

reg_Cin_p_reg : registre
 port map(clk, tmpH(127), Cin_p_reg);

reg_Cin_m_reg : registre
 port map(clk, tmpG(127), Cin_m_reg);

-------------------------------------------------------------
-------------------------------------------------------------
---       ++++ ETAGE 1 DE L'OPERATEUR PIPELINE ++++       ---
-------------------------------------------------------------
-------------------------------------------------------------

---------- Retenues entrantes -----------

Cin_p <= mux2(mux2(Cin1 xor Cin2, Gnd, add_sous),
              Cin_p_reg,
              init1);
Cin_m <= mux2(not(mux2(Gnd, Cin1 xor Cin2, add_sous)),
              Cin_m_reg,
              init1);

-----   Mise en forme des entrées   -----
-----    ATTENTION, LES ENTREES     -----
----- NEGATIVES (_m) SONT INVERSEES -----

tmpA1_p(0) <= (NF_A nor D_A) and in_A(0);
tmpA1_m(0) <= not( not(NF_A) and D_A and Vd_A);
tmpB1_p(0) <= not(add_sous xor ((NF_B nor D_B) nand in_B(0)));
tmpB1_m(0) <= add_sous xor not( not(NF_B) and D_B and Vd_B);
in_logique : for i in 126 downto 1 generate
   tmpA1_p(i) <= (NF_A nor D_A) and in_A(i);
   tmpA1_m(i) <= Vss;
   tmpB1_p(i) <= not(add_sous xor ((NF_B nor D_B) nand in_B(i)));
   tmpB1_m(i) <= not(add_sous);
end generate in_logique;
tmpA1_p(127) <= not(NF_A or NS_A or D_A) and in_A(127);
tmpA1_m(127) <= not((NF_A nor D_A) and NS_A and in_A(127));
tmpB1_p(127) <= add_sous xor (not(NF_B or NS_B or D_B) and in_B(127));
tmpB1_m(127) <= add_sous xor not((NF_B nor D_B) and NS_B and in_B(127));

-------- Additionneur redondant ---------

in_red : for i in 127 downto 1 generate
```

```
         inst42_n : add42
          port map(tmpA1_p(i), tmpA1_m(i),
                   tmpB1_p(i), tmpB1_m(i),
                   tmpH(i-1),
                   tmpF(i), tmpG(i), tmpH(i)
                  );
        end generate in_red;
inst42_0 : add42
 port map(tmpA1_p(0), tmpA1_m(0), tmpB1_p(0), tmpB1_m(0), Cin_p,
          tmpF(0), tmpG(0), tmpH(0)
         );

----- Soustracteur de Han & Carlson -----
------- Génération des Pii et Gii -------

instPiib_0 : Piib
 port map(tmpF(0), Cin_m, tmpP_Lb(0));
instGiib_0 : Giib
 port map(tmpF(0), Cin_m, tmpG_Lb(0));
instGiib_bis_0 : Giib_bis
 port map(tmpF(0), Cin_m, tmpG_Lb_bis(0));

Liib : for i in 127 downto 1 generate
       instPiib_n : Piib
        port map(tmpF(i), tmpG(i-1), tmpP_Lb(i));
       instGiib_n : Giib
        port map(tmpF(i), tmpG(i-1), tmpG_Lb(i));
       instGiib_n_bis : Giib_bis
        port map(tmpF(i), tmpG(i-1), tmpG_Lb_bis(i));
      end generate Liib;

instPiib_128 : Piib
 port map(tmpH(127), tmpG(127), tmpP_Lb(128));
instGiib_128 : Giib
 port map(tmpH(127), tmpG(127), tmpG_Lb(128));
instGiib_128_bis : Giib_bis
 port map(tmpH(127), tmpG(127), tmpG_Lb_bis(128));

Lii : for i in 64 downto 0 generate
      tmpP_L(2*i) <= not tmpP_Lb(2*i);
      tmpG_L(2*i) <= not tmpG_Lb(2*i);
     end generate Lii;

------- 1er étage de cellules PG --------

L0 : for i in 63 downto 0 generate
     instPG_n : PG_b_c
      port map(tmpP_Lb(2*i+1), tmpG_Lb(2*i+1),
               tmpP_Lb(2*i),   tmpG_Lb(2*i),
```

```
              tmpP_L0(2*i+1),tmpG_L0(2*i+1)
             ) ;
    end generate L0 ;

------- 2ème étage de cellules PG -------

L1 : for i in 63 downto 1 generate
     instPG_n : PG_c_b
      port map(tmpP_L0(2*i+1), tmpG_L0(2*i+1),
               tmpP_L0(2*i-1), tmpG_L0(2*i-1),
               tmpP_L1(2*i+1), tmpG_L1(2*i+1)
             ) ;
    end generate L1 ;

tmpP_L1(1) <= not(tmpP_L0(1)) ;
tmpG_L1(1) <= not(tmpG_L0(1)) ;

---------- Signe du résultat ------------

signe_S <= (NS_A and NS_B) or (NS_A and D_B) or (NS_A and NF_B) or
           ( D_A and NS_B) or ( D_A and D_B) or ( D_A and NF_B) or
           (NF_A and NS_B) or (NF_A and D_B) or (NF_A and NF_B) ;

----- Retenue sortante soustracteur -----

Cin_HC_inputreg <= not(mux2(Cin2, tmp_Coutb(128), init1)) ;

----- Génération des types 1, 2 et 3 ----

d1_S1(0)     <= not(tmpP_Lb(0)) ;
d1_S1_bis(0) <= not(tmpP_Lb(0)) ;

d1_1 : for i in 127 downto 1 generate
       d1_S1(i)     <= not(tmpP_Lb(i) xor tmpG_Lb(i-1)) ;
       d1_S1_bis(i) <= not(tmpP_Lb(i) xor tmpG_Lb_bis(i-1)) ;
      end generate d1_1 ;

d1_2 : for i in 63 downto 0 generate
       d1_S2(i)     <= d1_S1(2*i) nor d1_S1(2*i+1) ;
       d1_S2_bis(i) <= d1_S1_bis(2*i) nor d1_S1_bis(2*i+1) ;
      end generate d1_2 ;

d1_2_mux : for i in 63 downto 0 generate
           d1_mux(i) <= mux2(d1_S2_bis(i),d1_S2(i),Cin_HC_inputreg) ;
          end generate d1_2_mux ;

-------------------------------------------------------------
-- Registre dans la structure du soustracteur (codage CC2) --
-------------------------------------------------------------
```

```
REG_Pii_Gii : for i in 0 to 63 generate
              reg0 : registre
               port map(clk, tmpP_Lb(2*i+1), tmpPiib_reg(2*i+1)) ;
              reg1 : registre
               port map(clk, tmpG_Lb(2*i+1), tmpGiib_reg(2*i+1)) ;
end generate REG_Pii_Gii ;

REG_L_G : for i in 0 to 64 generate
          reg0 : registre
           port map(clk, tmpP_L(2*i), tmpP_reg(2*i)) ;
          reg1 : registre
           port map(clk, tmpG_L(2*i), tmpG_reg(2*i)) ;
end generate REG_L_G ;

REG_L1_G1 : for i in 0 to 63 generate
            reg0 : registre
             port map(clk, tmpP_L1(2*i+1), tmpP_reg(2*i+1)) ;
            reg1 : registre
             port map(clk, tmpG_L1(2*i+1), tmpG_reg(2*i+1)) ;
end generate REG_L1_G1 ;

-------------------------------------------------------------
--     Registre pour propager init1 dans le 2eme etage     --
-------------------------------------------------------------

reg_init2 : registre
 port map(clk, init1, init2) ;

-------------------------------------------------------------
-- Registre pour propager Cout_p/m a travers le 2eme etage --
-------------------------------------------------------------

reg_Cout_p_fin : registre
 port map(clk, tmpH(127), Cout_p_fin) ;

tmpG127b <= not(tmpG(127)) ;
reg_Cout_m_fin : registre
 port map(clk, tmpG127b, Cout_m_fin) ;

-------------------------------------------------------------
--  Registre pour propager CT1,CT2,CT3 dans le 2eme etage  --
-------------------------------------------------------------

reg_CT1_reg : registre
 port map(clk, CT1, CT1_reg) ;

reg_CT2_reg : registre
 port map(clk, CT2, CT2_reg) ;
```

```
reg_CT3_reg : registre
 port map(clk, CT3, CT3_reg) ;

-------------------------------------------------------------
--- Registre du pipe dans la determination du signe de S  ---
-------------------------------------------------------------

reg_Signe_S_reg : registre
 port map(clk, Signe_S, Signe_S_reg) ;

-------------------------------------------------------------
--         Registre pour la Cin du soust. de H&C           --
-------------------------------------------------------------

reg_Cin_HC_reg : registre
 port map(clk, Cin_HC_inputreg, Cin_HC) ;

-------------------------------------------------------------
---       Registre pour la detection d'un digit maxi      ---
-------------------------------------------------------------

reg_d1_reg : for i in 0 to 63 generate
             reg : registre
              port map(clk, d1_mux(i), d1_reg(i)) ;
end generate reg_d1_reg ;

-------------------------------------------------------------
---      Registre pour propager le signal comparaison     ---
-------------------------------------------------------------

reg_comp2 : registre
 port map(clk, comp1, comp2) ;

-------------------------------------------------------------
-------------------------------------------------------------
---       ++++ ETAGE 2 DE L'OPERATEUR PIPELINE ++++       ---
-------------------------------------------------------------
-------------------------------------------------------------

------- 3ème étage de cellules PG -------

L2 : for i in 63 downto 2 generate
     instPG_n : PG_b_c
      port map(tmpP_reg(2*i+1), tmpG_reg(2*i+1),
               tmpP_reg(2*i-3), tmpG_reg(2*i-3),
               tmpP_L2(2*i+1), tmpG_L2(2*i+1)
              ) ;
    end generate L2 ;
```

```
L2_vide : for i in 1 downto 0 generate
          tmpP_L2(2*i+1) <= not(tmpP_reg(2*i+1)) ;
          tmpG_L2(2*i+1) <= not(tmpG_reg(2*i+1)) ;
         end generate L2_vide ;

------- 4ème étage de cellules PG -------

L3 : for i in 63 downto 4 generate
     instPG_n : PG_c_b
      port map(tmpP_L2(2*i+1), tmpG_L2(2*i+1),
               tmpP_L2(2*i-7), tmpG_L2(2*i-7),
               tmpP_L3(2*i+1), tmpG_L3(2*i+1)
              ) ;
    end generate L3 ;
L3_vide : for i in 3 downto 0 generate
          tmpP_L3(2*i+1) <= not(tmpP_L2(2*i+1)) ;
          tmpG_L3(2*i+1) <= not(tmpG_L2(2*i+1)) ;
         end generate L3_vide ;

------- 5ème étage de cellules PG -------

L4 : for i in 63 downto 8 generate
     instPG_n : PG_b_c
      port map(tmpP_L3(2*i+1), tmpG_L3(2*i+1),
               tmpP_L3(2*i-15), tmpG_L3(2*i-15),
               tmpP_L4(2*i+1), tmpG_L4(2*i+1)
              ) ;
    end generate L4 ;
L4_vide : for i in 7 downto 0 generate
          tmpP_L4(2*i+1) <= not(tmpP_L3(2*i+1)) ;
          tmpG_L4(2*i+1) <= not(tmpG_L3(2*i+1)) ;
         end generate L4_vide ;

------- 6ème étage de cellules PG -------

L5 : for i in 63 downto 16 generate
     instPG_n : PG_c_b
      port map(tmpP_L4(2*i+1), tmpG_L4(2*i+1),
               tmpP_L4(2*i-31), tmpG_L4(2*i-31),
               tmpP_L5(2*i+1), tmpG_L5(2*i+1)
              ) ;
    end generate L5 ;
L5_vide : for i in 15 downto 0 generate
          tmpP_L5(2*i+1) <= not(tmpP_L4(2*i+1)) ;
          tmpG_L5(2*i+1) <= not(tmpG_L4(2*i+1)) ;
         end generate L5_vide ;

------- 7ème étage de cellules PG -------
```

```
L6 : for i in 63 downto 32 generate
     instPG_n : PG_b_c
      port map(tmpP_L5(2*i+1), tmpG_L5(2*i+1),
               tmpP_L5(2*i-63), tmpG_L5(2*i-63),
               tmpP_L6(2*i+1), tmpG_L6(2*i+1)
              ) ;
    end generate L6 ;
L6_vide : for i in 31 downto 0 generate
          tmpP_L6(2*i+1) <= not(tmpP_L5(2*i+1)) ;
          tmpG_L6(2*i+1) <= not(tmpG_L5(2*i+1)) ;
         end generate L6_vide ;

------- 8ème étage de cellules PG -------

L7_pair : for i in 64 downto 1 generate
          instPG_n : PG_c_b
           port map(tmpP_reg(2*i), tmpG_reg(2*i),
                    tmpP_L6(2*i-1), tmpG_L6(2*i-1),
                    tmpP_L7(2*i), tmpG_L7(2*i)
                   ) ;
         end generate L7_pair ;

L7_impair : for i in 63 downto 1 generate
            tmpP_L7(2*i+1) <= tmpP_L6(2*i+1) ;
            tmpG_L7(2*i+1) <= tmpG_L6(2*i+1) ;
           end generate L7_impair ;

tmpP_L7(0) <= not(tmpP_reg(0)) ;
tmpG_L7(0) <= not(tmpG_reg(0)) ;
tmpP_L7(1) <= tmpP_L6(1) ;
tmpG_L7(1) <= tmpG_L6(1) ;

--- Signaux de sortie du soustracteur ---

Lcoutbimp : for i in 63 downto 0 generate
            tmp_Coutb(2*i+1) <= (tmpP_L7(2*i) nor not(Cin_HC))
                                   nor not(tmpG_L7(2*i)) ;
           end generate Lcoutbimp ;

Lcoutbpair : for i in 63 downto 0 generate
             tmp_Coutb(2*i+2) <= not(tmpP_L7(2*i+1) nand Cin_HC)
                                   nor tmpG_L7(2*i+1) ;
            end generate Lcoutbpair ;

tmp_S(0) <= tmpP_reg(0) xor Cin_HC ;

Lfinpair : for i in 63 downto 1 generate
           tmp_S(2*i)   <= not(tmpP_reg(2*i) xor tmp_Coutb(2*i)) ;
          end generate Lfinpair ;
```

```
Lfinimp : for i in 63 downto 0 generate
          tmp_S(2*i+1) <= tmpPiib_reg(2*i+1) xor tmp_Coutb(2*i+1);
         end generate Lfinimp;

----- Signaux de controle en sortie -----

tmp_Cout1 <= (Cout_p_fin xor Cout_m_fin xor tmp_Coutb(128));
tmp_Cout2 <= mux2(Gnd, Cout_p_fin nor not(Cout_m_fin), tmp_Coutb(128));

tmp_Vd_S  <= not(tmpP_reg(128) xor tmp_Coutb(128));

tmp_Dd_S  <= Signe_S_reg and (tmp_Coutb(128) xor tmp_Coutb(127));

tmp_NS_S  <= Signe_S_reg and not(tmp_Coutb(128) xor tmp_Coutb(127));

---- Génération des types 1, 2 et 3 (suite) ----

d1_3 : for i in 31 downto 0 generate
       d1_S3(i) <= d1_reg(2*i) nand d1_reg(2*i+1);
      end generate d1_3;

d1_4 : for i in 15 downto 0 generate
       d1_S4(i) <= d1_S3(2*i) nor d1_S3(2*i+1);
      end generate d1_4;

d1_5 : for i in 7 downto 0 generate
       d1_S5(i) <= d1_S4(2*i) nand d1_S4(2*i+1);
      end generate d1_5;

d1_6 : for i in 3 downto 0 generate
       d1_S6(i) <= d1_S5(2*i) nor d1_S5(2*i+1);
      end generate d1_6;

d1_7 : for i in 1 downto 0 generate
       d1_S7(i) <= d1_S6(2*i) nand d1_S6(2*i+1);
      end generate d1_7;

d1_S8 <= d1_S7(0) nor d1_S7(1);

d0 <= mux2(d1_S8, tmpP_L7(127), Cin_HC); -- car Cin_HC est
d1 <= mux2(tmpP_L7(127), d1_S8, Cin_HC); -- complementee!!!!

--------- Calcul de la longueur ---------

Cmpt_p1 : compteur_5bits_init1
 port map(pin_init_S, clk,
          Cpt_p1(0), Cpt_p1(1), Cpt_p1(2), Cpt_p1(3), Cpt_p1(4));
Cmpt : compteur_4bits
 port map(pin_init_S, clk, Cpt(0), Cpt(1), Cpt(2), Cpt(3));
```

```
out_mux_cpt(0) <= mux2(Cpt_p1(0), Cpt(0), (Cpt_p1(4) nor not(tmp_Dd_S)));
out_mux_cpt(1) <= mux2(Cpt_p1(1), Cpt(1), (Cpt_p1(4) nor not(tmp_Dd_S)));
out_mux_cpt(2) <= mux2(Cpt_p1(2), Cpt(2), (Cpt_p1(4) nor not(tmp_Dd_S)));
out_mux_cpt(3) <= mux2(Cpt_p1(3), Cpt(3), (Cpt_p1(4) nor not(tmp_Dd_S)));

reg_NSi : registre4
 port map(clk, in_reg_NS, out_reg_NS);

reg_Si : registre4
 port map(clk, in_reg_S, out_reg_S);

reg_s127 : registre
 port map(clk, tmp_S(127), s127_ret);

cmde_NS <= d0 and not(tmp_Dd_S);

Cmde_S <= mux2(d0, d1, s127_ret);

in_reg_NS(0) <= mux2(out_mux_cpt(0), out_reg_NS(0), cmde_NS);
in_reg_NS(1) <= mux2(out_mux_cpt(1), out_reg_NS(1), cmde_NS);
in_reg_NS(2) <= mux2(out_mux_cpt(2), out_reg_NS(2), cmde_NS);
in_reg_NS(3) <= mux2(out_mux_cpt(3), out_reg_NS(3), cmde_NS);

in_reg_S(0) <= mux2(out_mux_cpt(0), out_reg_S(0), cmde_S);
in_reg_S(1) <= mux2(out_mux_cpt(1), out_reg_S(1), cmde_S);
in_reg_S(2) <= mux2(out_mux_cpt(2), out_reg_S(2), cmde_S);
in_reg_S(3) <= mux2(out_mux_cpt(3), out_reg_S(3), cmde_S);

longueur(0) <= mux2(in_reg_NS(0), in_reg_S(0), tmp_NS_S);
longueur(1) <= mux2(in_reg_NS(1), in_reg_S(1), tmp_NS_S);
longueur(2) <= mux2(in_reg_NS(2), in_reg_S(2), tmp_NS_S);
longueur(3) <= mux2(in_reg_NS(3), in_reg_S(3), tmp_NS_S);

------------- Comparaisons --------------

EGAL <= not(tmpP_L7(128));
SUP  <= not(tmpG_L7(128));
tmp_result_comp <= CT1_reg xor mux2(mux2(EGAL, SUP, CT3_reg),
                                    EGAL nor SUP,
                                    CT2_reg);

-------------------------------------------------------------
---                REGISTRES DE SORTIE                    ---
-------------------------------------------------------------

reg_S : for i in 0 to 127 generate
        reg : registre
         port map(clk, tmp_S(i), pout_S(i));
end generate reg_S;
```

```
  reg_Cout1 : registre
   port map(clk, tmp_Cout1, pout_Cout1);

  reg_Cout2 : registre
   port map(clk, tmp_Cout2, pout_Cout2);

  reg_Vd_S : registre
   port map(clk, tmp_Vd_S, pout_Vd_S);

  reg_Dd_S : registre
   port map(clk, tmp_Dd_S, pout_Dd_S);

  reg_NS_S : registre
   port map(clk, tmp_NS_S, pout_NS_S);

  reg_lg_S : registre4
   port map(clk, longueur, pout_long_S);

  reg_comp : registre
   port map(clk, comp2, pout_comparaison);

  reg_result_comp : registre
   port map(clk, tmp_result_comp, pout_result_comp);

end arch_add;

configuration cfg_additionneur of additionneur is
for arch_add
end for;
end cfg_additionneur;
```

Le fichier `add_ctrl.vhd` contient la description VHDL de l'ensemble de l' unité d'addition.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.elem_fonctions.all;
use work.elem_composants.all;
use work.elem_unites.all;

entity add_ctrl is
port(pin_addA_H, plong_A           : in  std_ulogic_vector(3 downto 0);
     pin_addB_H, plong_B           : in  std_ulogic_vector(3 downto 0);
     pin_addC_H, pin_addC_L        : in  std_ulogic_vector(3 downto 0);
     pin_addS_H, pin_addS_L        : in  std_ulogic_vector(3 downto 0);
     pin_addregS_H                 : in  std_ulogic_vector(3 downto 0);
     pin_add_sous                  : in  std_ulogic;
     pin_comparaison               : in  std_ulogic;
     pin_digit_tab                 : in  std_ulogic;
     pin_Cinb_Cin                  : in  std_ulogic;
     pin_CT1, pin_CT2, pin_CT3     : in  std_ulogic;
     clk, clk2                     : in  std_ulogic;
     pin_OK                        : in  std_ulogic;
     pout_addA_H, pout_addA_L      : out std_ulogic_vector(3 downto 0);
     pout_addB_H, pout_addB_L      : out std_ulogic_vector(3 downto 0);
     pout_addC_H, pout_addC_L      : out std_ulogic_vector(3 downto 0);
     pin_data_A, pin_data_B        : in  std_ulogic_vector(127 downto 0);
     pin_NS_A, pin_NS_B            : in  std_ulogic;
     pin_tab_lg_A, pin_tab_lg_B    : in  std_ulogic_vector(3 downto 0);
     pin_reg_ctrl_A                : in  std_ulogic_vector(4 downto 0);
     pin_reg_ctrl_B                : in  std_ulogic_vector(4 downto 0);
     pin_reg_ctrl_C                : in  std_ulogic_vector(4 downto 0);
     pout_init                     : out std_ulogic;
     pout_init_S                   : out std_ulogic;
     pout_data_A, pout_data_B      : out std_ulogic_vector(127 downto 0);
     pout_NS_A, pout_NS_B          : out std_ulogic;
     pout_D_A, pout_D_B            : out std_ulogic;
     pout_Vd_A, pout_Vd_B          : out std_ulogic;
     pout_Cin2, pout_Cin1          : out std_ulogic;
     pout_add_sous                 : out std_ulogic;
     pout_comparaison              : out std_ulogic;
     pout_NF_A, pout_NF_B          : out std_ulogic;
     pout_CT1, pout_CT2, pout_CT3  : out std_ulogic;
     pout_addS_H, pout_addS_L      : out std_ulogic_vector(3 downto 0);
     pout_addregS_H, pout_addregS_L : out std_ulogic_vector(3 downto 0)
    );
end add_ctrl;
```

```
architecture arch_add_ctrl of add_ctrl is

signal d4_S, S                              : std_ulogic_vector(127 downto 0);

signal addA_H, lg_A, tab_lg_A, long_A,
       addB_H, lg_B, tab_lg_B, long_B,
       addC_H, addC_L,
       d2_addS_H, d3_addS_H, d4_addS_H,d2_addregS_H,
       mux_addregS_H, d3_addregS_H, d4_addregS_H,
       d2_addS_L, mux_addS_L, d3_addS_L, d4_addS_L,
       d4_long, d2_addregS_L, d3_addregS_L, d4_addregS_L,
       long_S, out_addS_H, out_addS_L, out_addregS_H,
       out_addregS_L       : std_ulogic_vector(3 downto 0);

signal Vss, Gnd, OK_synchro, lect_reg, lect_regb,
       digit_tab, Cinb_Cin, NS_S, Dd_S, Vd_S, Cout_p_S, Cout_m_S,
       HC_Cout_S,d2_Cin_HC, d2_CT1, d2_CT2,
       d2_CT3, init, d1_init_S, d2_init_S, d3_init_S,
       d0_S, d1_S, d4_comp,
       qA_0, qA_1, qA_2, qA_3, qA_4, qp1A_0, qp1A_1, qp1A_2, qp1A_3,
       qp1A_4, lgA_0, lgA_1, lgA_2, lgA_3, lgB_0, lgB_1, lgB_2,
       lgB_3, d4_NS, d4_Dd, d4_Vd, d4_HC_Cout,
       mux_Cin1, mux_Cin2   : std_ulogic;

begin

  Gnd <= '0';
  Vss <= '1';
  OK_synchro <= pin_OK and not(clk);

  ------------------------------------------------------------
  ---                 REGISTRES D'ENTREE                   ---
  ------------------------------------------------------------

  reg_lect_reg : registre
   port map(clk2, OK_synchro, lect_reg);

  reg_init1 : registre
   port map(clk2, lect_reg, d1_init_S);

  reg_init2 : registre
   port map(clk2, d1_init_S, d2_init_S);

  reg_init3 : registre
   port map(clk2, d2_init_S, d3_init_S);

  reg_init4 : registre
   port map(clk2, d3_init_S, pout_init_S);
```

```
lect_regb <= not(lect_reg);
reg_d2_init : registreRESET
 port map(clk, lect_regb, Vss, init);

reg_addA_H : registre4
 port map(lect_reg, pin_addA_H, addA_H);

reg_long_A : registre4
 port map(lect_reg, plong_A, lg_A);

reg_addB_H : registre4
 port map(lect_reg, pin_addB_H, addB_H);

reg_long_B : registre4
 port map(lect_reg, plong_B, lg_B);

reg_addC_H : registre4
 port map(lect_reg, pin_addC_H, addC_H);

reg_addC_L : registre4
 port map(lect_reg, pin_addC_L, addC_L);

reg_add_sous : registre
 port map(lect_reg , pin_add_sous , pout_add_sous);

reg_comparaison : registre
 port map(lect_reg , pin_comparaison , pout_comparaison);

reg_digit_tab : registre
 port map(lect_reg , pin_digit_tab , digit_tab);

reg_Cinb_Cin : registre
 port map(lect_reg , pin_Cinb_Cin , Cinb_Cin);

reg_d2_CT1 : registre
 port map(lect_reg , pin_CT1 , pout_CT1);

reg_d2_CT2 : registre
 port map(lect_reg , pin_CT2 , pout_CT2);

reg_d2_CT3 : registre
 port map(lect_reg , pin_CT3 , pout_CT3);

reg_longueur_A : registre4
 port map(lect_reg, pin_tab_lg_A, tab_lg_A);

reg_longueur_B : registre4
 port map(lect_reg, pin_tab_lg_B, tab_lg_B);
```

```
reg_d2_addS_H : registre4
 port map(lect_reg, pin_addS_H, d2_addS_H);

reg_d2_addS_L : registre4
 port map(lect_reg, pin_addS_L, d2_addS_L);

reg_d2_addregS_H : registre4
 port map(lect_reg, pin_addregS_H, d2_addregS_H);

--- Opération avec/sans retenue entrante ---

pout_Cin1 <= mux2(Gnd, pin_reg_ctrl_C(0), Cinb_Cin);
pout_Cin2 <= mux2(Gnd, pin_reg_ctrl_C(1), Cinb_Cin);

Inst_compteur : compteur_5bits
 port map(lect_reg, clk, qA_0, qA_1, qA_2, qA_3, qA_4);

---------- Signaux intermédiaires -----------
------ pour la génération des adresses ------
--------- de lecture des registre -----------

Inst_add_p1 : add_p1_5bits
 port map(mux_addS_L(3), mux_addS_L(2),mux_addS_L(1), mux_addS_L(0),
          qp1A_4, qp1A_3, qp1A_2, qp1A_1, qp1A_0);

--- Signaux pour l'adresse de la donnée A ---

long_A(0) <= mux2(lg_A(0), tab_lg_A(0), init);
long_A(1) <= mux2(lg_A(1), tab_lg_A(1), init);
long_A(2) <= mux2(lg_A(2), tab_lg_A(2), init);
long_A(3) <= mux2(lg_A(3), tab_lg_A(3), init);
long_B(0) <= mux2(lg_B(0), tab_lg_B(0), init);
long_B(1) <= mux2(lg_B(1), tab_lg_B(1), init);
long_B(2) <= mux2(lg_B(2), tab_lg_B(2), init);
long_B(3) <= mux2(lg_B(3), tab_lg_B(3), init);

--- Signaux pour l'adresse de la donnée A ---

pout_addA_L(0) <= mux2(long_A(0), qA_0, digit_tab);
pout_addA_L(1) <= mux2(long_A(1), qA_1, digit_tab);
pout_addA_L(2) <= mux2(long_A(2), qA_2, digit_tab);
pout_addA_L(3) <= mux2(long_A(3), qA_3, digit_tab);
pout_addA_H <= addA_H;

lgA_0 <= mux2(gnd, long_A(0), digit_tab);
lgA_1 <= mux2(gnd, long_A(1), digit_tab);
lgA_2 <= mux2(gnd, long_A(2), digit_tab);
lgA_3 <= mux2(gnd, long_A(3), digit_tab);
```

```
Inst_sous_A : soust_4bits
 port map(lgA_3, qA_3, lgA_2, qA_2, lgA_1, qA_1, lgA_0, qA_0,
         pout_NF_A) ;

--- Signaux pour l'adresse de la donnée B ---

pout_addB_L(0) <= mux2(long_B(0), qA_0, digit_tab) ;
pout_addB_L(1) <= mux2(long_B(1), qA_1, digit_tab) ;
pout_addB_L(2) <= mux2(long_B(2), qA_2, digit_tab) ;
pout_addB_L(3) <= mux2(long_B(3), qA_3, digit_tab) ;
pout_addB_H <= addB_H ;
lgB_0 <= mux2(gnd, long_B(0), digit_tab) ;
lgB_1 <= mux2(gnd, long_B(1), digit_tab) ;
lgB_2 <= mux2(gnd, long_B(2), digit_tab) ;
lgB_3 <= mux2(gnd, long_B(3), digit_tab) ;

Inst_sous_B : soust_4bits
 port map(lgB_3, qA_3, lgB_2, qA_2, lgB_1, qA_1, lgB_0, qA_0,
         pout_NF_B) ;

--- Signaux pour l'adresse de la donnée C ---

pout_addC_H <= addC_H ;
pout_addC_L <= addC_L ;

--- Signaux pour l'adresse de la sortie S ---

mux_addS_L(0) <= mux2(d2_addS_L(0), qA_0, digit_tab) ;
mux_addS_L(1) <= mux2(d2_addS_L(1), qA_1, digit_tab) ;
mux_addS_L(2) <= mux2(d2_addS_L(2), qA_2, digit_tab) ;
mux_addS_L(3) <= mux2(d2_addS_L(3), qA_3, digit_tab) ;

------- Propagation de l'adresse de S -------
----------- à travers le pipeline -----------

reg_d3_addS_L : for i in 0 to 3 generate
                reg : registre
                 port map(clk, mux_addS_L(i), d3_addS_L(i)) ;
end generate reg_d3_addS_L ;

reg_d4_addS_L : for i in 0 to 3 generate
                reg : registre
                 port map(clk, d3_addS_L(i), d4_addS_L(i)) ;
end generate reg_d4_addS_L ;

reg_d3_addS_H : for i in 0 to 3 generate
                reg : registre
                 port map(clk, d2_addS_H(i), d3_addS_H(i)) ;
end generate reg_d3_addS_H ;
```

```
reg_d4_addS_H : for i in 0 to 3 generate
                reg : registre
                 port map(clk, d3_addS_H(i), d4_addS_H(i)) ;
end generate reg_d4_addS_H ;

---------- Génération de l'adresse ----------
---------- du registre de controle ----------
---------- de sortie (en écriture) ----------

d2_addregS_L(0) <= qp1A_0 ;
d2_addregS_L(1) <= qp1A_1 ;
d2_addregS_L(2) <= qp1A_2 ;
d2_addregS_L(3) <= qp1A_3 ;
reg_d3_addregS_L : for i in 0 to 3 generate
                   reg : registre
                    port map(clk, d2_addregS_L(i), d3_addregS_L(i)) ;
end generate reg_d3_addregS_L ;

reg_d4_addregS_L : for i in 0 to 3 generate
                   reg : registre
                    port map(clk, d3_addregS_L(i), d4_addregS_L(i)) ;
end generate reg_d4_addregS_L ;

mux_addregS_H(0) <= mux2(d2_addS_H(0), d2_addregS_H(0), qp1A_4) ;
mux_addregS_H(1) <= mux2(d2_addS_H(1), d2_addregS_H(1), qp1A_4) ;
mux_addregS_H(2) <= mux2(d2_addS_H(2), d2_addregS_H(2), qp1A_4) ;
mux_addregS_H(3) <= mux2(d2_addS_H(3), d2_addregS_H(3), qp1A_4) ;

reg_d3_addregS_H : for i in 0 to 3 generate
                   reg : registre
                    port map(clk, mux_addregS_H(i), d3_addregS_H(i)) ;
end generate reg_d3_addregS_H ;

reg_d4_addregS_H : for i in 0 to 3 generate
                   reg : registre
                    port map(clk, d3_addregS_H(i), d4_addregS_H(i)) ;
end generate reg_d4_addregS_H ;

-------------------------------------------------------------
---                 REGISTRES DE SORTIE                   ---
-------------------------------------------------------------

pout_data_A <= pin_data_A ;
pout_data_B <= pin_data_B ;

pout_NS_A <= pin_NS_A ;
pout_NS_B <= pin_NS_B ;
```

```
  pout_Vd_A <= pin_reg_ctrl_A(2) ;
  pout_Vd_B <= pin_reg_ctrl_B(2) ;

  pout_D_A <= pin_reg_ctrl_A(3) ;
  pout_D_B <= pin_reg_ctrl_B(3) ;

  pout_init <= init ;
  ----
  reg_out_addS_H : registre4
   port map(clk, d4_addS_H, pout_addS_H) ;
  ----
  reg_out_addS_L : registre4
   port map(clk, d4_addS_L, pout_addS_L) ;
  ----
  reg_out_addregS_H : registre4
   port map(clk, d4_addregS_H, pout_addregS_H) ;
  ----
  reg_out_addregS_L : registre4
   port map(clk, d4_addregS_L, pout_addregS_L) ;

end arch_add_ctrl ;

configuration cfg_add_ctrl of add_ctrl is
for arch_add_ctrl
end for ;
end cfg_add_ctrl ;
```