HAL
open science

Une méthodologie de conceptionde circuits intégrés quasi-insensibles aux délais :application à l'étude et à la réalisation d'un processeur RISC 16-bit asynchrone

Pascal Vivet

To cite this version:

Pascal Vivet. Une méthodologie de conceptionde circuits intégrés quasi-insensibles aux délais :application à l'étude et à la réalisation d'un processeur RISC 16-bit asynchrone. Micro et nanotechnologies/Microélectronique. Institut National Polytechnique de Grenoble - INPG, 2001. Français. NNT : . tel-00002974

HAL Id: tel-00002974

https://theses.hal.science/tel-00002974

Submitted on 11 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.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|_|_|_|_|_|_|_|_|_|_|_|

THESE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : « MICROELECTRONIQUE »
préparée au laboratoire de **FRANCE TELECOM R&D, Meylan**
dans le cadre de ***l'Ecole Doctorale « ELECTRONIQUE, ELECTROTECHNIQUE, AUTOMATIQUE, TELECOMMUNICATIONS, SIGNAL »***

présentée et soutenue publiquement

par

Pascal VIVET

le 21 Juin 2001

UNE METHODOLOGIE DE CONCEPTION DE CIRCUITS INTEGRES QUASI-INSENSIBLES AUX DELAIS : APPLICATION A L'ETUDE ET A LA REALISATION D'UN PROCESSEUR RISC 16-BIT ASYNCHRONE.

Directeur de thèse : Marc Renaudin

JURY

M. Pierre Gentil	**, Président**
M. Daniel Litaize	**, Rapporteur**
M. Patrice Quinton	**, Rapporteur**
M. Marc Renaudin	**, Directeur de thèse**
M. Jean-Pierre Schoellkopf	**, Examinateur**
M. Patrice Senn	**, Examinateur**

A mon père

Remerciements

Le travail présenté dans cette thèse a été effectué au sein du laboratoire DTM/CET de France Telecom R&D, anciennement Centre National d'Etudes des Télécommunications de Grenoble.

Je remercie Daniel Bois, ancien directeur du site du CNET Grenoble, et Patrice Senn, chef du département DTM/CET (aujourd'hui laboratoire DIH/OCF), pour m'avoir permis d'effectuer mes travaux de recherches au CNET.

Je tiens tout particulièrement à remercier Marc Renaudin, mon directeur de thèse, anciennement maître de conférences à l'E.N.S.T de Bretagne, aujourd'hui chef du groupe CIS au laboratoire TIMA et professeur à l'ENSERG de Grenoble, pour m'avoir encadré tout au long de ces travaux. Ces années passées ensembles furent toujours l'occasion de discussions passionnantes, je lui dois des voies de recherches originales, ainsi qu'une formation scientifique et d'esprit formidable. Je le remercie du temps qu'il a pu me consacrer, ainsi que de la confiance et du soutien qu'il a su m'accorder pour terminer à bien ce manuscrit.

Je tiens à remercier Daniel Litaize, professeur à l'IRIT, Université Paul Sabatier de Toulouse, et Patrice Quinton, professeur à l'INRIA / IRISA, Université de Rennes, d'avoir accepté d'être les rapporteurs de mes travaux de thèse. J'adresse aussi mes remerciements à Pierre Gentil, professeur à l'ENSERG, de m'avoir fait l'honneur de présider mon jury de thèse.

Je remercie aussi Jean-Pierre Schoellkopf, chef du département ADT à STMicroelectronics Crolles d'avoir participé à mon jury de thèse, ainsi que de m'avoir accordé sa confiance et son soutien pour terminer ce manuscrit.

Je salue et j'adresse tous mes remerciements à tous mes anciens collègues et amis du CNET Grenoble, avec qui j'ai passé des années formidables, qui m'ont apporté leur support et leurs compétences dans la bonne humeur et dans des domaines très variés : Rolland, Tomas, Luc, Henri, Philippe, André, Jacky, … Tous mes remerciements aussi aux nombreux stagiaires qui ont participé et contribué à l'ensemble de ces travaux : Fatima, Aissam, Kham, Didier, Edith, Claire. Je veux aussi exprimer toute ma sympathie et adresser mon amitié aux « anciens » thésards du CNET : Frédéric, Eric, Gilles, Christophe D., Christophe S., ...

Tous mes remerciements aussi à l'ensemble de mes « nouveaux » collègues de STMicroelectronics à Crolles : Frédéric, Alexandre, Philippe F., Philippe R. pour m'avoir encouragé dans la bonne humeur.

Un immense MERCI à tous mes proches, famille et amis, qui m'ont apporté leur soutien tout au long de ce travail de longue haleine. Enfin, une pensée très tendre pour « ma » petite Christine, et une immense marque d'affection pour notre petite fille qui poindra le bout de son nez en Septembre.

Tables des matières

INTRODUCTION **1**

Chapitre 1 : Etat de l'art sur la conception des circuits asynchrones **5**

Introduction 5
1.1. Concepts de base 6
1.1.1. Le mode de fonctionnement asynchrone 7
1.1.2. Un contrôle local 8
1.1.2.1. Protocoles de communications 8
1.1.2.2. Codage des données 10
1.1.3. Caractéristiques des opérateurs asynchrones 12
1.1.4. Conclusion 13
1.2. Avantages et potentiels des circuits asynchrones 14
1.2.1. Calcul en temps minimum, en temps moyen 14
1.2.2. Distribution du contrôle, un pipeline élastique 15
1.2.3. Un support fiable pour les traitement non-déterministes 15
1.2.4. Absence d'horloge 16
1.2.5. Faible consommation 16
1.2.6. Faible bruit et système radio-fréquence 18
1.2.7. Variation des temps de propagation, Migration 19
1.2.8. Modularité, Réutilisation 20
1.3. Méthodes & outils de conception de circuits asynchrones 21
1.3.1. Classification des circuits asynchrones 21
1.3.1.1. Circuits Insensibles aux Délais (Delay Insensitive) 23
1.3.1.2. Circuits Quasi Insensibles aux Délais (Quasi Delay Insensitive) 23
1.3.1.3. Circuits Indépendants de la Vitesse (Speed Independant) 23
1.3.1.4. Circuits de Huffman 24
1.3.1.5. Micropipeline 24
1.3.2. Méthodologies et outils de conception 26
1.3.2.1. Circuits Insensibles aux Délais 26
1.3.2.2. Circuits Quasi Insensibles aux Délais 26
1.3.2.3. Circuits Indépendant de la vitesse 27
1.3.2.4. Circuits de Huffman 28
1.3.2.5. Micropipeline 29
1.3.3. Conclusion 29
1.4. Différents microprocesseurs asynchrones 30
1.4.1. Produits commerciaux 30
1.4.1.1. Microcontrôleur 80C51 (Philips Research & Semiconductor, 1998) 30
1.4.1.2. Data Driven Media Processor (Sharp, 1999) 31
1.4.2. Processeurs asynchrones 32
1.4.2.1. Caltech Asynchronous Processor (Caltech, 1989) 32
1.4.2.2. Amulet 1, 2e, 3i (Université de Manchester, 1995, 1996, 2000) 32
1.4.2.3. Titac-2 (Université & Institut de Technologie de Tokyo, 1994, 1997) 33
1.4.2.4. MiniMIPS (Caltech, 1999) 34
1.5. Conclusion 35

Partie I Une méthodologie de conception de circuits asynchrones quasi-insensibles aux délais 37

Chapitre 2 : Un environnement de simulation pour circuit mixte synchrone/asynchrone basé sur la traduction CHP vers VHDL 39

Introduction 39
2.1. Proposition d'une méthode de conception mixte synchrone / asynchrone 40
2.1.1. Objectif d'un tel environnement de conception 40
2.1.2. Présentation de la méthode de conception 41
2.2. Le langage CHP 43
2.2.1. Type de données, Opérateurs et Variables 44
2.2.2. Canaux et actions de communication 45
2.2.3. Instructions élémentaires 47
2.2.4. Opérateurs de composition. 47
2.2.5. Commandes gardées, instructions de contrôle et déterminisme 48
2.2.6. Modélisation structurelle 50
2.2.7. Utilisation de facilités des HDLs 52
2.2.8. Conclusion 53
2.3. Le traducteur CHP_2VHDL 53
2.3.1. Présentation de l'outil 53
2.3.2. Modélisation de la concurrence 55
2.3.3. Modélisation structurelle 56
2.3.4. Canaux et actions de communication 56
2.3.5. Modélisation temporelle 58
2.3.6. Les structures de contrôle 59
2.3.7. Aide à la co-simulation 60
2.3.8. Gestion de projet et exemple de simulation 62
2.4. Conclusion 63

Chapitre 3 : Synthèse de circuit asynchrone quasi-insensible aux délais en cellules standard 65

Introduction 65
3.1. Synthèse de circuit QDI en cellules standard 66
3.1.1. Présentation de la méthode de conception 66
3.1.1.1. Expansion des communications et codage des canaux. 67
3.1.1.2. Optimisation des HSE : Réordonnancement 69
3.1.1.3. Génération des règles de production. 70
3.1.1.4. Optimisation logique 73
3.1.1.5. Conclusion 73
3.1.2. Choix et adoption d'un style de synthèse 74
3.1.2.1. Protocole WCHB 74
3.1.2.2. Protocole séquentiel 76
3.1.2.3. Politique d'initialisation 77
3.1.2.4. Conclusion 77
3.2. Définition d'un CHP synthétisable 78
3.2.1. Définition du CHP synthétisable 79
3.2.2. Exemples de décompositions CHP pour obtenir un CHP synthétisable 80
3.2.2.1. Extraction de variables 80

3.2.2.2. Transformation en machine à état 83
3.2.3. Conclusion 84
3.3. Conception d'une bibliothèque de cellules spécifiques....85
3.3.1. Définition de la bibliothèque 85
3.3.1.1. Portes de Muller....85
3.3.1.2. Portes de Muller généralisées 86
3.3.1.3. Macro-cellules 87
3.3.1.4. Additionneur : Full Adder 1-bit. 88
3.3.1.5. Portes diverses 90
3.3.2. Conception de la bibliothèque 91
3.3.3. Conclusion 92
3.4. Proposition d'un flot de conception....93
3.5. Conclusion 96

Partie II Le processeur Aspro : conception d'une architecture RISC 16-bit quasi-insensible aux délais 97

Chapitre 4 : Spécification et architecture du processeur Aspro 99

4.1. Introduction....99
4.1.1. Pourquoi un processeur ?....99
4.1.2. Une architecture adaptée à l'asynchrone 100
4.1.3. Applications et potentiels....101
4.1.4. Environnement logiciel....102
4.1.5. Conclusion 102
4.2. Jeu d'instructions et périphériques 103
4.2.1. Une architecture RISC 16-bit scalaire 103
4.2.2. Périphériques et signaux externes....104
4.2.2.1. Port mémoire programme 104
4.2.2.2. Ports parallèles....104
4.2.2.3. Liens séries 105
4.2.2.4. Gestion des interruptions, mode veille....106
4.2.2.5. Reset, mode de boot....107
4.2.2.6. Système d'alimentation 107
4.2.3. Une architecture personnalisable 108
4.2.4. Jeu d'instructions....108
4.2.4.1. Instructions Arithmétiques et Logiques. 109
4.2.4.2. Instructions Load / Store....109
4.2.4.3. Instructions de branchement, de contrôle 111
4.2.4.4. Instructions utilisateurs....112
4.2.5. Encodage du jeu d'instructions....113
4.3. Architecture du processeur Aspro....114
4.3.1. Introduction à la notion de pipeline asynchrone....115
4.3.2. Architecture du processeur 117
4.3.2.1. Boucle de Fetch-Decode....118
4.3.2.2. Boucle de branchement....119
4.3.2.3. Chemin de données....120
4.3.2.4. Synchronisation Load-Store / Mémoire programme 121
4.3.2.5. Mécanisme d'interruption et mise en veille....122

4.3.2.6. Personnalisation du processeur 123
4.4. Optimisation du code à la compilation 123
4.5. Conclusion 125

Chapitre 5 : Le processeur Aspro : Architecture et micro-architecture 127

Introduction 127
5.1. Architecture des mémoires 128
5.1.1. Entrelacement entre différents sous-blocs mémoires 129
5.1.2. Optimisation de l'architecture de la mémoire 131
5.1.3. Conclusion et comparaison avec une réalisation synchrone 133
5.2. Etude des unités d'exécution 134
5.2.1. Unité arithmétique et logique 134
5.2.1.1. Unité logique 135
5.2.1.2. Unité Arithmétique 135
5.2.1.3. Unité de comparaison 136
5.2.1.4. Unité décaleur 138
5.2.1.5. Conclusion 140
5.2.2. Unité de Multiplication-accumulation 141
5.2.3. Unité de branchement 144
5.2.4. Unité Load-Store 148
5.2.5. Conclusion sur les unités d'exécution 152
5.3. Banc de registres 154
5.3.1.1. Vue globale du banc de registre 154
5.3.1.2. Mécanisme de réservation 156
5.3.1.3. Implémentation du registre 157
5.3.1.4. Conclusion 159
5.4. Arithmétique et calcul en temps moyen 159
5.4.1. Additionneur 159
5.4.1.1. La cellule *Full-Adder* 160
5.4.1.2. Additionneur séquentiel 161
5.4.1.3. Additionneur à retenue bondissante 162
5.4.1.4. Optimisation en débit des additionneurs 164
5.4.1.5. Conclusion 165
5.4.2. Multiplieur - accumulateur. 165
5.5. Boucle de Fetch : Décodeur et Unité PC. 172
5.5.1. Le décodeur 173
5.5.1.1. Pré-décodage 174
5.5.1.2. Contrôle du décodeur : branchements et interruptions 174
5.5.1.3. Décodeur instruction 177
5.5.2. Unité PC. 179
5.5.3. Initialisation de la boucle de Fetch 181
5.5.4. Conclusion 182
5.6. Les périphériques 183
5.7. Conclusion sur l'étude du processeur 186

Chapitre 6 : Le processeur ASPRO : synthèse, optimisation et résultats silicium 189

Introduction 189
6.1. Conception du cœur du processeur Aspro 190

6.1.1. Synthèse de processus combinatoires 190
6.1.2. Synthèse de processus séquentiels 195
6.1.3. Synthèse de choix non-déterministe 197
6.1.4. Codage des signaux d'acquittement 198
6.1.5. Conclusion 200
6.2. Conception des interfaces 201
6.2.1. Interface avec des mémoires synchrones 201
6.2.2. Interface liens série : conversion 2 phases / 4 phases. 204
6.3. Analyse des performances et optimisation 206
6.4. Optimisation du processeur Aspro 211
6.4.1. Implémentation, optimisation 211
6.4.2. Implémentation physique du processeur 213
6.5. Test et mesures de performances sur le circuit 215
6.5.1. Test du circuit 215
6.5.2. Mesures de performances 217
6.6. Conclusion 220

Conclusion 223

Publications 231

Bibliographie 233

INTRODUCTION

La prépondérance historique des circuits synchrones est un fait indéniable. La conception des circuits numériques emprunte aujourd'hui de manière quasi-systématique l'hypothèse synchrone. E.F Moore disait en 1956 « le temps est supposé être constitué d'événements discrets … en procurant une unique source d'horloge, il est même possible d'organiser des composants asynchrones entre eux de manière à ce qu'ils interagissent dans les évènements discrets d'une machine synchrone ». Cette hypothèse réductrice de synchronisation a permis pendant des années d'exploiter le potentiel des technologies d'intégration et de concevoir des circuits de plus en plus complexes et rapides.

Au début de l'intégration des circuits numériques, le but était l'intégration des circuits et non la recherche de performances : le mot d'ordre était d'intégrer le plus de fonctions possibles dans le moins de surface. L'hypothèse synchrone était alors adaptée. Le temps étant une ressource gratuite, l'horloge permettait d'organiser le séquencement des circuits de manière aisée. Les outils CAO s'orientèrent alors vers ce type de méthodologie. Les langages de description de matériel offrirent le moyen de modéliser au niveau RTL. Ceci permettait de séparer la description fonctionnelle des performances temporelles. Les flots de conception de type ASIC étaient nés : en utilisant une logique CMOS, des cellules standard et les outils de synthèse sur les langages HDL, il devint possible de concevoir avec beaucoup de productivité des systèmes relativement complexes.

Par la suite, les performances deviennent un argument stratégique. Dans la course au MHz, la période d'horloge étant une ressource de plus en plus précieuse, l'horloge n'est alors plus une « alliée ». En particulier pour la conception des processeurs hautes performances, la diminution de la période d'horloge impose de nouvelles méthodologies de conception : utilisation de logique dynamique, arbres d'horloge sophistiqués, micro-architectures super-pipelinées. Les outils de CAO, toujours basés sur les HDL, sont contraints à tous les niveaux par les performances : optimisation des arbres d'horloges (gigue), analyse statique de timing, insertion de délais, synthèse logique contrainte par le placement (et réciproquement), analyse du bruit de couplage, arrêt de l'horloge afin de réduire la consommation (« gated clock »). Cette problématique apparaît à son tour dans les méthodologies de conception de type ASIC.

La principale difficulté rencontrée aujourd'hui réside dans la convergence des « *timings* » : il est nécessaire d'itérer de nombreuses fois entre synthèse logique, phase de placement & routage, extraction des interconnexions, estimations de performances. Bien que les outils de CAO soient de plus en plus évolués, on observe une perte de productivité : le nombre de transistors par circuit augmente plus vite que le nombre de transistors conçus par ingénieur. La réalisation de circuit synchrone haute complexité - haute performance devient de plus en plus difficile. Afin de respecter l'hypothèse synchrone, il est nécessaire d'avoir confiance dans la caractérisation des dispositifs (portes et interconnexions). La conception des arbres d'horloge devient de plus en plus délicate, il se pose des problèmes de consommation et de bruit dus à l'horloge (couplage substrat). La contrainte de consommation devient aussi prépondérante pour tous les circuits embarqués : arrêt et redémarrage conditionnels de la logique et des horloges.

Avec les nouvelles technologies à très fort potentiel d'intégration arrivent de nouveaux challenges : dispersions technologiques de plus en plus fortes au sein d'une même puce, prédominance des interconnexions sur la logique (délais, bruit), consommation de plus en plus élevée (aléas, horloge), et enfin problématique de l'intégration des systèmes sur une puce (« System-on-Chip »). Ce dernier point devient prédominant. Afin de concevoir des systèmes sur une puce, il est nécessaire d'avoir aussi bien des compétences circuit mais aussi système. Ceci impose de disposer de niveaux d'abstractions élevés (synthèse comportementale sur des langages de haut niveau), et de disposer au niveau circuit d'un certain potentiel de modularité, réutilisation, migration (notion de « IP reuse »). Dans ce contexte, l'hypothèse de synchronisation par horloge globale apparaît trop réductrice pour offrir un niveau d'abstraction élevé.

Au contraire, grâce à la suppression de l'horloge globale et à un mécanisme de synchronisation local, les circuits asynchrones semblent tout à fait adaptés pour répondre à ces nouveaux challenges. De plus en plus d'acteurs s'intéressent à ce style de circuits [HAUC 95], [RENA 00b]. De forts potentiels peuvent en être attendus : robustesse, calcul en temps moyen, consommation conditionnelle, faible bruit, modularité, niveau d'abstraction plus élevé. De nombreux prototypes de circuits asynchrones d'une certaine complexité ont été réalisés avec succès et montrent de bons niveaux de performances. Le principal enjeu réside aujourd'hui dans l'adoption d'une méthodologie de conception de circuit asynchrone adaptée et à la hauteur des enjeux visés. A chaque rupture technologique et méthodologique, les concepteurs perdent en partie leur savoir faire, il est alors nécessaire de fournir en retour des avantages conséquents, en terme de savoir faire et d'outils CAO.

Dans ce contexte, le travail de thèse s'est focalisé dans deux directions : un travail au niveau méthodologique d'une part et la conception d'un prototype de processeur d'autre part. Ces deux études ne sont évidemment pas dissociées, elles ont été réalisées conjointement. La première permettant la réalisation du prototype, la seconde étant une preuve de faisabilité et un vecteur de recherche pour nourrir la réflexion à la fois sur les méthodes de conception et les architectures asynchrones.

Le chapitre 1 du manuscrit présente tout d'abord un état de l'art sur la conception des circuits asynchrones. Nous décrivons les concepts de base des circuits asynchrones : au contraire des circuits synchrones, ces circuits sont localement synchronisés grâce à un mécanisme de signalisation et de communication locale entre tous les éléments constituants le

circuit. Nous montrons alors les principaux bénéfices que l'ont peut tirer de ce principe de localité : robustesse par rapport aux conditions de fonctionnement, calcul en temps moyen à opposer à une approche temps de calcul pire cas synchrone, pipeline élastique, faible consommation, faible bruit, modularité, niveau d'abstraction plus élevé. Nous présentons ensuite les différents types de circuits asynchrones existants et les méthodologies de conception associées. Ce vaste spectre de solution est à l'image des différentes possibilités de relâchement des synchronisations dans le circuit. Ceci n'est néanmoins pas en faveur d'une vision unificatrice des circuits asynchrones et n'aide pas à leur valorisation dans la communauté des concepteurs synchrones. Pour conclure ce premier chapitre, afin de montrer le niveau de maturité des circuits asynchrones, nous présentons succinctement des processeurs asynchrones d'une complexité significative (y compris commerciaux) et leur résultats de performances.

Les chapitres 2 et 3, qui constituent la première partie du manuscrit, proposent une méthodologie de conception de circuits asynchrones quasi-insensibles aux délais réalisés en cellules standard basée sur le langage CHP. Le but est de proposer une méthodologie de conception qui soit adaptée aux enjeux de l'intégration des systèmes complexes dans les technologies avancées, avec comme contrainte son insertion dans les flots de conception existants : plate-forme de simulation, circuits en cellules standard.

Nous avons porté notre choix sur le style de circuit asynchrone Quasi-Insensible aux Délais. Ces circuits sont les plus robustes ; ils ont été initialement proposés et étudiés par le groupe du professeur A. Martin à l'université de Caltech [MART 90]. Ce groupe a développé une méthode de conception descendante qui à partir d'un langage de haut niveau (le langage CHP) permet d'obtenir le circuit par des règles de transformations successives. Le langage CHP modélise des processus concurrents synchronisés par des canaux de communications. Ce langage de modélisation issu du langage CSP est tout à fait adapté à la modélisation de circuit asynchrone et à leur mécanisme de synchronisation local par passage de message. La méthodologie de conception développé par Caltech a été éprouvée sur de nombreux exemples de circuits. Toutefois, aucun outil d'aide à la conception (simulation / synthèse) n'existe, cette méthode est totalement manuelle et de plus vise à concevoir des circuits dédiés.

Partant de ces acquis, nous avons tout d'abord enrichi cette méthodologie de conception par des possibilités de simulation. Dans le chapitre 2, nous présentons un traducteur CHP vers VHDL que nous avons développé qui permet l'étude de systèmes asynchrones modélisés en CHP dans un flot de simulation standard.

Dans le chapitre 3, nous présentons ensuite les différents choix d'implémentation bas niveau que nous avons effectués afin de cibler une bibliothèque de cellules standard et non des circuits dédiés. Ce travail nous a permis de définir une restriction du langage CHP afin qu'il soit synthétisable, ainsi qu'une bibliothèque de cellules standard qui a été réalisée dans une technologie cible. En conclusion de cette première partie du manuscrit, nous donnons une description du flot de conception complet que nous avons mis au point et utilisé pour réaliser le processeur : depuis la modélisation en CHP, jusqu'à l'implémentation sur les bibliothèques et la validation du design par co-simulation en VHDL.

Les chapitres 4, 5 et 6 constituent la deuxième partie du manuscrit et présentent respectivement la définition, l'étude et la réalisation d'un prototype de processeur RISC 16-bit dénommé Aspro. Ce processeur est une architecture originale qui intègre plusieurs unités d'exécutions concurrentes, dont une unité utilisateur et des périphériques dédiés.

Le jeu d'instructions du processeur et son architecture sont présentés dans le chapitre 4. En partant d'un exemple de décomposition sur le langage CHP, nous présentons le principe du pipeline asynchrone et ses conséquences en termes d'architecture et de synchronisation. Grâce à l'application de ces principes, l'architecture du processeur permet l'envoi des instructions dans l'ordre et leur terminaison dans le désordre, ceci étant rendu possible par l'utilisation d'un mécanisme local de réservation au sein du banc de registres du processeur.

Le chapitre 5 présente ensuite en détail la micro-architecture du processeur. Cette présentation est étayée point par point par le modèle CHP de chaque bloc et l'analyse des dépendances de données du modèle. Nous montrerons que par décomposition sur le langage CHP il est possible de concevoir des unités à temps de calcul minimum en fonction à la fois des instructions et des données. Ce principe permet de concevoir des architectures complexes tout en observant localement les dépendances de données inhérentes à la spécification d'origine. En particulier, nous montrerons qu'en asynchrone il est possible de décorréler les paramètres de latence et de débit. Ainsi, l'architecture du processeur utilise de manière quasi-systématique un principe d'entrelacement qui permet d'optimiser à la fois latence et débit des opérateurs élémentaires.

Pour terminer, le chapitre 6 présente l'implémentation bas niveau du processeur en utilisant la méthode de conception exposée dans la première partie du manuscrit ; différents exemples de synthèse seront donnés afin d'enrichir les exemples présentés dans le chapitre 3. Nous montrons ensuite comment il est possible d'analyser et d'optimiser les performances globales d'un système asynchrone. Nous noterons les différences avec un système synchrone et le lien avec le type de logique utilisé. Enfin, nous présenterons les mesures effectuées sur le processeur après sa fabrication et les comparerons avec les processeurs asynchrones existants.

Afin de guider le lecteur, ce dernier pourra lire de manière relativement aisée les différentes sous parties du manuscrit dans l'ordre qui lui convient. Le chapitre 1 donne un rapide aperçu de l'état de l'art sur la conception des circuits asynchrones et la justification de nos choix méthodologiques. Il est ensuite conseillé de lire le début du chapitre 2 qui introduit la syntaxe et la sémantique du langage CHP.

Les deux grandes parties du manuscrit sont relativement décorrélées : la première partie regroupant les chapitres 2 et 3 présentent notre proposition de flot de conception de circuits asynchrones Quasi-Insensibles aux Délais en cellules standard ; la deuxième partie du manuscrit regroupant les chapitres 4, 5, 6 est dédiée à l'étude et la réalisation du processeur Aspro. Ainsi, le lecteur pourra s'intéresser directement à la présentation du processeur Aspro dans les chapitres 4 et 5. Cependant, le chapitre 6 nécessite la lecture complète de la première partie.

La conclusion du manuscrit donnera une rétrospective du travail de thèse et présentera rapidement la réalisation d'un prototype de microcontrôleur pour carte sans contact. Enfin, nous présenterons les différents enjeux de la conception de circuits asynchrones et la suite des travaux de recherche qu'il reste à mener, tant au niveau méthodologie de conception qu'au niveau architecture de systèmes asynchrones complexes.

Chapitre 1 :

Etat de l'art sur la conception des circuits asynchrones

Introduction

Lorsque la conception des circuits numériques a débuté, il n'existait pas de distinction entre circuit synchrone et asynchrone. Les circuits synchrones correspondent à une classe restreinte de circuits qui sont séquencés par un signal périodique uniformément distribué : l'horloge. Au contraire, les circuits asynchrones sont des circuits dont le contrôle est assuré par tout autre méthode que le recours à un signal d'horloge. Ces circuits sont couramment appelé auto-synchronisés (« *self-timed* »). Très vite le style de conception synchrone s'est imposé pour répondre à des besoins de calcul croissants et pour s'adapter à une technologie encore bridée.

Pourtant, l'étude des circuits asynchrones a commencé au début des années 1950 pour concevoir des circuits à relais mécaniques. En 1956, Muller et Bartky de l'université d'Illinois ont travaillé sur la théorie des circuits asynchrones. Huffman est le premier à concevoir des machines à états asynchrones en 1968 avec ses travaux en « switching theory ». Par la suite, Muller propose d'associer un signal de validité aux données en introduisant un protocole de communication quatre-phases (cf. paragraphe 1.1.2). En 1966, le « Macromodule Project » à l'université de Washington [CLAR 67] montre qu'il est possible de concevoir des machines spécialisées complexes par simple composition de blocs fonctionnels asynchrones. Par la suite, Seitz introduit un formalisme proche des réseaux de Pétri pour concevoir des circuits

asynchrones, ce qui aboutit à la conception du premier calculateur data-flow (DDM1) [DAVI 78]. Enfin, Sutherland a largement contribué à l'intérêt croissant porté par les institutions académiques mais aussi industrielles à la conception de circuits asynchrones en publiant un article maintenant célèbre intitulé « Micropipeline » [SUTH 89]. Depuis, les travaux sur la conception de circuits asynchrones ne cessent de s'intensifier [HAUC 95].

Aujourd'hui, malgré une prépondérance historique des circuits synchrones, des outils de conception associés qui ne cessent de progresser, et de la formation des ingénieurs consacrée à ce style de circuit, de plus en plus d'acteurs s'intéressent au style de conception asynchrone en raison des difficultés croissantes rencontrées pour concevoir les circuits synchrones. Lors de la conférence ASYNC'2000 à Eilat, Israël, une présentation invitée de A. Kolodny de Intel Corporation posait clairement la question suivante : les circuits synchrones montrent des niveaux de performances remarquables mais ne paraissent plus adaptés aux technologies actuelles, en particulier dans la course au MHz. L'alternative asynchrone ne devrait-elle pas être prise plus au sérieux ?

Ce premier chapitre donne un rapide aperçu du style de conception asynchrone. Pour commencer, le paragraphe 1.1 présente les concepts de bases du mode de synchronisation des circuits asynchrones, le contrôle local qui permet de les implémenter et les différentes caractéristiques de ce style de circuit. Le paragraphe 2.2 montre ensuite les différents avantages et bénéfices que l'on peut tirer des circuits asynchrones par rapport aux circuits synchrones, en particulier en terme de vitesse, consommation, bruit, robustesse et propriété de modularité et de migration dans les technologies actuelles. Le paragraphe 3.3 présente ensuite les différentes classes de circuits asynchrones en fonction des hypothèses de délais nécessaires à leur implantation puis les différentes méthodologies de conception associées. Enfin, le paragraphe 1.4 présente des exemples de circuits asynchrones conséquents, avec la présentation de deux processeurs asynchrones commercialisés et différents prototypes académiques. Ce chapitre est largement inspiré d'un rapport écrit par Marc Renaudin sur l'état de l'art de la conception des circuits asynchrones [RENA 00b] [RENA 00c].

1.1. Concepts de base

La conception de la plupart des circuits intégrés logiques est facilitée par deux hypothèses fondamentales : les signaux manipulés sont binaires, et le temps est discrétisé. La binarisation des signaux permet une implémentation électrique simple et offre un cadre de conception maîtrisé grâce à l'algèbre de Boole. La discrétisation du temps permet de s'affranchir des problèmes de rétroactions (boucles combinatoires), ainsi que des fluctuations électriques transitoires.

Si les circuits asynchrones conservent un codage discret des signaux, la plupart du temps binaire, ils se distinguent des circuits synchrones car ils ne sont pas commandés par un signal périodique unique, l'horloge. Les circuits asynchrones définissent une classe plus large de circuits, dont le contrôle ou séquencement est assuré par tout autre méthode que le recours à un signal périodique globalement distribué.

La suite de ce paragraphe présente les concepts de bases qui permettent la conception de cette classe de circuit. On présentera le principe de fonctionnement asynchrone, les caractéristiques de ces circuits, et plus particulièrement les protocoles de communications et

le codage des données qui constituent le moyen de contrôler ces circuits sans faire appel à un mécanisme de signalisation global.

1.1.1. Le mode de fonctionnement asynchrone

Ce paragraphe est destiné à clarifier l'utilisation du terme « asynchrone » dans le contexte de la conception de circuits numériques. Cela nous permet d'introduire le mode de fonctionnement asynchrone et ses différences avec le mode de fonctionnement synchrone.

« Asynchrone » signifie qu'il n'existe pas de relation temporelle à priori entre des évènements. Dans un système intégré, ces évènements sont des évènements au sens large (contrôle ou données) implémentés par des signaux électriques. Il faut donc définir ce qu'est un signal « asynchrone ».

Si on prend l'exemple d'un signal d'interruption appliqué à un microprocesseur (que celui-ci soit synchrone ou asynchrone), on qualifie le signal d'interruption d'asynchrone par rapport au fonctionnement du microprocesseur. Cette situation est délicate à résoudre puisqu'il faut échantillonner un signal pour mesurer son niveau sous contrôle d'un événement (par exemple l'horloge du processeur) qui n'a aucune relation temporelle avec ce signal extérieur. Dans cet exemple, le mot asynchrone qualifie une indétermination sur la relation d'ordre entre le signal d'interruption et le signal d'horloge, et donc sur le niveau de ce premier.

Quand on parle de circuits asynchrones, on qualifie des circuits qui gèrent des signaux asynchrones entre eux mais dont le comportement est parfaitement déterminé. Par exemple supposons deux signaux qui transportent de l'information sous forme de changement de niveau. La spécification est telle qu'il n'existe pas à priori de relation de causalité entre ces deux signaux, ces signaux sont donc asynchrones. Cependant, il est garanti qu'un évènement doit se produire sur ces deux signaux. Dans ce système, il y a indétermination sur les instants d'occurrence des événements de chaque signal, mais le fait que les événements aient lieu est absolument déterminé. La porte de Muller présentée au paragraphe 3.1 implémente naturellement le rendez-vous entre deux événements asynchrones. De manière générale, on parle de synchronisation d'évènements : un évènement est généré en sortie si et seulement si il y a événement sur les deux entrées de la porte, quelque soit les instants d'occurrence.

Les circuits asynchrones fonctionnent donc avec la seule connaissance de l'occurrence des évènements, sans connaissance de l'ordre. Le fonctionnement est similaire à celui des systèmes flot de données. On peut ainsi spécifier l'enchaînement des évènements sous forme d'un graphe de dépendances (réseau de Pétri par exemple). L'évolution du système est garantie par l'évolution conjointe, possiblement concurrente, des éléments qui le compose. Chaque élément évolue avec les seules informations reçues des éléments auxquels il est connecté. Les règles d'activation sont similaires à celles des réseaux de Pétri : une place est activée si toutes les entrées sont marquées, l'exécution a lieu si toutes les sorties ne sont pas marquées, après exécution, les sorties sont marquées, les marques des entrées sont supprimées. L'analogie avec le modèle des processus séquentiels communicants [HOAR 78] est aussi très forte. Dans ce modèle, les processus se synchronisent par passage de messages via des canaux de communication. Pour échanger des informations, deux éléments doivent se synchroniser, ils échangent leurs informations à travers des canaux de communication puis peuvent continuer leur flot d'exécution de manière indépendante. Le langage CHP que nous présentons dans le chapitre 2 est directement issu de ce modèle [MART 90].

Ainsi dans les circuits asynchrones, la seule connaissance de l'occurrence des événements est suffisante pour implémenter la synchronisation, il n'est pas nécessaire d'introduire de mécanisme global d'activation du système. C'est effectivement la différence avec les

systèmes synchrones. Dans un système synchrone, tous les éléments évoluent ensemble lors de l'occurrence d'un évènement sur le signal d'horloge : l'exécution de tous les éléments se trouve synchronisée. Ce mécanisme de synchronisation introduit une contrainte temporelle globale : afin d'obtenir un fonctionnement correct, tous les éléments doivent respecter un temps d'exécution maximum imposé par la fréquence du mécanisme d'activation. A l'opposé, les systèmes asynchrones évoluent de manière localement synchronisée, le déclenchement des actions dépend uniquement de la présence des données à traiter. Ainsi, la correction fonctionnelle est indépendante de la durée d'exécution des éléments du système.

1.1.2. Un contrôle local

Comme nous l'avons présenté, le point fondamental du mode de fonctionnement asynchrone est que la synchronisation et le transfert d'informations sont effectués localement. Ceci est effectué par une signalisation adéquate. Le contrôle local doit remplir les fonctions suivantes : être à l'écoute des communications entrantes, déclencher le traitement localement si toutes les informations sont disponibles et produire des valeurs sur les sorties. De plus, pour que l'opérateur sache qu'il est autorisé à émettre de nouvelles valeurs sur ses sorties, il doit être informé que les valeurs qu'il émet sont bien consommées (reçues) par l'opérateur en aval. Ainsi, pour permettre un fonctionnement correct indépendamment du temps, le contrôle local doit implémenter une signalisation bidirectionnelle. Les communications sont dites à poignées de mains ou de type requête-acquittement (Figure 1-1).

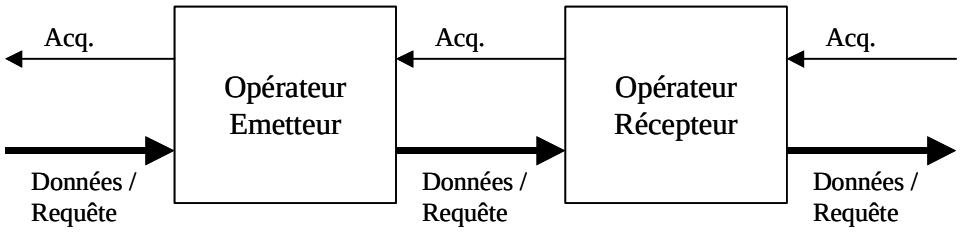


Figure 1-1 : Communication de type requête / acquittement entre opérateurs asynchrones

Toute action doit être acquittée par le récepteur afin que l'émetteur puisse émettre à nouveau. Cette signalisation bidirectionnelle offre un mécanisme qui permet de garantir la synchronisation et la causalité des évènements au niveau local et donc la correction fonctionnelle du système dans son ensemble.

1.1.2.1. Protocoles de communications

Pour implémenter une signalisation bidirectionnelle, deux types de protocoles sont couramment utilisés : le protocole deux phases, encore appelé NRZ (Non Retour à Zéro) ou « *half-handshake* » et le protocole quatre phases, encore appelé RZ (Retour à Zéro) ou « *full-handshake* ». Ils sont décrits respectivement dans les Figure 1-2 et Figure 1-3. Dans les deux cas, il faut noter que tout événement sur un signal par l'émetteur est acquitté par un événement sur un signal du récepteur, et vice-versa. Ce mécanisme permet de s'assurer de l'insensibilité au temps de traitement dans l'opérateur. Un changement des données est acquitté par le signal d'acquittement, un changement du signal d'acquittement est acquitté par un changement des données, et ainsi de suite pour l'échange de données suivant. Ainsi seul

importe l'occurrence des évènements localement entre l'émetteur et le récepteur, et non leur temps relatif, ni leur ordre respectif par rapport à l'entrée de l'émetteur ou la sortie du récepteur.

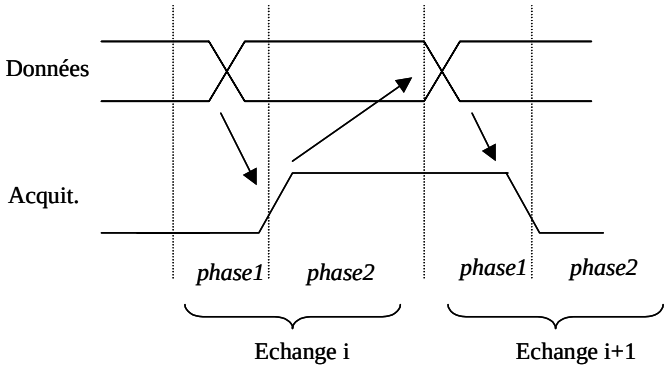


Figure 1-2 : Principe du protocole deux phases

Pour le protocole deux phases (Figure 1-2), on observe les phases suivantes :

- Phase 1 : c'est la phase active du récepteur qui détecte la présence de nouvelles données, effectue le traitement et génère le signal d'acquittement.
- Phase 2 : c'est la phase active de l'émetteur qui détecte le signal d'acquittement et émet de nouvelles données si disponibles.

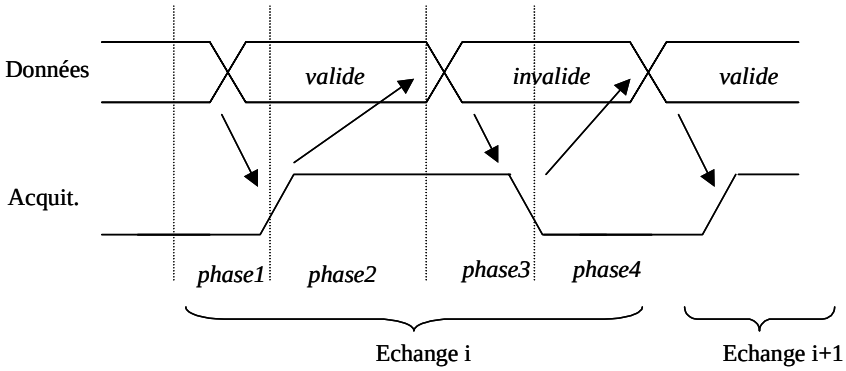


Figure 1-3 : Principe du protocole quatre phases

Pour le protocole quatre phases (Figure 1-3), on observe les phases suivantes :

- Phase 1 : le récepteur détecte la présence de nouvelles données, effectue le traitement et génère le signal d'acquittement.
- Phase 2 : l'émetteur détecte le signal d'acquittement, et invalide les données (retour à zéro)
- Phase 3 : le récepteur détecte le passage des données dans l'état invalide et place le signal d'acquittement dans son état initial
- Phase 4 : l'émetteur détecte la remise à zéro de l'acquittement, il est autorisé à émettre de nouvelles données si disponibles.

Ces deux protocoles implémentent un protocole de type requête-acquittement. Nous n'avons pas pour l'instant précisé de relation entre requête et données dans ces protocoles. Nous verrons dans le paragraphe suivant que cela dépendant du codage des données.

A ce stade de la présentation, il est difficile de conclure sur la pertinence de l'un ou de l'autre protocole. Le protocole quatre phases requiert deux fois plus de transitions que le protocole deux phases, il serait à priori plus lent et plus consommant. Toutefois, les techniques d'optimisation du pipeline asynchrone (paragraphe 6.4) permettent de cacher la pénalité de la phase de remise à zéro du protocole quatre-phases. De plus, le protocole deux phases requiert un matériel souvent plus important que le protocole quatre phases car il doit détecter des transitions, et non des changement de niveaux [SUTH 89]. Ainsi, même si le protocole deux phases contient moins de transitions, cela est compensé par la complexité du matériel.

De manière générale, le protocole quatre phases est majoritairement utilisé pour implémenter le cœur logique des circuits asynchrones, il a permis de réaliser les circuits les plus rapides. On peut citer l'expérience de l'Amulet. La première version Amulet1 [WOOD 97] a été conçue en deux phases puis suivi de l'Amulet2 [FURB 99] conçu en quatre phases. Cependant, lorsque les signaux doivent traverser des éléments avec des délais importants, comme les plots par exemple, un protocole deux phases est efficace en performance (cf. chapitre 4, paragraphe 2).

1.1.2.2. Codage des données

Pour ces deux types de protocole, nous n'avons pas précisé comment détecter la présence d'une donnée et comment générer un signal qui indique une fin de traitement. La réponse réside dans l'adoption d'un codage particulier pour les données.

Il est en effet impossible d'utiliser un unique fil par bit de donnée : cela ne permet pas de détecter que la nouvelle donnée prend un état identique à la donnée précédente. Deux types de solution sont possibles, soit la création d'un signal de requête associé aux données, soit l'utilisation d'un codage insensible aux délais bifilaire ou double-rail par bit de donnée [VERH 88].

- **Codage double-rail**

Avec deux fils par bit de données, quatre états sont disponibles pour exprimer les valeurs logiques '0' et '1'. Deux codages sont couramment utilisés, l'un utilisant trois états seulement et l'autre utilisant les quatre états (Figure 1-4).

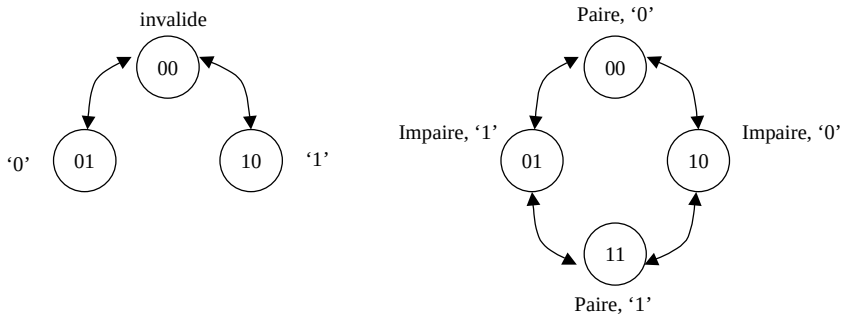


Figure 1-4 : Diagramme de transition d'états des codages trois états et quatre états double-rail.

Pour le codage trois états, un fil encode la valeur '0', l'autre la valeur '1'. Lorsque les deux fils sont à 0, il n'y a pas de valeur valide. Le fil 0 à 1 indique la valeur '0', le fil 1 à 1 indique

la valeur ‘1’. L’état 11 est interdit. Ainsi, chaque changement de valeur valide implique de repasser par l’état invalide 00.

Ce codage garantit que le passage d’un état à un autre se fait toujours par changement de l’état d’un seul fil sur les deux, ce qui est détectable sans aléa. Ce codage est parfaitement adapté au protocole quatre phases où les données changent deux fois d’états, d’un état valide à un état invalide. Enfin, le signal de fin de calcul d’un opérateur peut facilement être généré en détectant qu’un des bits de sortie est passé à 1.

Pour le codage quatre états, les valeurs ‘0’ et ‘1’ sont codées avec deux combinaisons. Par bit, l’une des combinaisons est considérée paire, l’autre impaire. Chaque fois qu’une donnée est émise on change sa parité. Ce codage permet de passer d’une valeur à une autre sans passer par un état invalide, ce qui est parfaitement adapté au protocole deux phases. Tout comme un code de Gray, le passage d’un état à un autre ne change qu’un seul bit du code, ce qui se détecte sans aléa [MCAU92]. L’analyse de la parité permet alors de détecter la présence d’une nouvelle donnée pour générer un signal de fin de calcul.

Pour ces codages trois et quatre états, l’information de validité des données est directement détectable dans le codage double-rail de la donnée. Il n’y a donc pas besoin de signal de requête explicite pour préciser qu’une donnée est valide et donc implémenter le protocole. Ce type de codage permet d’implémenter des circuits avec peu d’hypothèses temporelles (cf. paragraphe 1.3.1)

Quoique élégante, l’implémentation du codage quatre états est plus complexe, plus lente et consomme plus que celle du codage trois états. En effet, le codage quatre états nécessite d’implémenter une petite machine à états dans chaque cellule pour mémoriser l’état courant, générer le code de Gray et détecter le changement de parité. Plusieurs logiques ont été proposées pour implémenter le codage trois états dans les opérateurs asynchrones. Une comparaison de ces logiques est proposée dans [NIEL 94]. [ELHA95] propose également une étude approfondie de la logique cascode différentielle (DCVSL) qui constitue l’implémentation la plus efficace du codage trois états.

- **Codage données groupées**

Le codage double-rail est coûteux en complexité car double le nombre de fils par rapport aux réalisations synchrones. Afin de réduire ce coût et simplifier la détection d’une donnée multi-bit, il est aussi possible de séparer l’information de contrôle de l’information de donnée proprement dite. On crée explicitement un signal de contrôle unique, appelé « requête », qui s’apparente à un signal d’horloge local. Ce signal est utilisé pour déclencher la mémorisation ou le traitement des données associées. Cette technique est appelée « *Bundled-Data* » ou « données groupées » [SUTH 89]. Ceci permet d’encoder chaque bit de donnée avec un unique signal mais au prix d’une hypothèse temporelle dans le récepteur afin de garantir que l’événement sur le signal de requête succède la disponibilité des données associées.

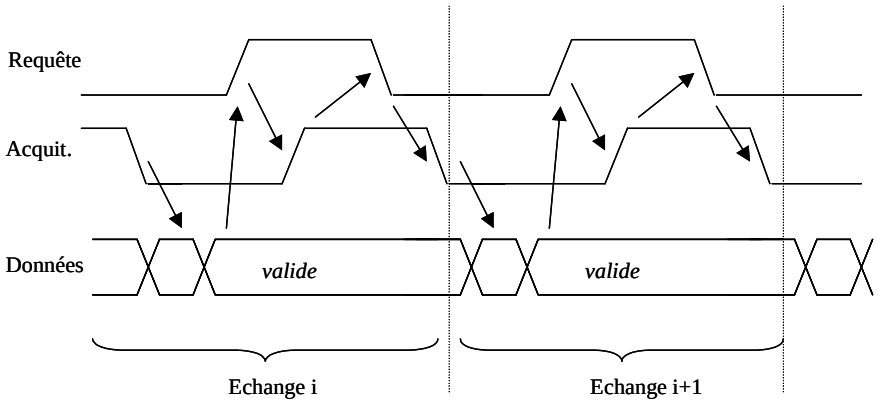


Figure 1-5 : Protocole 4 phases « données groupées »

La Figure 1-5 montre l'exemple d'un protocole quatre phases données groupées. Ce type de codage où on sépare l'information de requête des données peut aussi bien s'adapter à un protocole deux phases. Dans ce cas, chaque transition sur le signal de requête indique une nouvelle valeur sur les données.

1.1.3. Caractéristiques des opérateurs asynchrones

D'une manière générale, un opérateur asynchrone peut être considéré comme une cellule réalisant une fonction et communiquant avec son environnement avec des canaux de communications banalisés. Ces canaux de communications correspondent aux protocoles de type requête acquittement que nous venons de présenter. Le point clé est que ces canaux de communications servent non seulement à échanger des données mais aussi à synchroniser les éléments entre eux. Ainsi, l'opérateur peut aussi bien être une fonction au niveau bit, au niveau arithmétique ou même un algorithme complexe, tout en utilisant la même sémantique de communication. De même, le protocole de communication étant respecté, l'opérateur asynchrone peut être à mémoire ou combinatoire.

Par son implémentation, un opérateur asynchrone possède une sémantique de synchronisation plus riche qu'un opérateur synchrone. Il peut être caractérisé de manière générale par les quatre paramètres suivants :

- **Latence**

Le temps de latence correspond à la chaîne combinatoire la plus longue pour qu'une sortie soit l'image fonctionnelle d'une entrée. Il est important de préciser que cette caractéristique ne dépend pas du fait que l'opérateur soit combinatoire ou à mémoire. En effet, en asynchrone, un opérateur de mémorisation inoccupé se comporte comme une cellule combinatoire qui laisse traverser la donnée. Il n'y a pas besoin d'attendre un signal de synchronisation global pour faire avancer la donnée dans l'étage suivant. Ainsi, la vitesse maximale de fonctionnement d'un opérateur est toujours mesurée à vide, c'est à dire lorsque toutes les ressources de mémorisation sont inoccupées. De plus, le temps de traversé de la logique peut être variable en fonction des données d'entrée. Cette variation dépend de l'algorithme et de l'implémentation choisie. Pour être précis, on pourra ainsi déterminer une latence minimale, moyenne et maximale de l'opérateur.

- **Temps de cycle**

Le temps de cycle caractérise le temps minimum qui sépare l'acceptation de deux informations en entrée de l'opérateur. En général, ceci correspond au temps pour échanger une donnée entre deux ressources de mémorisation successives. Ce paramètre caractérise la bande passante de l'opérateur. Si l'opérateur intègre plusieurs étages avec mémorisation, les temps de cycle entre chaque étage peuvent être bien sûr différents. Ainsi, le temps de cycle vu de l'entrée de l'opérateur peut varier. Il est au minimum égal au temps de cycle de l'étage d'entrée de l'opérateur et au maximum égal au temps de cycle de l'étage le plus lent de l'opérateur. Ainsi, le temps de cycle de l'opérateur, vue de l'entrée, dépend de l'occupation des ressources internes de l'opérateur. Si celui-ci est utilisé sporadiquement, le temps de cycle est celui de l'entrée, si il est utilisé en continu, le temps de cycle se cale sur le temps de cycle de l'opérateur le moins performant.

- **Profondeur de pipeline**

La profondeur de pipeline d'un opérateur asynchrone définit le nombre maximum de données ou informations que l'opérateur peut mémoriser. Cela correspond à la différence maximale entre l'indice de la donnée présente à la sortie et l'indice de la donnée présente à l'entrée, lorsque toutes les ressources de mémorisation de l'opérateur sont occupées. La même définition est utilisée pour caractériser un pipeline synchrone. Cependant dans le cas asynchrone, le nombre de données présentes dans le pipeline n'est pas imposée et peut varier dynamiquement en cours d'exécution. C'est pourquoi il faut définir une profondeur de pipeline maximale.

- **Protocole de communication**

Afin qu'un opérateur asynchrone puisse échanger des informations avec son environnement, il est nécessaire de caractériser à ses interfaces le protocole de communication utilisé et le codage de donnée associé. Il est important de spécifier le protocole utilisé au niveau de chaque cellule pour pouvoir les connecter entre elles. Plusieurs protocoles peuvent être indifféremment utilisés dans un même design. La caractéristique fondamentale et commune à tout les protocoles d'échange asynchrone utilisé aujourd'hui est qu'ils assurent la détection de la présence d'une donnée en entrée, assurent la génération d'une signalisation indiquant d'une part que la donnée a été consommée en entrée et d'autre part qu'une information est disponible en sortie. Cette signalisation bidirectionnelle entre opérateurs asynchrones permet l'implémentation de circuits fonctionnant sur un schéma « flot de données ».

1.1.4. Conclusion

Les concepts de base que nous venons de présenter sont utilisés couramment dans les systèmes informatiques depuis des années : VME, PCI, etc. La communication « asynchrone » est largement répandue pour interfacer dans un système complet les macro-composants entre eux : mémoires, processeurs, périphériques, etc. Il s'agit toujours de protocoles de type quatre-phase données-groupées.

Cependant, ce qui différencie ces protocoles niveaux systèmes de ceux utilisés pour la conception de circuits asynchrones est l'échantillonnage des signaux de contrôle par une horloge. Les signaux de requête et d'acquittement doivent respecter des temps de pré-

positionnement et de maintien relativement à l'horloge. Ces systèmes pourraient ainsi être qualifiés de « pseudo-asynchrones ». Ceci pose d'ailleurs des difficultés lorsque le système est constitué de différentes horloges. Il faut dans ce cas avoir recours à des éléments réellement asynchrones comme des arbitres. Ce type de problème est aussi bien présent dans les systèmes non intégrés que dans les systèmes intégrés.

Ainsi, dans la conception de circuits intégrés asynchrones, on se permet l'utilisation et la mise en œuvre des concepts de communication et de protocole au niveau le plus fin de la réalisation des circuits. Comme on l'a vu, les opérateurs asynchrones sont caractérisés par un nombre de paramètres plus important que les opérateurs synchrones, tels que latence, temps de cycle, profondeur de pipeline. Ces paramètres existent dans le cas synchrone mais n'ont pas besoins d'être considérés de la même manière (cf. paragraphe 6.4). L'approche « pire cas » synchrone combinée à un protocole de communication simplifié ne permet pas de tirer parti de ces paramètres. Le mode de fonctionnement asynchrone apparaît donc plus riche que le mode synchrone. C'est la complexité du protocole de communication et la puissance de synchronisation qu'il possède qui permettent d'exploiter toutes les finesses d'exécution de la logique : variabilité de la latence suivant les données, variabilité du temps de cycle suivant le remplissage du pipeline, variabilité du nombre d'éléments dans le pipeline. L'approche pire cas synchrone permet de concevoir un système en manipulant des vues externes simples, voire réductrices des opérateurs, cela peut se réduire à une bande passante et à un taux de pipeline. Il n'est alors pas nécessaire de considérer les schémas d'exécutions internes qui sortent du cadre de la chaîne critique.

Cette différence de propriété n'est bien sûr pas suffisante pour pouvoir conclure sur la pertinence de l'approche asynchrone par rapport à l'approche synchrone. Cependant, on peut déjà imaginer que le spectre d'architecture sera élargi par la prise en compte de ces nouveaux paramètres [RENA 97]. Le paragraphe suivant présente le lien entre les propriétés du mode de fonctionnement asynchrone et les potentiels que l'on peut en attendre pour concevoir des circuits et systèmes intégrés.

1.2. Avantages et potentiels des circuits asynchrones

1.2.1. Calcul en temps minimum, en temps moyen

Une première conséquence du mode de fonctionnement asynchrone de type flot de données est qu'un opérateur peut évaluer une fonction en un temps variable, compris entre une borne inférieure et une borne supérieure. Ce temps correspond au temps de traversée des données de l'entrée vers la sortie de l'opérateur, c'est la latence directe. Ce temps de traversée de l'opérateur peut aussi posséder des variations en fonction du chemin utilisé par les données et donc en fonction de leurs propres valeurs.

Comme l'opérateur asynchrone implémente un protocole de communication, la donnée est signalée dès que possible et donc utilisée à sa sortie par l'opérateur suivant au plus tôt. Ainsi, de manière naturelle les circuits asynchrones possèdent des variabilités de temps de calcul qui sont exploitables grâce aux mécanismes de synchronisation locale. Le calcul s'effectue au plus tôt, suivant la fonction à évaluer, suivant les données, suivant le chemin électrique parcouru. Les variations de temps de calcul et les gains en performance qu'on peut en attendre seront d'autant plus grands que l'algorithme implémenté exhibe des dépendances fonctionnelles importantes en fonction des données.

De plus, les caractéristiques de vitesse d'un circuit dépendent de paramètres physiques qui influencent le fonctionnement des dispositifs élémentaires, tels que variations des paramètres technologiques, température ou tension d'alimentation. Le fonctionnement flot de données des circuits asynchrones les rend très robustes vis-à-vis de ces variations. Comme toute fin de traitement est détectée et signalée au niveau de chaque cellule, les variations de vitesse induites par des modifications des paramètres physiques ne modifient pas le comportement fonctionnel mais seulement les performances. Les circuits asynchrones fonctionnent à la vitesse maximale permise par les dispositifs élémentaires et les conditions de fonctionnement.

D'une manière générale, l'approche asynchrone favorisera plutôt une optimisation du temps de calcul moyen plutôt que du temps de calcul pire cas. Contrairement au cas synchrone, les performances globales d'un système ne sont pas contraintes par les performances pire cas d'un bloc localement sous utilisé. Le concepteur doit porter son effort de conception vers l'optimisation des temps de calcul moyen des blocs en moyenne les plus utilisés. Pour un opérateur asynchrone, si on peut facilement caractériser sa latence minimale et sa latence maximale en recherchant les chaînes qui minimisent/maximisent les dépendances fonctionnelles et les temps de transfert dans les dispositifs, il est plus difficile d'évaluer sa latence moyenne. Ceci nécessite de modéliser les propriétés statistiques des entrées et les dépendances fonctionnelles de l'algorithme. C'est possible analytiquement dans certain cas. Néanmoins, même si en général le temps moyen est difficilement caractérisable, peut-on rêver de meilleur mode de fonctionnement ? Le temps de traversée sera toujours le temps le plus court pour réaliser la fonction demandée, étant donnée le chemin emprunté par les données, la vitesse des dispositifs élémentaires et les conditions de fonctionnement, alors que la correction fonctionnelle reste garantie.

1.2.2. Distribution du contrôle, un pipeline élastique

Tout comme en synchrone, la notion de pipeline signifie en asynchrone qu'un opérateur effectue en parallèle des tâches différentes sur des données successives distinctes. La notion de pipeline élastique correspond au fait qu'en asynchrone un pipeline possède une capacité variable. En effet, grâce aux mécanismes de synchronisation locale des protocoles asynchrones, les registres de pipeline se comportent comme une pile de type « Fifo ». Ainsi, les données progressent dans le pipeline aussi longtemps qu'elles ne rencontrent pas de ressources occupées, et ceci de manière indépendante des données qui les suivent. Au contraire en synchrone, c'est l'occurrence d'un front d'horloge sur tous les registres de pipeline qui déclenche le déplacement des données. Dans ce cas, l'horloge impose une synchronisation relative des données entre elles : une fois entrées dans le pipeline, deux données sont toujours séparées du même nombre d'étages. En asynchrone, cela n'est pas nécessairement le cas, et nous verrons que cela peut être exploité au profit de la vitesse et de la consommation (chapitre 5).

1.2.3. Un support fiable pour les traitement non-déterministes

Dans un circuit synchrone, il est délicat de traiter un signal asynchrone de l'horloge. Comme nous le disions dans le paragraphe 1.1, l'exemple typique d'un tel signal asynchrone est le traitement d'un signal externe d'interruption dans un microprocesseur. L'échantillonnage d'un signal asynchrone peut faire apparaître un état métastable dont la durée est indéterminée, non bornée [KLEE 87]. Comme les circuits synchrones imposent à tous les signaux un temps d'établissement borné compatible avec la période d'horloge, ils ne

permettent pas d'assurer un traitement de ces signaux asynchrones d'une manière correcte dans 100% des cas. Généralement des mécanismes de division d'horloge sont mis en œuvre pour limiter la probabilité d'apparition d'états métastables.

Au contraire dans le cas asynchrone, il est possible d'attendre autant que nécessaire la fin de l'état métastable car la correction fonctionnelle ne dépend pas des temps de traitement. On peut alors utiliser des montages qui permettent d'assurer l'établissement d'un état stable, sans se préoccuper de la durée de l'état métastable intermédiaire. Les cellules présentées dans le chapitre 3 paragraphe 3 permettent d'implémenter ces traitements. Ainsi, il est possible d'arbitrer sans risque d'erreur deux signaux de requête s'adressant à une ressource unique. Il est aussi possible d'échantillonner de façon fiable l'occurrence d'un signal externe comme un signal d'interruption pour le synchroniser avec l'exécution en cours.

1.2.4. Absence d'horloge

L'avantage évident des circuits asynchrones, en raison de l'absence d'horloge globale, est que tous les problèmes de conception liés à la manipulation d'horloges sont supprimés. Les problèmes d'horloge sont dans les technologies actuelles de plus en plus présents car ces technologies autorisent la conception de circuits de plus en plus complexes et de plus en plus rapides. Aujourd'hui, la conception des circuits d'horloge est devenue une question de toute première importance puisqu'ils peuvent directement limiter les performances du système synchrone. Le processeur Alpha de DEC/Intel est conçu avec un chemin critique de moins de 2 ns, une gigue d'horloge (clock skew) de plusieurs centaines de picosecondes est inacceptable puisque représenterait plus de 10% du temps de cycle. Le problème de conception des circuits d'horloge est bien sûr le plus critique pour les circuits à très haute performance. Néanmoins, de manière générale, l'approche synchrone a des conséquences à tous les niveaux du flot de conception : estimation des capacités d'interconnexions avant la phase de synthèse logique, caractérisations des timings des éléments de bibliothèques – en particulier les temps de setup et de hold -, estimation de timing après extraction sur layout, confiance dans la caractérisation de la technologie, conception électrique et placement physique des arbres d'horloge pour réduire les problèmes de gigues, génération de pattern de test afin de tester / valider les chemins critiques et trier les circuits après fabrication en fonction de leurs performances. Les techniques et outils de conception de circuits synchrones évoluent avec les technologies pour estimer de plus en plus précisément les temps d'arrivée de l'horloge sur les bascules d'un circuit.

Au contraire les circuits asynchrones n'utilisent pas d'horloge globale. Les éléments de synchronisation sont distribués dans l'ensemble du circuit, leur conception est plus facile à maîtriser. On peut penser en particulier au problème d'interconnexion et de synchronisation entre blocs dans un système tout intégré (system-on-chip). Comme pour certains circuits asynchrones, le fonctionnement est indépendant des retards qui peuvent être introduits sur les fils, le problème de gigue d'horloge est inexistant et le temps d'arrivée d'un signal d'un bout à l'autre du circuit n'a pas d'importance sur la correction fonctionnelle du système. Le principe de synchronisation locale des circuits asynchrones constitue alors directement un outil d'aide à la conception de systèmes complexes.

1.2.5. Faible consommation

Par rapport aux circuits synchrones, plusieurs facteurs de réduction de la consommation sont à prendre en compte. La première conséquence importante de la suppression de l'horloge

est la suppression de la consommation associée à celle-ci. Dans les circuits rapides, la consommation de l'horloge et des éléments de mémorisation peut représenter jusqu'à plus de 50% de la consommation du circuit. Cette consommation est due au chargement des circuits d'horloges et aux transitions dans les bascules, celle-ci est donc supprimée. De plus, dans les circuits numériques, une part non négligeable de la consommation provient de transitions non utiles en raison de la présence d'aléas dans les blocs de logiques combinatoires. En synchrone, ces aléas ne sont pas gênant fonctionnellement car ils doivent avoir disparus à l'arrivée du prochain front d'horloge, cependant ils représentent une consommation relativement importante. En asynchrone, la conception doit supprimer tout type d'aléas afin d'obtenir des circuits fonctionnellement corrects (cf. paragraphe 1.3). La part de consommation due aux aléas est donc supprimée dans les circuits asynchrones.

Un autre aspect important de réduction de la consommation concerne la mise en veille de la logique asynchrone à tout niveau de granularité. Dans un circuit synchrone, tous les blocs de logiques se trouvent alimentés en données et commutent tous au moment de l'arrivée de l'horloge, que ce soit des transitions utiles dans le blocs ou non. Par exemple dans un microprocesseur, la plupart des blocs n'ont pas besoin de travailler pour toutes les instructions : le multiplieur est utile pour une multiplication, un additionneur pour une addition et non le contraire. Comme le mécanisme de synchronisation à horloge est simplifié, tous les blocs consomment alors que fonctionnellement ils peuvent ne rien apporter. Au contraire, grâce au mécanisme de synchronisation locale des circuits asynchrones et à leur fonctionnement flot de données, seuls les blocs utiles reçoivent des données et le contrôle associé et donc consomment. Cet argument de réduction de la consommation en faveur de l'asynchrone est donc particulièrement intéressant dans le cas d'architectures irrégulières. Il est vrai que les concepteurs de circuits synchrones tentent de tirer parti de ces irrégularités fonctionnelles pour réduire la consommation en implémentant des mécanismes de contrôle sur l'horloge (« *gated-clock* »). Ceci est relativement complexe à mettre en œuvre au niveau fonctionnel et surtout au niveau électrique dans le cas des circuits rapides car cela nécessite d'insérer de la logique sur les arbres d'horloges alors qu'on cherche à optimiser ses caractéristiques de gigue. Ces optimisations sont possibles dans le cas synchrone mais difficiles à mettre en œuvre alors que la propriété de mise en veille de la logique à tous les niveaux de granularité est gratuite dans le cas asynchrone.

Une conséquence et un autre avantage de cette activité conditionnelle de la logique est que le redémarrage de la logique est aussi instantané. La gestion de la consommation est donc gérée par le matériel à tous les niveaux de l'architecture : il n'est pas nécessaire de concevoir un logiciel parfois complexe pour contrôler l'activation et la désactivation de tout ou partie du système.

Une dernière propriété intéressante des circuits asynchrones pour la faible consommation est leur robustesse et leur adaptation aux conditions de fonctionnement. Comme la puissance varie avec le carré de la tension, il est aisé de réduire la tension d'alimentation pour limiter la puissance consommée. Cette réduction de la tension d'alimentation des circuits asynchrones peut se faire avec un matériel et un temps de conception minimum. Contrairement au cas synchrone, il n'y a pas besoin d'adapter et de caractériser la fréquence d'horloge aux différentes conditions de fonctionnement car la correction fonctionnelle est garantie quelques soient les délais dans les cellules élémentaires. De manière statique, on peut ainsi aisément échanger performance contre faible consommation dans un circuit asynchrone : le circuit sera d'autant moins consommant à tension réduite qu'il est performant à tension nominale.

De plus, il est possible de faire varier dynamiquement la tension d'alimentation du circuit afin de réduire la consommation. La Figure 1-6 présente le système de traitement proposé dans [NIEL 94]. Ce système de correction d'erreur mis au point par Philips [BERK 94] évalue la charge de calcul qu'il doit effectuer à l'aide d'un indicateur de remplissage de la pile Fifo qui mémorise en entrée les données à traiter. Si la pile est vide, le calcul est trop rapide, le système se régule en diminuant la tension d'alimentation ce qui diminue sa consommation. Respectivement, si pile est pleine, le calcul est effectué trop lentement, le système augmente sa tension d'alimentation de façon à délivrer la puissance de calcul nécessaire afin de traiter les données en temps réel. La consommation est donc régulée dynamiquement en fonction de la quantité d'information à traiter : le traitement est effectué à consommation minimale suivant le taux d'erreur présent dans le flux de données entrant. L'algorithme de correction d'erreur utilisé est qualifié de paresseux : sa complexité est faible lorsqu'il n'y a pas d'erreur. Ainsi contrairement au cas synchrone, l'approche algorithmique de ce système consiste à favoriser les irrégularités plutôt que d'essayer de régulariser le traitement afin de l'adapter à un critère pire cas. Cette approche algorithmique est donc tout à fait différente du cas synchrone, celle-ci est rendue possible grâce à la robustesse et à l'auto adaptation de la logique asynchrone en fonction des conditions de fonctionnement.

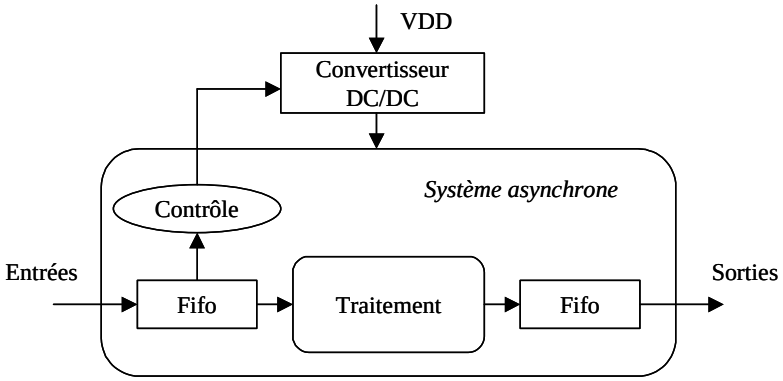


Figure 1-6 : Régulation dynamique de la tension d'alimentation en fonction de la charge de traitement.

Malgré toutes ces propriétés intéressantes, il est difficile de conclure de façon certaine sur la faible consommation des circuits asynchrones. En effet, en raison des mécanismes de synchronisation locale, les circuits asynchrones coûtent chers à implémenter. Les protocoles asynchrones représentent plus de transitions que le mécanisme de synchronisation à horloge. Même si la suppression de l'horloge permet de supprimer toute consommation associée à celle-ci, il est difficile aujourd'hui de conclure de manière absolue. Néanmoins, la plupart des études ont montrées que des gains satisfaisants en consommation étaient possibles [GAGE 98], [GARS 99].

1.2.6. Faible bruit et système radio-fréquence

L'autre conséquence de la suppression d'horloge et de la distribution du contrôle dans la structure du circuit est que les problèmes de pics de consommation sont inexistants. En effet, l'activité électrique d'un circuit asynchrone est bien mieux répartie dans le temps que pour un circuit synchrone. Il n'y a pas d'instant prédéfini pour activer un opérateur comme c'est le cas avec les fronts d'horloges. Ainsi, la consommation dans les lignes du circuit est distribuée

dans le temps, que ce soit au niveau de la logique qu'au niveau des éléments de mémorisation. De plus, la consommation dans les circuits d'horloge est bien sûr supprimée. Cette propriété intrinsèque des circuits asynchrones permet de limiter le bruit dans les lignes d'alimentation, et ainsi de réduire la puissance des ondes électromagnétiques émises par le circuit par rapport au cas synchrone. Ceci est donc particulièrement intéressant pour des circuits qui possèdent des contraintes fortes de compatibilité électromagnétique afin d'être utilisés dans des environnements spécifiques.

Cette propriété est à l'origine de la commercialisation par Philips du premier circuit asynchrone, un microcontrôleur 8-bit compatible 80c51 [GAGE 98]. Celui-ci a été intégré dans un pager grand public, Myna™ (cf. paragraphe 1.4.1). Pour cette application, le faible rayonnement électromagnétique des circuits asynchrones a permis la conception d'un système intégré mixte numérique radio-fréquence.

En effet, l'évolution des technologies conduit à intégrer sur un même substrat un « front-end » analogique radio-fréquence et des traitements numériques sans avoir recours à une fréquence intermédiaire. Ainsi, le système numérique synchrone fonctionnant à des fréquences élevées constitue une source de bruit très importante, via les alimentations et le substrat, pour les parties analogiques extrêmement sensibles à ces paramètres. Les circuits asynchrones présentent alors une alternative sérieuse pour concevoir un système numérique sous contrainte de limitation de bruit. On peut même imaginer d'orienter la conception des circuits asynchrones afin de réduire d'autant plus le bruit : limiter le courant consommé, limiter la dérivée de ce courant ; ceci étant possible en contrôlant les signaux locaux de requête-acquittement.

Dans cette optique, nous avons conçu à France Telecom R&D Meylan, en collaboration avec TIMA, un circuit pour carte sans contact (cf. conclusion du manuscrit). Ce circuit intègre l'ensemble du système de réception, d'alimentation avec une self intégrée sur le circuit et de traitement numérique [ABRI 00]. Le fait que le microcontrôleur soit asynchrone permet de le faire fonctionner pendant les phases de réception / émission, ce qui n'était pas possible jusqu'alors avec des processeurs synchrones. La conception du circuit, ainsi que la partie software est rendue plus simple car le processeur est robuste vis-à-vis des variations de tension, et le programme n'a pas besoin d'être interrompu puis redémarré pendant que la partie radio-fréquence est en cours d'émission / réception. La société Philips a développé un système similaire basé sur son microcontrôleur 80c51 [KESS 00].

1.2.7. Variation des temps de propagation, Migration

De manière générale, les circuits asynchrones sont robustes par rapport aux performances des dispositifs élémentaires et aux conditions de fonctionnement. Cela est intéressant comme on l'a vu dans le paragraphe 1.2.5 pour échanger aisément performance du système contre sa consommation en adaptant la tension d'alimentation. Cette propriété devient aussi un avantage prépondérant pour les technologies avancées où les dispersions physiques des dispositifs élémentaires sont de plus en plus importantes. Ainsi aujourd'hui, tous les facteurs qui induisent des variations de temps de propagation dans les dispositifs et les interconnexions posent problème pour la conception des circuits synchrones. Dans les technologies futures, tous les Roadmap (technologies et vendeurs de CAO) identifient les points suivants comme facteurs de variation de temps de propagation des signaux : gradient de température dans les composants, appels de courant dans les réseaux d'alimentation, le couplage des lignes d'interconnexions (Cross-talk). Le style de conception asynchrone

apporte ainsi une solution à tous les phénomènes physiques et technologiques dont les effets sur les variations de temps de propagation sont mal maîtrisés et difficilement caractérisables.

De plus, les technologies évoluant rapidement, l'industrie est confrontée à des problèmes de migration de circuits, blocs de circuits, ou librairies. On peut aussi élargir la définition de la migration technologique en y englobant les changements de style de conception : cellules standard, semi-dédié, dédié, ou encore des changements d'architectures pour une fonction donnée.

Ainsi, non seulement les circuits asynchrones sont robustes par rapport aux variations des temps de propagation dans les technologies, mais ils se prêtent alors plus facilement aux différentes migrations technologiques. En raison de leur propriété de synchronisation locale, les circuits asynchrones sont fonctionnels quelque soit la réalisation des cellules qui le constituent pourvu que le protocole de communication reste respecté. Ainsi, il est possible au niveau technologique de modifier l'implémentation des cellules de base sans modifier la fonction. La performance obtenue sera simplement la vitesse maximale permise par la nouvelle technologie étant donné la structure du circuit. La migration technologique peut aussi être progressive. Il est possible de migrer un sous-ensemble de cellules sans que la fonction du circuit soit altérée. Par exemple, on peut imaginer re-concevoir un opérateur du chemin le plus fréquent en style dédié et le substituer à l'ancien afin d'optimiser globalement les performances ou la consommation suivant les objectifs requis.

Au niveau système, la migration est aussi aisée que ce soit pour un système intégré ou non. Tant que la spécification fonctionnelle et les interfaces ne sont pas modifiés, on peut remplacer un circuit asynchrone par son équivalent fonctionnel plus rapide sans avoir à re-concevoir l'assemblage système (par exemple le PCB pour un système non intégré). L'ensemble du système tirera parti des performances du nouveau circuit.

1.2.8. Modularité, Réutilisation

La modularité des circuits asynchrones est quasi-parfaite. Elle est due à la localité du contrôle et à l'utilisation par tous les opérateurs d'un protocole de communication bien spécifié. Il est en effet très facile de construire une fonction – un système – complexe en connectant des modules préexistants [CLAR 67]. Ceci a été parfaitement démontré lors de la conception du circuit Amphin qui regroupe 256 processeurs élémentaires [ROBI 97a], [ROBI 97b]. Cette modularité aide aussi à répartir les tâches de conception de blocs distincts sur différentes équipes de concepteurs. La spécification explicite du protocole aux interfaces des blocs asynchrones permet de les interconnecter par simple assemblage. Ceci permet de construire des systèmes flot de données, tout comme on peut assembler un système composé de blocs synchrones autour d'un bus de communication. Dans le cas asynchrone, la modularité entre blocs est facilitée sur deux aspects. Tout d'abord, la conception du système est facilitée : il n'y a pas besoin de spécifier un bloc en fonction du nombre de cycles horloge pour obtenir et échanger des données avec le bloc distant, la synchronisation est effectuée par les données, c'est bien un modèle flot de données. Les périphériques du processeur Aspro (paragraphe 4.2) ont ainsi été définis. Ensuite au niveau implémentation du circuit asynchrone, lorsqu'il n'y a pas d'hypothèses temporelles qui garantissent le fonctionnement, la conception au niveau physique est rendue plus facile. Les phases de placement-routage sont moins contraintes, sauf si ce n'est pour des questions de performances. La correction fonctionnelle du système est garantie quelque soit les délais dans les interconnexions aux interfaces.

Ces différents aspects sont d'autant plus intéressants avec l'évolution des technologies. Dans les technologies futures [SIA 97], il sera ainsi nécessaire pour concevoir des circuits synchrones de prévoir plusieurs cycles horloge pour transférer des données d'un bout à l'autre d'un système tout intégré (system-on-chip). Au niveau des méthodes de conception de circuits synchrones, il sera extrêmement difficile de garantir les délais localement et globalement dans de tels systèmes. Ainsi, les roadmap prévoient que les systèmes tout-intégrés seront composés de blocs fonctionnant avec des horloges locales. C'est le concept de circuit Localement-Synchrone Globalement-Asynchrone (GALS). Les échanges inter-blocs, supportés par des bus de communication « longs », ne pourront se faire à la fréquence interne des blocs. Il faudra avoir recours à des mécanismes de synchronisation capables de garantir des communications fiables entre blocs contrôlés par des horloges distinctes. Ce type de conception est typiquement du ressort de la conception asynchrone. Il sera plus facile et plus adapté de concevoir des systèmes de communication implémentés dans le style asynchrone (en particulier par rapport aux problèmes de traitements indéterministes), que concevoir des systèmes de communications synchrones multi-horloges [BAIN 98].

Ainsi pour conclure, les propriétés de robustesse / modularité / migration des circuits asynchrones en raison de leur synchronisation locale sont particulièrement intéressantes lorsqu'on souhaite promouvoir la réutilisation de blocs dans une entreprise, ou d'un point de vue plus général, l'échange de propriétés intellectuelles (IPs) dans le cadre d'implémentation de systèmes complexes.

1.3. Méthodes & outils de conception de circuits asynchrones

Comme nous l'avons présenté dans le paragraphe 1.1, contrairement aux circuits synchrones, le fonctionnement correct d'un circuit asynchrone ne dépend pas de la distribution des retards dans les portes et les connexions. Afin d'être indépendant des temps de propagation, les éléments de la logique sont contrôlés par des signaux à « poignée de main ». C'est le prix à payer pour supprimer la synchronisation globale réalisée par l'horloge. Ceci n'est cependant pas suffisant : comme toute transition de signal peut être interprétée comme un événement porteur d'information, la logique asynchrone doit être exempte d'aléas. En conséquence, les circuits asynchrones requierent une attention particulière et fine de toutes les parties du circuit, que ce soit pour les parties combinatoires ou séquentielles.

Néanmoins, certains types de circuits asynchrones relâchent la contrainte de correction fonctionnelle indépendamment des délais en introduisant des hypothèses temporelles. Ceci conduit alors à une réalisation et une conception plus simple. Il existe ainsi une large panoplie de techniques de conception de circuits asynchrones qui se caractérisent par leurs hypothèses temporelles. Le paragraphe 1.3.1 introduit les différents modèles de délais et la terminologie communément utilisée pour classifier les différents types de circuits asynchrones suivant le modèle de délai adopté. Le paragraphes 1.3.2 présente ensuite les différentes méthodes de conception de circuit asynchrone associées.

1.3.1. Classification des circuits asynchrones

Les circuits asynchrones sont communément classés suivant le modèle de délai et d'environnement du circuit adopté. Par définition, on dit qu'un délai peut être « fixe » : il

possède une valeur déterminée. Un délai peut être « borné », dans ce cas il possède une valeur située dans un intervalle connu. Enfin un délai peut être « non-borné », sa valeur est finie mais inconnue. Le modèle de délai d'un circuit logique est alors défini par les différents éléments le constituant : c'est à dire le modèle de délai des composants élémentaires (les portes logiques) et le modèle de délai de ses interconnexions (les fils). Enfin, il est important de caractériser le comportement de l'environnement dans lequel s'insère le circuit. Si l'environnement interagit avec le circuit sans contrainte de délai, on parle de fonctionnement en « mode d'entrée/sortie ». Dans le cas contraire, il existe des contraintes temporelles pour que circuit et environnement interagissent. Un mode communément utilisé est le mode « fondamental ». Ce modèle impose à l'environnement d'attendre la stabilisation du circuit avant de changer à nouveau les entrées du circuit. On peut penser par exemple dans le cas synchrone au temps de maintien d'une flip-flop.

La plupart des études sur les circuits asynchrones visent alors un compromis entre robustesse et complexité. La Figure 1-7 présente la terminologie habituellement utilisée pour qualifier les circuits asynchrones. Cette classification est effectuée suivant le modèle de délai et d'environnement choisis.

- Les circuits insensibles aux délais
  (**Delay Insensitive** circuits)
- Les circuits quasiment insensibles aux délais
  (**Quasi Delay Insensitive** Circuits)
- Les circuits indépendants de la vitesse
  (**Speed Independant** Circuits)
- Les circuits micro-pipeline
  (**Micropipeline** circuits)
- Les circuits de Huffman
  (**Huffman** Circuits)

Hypothèses temporelles de plus en plus fortes, Robustesse de plus en plus faible

Figure 1-7 : Terminologie des différentes classes de circuits asynchrones.

Il est à noter que nous n'utilisons pas ici le terme de circuits auto-séquencés (« self-timed »). Ce terme souvent utilisé pour désigner les circuits asynchrones ne correspond en fait à aucune définition précise. Il désigne simplement des circuits dont le séquencement est déterminé par des signaux internes plutôt que par un signal d'horloge. Ainsi, auto-séquencé est à mettre en opposition à synchrone mais ne renseigne pas sur le type de circuit asynchrone utilisé.

Dans la suite, nous commençons par décrire les circuits asynchrones dont le fonctionnement respecte fondamentalement la notion d'asynchronisme telle que nous l'avons définie dans le paragraphe 1.1. Nous présentons ensuite les classes dérivées qui introduisent des hypothèses temporelles de plus en plus fortes. Ces dernières se situent entre le mode de fonctionnement asynchrone pur et le mode de fonctionnement synchrone.

1.3.1.1. Circuits Insensibles aux Délais (Delay Insensitive)

Cette classe de circuits utilise un mode de fonctionnement purement asynchrone. Ces circuits sont basés sur un modèle de délai pour les fils et les éléments qui est « non-borné ». Aucune hypothèse temporelle n'est introduite : ils sont corrects indépendamment des délais dans les éléments (fils et portes) qui les constituent. Ainsi, un circuit répondra toujours correctement à une sollicitation externe pourvu qu'il ait assez de temps pour calculer.

Ce modèle de fonctionnement impose des contraintes extrêmement fortes. En effet, la plupart des circuits conçus aujourd'hui utilisent des portes logiques à une seule sortie (portes and, or, …). L'adoption du modèle de délai non-borné ne permet pas leur utilisation car la sortie de ces portes peut changer si une seule des entrées est modifiée. Ceci ne respecte pas le modèle car tous les composants de ce type de circuit doivent s'assurer que toutes les entrées actives sont acquittées avant de pouvoir produire une nouvelle sortie. La seule porte à une sortie qui respecte cette règle est la porte de Muller. Malheureusement, les fonctions implémentables avec seulement des portes de Muller sont très limitées. L'unique solution est d'utiliser un modèle de circuit basé sur des portes complexes. Dans ce cas, le circuit est composé d'éléments plus complexes que de simples portes logiques, et qui peuvent posséder plusieurs entrées et plusieurs sorties [EBER 91], [HAUC 95].

1.3.1.2. Circuits Quasi Insensibles aux Délais (Quasi Delay Insensitive)

Cette classe de circuit adopte le même modèle de délais non-borné pour les connexions mais y ajoute la notion de fourche isochrone (« isochronic fork »).

Une fourche est par définition un fil qui connecte un émetteur à plusieurs récepteurs. On la qualifie d'isochrone lorsqu'on suppose que les délais entre l'émetteur et les différents récepteurs sont identiques. A.J. Martin [MART 93] a montré que l'hypothèse temporelle de fourche isochrone est la plus faible à ajouter aux circuits insensibles aux délais pour les rendre réalisables avec des portes à plusieurs entrées et une seule sortie. Dans cette hypothèse, on peut en effet se permettre de ne tester qu'une branche d'une fourche et donc l'acquitter en supposant que le signal s'est propagé de la même façon dans les autres branches.

En pratique, l'hypothèse temporelle de fourche isochrone est assez faible et est facilement remplie par une conception soignée, en particulier au niveau du routage et des seuils de commutation. Il suffit finalement que la dispersion des temps de propagation jusqu'aux extrémités de la fourche soit inférieure au délai des opérateurs qui lui sont connectés (au minimum une porte et un fil) dont une sortie interagit avec la fourche [MART 90b], [BERK 92].

1.3.1.3. Circuits Indépendants de la Vitesse (Speed Independant)

Les circuits indépendants de la vitesse font l'hypothèse que les délais dans les fils sont négligeables tout en conservant un modèle non-borné pour les délais dans les portes. Ce modèle, qui n'est pas réaliste dans les technologies actuelles, est en pratique équivalent au modèle QDI à l'exception des fourches qui sont alors toutes considérées isochrones. Même si pendant longtemps la communauté a tenté de cerner les différences entre ces deux modèles, il y a aujourd'hui consensus pour considérer les modèles QDI et SI similaires. Hauck [HAUC 95] montre avec le schéma de la Figure 1-8 comment une fourche isochrone peut être représentée par un circuit indépendant de la vitesse.

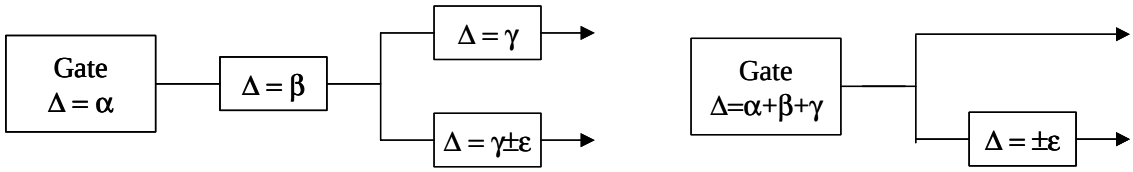


Figure 1-8 : équivalence entre modèle QDI (à gauche) et SI (à droite).

Cependant, il semble plus pertinent d'utiliser le modèle QDI car celui-ci identifie les fourches isochrones de celles qui ne le sont pas. Ceci offre alors un moyen de connaître et donc de vérifier les connexions du circuit qui ne sont pas insensibles aux délais au cours des différentes étapes de conception.

1.3.1.4. Circuits de Huffman

Les circuits de cette classe utilisent un modèle de délais identique aux circuits synchrones. Ils supposent que tous les éléments du circuit et les connexions sont bornés ou même de valeurs connues. En introduisant des hypothèses temporelles du même ordre que celle des circuits synchrones, bien que localement synchronisés, leur conception repose sur l'analyse des délais dans tous les chemins et les boucles de façon à dimensionner les signaux de contrôle locaux qui s'apparentent d'avantage à des horloges locales. Ces circuits sont d'autant plus difficiles à concevoir et caractériser qu'une faute de conception ou de délai les rend totalement non fonctionnel.

1.3.1.5. Micropipeline

La technique de micropipeline a été introduite initialement par Ivan Sutherland [SUTH 89]. Les circuits de cette classe sont composés de parties contrôles insensibles aux délais qui commandent des chemins de données conçus en utilisant le modèle de délai borné.

La structure de base de ces circuits correspond à une file de type Fifo composée de portes de Muller connectées tête-bêche (Figure 1-9).

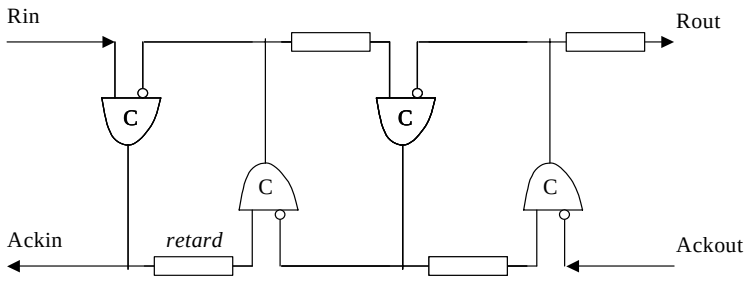


Figure 1-9 : Structure de base des circuits Micropipelines.

Cette structure implémente un protocole deux phases. Si on suppose tous les signaux à zéro initialement, une transition positive sur Rin provoque une transition positive sur Ackin qui se propage également à l'étage suivant. Le deuxième étage après un certain délai produit une transition positive qui se propage vers l'étage suivant et d'autre part qui revient au premier étage l'autorisant à transmettre une transition négative cette fois. Les transitions de signaux se propagent dans la structure tant qu'elles ne rencontrent pas une cellule occupée. Ce fonctionnement de type Fifo est bien insensible aux délais.

Cette structure de contrôle simple peut alors être enrichie d'opérateurs de mémorisation et d'opérateurs de traitement combinatoires. Le codage des données et le protocole associé est du type deux-phases données-groupées (Figure 1-10).

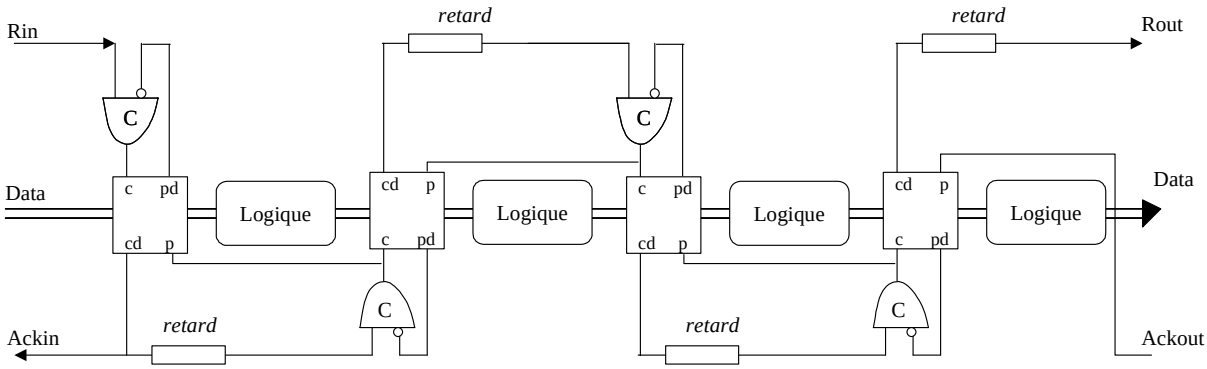


Figure 1-10 : Structure Micropipeline avec traitement et registres.

Dans cette architecture, les registres capturent les données sur occurrence d'un événement sur le signal C (« capture »). Ils produisent le signal Cd (« capture done») une fois la donnée mémorisée dans le registre. Durant cette phase, la donnée précédemment mémorisée est maintenue en sortie du registre. Lorsque P (« pass ») est actif, le registre laisse passer la donnée en sortie. Le signal Pd (« pass done ») signale que le registre est transparent. Ainsi, lors d'une occurrence du signal Rin, les données d'entrées sont stockées dans le premier étage. Rin est alors transmis vers l'étage suivant qui stocke le résultat transmis par la logique du premier étage. La propagation du signal Rin est retardée de façon à vérifier que la logique combinatoire est stable avant la capture du résultat dans l'étage suivant. Une fois la capture effectuée dans le deuxième étage, le premier registre est rendu passant ce qui permet le traitement dans le premier étage et autorise la capture d'un nouvel événement sur Rin.

Avec la logique qu'il contient, cette structure se comporte comme le circuit de la Figure 1-9. C'est une Fifo qui peut contenir un nombre variable de données, les données progressant le plus loin possible dans la logique en fonction du nombre d'étages vides. Dans cette architecture, le contrôle et les registres sont spécifiques. Quant à elle, la logique combinatoire est équivalente à la logique synchrone. Le problème des aléas est écarté grâce à l'insertion de délai sur les signaux de contrôle. L'hypothèse de délai consiste alors à vérifier que les données sont stables avant la mémorisation de celles-ci. Dans ce cas, les signaux de contrôle s'apparentent en fait à des horloges locales. Comme les délais sont fixes, cette structure ne permet pas de tirer parti de la variation dynamique du temps de traitement des opérateurs.

Un certain nombre d'optimisations peuvent être apportées à cette structure initiale. Des circuits indépendants de la vitesse peuvent être obtenus en détectant localement la fin de calcul de la logique. Ceci peut être effectué par exemple en remplaçant la logique combinatoire par de la logique à précharge [FURB 96a]. Il est aussi possible de supprimer les latchs explicites en utilisant de la logique différentielle à précharge [RENA 96]. Des bascules de mémorisation sur double front peuvent être utilisées dans le cas de protocole deux phases [YUN 96]. Enfin, des protocoles quatre-phases optimisés permettent aussi d'obtenir un taux de remplissage de la structure important afin d'augmenter les performances [FURB 96b].

Le micropipeline correspond à une proposition initiale de Sutherland [SUTH 89]. Il faut en fait considérer que les circuits micropipelines correspondent plutôt à une classe

d'architectures de circuit asynchrone et non directement à un modèle de délai de circuit asynchrone. De nombreuses méthodes de conception s'attachent à concevoir des circuits dits « Micropipeline » mais avec des modèles de délais plus ou moins différents. L'idée générale de ces approches est de séparer dans la spécification le contrôle des données et de les implémenter séparément dans des styles différents. En particulier, les contrôleurs des registres de pipeline peuvent être obtenus avec un grand nombre de méthodes et avec des hypothèses de délais plus ou moins fortes.

1.3.2. Méthodologies et outils de conception

Il existe aujourd'hui quelques outils de conception qui s'intéresse à la synthèse de circuits asynchrones mais aucun n'est commercialisé. Philips est le seul industriel qui possède un environnement de conception complet mais il est d'usage privé [BERK 93]. Les laboratoires universitaires possèdent un certain nombre d'outils mais qui souffrent d'une faible compatibilité entre eux et avec les environnements commerciaux, et qui trop souvent ne couvrent pas l'ensemble des phases de conception. Cependant, ceux-ci sont suffisants pour permettre à la communauté universitaire de concevoir des circuits fonctionnels. Voici une liste de publication sur les méthodologies de conception de circuits asynchrones qui peut être consultée : [HAUC 95], [BIRT 95], [LAVA 93], [PROC 95], [KISH 94], [BRZO 95], [VARS 86]. Le site Web suivant recense tous les outils disponibles :

http://www.cs.man.ac.uk/async/tools/index.html.

Ce paragraphe ne se veut pas exhaustif sur l'étude de l'ensemble des méthodologies de conception de circuits asynchrones, ce pourrait être l'objet d'un document complet, mais se veut un reflet du panorama existant.

1.3.2.1. Circuits Insensibles aux Délais

Afin de s'affranchir des limitations imposées par ce style de conception (cf. paragraphe précédent), il a été proposé de concevoir un ensemble de modules de base pas nécessairement insensibles aux délais en interne mais qui garantissent l'insensibilité aux délais du système une fois assemblés. Molnar, Fang et Rosenberg [MOLN 85], [ROSE 88] ont développé une méthodologie de conception de tels modules basée sur le formalisme des I-nets (graphes similaires aux réseaux de Pétri). A partir de ces modules, Brunvand et Sproull ont proposés un outil de synthèse utilisant le langage Occam comme formalisme d'entrée [BRUN 89]. A chaque structure du langage correspond alors un module ou ensemble de module permettant son implémentation.

Une approche proposée sur la théorie des traces (« trace theory ») a été proposée par Ebergen [EBER 91]. Cette méthode propose aussi un ensemble de module de base. Cependant le langage de spécification utilisé est le même pour décrire les modules et le circuit, ce qui le rend primitif et difficile à utiliser.

Plus récemment, des travaux de l'université de Groningen visent à dériver des circuits insensibles aux délais à partir d'un langage formel par raffinement de programmes [MALL 99].

1.3.2.2. Circuits Quasi Insensibles aux Délais

La conception de ces circuits a été développé à Caltech par le groupe de A. Martin. Depuis plus de dix ans, le groupe travaille sur une méthodologie de conception de circuits

asynchrones qui offre à la fois des facilités de conception dans un langage de haut niveau et une méthodologie de synthèse qui assure la correction des circuits [MART 93].

Le langage de spécification, appelé CHP, est issu du langage CSP de C.A.R Hoare [HOAR 78]. Un programme CSP consiste en un ensemble de processus qui calculent en parallèle et communiquent entre eux à travers des canaux. Ce modèle de processus communicants est similaire à la synchronisation locale des circuits asynchrones et à leur mode de fonctionnement flot de données. A partir d'une spécification du circuit dans ce langage, des règles de transformations successives permettent d'obtenir des équations booléennes qui s'implémentent dans la technologie cible. Cette méthode de synthèse est éprouvée et a permis de concevoir un certain nombre de circuits d'une complexité significative qui sont certainement parmi les plus performants aujourd'hui (cf. paragraphe 1.4).

Cependant, cette méthodologie de conception est éloignée des méthodes de conception classiques synchrones et ce pour plusieurs raisons. Tout d'abord, il n'existe actuellement aucun outil d'aide à la conception, que ce soit au niveau de la simulation du langage CHP de spécification ou au niveau de la synthèse logique de ce type de circuits. De plus, la méthode de synthèse s'attache à concevoir uniquement des circuits dédiés optimisés (« *full-custom* »). Au contraire, les approches « cellules standard » couramment utilisées aujourd'hui, même si elles ne peuvent offrir les mêmes niveaux de performances que des circuits dédiés, permettent une conception plus rapide et surtout offrent des possibilités de migration technologique plus importantes. Ces deux paramètres sont prépondérants dans un contexte industriel.

1.3.2.3. Circuits Indépendant de la vitesse

La conception des circuits indépendants de la vitesse est actuellement l'une des méthodologies la plus étudiée et utilisée. Elle est basée sur la spécification des circuits à l'aide de graphes. Il existe principalement deux types de graphes : des graphes de transitions de signaux (signal transition graph ou STG) et des diagrammes de changement de signal (change diagram ou CD). Les STG ont été introduits par Chu, Leug et Wanuga [CHU 85] [CHU 87] alors que les CD ont été introduits par Kishinevsky, Kondratyev, Taubin et Varshavsky [KISH 92]. Les CD sont présentés en détail dans [KISH 94] mais ne sont plus étudiés aujourd'hui.

Un STG spécifie un circuit à l'aide d'un formalisme proche des réseaux de Pétri dans lequel les transitions sont étiquetées par les noms des signaux. Ainsi, lorsqu'une transition est passée cela correspond à l'occurrence d'une transition du signal correspondant. La notation associe aux signaux le sens de la transition : un « + » pour une transition positive et réciproquement un «- » pour une transition négative. Il existe différents type de STG. Les premiers STG utilisés sont les graphes de transitions de signaux marqués (marked graphs ou STG/MG). Ce sont des réseaux de Pétri dans lesquels chaque place possède au plus une entrée et une sortie. Ce type de graphe ne permet pas de spécifier les conflits ou les choix, ainsi une transition active ne peut être désactivée qu'en étant passée. Ceci restreint en conséquence la classe des circuits modélisables.

Restreindre les STG avait pour seul objectif de simplifier la procédure de synthèse. Ces graphes ont ensuite été étendu de façon à permettre la spécification de choix, conjointement au développement d'algorithmes de synthèse plus performants. Les différentes propriétés de ces graphes ont été étudiées, telles que : vivacité (liveness), sûreté (safety), persistance (persistency), assignation cohérente d'états (consistent state assignement), assignation unique d'états (unique state assignement), graphes à cycle unique de transition (single cycle transitions). Des algorithmes ont été développés afin de tester ces propriétés et même transformer les graphes pour qu'elles soient remplies (voir [HAUC 95] pour une rapide

description). La principale difficulté de la synthèse logique consiste bien sûr à obtenir une implémentation et un mapping technologique sans aléa. L'outil couramment utilisé aujourd'hui par la communauté pour effectuer la synthèse de contrôleurs asynchrones à partir de STG est *Petrify* [KOND 98], [PAST 98].

Ainsi, les méthodes de synthèse existantes se distinguent les unes des autres suivant la nature des graphes utilisés pour spécifier le problème, et suivant leur propriété et leur implémentation. En raison de limitations pour permettre leur implémentation, la synthèse de circuits indépendants de la vitesse à partir de STG n'est pas générale et ne permet pas la description et la synthèse de circuits quelconques. Ces techniques peuvent être appliquées avec succès à la conception de contrôleurs asynchrones de faible complexité (quelques dizaines de signaux). Cependant, la méthode souffre de l'absence d'une méthodologie de haut niveau qui permettrait de structurer la conception du circuit. Ainsi leur utilisation pour la synthèse de systèmes complets est impossible en pratique en raison de l'explosion du nombre d'états.

1.3.2.4. Circuits de Huffman

Pour ce type de circuit, la fonction est exprimée à l'aide d'une table de flots. On suppose que tous les états instables conduisent directement à des états stables avec au plus une transition sur chacune des sorties. Comme il n'y a pas d'horloge pour échantillonner les entrées, il faut assurer que les changements des entrées qui causent le passage par des états instables ne conduisent pas à l'apparition d'aléas sur les sorties.

Tous les aléas dus au changement d'une seule entrée peuvent être supprimés en ajoutant des sommes de produit à la fonction [LAVA 93]. Cependant, cette technique ne garantie pas un fonctionnement correct si plusieurs entrées changent en même temps avant la stabilisation des sorties. Les premiers circuits de ce type étaient restreint au mode fondamental. Ceci obligeait ainsi à ajouter des délais dans les boucles de rétroactions et à assurer que l'environnement n'apporte une nouvelle transition sur les entrées avant la stabilisation du circuit.

Différentes méthodes ont été proposées afin de pouvoir concevoir des circuits basés sur le modèle de délai borné mais autorisant le changement de plusieurs entrées [HOLL 82], [NOWI 91], [YUN 92]. La description des circuits en mode « burst » se fait en affectant un ensemble de signaux à chaque transition d'un graphe de type réseau de Pétri. Nowick [NOWI 95] est l'un des principaux contributeurs dans le domaine de la conception de machines à états « Burst mode » avec l'outil de synthèse *Minimalist*.

Malgré tout, l'utilisation du modèle de délai borné posent un certain nombre de problèmes pour la conception de circuits asynchrones. Tout d'abord, ces méthodologies de conception ne peuvent s'appliquer qu'à la conception de contrôleur asynchrone de faible complexité. Il est impossible de réaliser des parties opératives efficaces en complexité et performance (le respect du mode fondamental en raison du nombre élevé de changement sur les entrées est coûteux). Le dernier problème est celui du test de ces circuits. L'ajout de sommes de produits pour éviter les aléas peut rendre difficilement observable certaines fautes. De plus, le test de ces circuits requiert de tester les fautes dues aux délais puisque le fonctionnement du circuit dépend de la valeur des délais introduits dans les boucles de rétroaction notamment. Dans un circuit synchrone, une faute de délai peut se corriger en ralentissant l'horloge, c'est impossible pour ce type de circuit.

1.3.2.5. Micropipeline

Les circuits Micropipeline tel que présentés dans le paragraphe 1.3.1 présentent une architecture régulière dont l'implémentation est relativement simple, même si cela nécessite de caractériser les délais de la logique combinatoire pour retarder en correspondance les signaux de contrôle locaux. Comme nous le précisions, différentes optimisations de cette structure Micropipeline sont possibles en utilisant des protocoles optimisés ou de la logique plus performante. Par exemple, les anneaux auto-séquencés valorisent facilement le style de conception Micropipeline de part leur régularité [WILL 94], [ELHA 95], [MATS 97], [SPAR 93].

Cependant, les circuits réels, tel que des microprocesseurs, utilisent des chemins de données beaucoup plus complexes incluant des boucles et des structures de décisions dépendantes des données. Il apparaît ainsi nettement que les réalisations basées sur le concept de Micropipeline ont conduit à des circuits fonctionnels au prix d'efforts de conception énormes [FURB 94]. Ainsi, une méthodologie de conception de haut niveau adaptée est ce qui a longtemps manqué à la technique Micropipeline.

Afin de permettre la conception de circuit Micropipeline complexes, des travaux récents se focalisent sur la synthèse à partir de langages de haut niveau : langages de type CSP ou langages de description de matériel (HDL). On peut citer évidemment le langage Tangram développé par Philips qui est un précurseur et permet des réalisations performantes. Plus récemment, l'université de Manchester a développé les langages Lard et Balsa [ENDE 98], [BARD 97], [EDWA 99]. A ce jour, les outils associés ne permettent pas encore des implémentations efficaces. De leur côté, l'université d'Utah [JACO 00] et le Polytechnico di Turino [BLUN 00] proposent aujourd'hui des outils de conception utilisant le langage de description de matériel Verilog. Pour tous ces outils, la première étape consiste à séparer dans la spécification de départ le contrôle des chemins de données. Ceux-ci sont ensuite synthétisés séparément. Le chemin de données, décrit sous forme de logique combinatoire en HDL, est synthétisé avec les outils commerciaux de synthèse synchrone. De son côté, le contrôle est synthétisé avec les outils de conception spécifiques asynchrone (synthèse sans aléa). Ces contrôleurs asynchrones peuvent être synthétisés de différentes manières : sous forme de STG avec l'outil *Petrify*, sous forme de « burst mode » FSM avec les outils *Minimalist*, *3D*, etc. Il faut noter que la partie délicate de ces approches consiste à analyser les dépendances de données de la spécification HDL pour obtenir automatiquement les contrôleurs asynchrones adéquats. Le risque de ce type d'approche est d'obtenir des contrôleurs asynchrones dont la complexité dépasse la capacité de synthèse des outils dédiés asynchrones.

1.3.3. Conclusion

Tout comme il existe plusieurs classes de circuits asynchrones, incluant de nombreux compromis possibles, les méthodologies de conception associées sont nombreuses. Ceci est simplement le reflet du vaste spectre de solutions qu'offre le relâchement des synchronisations. A ce jour, il n'existe pas de vision unificatrice des différentes méthodologies de conception de circuits asynchrones. Ainsi, il n'y a pas de consensus sur le type de circuit asynchrone le plus approprié : cela semble fortement dépendre des objectifs et des moyens visés. Si on considère les avantages et potentiels des circuits asynchrones présentés dans le paragraphe 1.2, les circuits DI, QDI et SI semblent les mieux adaptés pour concevoir des systèmes robustes dans les technologies avancées, tout en visant de bonnes performances.

1.4. Différents microprocesseurs asynchrones

Nous proposons dans ce paragraphe un panorama des processeurs asynchrones conçus et réalisés au cours des dernières années en faisant la différence entre deux processeurs commerciaux et des designs universitaires. Il existe bien sûr beaucoup d'autres circuits asynchrones prototypes, qu'ils aient été fabriqués ou non, conçus par des industriels ou des universitaires, mais aucun autre de commercial. Il serait trop long de les décrire et nous avons préféré nous concentrer sur les processeurs dont la complexité impose un savoir faire important dans le domaine de la conception des circuits asynchrones.

1.4.1. Produits commerciaux

1.4.1.1. Microcontrôleur 80C51 (Philips Research & Semiconductor, 1998)

Le seul processeur conventionnel et asynchrone présent aujourd'hui sur le marché est un microcontrôleur compatible avec l'Intel 80C51. Il a été conçu par Philips Research Labs en collaboration avec l'université de Eindhoven [GAGE 98], celui-ci est commercialisé par la division semi-conducteur de Zurich.

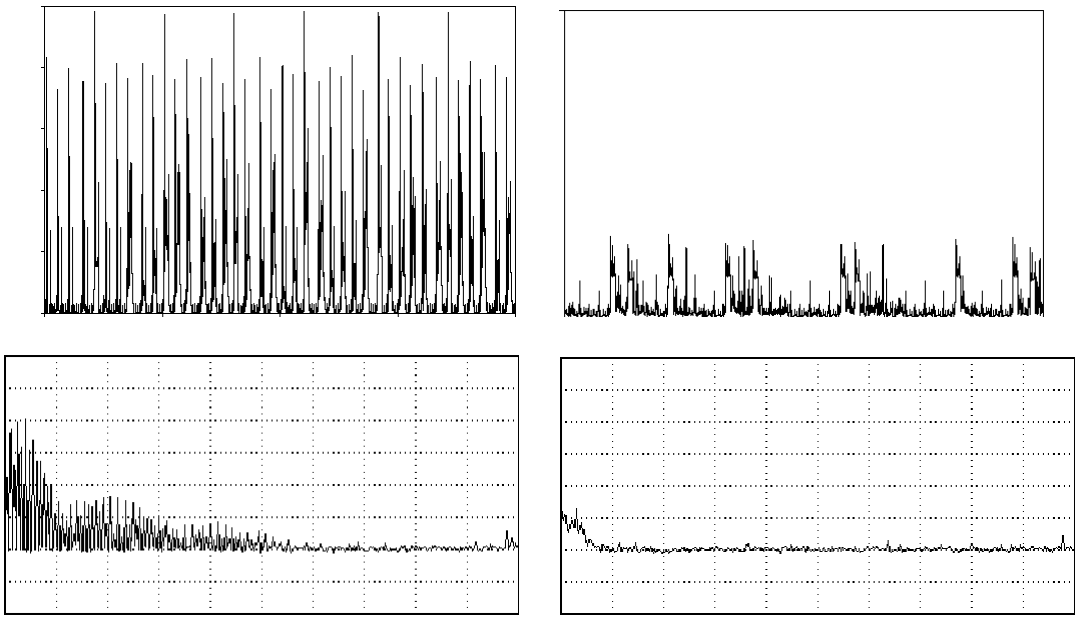
Ce contrôleur 80C51 est une architecture CISC 8-bits. La version asynchrone a été conçue avec l'environnement de développement basé sur le langage Tangram. Il est réalisé avec une technique micropipeline et totalement implémenté en cellules standard, dans une technologie CMOS 0.5µm. La conception du cœur a pris seulement 6 mois de développement. Ceci est la preuve de la pertinence de la méthodologie de conception basée sur le langage Tangram et de la maturité des outils associés. C'est un processeur peu performant, il a été optimisé pour la faible consommation et le faible bruit. Comme beaucoup de contrôleur, il possède plusieurs modes sélectables par logiciel pour réduire la consommation (idle, power-down, …). Par rapport à son équivalent synchrone, le 80C51 asynchrone montre un gain de consommation d'un rapport 4.

Outre le cœur de microcontrôleur, le composant intègre les multiples fonctions suivantes : un convertisseur DC/DC afin de mettre en œuvre un système de traitement à consommation minimale tel que présenté Figure 1-6 (paragraphe 1.2), un détecteur du niveau d'alimentation des batteries, 20ko de ROM OTP, 1ko de RAM, une interface I^2C, un UART, 2 timers, une horloge temps réel, un générateur de tonalité, un convertisseur numérique/analogique, un ensemble de fonctions d'émission / réception spécialement conçues pour les Pagers (démodulateur FSK, filtre, récupérateur d'horloge, …). La documentation technique complète de cette famille de processeurs est disponible sur les sites Web suivants :

http://www-us.semiconductors.com/pip/PCA5007H
http://www-us.semiconductors.com/pip/PCA5010H

De plus, outre sa faible consommation, la faible émission de bruit électromagnétique est un atout important de ce microcontrôleur. Philips le présente d'ailleurs comme un circuit dédié aux applications « Pager » puisqu'il permet la conception d'un « Pager tout soft ». En effet, si aucune précaution n'est prise dans le cas synchrone, l'émission d'ondes électromagnétique par le circuit numérique synchrone dans le boîtier du Pager brouille la réception des informations. La solution communément adoptée est de confier la réception à un circuit dédié spécifique de faible complexité et peu rayonnant. Celui-ci en mode de réception des messages stoppe l'horloge des autres circuits puis la réactive pour effectuer l'exploitation des messages (affichage, …). Grâce à la faible émission électromagnétique du microcontrôleur asynchrone,

Philips a pu intégrer l'application complète sur un unique circuit, ce qui diminue le coût matériel du système et offre une solution plus simple au niveau logiciel. La Figure 1-11 décrit les spectres en courant mesurés des versions synchrones et asynchrones du 80C51.



Version synchrone ***Version asynchrone***

Figure 1-11 : Consommation en courant et spectres correspondants des versions synchrone et asynchrone du contrôleur 80C51.

Plus récemment, ce cœur de microcontrôleur 80C51 a été intégré dans une carte à puce dans le cadre d'un projet européen Descale [KESS 00].

1.4.1.2. Data Driven Media Processor (Sharp, 1999)

La société Sharp, associée aux universités de Osaka et Kochi a développé le premier processeur multimédia flot de données [TERA 99]. L'utilisation de la technique asynchrone a permis la conception d'un processeur qui exécute de 2.4 à 14.4 Gops en consommant moins de 2 Watts. Le composant est conçu dans une technologie 0.25µm, 4 niveaux de métaux, 2.5 Volts.

L'architecture du Media Processor intègre huit unités de traitement super-pipelinées pouvant fonctionner en parallèle : deux unités arithmétiques et logiques générales et six unités dédiées au traitement d'images (filtrage 1D, 2D, 3D, unité de zoom, table de correspondance et unité de traitements arithmétiques avancés).

Ce composant est commercialisé dans des téléviseurs numériques et des systèmes de capture / traitement d'images. Les avantages et mérites de ce composant totalement asynchrone sont vantés sur le site Web suivant :

http://www.sharpsdi.com/DDMPhtmlpages/DDMPmain.html.

1.4.2. Processeurs asynchrones

1.4.2.1. Caltech Asynchronous Processor (Caltech, 1989)

Le premier processeur asynchrone a déjà 10 ans (1989), il a été conçu au California Insitute of Technology (Caltech) par le groupe de A. Martin [MART 89]. Il est dénommé CAP pour *Caltech Asynchronous Processor*. Il s'agit d'un prototype dont les fonctionnalités ont été simplifiées tout en restant représentatives d'un microprocesseur, et dont l'objectif était de montrer la faisabilité et l'intérêt de la méthode de conception asynchrone développée par ce groupe. Ce processeur est basé sur une architecture RISC 16-bits et comporte 16 registres. Le style de conception est de type QDI, la méthodologie étant basée sur le langage CHP. Il a été implémenté en circuit dédié (« full custom ») dans une technologie CMOS 1.6µm. Le circuit est fonctionnel et offre les performances suivantes : 26 Mips pour une consommation de 1.5 W à 10 Volts, 18 Mips pour une consommation de 225 mW à 5 Volts. A 2 Volts, le processeur délivre encore 5 Mips pour une consommation de 10.4 mW (voir site Web http://www.async.caltech.edu/).

Une version améliorée de ce même processeur a été implémentée dans une technologie AsGa en 1994 afin de démontrer l'indépendance de l'approche vis-à-vis de la technologie [TIER 94]. Ce dernier montre une performance de 100 Mips pour une consommation de 2 Watts.

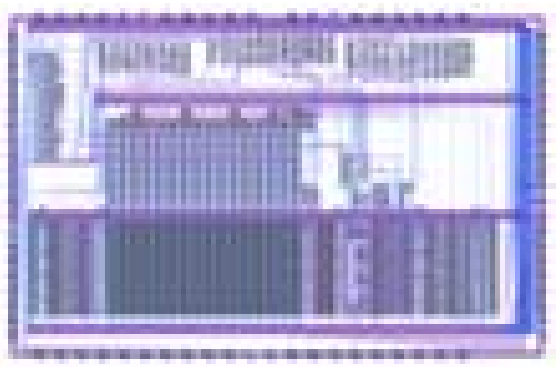


Figure 1-12 : CAP, Caltech.

1.4.2.2. Amulet 1, 2e, 3i (Université de Manchester, 1995, 1996, 2000)

Les microprocesseurs Amulet 1, 2e, 3i, ont été conçus à l'université de Manchester (site Web http : //www.cs.man.ac.uk/amulet/projects/). Ces trois versions implémentent une architecture RISC 32-bits ARM.

Le premier processeur, l'Amulet1 [WOOD 97] était conçu selon le style Micropipeline tel que proposé initialement par I. Sutherland, c'est à dire en utilisant un protocole deux-phases données groupées. Ce processeur avait des performances comparables aux processeurs ARM du moment. L'amulet2 est une version améliorée de l'Amulet1. Ce processeur intègre entre autre un chemin de données « full custom », une mémoire cache et un timer. Celui-ci utilise toujours le style Micropipeline mais utilise une signalisation quatre-phases données groupées, il a été implémenté dans une technologie CMOS 0.5µm. Le circuit Amulet2e intègre le cœur de processeur Amulet2 ainsi qu'un interface mémoire flexible (RAM, contrôleur DRAM) [FURB 97], [FURB 99]. L'Amulet2e montre une performance de 42 Mips Dhrystone pour une consommation de 150 mW à une tension de 3.3 Volts. Cette version a permis de corriger

quelques « bugs » de l'Amulet2 mais les résultats n'ont cependant pas permis de montrer une supériorité de ce circuit par rapport à la version synchrone équivalente.

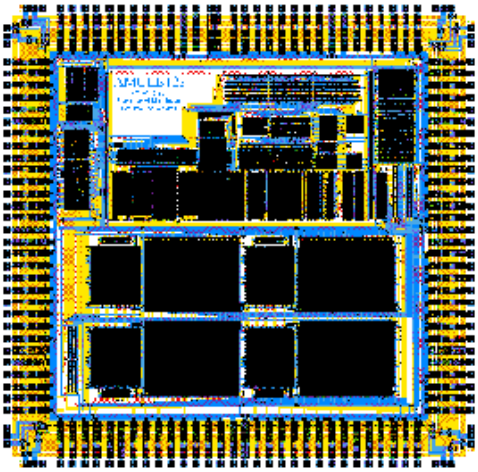


Figure 1-13 : Amulet2e, Université de Manchester.

Le dernier processeur développé par l'université de Manchester est appelé Amulet3i. C'est un système intégré complet qui contient un cœur Amulet3, 8ko de RAM, 16ko de ROM, plusieurs canaux DMA, une interface mémoire externe programmable, un bus interne asynchrone [BAIN 98], une interface interne bus-synchrone / bus-asynchrone, un support pour le debug, une interface pour le test [GARS 99], [GARS 00]. Ce circuit a été conçu dans une technologie 0.35µm, 3 niveaux de métaux, 3.3 Volts.

Les performances sont de 100 Drhystones Mips pour une consommation de 215 mW à une tension de 3.3 Volts. Ceci est comparable avec les performances d'un processeur ARM9 cadencé à 120 Mhz. Ce nouveau processeur rivalise avec les processeurs synchrones en terme de vitesse, consommation, et surface. Il possède de plus un avantage unique en terme de gestion de la consommation et d'émission électromagnétique. Ce processeur Amulet3i devrait être commercialisé, il a été conçu en partenariat avec la société canadienne Cogency Technology Inc. et avec la société Américaine ASIC Alliance Corporation.

1.4.2.3. Titac-2 (Université & Institut de Technologie de Tokyo, 1994, 1997)

Le microprocesseur Titac-2, qui fait suite au processeur Titac-1 [NANY 94], a été réalisé en 1997 par l'université et l'institut technologique de Tokyo [TAKA 97] (voir le site Web suivant : http://www.hal.rcast.u-tokyo.ac.jp/titac2/index.html).

L'architecture de ce processeur est celle d'un RISC 32-bits MIPS R2000. Il est structuré en un pipeline de 5 étages, il possède 40 registres et intègre 8.6 ko de mémoire cache. C'est un circuit asynchrone dont la conception est basée sur le modèle SDI (Scalable Delay Insensitive). Ce modèle ajoute des hypothèses temporelles au modèle QDI avec un critère de localité. Ce circuit a été réalisé en cellules standard en technologie CMOS 0.5µm.

Ce processeur montre une performance de 54 Mips pour une consommation de 2.11 Watts à une tension de 3.3 Volts. Pour une architecture relativement proche, ses performances sont supérieure à celles du processeur Amulet2e mais sa consommation est beaucoup plus élevée.

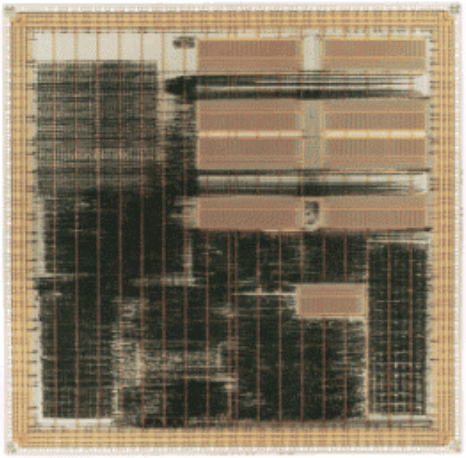


Figure 1-14 Titac-2, Université de Tokyo.

1.4.2.4. MiniMIPS (Caltech, 1999)

Le microprocesseur le plus performant a été réalisé par le groupe de Caltech. Il s'agit du MiniMIPS dont l'architecture a été présentée en 1997 [MART 97] et dont les performances réelles ont été mesurées début 1999. Ce processeur est une architecture RISC 32-bits MIPS R3000, auquel il manque les instructions d'accès mémoire sur demi-mots, le TLB (Table Lookaside Buffer) et le mécanisme d'interruption externe. Il intègre 32 registres et deux mémoires caches de 4 Koctets.

Ce processeur, tout comme le premier processeur CAP, est basé sur le style de conception QDI et la même méthodologie. Son niveau de pipeline est très élevé, le nombre de transitions par étage étant limité à une vingtaine par cycle, avec en moyenne 8 étages par instruction. Ce processeur a été réalisé en technologie CMOS 0.6µm et présente des performances remarquables. A tension nominale 3.3 Volts, il présente 180 Mips crête pour une consommation de 4 Watts. A 2 Volts, il offre 100 Mips pour une consommation de 850 mW, et à 1.5 Volts 60 Mips pour une consommation de 220 mW.

Ce niveau de performance de 180 Mips est de plus de deux fois supérieur à celle du processeur commercial synchrone MIPS R4200 fabriqué en 1994 dans une technologie équivalente.

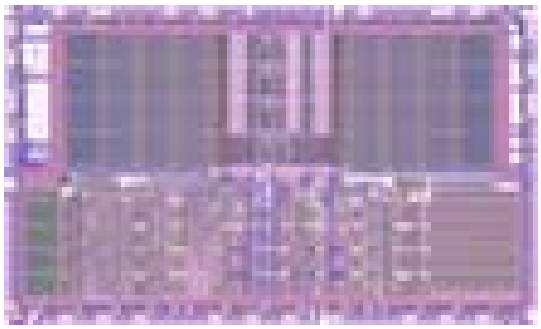


Figure 1-15 : MiniMIPS, Caltech.

1.5. Conclusion

Les circuits asynchrones utilisent un mode de fonctionnement totalement différent des circuits synchrones. Contrairement à une synchronisation globale, la synchronisation de ces circuits est effectuée localement grâce à une signalisation bidirectionnelle entre tous les éléments du circuit. Ceci s'implémente avec des protocoles de communication et un encodage des données adaptés. De nombreux types de circuits asynchrones co-existent : ils sont classés suivant les hypothèses temporelles utilisées pour les implémenter. Les différentes méthodologies de conception associées ont permis la réalisation de circuits asynchrones d'une complexité significative, y compris des circuits commercialisés. Comme on a pu le montrer dans le paragraphe précédent, leurs résultats de performances montrent un certain niveau de maturité des méthodologies de conception.

Cependant, si on se place dans le contexte du début de la thèse en 1996, la communauté asynchrone avait largement porté ses efforts sur le développement d'outils pour la conception de contrôleurs asynchrones. Ces outils peuvent être qualifiés de bas niveau car ils manipulent des fonctions logiques et se focalisent uniquement sur la suppression des aléas. Ces outils utilisent un formalisme d'entrée tel que des graphes (CD, STG, « burst mode »). Ils permettent l'implémentation efficace de contrôleurs de faible complexité mais sont inadaptés pour concevoir des systèmes et architectures complexes.

Au contraire, les méthodes de conception basées langage sont adaptées à la spécification de systèmes et à leur conception. Ceci est depuis longtemps le cas pour les méthodes de conception dédiées aux circuits synchrones grâce à l'utilisation répandue des langages de description de matériel. Dans le domaine de l'asynchrone, les précurseurs sont Erik Brunvand et le langage Occam [BRUN 89], Alain Martin et le langage CHP [MART 90], Vankatech Akella et le système Shilpa [AKEL 92], [GOPA 92], Kees Van Berkel et le langage Tangram chez Philips Research [BERK 93].

Vu les différentes études sur les circuits asynchrones, nous avons choisi de nous intéresser à la méthodologie de conception proposée par Caltech pour les raisons suivantes :
- les circuits asynchrones quasi-insensibles aux délais offrent des performances très intéressantes (processeur MiniMIPS)
- cette méthodologie de conception est basée sur un langage de haut niveau : le langage CHP,
- la robustesse intrinsèque de ce type de logique est un avantage indéniable pour l'intégration de systèmes complexes dans les technologies avancées.

A partir de la méthodologie de conception proposé par Caltech, le travail de thèse s'est dirigé dans les directions suivantes. Tout d'abord, offrir des moyens de simulations adaptés à ce type de formalisme qui puisse s'insérer dans les flots de conception traditionnels. Ce travail a donné lieu à la réalisation d'un traducteur CHP vers VHDL qui est présenté dans le chapitre 2. Ensuite, proposer une méthode de conception basée sur des cellules standard et non sur des circuits « *full-custom* ». Le chapitre 3 présente les différents choix de synthèse bas-niveau effectués afin de pouvoir implémenter en cellules standard ce type de circuit. En conclusion du chapitre 3, nous présentons le flot de conception développé.

Parallèlement à la définition de la méthodologie de conception, nous avons étudié, conçu et fabriqué un microprocesseur RISC 16-bit asynchrone. Les deux études ont été réalisées

conjointement : la méthodologie de conception permettant la réalisation du prototype, le projet de processeur permettant de nourrir la réflexion sur le flot de conception et de l'évaluer.

Par ailleurs, l'objectif de cette étude de conception consiste à exploiter au maximum les bénéfices que le mode de fonctionnement asynchrone peut apporter à une architecture de processeur. Le processeur dénommé Aspro est constitué d'une architecture originale qui intègre quatre unités d'exécution indépendantes et un mécanisme de réservation dans le banc de registres afin de pouvoir terminer l'exécution des instructions dans le désordre. Son architecture et son jeu d'instructions sont présentés dans le chapitre 4. La micro-architecture du processeur est étudiée en détail dans le chapitre 5. Celle-ci est obtenue par raffinement successifs de programmes CHP. Enfin, le chapitre 6 traite de l'implémentation de l'architecture, de son optimisation et des performances obtenues. Le processeur a été réalisé avec succès dans une technologie 0.25µm de STMicroelectronics et montre des performances intéressantes. Les résultats de celui-ci sont finalement comparés aux autres processeurs asynchrones existants.

Partie I

Une méthodologie de conception de circuits asynchrones quasi-insensibles aux délais

Cette première partie du manuscrit regroupe les chapitres 2 et 3. Celle-ci propose une méthodologie de conception de circuits asynchrones quasi-insensibles aux délais réalisés en cellules standard. En partant de la proposition initiale d'A. Martin [MART 90], nous proposons dans le chapitre 2 une extension du langage CHP, puis nous présentons la réalisation d'un traducteur CHP vers VHDL. Cet outil offre un moyen de simuler des spécifications en langage CHP et grâce à l'utilisation d'un langage standard tel que VHDL ceci offre une plate-forme de simulation de circuit mixte synchrone / asynchrone. Ensuite, le chapitre 3 présente une méthodologie de conception de circuits quasi-insensibles aux délais implémentés en cellules standard. Cette méthode propose un choix de protocole asynchrone et de mapping technologique, défini une restriction du langage CHP dit synthétisable et propose une bibliothèque de cellules standard pour pouvoir effectuer la synthèse de ce type de circuit. En conclusion de cette première partie, nous donnons une description du flot de conception complet que nous avons mis au point et utilisé pour réaliser le processeur Aspro.

Chapitre 2 :

Un environnement de simulation pour circuit mixte synchrone/asynchrone basé sur la traduction CHP vers VHDL

Introduction

Ce chapitre présente un environnement de simulation qui permet de concevoir de manière conjointe des circuits synchrones et asynchrones. Cette méthode est principalement basée sur le développement d'un outil appelé « CHP_2VHDL » qui traduit automatiquement des spécifications écrites en CHP, un langage proche de CSP, en un programme VHDL [RENA 98a], [RENA 99b]. Ce travail suit deux motivations principales. Tout d'abord, ceci permet d'offrir aux concepteurs de circuits asynchrones un environnement de conception et de simulation puissant, pouvant mélanger à la fois des spécifications CHP, des programmes HDL et des descriptions niveau porte. Ensuite, cet environnement offre aux concepteurs synchrones habitués aux approches de conception descendante basée sur les HDL la possibilité d'inclure des circuits sans horloge dans leur design.

Pour commencer, nous introduisons dans le paragraphe 2.1 les objectifs de cet environnement de conception, ainsi que la justification des choix de notre approche. Une vue générique de la méthode de conception mixte synchrone / asynchrone est alors présentée. Le langage CHP qui est utilisé est une extension de la proposition initiale de A. Martin [MART90]. Celui-ci a été enrichi pour répondre à nos besoins de simulations. La syntaxe et la

sémantique du langage sont présentées paragraphe 2.2. La méthode de traduction en VHDL, l'outil CHP_2VHDL qui a été développé, ainsi que son environnement logiciel sont ensuite décrits dans le paragraphe 2.3.

2.1. Proposition d'une méthode de conception mixte synchrone / asynchrone

2.1.1. Objectif d'un tel environnement de conception

Aujourd'hui, les technologies VLSI offrent des potentiels croissants en terme de vitesse, consommation et complexité. Cette complexité ne se gère plus manuellement depuis des années déjà. La communauté des concepteurs synchrones a développé des méthodologies de conception descendante, basées sur l'utilisation de langage de haut niveau. Les vendeurs d'outils de CAO (Conception Assistée par Ordinateur) proposent une panoplie complète d'outils, depuis la validation d'une spécification jusqu'à la synthèse, le placement & routage, la génération de vecteurs de test, le tout étant – plus ou moins – compatible et basé sur un ensemble de formats et de langages standardisés.

Les approches descendantes (*top-down*) basées sur les langages présentent des avantages très importants [MEYE89]. Depuis une première spécification de haut niveau jusqu'à un niveau de description structurel à grain fin, le problème peut être décrit en utilisant un même langage. Ceci offre une continuité sémantique entre tous les niveaux de descriptions, y compris les environnements de test, et constitue donc un outil puissant pour faire de l'exploration architecturale. Ainsi, les approches langages, en cachant les aspects liés à l'implémentation, permettent d'étudier des architectures tout en programmant, ceci est couramment appelé « la programmation VLSI ».

Le niveau de description RTL (*Register Transfert Level*) est maintenant supporté par les outils commerciaux et est largement adopté par l'industrie [LIS 89]. Le niveau de description « comportemental » fournit un niveau d'abstraction plus élevé en évitant de spécifier le séquencement au cycle horloge près. Cette technologie est encore émergente et reste peu adoptée. Néanmoins, les langages de description de matériel n'utilisent pas le concept de « canal », tel qu'il est définit dans le langage CSP [HOAR78]. Au contraire, la communauté de concepteur asynchrone utilise depuis longtemps des langages dérivés de CSP [MART90] [MAY90] [BERK91] [BRUN89]. Cette différence n'aide pas à la convergence des méthodes et outils entre circuits synchrones et asynchrones.

De plus, même si de nombreux langages basés sur le CSP sont largement utilisés pour modéliser des circuits asynchrones, il n'existe pas aujourd'hui de réel consensus sur un unique langage de spécification dédié à la modélisation de circuits asynchrones. L'université de Caltech a proposé le langage CHP [MART90] [MART93] mais ne possède pas d'environnement de conception complet. De son côté, Philips a défini Tangram [BERK91] [BERK93]. C'est actuellement le flot de conception le plus intégré et complet, mais il n'est pas disponible . L'université d'Utah [BIRT95] a développé un outil basé sur Occam, l'université de Manchester a développé LARD [ENDE98]. Tous ces langages utilisent le même concept de base de CSP : des processus concurrents communiquant par passage de message via des canaux. D'autres formalismes ont été proposés pour modéliser les circuits asynchrones, ils sont principalement basés sur les graphes [KISH94] [KUDV96] [YUN92]. Ces approches sont éloignées des approches langages utilisées aujourd'hui par l'industrie,

mais elles peuvent néanmoins être utilisées en tant que formats intermédiaires (lors des phases de synthèse en particulier). Il existe ainsi un nombre important de méthodes et d'outils différents, qui sont peu compatibles entre eux (voir [HAUC95] [BIRT95] pour une évaluation). Tout ceci n'est pas en faveur de la convergence des méthodes de conception.

Tandis que le SIA Roadmap [SIA97] prévoit la conception de systèmes incluant de nombreux espaces temporels dans les futures technologies, il est souhaitable de promouvoir et d'encourager des méthodes et flots de conception qui supportent la conception de circuits comportant à la fois le mode synchrone et le mode asynchrone. Ceci permettrait ainsi de concevoir aisément des systèmes du type Globalement Asynchrone Localement Synchrone [CHAP 84], [HEMA 99], [MUTT 00].

La méthode de conception que nous proposons dans cette thèse s'inscrit donc dans un cadre plus large qu'une conception uniquement dédiée aux circuits asynchrones. Nous voulons ainsi proposer une méthode de conception de circuits mixte synchrone / asynchrone. Les deux styles de circuit, avec leurs propres avantages et inconvénients peuvent y être mélangés, en utilisant leurs outils de conception dédiés. Le but de notre approche est donc : d'offrir une plate-forme de conception mixte synchrone / asynchrone, de se baser sur des formalismes standard, d'être compatible avec les outils commerciaux et que la migration technologique de blocs existants soit possible.

Nous présentons dans le paragraphe suivant les choix que nous avons faits afin de répondre à ces différents besoins : vis à vis de la modélisation des circuits asynchrones, de la simulation, et de la synthèse.

2.1.2. Présentation de la méthode de conception

Comme nous le précisions ci-dessus, des langages issus de CSP sont largement utilisés afin de modéliser les circuits asynchrones. Le langage CSP [HOAR78] permet de décrire des processus concurrents communiquant par passage de message via des canaux. Ce langage, qui utilise les « commandes gardées » de Dijkstra [DIJK75], correspond parfaitement à la représentation du style de circuits « localement synchronisés » que sont les circuits asynchrones. Les commandes gardées sont relativement coûteuses à implémenter en informatique mais reflètent bien la notion de choix au niveau matériel. Enfin, le langage CSP est parfaitement adapté à la vérification formelle de programme.

Nous avons donc choisi le langage CHP, qui a été défini par A.J. Martin [MART90], auquel nous avons rajouté quelques extensions présentées dans le paragraphe 2.2, afin d'enrichir les possibilités de modélisation et de simulation. Ainsi, contrairement au langage VHDL, le langage CHP permet de modéliser des canaux, et offre plus de flexibilité pour modéliser la concurrence et le séquencement. Comme nous avons choisi ce langage CHP, le flot de conception doit pouvoir offrir les points suivants. Tout d'abord, il est nécessaire de pouvoir simuler des spécifications CHP à tout niveau de granularité afin que le concepteur puisse valider son modèle asynchrone par simulation. Ensuite, il faut pouvoir réutiliser tout ou partie de circuits existants (c'est la notion de *IP reuse*). Ceci signifie qu'il faut pouvoir simuler dans le même environnement à la fois des programmes CHP, des programmes HDL et/ou des blocs niveau porte. Ceci doit être effectué bien sûr avec des outils commerciaux. Ceci évite tout d'abord de tout re-développer, et surtout facilite grandement la migration technologique. La migration technologique des circuits nécessite la non migration des outils !

Afin de pouvoir répondre à ces besoins, au lieu de développer un simulateur CHP - ce qui nous aurait donné plus de souplesse - , nous avons choisi de développer un traducteur CHP vers un langage de description de matériel (HDL). Cette approche permet de dépendre

uniquement de langages standardisés et non d'outils commerciaux (ce qui est le cas de la méthodologie basée sur StateChart [KOL96]). Nous aurions aussi pu enrichir un HDL existant afin de pouvoir spécifier et simuler des circuits asynchrones, méthode proposée dans [BLUN 00], [JACO00], mais nous avons éliminé cette solution pour supprimer les limitations des HDL et s'abstraire de leur syntaxe complexe et verbeuse (lourde en tant que langage de spécification, mais bien sûr utile pour la conception).

En terme de langages de description de matériel, nous avions le choix entre VHDL et Verilog qui sont couramment utilisés dans l'industrie. VHDL étant un réel standard IEEE et largement utilisé en Europe [AIRI90], c'est celui que nous avons choisi. Bien sûr le langage Verilog pourrait être utilisé de la même manière. Ceci nous permet alors d'avoir accès à la simulation à tous les niveaux : fonctionnelle, niveau porte, avant et après routage, ainsi que l'accès aux standard existants et/ou aux bibliothèques privées. Le langage CHP que nous avons adopté et enrichi est présenté dans le paragraphe 2.2. L'outil de traduction CHP_2VHDL est lui présenté dans le paragraphe 2.3.

Ensuite, au niveau du choix de la méthode de synthèse de circuits asynchrones, nous voulons de la même manière privilégier la réutilisabilité et la migration technologique. Comme nous le précisions dans le chapitre 1, notre choix s'est porté sur la conception de circuits asynchrones quasi-insensibles aux délais (QDI), afin d'offrir dans les technologies VLSI avancées une grande robustesse (vis-à-vis des variations/dispersions technologiques, températures, tension, crosstalk), que ce soit au niveau de la validité fonctionnelle d'un circuit dans une technologie donnée, ou de sa migration dans une technologie plus évoluée. Ensuite, afin d'être compatible avec les flots de conception synchrone standard et de pouvoir offrir de réelles possibilités de migration, il est absolument nécessaire d'offrir une méthode de synthèse basée sur l'utilisation de cellules standard, contrairement à des circuits asynchrones comme [MART97] [FURB97] qui sont réalisés en *full-custom*. La méthode de synthèse de circuits QDI que nous proposons est présentée dans le chapitre 3. Enfin, il est important de noter que cet aspect robustesse des circuits quasi insensible aux délais pourra être utilisé pour jouer sur le ratio vitesse / consommation du circuit obtenu [BERK 94], [NIEL 94].

L'idée générale autour du traducteur CHP vers VHDL que nous avons développé est donc de pouvoir spécifier un système dans un langage de haut niveau, utiliser des méthodes formelles associées, et enfin utiliser différents flots de simulation / synthèse. Ce concept permet d'analyser des programmes CHP et de générer différents style de matériel. Le traducteur génère aujourd'hui du VHDL fonctionnel pour la (co-)simulation. Néanmoins, d'autres traducteurs pourraient être inclus dans cet environnement afin de générer : a) du VHDL RTL ou comportemental pour obtenir des circuits synchrones, b) du VHDL modifié pour obtenir des circuits localement synchronisés (*bounded delay* [TAN98]), c) du CHP synthétisable tel que défini dans le chapitre 3 pour générer des circuits QDI, d) du langage « C » pour l'exécution sur des systèmes processeurs ou multi-processeurs.

La figure 2-1 présente une idée de ce que peut être un flot de conception / simulation pour circuit mixte synchrone / asynchrone.

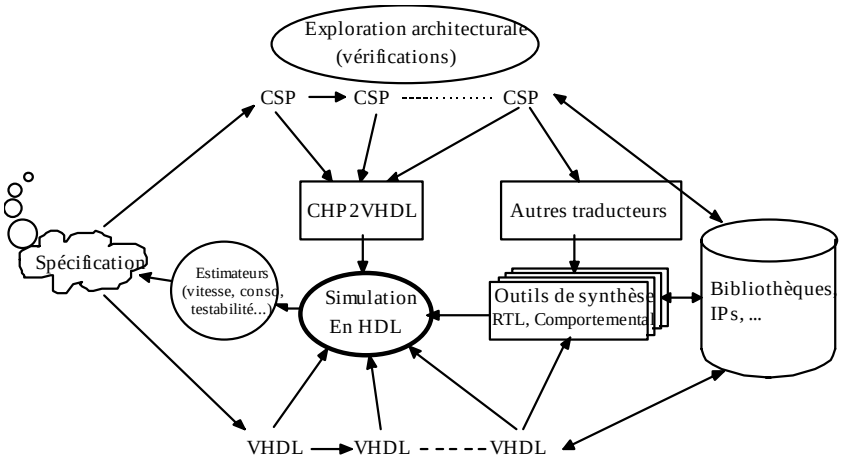


Figure 2-1 : Synopsis d'un flot de conception pour circuit mixte synchrone / asynchrone.

L'environnement de conception proposé profite des avantages des derniers développements des outils CAO, tout en nécessitant peu de développements spécifiques. L'outil CHP_2VHDL est donc un premier exemple de ce qui peut être fait pour proposer des méthodes de conception de circuits mixte. Ainsi, cet environnement possède deux chemins de spécifications distincts qui convergent vers un même flot de simulation HDL. Dans un tel cadre, il est alors possible de calculer des estimateurs depuis des résultats de simulations, et ensuite de partitionner le système en parties synchrones et asynchrones. La même spécification peut être simulée / synthétisée aussi bien en mode synchrone qu'en asynchrone. Ainsi, non seulement cet environnement permet d'associer les domaines synchrones et asynchrones, il offre aussi l'opportunité de les comparer et de conjointement analyser leurs avantages et inconvénients.

Cette idée de flot de conception est très générale : c'est la preuve qu'il reste encore beaucoup de travail de recherche en terme de méthodes et d'outils de conception. Outre les problèmes d'outils de synthèses spécifiques, il faudra en particulier s'attacher au problème de la génération automatique d'interfaces (et de leur vérification) entre les différents zones de fonctionnement du système mixte synchrone / asynchrone. Il est possible d'effectuer soit la synthèse d'interface matériel, soit la synthèse d'interface pour la simulation. Bien sûr, ceci est supporté par le flot de conception grâce à l'utilisation d'un HDL.

Cet aspect interface a été pris en compte dans le traducteur CHP_2VHDL afin d'insérer des interfaces spécifiques pour pouvoir effectuer de la co-simulation VHDL entre le CHP traduit et des *netlist* portes de circuits QDI (voir paragraphe 2.3.7). Enfin, le flot de conception que nous avons utilisé pour implémenter le processeur Aspro sera présenté plus en détail a la fin du chapitre 3.

2.2. Le langage CHP

Le langage CHP que nous avons défini respecte la proposition initiale d'Alain Martin [Mar90]. Néanmoins, la syntaxe d'origine a été complétée et étendue pour répondre à nos besoins de modélisation et de simulation. Tous ces choix visent à rendre le langage CHP plus puissant, plus lisible afin qu'il soit adapté à la vérification et à l'investigation par simulation.

L'objectif de cette section est de présenter la syntaxe et la sémantique du langage CHP telles que nous les avons définies et implémentées dans le traducteur CHP_2VHDL. L'accent sera donné en particulier sur les points nouveaux apportés au langage. Le traducteur et plus spécifiquement la méthode de traduction seront présentés dans la section 2.4.

Afin de faciliter la lecture, la convention suivante a été adoptée pour présenter la syntaxe : les mots clés sont écrits en majuscule, les identificateurs en minuscules, les règles en italiques, les choix séparés par un "/", les règles optionnelles étant entre parenthèses.

2.2.1. Type de données, Opérateurs et Variables

Les différents types disponibles sont les types de base suivants :

bit `(BIT)`, vecteur de bit `(BIT[n..p])`, entier `(INTEGER)` et réel `(REAL)`.

Les conversions de type doivent être explicites lorsqu'elles sont nécessaires. Par exemple, si x et y sont des variables respectivement déclarées réelle et entière, l'affectation `y:=x` doit être écrite `y:=(integer)x`.

Des vecteurs de ces types de base peuvent être déclarés de la manière suivante :

`x[n..p] :` *`type_de_base`* `;`

L'index d'un vecteur est toujours de type entier et une conversion de type doit être utilisée si nécessaire.

Aucun autre type n'est autorisé : pas de chaîne de caractères, pas de type composite (structure), pas de tableau de dimension multiple, pas de pointeur, etc … Le langage CHP se veut un langage de description de matériel et non un langage de programmation. Par conséquent, tout type de haut niveau doit être modélisé explicitement par l'utilisateur suivant ses propres besoins.

Les opérateurs classiques tels que les opérateurs arithmétiques (+, -, *, /, mod), les opérateurs logiques (and, or, xor, not), les opérateurs de comparaisons (=,/=,<,>,<=,>=) sont disponibles sur ces types. Les expressions sont composées à partir d'opérateurs, de littéraux et de variables. Il est aussi possible d'effectuer des appels de fonction dans les expressions (voir paragraphe 2.2.7).

Les variables doivent être déclarées avec leur type et peuvent être initialisées. La syntaxe est la suivante :

`VARIABLE var1, var2, … :` *`var_type`* `{ :=` *`var_init`* `} ;`

La zone de déclaration des variables est strictement limitée à l'intérieur des processus. Toute variable est donc locale à un processus, il n'est pas possible de déclarer de variables globales. En conséquence, si deux processus distincts veulent partager une variable, ils doivent communiquer via des canaux afin d'échanger le contenu de cette variable. Les canaux et les actions de communication font l'objet du paragraphe suivant.

2.2.2. Canaux et actions de communication

- **Déclaration de canal**

Si deux processus P0 et P1 veulent échanger une variable x, ils doivent communiquer en utilisant un canal. Ce canal doit être déclaré avec son type, la syntaxe est la suivante :

```
CHANNEL can1, can2, … : bit_type ;
```

Les canaux doivent avoir le type de la donnée transmise à travers celui-ci. Pour une question de simplicité et parce que c'est directement compatible avec une implémentation matérielle, le type des canaux est restreint au type bit et vecteur de bit. Ceci n'est pas réellement une limitation puisque la conversion de type est possible dans les processus. Il est aussi possible de déclarer des vecteurs de canaux, dans ce cas la syntaxe est :

```
CHANNEL can1[n..p], … : bit_type ;
```

Les vecteurs de canaux sont utiles pour créer des instances multiples de composant (voir paragraphe 2.2.6). Dans ce cas, chaque élément d'un vecteur de canaux est un unique canal et son index peut être calculé grâce à une fonction entière.

- **Actions de communication sur un canal**

Un processus est connecté à un canal lorsque celui-ci est spécifié dans sa liste de port. Introduisons par un exemple la syntaxe de déclaration de port de processus ainsi que celle des actions de communication :

```
PROCESS P0 PORT ( C : OUT {ACTIVE} bit_type )
VARIABLE x : bit_type;
*[ … C!x … ]

PROCESS P1 PORT ( C : IN {PASSIVE} bit_type )
VARIABLE y : bit_type;
*[ … C?y … ]
```

Cet exemple montre que le processus P0 est connecté au canal C en sortie et P1 est connecté au canal C en entrée. L'action de communication `"C!x"` spécifie que le processus P0 écrit le contenu de la variable x sur le port de sortie C, réciproquement l'action `"C?y"` spécifie que le processus P1 lit le port d'entrée C et stocke la valeur lue dans sa variable locale y. Le résultat des deux actions de communications est alors équivalent à une affectation du type y:=x.

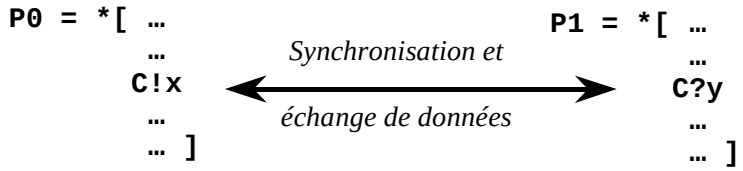


Figure 2-2 : communication entre processus.

La sémantique de synchronisation par communication telle qu'elle est définie dans le langage CSP [HOAR78] est respectée : lors des actions de communications "!" et "?", il y a synchronisation des deux processus P0 et P1. Au sens informatique du terme, on parle de *rendez-vous* : si P0 émet une donnée sur le port C et que P1 n'est pas prêt à recevoir, P0 est en

attente d'émission jusqu'à la disponibilité de P1, réciproquement si P1 est en réception sur le port C, la communication est suspendue jusqu'à ce que P0 soit en émission (voir Figure 2-2).

La définition de ce mode de synchronisation entraîne qu'un canal de communication ne possede pas de mémoire : le nombre de transaction effectuées sur un port de sortie est toujours égal au nombre de transaction effectuées sur le port d'entrée équivalent. Ce sont des communications par passage de message, elles s'effectuent jeton par jeton. Néanmoins, des mémoires peuvent être explicitement rajoutés en utilisant des processus. En repartant de l'exemple précédent, si on ajoute un canal CC, le processus P01 = *[C?c ; CC!c] et que P1 est à présent connecté au canal CC, la contrainte de synchronisation entre P0 et P1 est relâchée grâce au point mémoire correspondant à la variable "c" du processus P01.

Les canaux sont le seul moyen pour synchroniser des processus, qu'il y ait ou non échange de données. Dans le cas où il n'y a pas d'échange de donnée, le canal est déclaré avec le type `BIT` et la syntaxe est alors réduite à :

`*[ … C! … ]` et `*[ … C? … ]`

Pour les actions d'écriture, la syntaxe générale est "`C!`*`expression`*", l'expression pouvant être une expression quelconque, une variable ou tout simplement une constante.

Enfin, la notation réduite "`C?`" effectue l'action de lecture sur un canal sans mémoriser la valeur présente dans ce canal. Ce raccourcit d'écriture permet de modéliser l'évanouissement d'une donnée.

- **Déclaration de port et connexion des canaux**

La syntaxe générale de déclaration de port est la suivante :

```
PORT ( port1, port2, … : IN/OUT {ACTIVE/PASSIVE} bit_type ; …
)
```

Comme présenté dans l'exemple précédent, on doit spécifier la direction d'un port lors de sa déclaration. Cette direction (`IN` ou `OUT`) indique le sens de transmission de la donnée à travers le canal. Ainsi, une action d'écriture "!" ne peut s'appliquer qu'à un port de sortie et réciproquement une action de lecture "?" sur un port d'entrée. Les attributs de protocole `ACTIVE` et `PASSIVE` sont optionnels, ils permettent de spécifier quel port initialise la communication à travers le canal. Par défaut, un port de sortie est actif et initialise la communication tandis qu'un port d'entrée est par défaut passif et attends que la communication commence.

Les canaux sont toujours des liens de communication point à point entre processus. Les ports de connexion des canaux doivent en conséquence mettre en correspondance les attributs de direction et de protocole.

Ces différents attributs existaient dans la sémantique du langage CHP initial mais n'apparaissaient pas dans la syntaxe. La syntaxe style VHDL que nous avons choisie permet ainsi de spécifier lisiblement les attributs de canaux (en vue de la documentation du code source) et d'autoriser une vérification automatique par l'outil de traduction.

- **Opérateur de sonde**

L'opérateur de sonde (« probe » en anglais) est disponible pour tester l'activité d'un canal, il est utilisé dans les commandes gardées (voir paragraphe 2.2.5).

Si C est un port passif, l'opérateur "#C" (on dit sonde C) est une fonction booléenne qui rend vrai si une communication est en attente sur le port C. Ainsi, l'introduction des attributs de protocole nous permet de sonder aussi bien des ports d'entrée que des ports de sortie. Sonder un port d'entrée permet de tester si une donnée est en attente sur le canal tandis que sonder un port de sortie permet de tester si le processus distant demande une donnée via ce canal. L'opérateur # est seulement défini pour les ports passifs, il n'est donc pas possible de sonder les 2 ports d'un même canal, ce qui évite l'inter blocage entre les processus.

De plus, la syntaxe de l'opérateur de sonde a été étendue pour les ports d'entrée passifs. L'expression "#C=*valeur*" rend vrai si une communication est en attente sur le canal C et que la donnée présente dans le canal est égale à *valeur*. Ce mécanisme permet de tester la donnée présente dans un canal sans terminer la communication sur ce même canal (voir exemple dans le chapitre 3).

- **Conclusion**

En conclusion, on notera la différence entre canal et signal. Un signal est une notion physique. Dans un système électronique, le signal est un signal électrique, il code une donnée mais ne contient pas d'information de synchronisation, il peut faire l'objet de connexion multipoints, y compris avec des possibilités de contention (bus 3 états, …). Au contraire, canaux et actions de communication sont des notions abstraites issues de l'informatique. Quelque soit leurs implémentations logicielles ou matérielles, les canaux sont des liens point à point entre processus, ils permettent d'échanger des données et surtout de synchroniser et donc rendre séquentiel les processus.

2.2.3. Instructions élémentaires

Les instructions élémentaires suivantes sont disponibles dans les processus :

- Les actions de communications "!"et "?" telles que définies dans le paragraphe précédent.
- L'assignation de variable : y:=f(x)
- Une instruction nulle dénommée SKIP.
- Les instructions de contrôle (voir paragraphe 2.2.5)
- Les appels de procédures (voir paragraphe 2.2.7)
- Une instruction pour spécifier des délais : WAIT {*delay*}. Si aucun délai n'est donné, le processus est suspendu jusqu'à la fin de la simulation.
- Des instructions ont été rajoutées pour générer des traces pendant la simulation. PRINT("*string*"} et ERROR("*string*") génèrent le message *string* en cours de simulation et respectivement continuent ou arrêtent la simulation.

2.2.4. Opérateurs de composition.

Deux opérateurs de composition d'instructions sont disponibles. Ce sont les opérateurs séquentiel et concurrent, respectivement dénoté ";" et ",". (L'opérateur de coïncidence "•" [Mar90] n'est pas disponible). Supposons que S1 et S2 soient deux instructions (ou blocs d'instructions), on a :

- "S1 ; S2" signifie que S2 ne s'exécutera que lorsque S1 sera terminée.
- "S1 , S2" signifie que S1 et S2 peuvent potentiellement s'exécuter en parallèle.

L'opérateur de séquentialité est associatif tandis que l'opérateur de concurrence est associatif et commutatif. Ce dernier a la plus grande priorité. Il est possible d'expliciter des priorités différentes en utilisant des crochets, par exemple : "`S1 , [ S2 ; S3 ]`".

Pour l'instant et ce pour des raisons de difficulté de traduction en VHDL, l'opérateur de concurrence "," est limité dans le traducteur CHP_2VHDL aux actions de communications (voir justification paragraphe 2.3.2.).

2.2.5. Commandes gardées, instructions de contrôle et déterminisme

En CHP, la modélisation du contrôle (instructions conditionnelles, boucles, etc …) emprunte la syntaxe des commandes gardées de E. Dijsktra [Dij75].

Supposons que G soit une fonction booléenne et S une liste d'instructions, la commande gardée "`G => S`" signifie que la commande S est exécutée si et seulement si la garde G est vraie.

A partir de cette syntaxe, il est construit deux types de structure de contrôle : des sélections et des répétitions. Pour chacune d'entre elles, il est possible de spécifier ou non de l'indéterminisme. On obtient donc quatre types de structure qui sont : (1) une sélection déterministe, (2) une répétition déterministe, (3) une sélection indéterministe, (4) une répétition indéterministe. Soient n gardes G_1, … , G_n et n instructions S_1, … , S_n, la syntaxe est la suivante :

```
(1)        [ G1 => S1 @  ... @  Gn => Sn
             { @ OTHERS => SOthers } ]
(2)       *[ G1 => S1 @  ... @  Gn => Sn ]
(3)        [ G1 => S1 @@ ... @@ Gn => Sn ]
(4)       *[ G1 => S1 @@ ... @@ Gn => Sn ]
```

Pour une sélection déterministe (1), on doit toujours avoir au plus une garde de vraie à tout instant. Le schéma d'exécution d'une sélection est le suivant : suspension du processus jusqu'à ce que l'une des gardes rende vrai puis exécution de l'instruction correspondant à cette garde.

Cependant, la suspension est effective uniquement lorsque des sondes sont présentes dans les gardes. Dans le cas contraire, lorsque les gardes ne portent que sur des variables, tous les éléments sont disponibles pour évaluer les gardes, la sélection est alors immédiatement exécutée. Dans ce dernier cas, on doit donc avoir au moins une et au plus une garde de vraie, sinon il y a erreur. Afin de simplifier l'écriture des gardes, le mot clé optionnel `OTHERS` a été introduit. Les instructions par défauts ainsi spécifiées sont exécutées quand aucune garde n'est vraie dans la sélection. La Figure 2-3 ci dessous donne différents exemples de sélections déterministes.

`[ #A => A?x ; S!x`
`@ #B => B?x ; S!x`
`]`
	En raison des sondes, l'exécution est suspendue jusqu'à ce que #A ou #B rende vraie, c'est à dire qu'une donnée soit présente sur l'un des deux canaux A ou B (et uniquement un, c'est la condition de déterminisme).

`[ x=0 => S0`
`@ x=1 => S1`
`]`
	Dans ce cas, il n'y a pas suspension du processus, les gardes sont évaluées et l'instruction correspondant à la garde vraie est exécutée. Si x vaut 2, il y a erreur.

`[ x=0 => S0`
`@ x=1 => S1`
`@ others => SKIP`
`]`
	Ici, un mécanisme par défaut est modélisé. Si x=0, on exécute S0. Si x=2, on exécute l'instruction nulle (`SKIP`) et il n'y a pas erreur.

Figure 2-3 : Exemples de sélections déterministes

Dans une répétition déterministe (2), il doit toujours y avoir au plus une garde de vraie. Le schéma d'exécution d'une répétition est le suivant : suspension du processus jusqu'à ce qu'une garde soit vraie, puis répétition de l'exécution de l'instruction correspondant à la garde vraie (pas nécessairement la même) et ce jusqu'à ce que plus aucune garde ne soit vraie.

A partir de la syntaxe de répétition, une boucle infinie peut s'écrire : `*[ true => … ]`. Par souci de simplification, le raccourci d'écriture suivant est introduit :

```
*[ … ]
```

Pour les sélections (1) et les répétitions (2), des vérifications sont effectuées sur les gardes par le traducteur (voir paragraphe 2.3.6). Les propriétés suivantes sont vérifiées : l'exclusivité entre les gardes, l'utilisation correcte du mot clé `others`, le fait qu'au moins une garde soit toujours vraie. En cas d'erreur, un message est généré.

- **Indéterminisme**

Pour les structures de choix (1) et (2), les gardes sont toujours déterministes : le programme doit garantir que les gardes soient toujours stables et mutuellement exclusives. Ceci n'est bien sûr pas toujours le cas. Les exemples de la Figure 2-4 montrent des gardes qui sont non stables ou non exclusives. Dans ces cas, il y a nécessité d'un arbitrage au niveau matériel et ceci doit être spécifié dans le modèle en utilisant les sélections et répétitions non-déterministes (3) et (4).

Leur schéma d'exécution est équivalent à (1) et (2) sauf que plusieurs gardes peuvent être évaluées vraies au même instant et qu'une unique garde est arbitrairement sélectionnée parmi les gardes en conflits.

`[ #INT =>`
`  -- Traitement interruption`
`@@ not(#INT)  =>`
`  -- Traitement instruction`
`  -- suivante`
`]`
	Test d'interruption dans un microprocesseur. C'est un exemple de gardes instables : à tout moment lors du test de la non-présence d'interruption, il peut arriver une interruption.

`[  #A => A?x ; S!x`
`@@ #B => B?x ; S!x`
`]`
	Exemple de multiplexeur où l'environnement ne garantit pas l'exclusivité entre les canaux A et B : l'indéterminisme apparaît au niveau de la spécification.

Figure 2-4 : Exemples de sélections indéterministes

Pour simuler en VHDL les structures indéterministes, une fonction aléatoire de choix est utilisée (voir paragraphe 2.3.6). Afin de pouvoir simuler des fonctions aléatoires qui soient non-corrélées, l'utilisateur a la possibilité de spécifier un paramètre réel qui est la graine de la fonction aléatoire. La syntaxe est la suivante :

`[ structure_non_déterministe ](seed)`

- **Analyse probabiliste des gardes.**

Afin de pouvoir étudier le comportement statistique des programmes ("*code profiling*"), l'utilisateur peut générer en cours de simulation grâce à un drapeau "`traceon`" des fichiers contenants des informations sur les gardes. La syntaxe est :

`[ structure_de_choix ](traceon)`

Pour les structures de choix déterministes, on écrit dans un fichier le nom du processus, un label pour la structure, la date et la garde sélectionnée, pour les structures déterministes on écrit de plus la liste des gardes en contention. Ces informations seront donc générées en cours de simulation : elles seront spécifiques à l'application et aux données traitées, elles pourront ensuite être triées et analysées. On obtient ainsi facilement les probabilités sur les différentes gardes, informations qui permettent par la suite d'optimiser la spécification ainsi que le processus de synthèse.

- **Conclusion**

Les commandes gardées ont été très peu utilisées en informatique car elles sont relativement coûteuses à exécuter sur des machines séquentielles. Si on compare avec des structures "si alors sinon", il est en effet nécessaire d'évaluer toutes les gardes avant de pouvoir décider quelle branche exécuter. On pourrait les assimiler aux structures classiques de choix "case", mais leur syntaxe est plus riche car les gardes peuvent porter sur n'importe quelle équation booléenne et non seulement sur des constantes entières. Ces structures de contrôle comportent de plus une information de synchronisation dans le cas où les gardes portent sur des canaux (sémantique de l'opérateur de sonde). Les structures de sélection et de répétition sont donc parfaitement adaptées à la spécification de matériel et apportent une très grande lisibilité fonctionnelle. Enfin, ce formalisme permet de tenir compte de l'indéterminisme dès la spécification, ce qui n'est pas le cas avec les langages de programmation (logiciel ou matériel) classique. On peut remarquer que seul l'indéterminisme est spécifié ici, et non la manière d'arbitrer le conflit.

2.2.6. Modélisation structurelle

La modélisation structurelle était quasi inexistante dans la proposition initiale du langage CHP [MAR 90], tous les processus étaient déclarés à plat. Afin de pouvoir gérer la complexité des problèmes VLSI, il a été défini une approche de modélisation hiérarchique. La syntaxe présentée ici est bien sûr empruntée aux langages de description de matériel mais légèrement adaptée afin d'obtenir une syntaxe simple et régulière. La hiérarchie est seulement basée sur des processus et des composants. C'est effectivement moins modulaire mais beaucoup moins prolixe qu'en VHDL où il faut à la fois déclarer entité, architecture, composant, processus et configuration. La connectivité entre blocs repose uniquement sur des canaux (voir 2.3.2), il n'est pas possible de déclarer de signaux.

- **Déclaration de composant**

Un composant est la vue globale de la spécification. Sa vue externe déclare, si il y en a, des paramètres génériques et des ports de communication tels que définis dans 2.3.2. Le corps d'un composant est constitué d'instructions concurrentes tel que des instances de composants et des déclarations de processus. Les canaux de communications doivent être déclarés, ils permettent de connecter les différents processus et instances de composants. La syntaxe de déclaration d'un composant est la suivante :

```
COMPONENT Comp_name
   Déclaration_de_paramètres_génériques
   Déclaration_de_ports_de_communication
Déclaration_de_canaux
BEGIN
    Instances_de_composants
    Déclarations_de_processus
END Comp_name;
```

- **Instance de composant**

Tout composant déclaré peut être instancié dans tout autre composant. L'instance de composant peut-être simple ou multiple. La syntaxe d'une instance simple est la suivante :

```
Inst_name : Comp_name GENERIC MAP ( par1, par2, … )
                      PORT MAP ( chan1, chan2, … )
```

Comp_name est le nom du composant à instancier et *Inst_name* son nom d'instance. L'instance d'un composant crée alors un processus équivalent à ce composant avec une mise en correspondance des paramètres génériques et des ports de communication. Le passage des paramètres, que ce soit pour les génériques ou les ports de communication, se fait toujours dans l'ordre de la déclaration initiale de ce composant.

L'instance multiple est un moyen efficace pour créer plusieurs instances d'un même composant en une seule fois. Sa syntaxe est :

```
< label : FOR index IN n TO p : instance_simple >
```

Dans ce cas, les canaux utilisés pour effectuer la connexion multiple sont des vecteurs de canaux dont les index sont calculés grâce à des fonctions entières portant sur *index*. Par exemple, si on veut connecter 10 inverseurs en anneau, on écrit :

```
< n_inverseurs : FOR i IN 1 TO 10 :
  inv_i : inverseur PORT MAP ( C[i], C[(i+1) mod 10] )
>
```

- **Déclaration de processus**

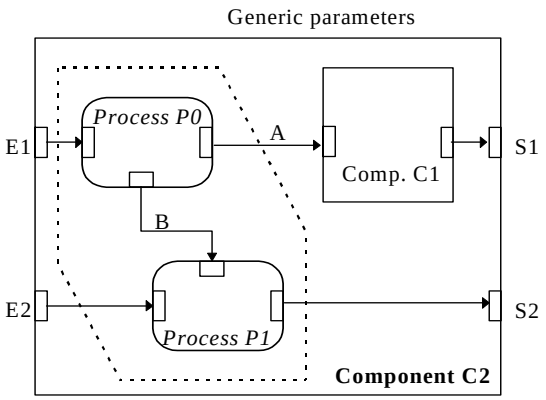
Les processus sont déclarés dans les composants. Contrairement au langage VHDL, chaque processus doit déclarer ses ports de communications, tout comme un composant. Ceci permet de déclarer proprement les attributs des canaux (direction et protocole) pour ensuite effectuer les vérifications nécessaires. Lorsque le processus est l'unique processus d'un composant, les ports de celui-ci sont par défaut ceux du composant et n'ont pas besoin d'être déclarés. Le processus doit définir ses variables locales et son corps est constitué

d'instructions : il peut contenir une partie initialisation ainsi qu'une boucle infinie. La syntaxe de déclaration d'un processus est la suivante :

```
PROCESS Process_name
  Déclarations_de_port
  Déclarations_de_variable
[ initialisations ;
  *[ … instructions … ]
]
```

Dans un processus, les opérateurs séquentiel et parallèle tel qu'ils sont définis dans 2.2.4 peuvent être utilisés. On rappelle que processus et instances de composants sont eux par nature tous concurrents, leur synchronisation étant effectuée grâce à des actions de communication sur des canaux. La

Figure 2-5 donne un exemple de modélisation structurelle.




```
COMPONENT C2 GENERIC ( n : integer )
            PORT ( E1,E2 : IN bit ;
                   S1,S2 : OUT bit[1..n] )
CHANNEL A, B : bit ;
BEGIN

PROCESS P0 PORT ( E1: IN bit; A,B: OUT bit )
variable x : bit;
[ A!0 ;
  *[  E1?x ; ... ; A!x , B!(x+1)  ]
]

C1_in_C2 : C1 PORT MAP ( A, S1 )

PROCESS P1 PORT ( E2,B: IN bit; S2: OUT bit )
*[  E2? , B? ; ... ; S2!   ]

END C2 ;
```

Figure 2-5 : Exemple de modélisation structurelle, schéma et code CHP correspondant.

La modélisation structurelle offerte par cette syntaxe est suffisamment riche pour envisager l'écriture de problème complexe. On peut noter en conclusion les différences suivantes avec le langage VHDL : i) la notion VHDL de configuration et de couple entité/architecture n'est pas disponible mais peut s'implémenter avec une politique de renommage des composants ; ii) les possibilités de synchronisation/parallélisme en CHP sont plus riches sachant que l'opérateur de concurrence est disponible dans les processus (et ce malgré la restriction introduite dans 2.3.2.) ; iii) les entêtes de processus CHP doivent être déclarées contrairement au VHDL ce qui permet une meilleure lisibilité structurelle et fonctionnelle de chaque processus.

2.2.7. Utilisation de facilités des HDLs

En CHP, il n'est pas possible de déclarer de procédures ou de fonctions. L'utilisateur peut et doit modéliser l'appel de primitives en utilisant des processus. Il effectue ainsi sa propre gestion des ressources matérielles : accès partagé à une unique ressource ou duplication des ressources. Cependant, de manière à ouvrir le langage CHP aux facilitées offertes par les langages de description de matériel, des fonctions et procédures HDLs peuvent être appelées dans un programme CHP. Ce choix délibéré permet de préserver la simplicité et la régularité du langage CHP tout en offrant accès aux packages du HDL visé : fonctions mathématiques,

arithmétiques, logiques, gestions des entrées-sorties, accès au fichiers, etc … Dans ce cas, l'utilisateur doit rajouter une clause "`USE this_hdl_pkg;`" dans l'entête du composant pour préciser qu'un package HDL est utilisé.

Les appels de fonctions et procédures doivent respecter la règle suivante : les appels ne sont possibles que dans le corps des processus et leurs paramètres d'entrées-sorties ne peuvent porter que sur des objets déclarés en CHP : variables, paramètres génériques, constantes. Cette limitation n'est bien sûre pas suffisante pour empêcher l'utilisateur d'écrire du code très bizarre, y compris avec des effets de bord. Mais c'est un moyen très pratique. Par exemple, lors de la simulation d'un design complexe composé de multiples processus concurrents, on peut vouloir extraire des estimateurs globaux. Ceci peut être effectué en utilisant des variables globales mises à jour par appel de procédure depuis tous les processus. L'exemple classique est la génération de traces dans un fichier.

2.2.8. Conclusion

Le langage CHP que nous venons de présenter est très simple. Les types et opérateurs utilisés sont relativement restreints, il n'y a pas de fonctions ou de procédures, la modélisation hiérarchique est empruntée au langage VHDL mais réduite au stricte minimum. Toutes les fonctionnalités doivent donc être décrites avec des processus concurrents synchronisés par passage de message à travers des canaux. Dans le langage, les canaux restent des notions abstraites, il n'est pas explicité comment communiquer et synchroniser les processus. Enfin, le contrôle se décrit avec des commandes gardées, ce qui offre une grande lisibilité fonctionnelle et permet de spécifier l'indéterminisme. Ce langage parait à première vue limité, il est pourtant très puissant, comme le montrera le chapitre 4 sur l'architecture du processeur Aspro pour étudier des systèmes matériels asynchrones.

2.3. Le traducteur CHP_2VHDL

Dans cette section, le traducteur CHP vers VHDL est présenté. Dans un premier temps, nous nous attachons à décrire la vue d'ensemble de l'outil et ses concepts de base. Ensuite, on explicite comment ont été mises en correspondance les sémantiques du langage CHP et du langage VHDL pour effectuer la traduction. On s'attachera en particulier aux problèmes de traduction de la concurrence, des canaux et des actions de communications, des structures de contrôle, et enfin au problème de la co-simulation.

2.3.1. Présentation de l'outil

Le traducteur CHP_2VHDL a été écrit en C et représente environ 3000 lignes de code. La grammaire du langage CHP telle que présentée dans le paragraphe précédent a été implémentée avec les outils Unix lex et yacc. Une vue d'ensemble de l'architecture de l'outil et de son environnement est donnée dans la Figure 2-6.

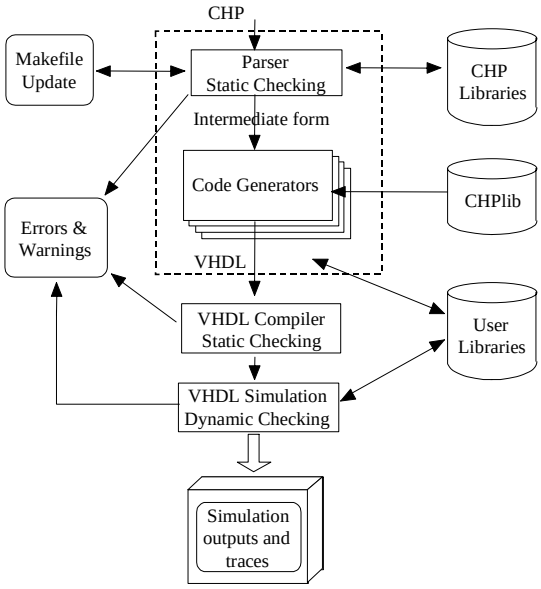


Figure 2-6 : Architecture et environnement du traducteur CHP$_2$VHDL

Le schéma de fonctionnement de l'outil est le suivant : l'analyseur grammatical lit le fichier source CHP et génère une forme intermédiaire. A ce moment, les erreurs de syntaxe sont rapportées. Comme expliqué dans le paragraphe 2.3.2., l'opérateur de concurrence "," est extrait et l'outil génère un message si il ne peut pas être implémenté. Ensuite, la forme intermédiaire du CHP est vérifiée : déclaration correcte des génériques, des ports, des canaux, des variables, des instances de composants, des types. Des vérifications complémentaires sont effectuées sur les canaux : les canaux sondés doivent être passifs, actions de lecture/écriture respectivement sur des canaux d'entrée/sortie, mise en correspondance des attributs de direction et protocole pour tous les canaux, cohérence du « port mapping ». Toutes ces vérifications sont faites au moment de la compilation. Les vérifications sur les gardes sont elles effectuées pendant la simulation (voir paragraphe 2.3.6). Enfin, un fichier "makefile" associé au projet courant est mis à jour si nécessaire (voir paragraphe 2.3.8).

La forme intermédiaire est alors traduite par le générateur de code VHDL. Un fichier .vhd est généré par déclaration de composant CHP. Comme montré dans la Figure 2-6, d'autres générateurs de code pourrait facilement être ajoutés à ce niveau : Verilog d'une manière très similaire, langage C, etc …

De façon à fournir un environnement de simulation qui soit flexible et facilement modifiable par l'utilisateur, le VHDL généré fait appel à des fonctions et procédures. Ces primitives sont rassemblées dans un package appelé CHPlib qui doit être fourni avec le générateur de code. Comme montré dans la suite, ce package offre le maximum de flexibilité à l'utilisateur afin qu'il puisse construire et adapter son propre environnement de simulation. Un package CHPlib par défaut a été développé avec l'outil. Il contient les définitions de type, les procédures de communication/synchronisation, les fonctions de choix pour les structures de contrôle, les procédures de trace, les fonctions aléatoires et enfin les composants d'interface pour effectuer des co-simulations. Tous ces points sont justifiés et explicités dans les paragraphes qui suivent.

Enfin, des packages contenant des procédures et fonctions du HDL visé peuvent être appelés depuis le code CHP, comme montré dans le paragraphe 2.2.7.

2.3.2. Modélisation de la concurrence

De manière à traduire le CHP en VHDL, le premier problème qui se pose est la manière de modéliser la concurrence. En CHP, l'ensemble des processus et composants sont concurrents, sachant que la concurrence est aussi autorisée dans les processus grâce à l'opérateur ",". En VHDL par contre, les composants (ou plutôt couple entité/architecture) et processus sont aussi concurrents mais les processus VHDL sont eux strictement séquentiels. Il n'y a donc pas de mise en correspondance directe de la concurrence entre les deux langages.

Deux solutions sont possibles. La première est de traduire un processus CHP par un couple entité/architecture en VHDL. Cela permet d'obtenir le parallélisme de manière immédiate. Le séquencement doit alors être introduit par décomposition de processus en utilisant des canaux. L'opérateur ";" doit donc être extrait par l'outil, ce qui nécessite une analyse complète des dépendances de données. La solution duale est de traduire un processus CHP par un processus VHDL. Dans ce cas, le séquencement ne coûte rien. Par contre, l'opérateur de concurrence "," doit être extrait par décomposition de processus, ce qui demande aussi une analyse des dépendances de données.

Afin de réduire les efforts de développement logiciel, nous avons retenu la deuxième solution avec la limitation suivante : *l'opérateur de concurrence ne peut porter que sur des actions de communications, dont les canaux ne sont pas sondés.*

Cette restriction réduit grandement la complexité de l'analyse des dépendances de données. Le traducteur effectue alors automatiquement la décomposition suivante quand il rencontre l'opérateur "," sur des actions de communications. Soit par exemple le processus P0 avec deux actions de communications concurrentes, une lecture et une écriture :

```
P0 = *[ … ; [ A!a , B?b ] ; … ]
```

Les deux actions de communications sont extraites en créant quatre canaux et deux processus PA et PB, tandis que P0 devient P0' :

```
PA = *[ CA?x ; A!x ; RA? ]
PB = *[ CB?  ; B?x ; RB!x ]
P0' = *[ … ; [ CA!a ; CB! ; RA? ; RB?b ] ; … ]
```

Dans le processus P0', la séquence [CA!a; CB!] lance l'exécution des processus PA et PB, tandis que la séquence [RA?; RB?b] synchronise leur terminaison. Pour préserver la correction fonctionnelle et temporelle de la spécification initiale, les actions de communications ajoutées sont configurées pour ne pas consommer de temps de simulation (voir paragraphe 2.3.5). Ainsi l'ordre dans les séquences précédentes est totalement arbitraire. Associé à cette transformation, l'outil substitue toute autre référence aux canaux A et B avec la nouvelle séquence équivalente.

Cette transformation se généralise facilement à un nombre quelconque d'actions de communications en parallèle. Dans le cas où une ", " ne peut être traduite, l'outil génère un message et la remplace par un ";". C'est bien sûr de la responsabilité du concepteur de vérifier si la simulation sera valide ou non.

En conclusion, on peut remarquer que même si l'opérateur de concurrence n'est pas strictement nécessaire en terme de modélisation (en pur CSP, il n'existe pas [Hoa78]), il est très utile pour composer des actions de communications. Son absence nécessiterait de vérifier l'ordre des communications pour éviter tout inter blocage. De plus, cela permet aussi de faire soi-même une décomposition en processus et de lancer en parallèle l'exécution des processus "fils". Malgré cette implémentation limitée de l'opérateur de concurrence, cette version de l'outil s'est révélée performante et n'a pas introduit de limitations vis-à-vis de la modélisation.

2.3.3. Modélisation structurelle

Grâce au choix de traduction de la concurrence en CHP, la traduction du modèle structurel est immédiate. Tout processus CHP est traduit par un processus VHDL, un composant CHP est traduit par un couple entité/architecture VHDL. Pour les composants CHP sans ports, à savoir les entités de test (*testbench*), une configuration VHDL par défaut est générée pour effectuer la simulation. Dans le cas contraire, une vue composant VHDL du composant CHP est générée en prévision d'une future instance de ce composant.

La connexion des canaux entre processus CHP est elle traduite par les signaux équivalents permettant d'implémenter les canaux (voir paragraphe 2.3.4.). Une instance de composant est donc traduite par l'instance du composant VHDL équivalent avec ses signaux correspondants. Pour les instances multiples, on utilise le mot clé `GENERATE` qui est l'équivalent VHDL.

2.3.4. Canaux et actions de communication

En VHDL, aucune sémantique de synchronisation n'existe telle que définie en CSP. Les pilotes de signaux VHDL [AIRI 90] scrutent les évènements sur les signaux uniquement à la date de leur émission mais ne les mémorisent pas. Il n'est donc pas possible d'utiliser un unique signal pour modéliser un rendez-vous sur un canal : il faut expliciter la synchronisation, soit par sémaphore, soit par protocole de requête/acquittement tel qu'implémenté en matériel. C'est cette deuxième solution que nous avons adoptée pour rester compatible avec une approche circuit. Tout comme l'implémentation matérielle des canaux (voir chapitre 3), il faut choisir l'encodage des canaux et le protocole de communication utilisé.

- **Encodage des canaux**

Le minimum à générer par canal est deux signaux : un signal pour coder la requête et la donnée, et un signal pour l'acquittement. Pour ces deux signaux, il n'est malheureusement pas possible d'utiliser un unique signal VHDL composé d'un champ donnée et d'un champ acquittement, car ils sont de directions opposées. L'encodage de la requête et de la donnée se fait facilement en utilisant un type VHDL énuméré. Les signaux sont du type `CHP_BIT`, type définit dans le package CHPlib. Le signal de donnée est du type du canal (bit ou vecteur de bit) tandis que le signal d'acquittement est toujours codé sur un seul bit. Pour les vecteurs de canaux, on génère les vecteurs de signaux correspondants.

Par exemple, le canal "`Channel A : BIT[0..n];`" est traduit par l'ensemble des deux signaux VHDL suivants : "`Signal Adata: CHP_BIT(0 to n); Signal Aack: CHP_BIT;`".

- **Actions de communication**

Pour traduire une action de communication, le minimum à générer est un appel de procédure avec les paramètres suivants : le signal de donnée, le signal d'acquittement, la valeur à lire ou à écrire. Par exemple, si A est un port de sortie actif, l'action de communication `A!x` sera traduite par l'appel de procédure suivant :

```
CHP_write_active(Adata,Aack,x);
```

A part cette liste de paramètres qui est tout le temps générée, tout autre paramètre peut être rajouté par l'utilisateur. Ce moyen permet par exemple de spécifier des délais dans les

communications (voir paragraphe 2.3.5). Grâce à l'utilisation de procédures, l'utilisateur a une grande flexibilité et peut implémenter n'importe quel type de protocole à poignée de main.

- **Protocoles**

Suivant les conventions précédentes, un jeu de procédures de communication a été développé en VHDL et est disponible dans le package CHPlib. Il y a quatre types de procédures, suivant les attributs des ports `IN/OUT` et `ACTIVE/PASSIVE`, ainsi que deux fonctions pour les sondes (« probe »). Ces primitives sont de plus surchargées suivant le type du canal : bit ou vecteur de bit.

Pour une question de simplicité, on a choisi un protocole quatre phases avec un codage trois états [Hau95]. Les signaux de données sont codés en utilisant le type VHDL *std_ulogic*. La valeur zéro est codée avec '0', la valeur un avec '1', l'état invalide étant codé avec 'Z'. Le niveau 'U' est gardé pour les signaux non-initialisés et 'X' est gardé pour les exceptions arithmétiques.

Suivant l'attribut `ACTIVE/PASSIVE` qui spécifie quel port du canal initialise la communication, on définit deux jeux de procédures de lecture et d'écriture. Pour le protocole utilisé par défaut (voir Figure 2-7), les procédures CHP_write_active() et CHP_read_passive() sont définies en VHDL de la manière suivante :

```
CHP_write_active( Cdata : out CHP_BIT;
                  Cack  : in CHP_BIT;
                  var   : in CHP_BIT )
begin
  Cdata <= var;
  wait until Cack='0';
  Cdata <= 'Z';
  wait until Cack='1';
end;
```

```
CHP_read_passive( Cdata : in CHP_BIT;
                  Cack  : out CHP_BIT;
                  var   : out CHP_BIT)
begin
  wait until CHP_probe(Cdata);
  var := Cdata;
  Cack <= '0';
  wait until CHP_invalid(Cdata);
  Cack <= '1';
end;
```

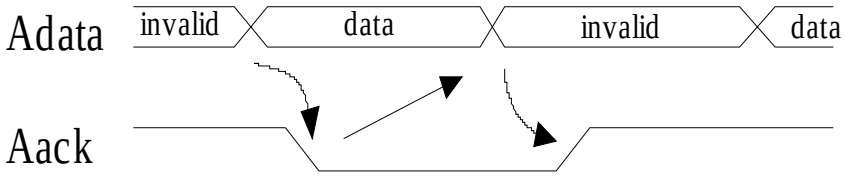


Figure 2-7 : Protocole quatre-phases "Out Active / In Passive"

Les fonctions CHP_probe() et CHP_invalid() permettent respectivement de sonder le canal (voir paragraphe 2.2.2) ou de tester son inactivité. CHP_probe() rend vrai si une donnée est présente dans le canal, réciproquement CHP_invalid() rend vrai si tous les bits du canal sont à l'état invalide.

Réciproquement pour le protocole symétrique, les procédures CHP_write_passive() et CHP_read_active() s'écrivent :

```
CHP_write_passive( Cdata : out CHP_BIT;
                   Cack  : in CHP_BIT;
                   var   : in CHP_BIT )
begin
  wait until Cack='0';
  Cdata <= var;
  wait until Cack='1';
  Cdata <= 'Z';
end;
```

```
CHP_read_active( Cdata : in CHP_BIT;
                 Cack  : out CHP_BIT;
                 var   : out CHP_BIT )
begin
  Cack <= '0';
  wait until CHP_probe(Cdata);
  var := Cdata;
  Cack <= '1';
  wait until CHP_invalid(Cdata);
end;
```

Lorsque les actions de communications sont réduites à "C!" et "C?", les procédures sont équivalentes mais sans leurs paramètres de valeur (on envoie un 1 par défaut).

En conclusion, on peut noter qu'il serait facile d'implémenter un protocole deux phases en utilisant un encodage quatre états. Toujours avec le type énuméré *std_ulogic*, la valeur zéro serait codée avec '0' et 'L', la valeur un avec '1' et 'H'. Néanmoins, ce type de protocole serait moins lisible en simulation que le protocole quatre phases. L'implémentation des actions de communications tel que proposée dans le package CHPlib a donné des simulations facilement compréhensibles, ce qui aide la phase de *debug* dans l'écriture des modèles.

Grâce au jeu de primitives du package CHPlib, la simulation des actions de communications est facilement modifiable par l'utilisateur, suivant ses propres besoins. Il faut tout de fois rappeler que le protocole visé ici est un protocole pour la simulation de modèle CHP, ce protocole n'a donc pas de raison d'être strictement identique à son implémentation matérielle. Pour répondre au problème de la co-simulation, les aspects interfaces de protocoles entre simulation d'un modèle CHP et simulation d'un résultat de synthèse seront présentés dans le paragraphe 2.3.7.

2.3.5. Modélisation temporelle

A l'origine, le CHP ne possède pas de clauses de spécifications temporelles. Deux raisons nous ont poussées à ajouter une fonctionnalité de modélisation temporelle. La première évidente est que les concepteurs ont besoin d'analyser le comportement temporel de leur design. La seconde plus subtile est qu'une simulation VHDL s'effectue en un unique delta délai [AIRI 90] si aucun délai n'est spécifié et donc la simulation ne peut être observée.

Comme présenté dans le paragraphe 2.2.3, il est tout d'abord possible de spécifier un délai en utilisant la syntaxe "`Wait {delay}`". Ceci permet de spécifier un délai associé à n'importe quel type de traitement (arithmétique, etc …).

De plus, il est possible de spécifier les délais associés aux actions de communications. Les délais associés aux différentes phases du protocole de communication peuvent être préciser de deux manières : tout d'abord en fixant les constantes de temps dans le package CHPlib, ou directement en donnant une liste de délais pour chaque action de communication comme "`A!(20)a`" ou "`A!(20,15,20,15)a`". Comme le traducteur génère simplement un appel de procédure, il suffit de surcharger les procédures de communications VHDL du package CHPlib avec la liste de paramètres et le comportement correspondants. Par exemple, l'action d'écriture "`A!(d1,d2)a`" est traduit par :

```
CHP_write_active(Adata,Aack,a,d1,d2);
```

tandis que la procédure sera définie par :

```
CHP_write_active(Cdata,Cack,x,t_valid,t_invalid)
  Cdata <= x after t_valid;
  Wait until Cack='0';
  Cdata <= 'Z' after t_invalid;
  Wait until Cack='1';
```

Par défaut, quand aucun paramètre n'est spécifié, les procédures utilisent les constantes de temps par défaut déclarées dans le package CHPlib. Ceci résout le problème de la simulation dans un delta délai. Ce mécanisme de passage de paramètre offre aussi la possibilité à l'utilisateur de donner des délais aléatoires, puisque n'importe quel paramètre de temps peut-être substitué par l'appel à une fonction aléatoire.

En conclusion, la modélisation temporelle fournie par l'outil est suffisamment flexible pour autoriser n'importe quel type de génération et simulation de délais : simulation par zone de fonctionnement (température / tension / techno), délais bornés, délais aléatoires. Enfin, cela permet aussi de rétro-annoter un modèle fonctionnel CHP avec des valeurs de temps issues d'un résultat de synthèse (que ce soit par analyse de timing statique ou dynamique) afin d'effectuer une simulation fonctionnelle temporelle, donc à la fois rapide et précise.

2.3.6. Les structures de contrôle

La traduction VHDL des structures de contrôle CHP est relativement directe. Le code VHDL généré pour les quatre type de structure de contrôle (sélection / répétition, déterministe / non-déterministe) est générique. Suivant le type de structure de choix, tout ou partie du code suivant est généré. Par exemple, une sélection déterministe sans sondes est traduite en VHDL en utilisant seulement les instructions (1) et (4).

```
(1) G:=(G1, … , Gn);

(2) while false(G) loop
(2)   wait on channel_signal_list;
(2)   G:=(G1, … , Gn);
(2) end loop;

(3) while not(false(G)) loop

(4)   case CHP_which(G) | case CHP_arbiter(G)
(4)   when 1 => S1;
(4)   …
(4)   when n => Sn;
(4)   { when others => Sothers; }
(4)   end case;

(3)   G:=(G1, … , Gn);
(3) end loop;
```

G est un vecteur booléen de taille n, Gi sont les n gardes et Si les n instructions. La ligne (1) est toujours générée, elle permet d'évaluer toutes les gardes. Lorsque des canaux sont sondés dans les gardes, il est nécessaire de suspendre le processus jusqu'à ce qu'une garde soit vraie. Cette synchronisation sur les canaux est implémentée par la boucle (2). Si les gardes ne portent que sur des variables, la boucle n'est pas générée.

Une sélection est modélisée par la structure case (4), tandis qu'une répétition est modélisée par la structure case englobé par la boucle (3). Pour les structures déterministes, la fonction CHP_which() est appelée. Elle compte le nombre de gardes vraies, vérifie la validité et l'exclusivité des gardes et finalement génère une erreur ou renvoie le numéro de la garde choisie. Réciproquement, pour les structures non-déterministes, on utilise la fonction

CHP_arbiter() qui compte le nombre de gardes vraies, effectue un choix si plusieurs gardes sont en conflit et renvoie le numéro de la garde choisie. L'instruction correspondante est alors exécutée.

On peut remarquer que la vérification d'exclusivité entre les gardes pourrait être faite au moment de la compilation pour les gardes portant sur des variables (ceci impliquerait du calcul formel sur les gardes). Par contre, cette vérification doit être faite en cours d'exécution pour les gardes portant sur des canaux. Par souci de simplification, toutes les vérifications sur les gardes sont exécutées de manière dynamique.

Les deux fonctions CHP_which() et CHP_arbiter() sont fournies dans le package CHPlib. Pour fournir plus de flexibilité, comme le choix non-déterministe n'est pas totalement spécifié par le langage CHP, la fonction CHP_arbiter() utilise une fonction aléatoire CHP_random(seed) définie par l'utilisateur. Le paramètre seed introduit dans le paragraphe 2.2.5 est passé à cette fonction. Lorsqu'il n'est pas spécifié, une valeur par défaut est utilisée. Ce paramètre permet d'initialiser le tirage aléatoire et donc de simuler des arbitres non-corrélés entre instances de processus.

Les options de trace sont disponibles pour les structures de contrôle. Le paramètre "traceon" introduit dans le paragraphe 2.2.5 est propagé par le traducteur aux fonctions CHP_which() et CHP_arbiter(). Selon sa valeur, les primitives CHP_trace_which() et CHP_trace_arbiter() sont appelées, procédures fournies dans le package CHPlib. Pour les choix déterministes, CHP_trace_which() sauve le nom du processus, un label pour la structure, la date, et la garde sélectionnée ; pour les choix non-déterministes CHP_trace_arbiter() sauve en supplément la liste des gardes en contention. Ces informations sont alors générées en cours de simulation dans un unique fichier qui peut-être trié et analysé par la suite. Ce *profiling* fournit typiquement les probabilités sur les différentes gardes. Cette analyse de l'exécution des programme CHP permet à la fois d'améliorer la spécification initiale et de guider le processus de synthèse, que ce soit dans un objectif d'optimisation en vitesse et/ou en consommation (voir chapitre 3).

2.3.7. Aide à la co-simulation

D'une manière générale, les méthodologies « top-down » basées sur les langages de description de matériel offrent des possibilités de co-simulation à tous les niveaux : comportemental, flot de donnée, structurel, cellules standard. Cette flexibilité dans la manipulation des différents niveaux d'abstraction est possible parce que les modèles sont tous basés sur une unique sémantique de synchronisation : l'horloge. Ainsi du point de vue de l'utilisateur, la modularité est supportée par le HDL grâce aux concepts de configuration et de bibliothèque de composants.

Dans la méthodologie telle que proposée dans le paragraphe 2.1.2, plusieurs protocoles de communications peuvent coexister (synchrone, bundle-data, quasi insensible aux délais). La modularité n'est plus gratuite : elle reste possible mais nécessite des interfaces entre les différents styles de synchronisation. Ces interfaces sont d'une complexité variable et peuvent être implémentés dans le HDL visé : conversion de protocole, encodage/décodage des données, vérification de contraintes temporelles, etc …

Pour notre projet de processeur Aspro, l'un de notre premier objectif est de supporter la co-simulation en VHDL de blocs fonctionnels traduits depuis le CHP et de blocs QDI niveau

porte. Le premier problème qui se pose est celui de l'initialisation. En matériel (voir chapitre 5), un bloc niveau porte nécessite dans le cas général un reset global pour initialiser tous ses nœuds internes. En CHP, le reset n'est pas nécessaire puisque les signaux VHDL des canaux sont implicitement initialisés à la date zéro. De manière à pouvoir contrôler l'état de reset à la fois des blocs CHP et des blocs niveau porte, le traducteur ajoute optionnellement tout ce qui est nécessaire au code VHDL pour pouvoir : i) propager le signal de reset à tous les processus et composants CHP, ii) initialiser les processus CHP quand le reset est actif. Ainsi, le signal reset est asynchrone, il peut être activé à tout moment pour interrompre la simulation et la redémarrer.

Ensuite, il n'y a pas de raison que les protocoles de communication soient identiques entre les blocs CHP et niveaux porte. Il faut donc effectuer la conversion de protocole (2 ou 4 phases, actif/passif), l'encodage/décodage des données (trois états "0,1,Z" contre trois états "double rail") et la synchronisation entre les bits pour les bus.

Ainsi, plusieurs composants d'interfaces ont été développés en VHDL et sont disponibles dans le package CHPlib. Ces interfaces sont sensibles au reset. Lorsque reset est activé, ils initialisent leurs sorties, puis vérifient à la fin du reset que leurs entrées sont correctement initialisées. Enfin, lors de l'écriture de ces interfaces, il faut faire attention à ne pas ajouter ou supprimer d'étages de mémorisation. Cela pourrait modifier le comportement du modèle et même introduire des inter blocages.

Une dernière facilité offerte par le traducteur est qu'il insère automatiquement ces interfaces quand nécessaire. Pour cela, l'outil génère une netlist VHDL qui correspond à l'instance du bloc niveau porte dans le bloc CHP. La Figure 2-8 donne l'exemple d'une unité arithmétique et logique synthétisée en porte logique qui est instanciée dans le modèle CHP du microprocesseur.

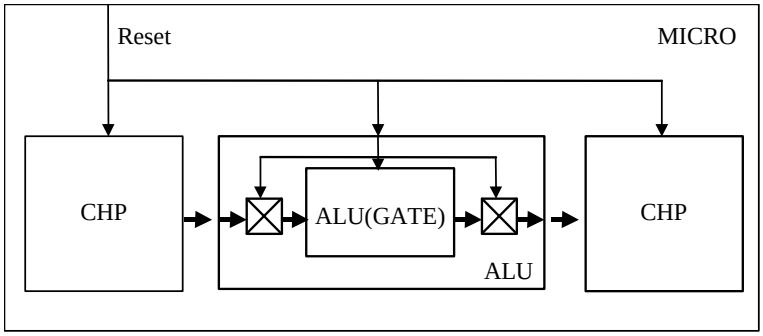


Figure 2-8 : Reset et interfaces sur les canaux pour co-simuler CHP et blocs niveau porte

Pour effectuer la co-simulation, il reste alors à écrire une nouvelle configuration VHDL. Celle-ci est simplement écrite manuellement de la manière suivante, on spécifie que le bloc ALU est en porte, tous les autres restant par défaut en CHP :

```
Configuration cfg_micro_alu_gate of micro is
  For the_alu : ALU use entity MICRO_GATE_LIB.ALU(gate)
  end for;
end cfg_micro_alu_gate;
```

Cette co-simulation VHDL permet de vérifier très rapidement que le bloc ALU a été synthétisé correctement : correction fonctionnelle, correction des protocoles et de la phase de reset, tout cela dans l'environnement de simulation d'origine, à savoir le modèle CHP du

microprocesseur. On voit ici tout l'intérêt de faire appel à un langage standard comme VHDL : l'utilisateur dispose d'une approche de conception descendante avec validation des blocs par étape, le tout avec un unique simulateur.

Enfin, l'interfaçage entre circuits synchrones et asynchrones peut-être effectué de la même manière. La gestion du reset offerte par l'outil, associé avec la possibilité pour l'utilisateur de définir des interfaces dans l'HDL visé offre donc un moyen général de co-simuler des blocs mettant en œuvre différentes formes de synchronisation et de communication.

2.3.8. Gestion de projet et exemple de simulation

En terme d'environnement de travail, le traducteur chp2vhdl a été développé et utilisé sous Unix en Solaris 5.3. Le compilateur et simulateur VHDL utilisés sont VSS© de Synopsys. L'environnement de travail minimum est très succinct puisqu'il suffit d'avoir compilé pour VSS le package CHPlib fournit avec le traducteur.

Comme précisé dans le paragraphe 2.3.1, le traducteur maintient à jour un fichier "makefile" associé au projet. Au début du projet, un "makefile" vide est créé par l'outil dans lequel il suffit de spécifier quelques variables d'environnement Unix. Le makefile sera mis à jour par le traducteur suivant les instances des différents composants CHP. Ainsi, une simple commande make permet d'effectuer la tâche fastidieuse qui correspond à l'enchaînement minimum de compilations chp et vhdl nécessaires suivant les dates de mise à jour des différents fichiers.

L'exemple de la Figure 2-9 présente un fichier merge.chp et bench_merge.chp qui contiennent respectivement la description d'un composant *MERGE* et son environnement de test *BENCH_MERGE*.

Pour effectuer la première compilation chp de ce modèle, il faut taper :

chp2vhdl merge.chp

Si le modèle est correct, un fichier MERGE.vhd est généré dans le sous-répertoire VHDL et le makefile est mis à jour. Pour effectuer la compilation vhdl de MERGE.vhd, il suffit alors de faire :

make

De la même manière, la compilation du fichier bench_merge.chp est effectuée par :

chp2vhdl bench_merge.chp

make

Par la suite, il suffit de taper make pour recompiler l'ensemble d’un ou des fichiers modifiés. Pour lancer la simulation, il faut appeler le simulateur VHDL avec la configuration *BENCH_MERGE_SIMU* créée à cet effet par le traducteur. Les traces de simulation Figure 2-9 obtenues avec VSS montrent l'arbitrage entre les deux flux non-exclusifs arrivant sur les ports E1 et E2.

```
--------------------
-- fichier merge.chp
--------------------

Component merge
   Port ( E1  : in bit[7..0];
          E2  : in bit[7..0];
          S   : out bit[7..0] )
Begin
Process
Variable x : bit[7..0];
*[    -- choix indéterministe
     [  #E1 => E1 ? x ; S ! x
     @@ #E2 => E2 ? x ; S ! x
     ]
]
End merge;
```

```
--------------------------
-- fichier bench_merge.chp
--------------------------

Component bench_merge
Channel A,B,C : bit[7..0];
Begin

m : merge Port Map (A,B,C)

Process stimuli_A Port ( A : out bit[7..0] )
Variable a : bit[7..0] := x"00";
*[  A!a ; a:=a+x"01"  ]

Process stimuli_B Port ( B : out bit[7..0] )
Variable b : bit[7..0] := x"00";
*[  B!b ; b:=b-x"01"  ]

Process read_C Port ( C : in bit[7..0] )
*[  C?  ]

End bench_merge;
```

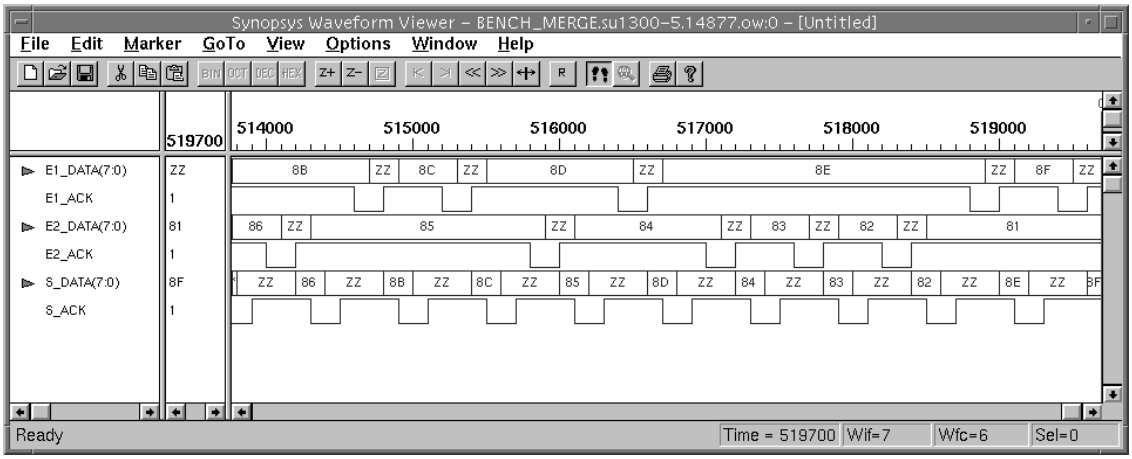


Figure 2-9 : Exemple de modèle CHP d'un multiplexeur, son environnement de test et les courbes de simulations obtenues avec VSS © de Synopsys.

2.4. Conclusion

Le langage CHP que nous venons de présenter est un langage de spécification de circuits asynchrones. De nombreux exemples de modélisations seront donnés dans le chapitre 5. Dans ce langage, on observe une bonne lisibilité fonctionnelle grâce aux canaux, aux gardes, aux opérateurs de concurrence et de séquentialité. On note aussi une bonne lisibilité structurelle grâce à la modélisation hiérarchique. Même si aujourd'hui l'opérateur de concurrence est limité en raison de restriction du VHDL, ceci permet de modéliser un certain niveau de concurrence au sein des processus. Il serait nécessaire de chercher des solutions afin d'étendre cet opérateur (analyse complète des dépendances de données du modèle et extraction de l'opérateurs de concurrence).

Tel que présenté aujourd'hui, le langage CHP est un langage informatiquement pauvre : pas de constantes, de fonctions, utilisation de types limités… Néanmoins, ceci a toujours été

suffisant pour nos besoins de modélisations. Il serait effectivement intéressant d'enrichir le langage pour une utilisation industrielle, sans toutefois tomber dans la verbosité de VHDL.

L'outil de traduction CHP_2VHDL que nous avons développé permet donc de valider par simulation des programmes CHP de tout niveau de granularité. Grâce à l'utilisation d'une plate-forme de simulation puissante offerte par les outils standard basés sur VHDL, il est possible d'effectuer tout type de co-simulation : CHP fonctionnel, *netlist* de portes, blocs synchrones (mémoires RAMs, blocks IPs), bibliothèques de fonctions…

Le traducteur lui-même est un environnement de simulation relativement générique. Avec l'utilisation de package VHDL, l'utilisateur a la possibilité de paramétrer sa simulation : choix de différents protocoles de communication, gestion du temps, génération de traces. Grâce à la structure logicielle de l'outil, il serait possible de rajouter aisément d'autres modules de traduction. On peut penser en particulier à la génération de Verilog. Ainsi, l'approche de traduction du langage CHP nous permet d'adapter l'outil en fonction de l'évolution des langages standard et des outils CAO associés. Par exemple, des travaux sont actuellement effectués sur le VHDL orienté objet [SWAM95] et comment effectuer des mécanismes de passage de message dans ce langage [PUTZ98]. Notre environnement de conception pourrait facilement être configurer pour profiter de ces nouveaux derniers développements.

Pour conclure, le traducteur CHP_2VHDL a bien sûr été développé dans le contexte de la réalisation du processeur ASPRO, mais comme nous l'avons présenté dans le paragraphe 2.1, il fait partie d'une méthode de conception plus large en vue de concevoir des systèmes mixtes synchrones / asynchrones. Grâce à l'utilisation d'un environnement de simulation unique basée sur un HDL, il est ainsi possible de mélanger différents style de synchronisation et de les analyser conjointement. On obtient ainsi une convergence des circuits synchrones et asynchrones.

Chapitre 3 :

Synthèse de circuit asynchrone quasi-insensible aux délais en cellules standard

Introduction

Ce chapitre présente la méthode d'implémentation du processeur Aspro. La méthode de synthèse est largement inspirée de la méthode de conception de circuit Quasi Insensibles aux délais proposée par A. Martin [MART 90], [MART 93]. Dans son cadre le plus général, cette méthode a l'avantage d'offrir à chacune des étapes de synthèse un vaste ensemble de choix. Cette approche laisse en conséquence de grands espaces de liberté au concepteur afin de pouvoir librement optimiser la logique en fonction des critères classiques de vitesse / surface / consommation. De manière systématique, A. Martin a toujours utilisé cette méthode afin de concevoir des circuits dédiés (« *full-custom* ») [MART 97].

Cependant pour nous, l'objectif de la méthode de conception de circuit asynchrone est de cibler des circuits utilisant des cellules standard afin de pouvoir concevoir des circuits rapidement et utiliser les flots de conception classiques tel que présenté dans le chapitre 2 : placement & routage, back-annotation, etc. Le travail a donc consisté à effectuer des choix de synthèse bas niveau afin de pouvoir cibler une bibliothèque de cellules. Cette dernière doit être contrainte afin de pouvoir utiliser à la fois les bibliothèques standard du fondeur et un nombre de cellules spécifiques, si nécessaire, le plus réduit possible.

Une rapide présentation de la méthode de synthèse de A. Martin permet de montrer dans le paragraphe 3.1 les choix de synthèse qui ont été effectués : « réordonnancement » symétrique et décomposition logique de type DIMS - *Delay Insensitive Min-term Synthesis* - [MULL 65], [DAVI 92], [SPAR 93]. Ces choix de synthèse ont permis aussi de définir un CHP dit « synthétisable ». Cette restriction du langage CHP est expliquée et justifiée paragraphe 3.2. De plus, on montrera par des exemples qu'il est toujours possible de se ramener à ce sous ensemble du langage par décomposition de processus.

La bibliothèque de cellule spécifique qui a été développée est constituée principalement de portes de Muller, elle est présentée paragraphe 3.3. Enfin, nous montrerons dans le paragraphe 3.4 l'ensemble du flot de conception qui a été mis au point et utilisé pour effectuer la conception du processeur Aspro : depuis le modèle CHP fonctionnel jusqu'aux simulations back-annotées après placement & routage.

3.1. Synthèse de circuit QDI en cellules standard

3.1.1. Présentation de la méthode de conception

Dans ce premier paragraphe, nous donnons un rapide résumé de la méthode générale de synthèse de circuits quasi-insensibles aux délais proposée par A. Martin dans [MART 90]. La méthode de conception est composée de plusieurs étapes (Figure 3-1) : écriture d'un modèle dans le langage CHP, décomposition de ce modèle, expansion des communications, réordonnancement, génération des règles de production, et enfin décomposition et optimisation logique des règles de production sur la cible technologique choisie.

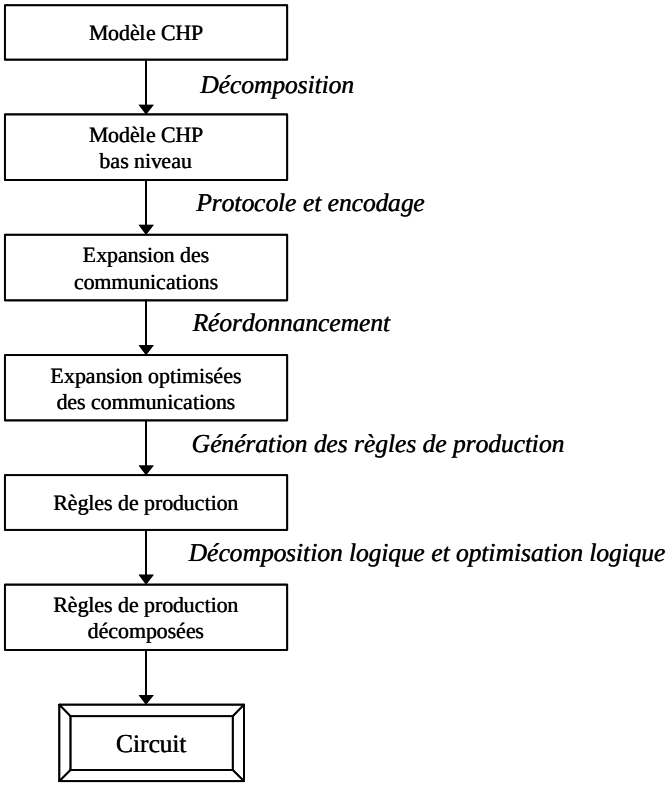


Figure 3-1 : Méthode de compilation de circuit [MART 90]

Pour montrer dans sa généralité les différentes phases de conception, nous prenons en exemple le modèle CHP du processus suivant :

```
P = *[ E?x ; S!x ]
```

Où x est une variable de type bit, E (resp. S) est un canal connecté en entrée (resp. en sortie) au processus P. Les canaux E et S sont du même type que la variable x. Ce processus est le plus simple que l'on puisse imaginer. Il n'a pas besoin d'être décomposé : il peut être directement synthétisé en logique.

3.1.1.1. Expansion des communications et codage des canaux.

La première phase de conception consiste à écrire les HSE (« Hand Shaking Expansions » ou expansions des communications), c'est à dire le protocole de communication utilisé pour implémenter chaque canal de communication. Le protocole de communication d'un canal se définit par *l'enchaînement des actions sur les éléments constituants ce canal*. Le canal CHP n'est plus alors seulement une notion abstraite de moyen de synchronisation entre processus mais prend à ce moment une forme matérielle. Cette transformation doit bien sûr garder la sémantique de synchronisation du canal d'origine : le protocole doit implémenter le mécanisme de rendez-vous. Le choix d'un protocole impose en conséquence de choisir à la fois le codage des canaux (codage des données, des signaux de requêtes et d'acquittements) et le protocole associé à ce codage.

Afin d'obtenir une logique insensible aux délais, il est nécessaire de choisir un codage adapté [VERH88]. Pour supprimer toute hypothèse de délai, la signalisation doit être encodée avec la donnée. Les données ne peuvent donc pas être séparées du signal de requête comme dans les protocoles de type données-groupées (« bundle-data protocol») [SUTH 89]. Le codage que nous utilisons est un codage double-rail trois-états tel que nous l'avons présenté dans le chapitre 1. Chaque bit d'un canal E est implémenté avec deux signaux : un signal E_0 pour encoder la valeur 0, un signal E_1 pour encoder la valeur 1 (Tableau 3-1).

Channel E : Bit	E_0	E_1
E=0	1	0
E=1	0	1
E=invalide	0	0

Tableau 3-1 : codage double rail - trois états

Lorsque E_0='1', la donnée du canal E vaut 0. Respectivement lorsque E_1='1', la donnée du canal E vaut 1. L'état '00' signale l'invalidité du canal. L'état '11' est interdit. Par extension du codage double rail, on peut facilement définir un codage *n*-rail dans lequel une valeur *n*-aire est encodée avec *n*-signaux (une transition à 1 sur le signal *i* encode la valeur *i*).

Au codage double rail, on associe facilement un protocole quatre phases afin que le canal repasse toujours dans l'état invalide entre l'échange de deux données successives. Si on note Ea le signal d'acquittement du canal E, on peut définir les deux protocoles suivants :

- émission de la valeur 0 : E_0 + ; [/Ea] ; E_0 - ; [Ea]
- émission de la valeur 1 : E_1 + ; [/Ea] ; E_1 - ; [Ea]

Par convention d'écriture, 'S+' (resp. 'S-') signifie la montée à l'état logique 1 (resp. 0) du signal S. '[S]' (resp. '[/S]') signifie l'attente du passage du signal S à 1 (resp. 0). Par souci de simplification, on notera par la suite Ei l'un des deux rails de données E_0 ou E_1.

Le protocole précédent Ei + ; [/Ea] ; Ei - ; [Ea] est qualifié d'*actif* car il commence par Ei+ qui signifie l'émission de la donnée. Réciproquement, le protocole [Ea] ; Ei + ; [/Ea] ; Ei - est qualifié de *passif* car il commence par [Ea] signifiant la synchronisation sur le signal d'acquittement.

Ces protocoles actif ou passif peuvent être utilisés de manière équivalente pour un canal connecté à un port d'entrée ou à un port de sortie. On peut ainsi utiliser les protocoles suivants : entrée passive, entrée active, sortie passive, sortie active. Cependant, afin de pouvoir aisément composer les processus entre eux et donc éviter tout inter-blocage, il est strictement nécessaire que les protocoles de communication soient compatibles entre le port de sortie et le port d'entrée connectant un même canal.

Pour notre exemple `P=*[E?x;S!x]`, on le réécrit tout d'abord de la manière suivante afin d'expliciter la valeur de la variable x :

```
P = *[  E?x ; [ x=0 => S!0
              @ x=1 => S!1
      ]       ]
```

Si on choisit comme protocole E passif et S actif, l'expansion des communications consiste alors à remplacer chaque action de communication par son protocole équivalent. On obtient les deux gardes suivantes, l'une pour x=0, l'autre pour x=1 :

```
P = *[  [ [E0]; x:=0; Ea-; [/E0]; Ea+
        @ [E1]; x:=1; Ea-; [/E1]; Ea+
        ] ;
        [ x=0 => S0+; [/Sa]; S0-; [Sa]
        @ x=1 => S1+; [/Sa]; S1-; [Sa]
      ] ]
```

Dans ces HSE, on peut remarquer que la variable x n'est pas strictement nécessaire. C'est une variable temporaire qui est directement transmise du canal d'entrée E au canal de sortie S. Les deux structures de gardes peuvent êtres associées entre elles car chaque garde correspond à la valeur 0 ou 1 de la variable x. On peut alors réécrire ce programme de la manière suivante :

```
P = *[  [  [E0]; Ea-; [/E0]; Ea+  ;  S0+; [/Sa]; S0-; [Sa]
        @  [E1]; Ea-; [/E1]; Ea+  ;  S1+; [/Sa]; S1-; [Sa]
      ]  ]
```

Cette écriture ne fait plus apparaître que les signaux composants les canaux. La variable du processus d'origine a disparu : elle est directement explicitée dans le protocole grâce au codage double rail. Ainsi, une fois effectué le choix du codage des données et du protocole, l'expansion des communications est une opération simple et régulière. Pour obtenir les HSE, il suffit de réécrire le programme CHP en remplaçant chaque action de communication ? ou ! par son protocole équivalent.

Comme nous le présentions dans le paragraphe 1.1, un autre codage double rail est possible. C'est un codage quatre états auquel on associe facilement un protocole deux phases. A première vue, ce protocole deux phases permet de supprimer la phase de retour à zéro et donc optimise les performances. En pratique, en raison du codage quatre-états, il donne des circuits plus complexes, plus lents, qui sont relativement difficiles à implémenter [MCAU

92]. Il ne peut être utilisé efficacement que dans des processus de type « Fifo ». Il sera en particulier utilisé pour implémenter les liens série du processeur Aspro (voir chapitre 6).

3.1.1.2. Optimisation des HSE : Réordonnancement

L'expansion des communications précédente n'est pas directement synthétisable car se pose un problème de conflit d'état. En effet, si on code les états de ce processus avec ses propres signaux, on observe que le processus a le même état après Ea+ et après [Sa]. Deux solutions sont possibles pour coder de manière unique tous les états : soit introduire une variable supplémentaire, soit effectuer un réordonnancement au sein du protocole (« *reshuffling* » en anglais). De manière générale, la première solution n'est pas la plus efficace car elle ajoute du matériel et peut donner une logique plus lente. Cette première solution peut néanmoins être nécessaire si le réordonnancement n'est pas suffisant pour lever l'ambiguïté sur tous les états. Quant à elle, l'étape de réordonnancement consiste à brasser les différents éléments du protocole entre eux, tout en gardant les dépendances de données entre les signaux (comme l'action [E0] ; S0+), jusqu'à obtenir un protocole dont tous les états puissent être uniquement codés avec ses propres signaux. C'est dans cette étape que le concepteur peut introduire un certain niveaux de parallélisme entre les signaux et donc optimiser les performances de la logique.

De nombreuses solutions de réordonnancement ont été proposées et étudiées. Toutes permettent une implémentation mais ne sont pas toutes aussi efficaces les unes que les autres en terme de vitesse et de complexité. Voici les solutions les plus couramment utilisées [LINE 95], elles sont classées dans l'ordre du protocole le plus séquentiel au plus concurrent.

Afin de simplifier l'exemple, on ne donne l'expansion des communications optimisée que pour un unique rail, sachant que son écriture est facilement extensible au double rail. On note Ei pour E0 ou E1 et So pour S0 ou S1. Voici les HSE avant optimisation et les quatre réordonnancements les plus couramment utilisés :

HSE avant réordonnancement :

```
P = *[  [Ei]; Ea-; [/Ei]; Ea+  ;  So+; [/Sa]; So-; [Sa] ]     (0)
```

dépendance de données entre Ei et So

Réordonnancement combinatoire :

```
P = *[  [Ei]; So+; [/Sa]; Ea-; [/Ei]; So-; [Sa] ; Ea+  ]      (1)
```

Réordonnancement WCHB [1]:

```
P = *[  [Ei^Sa]; So+; Ea-; [/Ei^/Sa]; So-; Ea+ ]              (2)
```

Réordonnancement PCHB :

```
P = *[  [Ei^Sa]; So+; Ea-; [/Sa]; So-; [/Ei]; Ea+ ]           (3)
```

Réordonnancement PCFB :

```
P = *[  [Ei^Sa]; So+; Ea-; [[/Sa]; So-] , [[/Ei]; Ea+]]       (4)
```

[1] L'opérateur ' ^' est l'opérateur « et » logique. Réciproquement, on notera par 'v' l'opérateur « ou » logique.

On peut tout d'abord observer que l'unique dépendance de donnée [Ei] => So+ du processus (0) est bien respectée dans tous les protocoles après réordonnancement. On peut aussi noter que toute phase de calcul est toujours séparée d'une phase de remise à l'état invalide des canaux E et S afin de respecter le codage double-rail. Le réordonnancement a donc bien consisté à entrelacer deux protocoles quatre phase entre eux.

Le réordonnancement (1) est la solution la plus séquentielle. La donnée Ei est transmise de l'entrée vers la sortie puis est acquitté une fois que la sortie est acquittée à son tour. Cette séquence est valable aussi bien pour la phase de calcul que pour la phase de remise à zéro. Pour cet exemple, son implémentation se réduit à deux fils : So =Ei et Ea=Sa. Par contre, c'est le protocole le plus lent à priori en temps de cycle puisque tout est séquentiel. A titre de remarque, le modèle CHP devrait en fait s'écrire *[[#E => S! ; E?]] afin d'être plus proche de son implémentation.

Dans les trois autres protocoles (2), (3), (4), le calcul est plus rapide. En particulier, le temps de cycle est diminué. En effet, si le canal de sortie est disponible ([Sa]), la donnée [Ei] est transmise par So+ puis est aussitôt acquittée par Ea- et ce sans attendre l'acquittement de la sortie [/Sa]. Ceci permet de commencer la remise à zéro du protocole sur E indépendamment du temps de calcul sur S. Pour ces trois protocoles, il y a rendez vous au sein même du processus entre la disponibilité de l'étage suivant ([Sa]) et l'arrivé d'une nouvelle donnée ([Ei]). C'est cette condition qui autorise le recouvrement des actions entre les canaux E et S : on obtient une structure pipelinée.

Le protocole **WCHB** (*Weak Condition Half Buffer*) synchronise la phase de remise à zéro des deux protocoles : si l'entrée est redescendue ([/Ei]) et que la sortie est acquittée ([/Sa]), la sortie est remise à zéro (So-) et l'acquittement est repositionné (Ea+) pour la donnée suivante. Ce protocole permet une implémentation très régulière car les conditions pour générer So+ et So- sont symétriques sur les signaux Ei et Sa.

Le protocole **PCHB** (*Precharge Half Buffer*) permet de remettre à zéro la sortie (So-) dès que celle-ci est acquittée ([/Sa]) sans attendre l'invalidité de l'entrée puis repositionne Ea pour la donnée suivante une fois que l'entrée est remise à zéro ([/Ei]). Ce protocole est plus performant que le protocole WCHB lorsque la sortie est plus rapide que l'entrée. En effet dans ce cas, la sortie sera remise à zéro sans attendre la remise à zéro de l'entrée.

Le protocole **PCFB** (*Precharge Full Buffer*) est le protocole le plus rapide car il permet de remettre à zéro les protocoles sur E et S en parallèle. En particulier, il permet de mémoriser une donnée complète dans chaque processus, d'où son nom de full-buffer au lieu de half-buffer. Il est cependant plus coûteux en matériel car il nécessite une variable supplémentaire pour mémoriser le fait que les deux phases de remise à zéro soient finies avant de recommencer le cycle suivant. Dans [LINE 95], ces deux protocoles sont qualifiés de *pré-charge* car la porte CMOS qui implémente le signal So ressemble à une cellule à pré-charge.

Afin de comparer plus précisément les performances de ces différents protocoles et conclure sur leur utilisation, il est avant tout nécessaire de voir comment ils sont implémentés en logique. C'est l'objet du paragraphe suivant.

3.1.1.3. Génération des règles de production.

Après l'étape d'écriture des HSE et leur réordonnancement, la génération des règles de production consiste à écrire les conditions booléennes minimales pour obtenir la mise à 0 et à 1 des signaux de sortie du processus en fonction de ses signaux d'entrées. Contrairement au

protocole qui se présente comme une exécution séquentielle (même s'il est spécifié avec un certain niveau de parallélisme), les règles de production sont des conditions booléennes qui sont toutes concurrentes entre elles. Cette transformation du protocole à un ensemble de conditions booléennes est possible et correcte car les HSE après réordonnancement ne doivent plus comporter de conflit d'état : les états le long du protocole sont tous définis et distincts. Dans le cas où le réordonnancement ne serait pas suffisant pour supprimer les conflits d'états, il serait nécessaire d'ajouter des variables supplémentaires dans les HSE afin de lever les ambiguïtés.

Reprenons l'exemple du processus P = *[E?x ; S!x] écrit avec le protocole WCHB.

```
P = *[ [ [E0^Sa]; S0+; Ea-; [/E0^/Sa]; S0-; Ea+
       @ [E1^Sa]; S1+; Ea-; [/E1^/Sa]; S1-; Ea+
     ] ]
```

Les règles de production du processus P sont les suivantes :

```
{  E0 ^  Sa => S0+
{ /E0 ^ /Sa => S0-

{  E1 ^  Sa => S1+
{ /E1 ^ /Sa => S1-

{  S0 v  S1 => Ea-
{ /S0 ^ /S1 => Ea+
```

Ces conditions booléennes peuvent ensuite facilement s'implémenter avec des réseaux de transistors. Le test à 1 d'un signal correspond à un transistor NMOS, réciproquement le test à 0 d'un signal correspond à un transistor PMOS. Le « et » logique '^' se traduit par une connexion des transistors en série, réciproquement le « ou » logique 'v' est implémenté par des transistors en parallèle. La bonne polarité du signal de sortie s'obtient en ajoutant un inverseur sur le signal de sortie lorsque nécessaire.

Pour notre exemple, les règles de production des deux rails de donnée S0 et de S1 peuvent s'implémenter avec une porte de Muller [MULL 65] (Figure 3-2). Le signal d'acquittement Ea est lui facilement implémenté avec une porte NOR. Finalement, le résultat de synthèse du processus P = *[E?x ; S!x] utilisant un protocole WCHB est donné Figure 3-3. Ce schéma est couramment appelé *half-buffer*.

A	B	Z
0	0	0
0	1	Z^{-1}
1	0	Z^{-1}
1	1	1

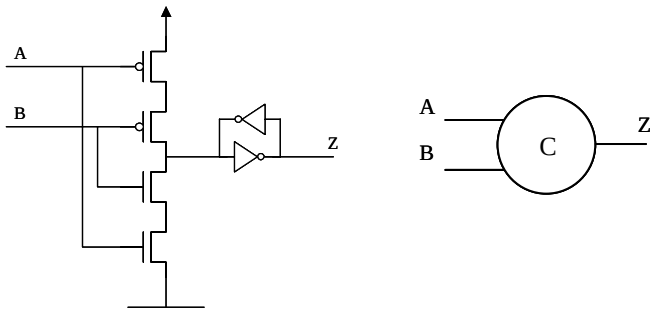


Figure 3-2 : Exemple de porte de Muller à deux entrées : table de vérité, schéma transistor et symbole.

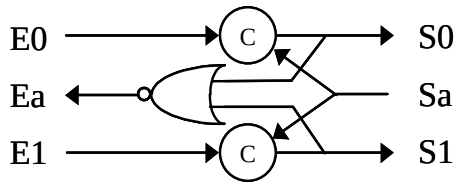


Figure 3-3 : "Half buffer" double-rail (protocole WCHB)

Reprenons à présent le même exemple P = *[E?x ; S!x] mais en utilisant le protocole PCHB. Voici tout d'abord l'expansion des communications après « réordonnancement » :

```
P = *[ [ [E0^Sa]; S0+; Ea-; [/Sa]; S0-; [/E0] ; Ea+
       @ [E1^Sa]; S1+; Ea-; [/Sa]; S1-; [/E1] ; Ea+
     ] ]
```

Les règles de production du processus P sont les suivantes :

```
{  E0 ^  Sa ^  Ea => S0+
{       /Sa ^ /Ea => S0-

{  E1 ^  Sa ^  Ea => S1+
{       /Sa ^ /Ea => S1-

{  S0 v  S1 => Ea-
{ /E0 ^ /E1 => Ea+
```

Ces équations donnent à priori des portes complexes non symétriques. Cependant, il est parfois possible de rajouter certains termes afin de les symétriser. Les règles de production du signal Ea peuvent être réécrites (renforcées) ainsi tout en respectant le protocole PCHB :

```
{ ( S0 v  S1) ^ ( E0 v  E1) => Ea-
{ (/S0 ^ /S1) ^ (/E0 ^ /E1) => Ea+
```

Les réseaux de transistors pour les rails de sortie S0 et S1 sont donc des portes complexes non symétriques. Quant à lui, le signal d'acquittement Ea peut être construit avec deux portes NOR et une porte de Muller (Figure 3-4).

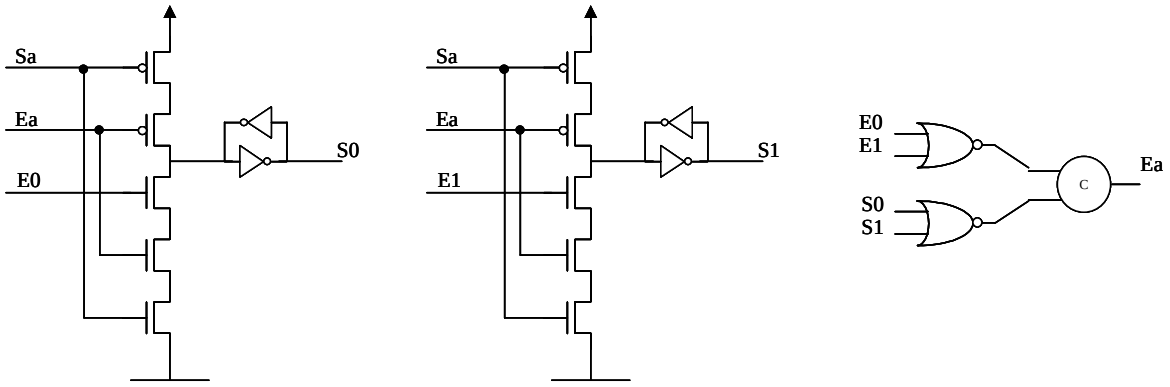


Figure 3-4 : « Half-buffer » double-rail (protocole PCHB)

L'implémentation de ce même processus en utilisant un protocole PCFB est similaire au schéma précédent mais avec une logique d'acquittement différente pour le signal Ea. On obtient de la même manière des portes complexes pour les rails de sortie S0 et S1.

3.1.1.4. Optimisation logique

Après la génération des règles de production, la dernière étape de conception consiste ensuite à optimiser la logique. Les règles de production telles que présentées dans les exemples ci-dessus implémentent directement le circuit. Cependant pour des exemples plus complexes, il est souvent nécessaire d'effectuer tout ou partie des transformations suivantes :

- symétriser les règles de production lorsque cela est possible comme on l'a vu précédemment afin d'obtenir une logique statique. Lorsque cela n'est pas possible, la porte dynamique doit être rendu semi-statique en ajoutant un point mémoire avec deux inverseurs re-bouclés. C'est le cas du schéma transistor de la porte de Muller Figure 3-2 et des portes complexes qui implémentent le protocole PCHB.
- factoriser les termes communs entre certaines équations pour ne pas dupliquer le matériel.
- ajouter des signaux intermédiaires afin de décomposer les règles de production qui ne peuvent pas être implémentées en un unique réseau de transistors en série. On se limite en général à 3 transistors PMOS en série et 4/5 transistors NMOS en série. Ces contraintes dépendent bien sûr de la technologie cible considérée et des performances attendues des cellules de base.
- ajouter des inverseurs sur les signaux d'entrées lorsque nécessaire. En effet, si une règle de production fait intervenir à la fois des signaux positifs et négatifs, il est nécessaire d'inverser les signaux d'entrés concernés afin de pouvoir construire un unique réseau de NMOS ou PMOS. Les exemples précédents ne sont pas concernés car toutes les entrées ont la même polarité pour chaque équation booléenne. Ce problème d'inversion des signaux fait partie du problème global de l'optimisation et de la décomposition logique. Il est qualifié de « bubble-reshuffling » dans [MART 90].

Toutes ces optimisations sont délicates car elles doivent respecter le modèle QDI. De nouveaux signaux peuvent être créés pour différentes raisons : inversion d'une entrée, factorisation d'un terme commun, décomposition d'une règle de production trop grosse. Ces signaux doivent apparaître dans les règles de production déjà existantes sans bien sûr modifier le bon déroulement du protocole, mais surtout ils ne doivent être activés que lorsqu'ils sont explicitement utilisés. Toute transition non-acquittée reviendrait à ajouter une porte dans une fourche isochrone [BERK 92] : l'insensibilité aux délais ne serait plus vérifiée.

3.1.1.5. Conclusion

Comme on a pu le montrer dans les paragraphes précédents, de nombreux choix de conception dans la méthode de A. Martin sont offerts au concepteur afin de synthétiser son modèle CHP en un circuit QDI : choix de codage, de protocole, de réordonnancement, et enfin problème de l'optimisation logique. L'exemple qui a été traité précédemment est bien sûr simpliste mais il permet de montrer les différentes étapes de conception et les difficultés rencontrées. La partie la plus délicate de la conception réside principalement dans le choix de réordonnancement et le problème général de l'optimisation logique.

Afin d'obtenir les meilleures performances pour le processeur Mini-MIPS [MART 97], A. Martin a utilisé toute les ressources de sa méthode. Il n'hésite pas à changer de réordonnancement et à optimiser la fonction logique de chaque arbre CMOS, en fonction du bloc à implémenter et de son environnement. Ceci représente un savoir-faire et un temps de

conception considérable. Au final, le design du Mini-MIPS est totalement réalisé en circuit dédié avec des portes complexes.

Réciproquement, notre objectif principal est de définir une méthode de conception de circuits asynchrones qui vise des circuits à cellules standard, et non des circuits dédiés. Le travail de thèse a donc consisté à définir comment utiliser la méthode générale de A. Martin afin de respecter cette contrainte. En conséquence, certains choix systématiques ont été effectués afin de :

- obtenir un circuit implémentable avec des bibliothèques de cellules standard : les bibliothèques standard des fondeurs et un ensemble de cellules spécifiques le plus réduit possible afin de limiter les efforts de conception de bibliothèque.
- faciliter la conception et obtenir des règles de synthèse les plus régulières possibles afin de pouvoir envisager à terme le développement d'un outils de synthèse spécifique.

Ces différents choix sont présentés dans le paragraphe suivant.

3.1.2. Choix et adoption d'un style de synthèse

Si on compare les réordonnancement WCHB, PCHB, et PCFB du paragraphe 3.1.1, les protocoles PCHB et PCFB sont les plus rapides car ils permettent d'effectuer la remise à zéro des canaux de sortie indépendamment de la remise à zéro des entrées du bloc. Au contraire, le protocole WCHB synchronise à la fois les canaux d'entrée et de sortie pour la phase montante et la phase descendante du protocole, il est donc à priori plus lent. Il faut toutefois faire attention à ne pas tirer de conclusion hâtives. Une fois synthétisés, les performances en vitesse des différents protocoles peuvent variés fortement en fonction des processus voisins et du nombre d'entrées-sorties du processus (et donc de la décomposition logique qui en résulte).

Cependant, grâce à cette propriété, le protocole WCHB permet d'obtenir des règles de production qui sont symétriques. Ces règles peuvent ensuite se décomposer et s'implémenter avec uniquement des portes de Muller et des portes logiques classiques. C'est ce protocole qui a été choisi de manière systématique pour implémenter le processeur Aspro.

3.1.2.1. Protocole WCHB

Sur un exemple, nous montrons différentes possibilités d'optimisations et comment décomposer les règles de production avec uniquement des portes de Muller et des portes « ou ». Voici en exemple le modèle CHP du « et » logique de deux entrées A et B double-rail:

```
*[ A?a , B?b ; S!(a and b) ]
```

Ce modèle est réécrit de la manière suivante afin d'expliciter la table de vérité de la fonction « et » :

```
*[ A?a , B?b ; [ a=0 and b=0 => S!0
               @ a=0 and b=1 => S!0
               @ a=1 and b=0 => S!0
               @ a=1 and b=1 => S!1
 ]             ]
```

En utilisant le codage double rail, si on utilise le protocole WCHB, on peut écrire l'expansion des communications suivantes :

```
*[ [A0^B0^Sa]; S0+;Aa-,Ba-; [/A0^/B0^/Sa]; S0-;Aa+,Ba+]
 @ [A0^B1^Sa]; S0+;Aa-,Ba-; [/A0^/B1^/Sa]; S0-;Aa+,Ba+]
 @ [A1^B0^Sa]; S0+;Aa-,Ba-; [/A1^/B0^/Sa]; S0-;Aa+,Ba+]
```

```
@ [A1^B1^Sa]; S1+;Aa-,Ba-; [/A1^/B1^/Sa]; S1-;Aa+,Ba+]
] ]
```

Ce qui donne les règles de production suivantes :

```
(A0^B0^Sa) v (A0^B1^Sa) v (A1^B0^Sa)          => S0+
(/A0^/B0^/Sa) ^ (/A0^/B1^/Sa) ^ (/A1^/B0^/Sa) => S0-

(A1^B1^Sa)    => S1+
(/A1^/B1^/Sa) => S1-

(S0 v S1)     => Aa- , Ba-
(/S0 ^ /S1)   => Aa+ , Ba+
```

Les signaux d'acquittements Aa et Ba sont identiques. Ils sont obtenus par la porte NOR des rails de sortie S0 et S1. Ceci est donc comparable au schéma d'acquittement du *half-buffer* WCHB présenté paragraphe 3.1.1. Le rail S1 est obtenu par une porte de Muller à trois entrées : A1, B1 et Sa.

Cependant, il reste à résoudre un problème d'optimisation logique pour le signal S0. La règle de production optimale serait par exemple :

```
((A0^B0) v (A0^B1) v (A1^B0)) ^ Sa  => S0+
(/A0^/B0^/A1^/B1) ^ /Sa             => S0-
```

Cette règle de production correspond à un unique étage de transistor : un réseau NMOS qui calcule les min-termes et des transistors PMOS en série pour remettre à zéro la sortie S0. C'est une porte complexe non symétrique (Figure 3-5).

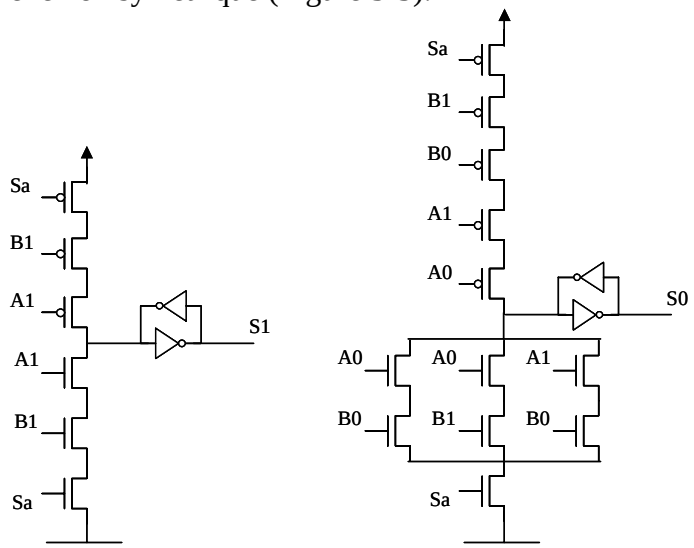


Figure 3-5 : « et » double-rail optimisé, protocole WCHB.

Réciproquement, au lieu d'optimiser l'équation logique en une unique équation, on peut créer trois signaux intermédiaires qui correspondent chacun à un min-terme. On obtient alors les règles de production suivantes :

```
A0^B0^Sa    => X0+          A0^B1^Sa    => X1+
/A0^/B0^/Sa => X0-          /A0^/B1^/Sa => X1-

A1^B0^Sa    => X2+          X0vX1vX2    => S0+
/A1^/B0^/Sa => X2-          /X0^/X1^/X2 => S0-
```

Ces règles de production correspondent à trois portes de Muller à trois entrées et à une porte « ou ». Le schéma équivalent du « et » logique double-rail décomposé en porte de Muller est donné Figure 3-6.

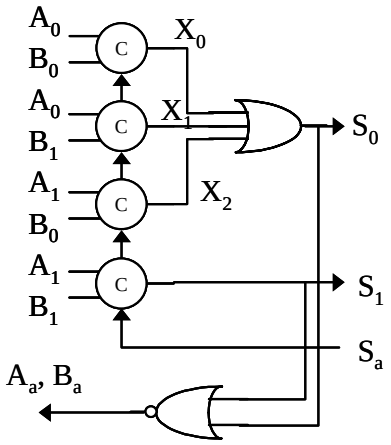


Figure 3-6 : « et » double-rail WCHB, décomposé en portes de Muller et OU.

A titre de comparaison, en utilisant le protocole PCHB, comme celui-ci oblige la remise à zéro des sorties avant celle des entrées, on obtiendrait des règles de production non symétriques comme pour le half-buffer présenté figure 3-4. Il est à noter qu'un tel circuit serait en général préférable à celui de la Figure 3-5 car évite les 5 transistors PMOS en série. Le protocole PCHB donne une logique proche de la logique DCVSL [CHU 87], [RENA 96], [ELHA 95] : la cellule DCVSL intègre à la fois la fonction avec un arbre complexe de transistors NMOS et le protocole avec les signaux de requêtes et d'acquittements. Comme pour le protocole WCHB « optimisé », on obtiendrait une cellule complexe non symétrique.

3.1.2.2. Protocole séquentiel

Pour les blocs dont les performances sont moins critiques, il peut être souhaitable d'utiliser le protocole séquentiel afin de diminuer la consommation ou la surface par exemple. Voici comment implémenter l'exemple précédent avec le protocole séquentiel. L'expansion des communications s'écrit alors :

```
*[ [A0^B0]; S0+; [/Sa]; Aa-,Ba-; [/A0^/B0]; S0-; [Sa]; Aa+,Ba+]
 @ [A0^B1]; S0+; [/Sa]; Aa-,Ba-; [/A0^/B1]; S0-; [Sa]; Aa+,Ba+]
 @ [A1^B0]; S0+; [/Sa]; Aa-,Ba-; [/A1^/B0]; S0-; [Sa]; Aa+,Ba+]
 @ [A1^B1]; S1+; [/Sa]; Aa-,Ba-; [/A1^/B1]; S1-; [Sa]; Aa+,Ba+]
 ] ]
```

Les règles de production sont les suivantes :

```
(A0^B0) v (A0^B1) v (A1^B0)       => S0+
(/A0^/B0) ^ (/A0^/B1) ^ (/A1^/B0) => S0-

(A1^B1)    => S1+
(/A1^/B1)  => S1-

/Sa   => Aa- , Ba-
 Sa   => Aa+ , Ba+
```

Les signaux d'acquittement Aa et Ba sont égaux au signal d'acquittement Sa. Le signal S1 est une porte de Muller à deux entrées. Pour le signal S0, on peut obtenir deux types de résultat : soit une porte complexe (identique à celle de la Figure 3-5 mais sans les transistors correspondant au signal Sa), soit la décomposition en trois portes de Muller à deux entrées et une porte « ou ».

Comme pour le protocole WCHB présenté ci-dessus, c'est ce type de décomposition qui sera choisi pour Aspro de manière systématique. Son schéma est donné Figure 3-7.

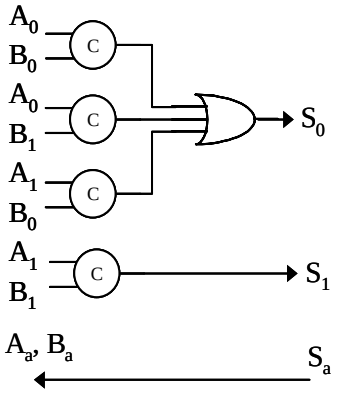


Figure 3-7 : « et » double-rail, version combinatoire.

3.1.2.3. Politique d'initialisation

Dans la logique QDI, il est nécessaire d'initialiser les protocoles de communications afin que tout processus puisse démarrer dans son état initial. Cet état initial doit correspondre au premier état des HSE après réordonnancement.

Par définition, la porte de Muller est une porte séquentielle, elle doit être correctement initialisée. Celle-ci peut s'initialiser toute seule si ses entrées sont initialisées à la même valeur, dans ce cas elle propage les valeurs d'initialisations. Par contre, elle doit être explicitement initialisée par un signal de reset dans le cas où ses entrées sont initialisées à des valeurs distinctes.

Vu l'expansion des communications choisie, les rails de données doivent être initialisés à 0, réciproquement les signaux d'acquittements doivent être initialisés à 1. Ainsi, pour le protocole combinatoire, la logique s'initialise toute seule en propageant le reset. Cependant, pour le protocole WCHB, la logique doit être initialisée. Ainsi, chaque porte de Muller qui implémente un rendez-vous entre un signal de donnée et un signal d'acquittement devra être explicitement initialisée. La valeur d'initialisation dépend du type du signal généré, soit 0 pour un rail de donnée, soit 1 pour un signal d'acquittement. Pour les deux exemples précédents, les portes de Muller de la Figure 3-6 doivent être initialisée à 0, réciproquement les portes de Muller de la Figure 3-7 seront initialisée par leurs propres entrées.

Cette politique de reset est simple et suffisante. Si elle est appliquée à tous les blocs de la hiérarchie, elle permet d'initialiser correctement l'ensemble de la logique au niveau électrique. Le problème spécifique de l'initialisation fonctionnelle de l'architecture sera présenté chapitre 5 et 6.

3.1.2.4. Conclusion

Comme on a pu le présenter avec l'exemple précédent du « et » logique double-rail, le protocole WCHB, associé à une décomposition des règles de production en un ensemble de

termes qui correspondent à la table de vérité de la fonction à implémenter, permet d'obtenir une logique qui peut s'implémenter avec uniquement des portes de Muller et des portes logiques classiques. Cette logique est donc similaire à la logique DIMS (*Delay Insensitive Min-Term Synthesis*) [DAVI 92], [SPAR 93].. Ce type de logique est cependant plus général car la méthode de A. Martin permet d'intégrer complètement le protocole de communication au sein de la logique : les signaux de données et d'acquittements ne sont pas nécessairement séparés. D'autres exemples de synthèse seront présentés chapitre 6 afin de justifier ces propos.

Par extension dans la suite du document, on qualifiera la version utilisant le protocole WCHB de « pipelinée » (Figure 3-6), la version séquentielle sera qualifiée de « combinatoire » (Figure 3-7). On peut noter que ces deux schémas sont très proches. Il est effectivement souvent facile d'intégrer la logique d'acquittement dans la logique de calcul. Ceci permet d'obtenir une version pipelinée d'un processus pour un coût en surface relativement faible en comparaison de la version combinatoire, tout en offrant de meilleures performances. Ce style de synthèse permet ainsi de dériver facilement une version pipelinée ou combinatoire d'un même schéma suivant le niveau de performance voulu. Ceci aide le concepteur à optimiser globalement la performance du design (voir chapitre 6).

Que ce soit pour la version WCHB ou combinatoire, cette logique peut donc être directement implémentée avec une approche cellule-standard : portes de Muller et portes de type « et », « ou ». La politique de reset est régulière et ne pose pas de problème particulier. La bibliothèque de cellules spécifiques sera présentée dans le paragraphe 3.3. Bien sûr, ce type de décomposition logique ne donne pas le même niveau de performance en vitesse et densité que des portes complexes directement issues des règles de production optimisées [LINE 95]. C'est le prix à payer afin de réduire le temps (et le coût) de développement des bibliothèques. Comme pour toute bibliothèque « synchrone », il y a nécessairement un compromis coût / performances.

Enfin, l'exemple précédent de la fonction « et » logique était bien sûr très simple. D'autres exemples (chapitre 6), même complexes, peuvent être aisément traités de la même manière. Cependant, tout modèle CHP ne peut pas être directement synthétisé et décomposé directement en portes de Muller tel que présenté ci-dessus. Certaines propriétés du modèle CHP doivent être respectées : elles sont présentés dans le paragraphe 3.2 suivant. En particulier, ces règles nous ont permis de définir un CHP dit « synthétisable ».

3.2. Définition d'un CHP synthétisable

Tout comme est défini le VHDL RTL par rapport aux circuits synchrones, nous avons défini un CHP synthétisable pour la conception de circuits asynchrones quasi-insensibles aux délais. Ce CHP synthétisable est une restriction de l'ensemble du langage CHP présenté chapitre 2. Il est directement synthétisable en cellules standard par la méthode de A. Martin grâce aux choix présentés dans le paragraphe précédent.

Les restrictions sur le langage sont de deux types différents : elles portent sur l'utilisation des variables et sur l'utilisation de l'opérateur séquentiel. Ces conditions sont définies et expliquées dans le paragraphe ci-dessous. De plus, si un processus CHP ne remplit pas ces conditions, celui-ci peut être transformé par décomposition de processus en un ensemble de processus synthétisables. Différents types de transformation sont possibles. Elles sont présentées dans le paragraphe 3.2.2.

3.2.1. Définition du CHP synthétisable

- **Utilisation des variables**

En CHP, les variables sont par définition locales aux processus. Cependant, celles-ci peuvent être de différentes natures : soit des variables dites "de mémorisation", soit des variables dites "intermédiaires".

- ✓ *Variables de mémorisation*. Ces variables correspondent à des éléments mémoires du processus. Leurs valeurs dépendent de l'état précédent du processus. Elles sont les véritables mémoires de l'architecture, typiquement ce sont des variables d'états, des registres, des mémoires.
- ✓ *Variables intermédiaires*. Au cours de l'exécution du processus, ces variables sont lues sur des canaux d'entrées, peuvent faire l'objet de calculs intermédiaires (commandes gardées, arithmétique, fonction complexe, etc.) puis émises sur des ports de sortie.

Seuls les processus contenant uniquement des variables intermédiaires sont synthétisables. Ces processus ont ainsi un fonctionnement de type production / consommation : toute valeur émise sur un canal de sortie est toujours uniquement fonction de valeurs lues sur les canaux d'entrées. Ces processus sont qualifiés de processus *combinatoires*. Le processus `*[E?x;S!x]` en est l'exemple le plus simple.

- **Utilisation de l'opérateur séquentiel**

La deuxième restriction concerne l'utilisation de l'opérateur séquentiel "**;**". La règle est la suivante :

Dans chaque branche d'un programme CHP, chaque action de communication (lecture ou écriture) ne doit pas accéder en séquence à un même canal. Cette propriété permet de garder le schéma production / consommation des processus.

En effet, prenons l'exemple `*[ E?x; S!x; S!x ]`. La variable x est obtenue par lecture sur le canal E puis est consommée deux fois par écriture sur le canal S. Ce n'est plus uniquement une variable intermédiaire. Etant consommée par le premier `S!x`, la variable x doit être mémorisée pour générer le deuxième `S!x`. Cette variable n'est pas directement synthétisable par un simple réordonnancement mais nécessite soit une décomposition sur le modèle CHP, soit l'introduction directe de cette variable dans les HSE.

Afin d'être synthétisable, l'opérateur séquentiel ne doit pas correspondre à l'enchaînement d'action sur des ressources uniques. Il doit simplement être l'image d'une dépendance de données entre des canaux d'entrées et de sorties distincts.

- **Justification**

Afin d'essayer de justifier rapidement ces contraintes sur le CHP synthétisable, reprenons la méthode de synthèse présentée paragraphe 3.1.1. Tout d'abord, le fait de choisir uniquement des variables intermédiaires permet de les cacher dans l'étape d'expansion des communications. Une variable intermédiaire peut toujours être directement explicitée dans les HSE grâce au codage double rail et au protocole quatre phase, et ce conjointement avec la fonction logique du processus. Les HSE ne dépendent donc pas d'un état précédent : c'est une boucle infinie qui s'écrit uniquement avec les signaux composants les canaux pris dans l'état courant. Ensuite, la restriction sur l'opérateur séquentiel « ; » permet d'écrire le

réordonnancement choisi de manière systématique entre les canaux d'entrées et de sorties. Il ne peut pas exister de conflit d'état entre deux protocoles successifs sur un même canal. Ainsi, le mécanisme de consommation / production des processus combinatoires non-séquentiels permet d'écrire systématiquement un réordonnancement. Cette propriété est générale, elle est vérifiée que ce soit pour un protocole combinatoire ou pipelinée WCHB.

- **Conclusion**

Seuls les *processus combinatoires* sont directement synthétisables en cellules standard avec la méthode et les choix de synthèse présentés paragraphe 3.2. Ces processus ont ainsi la forme suivante :

```
[ {initialisation} ;
  *[ lecture canaux d'entrée ; calcul ; écriture canaux de sortie ]
]
```

La partie initialisation est optionnelle, elle permet de produire une valeur initiale et ainsi initialiser le séquencement. (cf. exemples paragraphe 3.2.2). Ensuite, dans une boucle infinie, le processus lit des canaux d'entrées, effectue certains traitements (gardes, arithmétiques, fonctions quelconques) puis émet des valeurs sur des canaux de sortie, sachant que l'opérateur « ; » est restreint à la séparation d'actions sur des ressources distinctes.

Tous ces processus peuvent être indifféremment synthétisés sous une forme combinatoire ou pipelinée comme présenté dans l'exemple du « et » logique. Dans le cas d'une synthèse pipelinée, les variables de mémorisation introduites par le pipeline sont alors équivalentes aux variables intermédiaires. Réciproquement pour une synthèse combinatoire, les variables intermédiaires disparaissent totalement.

A titre de remarque, il est à noter que ces critères d'implémentation sur le CHP ne concernent pas uniquement notre choix de synthèse (protocole WCHB, décomposition logique et mapping sur des portes de Muller) mais aussi tout style de réordonnancement et de décomposition logique (porte complexe ou non). En effet, ces contraintes de synthèse portent directement sur le modèle CHP, ainsi elles interviennent relativement tôt dans la méthode de synthèse.

3.2.2. Exemples de décompositions CHP pour obtenir un CHP synthétisable

Si un processus n'est pas synthétisable tel que défini dans le paragraphe précédent, c'est à dire séquentiel ou comportant des variables non-intermédiaires, il est possible d'obtenir par décomposition de processus un ensemble de processus CHP synthétisables qui est équivalent au modèle d'origine. Ce paragraphe n'a pas la prétention de présenter une étude complète sur la décomposition de modèle CHP en modèle dit « synthétisable ». Plusieurs exemples et solutions de décomposition sont simplement donnés dans les paragraphes suivants. Ces différentes décompositions permettent de réaliser l'extraction de variables ou de transformer en machine à état des éléments séquentiels.

3.2.2.1. Extraction de variables

Afin d'extraire une variable de mémorisation d'un processus CHP, nous présentons deux types de transformation : soit l'extraction de la variable sous la forme d'un registre, soit comme une variable tournante.

- **Extraction d'une variable dans un registre.**

Supposons un processus P0 contenant un élément mémorisant x et un certain nombre d'entrées sorties. On crée un processus registre Reg-$_X$ associé à cette variable. Celui-ci peut s'écrire de la manière suivante :

```
Regx : *[  RWx?rw ;  [ rw=0 => Wx?x   -- écriture de x
                     @ rw=1 => Rx!x   -- lecture de x
         ]           ]
```

Ce registre Reg-$_X$ n'est bien sûr pas synthétisable tel que défini dans le paragraphe précédent puisqu'il contient la variable x qui est un élément mémoire. Cependant, une synthèse en cellules standard est possible même si elle n'est pas directe (cf. chapitre 6). Pour la suite, on pourra considérer que ce registre forme un élément de base de la bibliothèque de cellules.

Le processus d'origine contenant la variable x est alors transformé de la manière suivante :
- tout accès en écriture à la variable x est remplacé par la séquence `[ RWx!0 , Wx!x ]`
- tout accès en lecture à la variable x est remplacé par la séquence `[ RWx!1 , Rx?x ]`

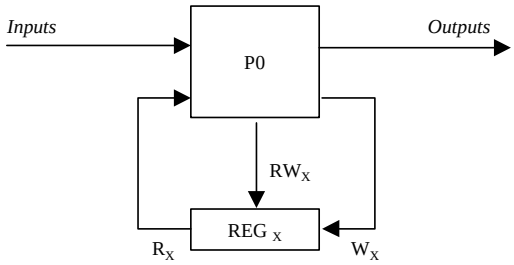


Figure 3-8 : Extraction d'une variable sous forme de registre.

La Figure 3-8 présente la vue équivalente du processus P0 après extraction de la variable. La variable est dorénavant stockée dans le registre et les canaux de sorties de P0 ne sont plus que fonctions des canaux d'entrées et du canal Rx. Le processus P0 est dorénavant combinatoire, il est synthétisable. La décomposition du banc de registre d'Aspro est un exemple d'extraction de variable en tant que registre (voir chapitre 5).

Enfin, il est parfois possible d'optimiser cette décomposition si les dépendances de données du processus P0 sont triviales. Ainsi, le canal Rx peut s'avérer être égal à l'un des canaux de sorties de P0, réciproquement Wx peut-être égal à l'un des canaux d'entrée de P0.

- **Extraction d'une variable comme variable « tournante »**

Supposons un processus P0 contenant un élément mémorisant x et un certain nombre d'entrées sorties. On crée un processus Loopx associé à la variable x qui s'écrit (La notation C signifie *current value* et N *next value*) :

```
Loopx : [ Cx!0 ;                  -- initialisation de la boucle
          *[ Nx?x ; Cx!x ]        -- propagation de la valeur
        ]
```

La synthèse en cellules standard du processus Loopx est donnée chapitre 6. Ce processus est équivalent à un *half-buffer* mais avec une différence sur sa politique de reset.

On effectue la transformation suivante sur le processus P0 :
- tout accès en écriture à la variable x est remplacé par la séquence [C_X? , N_X!x]
- tout accès en lecture à la variable x est remplacé par la séquence [C_X?x ; N_X!x]

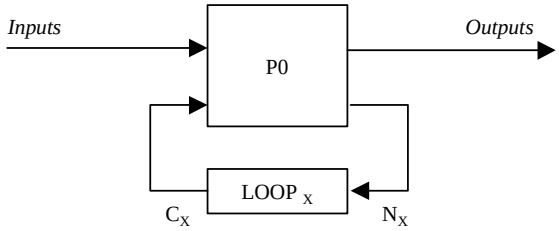


Figure 3-9 : Extraction d'une variable sous forme de variable tournante

La Figure 3-9 donne la vue équivalente du processus P0 après extraction de la variable. Contrairement à l'exemple précédent où la variable était mémorisée dans un registre fonctionnel, la variable est à présent mise à jour par P0 puis mise en attente dans la boucle de retour jusqu'à sa future utilisation. L'effet mémoire ne provient donc plus d'une véritable ressource mémoire mais d'un simple processus qui temporise et retransmet la donnée.

En raison de cette mémoire simpliste, il est nécessaire en écriture d'acquitter la valeur précédente (C_X?) pour pouvoir générer la nouvelle valeur. Réciproquement en lecture, comme la variable est consommée (C_X?x), il faut réécrire sa valeur pour qu'elle soit disponible à l'accès suivant. Dans ce cas, une optimisation en consommation consiste à uniquement consulter le contenu de la variable afin d'éviter de réécrire sa valeur. La variable est ainsi seulement mise à jour en écriture. Voici à titre d'exercice la décomposition du registre Reg-x précédent en utilisant une variable tournante et un mécanisme de consultation :

```
Reg'X : *[ RWX?rw ; [ rw=0 => CX? , [WX?x;NX!x]  -- écriture de x
                    @ rw=1 => [ #CX=x => RX!x ]  -- lecture de x
         ]          ]

LoopX : [ CX!0 ;              -- valeur d'initialisation
          *[ NX?x ; CX!x ]    -- propagation de la valeur
         ]
```

Ces deux processus sont équivalents au registre Reg-x d'origine, ils sont bien tous deux synthétisables. Il est à noter que cette décomposition donne une solution moins intéressante en surface que le schéma de Reg-x présenté chapitre 6.

• Conclusion

Une décomposition en variable « tournante » est particulièrement efficace lorsque le processus d'origine doit effectuer une lecture et une écriture de la variable dans le même cycle. Dans ce cas, l'accès à la variable x dans P0 est remplacé par [C_X?x ; N_X!x] dont les deux termes sont alors utiles. On obtient un fonctionnement de type machine à état : calcul des sorties de P0 en fonction des entrées et de l'état courant et calcul de l'état suivant. Ce modèle de fonctionnement ne serait pas possible avec une décomposition en registre car dans le même cycle il faudrait effectuer deux accès au registre : lecture de la valeur précédente et écriture de la valeur suivante. Des exemples de décomposition propres aux machines à états sont présentés dans le paragraphe 3.2.2.2 suivant.

Les deux méthodes précédentes d'extraction de variables sont des méthodes de décomposition générales sur le CHP. Elles offrent des compromis intéressants et ne s'utilisent

pas dans les mêmes conditions. Une décomposition en registre est intéressante pour un fonctionnement de type registre lorsque les accès à la variable sont distincts en lecture et écriture. Réciproquement, une variable tournante sera adaptée pour implémenter des machines à états. En particulier, il est possible pour les variables tournantes d'effectuer du calcul dans le processus de retour Loopx afin de simplifier le processus P0. C'est par exemple le cas du décodeur d'instruction d'Aspro (paragraphe 5.5).

3.2.2.2. Transformation en machine à état

Lorsqu'un processus CHP contient des opérateurs séquentiels « ; » non-synthétisables, une transformation du processus en machine à état est possible. Cette transformation consiste à encoder les opérateurs ";" concernés par des variables d'état, puis à extraire ces variables en tant que variables tournantes. De nombreuses solutions sont possibles. Les différents compromis portent en particulier sur la manière d'assigner et d'encoder les états du processus initial.

Afin d'éclaircir la présentation, voici l'exemple succinct d'un décodeur d'instructions. Celui-ci génère différentes séquences d'ordres A, B, C, D, E qui sont des blocs d'instructions synthétisables dont les valeurs dépendent de la commande G :

```
*[ G?g ;  [ g=0 => A ; B
          @ g=1 => C ; D
          @ g=2 => E
 ]        ]
```

- **1er type de solution**

Une première solution consiste à assigner un état à chaque action qui suit un opérateur séquentiel. Un état initial est alors réservé pour exécuter l'ensemble des actions qui précèdent le premier « ; ». Pour notre exemple, deux états sont nécessaires pour les actions B et D. Soit un total de trois état avec l'état initial. La variable d'état peut être extraite sous forme de variable tournante, on obtient alors :

```
*[ CS?cs ;  [ cs=0 => G?g;  [ g=0 => A , NS!1
                            @ g=1 => C , NS!2
                            @ g=2 => E , NS!0 ]
            @ cs=1 => B , NS!0
            @ cs=2 => D , NS!0
 ]          ]
[ CS!0; *[ NS?ns; CS!ns ] ]         -- pour propager l'état
```

Le nombre d'état peut se trouver réduit dans le cas où certaines actions apparaissent à différent endroits du programme, celles-ci peuvent alors être factorisées.

- **2ème type de solution**

Une seconde solution consiste à directement encoder les opérateurs ";" de chaque garde par des variables d'état. L'introduction des variables d'état donne :

```
*[ G?g ;  [ g=0 => [A, ns:=1] ; [B, ns:=0]
          @ g=1 => [C, ns:=1] ; [D, ns:=0]
          @ g=2 => E
 ]        ]
```

L'extraction de la variable d'état *ns* permet alors d'obtenir les deux processus synthétisables suivants :

```
*[   [ #G=0 => CS?cs ; [ cs=0 => A, NS!1
                       @ cs=1 => B, NS!0, G? ]
     @ #G=1 => CS?cs ; [ cs=0 => C, NS!1
                       @ cs=1 => D, NS!0, G? ]
     @ #G=2 => E, G?
  ]  ]
[ CS!0; *[ NS?ns; CS!ns ] ]          -- pour propager l'état
```

Pour ce type de solution, le nombre d'états n'est pas donné par le nombre d'actions distinctes, mais par le nombre maximum d'états sur les branches du programme initial. Comme la garde sur G est toujours acquittée au dernier état, il n'y a plus besoin de mémoriser tous les états du processus.

De plus, la dernière garde `[ #G=2 => E ]` qui n'était pas séquentielle à l'origine n'est pas complexifiée inutilement dans ce cas. Elle reste simple car ne consomme pas d'état initial. Ceci permet d'optimiser la consommation et les performances des cas simples et favorables.

C'est ce type de décomposition en machine à état qui a été utilisé pour réaliser les processus séquentiel du processeur Aspro, en particulier pour le contrôle des registres (voir paragraphe 5.3), le contrôle de l'interface mémoire (voir paragraphe 5.5.2) et pour le décodeur d'instruction (paragraphe 5.5.1).

3.2.3. Conclusion

Nous avons défini un CHP dit « synthétisable » dont les restrictions concernent l'utilisation des variables de mémorisation et l'opérateur séquentiel « ; » . Ces restrictions proviennent directement de la méthode de synthèse de A. Martin. En effet, les variables de calcul peuvent être directement explicitées dans le protocole lors de l'expansion des communications. L'opérateur séquentiel, lorsque restreint à la modélisation des dépendances de données, est directement implémenté dans l'étape de réordonnancement.

Sans ces restrictions sur le langage CHP, il serait nécessaire pendant la phase de synthèse de construire des réordonnancement plus complexes et de rajouter des variables intermédiaires pour pouvoir implémenter variables d'états et opérateur de séquentialité « ; ». Réciproquement, grâce aux différentes possibilités de transformation sur le modèle CHP que nous avons présentées, il est possible de se ramener à un modèle CHP directement synthétisable. Ainsi, le matériel supplémentaire pour implémenter tout modèle CHP est introduit avant synthèse dans le modèle CHP de haut niveau au lieu que soient effectuées des transformations non-régulières pendant les étapes de synthèse bas niveau.

Cette approche est plus facile à appréhender que les différents choix de synthèse optimisés de A. Martin. La synthèse bas niveau reste régulière à partir du moment où un réordonnancement et une décomposition logique ont été choisis. Les transformations sur le modèle CHP sont elles-aussi régulières et pourraient être généralisées et automatisées. Ces transformations sont alors du ressort de la synthèse architecturale ou synthèse comportementale.

3.3. Conception d'une bibliothèque de cellules spécifiques

Grâce aux choix de synthèse et de mapping technologique qui ont été présentés dans le paragraphe précédent, le nombre de cellules spécifiques à développer est relativement restreint. Le jeu de cellules est principalement constitué de portes de Muller [MULL 65]. Certaines cellules plus spécifiques ainsi que des macro-cellules ont été ajoutées afin d'optimiser les performances du design en vitesse, consommation et densité. La liste de cellules et leur architecture sont présentées dans le paragraphe 3.3.1. La partie conception de la bibliothèque est rapidement présentée paragraphe 3.3.2.

3.3.1. Définition de la bibliothèque

3.3.1.1. Portes de Muller

La porte de Muller est une fonction séquentielle qui recopie la valeur de l'entrée sur sa sortie lorsque ses entrées sont égales (Tableau 3-2). Cette fonction peut être définie pour un nombre d'entrées quelconques.

S	0	1
0	0	S^{-1}
1	S^{-1}	1

Tableau 3-2 : table de vérité d'une porte de Muller à deux entrées A et B (S^{-1} dénote la valeur précédente de la sortie S).

Plusieurs architectures sont possibles pour implémenter cette fonction [MULL 65], [SUTH 89], [PIGU 99]. Tout d'abord, on peut noter qu'il est possible d'écrire la porte de Muller avec l'équation booléenne $S = A*B + (A+B)*S^{-1}$. Cette équation peut s'implémenter avec des portes classiques mais donne un circuit indépendant de la vitesse. Cette architecture aurait évité de développer des cellules spécifiques, mais comme l'objectif du projet est d'obtenir des circuits quasi-insensibles aux délais, cette possibilité n'a pas été retenue.

Nous avons défini des portes de Muller avec les critères suivants :
- 2, 3 ou 4 entrées
- normal, avec set (suffixe S), avec reset (suffixe R)
- sortie directe ou sortie inversée (suffixe B)
- sortie avec un fan-out[2] de 1 ou 2 (suffixe P)

L'ensemble de ces critères n'est pas défini pour toutes les cellules. La liste de portes de Muller dont on dispose est la suivante (soit 18 cellules) :
- CZ2[3], CZ2P, CZ2R, CZ2RP, CZ2S, CZ2SP, CZ2B, CZ2BR, CZ2BS
- CZ3, CZ3P, CZ3R, CZ3RP, CZ3B
- CZ4, CZ4P, CZ4B, CZ4BP

[2] Le fan-out est défini paragraphe 3.3.2
[3] Le nom générique CZ provient du nom de l'architecture de A. Martin : Statici**Z**ed C-element

Les portes de Muller à quatre entrées sont décomposées en porte de Muller à deux entrées. En effet, il n'est pas possible d'avoir quatre transistors PMOS en série dans la technologie cible considérée : problème de tenue à basse tension et vitesse très médiocre. La Figure 3-10 présente le schéma transistor des quatre portes suivantes : CZ2, CZ2B, CZ2R, CZ2S. A partir de ces schémas, le lecteur pourra facilement déduire la structure des portes de Muller à trois entrées.

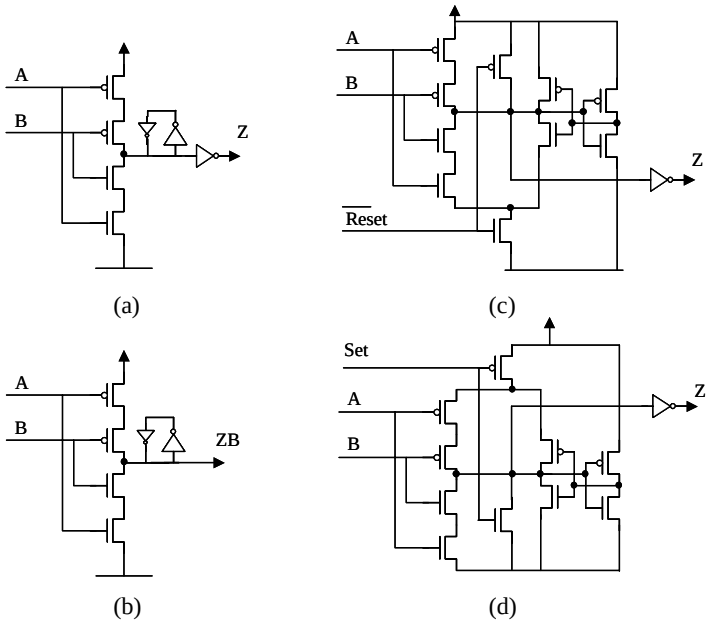


Figure 3-10 : portes de Muller à deux entrées :
a) normale, b) sortie inversée, c) avec reset, d) avec set

3.3.1.2. Portes de Muller généralisées

Les portes de Muller généralisées sont équivalentes aux portes de Muller à la différence suivante près : les signaux d'entrée qui permettent de mettre à 0 et à 1 le nœud de sortie sont distincts. Ceci permet d'implémenter les règles de production non–symétriques suivantes :

```
E1N ^ E2N => S+
E1P ^ E2P => S-
```

Le schéma de cette porte de Muller généralisée à deux entrées est donnée Figure 3-11. Ce type de porte sera en particulier utilisé pour implémenter les machines à états (voir exemples chapitre 6).

Seules des portes à deux entrées sont strictement nécessaires, il est en effet toujours possible de décomposer les règles de production en portes de Muller symétriques plus une porte de Muller généralisée. Les portes de Muller généralisées que nous avons définies sont les suivantes (soit 9 cellules) :

- GCZ2, GCZ2P, GCZ2R, GCZ2RP, GCZ2S, GCZ2SP, GCZ2B, GCZ2BR, GCZ2BS

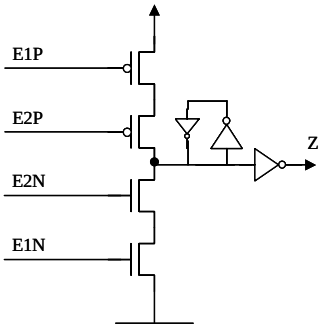


Figure 3-11 : porte de Muller généralisée à deux entrées (GCZ2).

3.3.1.3. Macro-cellules

En raison des choix de synthèse qui ont été effectués, un certain nombre de décompositions logiques en porte de Muller sont utilisées de manière systématique. Afin d'améliorer la densité du design et de diminuer les capacités dues au placement/routage (et donc d'améliorer les performances en vitesse), des macro-cellules ont été définies. Deux types de macro-cellules sont disponibles dans la bibliothèque : des cellules correspondant à des *half-buffer* ainsi que des cellules qui correspondent à l'assemblage de portes de Muller et de porte « ou » logique (soit 11 macro-cellules).

✓ **Half buffers.**

Plusieurs types de buffer sont disponibles suivant les critères suivants :
- toutes ces cellules disposent d'un signal de reset (actif bas)
- buffer simple (préfixe BR) ou « commandé » (préfixe BCR)
- buffer double-rail (suffixe _2R) ou triple-rail (suffixe _3R)
- sorties avec un fan-out de 1 ou 2 (suffixe P)

Les cellules suivantes sont ainsi définies :
- BR_2R, BR_2RP, BCR_2R, BCR_2RP
- BR_3R, BR_3RP, BCR_3R, BCR_3RP

La cellule BR_2R est exactement équivalente au schéma du half-buffer présenté paragraphe 3.1. Les cellules triple rail sont une simple extension du half-buffer double rail. Au lieu de coder une valeur binaire en double rail, il permet de coder une valeur ternaire en triple rail.

Le half-buffer « commandé » correspond à un half-buffer normal auquel est rajouté un signal de contrôle C. Ce type de cellule correspond à l'implémentation du programme CHP suivant : `[ #C => E?x ; S!x ]`. Il est implémenté avec des portes de Muller à trois entrées avec reset et une porte « nor » . La Figure 3-12 présente le schéma équivalent des portes BR_2R, BCR_2R, et BR_3R.

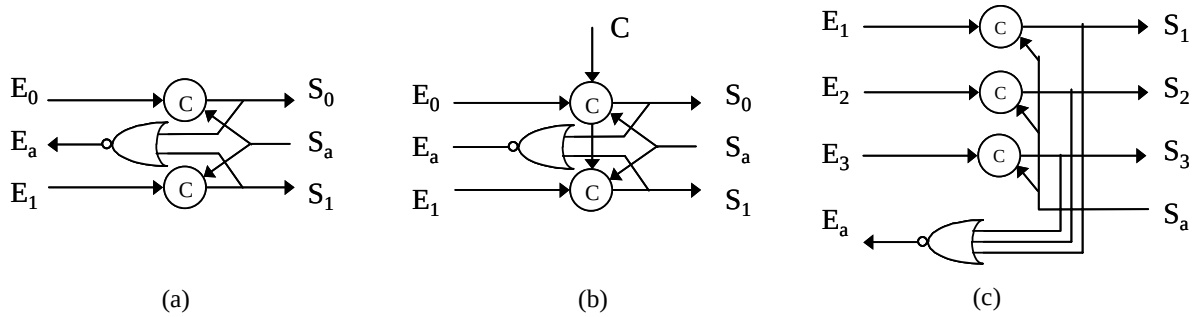


Figure 3-12 : macro-cellules de type half-buffer :
(a) double-rail, (b) double-rail "commandé", (c) triple-rail

✓ **Portes « COR »**

Ces macro-cellules sont constituées de portes de Muller et de portes « ou » logique. Les cellules suivantes sont ainsi définies :

- C2OR2, C2OR2P (C2OR2 avec fan-out de 2), C2OR2B (C2OR2 avec sortie inversée)

Une optimisation simple consiste à décomposer ces fonctions avec des portes CZ2B et NAND2, ce qui permet de gagner deux inverseurs. Le schéma équivalent de la porte C2OR2 est donné Figure 3-13.

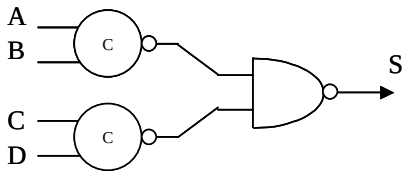


Figure 3-13 : porte C2OR2 (Muller(A, B) + Muller(C, D))

3.3.1.4. Additionneur : Full Adder 1-bit.

La cellule *Full-Adder* est une cellule qui calcule la somme de deux bits d'entrées A et B, d'une retenue entrante CI et génère un bit de somme S et une retenue sortante CO. Cette cellule est conçue afin que sa retenue sortante soit générée au plus tôt, c'est à dire sans attendre la présence de la retenue entrante CI lorsque A=B=0 ou A=B=1 (cf. modèle CHP de l'additionneur 1-bit paragraphe 5.4).

Une synthèse de cet additionneur double rail 1-bit en un ensemble de min-termes composé de portes de Muller à deux ou trois entrées est possible. Une optimisation possible de cette décomposition consiste à utiliser des portes C2OR2 (Figure 3-14).

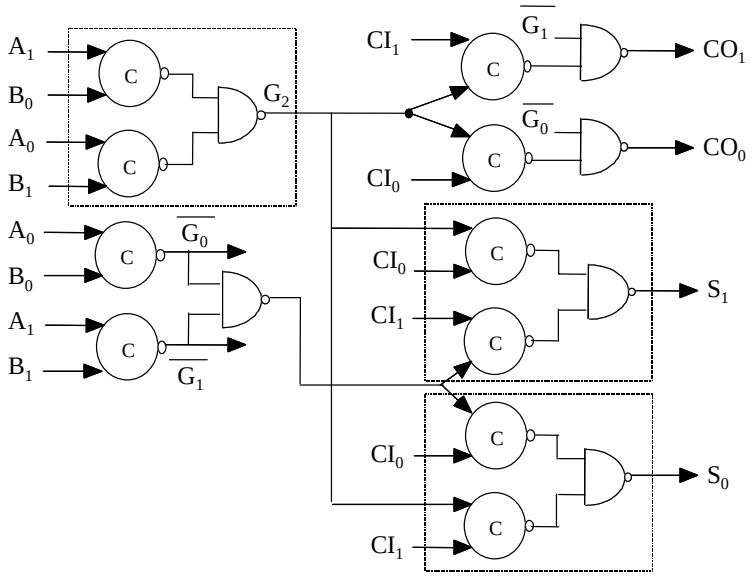


Figure 3-14 : additionneur 1 bit, décomposé en porte « C2OR2 »

Cependant, pour des raisons évidentes de densité et de performances, il était nécessaire pour le processeur Aspro d'avoir des cellules d'additions performantes, en particulier pour implémenter l'unité de multiplication-accumulation. Une cellule d'addition optimisée a été conçue dans la bibliothèque, son schéma transistor est présenté Figure 3-15. Cette cellule d'addition est une cellule non pipelinée qui respecte le protocole quatre phase. Elle génère sa retenue sortante au plus tôt et attend la remise à zéro de toutes ses entrées pour mettre à zéro ses sorties.

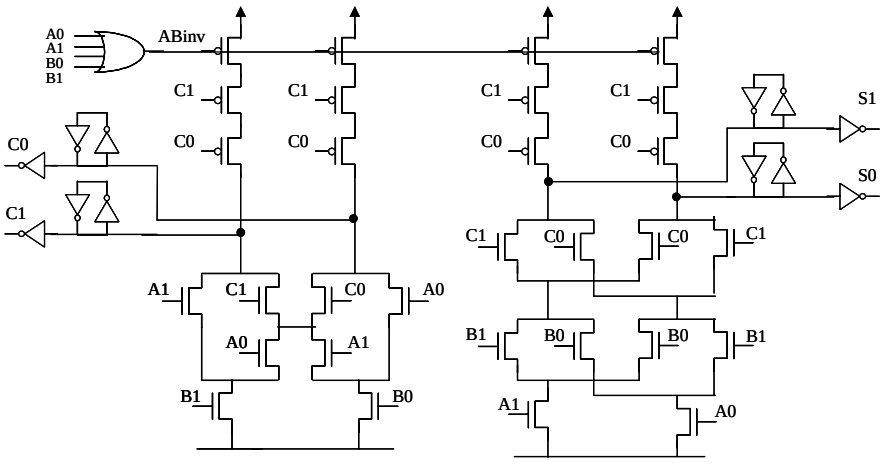


Figure 3-15 : Full-Adder 1-bit combinatoire, schéma en porte complexe

Dans cet exemple de la cellule d'addition, on retrouve ainsi le compromis vitesse / surface / coût de design entre une cellule complexe et une décomposition en un ensemble de portes de Muller. C'est la problématique qui a été présenté paragraphe 3.1.2 à propos des choix de synthèse bas niveau et de mapping technologique.

3.3.1.5. Portes diverses

Trois portes supplémentaires ont été conçues dans la bibliothèque : ME, SYNC et CV_UP.

Les portes ME et SYNC concernent spécifiquement la conception asynchrone. Ces cellules sont les briques de base indispensables pour implémenter les structures de choix CHP utilisant des gardes indéterministes. La cellule de synchronisation SYNC permet d'échantillonner un signal X à la demande d'un signal de contrôle C. Réciproquement, la cellule d'exclusion mutuelle ME permet d'effectuer l'arbitrage entre deux signaux X et Y à priori non exclusifs. Le protocole de communication implémenté par ces deux portes et leur schéma transistor respectifs sont présentés Figure 3-16 [SAKU 88].

```
Synchronisation :                Exclusion mutuelle :
[  C ^ X => U+                   [  X  => U+
@@ C ^/X => V+ ] ;               @@ Y  => V+ ] ;
  /C     => U-, V-               /X ^ /Y => U-, V-
```

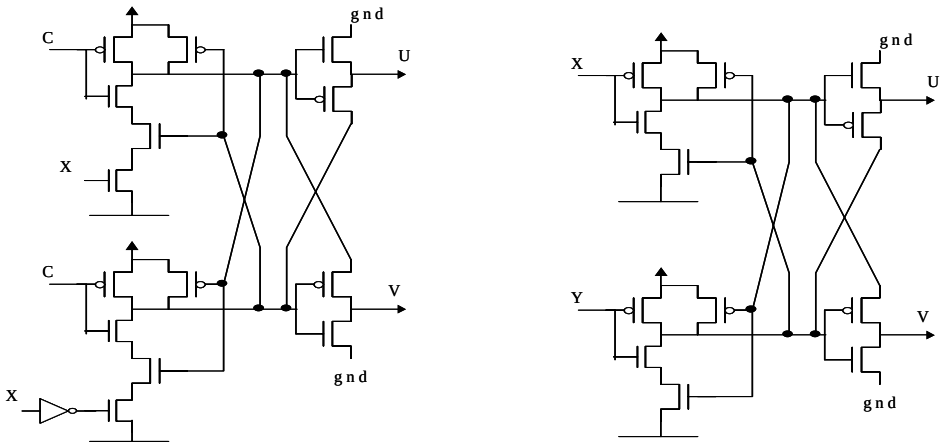


Figure 3-16 : Schéma des cellules de synchronisation SYNC et d'exclusion mutuelle ME

Ces cellules respectent le protocole quatre phase de manière évidente. Ainsi, elles s'insèrent facilement dans un bloc afin d'implémenter des gardes indéterministes. Un exemple d'utilisation sera montré avec le décodeur d'Aspro (test du signal d'interruption, chapitre 6).

Enfin, il faut surtout noter que ces deux cellules génèrent leurs signaux de sortie U et V avec un temps de réponse non borné. C'est le temps qui permet de résoudre un éventuel état métastable entre les deux inverseurs croisés. Leur architecture transistor permet ainsi de garantir l'exclusion des deux signaux de sortie U et V quelques soient les temps d'arrivée respectifs des signaux C et X ou X et Y.

La dernière porte spécifique est la cellule CV_UP qui est un convertisseur de tension. Cette cellule est alimentée à la tension $v_2>v_1$, elle convertit un signal d'entrée E (plus son complément _E) de niveau v_1 en un signal de sortie S de niveau v_2. Son schéma transistor est donné Figure 3-17. Cette cellule a été utilisée dans le processeur Aspro afin de convertir les niveaux de tension des signaux entre le cœur du processeur et les mémoires internes et entre le cœur et les plots du circuits. Cette cellule a été conçue pour fonctionner dans une large plage de tension de 0.9V à 2.5V.

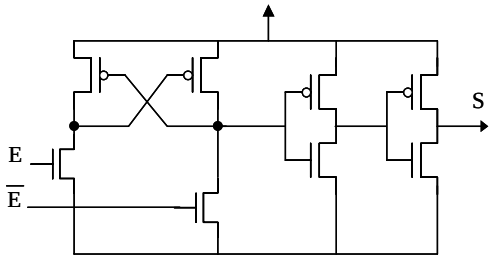


Figure 3-17 : cellule de conversion de tension CV_UP

3.3.2. Conception de la bibliothèque

La bibliothèque est au total constituée de 43 cellules spécifiques afin d'enrichir la bibliothèque standard du fondeur. Cette bibliothèque a été conçue dans la technologie HCMOS7 de STMicroelectronics (0.25µm, 6 niveaux de métaux).

La définition de la bibliothèque, la conception niveau transistor des cellules, ainsi que leur optimisation électrique ont été réalisés comme partie prenante du travail de thèse. Cependant, la conception niveau *layout* des cellules a été partagée entre une contribution conjointe du laboratoire de France Telecom R&D Meylan et du « *library-service* » de STMicroelectronics. La dernière phase de conception a été de caractériser la bibliothèque.

- **Etude électrique**

Les cellules ont été conçues afin d'être fonctionnelles dans une large gamme de tension (de 0.9v à 2.5v). Plusieurs types de cellules ont été conçues, avec un fan-out de 1 ou un fan-out de 2 (cellules avec suffixe P). Le fan-out est défini comme le ratio fan-in/fan-out d'un inverseur chargeant 5 inverseurs équivalents. La possibilité d'avoir des cellules avec plusieurs « drive » dans la bibliothèque permet de charger une capacité plus ou moins importante à la sortie de la cellule. C'est à dire la capacité totale sur un signal due : (i) au fan-in correspondant aux cellules suivantes, et (ii) à la capacité du nœud due au placement / routage.

Les tailles de transistors des cellules ont été optimisées afin d'obtenir un ratio vitesse/consommation le plus performant possible. Ces études électriques ont été effectuées grâce à des simulations de type *Spice*. Des compromis sont nécessaires. Ainsi, on se rend compte qu'une trop forte augmentation de la taille des transistors ne favorise plus la vitesse tandis que la consommation ne cesse d'être pénalisée. En particulier, comme la structure choisie pour les portes de Muller est composée de deux inverseurs rebouclés, l'inverseur de retour doit être « faible » afin d'éviter un conflit électrique trop important (consommation et perte en vitesse) lorsque la porte change d'état.

Enfin, une contrainte supplémentaire propre au fonctionnement « asynchrone » impose que les signaux de sortie des cellules soient monotones, ceci afin de garantir une logique exempte de tout aléa dynamique.

- **Caractérisation**

Après conception du niveau physique (*layout*) des cellules, la dernière phase du travail a été de caractériser les cellules de la bibliothèque afin d'obtenir des modèles VHDL avec timing *simulables* et *back-annotables*.

Le principe de la caractérisation consiste à effectuer des simulations électriques (*Spice*) de chaque cellule avec différents paramètres de simulations : pente des signaux d'entrée, capacité

de charge des cellules afin d'extraire les informations de temps de propagation et pente des signaux de sortie. Ces informations de timing permettent alors de construire un modèle VHDL tabulé. Le modèle VHDL obtenu définit à la fois la fonction de la cellule et ses caractéristiques temporelles. Ces modèles permettent alors de simuler en VHDL la *netlist* du circuit après synthèse. De plus, grâce au propriété de généricité du langage VHDL, il est ensuite possible de rétro-injecté des paramètres de timing dans le modèle VHDL de la cellule considérée. Ceci permet de simuler l'ensemble de la *netlist* du circuit en tenant compte des capacités dues au placement / routage : c'est le principe de la *back-annotation*.

Des outils puissants (outils de caractérisation / générateurs de bibliothèque) existent mais n'étaient pas adaptés aux cellules de Muller - en raison de la fonction séquentielle qui est différente des *latch* ou *flip-flop* synchrones -. Ainsi, bien que le principe d'une caractérisation était acquis, la caractérisation des cellules de la bibliothèque a nécessité un gros effort de conception. Nous avons développé des scripts spécifiques et contourné les points bloquants des outils existants (Library Compiler © Synopsys).

3.3.3. Conclusion

La bibliothèque que nous venons de présenter a été conçue en technologie HCMOS7 de STMicroelectronics. La bibliothèque représente environ 40 cellules qui sont principalement des portes de Muller : portes de Muller à 2, 3, 4 entrées, avec set, reset, portes de Muller généralisées, quelques cellules spécifiques telles qu'une cellule d'addition, des arbitres et enfin des macro-cellules pour les blocs les plus utilisés tels que les *half-buffers* afin d'augmenter la densité du design. Ces cellules ont été caractérisées / modélisées afin d'offrir un flot de conception classique : simulation, back-annotation en VHDL.

Cette bibliothèque, une fois associée à une bibliothèque standard du fondeur, permet d'effectuer la synthèse logique et le mapping technologique tels que nous l'avons présentés dans le paragraphe 3.1. La décomposition logique est effectuée en mappant les min-termes des règles de production sur des portes de Muller : ce choix n'impose pas l'utilisation de portes complexes, ce qui restreint le nombre de cellules nécessaires. Ainsi, la méthode de synthèse proposée dans le paragraphe 3.1, associée aux possibilités de modélisation du langage CHP synthétisable et à la bibliothèque de cellules standard que nous venons de présenter, permet d'effectuer une synthèse de haut niveau du langage CHP en circuit asynchrone quasi-insensibles aux délais. Des exemples de synthèse spécifiques au processeur Aspro seront présentés dans le chapitre 6.

Afin de conclure sur l'aspect méthodologie de conception de cette première partie du manuscrit, nous présentons plus en détail dans le paragraphe suivant l'ensemble du flot de conception qui a été mis au point et utilisé pour concevoir le processeur Aspro.

3.4. Proposition d'un flot de conception

La figure 3-18 présente le flot de conception qui a été utilisé pour concevoir le processeur Aspro. Pour la description du processeur et de sa micro-architecture, nous renvoyons le lecteur respectivement aux chapitres 4 et 5. Nous allons expliciter point par point les différentes étapes de conception et les outils utilisés.

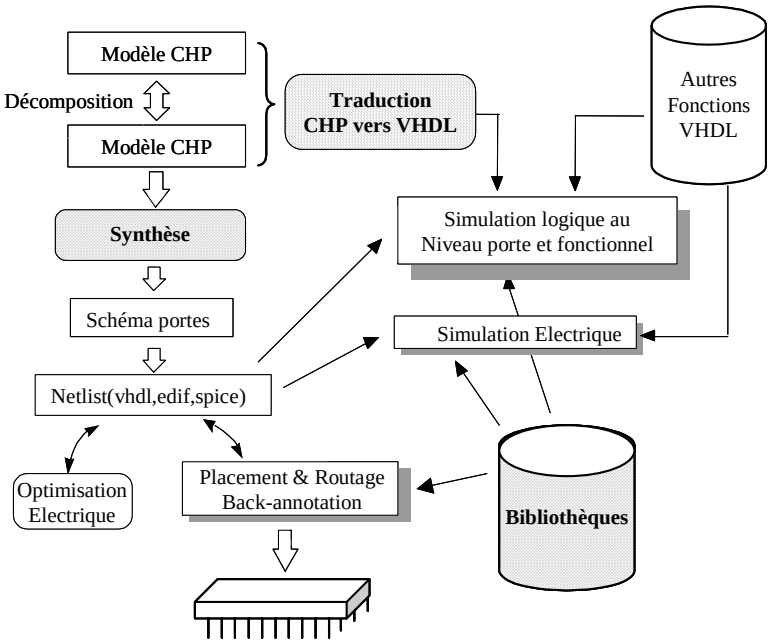


Figure 3-18 : Flot de conception utilisé pour le processeur Aspro

Les parties grisées du diagramme figure 3-18 correspondent aux différents points que nous avons développés et mis au point. Le reste des outils ou formats utilisés sont standard.

- **Modélisation CHP et validation.**

Une première spécification d'Aspro a été décrite comme un programme CHP séquentiel de haut niveau. Celui-ci a été traduit en VHDL grâce au traducteur CHP_2VHDL (chapitre 2) et validé par simulation (Synopsys VSS ©). Des programmes de test ont été écrits afin d'obtenir des simulations de référence. Ce modèle initial a ensuite été raffiné graduellement par décomposition en processus afin d'introduire de la concurrence (voir chapitre 5 pour la description de la micro-architecture) et obtenir un modèle bas niveau structurel, uniquement modélisé en CHP synthétisable (voir paragraphe 3.2). Le modèle CHP a été instrumenté afin d'observer les flots de données, les dépendances de données et les performances. Ceci a été en particulier possible grâce au possibilités de trace offerte par le traducteur. Tout au long de la phase de décomposition et d'étude des différentes solutions architecturales, le modèle CHP a été validé en VHDL par simulation. Les mêmes programmes de test ont été utilisés afin de garantir la correction fonctionnelle du nouveau modèle.

A titre d'indication, le modèle CHP de haut niveau du processeur représente environ 500 lignes de code. Le modèle bas niveau après décomposition contient environ 150 processus / composants distincts, pour 6500 lignes de code CHP.

A ce niveau de modélisation, les paramètres de timing qui ont été utilisés sont les paramètres par défaut des canaux CHP traduit en VHDL (paragraphe 2.4.4) , ils ne sont donc pas représentatifs du matériel qui sera synthétisé par la suite. Néanmoins, ceci est dans la plupart des cas suffisant pour analyser par simulation les performances du modèle : cela revient à supposer que tous les processus ont le même débit et latence. Pour plus de précision, il aurait bien sûr été possible d'introduire dans le modèle CHP des paramètres de timings plus réalistes, issus de résultats de synthèses préliminaires.

- **Synthèse logique**

En partant de modèles CHP synthétisables, la synthèse logique en circuit QDI a été effectuée avec la méthode qui a été présentée au début de ce chapitre 3. Pour résumer, nous utilisons donc un protocole quatre phase, associé à un codage double rail (voire *n*-rail par extension). Deux types de protocoles sont utilisés : le protocole WCHB pour obtenir des processus pipelinés, le protocole séquentiel pour obtenir des processus combinatoires. La décomposition logique est effectuée en mappant les règles de production sur des portes de Muller, ceci afin de ne pas utiliser de portes complexes et donc pouvoir se restreindre à une bibliothèque de cellules standard réduite. Les cellules utilisées sont en partie des cellules logiques de la bibliothèque standard du fondeur (And, Or, etc… plus les plots et des instances de mémoires synchrones), auxquelles ont été ajoutées les cellules de la bibliothèque présentée paragraphe 3.3. Les schémas logiques des différents blocs ont été saisis avec les outils Cadence© dans un environnement de saisie graphique. Ceci permet alors d'obtenir différents format de netlist : *spice*, *vhdl*, ou encore *edif*.

- **Validation niveau porte et estimation de performance.**

Comme notre bibliothèque de cellules standard est caractérisée, nous pouvons utiliser un flot de simulation classique. Au fur et à mesure de la phase de synthèse, les différents blocs après synthèse ont été validés par simulation VHDL un à un dans leur environnement CHP d'origine : c'est à dire le modèle CHP complet du processeur. Ceci a été effectué grâce aux possibilités de co-simulation CHP / VHDL présentées dans le paragraphe 2.4.7. Comme pour la validation haut niveau, les mêmes stimuli et programmes de test ont été utilisés afin de valider les différents blocs. Une fois le processeur complètement synthétisé, nous avons finalement obtenu une unique netlist VHDL qui a été validée par simulation de la même manière.

En plus de la validation fonctionnelle du résultat de synthèse de chaque bloc, des simulations VHDL avec timings ont été effectuées afin d'obtenir des informations de timings tel que temps de cycle et latence. Ceci nous a aidé à choisir entre différentes alternatives architecturales et aussi à effectuer l'optimisation globale du pipeline asynchrone (voir chapitre 6). Grâce à VHDL, les temps de simulation sont très rapides, ce qui permet aisément de comparer différentes solutions et de simuler/valider l'ensemble du processeur sur un nombre important de pattern de tests.

Des simulations niveau transistors ont aussi été effectuées pour obtenir des estimations de performance plus précises. Le simulateur Eldo © (Anacad / MentorG) – simulateur niveau *spice* -a été utilisé pour la simulation / validation de petits blocs. Le simulateur PowerMill ©

(Epic / Synopsys) – simulateur niveau *switch* - a été utilisé afin d'obtenir des estimations de consommation du processeur en effectuant la simulation de quelques dizaines d'instructions.

Enfin, des co-simulations en VHDL ont aussi été effectuées avec des blocs synchrones tel que les blocs mémoires utilisés pour implémenter les mémoires du processeur. Ceci a ainsi permis de valider les hypothèses temporelles (temps d'accès, setup, hold) des interfaces synchrones / asynchrones (chapitre 6) en utilisant les modèles VHDL des RAMs du fondeur.

A ce niveau du flot de conception, on peut noter que différents protocoles peuvent coexister dans une même simulation VHDL, mélangeant à la fois niveau porte et niveau fonctionnel : protocole quatre-phase double-rail niveau porte, protocole quatre-phase fonctionnel (celui des canaux CHP traduit en VHDL), même chose pour le protocole deux-phases des liens série du processeur, protocole données groupées pour les liens parallèles du processeur, protocole synchrone. Tout ceci est bien sûr possible grâce aux possibilités de modélisation mixte offerte par VHDL et à l'ajout d'interfaces (matériels ou fonctionnels pour la simulation) là où c'est nécessaire.

- **Placement-routage et back-annotation**

Le placement & routage a été effectué avec les outils Avanti! ©. Afin de gagner du temps, nous n'avons pas effectué un réel routage par bloc. On peut tout de même noter que les différents blocs mémoires ont été correctement placés sur les bords du circuit, ensuite différentes zones topologiques ont été définies entre les mémoires pour aider le placement des différents blocs de la hiérarchie. C'est aussi à ce moment que nous avons défini la couronne de plot du circuits : plots d'entrée et de sortie, plots d'alimentation (avec trois alimentations distinctes : pour le cœur, pour les mémoires, pour les plots) et inséré des cellules de conversion de tension CV_UP.

Après chaque placement & routage, nous obtenons un fichier de description de l'ensemble des capacités de routage. Ceci permet de re-calculer les timings du circuit pour chaque porte (on obtient un fichier *.sdf : standard delay file*), et donc de back-annoter la netlist VHDL. Nous pouvons alors simuler l'ensemble du processeur en VHDL tout en tenant compte des éléments de routage. Ceci permet de s'apercevoir que certains nœuds du circuits sont trop chargés.

En fonction des capacités de routage observées, la méthode d'optimisation consiste ensuite à remplacer les portes dont le fan-out est trop faible par la même porte logique mais avec un fan-out plus élevé (par exemple remplacer CZ2 par CZ2P). Lorsqu'une porte de fan-out plus élevé n'est pas disponible dans la bibliothèque, il faut alors insérer des buffers électriques ou des chaînes d'inverseurs afin de diviser la capacité de routage le long des nouveaux nœuds crées. Le but est donc bien d'obtenir des signaux électriques avec des pentes de montée / descente correctes, même si on perd en délai. Cette méthode d'optimisation est proposée par les outils de synthèse Synopsys et s'appelle l'IPO : *In Place Optimization*. Ensuite, au niveau de l'outil de placement & routage, il ne reste plus qu'à remplacer les cellules qui ont changé, et placer/router les nouvelles cellules.

Malheureusement, en raison de l'absence d'horloge dans nos circuits, nous n'avons pas pu utiliser les outils Synopsys. Nous avons donc développé un programme simple pour effectuer cette phase d'optimisation. Après un placement/routage initial, plusieurs itérations d'optimisation peuvent été nécessaires afin d'obtenir des performances électriques correctes. Après chaque itération, la nouvelle netlist VHDL est back-annotée et simulée afin d'effectuer sa validation fonctionnelle et obtenir de nouvelles estimations de performances.

Après ces différentes optimisations, il ne reste plus qu'à effectuer les dernières étapes de conception : vérification des règles de dessin (DRC), vérification de la cohérence entre le

schéma (la netlist) et le *layout* (LVS), et enfin génération du fichier GDS2, qui contient la description physique du circuit.

3.5. Conclusion

Le flot de conception de circuits asynchrones quasi-insensibles aux délais que nous venons de présenter, a permis de réaliser avec succès le processeur Aspro. Nous renvoyons le lecteur au chapitre 6 pour les résultats du processeur. Ce flot propose une méthode de conception utilisant des cellules standard, qui se base sur l'utilisation de formalismes standard et d'outils commerciaux. Nous avons donc contribué sur les différents points suivants : le développement d'un traducteur CHP vers VHDL, une méthodologie de synthèse en circuit quasi-insensible aux délais et une bibliothèque de cellules spécifiques.

La méthodologie de synthèse est manuelle à ce jour. Cependant, grâce aux possibilités de description du langage CHP synthétisable, notre méthodologie offre une approche de conception descendante quasi-systématique. Ce travail a ainsi donné lieu en 1998 lors d'un stage de 6 mois à un tout premier outil de synthèse logique [ALAD 98]. A partir d'une description en CHP synthétisable, cet outil permet de générer un circuit logique insensible aux délais qui est correct mais sans optimisation logique. Comme présenté dans le paragraphe 3.1, le problème de la décomposition et de l'optimisation logique est en effet un point délicat. Ainsi au niveau de la synthèse logique, on pourrait imaginer différents type d'optimisations : optimisation en vitesse, surface, consommation, avoir le choix de différents type de réordonnancement (combinatoire, pipeliné) afin d'optimiser les architectures. Un autre niveau de synthèse pourrait aussi être considéré afin d'effectuer automatiquement sur la spécification CHP des transformations telles que celles présentées dans le paragraphe 3.2.2 afin d'obtenir un CHP synthétisable.

Enfin de manière plus générale sur le flot de conception, de nombreux travaux sont encore nécessaires afin d'offrir une plate-forme de conception complète :

- amélioration et extension de ce tout premier outil de synthèse proposés,
- utilisation des outils standard pour obtenir des mesures de performance en effectuant du calculs de timings sur la netlist (*static timing analysis*),
- utilisation des outils standard pour effectuer les optimisations électriques (*IPO*),
- vérification de la validité de l'hypothèse de fourche isochrone après les étapes de placement & routage,
- étude du problème de la testabilité afin d'offrir des solutions de conception adaptées pour le test.

Ces différents points n'ont pas été abordés pendant le travail de la thèse, ils sont du ressort de la recherche actuelle. Les études à mener sont encore importantes et sont nécessaires pour pouvoir envisager une utilisation de ce type de méthodologie de conception dans un contexte industriel.

Partie II

Le processeur Aspro : conception d'une architecture RISC 16-bit quasi-insensible aux délais

Cette deuxième partie du manuscrit regroupe les chapitres 4, 5 et 6. Tout d'abord, le chapitre 4 propose une architecture de processeur asynchrone dénommé Aspro. C'est un processeur RISC 16-bit incluant une unité utilisateur et des périphériques dédiés. Son architecture originale permet d'exécuter les instructions dans l'ordre et de les terminer dans le désordre grâce à l'utilisation de quatre unités d'exécution concurrentes et à un mécanisme de réservation dans le banc de registre. Le chapitre 5 détaille par la suite la micro-architecture du processeur. A partir de l'étude du modèle CHP du processeur, nous montrons qu'il est possible de réaliser des unités à consommation et temps de calcul minimums en fonction des instructions et des données, le tout étant implémenté grâce à des mécanismes de control local. Pour terminer, le chapitre 6 présente l'implémentation du processeur et son optimisation en utilisant la méthodologie de conception proposée dans la première partie du manuscrit. Les résultats mesurés sur le circuit seront alors présentés.

Chapitre 4 :

Spécification et architecture du processeur Aspro

4.1. Introduction

En 1996 lors du début de la thèse, les différents objectifs du projet « Aspro » étaient multiples. On se posait alors les trois questions suivantes : a) est-ce possible de concevoir un circuit complexe asynchrone ? b) peut-on cibler des cellules standard en logique QDI ? c) peut-on spécifier et automatiser un flot de conception pour ce type de circuit ? L'objectif du projet Aspro était donc de réaliser un prototype pour démontrer la faisabilité en terme de circuit et obtenir des mesures de performances sur ce type de logique, le tout étant un vecteur de recherche pour nourrir la réflexion à la fois sur les méthodes de conception et les architectures asynchrones.

4.1.1. Pourquoi un processeur ?

Nous nous sommes intéressés à la conception d'un microprocesseur asynchrone pour plusieurs raisons. Tout d'abord, comme présenté dans le chapitre 1, Aspro n'est pas le premier microprocesseur asynchrone [GAGE 98], [TERA 99], [MART 89], [MART 97], [WOOD 97], [FURB 97], [NANY 94], [TAKA 97]. La conception d'un processeur permet donc d'avoir des points de comparaisons, même si il est très délicat de conclure en comparant des architectures différentes. S'attaquer à la conception d'un microprocesseur est un projet d'envergure et d'une complexité significative. L'équipe avait déjà différentes expériences de conception de

circuits asynchrones. Ainsi, il avait été étudié et conçu des opérateurs arithmétiques asynchrones (additionneur, multiplieur, diviseur) en logique DCVSL [RENA 94a], [RENA 94b], [ELHA 95]. Dans la suite de ces premières études, il a été conçu un système multiprocesseur asynchrone dédié au traitement d'image (traitement morphologique mathématique) [ROBI 96], [ROBI 97a], [ROBI 97b]. Ce dernier système est d'une complexité significative mais les briques de base dont il est composé sont relativement peu élaborées. En partant de ces différentes expériences, la conception d'un microprocesseur offre alors l'opportunité d'étudier de nombreux types de blocs architecturaux, tel que mémoires, banc de registres, unités d'exécution, unités arithmétiques, et unités de contrôle. Ceci permet d'explorer de nouvelles voies au niveau de l'architecture des circuits numériques et de proposer des architectures adaptées au style des circuits asynchrones. De plus, pour faire face à un certain niveau de complexité, il est nécessaire d'élaborer des techniques et méthodes de conception rigoureuses. Les méthodes de conception qui ont été proposées pour concevoir le processeur ont été présentés dans la première partie du manuscrit. Ainsi, même si il est difficile de comparer des résultats quantitatifs (vitesse, consommation, surface) entre différents microprocesseurs asynchrones, la conception d'un projet significatif tel que celui-ci nous autorise à tirer des conclusions sur la pertinence des méthodes de conception utilisées.

Nous avons délibérément choisi de ne pas cloner un microprocesseur synchrone existant. La définition et la conception d'un prototype permet de se libérer totalement de toute contrainte synchrone, que ce soit au niveau du jeu d'instructions ou de l'architecture. Ce choix laisse effectivement un grand espace de liberté pour le concepteur. La voie est ainsi libre pour explorer de nouvelles solutions architecturales adaptées au style asynchrone, aussi bien au niveau du jeu d'instruction que de l'architecture ou la micro-architecture du processeur. Le but est bien d'étudier et d'offrir des solutions architecturales différentes des solutions synchrones classiques. Ce processeur a été dénommé Aspro, acronyme de « *Asynchronous Processor* ».

4.1.2. Une architecture adaptée à l'asynchrone

Au début du projet, nous disposions bien sûr de peu d'expérience dans la conception de microprocesseur asynchrone. Les études de [ROBI 97] avaient montré que la notion d'asynchronisme pouvait s'appliquer à différents niveaux, aussi bien au niveau algorithmique, architecture système que implémentation circuit. Il paraissait alors évident que pour obtenir des performances élevées dans les systèmes VLSI il était nécessaire de relâcher les synchronisations à tout niveau du circuit. Ce relâchement des contraintes de synchronisations correspond à des architectures flot de données, architectures parfaitement adaptées au style de modélisation des processus communiquant utilisés dans le langage CHP. Il semblait alors naturel d'étudier et d'implémenter une architecture régulière et simple qui soit adaptée à une description flot de données, à savoir typiquement une architecture RISC [HENN 96].

Ce type d'architecture présente un flot de contrôle régulier qui permet d'utiliser la notion de pipeline pour effectuer en parallèle les différentes tâches d'exécution sur des données successives. Dans une architecture RISC (Reduced Instruction Set Computer), les instructions sont simplifiées le plus possible. Le jeu d'instructions est qualifié d'orthogonal, dans le sens où il n'y a pas deux manières distinctes dans le jeu d'instructions pour exécuter la même fonctionnalité. Le jeu d'instructions est donc réduit et peut être encodé sur un mot de longueur constante, ce qui simplifie à la fois la tâche de lecture instruction et de décodage instruction. Les accès aux données sont simplifiés, il n'y a pas de mode d'adressage complexe. Seules des instructions spécifiques ont le droit d'accéder aux données en mémoire, toute autre instruction

utilise les données qui sont stockées localement dans un banc de registres. C'est à la charge du compilateur d'effectuer les accès mémoires quand nécessaire pour charger / sauvegarder les registres. Ce type d'architecture permet ainsi de réduire les synchronisations au sein du circuit, aussi bien au niveau du contrôle du flot d'instruction, que du contrôle des données utilisées par ces instructions. Ceci est bien sûr en faveur du taux de parallélisme potentiel, comparativement à une architecture de microcontrôleur qui impose un fort séquencement. On trouvera dans [HENN 96] une comparaison plus complète entre architectures RISC et CISC.

Les propriétés que nous venons de présenter sont bien sûr très générales, elles s'appliquent aussi bien à des architecture implémentées avec des circuits synchrones ou des circuits asynchrones. Cependant, dans le cadre d'implémentation de circuits asynchrones, il paraissait beaucoup plus facile d'implémenter une architecture parallèle, qui puisse s'implémenter avec une description flot de données, plutôt qu'une architecture séquentielle, qui s'implémente usuellement avec des machines d'états synchrones. Non seulement cela semblait plus facile mais plus opportun d'étudier et d'implémenter des architectures relâchées pour exploiter le potentiel des circuits asynchrones en terme de vitesse, calcul en temps moyen, et modularité. Le but était donc d'exploiter par une implémentation asynchrone le potentiel de parallélisme offert par une architecture RISC. Nous avons défini pour Aspro une architecture RISC 16-bit, son jeu d'instructions est présenté en détail dans le paragraphe 4.2, l'architecture du processeur est présentée dans le paragraphe 4.3. Néanmoins, au fur et à mesure de l'avancée du projet, nous remarquerons qu'il est tout à fait possible et pertinent d'implémenter des architectures séquentielles en asynchrone, y compris en utilisant une description flot de données [ABRI 00].

4.1.3. Applications et potentiels

Les applications visées pour le processeur Aspro sont multiples. Tout d'abord, il se veut un processeur d'usage général. Ceci permet de montrer qu'un circuit asynchrone se comporte tout comme un circuit synchrone vu de l'utilisateur. Celui-ci possède un jeu d'instructions, un compilateur, des périphériques… La mise en application d'un circuit programmable est bien un facteur de reconnaissance de la logique asynchrone : le principe asynchrone ne se retrouve pas cantonné à des applications très spécifiques de type circuit dédié ASIC. Le jeu d'instructions est donc spécifié pour offrir une machine RISC complète : arithmétique, contrôle, etc.

En raison de la robustesse des circuits asynchrones, le processeur peut être utilisé dans des applications régulées en tension [NIEL 94]. Dans ce type de circuit, en adaptant la tension d'alimentation du circuit, il est extrêmement facile d'échanger puissance de traitement contre consommation d'énergie (cf. paragraphe 1.2). Le but est donc pour une application donnée de choisir un point de fonctionnement vitesse / consommation en fonction d'un budget consommation ou de contraintes de performances.

Afin de montrer les potentiels de modularité des systèmes asynchrones, l'architecture du processeur est enrichie d'une unité utilisateur ainsi que de périphériques dédiés (voir paragraphe 4.2.3). Ces unités sont directement intégrées au sein du cœur du processeur, ce qui enrichi le jeu d'instructions de base. La modularité des circuits asynchrones permet de connecter ces unités quelque soit leur fonctionnement interne et leur performance. L'unité utilisateur intègre une unité de multiplication-accumulation afin d'offrir des possibilités de traitement de type DSP. Le processeur peut alors être utilisé dans des applications gourmandes en puissance de calcul comme des applications de traitement du signal (FFT, code correction d'erreur) ou dans des applications de traitement d'image (codage/décodage,

segmentation). Le processeur peut alors facilement être intégré dans des applications de téléphonies, y-compris pour des applications embarquées en raison de sa faible consommation.

L'objectif ultime du processeur est de montrer que l'asynchrone est aussi adapté à une conception efficace de systèmes multiprocesseur intégrés dans les technologies avancées. Dans la suite des travaux de [ROBI 97], les périphériques du processeur ont été définies afin d'offrir de puissants moyens de communication et de synchronisation avec l'extérieur. Le processeur constitue alors une brique de base programmable pour construire des systèmes multiprocesseur. Ceci est donc une première étape vers la conception de systèmes multiprocesseur intégrés.

D'une manière plus générale, la motivation du processeur est de montrer que l'asynchrone est aujourd'hui adapté à la conception de systèmes complexes, pour pleinement tirer profit des potentiels des technologies avancées, principalement en terme de vitesse et consommation, tout en offrant des propriétés intrinsèques tel que robustesse, modularité, scalabilité, réutilisabilité, adaptabilité au conditions de fonctionnement.

4.1.4. Environnement logiciel

Comme cela a été présenté, nous avons décidé de ne pas cloner un microprocesseur synchrone existant afin de ne pas contraindre avec des concepts synchrones l'étude au niveau de l'architecture. Ce choix bien sûr nous prive de tout environnement logiciel existant : assembleur, compilateur, debugger, bibliothèques de logiciels. Le développement complet d'une chaîne de compilation est un travail considérable et nécessite des compétences particulières. Une solution plus aisée consiste à paramétrer un compilateur existant (par exemple gcc) afin de l'adapter au jeu d'instructions de la machine cible.

Néanmoins, le laboratoire de France Telecom R&D Meylan possédait une expertise en compilation optimisée avec les outils SAXO [BAUE 97]. Ces outils ont été développés pour effectuer de la compilation de langage C optimisée sur architectures DSP. Ceci consiste en une chaîne de compilation qui est paramétrée par le modèle d'instructions de la machine cible et de son architecture. Le jeu d'instructions du processeur Aspro a été discuté avec cette équipe afin de s'assurer que celui-ci était complet et orthogonal, pour qu'il puisse être la cible d'un tel compilateur. Le jeu d'instructions RISC d'Aspro a donc été défini (cf. paragraphe 4.2) et modélisé dans leurs outils. Ceci nous a alors permis d'obtenir une première version de compilateur. Malheureusement, en raison de manque de temps, le compilateur n'a pas pu être totalement validé et surtout optimisé. Pendant la conception du processeur, nous avons disposé uniquement de l'assembleur, ce qui était suffisant pour valider l'implémentation de l'architecture (cf. paragraphe 3.4). Le développement des outils est aujourd'hui un travail qui est poursuivi à TIMA afin de pleinement exploiter l'architecture et les performances du processeur (cf. paragraphe 4.4).

4.1.5. Conclusion

L'objectif du projet Aspro est donc de concevoir un microprocesseur RISC 16-bit performant, ayant des possibilités de traitement DSP (via l'unité utilisateur) et des périphériques intégrés. Ce processeur est bien sûr avant tout un microprocesseur asynchrone, il sera implémenté en cellules standard avec la logique quasi-insensible aux délais présentée dans le chapitre 3. Les objectifs initiaux en performances sont de 200 Mips pour une consommation de 500mW dans une technologie CMOS 0.25µm.

Le paragraphe 4.2 présente les spécifications du processeur : son jeu d'instructions, ses périphériques. Le paragraphe 4.3 présente ensuite globalement son architecture. La micro-architecture du processeur sera présentée dans le chapitre 5, tandis que l'implémentation, son optimisation et les résultats seront présentés dans le chapitre 6.

4.2. Jeu d'instructions et périphériques

4.2.1. Une architecture RISC 16-bit scalaire

Le jeu d'instructions retenu pour le processeur Aspro est un jeu d'instructions RISC 16-bit qui propose les instructions classiques d'arithmétiques, de branchements et d'accès mémoire, ainsi que quelques instructions plus spécifiques, 16 registres généraux, pas de registre d'état, des périphériques intégrés, et enfin une unité utilisateur.

L'architecture mémoire est une architecture Harvard. Il y a donc deux mémoires distinctes : une mémoire dédiée aux programmes et une mémoire dédiée aux données. Comme c'est une architecture 16 bits, l'espace adressable est limité à 64 k (que ce soit pour la mémoire données ou la mémoire programme). Afin de simplifier la conception de ce premier prototype Aspro, il n'a pas été prévu d'intégrer un mécanisme de mémoire cache mais simplement d'intégrer tout ou partie de l'espace adressable. Vu la complexité en surface des mémoires dans la technologie visée (technologie CMOS 0.25µm de STMicroelectronics), les ressources mémoires sont les suivantes : 48 Koctets de mémoire programme on-chip (soit 16 Kmots instructions codées sur 24 bits), 144 Koctets de mémoire programme accessible en externe via le port A et enfin 64 Koctets de mémoire données, accessibles par mot ou par octet, totalement implémentés sur le circuit.

Les accès en lecture/écriture à la mémoire données ne sont effectués qu'avec des instructions chargement-sauvegarde dédiées à cet effet. Aucune autre instruction ne peut accéder à la mémoire donnée : ceci correspond bien à une architecture RISC. Le mode d'adressage de la mémoire est restreint au mode indexé avec ou sans déplacement. Tout autre calcul d'adresse sera donc effectué via les registres banalisés.

Les 16 registres généraux sont accessibles sous format d'instructions trois opérandes : deux opérandes sources et un registre destination (par exemple l'instruction "add reg0, reg1, reg2" réalise l'opération "reg0 := reg1 + reg2"). Dans cette architecture, il n'a pas été prévu d'implémenter de registre d'état. Cet argument est en faveur de la régularité du pipeline du chemin de données du microprocesseur. Les calculs de branchements et le contrôle des exceptions arithmétiques seront donc fait par des instructions particulières dédiées à cet effet.

L'architecture proposée pour le processeur asynchrone Aspro est une architecture scalaire [HENN 96]. Les instructions sont lues et décodées en séquence puis envoyées à différentes unités d'exécution. Grâce à quatre unités d'exécution indépendantes et à un simple mécanisme de réservation dans le banc de registres, les instructions sont alors exécutées en parallèle dans les unités, et se terminent dans le désordre suivant les dépendances entre instructions. Ce schéma d'exécution, comme on le montrera dans le paragraphe 4.3, s'implémente aisément avec une architecture asynchrone et permet de tirer naturellement parti du parallélisme instruction.

4.2.2. Périphériques et signaux externes

La Figure 4-1 présente le descriptif des signaux externes du processeur Aspro. Le fonctionnement des entrées-sorties du processeur est détaillé dans les paragraphes qui suivent.

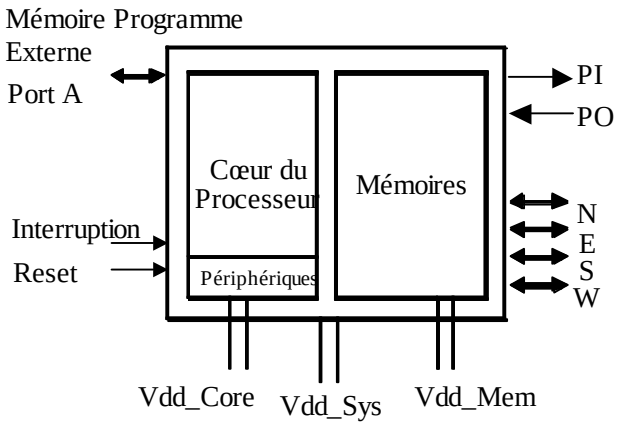


Figure 4-1 : Vue externe d'Aspro

4.2.2.1. Port mémoire programme

Le port A permet d'accéder à la mémoire programme externe. Son espace d'adresse est défini de $3FFF à $FFFF (soit 48Kmots de 24 bits). Ce port est conçu pour être compatible avec des mémoires synchrones existantes. Il est composé (Figure 4-2) d'un bus d'adresse 16 bits, d'un bus de donnée 24 bits et de deux signaux de contrôle Rreq et Rack (*Read request* et *Read acknowledge*).

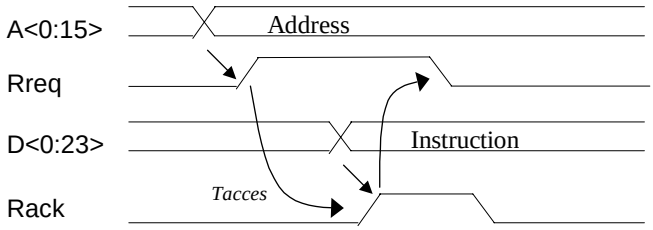


Figure 4-2: Protocole du port A : accès mémoire externe

Le signal Rreq est émis à 1 avec une adresse valide lorsqu'un accès mémoire est demandé, la mémoire doit répondre avec le signal Rack à 1 et avec la donnée. Le signal Rreq peut alors être connecté à l'entrée d'horloge de la mémoire. Le délai Rreq-Rack correspond alors au temps d'accès de la mémoire, il peut être émulé grâce à une ligne à retard correspondant au temps de cycle de la mémoire choisie. Un interface entre la logique de type bundle-data du port A et la logique QDI du cœur du processeur a été implémenté (voir chapitre 6).

4.2.2.2. Ports parallèles

Afin de pouvoir connecter divers périphériques (clavier, afficheur, timer, etc…), deux ports parallèles 16 bits sont disponibles. Ces deux ports d'entrées-sorties sont respectivement nommé PI et PO, ils sont définis à l'adresse $00 dans l'espace adressable des périphériques.

Comme pour le port A, dans le but de minimiser le nombre d'entrées-sorties du circuit, l'encodage double-rail présenté chapitre 3 n'est pas utilisé pour ces mots parallèles 16 bits. Ces deux ports sont alors constitués chacun d'un bus 16 bits et de leurs signaux de requête et d'acquittement. Ces ports implémentent un protocole quatre phases de type bundle-data comme montré dans la Figure 4-3. Une hypothèse de temps de *setup* existe et doit donc être vérifiée entre la validité des données et le signal de requête. Comme pour le port A, des conversions de protocole entre le cœur du circuit et ces périphériques sont implémentées.

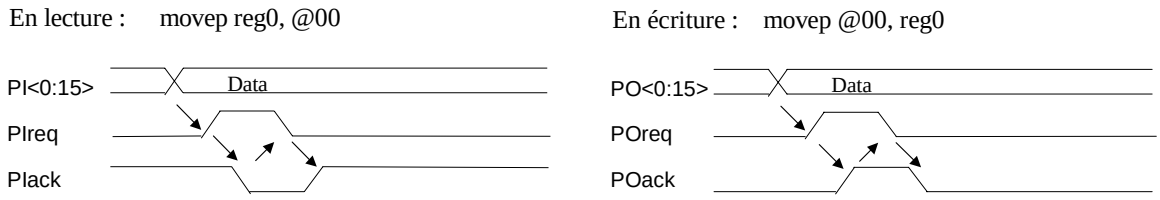


Figure 4-3 : Chronogramme des ports d'entrées/sorties PI et PO

Pour utiliser ces ports dans un système synchrone, plusieurs possibilités sont offertes. Les signaux de requête peuvent être assimilés à des signaux d'horloge, le signal d'acquittement étant alors généré par une ligne à retard sur ce même signal d'horloge. Dans le cas où aucune synchronisation n'est nécessaire, il est possible de connecter ensemble les signaux de requête et d'acquittement. Dans ce cas, les ports fonctionnent à vitesse maximale de manière autonome, le protocole étant correct par construction. Pour une mise en œuvre dans un système asynchrone, grâce aux signaux de requête et d'acquittement, ces ports peuvent permettre de dialoguer et de se synchroniser sur des modules externes comme avec les liens série présentés ci-dessous.

4.2.2.3. Liens séries

Dans l'optique de la mise en œuvre de systèmes multiprocesseurs et afin de montrer qu'il est possible de concevoir des liens de communication inter-processeurs robustes et rapides, Aspro intègre aussi des liens de communications plus spécifiques. Quatre liens séries bidirectionnels nommés Nord, Est, Sud, Ouest sont définis aux adresses respectives $01, $02, $03, $04 dans l'espace adressable des périphériques.

Les mots 16 bits du cœur du microprocesseur sont envoyés en série, le poids fort étant par convention envoyé en tête. Le protocole utilisé est un protocole QDI deux phases [Mar90]. Ces liens de communications sont extrêmement robustes car il n'y a aucune hypothèse de délai aux interfaces du processeur (contrairement au port A et aux deux ports parallèles PI et PO). De plus, l'utilisation d'un protocole deux phases permet de minimiser l'impact du délai important des plots d'entrée-sortie sur les performances. Ainsi dans le temps de cycle définissant l'envoi d'un bit, les plots du circuit ne sont traversés que deux fois au lieu de quatre dans le cas d'un protocole quatre phases.

Chacun des 8 ports (les quatre cardinaux, en entrée ou en sortie) est constitué de trois signaux : un signal pour encoder la valeur *zéro*, un signal pour encoder la valeur *un* et un signal d'acquittement. La Figure 4-4 montre en exemple l'écriture de la valeur 0110 sur le port de sortie Nord (port NO). Au reset, tous les signaux sont initialisés à zéro. Le protocole est dit actif en écriture et passif en lecture : pour un port de sortie, le processeur émet une transition sur l'un des signaux de données (signal 0 ou 1 suivant le bit à émettre) puis attends un acquittement par transition du signal d'acquittement ; réciproquement pour un port d'entrée, le processeur attends de recevoir une transition sur l'un des deux signaux de donnée pour ensuite

acquitter celle-ci. Ce protocole est effectivement insensible aux délais car chaque transition du processeur émetteur est acquittée par une transition du processeur récepteur.

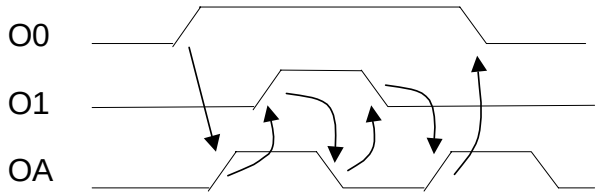


Figure 4-4 : Protocole des liens série DI 2 phases.

Les détails de l'implémentation de ce protocole DI deux phases et en particulier la conversion entre la logique quatre phases du cœur du microprocesseur et ces périphériques sont donnés dans le chapitre 6. Les performances obtenues sur testeur sont de l'ordre de 50 Mbit/s, soit 6.25Moctet/s.

La mise en œuvre de ce type de liens de communication permet de démontrer plusieurs points. Tout d'abord au niveau conception de système, ces liens sont très robustes. Ceci permet l'assemblage de composants de manière très rapide sans problème d'interconnexion, sans hypothèse de délais aux interfaces. En terme d'architecture de systèmes répartis, ces liens servent non seulement à échanger des données entre différentes ressources matérielles mais sont aussi et surtout un moyen de synchroniser ces différentes ressources matérielles. Cette synchronisation au niveau système respecte exactement la même sémantique que le CHP (voir chapitre 2). La synchronisation au niveau matériel, inhérente à l'implémentation asynchrone, permet d'implémenter au niveau système la synchronisation de tâches logicielles : *il y a unité de la sémantique de synchronisation du logiciel jusqu'au matériel*. Si on construit un réseau de microprocesseurs avec ces liens de communications, les éléments du réseau pour dialoguer, doivent se synchroniser afin de pouvoir échanger des données. Le modèle de programmation est donc totalement différent. Les différentes unités processeurs ne dialoguent pas par interruptions successives mais simplement par lecture ou écriture via des périphériques. Il suffit ainsi dans le programme d'effectuer la lecture ou l'écriture sur le périphérique pour imposer la synchronisation avec l'extérieur et échanger la(les) donnée(s).

La mise en œuvre d'applications complexes, les possibilités offertes par ces mécanismes de synchronisation ont été étudiées dans [ROBI 97], bien des travaux restent à mener pour continuer de les exploiter. Le prototype Aspro est justement une première brique de base programmable pour ces futurs systèmes multiprocesseurs asynchrones.

4.2.2.4. Gestion des interruptions, mode veille

Un unique niveau d'interruption est disponible pour Aspro avec possibilité de masquage. Le drapeau d'interruption est mis à jour par instruction (Dint : Disable Interrupt / Eint : Enable Interrupt) et est initialisé à zéro au reset. La gestion de l'interruption est précise [HENN 96] : si un "un" logique est présent sur le signal d'interruption INT, lorsque le masque d'interruption est valide, Aspro sauve son compteur de programme dans le registre 15, inhibe les interruptions, branche à une adresse prédéfinie ($0000) en mémoire interne et enfin acquitte l'interruption en émettant un "zéro" sur le signal INTack, ce qui indique à l'extérieur que l'interruption est prise. Dans cette version d'Aspro, il n'est pas défini différents niveaux et priorités d'interruption. Par contre, il est possible d'autoriser des interruptions imbriquées,

elles doivent être supportées par le programme. Notre objectif est de montrer que nos architectures asynchrones peuvent tester un signal externe au système et donc indéterministe.

Outre le mécanisme d'interruption, une instruction spécifique « Sleep » permet de se synchroniser sur ce même signal d'interruption. L'instruction « Sleep » met en veille le microprocesseur afin qu'il ne consomme pas, celui-ci redémarra à l'arrivée d'un *un* sur le signal d'interruption INT. Comme pour les liens séries présentés ci-dessus, cette instruction Sleep montre comment la synchronisation logicielle est gérée par le matériel. Ici, le programmeur a l'avantage de ne pas avoir à gérer les problèmes de gestion de l'alimentation au sein même du programme (redémarrage d'horloge, etc…), c'est très simplement implémenté par le matériel.

Le mécanisme d'interruption et d'exécution de l'instruction sleep est géré par le décodeur d'Aspro (voir paragraphe 5.5.1).

4.2.2.5. Reset, mode de boot

Le signal de reset est actif bas, il initialise tous les états internes du processeur (voir paragraphe 3.1.2 pour l'implémentation du reset dans la logique QDI). Une hypothèse de délai est nécessaire sur ce signal car son effet n'est pas observable (Il serait possible d'observer l'état d'initialisation du processeur mais au coût d'un arbre d'acquittement global sur l'ensemble de la logique du processeur, ce qui serait extrêmement coûteux en matériel). En pratique, vue de l'extérieur, l'hypothèse de délai est simple à vérifier - le temps de traversée du Reset sur la carte et les plots du circuit étant nettement supérieur à ce qui se passe au temps de reset à l'intérieur du circuit -.

Lorsque le signal Reset est relâché, le processeur démarre le programme de boot. Celui-ci consiste simplement à venir chercher la première instruction en mémoire externe sur le port A à l'adresse $FFF0. Pour cette première version d'Aspro, il n'était pas prévu d'intégrer de ROM sur le circuit, il n'était donc pas possible d'inclure de procédure de boot plus complexe.

4.2.2.6. Système d'alimentation

Afin de tirer parti des potentiels de robustesse et de gestion d'alimentation des circuits asynchrones, le processeur intègre trois réseaux d'alimentations distincts. Le premier Vdd-Sys permet d'alimenter la couronne de plots du circuit, il correspond à la tension d'alimentation nominale de la technologie (2.5V en HCMOS7). Le cœur logique et les mémoires ont des alimentations distinctes respectivement Vdd-Core et Vdd-Mem afin de pouvoir diminuer à volonté l'alimentation et donc la consommation du cœur et des mémoires. En effet, les mémoires seront réalisées avec des technologies standard (voir paragraphe 6.1), elles ne pourront pas être alimentées aussi bas que le cœur. Des cellules spécifiques (cellule CV_UP, chapitre 3) pour effectuer les conversions de niveaux de tensions sont insérées sur les signaux aux interfaces. Les contraintes sur les trois réseaux d'alimentation sont donc les suivantes :

$$\text{Vdd-Core} \leq \text{Vdd-Mem} \quad \text{et} \quad \text{Vdd-Core} \leq \text{Vdd-Sys}.$$

L'objectif est que le processeur soit fonctionnel dans une très grande plage de fonctionnement, typiquement de moins de 1V jusqu'à 2.5V, ceci pour disposer d'un grand éventail de points de fonctionnement vitesse/consommation. Le but est de démontrer que les circuits asynchrones quasi-insensibles aux délais sont parfaitement aptes à s'adapter aux conditions de fonctionnement suivant les traitements à effectuer [BERK 94].

4.2.3. Une architecture personnalisable

De manière à montrer que les circuits asynchrones peuvent facilement s'adapter à une demande croissante en terme de vitesse et de fonctionnalités spécifiques suivant les applications, Aspro est conçu pour être aisément personnalisable et ce de deux manières. L'idée est donc de disposer d'un cœur de processeur (un bloc « IP » dit de propriété intellectuelle sous le format d'une netlist de cellules standard par exemple) et de pouvoir y rajouter à moindre coût des fonctions matérielles spécifiques. Ceci est rendu possible grâce aux propriétés de modularité des circuits asynchrones. Comparativement à une conception synchrone, il est ainsi possible de rajouter des fonctionnalités très hétérogènes (en terme de vitesse) par rapport au cœur du processeur, le fonctionnement étant garanti par simple respect des protocoles aux interfaces.

Ainsi, au niveau méthode et flot de conception, un architecte pourrait concevoir rapidement une nouvelle version d'Aspro en ré-assemblant (placement-routage) le cœur du processeur Aspro existant avec les unités spécifiques adaptées à ses nouveaux besoins.

- **Des périphériques dédiés**

La première manière de personnaliser Aspro est d'intégrer des unités périphériques dédiées. Un espace adressable de 256 mots de 16 bits est réservé pour les périphériques. Ceux-ci sont accédés par des instructions spécifiques (Movep regi, @periph / Movep @periph, regi).

Parmi les 256 adresses disponibles, seules cinq adresses sont réservées dans cette première version d'Aspro, elles implémentent les périphériques présentés dans le paragraphe 4.2.2 : les ports parallèles PI et PO et les quatre liens série. Ainsi des mémoires double ports, des timers synchrones, des unités PWM, des interfaces dédiés peuvent être rajoutés comme autant de périphériques particuliers.

- **Une unité utilisateur**

La seconde manière de personnaliser Aspro est plus générale. Le concepteur peut intégrer une unité spécifique au sein même du cœur du processeur tout en rajoutant au jeu d'instructions les instructions spécifiques qui contrôlent cette unité utilisateur. Le nombre d'instructions utilisateurs réservées dans le jeu d'instructions est de 64 instructions. Ce jeu d'instructions utilisateur peut bien sûr spécifier aussi bien des codes opératoires que des ressources mémoires locales à cette unité.

L'architecture du microprocesseur est donc conçu pour pouvoir inclure une unité quelconque qui soit directement connecté : i) avec le décodeur d'instruction, ii) avec le banc de registres aussi bien en lecture qu'en écriture. Le décodeur est donc conçu pour détecter une instruction utilisateur, l'envoyer à l'unité correspondante et effectuer une opération entre zéro et trois opérandes (pas d'immédiat néanmoins).

Ce mécanisme est très peu coûteux, le matériel spécifique pour supporter cette possibilité de personnalisation du cœur du processeur est négligeable. Il permet ainsi de rajouter de manière efficace des fonctionnalités matérielles qui auraient été coûteuses à programmer en logiciel.

4.2.4. Jeu d'instructions

Le jeu d'instructions est un jeu d'instructions issu des architectures RISC. Les trois modes d'adressage sont : immédiat, indexé avec déplacement et relatif à pc. Les instructions sont des

instructions trois opérandes sur les registres avec possibilité d'utilisation d'immédiats (signé ou non signé suivant les instructions).

Le jeu d'instruction est tout d'abord présenté au niveau fonctionnel en différents groupes d'instructions : UAL, accès mémoires, instructions de contrôle et instructions utilisateurs. Le problème de l'encodage du jeu d'instructions est présenté dans le paragraphe 4.2.5.

4.2.4.1. Instructions Arithmétiques et Logiques.

Le jeu d'instructions pour les opérations d'UAL (Table 4-1) inclue les opérations classiques d'arithmétique, de logique et de décalage/rotation.

Néanmoins, certaines opérations peu coûteuses en matériel ont été rajoutées afin d'accélérer l'exécution des programmes (et réduire la consommation). Les opérations Min/Max signées ou non signées sont disponibles. Elles sont très utilisées dans des applications de traitement d'image, comme par exemple les algorithmes en morphologie mathématique [ROBI 97]. Une instruction de « bit reverse » a été rajoutée pour effectuer des opérations de FFT. C'est une instruction très facile à implémenter en matériel (il suffit d'échanger les bits de poids fort et de poids faible), alors qu'elle est relativement coûteuse à réaliser avec des instructions de décalage/masquage classique.

Add, Sub, Neg	Addition, Soustraction, Négation (toujours signé)
Min, Minu, Max, Maxu, Slt, Sltu	Min, Max, Comparaison (signé ou non)
And, Or, Xor, Not	Opérations logiques au bit près
Shl, Shr, Shrs	Décalage à gauche, à droite (signé ou non)
Bitr, Rotl, Rotr	Bit reverse, rotation à gauche ou à droite

Table 4-1: Instructions UAL

Les instructions de comparaisons (Slt : set if lower than / Sltu : set if lower than unsigned) permettent d'effectuer les tests de retenue et de débordement. Comme aucun registre d'état n'est implémenté dans Aspro, c'est le strict minimum pour que l'utilisateur puisse effectuer de l'arithmétique exacte et en particulier faire de l'arithmétique sur 32 ou 64 bits avec le jeu d'instruction 16 bits proposé.

Les instructions UAL sont toutes des instructions 3 opérandes, sauf les instructions Neg, Not et Bitr. Pour les instructions de décalage et de rotation, la valeur de décalage est spécifié par les 4 bits de poids faible du deuxième opérande, ce qui permet de faire des décalages/rotations gauche/droite entre 0 et 15. Pour l'ensemble des instructions UAL, des immédiats sur 4 bits peuvent être spécifiés en tant que deuxième opérande (ils sont signés uniquement pour les instructions Min, Max, Slt). C'est en général suffisant pour encoder des immédiats de faible valeur (0,1,-1).

En ce qui concerne les opérations de multiplication ou de division entières, elles ne sont pas fournies dans le jeu d'instructions de base. Elles pourraient être néanmoins rajoutées au jeu d'instructions en tant qu'instructions utilisateurs si cela était nécessaire.

4.2.4.2. Instructions Load / Store

Les instructions du type Load/Store (Table 4-2) permettent d'effectuer des chargements / rangements dans les différentes zones mémoires : mémoire données, mémoire programme, périphériques.

Divers	
Ldi regi, #imm	regi <= #imm
Lra regi, label:	regi <= pc + offset-to-label
Accès mémoire données	
Ld[b/w] regi, disp(regj)	regi <= Mem[regj+disp]
St[b/w] regi, disp(regj)	Mem[regj+disp] <= regi
Accès mémoire programme	
Ldpg regi	LdSt_Reg<= Mem_prog[regi]
Stpg regi	Mem_prog[regi] <= LdSt_Reg
Move LdSt_Reg[H/L], regi	LdSt_Reg[H/L] <= regi
Move regi, LdSt_Reg[H/L]	regi <= LdSt_Reg[H/L]
Accès périphériques	
Movep regi, @periph	regi <= Peripherals[@periph]
Movep @periph, regi	Peripherals[@periph] <= regi

Table 4-2 : Instructions d'accès mémoires

- L'instruction Ldi permet d'initialiser un registre avec un immédiat sur 16 bits.

- Accès à la mémoire données :

Les instructions habituelles de chargement (load) et de sauvegarde (store) permettent d'accéder à la mémoire par octet ou par mot (la mémoire données est adressée par octet). Pour les accès par octet, les bits de poids forts sont ignorés. Dans le cas d'un accès par mot, l'adresse paire immédiatement inférieure est utilisée (la mémoire est utilisée avec les poids forts aux adresses impaires, ce qui correspond à la convention dite « *little endian* »).

Pour ces instructions load/store, le mode d'adressage par défaut est indexé avec déplacement, le déplacement étant une valeur signée sur 8 bits (déplacement de –128 à +127). Ce mode d'adressage indexé est très souvent utilisé pour accéder à des éléments dans des structures de données comme les tableaux, les champs, les piles programmes utilisés dans les langages de haut niveaux.

- Accès à la mémoire programme.

L'accès à la mémoire programme est rendu nécessaire pour pouvoir écrire du code relogeable qui puisse être dynamiquement chargé en mémoire programme sans être ré-assemblé. Ceci nécessite de pouvoir référencer les données du programme (comme des appels de routine) relativement au compteur de programme et non par adresse absolue. Pour ce faire, une instruction Lra (Load PC Relative Adress) est la seule instruction qui nécessite d'être introduite. Elle calcule une adresse absolue en additionnant un offset et le compteur de programme courant, cet offset étant calculé une fois pour toute à l'édition de lien. Par exemple, voici comment peut s'écrire une instruction *switch* (table de saut en langage C) :

```
Table: bra   S1:                  ; Table de saut. Chaque tronçon
       bra   S2:                  ; de code Si finit par un jmp reg15
       bra   S3:
             …
Entry: Lra   reg0, Table:         ; calcule l'adresse absolue de la table
       Add   reg0, reg0, reg10    ; additionne l'index du switch contenu dans reg10
       Jsr   reg15, reg0          ; branche dans la table et sauve le pc de retour
```

Cette portion de code ne contient ainsi aucune adresse de branchement absolue.

En plus de pouvoir calculer des adresses programmes absolues pendant l'exécution, il est nécessaire de pouvoir accéder à la mémoire programme en lecture ou en écriture. Ceci permet d'écrire du code auto-modifiable, de charger une procédure de boot ou tout simplement de charger des données depuis la mémoire externe. Comme la mémoire programme fait 24 bits de large (taille instruction) et que le reste du processeur est sur 16 bits, une procédure de chargement en plusieurs temps est prévue afin de simplifier le matériel de l'interface d'accès à la mémoire programme. Un registre spécifique de 24 bits dénommé LdSt_Reg est chargé depuis les registres banalisés en deux temps (16 bits de poids faible, 8 bits de poids fort) puis son contenu est transféré en une seule fois vers la mémoire programme (et réciproquement). Ainsi quatre instructions sont introduites : ldpg et stpg pour échanger le contenu de LdSt_Reg avec la mémoire et Move LdSt_Reg_[H/L], regi pour échanger le contenu de LdSt_Reg (MSBs ou LSBs) avec les registres. Dans une procédure de boot, ceci permet par exemple de transférer facilement des instructions entre la mémoire programme externe et interne, le registre LdSt_Reg faisant tampon entre une lecture en externe et une écriture en interne.

- L'accès aux périphériques se fait avec des instructions de déplacement spécifiques. L'espace adressable pour les périphériques est de 256 adresses. Voir paragraphe 4.2.2 pour les périphériques disponibles.

4.2.4.3. Instructions de branchement, de contrôle

Les différentes instructions de branchement (Table 4-3) permettent d'effectuer des branchements conditionnels ou non, des branchements relatifs ou absolus, ainsi que des sauts à des sous-routines avec sauvegarde de l'adresse de retour. Les instructions de contrôle concernent principalement le traitement des interruptions.

Branchement conditionnel	
DBcc regi, regj / #imm, label:	Delayed branch, cc=(eq/ne/lt/ltu)
Bcc regi, regj / #imm, label:	Not-Delayed branch, cc=(eq/ne/lt/ltu)
Branchement non conditionnel	
Dbra label:	Delayed branch always : pc <= pc+offset
Djmp regi	Delayed jump : pc <= regi
Dbsr regi, label:	Delayed branch to subroutine : regi <= pc ; pc <= pc+offset
Djsr regi, regj	Delayed jump to subroutine : regi <= pc ; pc <= regj
Contrôle	
Nop	No operation
Dint	Disable Interrupt : int-flag <= 0
Eint	Enable Interrupt : int-flag <= 1
Drti	Delayed return from interrupt
Sleep	Sleep : mode veille, synchronisation sur signal interruption

Table 4-3 : Instructions de branchement et contrôle.

- Branchements, sauts

Les branchements, qu'ils soient conditionnels ou non, sont toujours coûteux à implémenter. En effet, pour une architecture pipelinée du type Fetch/Decod/Execute/WriteBack (que ce soit

une implémentation synchrone ou asynchrone), les instructions de branchement rompent le schéma régulier d'exécution du pipeline car elles nécessitent d'attendre le résultat et l'adresse de branchement avant de pouvoir continuer. Ceci impose de suspendre le pipeline, voire de supprimer les instructions qui ne doivent pas être exécutées si le branchement est pris.

Une solution consiste à introduire des branchements et sauts retardés [HENN 96]. Supposons que le pipeline soit rempli de N instructions, le branchement ou saut aura lieu N instructions plus tard, les N-1 instructions en cours dans le pipeline étant alors toujours exécutées. C'est donc à la charge du compilateur de remplir les « delay-slots » (nom donné aux instructions qui suivent le branchement retardé) avec des instructions utiles. Pour le processeur Aspro, il a été défini un schéma d'itération à deux delay-slots (voir paragraphe 4.3, il y a toujours trois instructions dans la boucle de « fetch »). Les branchements retardés ont donc lieu deux instructions plus tard. Par exemple, cette optimisation à la compilation permet de charger les paramètres d'entrées lors d'un appel de procédure :

```
Djmp   reg15, reg0          ; saut retardé : branche en reg0, sauve pc dans reg15
Ldw    reg1, 0(reg10)       ; charge reg1 et reg2 avec des valeurs en pile
Ldw    reg2, 2(reg10)
```

Toutes les instructions de branchement non conditionnels sont retardées : Djmp, Djsr, Dbsr, Dbra. Dans le cas où aucune instruction utile ne peut être exécutée dans les delay slots d'un branchement, il est nécessaire d'ajouter des instructions Nops. Les valeurs de déplacement pour Dbsr et Dbra sont codés sur 16 bits, ce qui autorise des branchements longs.

Il n'y a pas d'instruction spécifique pour exécuter un retour de procédure. Lors d'un appel de procédure (Djsr ou Dbsr), le compteur programme (pc) est sauvé dans un registre regi, le retour de procédure est donc exécuté par un "Djmp regi".

Comme il n'y a pas de registre d'état, les branchements conditionnels sont directement effectués par des comparaisons entre registres ou entre un registre et un immédiat. Quatre types de branchement sont disponibles : égal, non-égal, plus petit, plus petit signé. Les immédiats sont codés sur 4 bits ce qui permet les tests de comparaisons courants (zéro, un, …). La valeur de déplacement pour les branchements conditionnels est codée sur 10 bits signés, ce qui autorise des branchements de -512 à 511. Afin de ne pas pénaliser la densité du code avec des Nop superflus, comme les instructions de branchements conditionnels imposent souvent des dépendances sur les valeurs de registres, ces instructions peuvent être retardés ou non, il est donc défini les deux instructions de branchement DBcc et Bcc.

- Instructions de contrôle

L'instruction Nop ne fait rien. Comme mentionné ci-dessus, les Nops sont utilisés pour remplir les delay-slots non utilisés après une instruction de branchement. Le départ en interruption et le fonctionnement de l'instruction Sleep ont été décrits dans le paragraphe 4.2.2. Les instructions Dint et Eint permettent respectivement en écrivant le flag d'interruption d'interdire et d'autoriser les interruptions. L'instruction Drti permet de revenir d'interruption (on suppose que l'adresse de retour est positionnée dans le registre 15) et autorise de nouveau les interruptions.

4.2.4.4. Instructions utilisateurs

Il y a 64 instructions utilisateurs disponibles (6 bits de code utilisateur). Au niveau de l'utilisation des registres, ces instructions peuvent avoir l'un des six formats issus des

combinaisons suivantes : deux, un ou aucun opérandes sources (pas d'immédiat)[4], un ou aucun registre destination. Le code utilisateur est totalement libre, il peut en particulier être utilisé pour spécifier un numéro de registre local au sein du code instruction.

Afin d'obtenir des fonctionnalités de type DSP, l'unité utilisateur proposée pour Aspro intègre une unité de multiplication accumulation. L'unité dispose d'un multiplieur 16 par 16 avec accumulation sur 40 bits, la multiplication étant signée ou non signée et de quatre registres 40 bits d'accumulation distincts (acc0, …, acc3) qui peuvent être sauvés dans les registres avec différents modes : entier ou virgule fixe, test de saturation, calcul d'arrondi et par tronçon : bits de poids faible (15..0), bit médium (31..16), bits de poids forts (39..32).

La Table 4-4 donne le jeu d'instructions de l'unité utilisateur. Le choix de représentation entier ou virgule fixe est seulement spécifié lorsque le contenu d'un accumulateur est sauvegardé, la représentation n'a pas de sens lors de la multiplication accumulation.

Ce jeu d'instructions permet d'écrire facilement une routine de filtrage de type FIR (voir paragraphe 5.3) avec quelques lignes d'assembleur. Bien sûr, on ne pourra pas comparer les performances d'Aspro à une véritable architecture DSP. En effet, pour de meilleures performances en vitesse, les architectures de processeur dédié au traitement de signal peuvent exécuter en une seule instruction la boucle FIR grâce à un compteur de boucle intégré et des mémoires double ports. Ce jeu d'instructions utilisateur permet néanmoins d'offrir à Aspro une unité de multiplication entière.

Mpy[s/u] acci, regi, regj	Multiply (signed/unsigned) acci <= regi*regj
Macc[s/u] acci, regi, regj	Multiply-Accumulate acci <= acci + regi*regj
Sint.[l/m/h] regj, acci	Save integer acci [low, medium, high] in regj
Sints.[l/m] regj, acci	Save and saturate integer [low, medium]
Sintr.m regj, acci	Save and round integer medium
Sfp.[l/m/h] regj, acci	Save fixed point acci [low, medium, high] in regj
Sfps.[l/m] regj, acci	Save and saturate fixed point [low, medium]
Sfpr.m regj, acci	Save and round fixed point medium

Table 4-4 : Instructions unité utilisateur

4.2.5. Encodage du jeu d'instructions

Avec une problématique de type RISC, les contraintes de performances imposent de simplifier au maximum le format des instructions. Les instructions sont toutes codées sur une longueur constante de 24 bits. Pour chaque instruction, un unique mot 24-bit est lu en mémoire programme, ceci permet de pipeliner facilement lecture et décodage des instructions. Les différents champs tel que les codes instructions, les numéros de registres, sont alignés entre instructions afin que pour toute instruction, le décodeur puisse de manière la plus systématique générer ces champs vers les différentes unités. Ceci optimise à la fois vitesse, consommation et matériel. En contrepartie, il y a perte de place dans le code si des champs ne sont pas utilisés pour une instruction donnée.

De plus, il est parfois nécessaire d'aider le matériel à prendre des décisions en ajoutant dans le code instruction des informations pré-calculées à l'assemblage/compilation. Ce sont typiquement des informations à propos de l'utilisation des registres. Si un registre destination est égal au registre source, le matériel doit tout d'abord lire le registre puis le réserver pour

[4] Le décodeur ne peut pas définir l'extension de signe à réaliser sur un éventuel immédiat de 4 bits.

écriture. Si les deux opérandes sources sont égaux, le banc de registre doit envoyer le contenu d'un même registre vers les deux bus opérandes. Afin de ne pas imposer au matériel de comparer les numéros de registre pour savoir combien de registres et lesquels sont à lire et/ou écrire, quatre bits sont ajoutés dans le codage de l'instruction. Ces 4 bits dénommés Src-Op(3..0) spécifient pour chaque opérande (respectivement les deux registres sources, le registre destination, et l'immédiat) si les numéros de registre et la valeur immédiate sont des valeurs valides ou non. Ces bits sont directement envoyés au banc de registres pour effectuer les lectures et écritures nécessaires ainsi qu'à l'interface bus afin d'envoyer les opérandes et les immédiats vers les bonnes unités. Ce mécanisme est coûteux en terme de densité de codage mais facilite le décodage et permet de bonnes performances en vitesse, notamment en régularisant le décodeur et principalement le banc de registres (cf. paragraphe 5.3).

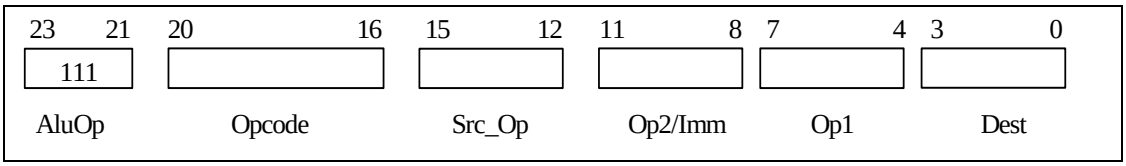


Figure 4-5 : Encodage des instructions Alu

L'encodage du jeu d'instruction d'Aspro n'est pas présenté en détail ici, il fait l'objet d'une documentation complète. A titre d'exemple, la

Figure 4-5 présente l'encodage des instructions arithmétique et logique. Op1, Op2, Dest sont les numéros de registres sources et destination, Imm est un immédiat encodé sur 4 bits (signé ou non), Src-Op les 4 bits précisant si ces numéros de registre/immédiat sont valides et Opcode le code de l'instruction Alu. Le premier champ spécifie que c'est une instruction de type Alu. Ainsi, il suffit de ces trois premiers bits au décodeur pour décider de l'envoi de l'instruction.

4.3. Architecture du processeur Aspro

L'objet de ce paragraphe est de présenter la vue d'ensemble de l'architecture du processeur [RENA 98b]. Cette architecture est obtenue par décomposition en processus d'un premier modèle CHP de haut niveau, fonctionnel et séquentiel, du processeur. Ce premier modèle CHP n'est pas décrit précisément ici, il représente un unique processus CHP d'environ 300 lignes de code. Ce modèle fonctionnel effectue en séquence la lecture d'une instruction, le décodage de celle-ci, la lecture des opérandes, l'exécution de l'instruction avec l'appel d'une fonction de haut niveau, puis enfin l'écriture du résultat dans les registres. A ce niveau, le modèle pourrait même être écrit en un langage quelconque (langage C par exemple) puisqu'il se suffit à lui même, il ne communique pas avec d'autres éléments, il est constitué d'une « simple » boucle infinie et de variables locales (y compris les ressources mémoires).

Le principe de la décomposition consiste alors à extraire de ce modèle séquentiel les différentes ressources de calcul, c'est à dire les unités d'exécution, et les ressources de mémorisation, c'est à dire les registres, mémoires, et périphériques. Vu les différentes expériences dans la conception de circuits asynchrones, il est nécessaire de minimiser le plus possible les synchronisations, et ceci à tous les niveaux de l'architecture du circuit [ROBI 97],

[RENA 97]. Le relâchement des synchronisations est en faveur de la vitesse. Le but est donc d'étudier les dépendances de données et les principales boucles de synchronisation du processeur. La justification de ces boucles provient évidemment de la spécification fonctionnelle du processeur, c'est à dire le jeu d'instructions.

4.3.1. Introduction à la notion de pipeline asynchrone

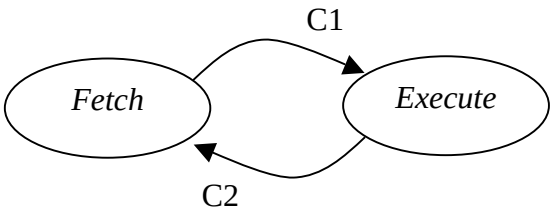
Nous allons tout d'abord montrer que la décomposition basée sur l'utilisation des processus communicant est source de parallélisme, et que celle-ci peut être plus ou moins exploitée suivant les synchronisation minimale à implémenter. Ce parallélisme potentiel correspond à la notion classique de pipeline. Ces concepts sont relativement classiques en asynchrone, ils ont déjà été étudiés par le groupe de Caltech dans [MART 93], [BURN 91].

Prenons l'exemple succinct du modèle séquentiel du processeur avec une fonction de lecture/décodage instruction (*Fetch*) et une fonction d'exécution (*Execute*) :

```
*[ Fetch ; Execute ]
```

Le but est d'isoler ces deux parties afin de créer deux blocs architecturaux distincts. Il est alors nécessaire d'introduire deux canaux de communication C1 et C2 comme suit :

```
*[ Fetch ; C1!instr ; C2? ]
*[ C1?instr ; Execute ; C2! ]
```



Le canal C1 permet d'envoyer l'instruction que le bloc de *Fetch* vient de lire et décoder. Le canal C2 précise que le bloc *Execute* a fini de l'exécuter. Dans ce nouveau schéma, la décomposition a créé deux entités distinctes mais n'a pas introduit de parallélisme puisque le modèle s'exécute toujours dans la séquence : *Fetch* ; C1 ; *Execute* ; C2. On a donc rien gagner en performance. Cependant, si le canal C2 ne porte qu'une information de synchronisation (*Execute* dit : « j'ai fini ») et que le bloc de *Fetch* sait toujours ce qu'il doit faire (lire l'instruction à l'adresse suivante), il n'y a donc pas de dépendance fonctionnelle, ce canal C2 est superflu, il peut être supprimé. On obtient alors :

```
*[ Fetch ; C1!instr ]
*[ C1?instr ; Execute ]
```

Dans ce cas, la décomposition a introduit du parallélisme car *Fetch* et *Execute* peuvent travailler concurremment. Ainsi, le bloc *Fetch* peut lire/décoder une première instruction Instr1, l'envoyer à *Execute* via le canal C1 et commencer à lire l'instruction suivante Instr2 sans attendre la terminaison de l'exécution de Instr1. C'est seulement à l'envoi de Instr2 que *Fetch* risque d'être suspendu si *Execute* n'est pas prêt à prendre l'instruction suivante car n'a pas fini l'exécution précédente de Instr1. Réciproquement, si *Execute* est plus rapide, il doit attendre l'instruction Instr2 pour pouvoir continuer à travailler. On a donc un système matériel qui lit/exécute deux instructions en parallèle. Grâce au canal de synchronisation C1, le séquencement est préservé entre instructions quelque soit les temps de lecture/décodage dans le bloc *Fetch* et le temps d'exécution dans *Execute*. Le système est donc pipeliné dans le sens classique où il y a recouvrement des actions entre les différentes tâches matérielles, sur des données distinctes. Ceci est bien sûr en faveur des performances.

Néanmoins, un système tel qu'un microprocesseur n'est bien sûr pas aussi simple. Il faut en particulier gérer les branchements qui perturbent le flot continu d'instructions. Dans ce cas, il est nécessaire et suffisant d'implémenter une synchronisation conditionnelle. Ainsi, le canal C2 ne peut pas être supprimé, il va indiquer que le bloc *Execute* a fini de calculer le branchement et si ce branchement est pris ou non. Cette synchronisation est pertinente seulement dans ce cas, toute autre instruction n'ayant pas besoin comme précédemment d'indiquer sa terminaison puisqu'il suffit d'incrémenter le compteur instruction. On écrit :

```
*[ Fetch; C1!instr; [ instr=branch => C2?gonogo;
                        [ gonogo=1 => AD?pc        -- Update PC
                        @ gonogo=0 => pc:=pc+1 ]
                     @ instr!=branch => pc:=pc+1
 ]                   ]
*[ C1?instr ;     [ instr=branch => Compute Branch & new address;
                       C2!gonogo ; [ gonogo=1 => AD!newPC ]
                  @ instr!=branch => Execute
 ]                ]
```

Ainsi, le bloc de *Fetch* lit, décode l'instruction, et l'envoi au bloc *Execute*. Si l'instruction est un branchement, *Fetch* attend le résultat *gonogo* du branchement sur le canal C2 et suivant ce branchement, il met à jour le compteur programme (PC) ou l'incrémente. Sinon, il continue sans se synchroniser comme dans l'exemple précédent. Réciproquement, le bloc *Execute* réalise l'exécution et indique à *Fetch* dans le cas d'un branchement quel en est le résultat. Un canal supplémentaire AD a été introduit afin d'envoyer l'adresse de branchement[5] dans le cas où le branchement est pris.

Au niveau performance, on observe alors que dans les cas courants, le pipeline se remplit comme dans l'exemple précédent, au contraire si il y a branchement, le bloc de Fetch doit se synchroniser sur Execute avant de pouvoir continuer. Cette synchronisation conditionnelle permet alors de gagner en performance dans les cas favorables et suspend le pipeline seulement quand nécessaire. (Un point important à vérifier correspond au nombre d'instructions présentes dans le pipeline et comment les supprimer dans le cas du branchement, ceci sera présenté en détail dans le paragraphe suivant). Le pipeline obtenu est alors élastique. Son fonctionnement s'adapte aux données qu'il manipule. La boucle principale *Fetch+Execute* du processeur, celle qui représente à priori la synchronisation la plus forte, est donc seulement fermée dans les cas de branchement, elle reste ouverte dans les cas contraire : le chemin de donnée se trouve la plupart du temps indépendant du reste du contrôle.

Il est surtout intéressant de noter que ce mécanisme implémente la fonction de branchement mais ne tient pas compte des performances relatives entre les blocs, ni de la profondeur du pipeline de chaque bloc (à quel cycle le branchement est présent, etc.). L'unité d'exécution peut être à son tour décomposée autant de fois que nécessaire, et donc pipelinée avec une profondeur de pipeline inconnue, sans pour autant avoir à modifier la fonctionnalité du *Fetch* : celui-ci attend le résultat de branchement sur C2 quelque soit ce qui se passe dans *Execute*. Seule la fonctionnalité est modélisée / implémentée ici : « si branchement, mettre à

[5] Ce type de branchement correspondrait à un branchement absolu conditionnel.

jour PC ». La modélisation par canal permet bien de dé-corréler implémentation de la fonction et optimisation des performances du système (cf. chapitre 6).

Par cet exemple, nous avons montré que la décomposition par création de processus communicants est source de parallélisme. Les blocs étant concurrents par nature, le parallélisme sera effectif si plusieurs informations (donnés / instructions) circulent à la fois dans l'architecture et que ces blocs imposent le moins possible de synchronisation entre ces informations. Afin d'optimiser globalement les performances, le travail de l'architecte correspond donc à étudier les dépendances de données dans la spécification originale et à les implémenter sous forme de canal de communication avec des synchronisations conditionnelles.

4.3.2. Architecture du processeur

Avant de donner une description précise des principales boucles de synchronisation du processeur, nous introduisons les grandes lignes de son architecture en expliquant le schéma d'exécution d'une séquence d'instructions.

Le processeur est un processeur scalaire avec émission des instructions dans l'ordre. Ce schéma est choisi pour sa simplicité. Cependant, un schéma d'exécution avec terminaison des instructions dans le désordre est adopté car est parfaitement adapté au mode de synchronisation des circuits asynchrones [RICH 96]. La Figure 4-6 présente le schéma bloc de l'architecture du processeur.

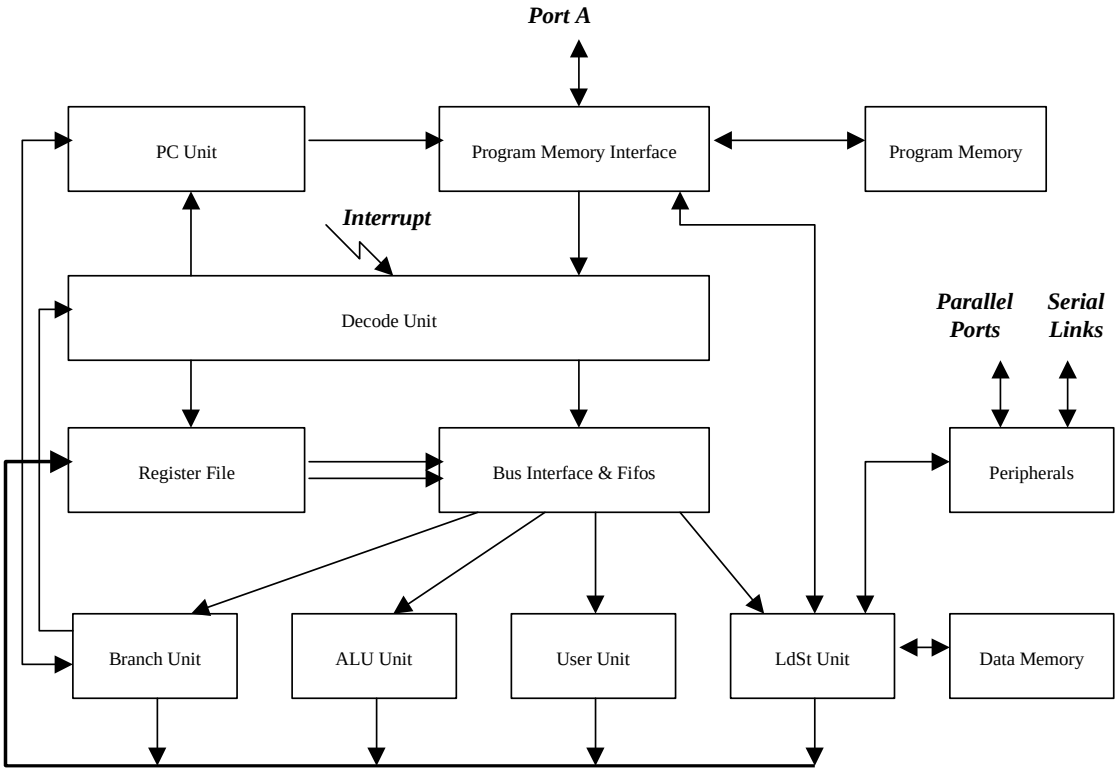


Figure 4-6 : Architecture du processeur Aspro

Les instructions sont lues en mémoire les unes après les autres puis décodées. Le décodeur élabore des commandes qui sont envoyées aux unités du chemin de données. Celui-ci est constitué de quatre unités d'exécution indépendantes : l'unité arithmétique et logique, l'unité utilisateur, l'unité load-store, et l'unité de branchement. Toutes ces unités sont désynchronisées ce qui implique qu'elles peuvent travailler en parallèle indépendamment les unes des autres dès qu'elles disposent de leurs propres opérandes. Pendant qu'une unité exécute une instruction avec son propre débit et sa propre latence, une autre unité peut être nourrie avec l'instruction suivante. Quand un résultat est disponible à la sortie d'une unité, il peut être ensuite écrit dans le banc de registres. L'architecture supporte un mécanisme d'écriture dans le désordre ce qui permet aux différentes unités d'écrire leur résultat au plus tôt et indépendamment les unes des autres. Aucun arbitrage n'est nécessaire, le fonctionnement est déterministe grâce à l'utilisation d'une stratégie de réservation des registres. La ré-écriture dans le banc de registres est ainsi faite quelque soit l'ordre et les temps d'arrivée du résultat des instructions. Ce mécanisme permet de tirer parti naturellement du parallélisme niveau instruction : les instructions peuvent s'exécuter en parallèle, se recouvrir dans le temps et se terminer dans le désordre tant qu'un aléa de données n'obligent pas une synchronisation « lecture-après-écriture » dans le banc de registres. L'architecture présente donc un pipeline d'exécution extrêmement élastique : les instructions s'exécutent en parallèle tant qu'elles le peuvent, suivant la disponibilité des opérandes et des unités ou se synchronisent à la moindre dépendance entre instruction : opérande non prêt, unité non-prête. Avec une optimisation correcte du code assembleur à la compilation, l'utilisateur peut pleinement tirer parti des performances de ce pipeline élastique (voir paragraphe 4.4).

4.3.2.1. Boucle de Fetch-Decode

La boucle Fetch-Decode est constituée de l'unité PC, de l'interface mémoire, de la mémoire programme et enfin du décodeur d'instructions (Figure 4-6). Cette boucle est utilisée pour l'exécution de toutes les instructions. L'unité PC qui contient le compteur programme (Program Counter) envoie tout d'abord l'adresse de l'instruction à la mémoire programme. L'accès mémoire est effectué via l'interface mémoire qui décide d'où obtenir l'instruction, soit en mémoire programme interne, soit en externe via le port A. Quand le mot instruction est obtenu, il est envoyé par l'interface au décodeur. Celui ci décode l'instruction et génère des commandes vers la boucle du chemin de données : contrôle du banc de registre, de l'interface bus, et opcode des unités. La synchronisation du Fetch avec le chemin de données existe donc pour toutes les instructions sauf pour Nop, Dint, et Eint. Ainsi, le décodeur nourri le chemin de données avec des instructions décodées, sauf lorsqu'il n'y a rien à faire (Dint et Eint modifient le flag d'interruption, opérations effectuées au sein du décodeur).

Afin de clore la boucle Fetch-Decode, le décodeur termine l'envoi de l'instruction en indiquant à l'unité PC quel est le type d'instruction qui vient d'être émise. Comme présenté dans le paragraphe 4.3.1, plusieurs types d'instructions sont à considérer. Si l'instruction est un branchement, l'unité PC sait alors qu'elle doit échanger des informations avec l'unité de branchement : soit attendre le résultat de branchement, soit obtenir une adresse de branchement, soit lui envoyer le PC pour le sauvegarder dans le banc de registre. PC est alors mis à jour suivant ce branchement. Au contraire, si le branchement n'est pas pris, ou que l'instruction ne correspond pas à un branchement, l'unité PC doit simplement incrémenter le compteur programme, puis recommencer : envoie de l'adresse à la mémoire, etc.

Pour optimiser les performances, cette boucle Fetch-Decode est bien sûr pipelinée. Le niveau de pipeline doit permettre de garantir un débit local élevé sur chacun de ses blocs. En

particulier, ceci a imposé de concevoir une mémoire programme pipelinée (paragraphe 5.1). Cependant, afin d'obtenir un débit global élevé (cf. paragraphe 6.3 pour l'optimisation des anneaux asynchrones), il est nécessaire de tenir compte de la latence totale de la boucle de Fetch. Vu la latence importante de la mémoire, cumulée avec la latence des différentes unités (PC+interface+décodeur), il est nécessaire d'avoir trois instructions (trois jetons) circulant dans la boucle afin de garantir un débit global proche du débit local. Ce schéma à trois instructions correspond à un mécanisme à deux delay-slot. Ainsi, il y a toujours deux instructions qui suivent l'instruction courante : lorsque l'unité PC reçoit du décodeur le type de l'instruction émise, la lecture des deux instructions suivantes a déjà commencé en mémoire. L'architecture est conçue pour garantir la présence de trois instructions en continu dans cette boucle. Les instructions et requêtes mémoires se suivent dans la boucle à leur propre vitesse. Ceci permet de remplir le pipeline de cette boucle de manière optimale afin d'obtenir un débit maximum.

4.3.2.2. Boucle de branchement

Pour les branchements relatifs, la valeur d'offset est directement envoyée par le décodeur à l'unité PC. Ceci permet par exemple d'exécuter l'instruction Dbra au niveau du Fetch sans avoir à se synchroniser avec le chemin de données (tout comme un Nop). Cependant, pour les instructions de branchement qui utilisent des valeurs présentent dans les registres (Bcc, DBcc, Djmp, Dbsr, Djsr), la boucle de Fetch-Decode doit se synchroniser avec la boucle du chemin de données et échanger des informations via l'unité de branchement.

Pour les branchements non-conditionnels (Djmp, Dbsr, Djsr), des adresses sont échangées entre l'unité PC et l'unité de branchement. Pour les sauts (Djmp, Djsr), l'unité de branchement envoie le contenu d'un registre comme adresse de branchement. Réciproquement pour les appels de sous programme (Dbsr, Djsr), l'unité PC envoie la valeur du PC courant à l'unité de branchement pour le sauvegarder dans le banc de registre. Comme tous ces branchements sont des instructions retardées, le décodeur n'a rien de particulier à faire, seule l'unité PC est concernée afin d'obtenir l'adresse de branchement. En effet, les deux instructions qui sont présentes dans le pipeline de la boucle de Fetch, et qui correspondent aux deux delay slot, sont à exécuter.

Pour les branchements conditionnels (Bcc, DBcc), une information branchement-pris/non-pris (le *gonogo* de l'exemple paragraphe 4.3.1) est calculé dans l'unité de branchement. Ce résultat de branchement est envoyé à l'unité PC, ce qui lui permet de déterminer l'adresse suivante : soit pc+offset, soit pc+1. Si ce branchement est retardé (DBcc), le décodeur n'est pas concerné puisque les instructions qui suivent le branchement, qu'il soit pris ou non pris, sont à exécuter. Néanmoins, si ce branchement n'est pas retardé (Bcc), le résultat de branchement doit aussi être envoyé au décodeur. En cas de branchement, le décodeur doit alors supprimer les deux instructions présentes dans le pipeline de la boucle de Fetch-Decode. Dans ce cas, afin de maintenir le nombre d'instructions égal à trois dans la boucle, le décodeur demande à l'unité PC de continuer à incrémenter son compteur pour obtenir les instructions suivantes. Au contraire si le branchement n'est pas pris, le décodeur continue à exécuter normalement les instructions qui suivent.

Avec la présence de trois instructions dans la boucle de Fetch, le mécanisme de branchement retardé est donc particulièrement efficace, car n'impose pas de synchronisation supplémentaire entre le décodeur et le chemin de données. Dans le cas de branchements retardés, le décodeur peut continuer à émettre des instructions sans être interrompu, seule l'unité PC est concernée pour obtenir son adresse de branchement. Il y a ainsi une suspension momentanée du Fetch (le temps que l'unité PC obtienne son adresse), mais l'exécution des

instructions qui suivent peut continuer sans attendre. Enfin, il y aura annulation d'instruction seulement dans le cas où le branchement est pris et non-retardé. Dans ce cas, deux instructions sont perdues, le pipeline d'exécution est suspendu jusqu'à ce qu'on obtienne l'instruction qui suit le branchement. C'est donc à la charge du compilateur d'exploiter au mieux la possibilité des branchements retardés et d'introduire des instructions utiles dans les delay slots des branchements.

4.3.2.3. Chemin de données

Le chemin de données inclut le banc de registres, l'interface bus et les unités d'exécutions. Nous allons présenter le comportement du chemin de données quand il exécute des instructions qui ne sont ni des branchements (cf. ci-dessus), ni des accès à la mémoire programme (cf. ci-après).

La boucle du chemin de données commence lorsqu'une instruction décodée est envoyée au banc de registres et à l'interface bus. Ceci permet de demander au banc de registres de lire aucun, un ou deux registres, et de réserver ou pas un registre en écriture. Au même moment, l'interface bus reçoit du décodeur des valeurs d'immédiats et des valeurs d'opcode. Ces opcodes sont stockées dans quatre fifos connectées aux unités d'exécution. Comme il y a des chemins de latences différentes entre le décodeur et les unités, un chemin court pour les opcodes via l'interface bus et un chemin long pour les données via le banc de registres et l'interface bus, ces fifos permettent de compenser la latence du banc de registres et donc d'optimiser le débit vue de l'entrée des unités. Ces fifos permettent de plus de diminuer la dépendance entre le chemin de données et la mémoire programme car peuvent éviter à la boucle de Fetch de se trouver suspendue lorsqu'une séquence d'instructions congestionne le chemin de données. Le but est donc de pouvoir niveler le débit à la sortie du décodeur d'instructions, de manière indépendante du flux d'instructions et des dépendances existantes entre les registres.

De plus, le débit de l'interface bus est plus rapide que celui du décodeur ou du banc de registres. Ceci lui permet donc de distribuer au plus vite vers les unités les opérandes obtenus en sortie du banc de registres et les opcodes provenant du décodeur. De la même manière, l'opcode et l'immédiat sont envoyés directement à l'unité si aucun opérande n'est attendu du banc de registres pour cette instruction.

Le banc de registres est le cœur du chemin de données. Celui-ci est conçu pour pouvoir envoyer deux opérandes en parallèle aux unités via l'interface bus et pouvoir réserver un registre en bloquant tout autre accès en lecture ou écriture à ce même registre. Pour pouvoir écrire le résultat des quatre unités d'exécution de manière indépendante, il y a quatre bus d'écriture connectés au banc de registres, un par unité. Ces quatre bus ferment alors la boucle du chemin de données.

L'architecture du banc de registres est faite de manière à ce que la réservation en écriture d'un registre n'empêche pas la lecture ou l'écriture de registres différents. Ceci signifie que tant qu'un registre réservé n'est pas accédé (en lecture ou écriture), le banc de registre peut continuer à fournir des opérandes et écrire les résultats des autres unités. Ce comportement permet de supporter le mécanisme de terminaison des instructions dans le désordre. Ainsi les unités peuvent s'exécuter en parallèle, chacune avec leur propre débit et latence, avec leurs propres opérandes et écrire de manière indépendantes leurs résultats dans le banc de registres. Tant qu'il n'y a pas de conflit sur un nom de registre entre les instructions, il y a un parfait entrelacement entre les unités. Réciproquement, dans le cas où une instruction accède à un registre réservé, le banc de registre attend que le registre soit libéré par l'écriture

correspondante. Dans ce cas, le flot d'instruction est suspendu jusqu'à ce que la dépendance soit résolue. Ainsi, le programme peut se mettre naturellement en attente sur un périphérique jusqu'à ce que celui-ci réponde.

Contrairement à la boucle de Fetch, il faut noter que le nombre d'instructions dans le chemin de données n'est pas nécessairement constant. Suivant les instructions reçues, leur séquence, les unités concernées, leurs latences, le taux de remplissage du chemin de données est très variable. Si toutes les instructions sont indépendantes, suivant la latence des unités concernées, on peut avoir plus de trois instructions. Par exemple, dans le cas d'une suite de chargement mémoire, ces instructions ayant une forte latence s'accumulent dans l'unité load-store et la mémoire données avant d'écrire leurs résultats dans les registres. Réciproquement, si il y a suspension du pipeline car une écriture n'est pas terminée, ceci bloque l'envoi de toute nouvelle instruction, dans ce cas, il ne reste qu'une instruction en cours dans le chemin de données.

Le schéma d'exécution est donc extrêmement élastique, ceci permet d'exploiter le parallélisme niveau instruction. Tout ceci est implémenté grâce à du contrôle local : au sein même du banc de registres, dans l'interface bus et dans les unités. Dans ce type d'architecture, il n'est pas nécessaire d'avoir une vue globale de l'état de la machine pour décider de l'envoi d'une instruction : si les opérandes sont disponibles, ou si la ressource matérielle pour l'exécuter est libre. La sémantique des processus communicant garantit le séquencement si il y a conflit. Le concepteur n'a pas à décider de l'envoi et de la répartition des instructions dans les ressources disponibles suivant la séquence d'instructions, c'est le matériel qui décide de manière dynamique de l'envoi des instructions, sans avoir à exécuter un algorithme de réservation / envoi compliqué. L'utilisation du parallélisme instruction est donc naturel à partir du moment où il y a des ressources matérielles concurrentes pour l'exploiter.

La boucle du chemin de données est fortement pipelinée, que ce soit le banc de registres ou les unités d'exécutions. Cependant, la latence globale de la boucle est optimisée afin de ne pas trop pénaliser les performances de l'architecture lorsqu'une dépendance sur un registre force une synchronisation lecture/écriture dans le banc de registres. Cette latence varie bien sûr en fonction de l'unité concernée, du type d'instruction dans chaque unité et de la valeur des opérandes à traiter.

Les quatre unités d'exécution sont les suivantes. L'unité Alu traite les opérations arithmétiques et logiques. Elle est elle même composée de quatre unités distinctes : arithmétique, comparaison, logique et décalage. L'unité de branchement exécute les calculs de branchement, le chargement d'immédiat (Ldi) et le calcul d'adresse sur PC (Lra). L'unité utilisateur implémente une unité de multiplication-accumulation avec en interne quatre registres d'accumulation sur 40-bits. L'unité Load-Store effectue les différents accès mémoires : mémoires données (par octet ou par mot), mémoire programme et l'accès aux périphériques. Toutes ces unités sont conçues pour offrir des latences qui dépendent des instructions concernées et des données correspondantes. Par exemple dans l'unité de branchement, un chargement d'immédiat (Ldi) est bien sûr plus rapide qu'un calcul d'adresse sur PC (Lra). L'unité load-store est l'unité qui présente les plus fortes latences en raison des accès mémoires.

4.3.2.4. Synchronisation Load-Store / Mémoire programme

Pour les deux instructions ldpg et stpg (load program et store program), le chemin de données doit accéder à la mémoire programme et donc se synchroniser avec la boucle de

Fetch. Dans ce cas, l'interface mémoire programme doit répondre à une requête provenant de l'unité load-store. Il se pose alors un problème de conflit d'accès à la mémoire programme puisqu'il faut synchroniser les requêtes provenant de l'unité PC pour effectuer le Fetch et celle-ci provenant de l'unité load-store. Pour ce faire, l'unité PC est informée par le décodeur qu'une telle instruction est en cours d'exécution. Celle-ci demande alors à l'interface mémoire programme de tout d'abord satisfaire la requête provenant de Load-Store puis demande de lire l'instruction suivante. Ce mécanisme correspond donc à l'insertion d'un jeton supplémentaire dans la boucle de Fetch temporairement, le temps de satisfaire cette instruction. Pour toutes les autres instructions, le schéma d'accès à la mémoire programme n'est pas modifié.

Ainsi pour ces deux instructions, la boucle de Fetch est suspendue le temps d'effectuer l'accès mémoire pour Load-Store. Cette synchronisation supplémentaire permet de ne pas pénaliser dans les cas courants la boucle de Fetch dont le débit et la latence sont critiques pour la performance du processeur. Ce mécanisme de synchronisation entre la boucle de Fetch et l'unité Load-Store est un point intéressant car montre qu'une synchronisation conditionnelle évite de perturber le fonctionnement d'un pipeline établi dans les cas réguliers. Cette synchronisation est effectuée uniquement quand nécessaire, comme pour les branchements.

Contrairement au cas synchrone, la boucle de Fetch n'a pas besoin de compter un certain nombre de cycles pour attendre la requête provenant de l'unité Load-Store, nombre de cycles qui correspondrait à la profondeur du pipeline synchrone entre Fetch et Load-Store. Par une simple synchronisation conditionnelle, l'unité PC se trouve suspendue jusqu'à ce qu'elle puisse satisfaire cet accès mémoire demandé par une autre ressource, puis redémarre. Ceci peut prendre un temps quelconque. En terme de modélisation d'architecture, ceci montre une fois de plus que ce type de synchronisation conditionnelle, à priori difficile à implémenter en synchrone, est naturel pour notre architecture. Ceci est possible car il y a dé-corrélation entre la fonction et les performances. Grâce à la sémantique des processus communicants, il n'est pas nécessaire de savoir tôt dans le temps de cycle de conception les performances relatives des blocs, la profondeur du pipeline de l'architecture, pour pouvoir écrire un modèle architectural. Ceci est une grande aide à la conception d'architecture, non seulement en terme de temps / productivité de conception, mais aussi au niveau des libertés laissées à l'architecte.

4.3.2.5. Mécanisme d'interruption et mise en veille.

Les interruptions sont gérées par le décodeur, au sein de la boucle de Fetch. A chaque instruction, le décodeur observe le signal d'interruption. Si celui-ci est positionné, que le flag d'interruption autorise le départ en interruption et que le décodeur n'est pas en cours d'exécution d'une instruction retardée (Delayed Branch), un départ en interruption est exécuté. Le décodeur annule alors l'instruction courante, ainsi que les deux instructions qui suivent dans le pipeline de Fetch, interdit les interruptions en positionnant le flag à zéro, puis enfin demande à l'unité PC la sauvegarde du compteur programme dans le registre reg15 (via l'unité de branchement) et de sauter à l'adresse $0000. Au sein du décodeur, ce mécanisme d'interruption est alors proche d'un branchement pris non-retardé, sachant que le compteur programme est de son côté géré par l'unité PC et l'unité de branchement. Enfin, le retour d'interruption (instruction Drti) est équivalent à un saut retardé « Djmp reg15 » pour lequel le décodeur autorise à nouveau les interruptions en positionnant le flag à un.

Le test d'interruption implémente une garde indéterministe, car il n'y a pas dans ce cas de synchronisation entre le décodeur et le signal d'interruption. C'est le seul endroit du processeur où il y a nécessité d'arbitrer entre le cœur du processeur et l'extérieur. Si l'interruption arrive légèrement trop tard et n'est pas prise à cette instruction car l'arbitre en

décide autrement, celle-ci sera alors prise à l'instruction suivante, ce qui ne change rien vu du modèle de programmation.

L'instruction Sleep (Mise en veille) est aussi implémentée dans le décodeur. Pour cette instruction, le décodeur ne scrute pas le niveau d'interruption mais au contraire se synchronise dessus. Dans ce cas, le décodeur, et donc l'ensemble du processeur, sont suspendus jusqu'à ce que le niveau d'interruption soit positionné à un, puis continue à exécuter le flot d'instructions sans effectuer de départ en interruption. Il est donc impossible de partir en interruption sur l'instruction qui suit une instruction Sleep.

La micro-architecture permettant de modéliser le mécanisme d'interruption et de mise en veille, et son implémentation, seront respectivement présentés dans les paragraphe 5.5 et 6.4.

4.3.2.6. Personnalisation du processeur

Comme présenté dans le paragraphe 4.2.3, le cœur du processeur est personnalisable, soit avec une unité utilisateur spécifique, soit avec des périphériques spécifiques. L'unité utilisateur est l'une des quatre unités du processeur. Celle-ci est totalement indépendante des autres unités et est connectée au reste du chemin de données, c'est à dire au banc de registres et à l'interface bus. De son côté le décodeur ne fait que transmettre l'opcode utilisateur à cette unité et ses opérandes vers le banc de registres. Ainsi, l'unité utilisateur peut avoir zéro, un ou deux opérandes sources, zéro ou un registre destination, ce nombre d'opérandes étant spécifié dans l'instruction grâce à l'utilisation des bits de src-op (paragraphe 4.2.5). A partir de ce moment, l'unité utilisateur peut effectuer n'importe quel traitement sur les opérandes spécifiés. La correction du programme est garantie par le mécanisme de terminaison dans le désordre implémenté au sein du banc de registre. Ainsi, quelque soit la latence de l'unité, le reste du cœur du processeur attend son résultat pour pouvoir continuer si il y a un aléa lecture / écriture sur les registres concernés. De la même manière, l'interface bus adapte son débit au débit d'entrée de l'unité utilisateur, si le programme accède en continu à cette même unité. Ainsi, suivant le recouvrement dans le programme entre les unités d'exécution, l'architecture du chemin de données permet de tirer pleinement parti des performances d'une unité utilisateur tout en garantissant la correction du programme.

De la même manière, l'unité périphérique est intégrée au sein de l'unité Load-Store. Celle ci a moins de flexibilité au niveau des opérandes, elle peut utiliser un opcode (l'adresse périphérique) et soit un opérande source, soit un opérande destination. Comme pour l'unité utilisateur, la correction du programme est garantie quelque soit ses performances grâce à la stratégie de terminaison dans le désordre.

4.4. Optimisation du code à la compilation

Ce paragraphe illustre comment le programme doit être optimisé afin de profiter pleinement de l'architecture du processeur.

Tout d'abord, les mémoires du processeur utilisent une architecture entrelacée (cf. paragraphe 5.1), ce qui permet de pipeliner des accès mémoires à des adresses consécutives non égale modulo 4. L'accès à des tronçons de programme continu permet ainsi de profiter pleinement de cette architecture. Le compilateur devra alors choisir avec soin les adresses de branchements afin de ne pas effectuer un branchement à une adresse égale modulo 4 à l'adresse de départ. Même si la correction fonctionnelle est garantie par l'architecture, ceci permet de ne pas ralentir l'exécution du programme lors d'un branchement en raison d'un

accès à un bloc mémoire non encore libre. La même contrainte s'applique à la mémoire donnée. Ceci sera particulièrement vrai lorsqu'on effectue une boucle de lecture / écriture sur un tableau par exemple. Ainsi, le compilateur devra choisir avec soin l'adresse de base et les offsets pour que les accès mémoires modulo 4 se repartissent entre des blocs mémoires distincts afin d'obtenir un débit mémoire optimal.

Le compilateur devra aussi optimiser l'utilisation des différents branchements. Comme la majorité des branchements (à un tronçon de code ou à un sous programme) sont retardés, le compilateur devra insérer des instructions utiles dans les delay slots de branchement. Ceci est facile à effectuer pour des appels de programme avec le chargement de ses paramètres d'entrée (paragraphe 4.2.4). Dans le cas contraire, l'utilisation de un ou deux Nop est nécessaire afin de remplir le pipeline de la boucle de Fetch en attendant le branchement effectif. Seul le branchement conditionnel peut ne pas être retardé. Ainsi, Bcc peut être utilisé à la place de DBcc si il n'est pas possible d'insérer des instructions utiles. L'instruction Bcc ralentit le programme car suspend le décodeur en attente du résultat de branchement (mieux vaut exécuter des Nop), mais ceci améliore la densité de code en supprimant ces Nop superflus.

Enfin, le compilateur devra surtout optimiser le code afin de tirer pleinement parti de l'exécution entrelacée permise par les quatre unités d'exécution parallèles et le mécanisme de terminaison / ré-écriture dans le désordre au sein du banc de registres. Comme pour toute architecture RISC pipelinée [HENN 96], les dépendances de données ralentissent l'exécution du programme. Le but de l'optimisation est alors de ré-ordannancer les instructions pour obtenir un recouvrement optimal entre les unités et repousser le plus possible les dépendances de noms entre les registres sources et destination. Pour illustrer ceci, prenons en exemple une routine de filtrage de type FIR (Figure 4-7).

```
Fir :  Macc   acc0, r0, r1      ;      multiplication-accumulation
       Ldw    r0, 0(r10)        ;      chargement      coefficient
       Ldw    r1, 0(r11)        ;      chargement      échantillon
       Add    r10, r10, #1      ;  mise  à  jour  ptr  coefficient
       Dbne   r7, #0, Fir:      ;        branchement        retardé
       Add    r11, r11, #1      ;  mise  à  jour  ptr  échantillon
       Sub    r7, r7, #1        ; mise à jour compteur boucle
```

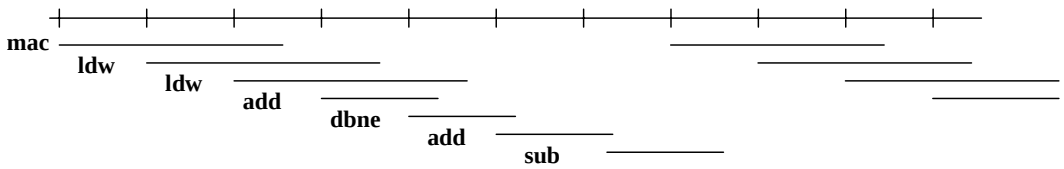


Figure 4-7 : Routine de filtrage FIR optimisée et diagramme temporel montrant le recouvrement des instructions

Ce type de routine revient à effectuer une boucle avec accumulation de produit sachant que échantillons et coefficients sont en mémoires données. L'optimisation consiste à dérouler la boucle et la réécrire sur une fenêtre plus grande. On suppose que la boucle est correctement initialisée avec son compteur de boucle et des valeurs d'initialisation. La séquence commence alors avec les instructions les plus longues : Mac et Ldw. Ainsi les chargements mémoire se superposent avec l'exécution de la multiplication-accumulation. Le reste de la boucle effectue la mise à jour des pointeurs mémoires et le branchement qui est retardé. Le branchement peut

être retardé car l'indice de boucle est pré-calculé et décalé pour sortir de la boucle. Les valeurs mémoires et les pointeurs mémoires sont alors disponibles pour l'itération suivante. Ce programme montre un recouvrement optimal et permet au processeur de tourner à sa vitesse maximale. En conclusion, tout comme pour une machine RISC synchrone, le maximum de performance sera donc obtenu avec une bonne technologie de compilation / optimisation [BAUE 97].

4.5. Conclusion

Nous venons de présenter le jeu d'instruction et l'architecture du processeur Aspro. Ce prototype est un microprocesseur de type RISC 16-bit incluant une unité utilisateur et des périphériques dédiés. Son jeu d'instruction a été présenté en détail. Celui-ci est relativement classique outre les mécanismes de branchement retardé. Nous avons alors montré comment nous pouvions obtenir un pipeline en effectuant des décomposition sur le langage CHP. L'introduction de canaux de communications permet d'implémenter des synchronisations conditionnelles. Ceci permet d'optimiser les performances tout en gardant la correction fonctionnelle. Ainsi, l'architecture du processeur que nous venons de présenter est une architecture originale qui décorrèle le plus possible la lecture / décodage instruction de son exécution. Afin d'exécuter les instructions, le principe du processeur est de les envoyer dans l'ordre dans le chemin de données, de les dispatcher vers des unités d'exécutions concurrentes et de terminer ces instructions dans le désordre. La correction du programme est garantie par un mécanisme local de réservation de lecture / écriture dans le banc de registre. Les instructions sont alors distribuées dans le processeur au plus tôt et s'exécutent avec des latences et temps de cycle minimums en fonction de l'unité concernée et des données. La performance de ce type d'architecture est pleinement exploitable avec un bon ordonnancement des instructions. Le chapitre 5 qui suit présente la micro-architecture du processeur avec la décomposition / implémentation de ses différentes unités fonctionnelles.

Chapitre 5 :

Le processeur Aspro :
Architecture et micro-architecture

Introduction

En partant de la spécification de l'architecture du processeur ASPRO : jeu d'instructions, périphériques, mode de fonctionnement, qui ont été définis dans le chapitre précédent, nous allons présenter dans ce chapitre son architecture de manière plus précise, et plus principalement sa micro-architecture. Toutes les explications d'architecture sont basées sur l'écriture de modèles CHP. Comme dans la méthode que nous avons présentée dans le chapitre 3, un premier modèle du processeur correspondant à la spécification a été décrit dans le langage CHP. Ce premier modèle est un modèle fonctionnel, séquentiel, de haut niveau. Le but est alors d'introduire du parallélisme grâce à des décompositions de processus, afin de raffiner ce modèle et obtenir un modèle à grain plus fin. Dans cette étape, l'introduction de canaux de communications pour échanger les informations entre les différentes ressources matérielles permet de spécifier les dépendances de données minimales entre les grands blocs fonctionnels. Une première étape de décomposition a été effectuée afin d'isoler les principaux blocs fonctionnels du processeur, ce sont ceux qui ont été présentés dans le chapitre 4 précédent.

Ce chapitre est structuré afin de détailler la micro-architecture de chacun de ces blocs. L'écriture CHP de chaque bloc, issu de cette première décomposition, est alors donnée comme modèle de départ. Des décompositions successives permettent de raffiner à leur tour

chacun de ses blocs afin d'en obtenir une description avec un grain le plus fin, qui soit uniquement décrit en CHP synthétisable (tel que défini dans le chapitre 3).

Cette méthode de décomposition montre comment un langage de description de processus communicant permet d'introduire du parallélisme et de modéliser les dépendances fonctionnelles minimales inhérentes à la spécification, c'est à dire le jeu d'instructions. On montrera ainsi qu'il est possible d'implémenter des unités fonctionnelles à temps de calcul et consommation minimum, en fonction à la fois des instructions et des données. Le niveau de parallélisme obtenu, que ce soit au niveau le plus fin ou au niveau bloc, correspond alors à la notion de pipeline.

Le corps de ce chapitre se décompose de la manière suivante. La paragraphe 5.1 présente l'architecture des mémoires. C'est un type d'architecture simple et régulier, sur lequel repose une grande partie des différentes unités du processeur. Nous présentons alors dans le paragraphe 5.2 les différentes unités d'exécution du processeur : unité arithmétique et logique, unité de multiplication-accumulation (l'unité utilisateur), unité de branchement, unité d'accès mémoires. Le banc de registres, qui connecte ces différentes unités d'exécutions et autorise un mécanisme d'écriture dans le désordre, est l'un des blocs principal du processeur. Son architecture est donnée dans le paragraphe 5.3. Ensuite, le paragraphe 5.4 traite plus spécifiquement d'arithmétique asynchrone : architectures spécifiques pour l'addition et la multiplication. Le paragraphe 5.5 présente la micro-architecture de la boucle lecture-instruction / décodage-instruction. Enfin, le paragraphe 5.6 présente les périphériques. L'implémentation de ces différentes unités et l'optimisation globale de l'architecture obtenue seront présentées dans le chapitre 6.

5.1. Architecture des mémoires

Comme premier exemple d'étude architecturale, nous présentons le choix d'architecture des mémoires. Pour la conception des mémoires d'Aspro se pose les contraintes de performances suivantes : fort débit et faible latence. Pour la mémoire programme, la contrainte la plus forte est celle de débit car chaque instruction nécessite l'accès à la mémoire programme en lecture. L'objectif est d'obtenir un temps de cycle mémoire de 2.5 ns (avant routage) pour atteindre les 200 Mips visés. La contrainte de débit sur la mémoire données est plus faible puisque celle-ci n'est pas nécessairement accédée à chaque instruction.

Pour l'implémentation de cette première version d'Aspro, il n'était pas prévu la réalisation de mémoires asynchrones. En conséquence, à la fois pour des raisons de densité, de rapidité de conception et surtout afin de respecter notre objectif de flot de conception basé cellules standard, les mémoires d'Aspro sont implémentées avec des blocs de mémoires standard synchrones obtenus par des générateurs de STMicroelectronics. Ceci impose l'étude et la réalisation d'interfaces entre logique synchrone et asynchrone, ce qui est présenté dans le paragraphe 6.2. Ces interfaces sont relativement simples à réaliser mais sont coûteux en performances. En effet, l'utilisation de blocs synchrones impose de prendre des marges temporelles sur les interfaces pour respecter les hypothèses de timing synchrones.

En conséquence, pour la mémoire programme, l'utilisation d'un unique bloc mémoire synchrone de 16k mots instructions et de son interface ne permet pas d'atteindre l'objectif de débit visé. La solution architecturale présentée dans le paragraphe suivant consiste à entrelacer l'accès à différents sous-blocs mémoire en parallèle [RENA 99a].

5.1.1. Entrelacement entre différents sous-blocs mémoires

Le principe consiste à pipeliner le décodeur d'adresse de la mémoire afin d'entrelacer les accès à différents blocs mémoires plus petits, donc plus rapides. L'objectif est d'atteindre un débit moyen égal au débit du décodeur d'adresse et de minimiser la latence.

L'architecture résultante s'obtient directement grâce à une décomposition CHP. A titre d'exemple, la décomposition CHP correspondante est donnée de manière complète afin de montrer la facilité d'écriture et la manière de dériver l'architecture à partir d'une simple manipulation sur les processus. Supposons que la mémoire s'écrive de la manière suivante (Figure 5-1) :

```
Mem = *[ RW?rw , AD?ad[n..0] ;
            [ rw=0 => DW?mem[ad] -- écriture
            @ rw=1 => DR!mem[ad] -- lecture
        ]   ]
```

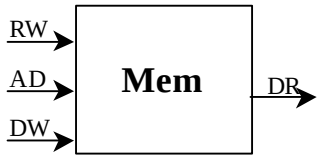


Figure 5-1 : Modèle CHP de la mémoire et sa vue symbole.

Le canal RW contient l'information de contrôle lecture ou écriture, le canal AD l'adresse mémoire, DW la donnée à écrire et enfin DR la donnée lue. En découpant la mémoire en deux sous-blocs mémoires égaux Mem0 et Mem1 de taille moitié de la mémoire initiale, on adresse chaque sous bloc par le bit d'adresse de poids faible (on peut prendre à priori n'importe quel bit d'adresse). La mémoire peut-être réécrite de la manière suivante :

```
Mem = *[ RW?rw , AD?ad[n..0] ;
            [ ad[0]=0 =>                  -- bloc mémoire mem0
                  [ rw=0 => DW?mem0[ad[n..1]] – écriture
                  @ rw=1 => DR!mem0[ad[n..1]] –- lecture
                  ]
            @ ad[0]=1 =>                  -- bloc mémoire mem1
                  [ rw=0 => DW?mem1[ad[n..1]] – écriture
                  @ rw=1 => DR!mem1[ad[n..1]] –- lecture
          ] ]     ]
```

Si on extrait les deux blocs mémoires Mem0 et Mem1 du processus précédent, on introduit pour chaque sous bloc mémoire Mem_i ses propres canaux de communication RW_i, AD_i, DW_i, DR_i. On obtient alors le processus P0 suivant composé en parallèle des deux blocs mémoires mem0 et mem1 (Figure 5-2).

```
P0 = *[ RW?rw , AD?ad[n..0] ;
            [ ad[0]=0 =>                  -- bloc mémoire mem0
                  RW0!rw, AD0!ad[n..1],
                  [ rw=0 => DW?dw ; DW0!dw – écriture
                  @ rw=1 => DR0?dr ; DR!dr –- lecture
                  ]
            @ ad[0]=1 =>                  -- bloc mémoire mem1
                  RW1!rw, AD1!ad[n..1],
                  [ rw=0 => DW?dw ; DW1!dw – écriture
                  @ rw=1 => DR1?dr ; DR!dr –- lecture
          ]   ]     ]
```

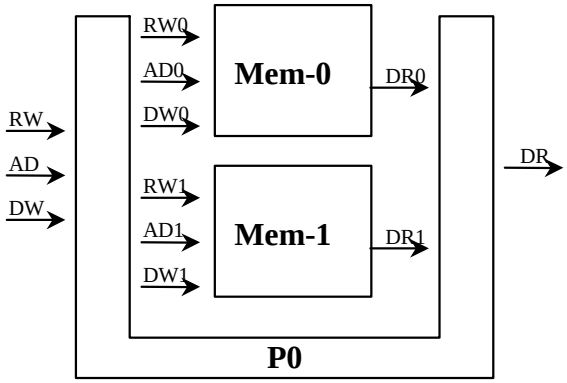


Figure 5-2 : Décomposition de la mémoire en trois processus P0, Mem0 et Mem1

Ainsi, suivant les valeurs de contrôle rw et ad[0], le processus P0 effectue une requête de lecture ou d'écriture vers l'une des deux mémoires mem0 et mem1, puis en cas de lecture retransmet le résultat en sortie. Cette architecture est encore séquentielle car toute nouvelle requête de lecture mémoire ne peut se faire qu'une fois que la donnée DR0 ou DR1 à été lue et envoyée sur la sortie DR. La décomposition n'a donc pas introduit de parallélisme, mais simplement isolé les blocs fonctionnels.

La décomposition suivante consiste à isoler les canaux de communication RW, AD, DW, DR dans des processus indépendants. Pour cela, le processus d'origine P0 doit générer des valeurs de contrôle vers chaque canal de donnée AD, DW, DR : soit les canaux de contrôle C-AD, C-DW, C-DR. On obtient alors les quatre processus suivants (Figure 5-3) : un décodeur P1, un multiplexeur P2 pour l'envoi des adresses, un multiplexeur P3 (un *split*) pour l'envoi des données en écriture et un démultiplexeur P4 (un *merge*) des données en lecture :

Décodeur (P1)

```
*[ RW?rw, AD[0]?ad[0];
   [ ad[0]=0 => RW0!rw, C-AD!0,
     [ rw=0 => C-DW!0
     @ rw=1 => C-DR!0
     ]
   @ ad[0]=1 => RW1!rw, C-AD!1,
     [ rw=0 => C-DW!1
     @ rw=1 => C-DR!1
 ] ] ]
```

Split Adresses (P2)

```
*[ C-AD?c, AD?ad[n..1];
   [ c=0 => AD0!ad[n..1]
   @ c=1 => AD1!ad[n..1]
]  ]
```

Split Données écriture (P3)

```
*[ C-DW?c, DW?dw ;
    [ c=0 => DW0!dw
    @ c=1 => DW1!dw
 ]  ]
```

Merge Données lecture (P4)

```
*[ C-DR?c ;
   [ c=0 => DR0?dr ; DR!dr
   @ c=1 => DR1?dr ; DR!dr
]  ]
```

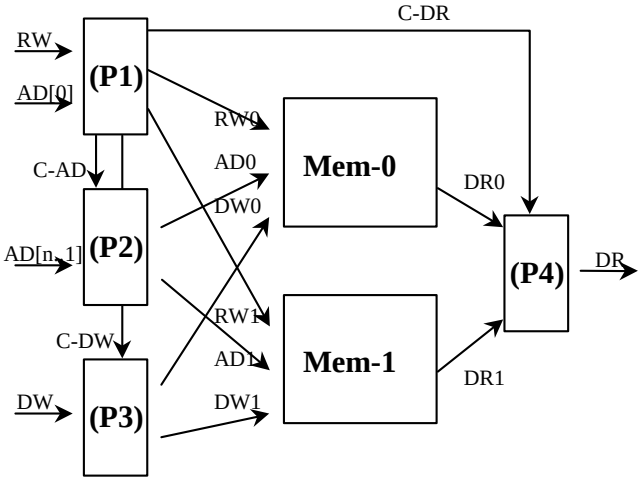


Figure 5-3 : Décomposition finale de la mémoire en deux sous-blocs

A présent, le décodeur P1 peut générer et empiler des requêtes de lecture / écriture vers les sous-blocs, sans nécessairement se synchroniser sur la fin de lecture. La commande de lecture C-DR permet de transmettre le bon résultat DR0 ou DR1 vers la sortie, pendant que le décodeur émet la requête suivante : soit vers le même bloc, soit vers l'autre bloc. La décomposition a donc introduit du parallélisme car entrées et sorties de la mémoire ne sont pas synchronisées dans un unique processus, comme c'était le cas avec le processus P0.

Les différents processus P1, P2, P3, P4 sont directement synthétisables en logique QDI avec la méthode présentée dans le chapitre 3. Les débits et latences obtenus dans cette architecture sont très variés. Les blocs mémoires sont relativement lents, par contre les processus de contrôle 1,2,3,4 présentent des latences faibles et surtout des débits élevés. Ainsi dans cette architecture, les blocs mémoires mem_i peuvent être accédés en parallèle pendant que les étages d'entrées distribuent des requêtes à différents blocs et ce toujours au plus tôt suivant la séquence d'adresse reçue. Si le même bloc mémoire est accédé, le débit est temporairement ralenti à l'entrée car il faut attendre la disponibilité de ce bloc. Dans le cas où tous les blocs sont utilisés concurremment, le débit de la nouvelle mémoire est alors égal au débit des étages d'entrées. Ce principe de décomposition peut être utilisé de manière récursive jusqu'à obtenir des blocs mémoires de la taille souhaitée et donc les performances correspondantes.

5.1.2. Optimisation de l'architecture de la mémoire

Pour que cette architecture soit optimale et plus performante que la mémoire d'origine, il faut trouver le nombre N de sous-blocs tel que :

- la latence résultante n'excède pas la latence d'origine :

$$L_{decoder(N)} + L_{split\ (N)} + L_{bloc16k/N} + L_{merge\ (N)} \leq L_{mem(16k)}$$

- le temps de cycle des différents processus permet un entrelacement maximum (N-1 requêtes sont envoyées pendant qu'un bloc mem_i est accédé) :

$$T_{bloc16/N} \geq (N-1) * \max(T_{decoder}, T_{split}, T_{merge})$$

D'une manière générale, on constate qu'un nombre important de blocs permet de gagner en temps de cycle mais au détriment de la latence. Voici les performances (latence et temps de

cycle avant routage) des différents processus pour une décomposition en 4 blocs de 4k mots instructions :

	Mémoire 16k	Mémoire 4k	Décodeur	Split	Merge
temps de cycle (ns)	10.1	7.2	1.8	2.3	2.3
Latence (ns)	7.0	5.5	0.3	0.5	0.5

Les équations de latence et débit sont bien respectées. Il y a très peu de perte de latence (0.3+0.5+5.5+0.5=7.3) par rapport à un unique bloc de 16k (latence 7 ns). Le recouvrement entre bloc s'effectue correctement (7.2 ns contre 3*2.3=6.9 ns), ce qui permet l'utilisation de 3 blocs en parallèle pendant que le décodeur distribue des requêtes au bloc suivant. Ainsi, le gain en débit est important : le temps de cycle vu de l'entrée est à présent en moyenne de 2.3 ns alors que les temps de cycles respectifs des blocs mémoires 4k et 16k sont de 7.2 ns et 10.1 ns.

L'architecture choisie pour la mémoire programme est donc une décomposition en 4 sous-blocs de 4k mots 24 bits. Le décodage d'adresse est effectué en une fois avec les deux bits de poids faible de l'adresse. Pour des instructions successives, les requêtes de lecture sont donc toujours envoyées à des blocs distincts, les blocs étant accédés en cycles modulo 4 (voir Figure 5-4). Ce choix est de plus intéressant pour une autre raison. En effet, l'adresse de la mémoire programme est générée par un additionneur dans l'unité PC (paragraphe 5.5.2). Grâce aux propriétés de celui-ci (paragraphe 5.7), les bits de poids faible sont toujours disponibles plus tôt que les bits de poids fort, ce qui permet de commencer le calcul de décodage P1 pendant que les bits d'adresses restant sont générés, puis consommés dans P2. Ainsi, les latences entre les différents chemins sont au mieux équilibrées.

Dans le cas où un même bloc est consécutivement accédé (cas de branchement ou de saut), le débit est réduit temporairement au débit d'un bloc. Le compilateur/assembleur peut optimiser ces cas en choisissant avec soin les adresses de branchements pour éviter les sauts à des adresses égales modulo 4.

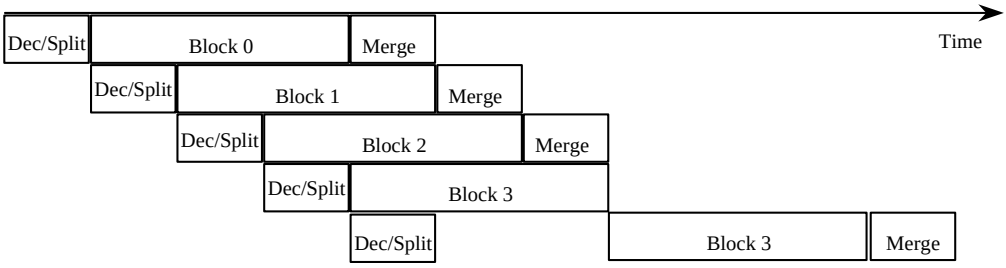


Figure 5-4 : recouvrement des accès entre les différents sous-blocs mémoires ou suspension dans le cas d'un accès consécutif

Une dernière optimisation de l'architecture concerne l'équilibre du pipeline obtenu (cf. paragraphe 6.3). Dans la Figure 5-3, le canal de commande C-DR contient le numéro de bloc où l'accès en lecture est effectué. Pour que l'entrelacement entre les N blocs puisse être possible, ce canal doit mémoriser les N-1 requêtes de lecture jusqu'à ce que la donnée soit disponible en sortie du Nième bloc. Ce mécanisme s'implémente simplement sur le canal C-DR avec une Fifo de profondeur N-1. Chaque élément i de la Fifo s'écrit de la manière

suivante : `*[C-DR`$_i$`?c ; C-DR`$_{i+1}$`!c]`. On retrouve ainsi l'élément de pipeline élémentaire (chapitre 3). Le schéma final de la mémoire est donné Figure 5.5.

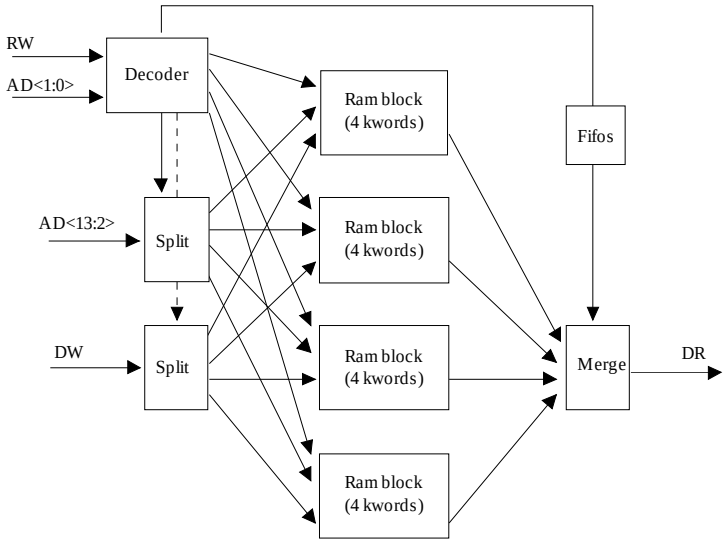


Figure 5-5 : Architecture de la mémoire programme d'Aspro : découpage en 4 blocs de 4k mots, avec entrelacement des accès.

5.1.3. Conclusion et comparaison avec une réalisation synchrone

Pour implémenter un schéma d'entrelacement équivalent avec une architecture synchrone, il serait nécessaire d'ajouter du contrôle (et donc une machine d'état finis) afin de pouvoir suspendre le pipeline en cas d'accès à un même bloc suivant la séquence d'adresse. Dans cette architecture asynchrone, le contrôle de l'entrelacement est naturellement implémenté par la synchronisation locale entre le décodeur d'adresse et les blocs correspondants (Processus P1). Les requêtes sont simplement distribuées vers les différents blocs, ces requêtes seront acceptées ou non suivant leurs disponibilités respectives. Il n'y a rien de plus à modéliser et à implémenter. Ceci est un argument en faveur de la facilité et de la rapidité de conception.

Au niveau performance, sans perte de latence, la mémoire asynchrone montre un débit moyen égal au débit de l'organe le plus rapide. Dans le cas synchrone, comme l'horloge dépend nécessairement du bloc le plus lent, on obtiendrait : i) un débit global égal au débit du bloc mémoire, ii) une latence égale à la somme des latences, soit le nombre d'étages de pipeline, à savoir 3. L'architecture asynchrone permet donc de rompre la relation synchrone : *Débit = N étages / Latence* et d'optimiser les paramètres latence et débit de manière indépendante.

L'architecture de cette mémoire est un exemple type de pipeline et de parallélisme qui permet d'optimiser le débit moyen sans perte de latence. C'est une architecture qu'on peut qualifier de type démultiplexeur / multiplexeur (« *split* / *merge* »). Le schéma d'entrelacement permet ainsi de cacher dans l'architecture des débits faibles, le tout s'implémentant de manière naturelle avec des processus communicants. Même pour un algorithme à priori régulier tel qu'une mémoire, l'approche asynchrone permet de tirer parti de l'irrégularité fonctionnelle due à l'enchaînement d'accès en moyenne à des adresses différentes, ce qui est une source de parallélisme.

5.2. Etude des unités d'exécution

Ce paragraphe traite de l'étude du chemin de données du microprocesseur. Les aspects purement arithmétiques (addition, multiplication) seront traités dans le paragraphe 5.4. Nous nous intéressons ici à la manière de décomposer les différentes unités d'exécution qui composent le processeur Aspro : l'unité arithmétique et logique (Alu), l'unité de multiplication accumulation (Macc), l'unité de branchement (Bru) et l'unité d'accès mémoires (Ld/St). L'objectif est de réaliser des unités à temps de calcul minimum (débit et latence) et consommation minimum, en fonction à la fois des instructions et des données. Pour cela, l'étude repose sur des décompositions de processus CHP, l'architecture obtenue est alors l'image des dépendances fonctionnelles.

5.2.1. Unité arithmétique et logique

Les instructions de l'unité arithmétique et logique (Alu) se regroupent en quatre type d'instructions : les instructions arithmétiques (add, sub, neg, not), les décalages et rotations (shl, shr, shrs, rotl, rotr), les instructions logiques (and, or, xor, bitr) et les instructions de comparaison (min, max, slt signé ou non signé). L'Alu est donc décomposée en quatre sous-unités distinctes.

Afin d'obtenir pour l'Alu un débit moyen élevé et de lever les contraintes de débit sur chaque sous-unité, un schéma d'entrelacement comme celui de la mémoire programme est adopté. L'encodage des instructions étant approprié, les 2 bits de poids faible du code opératoire de l'Alu sont décodés et utilisés pour envoyer les deux opérandes et le reste du code opératoire vers les sous-unités correspondantes (Figure 5-6). Comme pour la mémoire, les résultats de chaque unité sont ensuite envoyés dans l'ordre via le multiplexeur de sortie vers le banc de registre grâces aux requêtes stockées dans une fifo de profondeur adaptée.

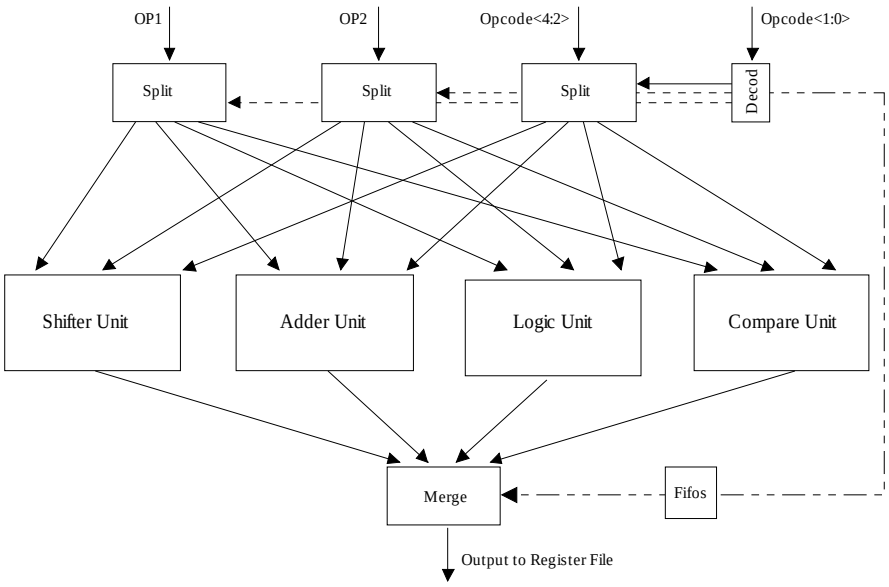


Figure 5-6 : Architecture de l'Alu avec entrelacement de 4 sous-unités

Cette architecture permet différents compromis débit / latence sur l'ensemble de l'alu suivant le taux de parallélisme potentiel entre unités. Dans un programme classique, une

moyenne de deux instructions consécutives dans des sous-unités distinctes semble facile à obtenir. Avec cette hypothèse, afin que le recouvrement entre sous-unités soit optimal, le temps de cycle moyen par sous-unité doit être au plus égal au double du temps de cycle des étages d'entrées-sorties, soit environ 5 ns. Le temps de cycle moyen de l'Alu est alors égal au débit des processus d'entrées, soit 2.5 ns.

Grâce à cette architecture entrelacée, les contraintes de débit sont faibles pour chaque sous-unité. Pour cette raison et afin de simplifier la logique d'acquittement, les unités seront simplement synthétisées avec une logique combinatoire (sauf l'unité de décalage qui est pipelinée en raison de sa structure régulière). Les paragraphes suivants présentent en détail l'architecture des quatre sous-unités de l'Alu.

5.2.1.1. Unité logique

L'unité logique s'écrit en CHP de la manière suivante (ce processus est identique pour les 16 bits i de l'unité) :

```
*[ OPCODE[4..2]?opc, OP1[i]?op1, OP1[15-i]?bitr, OP2[i]?op2;
    [ opc="000" =>        -- AND
          [ op1=1 and op2=1          => S[i]!1
          @ others                   => S[i]!0 ]
    @ opc="010" =>        -- OR
          [ op1=0 and op2=0          => S[i]!0
          @ others                   => S[i]!1 ]
    @ opc="100" =>        -- BRV
          [ bitr=0                   => S[i]!0
          @ bitr=1                   => S[i]!1 ]
    @ opc="110" =>        -- XOR
          [ op1=op2                  => S[i]!0
          @ others                   => S[i]!1
 ]  ]     ]
```

Le premier niveau de gardes effectue le décodage de l'opcode de l'unité, la valeur de sortie est ensuite générée dans le deuxième niveau de gardes suivant la valeur des bits d'entrées. L'instruction "Bit Reverse" consiste simplement à envoyer la valeur du bit (15-i) sur la sortie d'ordre (i). Elle est un peu particulière puisque dans ce cas le 2ème opérande n'est pas utilisé, ce bit devra donc être correctement acquitté. Cette unité a par construction une structure très régulière, temps de cycle et latence ne sont donc pas dispersés suivant les valeurs des opérandes. On obtient alors les chiffres de performance suivants : une latence de 1,1 ns et un temps de cycle de 3 ns.

5.2.1.2. Unité Arithmétique

L'unité arithmétique implémente les instructions Add, Sub, Neg et Not. Grâce aux propriétés du calcul signé en complément à deux, un unique additionneur est utilisé, l'unité arithmétique s'écrit alors de la manière suivante :

```
*[  OPCODE[4..2]?opc , OP1?op1 , OP2?op2 ;
    [ opc="110" =>  S!(op1+op2+0)        -- add
    @ opc="101" =>  S!(op1+not(op2)+1)   -- sub
    @ opc="010" =>  S!(not(op1)+op2+0)   -- not (avec op2=0)
    @ opc="011" =>  S!(not(op1)+op2+1)   -- neg (avec op2=0)
 ]  ]
```

Pour les instructions not et neg, un immédiat à zéro est directement encodé dans l'instruction. Ceci permet d'utiliser le 2ème opérande sans avoir à générer un zéro en entrée de l'additionneur. De plus, l'encodage est adapté tel que : la retenue entrante de l'additionneur est le bit 2 de l'opcode, les inversions sur les opérandes OP2 et OP1 sont respectivement effectuées de manière conditionnelle avec les bit 3 et 4 de l'opcode. Les deux fonctions d'inversions sont décrites ainsi :

```
*[  OPC?opc , E?x ;
    [ opc=0 => S!not[6](x)
    @ opc=1 => S!x
 ]  ]
```

Cet encodage optimal n'impose pas de décodage supplémentaire et permet l'utilisation directe des bits du code pour contrôler l'additionneur (Figure 5-7). En raison de l'absence de registre d'état, on peut noter que la retenue sortante de l'additionneur n'est ni utilisée, ni générée.

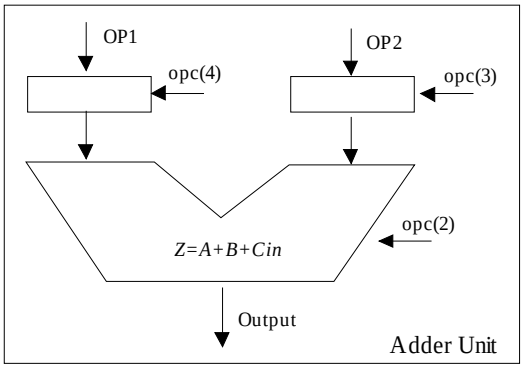


Figure 5-7 : Architecture de l'unité arithmétique

L'additionneur utilisé est un additionneur à retenue bondissante (voir paragraphe 5.4.1). L'unité arithmétique montre ainsi des performances qui sont fonctions de la valeur des opérandes, soit des latences comprises entre 1.3 ns et 2 ns et des temps de cycle entre 4 ns et 5.5 ns.

5.2.1.3. Unité de comparaison

L'unité de comparaison implémente les instructions Min, Max, Slt. Le bit de poids faible de l'opcode indique si l'opération est signée ou non. De manière évidente, on remarque que si l'opération est signée et que les bits de signes des opérandes sont différents le résultat de ces opérations est immédiat. Dans les autres cas, la soustraction des deux opérandes est nécessaire pour déterminer si l'opérande op1 est plus petit que l'opérande op2.

Afin d'optimiser les cas favorables et de ne pas pénaliser la latence des cas plus lents, la soustraction est toujours effectuée en parallèle du décodage des bits de signes, son résultat étant utilisé ou non suivant le signe des opérandes. En terme de consommation, ce n'est pas le plus optimal mais ceci permet de bonnes performances en vitesse. En effet, une soustraction conditionnelle nécessiterait du contrôle supplémentaire, ce qui imposerait une perte en latence et débit.

[6] Au niveau matériel, la fonction not est effectuée par simple inversion des signaux double rails (chapitre 3).

Le soustracteur est implémenté en utilisant un additionneur avec : i) inversion du deuxième opérande, ii) une retenue entrante à un, iii) le matériel réduit à son minimum pour ne générer que la retenue sortante Cout.

Le calcul du test de comparaison "op1<op2" peut s'écrire de la manière suivante :

```
*[ OP1[15]?signe1 , OP2[15]?signe2 , OPCODE[2]?calcul_signe;
    [ calcul_signe = 1    =>    -- calcul signé
         [ signe1=0 and signe2=1 => op1_op2!0 , COUT?
         @ signe1=1 and signe2=0 => op1_op2!1 , COUT?
         @ others         =>
              COUT?cout; [ cout=0 => op1_op2!1
                         @ cout=1 => op1_op2!0
         ]               ]
    @ calcul_signe = 0    =>    -- calcul non-signé
              COUT?cout; [ cout=0 => op1_op2!1
                         @ cout=1 => op1_op2!0
 ]  ]                    ]
```

Le canal op1_op2 contient alors le résultat du test "op1<op2". Celui-ci est donc calculé au plus tôt suivant l'instruction et le signe des deux opérandes. On voit de plus que la retenue Cout est toujours consommée car celle-ci est toujours générée. Le processus de sortie (voir Figure 5-8) est alors simple : il envoie la valeur de op1, op2 ou génère 0 ou 1 suivant le résultat du test op1_op2 et l'instruction reçue (min, max, ou slt).

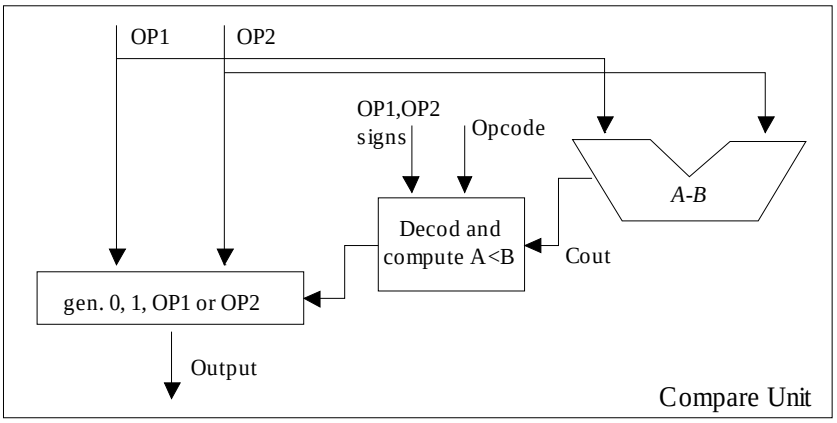


Figure 5-8 : Architecture de l'unité de comparaison

Cette unité implémente un calcul au plus tôt, ce qui permet d'importantes dispersions en latence, entre 1,2 ns et 2,5 ns. Néanmoins, le temps de cycle est relativement stable (5.5 ns) car le soustracteur est toujours activé. Son temps de cycle intervient donc toujours dans le temps de cycle de l'unité. Cette architecture correspond donc à un compromis latence / débit / consommation qui privilégie la latence.

5.2.1.4. Unité décaleur

L'unité décaleur implémente les instructions de décalage à gauche, décalage à droite signé ou non-signé, rotation à droite ou à gauche. La valeur de décalage est codée sur les 4 bits de poids faible du deuxième opérande, ce qui autorise des décalages de 0 à 15. L'architecture adoptée pour le décaleur est de type logarithmique. Ce type de solution est intermédiaire entre une architecture purement combinatoire et une architecture à base de registre à décalage et offre un bon compromis vitesse / consommation.

L'unité est constituée de quatre étages distincts qui se chargent respectivement des décalages d'ordre 1, 2, 4, ou 8 bits (voir Figure 5-9). Il suffit d'utiliser la valeur du bit de OP2 correspondant pour décider s'il y a ou non décalage dans l'étage considéré. L'architecture présente un pipeline régulier associant contrôle et données, basé sur l'association de cellules élémentaires échangeant les données au sein de chaque étage.

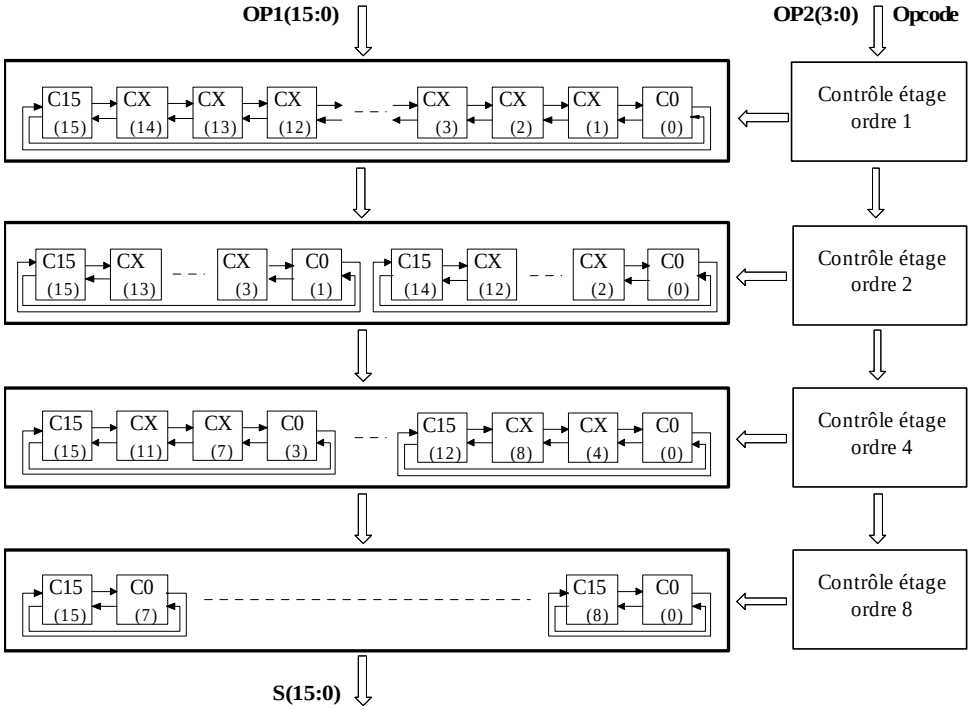


Figure 5-9 : architecture de l'unité de décalage

Au niveau fonctionnel, on peut remarquer que l'instruction de décalage à droite signé (shrs) est équivalente à un décalage à droite non signé (shru) si l'opérande OP1 est positif. Cette assimilation est effectuée dans le premier étage de contrôle. La décomposition en processus CHP montre que trois cellules différentes sont nécessaires pour faire transiter les données dans le réseau de connexion. En effet, tandis que les opérations de rotation sont régulières, les opérations de décalage nécessitent la génération de 0 ou de 1 sur les bits d'extrémité pour effectuer : soit l'extension de signe sur les bits de poids forts pour les décalages à droite, soit la génération de zéro sur les bits de poids faible pour les décalages à gauche. On obtient alors les trois cellules suivantes, toutes trois ayant la même vue externe (Figure 5-10).

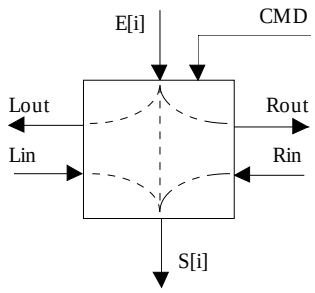


Figure 5-10 : cellules de décalage (type 15, X, 0)

```
CellX : Cellules du milieu
*[ CMD_X?cmd , E?x ;
    [ cmd=0 => S!x                              -- pas de décalage
    @ cmd=1 => ROUT!x , [LIN?lin; S!lin]        -- ROTR / SRU / SRS
    @ cmd=2 => LOUT!x , [RIN?rin; S!rin]        -- ROTL / SL
 ]  ]
Cell15 : Cellules pour les bits de poids fort
*[ CMD_15?cmd , E?x ;
    [ cmd=0 => S!x                              -- pas de décalage
    @ cmd=1 => ROUT!x , [LIN?lin; S!lin]        -- ROTR / SRU / SRS
    @ cmd=2 => LOUT!x , [RIN?rin; S!rin]        -- ROTL
    @ cmd=3 => LOUT!0 , [RIN?rin; S!rin]        -- SL
 ]  ]
Cell0 : Cellules pour les bits de poids faible
*[ CMD_0?cmd , E?x ;
    [ cmd=0 => S!x                              -- pas de décalage
    @ cmd=1 => ROUT!x , [LIN?lin; S!lin]        -- ROTR
    @ cmd=2 => LOUT!x , [RIN?rin; S!rin]        -- ROTL / SL
    @ cmd=3 => ROUT!0 , [LIN?lin; S!lin]        -- SRS & OP1>0 / SRU
    @ cmd=4 => ROUT!1 , [LIN?lin; S!lin]        -- SRS & OP1≤0
 ]  ]
```

Les quatre étages de l'unité sont construits par assemblage de ces différentes cellules, dont la répartition dépends du numéro de l'étage (voir Figure 5-9). Par exemple, dans le premier étage il y a une cellule 15, 14 cellules x et une cellule 0 tandis que dans le dernier étage il y a 8 cellules 15 et 8 cellules 0. Comme on pouvait s'y attendre, l'opération de non décalage coûte peu, le bit est simplement retransmis à la sortie.

Dans le processus de synthèse de ces cellules, afin de réduire la consommation et d'optimiser la latence pour les communications intra-étages, seuls les canaux d'entrées-sorties E et S des cellules sont pipelinés. Tous les canaux RIN, ROUT, LIN, LOUT utilisés pour effectuer les décalages sont synthétisés de manière combinatoire. L'unité de décalage est ainsi constituée globalement d'un pipeline à quatre étages, chaque étage étant constitué de logique combinatoire.

La partie contrôle de cette unité pipelinée consiste tout d'abord pour le premier étage à effectuer le décodage de l'opcode et à tenir compte du bit de signe de l'opérande OP1. Puis dans chaque étage *i*, en fonction de l'opération et du bit de décalage OP2[i], le processus de

contrôle transmet soit la commande de décalage appropriée aux 16 cellules, soit 16 commandes de non-décalage (cf. Figure 5-11).

```
*[  OP2?op2[3..1] ,                          -- 3 bits restants de décalage
    CMD_15?c15, CMD_X?cx, CMD_0?c0 ;         -- contrôle de l'étage précédent

    [ op2[1]=0 =>                        -- on ne décale pas
       C15!0, C14!0, C13!0, C12!0, C11!0, C10!0, C9!0, C8!0,
       C7!0, C6!0, C5!0, C4!0, C3!0, C2!0, C1!0, C0!0

    @ op2[1]=1 =>                        -- on décale
       C15!c15, C14!c15,
       C13!cx, C12!cx, C11!cx, C10!cx, C9!cx, C8!cx,
       C7!cx, C6!cx, C5!cx, C4!cx, C3!cx, C2!cx,
       C1!c0, C0!c0
    ] ,

    -- vers l'étage 4 suivant : propagation des 2 bits restants de décalage et du contrôle
    N_OP2!op2[3..2] , N_CMD_15!c15 , N_CMD_X!cx , N_CMD_0!c0
 ]  ]
```

Figure 5-11 : bloc de contrôle du deuxième étage du décaleur

Le contrôle est ainsi pipeliné, tout comme le décalage des données. Ce type de contrôle évite de pré-calculer toutes les valeurs de contrôle dès le premier étage. Chaque étage de pipeline possède ainsi son propre bloc de contrôle qui génère les valeurs de contrôle pour les cellules de l'étage en fonction de la valeur de décalage et transmet les informations à l'étage suivant. On obtient ainsi une architecture régulière avec l'assemblage de modules élémentaires : des cellules de propagation et un bloc de contrôle par étage. Ceci permet une très grande lisibilité fonctionnelle de la logique mise en œuvre.

L'architecture du décaleur permet ainsi de repartir contrôle et chemin de données dans un pipeline qui est équilibré entre les différents chemins. Le décaleur présente un temps de cycle faible de l'ordre de 2.5 ns, pour une latence totale de 2.5 ns. Ces performances permettent ainsi une utilisation intensive de l'Alu pour effectuer des décalages.

5.2.1.5. Conclusion

L'unité arithmétique et logique est donc composée de quatre sous unités distinctes. L'architecture démultiplexeur / multiplexeur adoptée pour celle-ci permet d'obtenir l'entrelacement des accès aux sous unités. On obtient ainsi des performances en débit élevées en moyenne sur cette unité si on suppose un enchaînement d'instructions s'adressant à des sous-unités différentes, à savoir deux instructions successives sur des sous unités distinctes, sauf pour le décaleur.

Les contraintes de débit étant en conséquence plus faible par unité, les quatre sous-unités peuvent être synthétisées de manière combinatoire, ce qui simplifie la logique d'acquittement, exception de l'unité de décalage en raison de sa structure régulière en 4 étages de pipeline. En conclusion, l'Alu obtient des performances en débit / latence qui montrent de fortes dépendances par rapport aux données, aux instructions et à la succession d'instructions.

5.2.2. Unité de Multiplication-accumulation

L'unité utilisateur d'Aspro est une unité de multiplication accumulation. Cette unité intègre un opérateur de multiplication 16x16 avec accumulation sur 40bits (voir paragraphe 5.4.2 pour son implémentation) et quatre registres d'accumulation 40 bits. Ces registres peuvent être interprétés suivant différents modes de lecture et envoyés sur 16 bits vers le banc de registre. La décomposition CHP qui suit s'attache à présenter comment s'ordonne la circulation des données dans cette unité utilisateur.

L'unité reçoit en entrée l'opcode de l'opération à effectuer, 2 opérandes sources et génère un résultat 16bits en sortie. Le numéro du registre d'accumulation est directement encodé dans l'opcode de l'opération. L'unité peut alors exécuter des instructions de multiplications (*mpy*), des multiplications accumulations (*macc*) ou sauvegarder un résultat (*save_acc*). Afin de simplifier le matériel du multiplieur, l'instruction *mpy* sera effectuée comme un *macc* mais avec une entrée d'accumulation initialisée à zéro.

Le modèle CHP de l'unité peut s'écrire de la manière suivante (par soucis de simplification, la fonction d'évaluation du mode de lecture n'est pas détaillée) :

```
*[ OPCODE?opc ; no_acc:=opc[5..4];   -- lecture opcode et numéro de registre
   [ opc=mpy or opc=macc =>
      OP1?op1 , OP2?op2 ,                          -- lecture opérandes
      [ opc=mpy  => acc:=0                         -- initialisation à 0
      @ opc=macc => acc:=Accus[no_acc]             -- lecture registre
      ];
      [ opc=signed   => acc := macc(acc,op1,op2,signed)
      @ opc=unsigned => acc := macc(acc,op1,op2,unsigned)
      ];
      Accus[no_acc]:=acc                           -- ré-écriture registre

   @ opc=save_acc =>                               -- sauvegarde accu
      acc:=Accus[no_acc] ;                         -- lecture registre
      -- calcul du résultat sur 16 bits :
      -- suivant mode (int/fp) décalage de un à gauche, puis test de saturation
      -- ou calcul d'arrondi, et envoi du bon tronçon L/M/U/R
      res :=f(acc,opc) ; S!res                     -- envoi résultat
 ] ]
```

A partir de ce premier modèle CHP, nous allons définir les différentes ressources matérielles nécessaires : registres et opérateurs. Chaque registre d'accumulation peut donc être écrit (*mpy*), lu et écrit (*macc*) ou uniquement lu (*save_acc*). Afin d'isoler ces registres, nous définissons un banc de registres qui contient les quatre accumulateurs. Ce banc de registres d'accumulation possède la vue externe suivante : un canal de lecture / écriture RW, le numéro de registre NO et les deux canaux de données : la donnée DR en lecture et la donnée DW en écriture. Ce banc de registres pourra s'implémenter de la même manière que le banc de registre principal d'Aspro (voir paragraphe 5.3).

En sortie de ces registres, un multiplexeur est nécessaire pour propager le contenu DR d'un registre soit vers le bloc de sortie, soit vers l'opérateur Macc ou bien générer un zéro dans le cas d'une multiplication. L'opérateur de multiplication-accumulation possède quatre entrées : les opérandes OP1 et OP2 16-bits, l'entrée d'accumulation 40-bits, et une entrée de contrôle

signé / non-signé. Il envoie directement son résultat en sortie vers les registres. Le bloc de sortie qui évalue les différents modes de lecture possède trois entrées de contrôle (entier / fixed-point, saturation / no-saturation et bits-de-poids-faible / bits-médian / bits-de-poids-forts / arrondi), la valeur d'accumulation en entrée. Il envoie alors son résultat sur 16-bits vers le banc de registres du processeur.

Ces différents processus s'écrivent de la manière suivante :

```
Reg_Accus :                          Mux :
*[ RW?rw, NO?no ;                    *[ C-MUX?cmd ;
   [ rw=0 => DW?Accus[no]               [ cmd=0 => Acc-Input!0
   @ rw=1 => DR!Accus[no]               @ cmd=1 => DR?x; Acc-Input!x
 ] ]                                    @ cmd=2 => DR?x; Acc-result!x
                                      ] ]

Macc :
*[ OP1?op1, OP2?op2, Acc-Input?acc, C-SIGN?sign ;
      [ sign=0 => DW!macc(acc,op1,op2,unsigned)
      @ sign=1 => DW!macc(acc,op1,op2,signed)
 ]    ]

Bloc de sortie :
*[ C1?c1, C2?c2, C3?c3, Acc-result?acc ; S!f(acc,c1,c2,c3) ]
```

La décomposition du modèle initial de l'unité consiste à extraire ces différentes ressources. Le processus initial n'a plus qu'à effectuer le décodage de l'opcode et générer les commandes de contrôle des différentes blocs. On obtient alors le processus de contrôle suivant :

```
*[ OPCODE?opc ; no_acc:=opc[5..4];    -- lecture opcode, numéro de registre
   [ opc=mpy or opc=macc =>
      C-SIGN!opc[0],                                  -- signe du macc
      [ opc=mpy  =>    RW!0, NO!no, C-MUX!0           -- init. 0, écriture
      @ opc=macc =>    RW!1, NO!no, C-MUX!1 ;         -- lecture registre (i)
                       RW!0, NO!no                    -- puis ré-écriture
      ]
   @ opc=save_acc =>                                  -- sauvegarde résultat
      RW!1, NO!no, C-MUX!2,                           -- lecture registre
      C1!fp_int,                                      -- fixed point / integer
      C2!sat_nosat,                                   -- saturation
      C3!lmur                                         -- L/M/U/Round
 ] ]
```

Ce processus de contrôle n'est pas directement synthétisable tel que défini dans le chapitre 3. En raison de l'opérateur séquentiel « ; » (i), la garde de lecture / écriture successive pour l'instruction *macc* doit être décomposée sous forme d'une machine à état. Comme présenté dans le chapitre 3, seule cette garde est modifiée avec la lecture d'une variable d'état et sa mise à jour. Il est ainsi nécessaire de rajouter un couple de canaux CS / NS pour lire / écrire localement cet état. Le code CHP est relativement évident, voici la modification à effectuer sur cette garde, les autres gardes ne changeant pas :

```
@ #OPC=macc => CS?cs; [cs=0 => RW!1, NO!no, C-MUX!1, NS!1   -- lecture registre
                       @cs=1 => RW!0, NO!no, NS!0, OPC?     -- écriture, aquit. OPC
@ #OPC=...             ]
```

On obtient ainsi un ordonnancement des différentes tâches de l'unité de multiplication-accumulation qui est contrôlé par un unique décodeur (Figure 5-12). Chaque ressource matérielle est extraite par décomposition et se réduit à des tâches matérielles simples. Les données circulent dans l'unité : du registre vers l'opérateur de multiplication-accumulation puis réécriture dans le registre ou du registre vers le bloc de sortie. Le séquencement, vu du bloc de contrôle, est uniquement nécessaire lorsqu'on fait appel à la même ressource matérielle. Les canaux de contrôle RW et NO implémentent alors le bon ordonnancement de l'action de lecture / ré-écriture au sein des registres. Pour le reste des canaux de contrôle, toutes les commandes sont émises en parallèle. Les différents blocs se synchronisent alors avec le passage de la donnée et consomment les commandes correspondantes successivement.

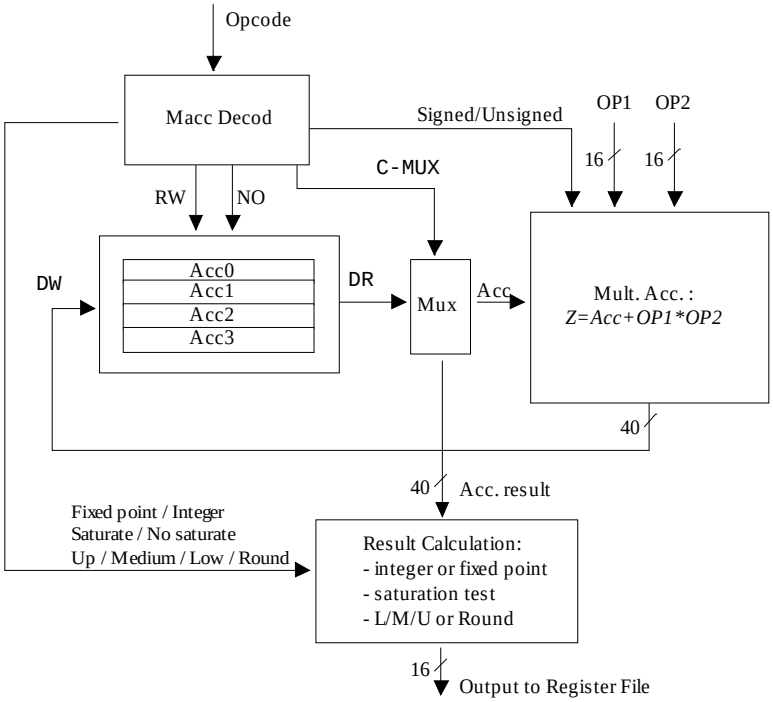


Figure 5-12 : Unité de Multiplication Accumulation

Cette unité à priori complexe en terme de séquencement en raison des registres propres à l'unité s'implémente ainsi de façon efficace avec une simple description flot de donnée de l'architecture. Le schéma bloc et son fonctionnement correspondent à l'image de l'algorithme fonctionnel.

A titre de remarque, une autre solution de décomposition serait possible. Comme pour le banc de registre du processeur, on pourrait imaginer un mécanisme de ré-écriture au sein de chaque registre (paragraphe 5.3). Le séquencement serait alors effectué dans le registre, et non dans le bloc de contrôle qui générerait en une seule fois une commande de lecture-écriture et le numéro du registre. Cette approche n'a pas été choisie pour deux raisons. Tout d'abord, il n'est pas vraiment nécessaire d'avoir dans cette unité utilisateur un fort taux de parallélisme, comme pour l'ensemble du chemin de données du processeur. De plus, le séquencement coûte

peu à cet endroit car ne modifie qu'une garde dans le processus de contrôle, où toutes les informations sont disponibles localement (il n'y a pas besoin de comparer les numéros de registre, car on ré-écrit par défaut celui qui est lu).

D'une manière générale, on peut noter que l'unité utilisateur peut ainsi avoir n'importe quel type de séquencement en interne. A partir du moment où les entrées-sorties de l'unité son respectées, celle-ci peut s'insérer directement dans le reste du processeur, sans que celui-ci ait une quelconque connaissance à priori de son fonctionnement interne et de ses performances locales. Ceci est bien sûr possible grâce au mode de fonctionnement asynchrone modélisé par les canaux de communications : synchronisation et échange de données.

L'opérateur Macc implémenté est une unité relativement lente (voir paragraphe 5.4.2) comparé au reste de l'unité. Celui-ci est simplement instancié dans l'unité, le mode de fonctionnement asynchrone garantissant la correction fonctionnelle de l'ensemble. Afin de ne pas ralentir le reste du processeur (via l'interface bus), il suffit de rajouter un étage de pipeline (Fifo) sur les opérandes OP1 et OP2 pour les mémoriser et relâcher la contrainte de débit sur ceux–ci. Ainsi dans l'hypothèse où l'instruction *Macc* n'est pas utilisée à toutes les instructions, l'opérateur Macc effectue son calcul lentement sans pénaliser le reste du chemin de données du processeur (voir la routine de filtrage FIR paragraphe 4.4).

5.2.3. Unité de branchement

L'unité de branchement (BRU) exécute les instructions de branchements retardés (DBcc) ou non-retardés (Bcc), le chargement d'immédiat (Ldi), le calcul d'adresse relatif à PC (Lra) et les instructions de sauts (Djmp, Drti, Djsr, Dbsr).

Cette unité dialogue et échange des données entre le banc de registre, l'unité PC et le décodeur (voir Figure 5-13). L'unité peut avoir deux opérandes en entrées : OP1 et OP2. L'opérande OP1 correspond toujours au contenu d'un registre tandis que l'opérande OP2 peut soit correspondre au contenu d'un registre, soit à un immédiat transmis via l'interface bus (pour les instructions Dbcci/Bcci/Lra/Ldi).

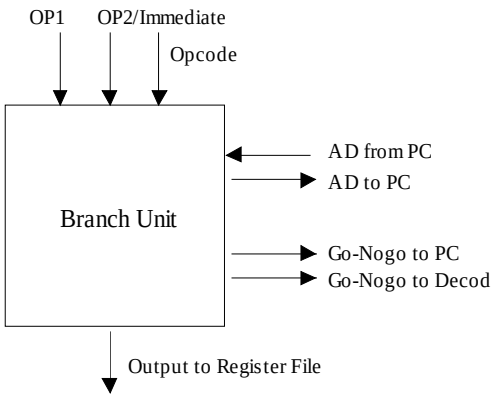


Figure 5-13 : Vue externe de l'unité de branchement

Pour un saut (Djmp, Djsr), le compteur de programme PC est mis à jour avec le contenu d'un registre. Pour un branchement à un sous-programme (Djsr, Dbsr), le PC est sauvegardé dans un registre. Pour les instructions de branchement (Bcc, DBcc), suivant la valeur des deux

opérandes OP1 et OP2 et du code condition, une information du type branchement pris / branchement non-pris (dénommé *gonogo*) est calculé par l'unité et est envoyé à l'unité PC et / ou au décodeur (voir paragraphe 5.5 pour le fonctionnement des instructions de branchement au sein de la boucle de *Fetch*).

De manière fonctionnelle, l'unité de branchement peut se modéliser ainsi :

```
*[ OPCODE?opc[3..0] ;
   [ opc[3]=0 =>                                    -- Bcc & DBcc
        OP1?op1, OP2?op2 ;
        [ opc[2..1]=00 => gonogo := seq(op1,op2)   -- cc=EQ
        @ opc[2..1]=01 => gonogo := sneq(op1,op2)  -- cc=NQ
        @ opc[2..1]=10 => gonogo := slt(op1,op2)   -- cc=LT
        @ opc[2..1]=11 => gonogo := sltu(op1,op2)  -- cc=LTU
        ] ;
        [ opc[0]=0 =>              -- Bcc : Non Delayed Branch
             GONOGO_PC!gonogo , GONOGO_DEC!gonogo
        @ opc[0]=1 =>              -- DBcc : Delayed Branch
             GONOGO_PC!gonogo
        ]
   @ opc=1110 => OP1?op1; toPC!op1                  -- Djmp/Drti
   @ opc=1111 => [OP1?op1; toPC!op1], [fromPC?pc; S!pc]  -- Djsr
   @ opc=110x => fromPC?pc; S!pc                         -- Dbsr
   @ opc=100x => OP2?op2, fromPC?pc; S!(op2+pc)          -- Lra
   @ opc=101x => OP2?op2; S!op2                          -- Ldi
  ] ]
```

Ce modèle CHP fonctionnel montre que l'unité de branchement a peu de calcul effectif outre les calculs de branchement et une addition pour l'instruction Lra. Son rôle principal est bien de synchroniser le chemin de données du microprocesseur avec la boucle de fetch et donc d'échanger des données (adresses, valeurs ou contrôle) entre l'unité PC, le décodeur et le banc de registres.

- **Décomposition du chemin de données de l'unité**

L'objectif de la décomposition CHP est de traiter indépendamment chaque canal d'entrée-sortie de l'unité. Les ressources de calcul nécessaires dans cette unité sont un additionneur pour effectuer l'instruction Lra et un bloc pour exécuter les calculs de branchements qui utilise deux comparateurs (voir paragraphe suivant). Le reste des opérations consiste simplement à transmettre une(des) entrée(s) vers une(des) sortie(s). Le but est alors d'envoyer les opérandes OP1, OP2, PC vers les bons opérateurs et de fabriquer les canaux de sortie à partir du résultat de ces unités. Comme pour l'unité arithmétique et logique, on obtient une architecture de type démultiplexeur / multiplexeur (Figure 5-14) où sont isolés les différents canaux de l'unité et les opérateurs.

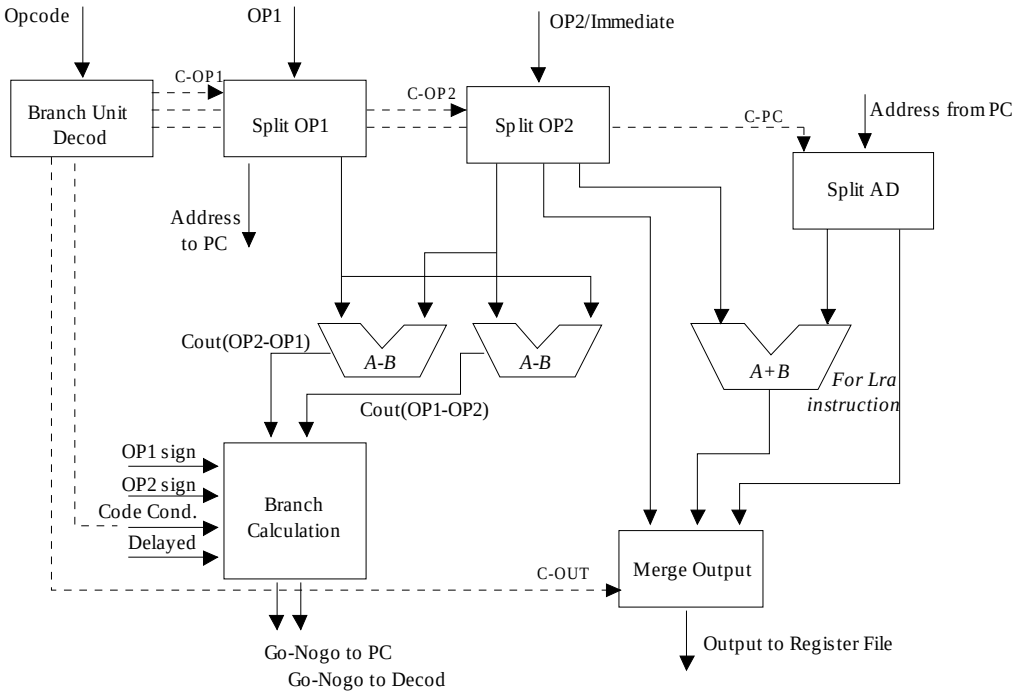


Figure 5-14 : Architecture de l'unité de branchement

Après extraction des différents canaux de données, le processus initial de l'unité se trouve alors réduit à décoder l'opcode et générer en parallèle les différents canaux de contrôle des multiplexeurs nécessaires pour faire transiter les données correspondant à chaque instruction. Si on appelle respectivement C-OP1, C-OP2, C-PC, C-Out les canaux de contrôle des processus démultiplexeur opérande OP1, démultiplexeur opérande OP2, démultiplexeur opérande PC, et multiplexeur canal de sortie S, on obtient le modèle suivant :

```
*[ OPCODE?opc[3..0] ;
    [ opc[3]=0 => C-OP1!0, C-OP2!0,            -- Bcc & DBcc
                  Code-Cond!opc[2..1], Delayed!opc[0]
    @ opc=1110 => C-OP1!1                  -- Djmp/Drti
    @ opc=1111 => C-OP1!1, C-PC!0, C-Out!0     -- Djsr
    @ opc=110x => C-PC!0, C-Out!0              -- Dbsr
    @ opc=100x => C-OP2!1, C-PC!1, C-Out!1     -- Lra
    @ opc=101x => C-OP2!2, C-Out!2             -- Ldi
 ]  ]
```

Les modèles des différents processus de multiplexage (Figure 5-14) ne sont pas donnés : ils sont équivalents à ceux utilisés par exemple dans la mémoire programme (paragraphe 5.1). Les valeurs des constantes utilisées pour les différentes commandes C-OP1, C-OP2, C-PC, C-Out sont arbitraires : elles correspondent simplement au numéro de la garde du démultiplexeur ou multiplexeur correspondant. Ces valeurs seront codées en multi-rail lors du processus de synthèse (cf. chapitre 3). Pour les instructions de branchement Bcc et DBcc, il est nécessaire d'envoyer au bloc de calcul de branchement le code condition (eq/ne/lt/ltu) ainsi que le code branchement retardé / non-retardé.

Ce processus assure ainsi la fonction de décodage et de contrôle de l'unité de branchement. Dans ce bloc de contrôle, toutes les valeurs de contrôle sont envoyées en parallèle. Celles-ci

sont alors consommées au fur et à mesure du passage des données dans l'unité. En conséquence, les données parcourent dans l'architecture un chemin à latence minimale entre entrées et sorties suivant l'instruction. Par exemple pour l'instruction Ldi, la valeur d'immédiat passe directement de l'entrée à la sortie, tandis que pour l'instruction Lra, la sortie est le résultat de l'addition des deux opérandes OP2 et PC.

- **Calcul du branchement**

Pour le calcul de branchement *gonogo*, l'objectif est d'obtenir une latence faible afin de pénaliser la boucle de *Fetch* du processeur le moins possible. Celle-ci est en effet en attente du résultat de branchement pour continuer à émettre des instructions. Pour cela, comme pour l'unité de comparaison (paragraphe 5.2.1), on implémente un calcul au plus tôt suivant le code condition de l'instruction et le signe des données.

Dans le cas où les bits de signes des deux opérandes sont différents, on peut déterminer directement le résultat du branchement. Dans le cas contraire, il faut effectuer des soustractions : OP1-OP2 pour les tests de comparaison, OP1-OP2 et OP2-OP1 pour les tests d'égalité (on pourrait en fait comparer les bits un à un mais cette méthode permet de réutiliser le soustracteur qui effectue OP1-OP2).

Afin de simplifier le matériel et de ne pas pénaliser la latence des cas critiques par ajout de contrôle, les deux soustractions sont toujours effectuées et ceci en parallèle du décodage des combinaisons de bits de signe. La génération du test de branchement *gonogo* s'écrit alors en CHP de la manière suivante (C1 et C2 sont respectivement les retenues sortantes des soustractions OP1-OP2 et OP2-OP1) :

```
*[ Code-Cond?cc , OP1[15]?sign1 , OP2[15]?sign2 ;
    [ sign1/=sign2 =>                        -- signes différents
         [ cc=eq  => GONOGO!0, C1?, C2?
         @ cc=ne  => GONOGO!1, C1?, C2?
         @ cc=lt  => GONOGO!sign1, C1?, C2?
         @ cc=ltu => GONOGO!sign2, C1?, C2?
         ]
    @ sign1=sign2 =>                    -- signes égaux
         [ cc=eq  => C1?c1, C2?c2 ; [c1=c2=1 => GONOGO!1
                                    @ c1/=c2 => GONOGO!0 ]
         @ cc=ne  => C1?c1, C2?c2 ; [c1=c2=1 => GONOGO!0
                                    @ c1/=c2 => GONOGO!1 ]
         @ cc=lt or cc=ltu => [C1?c1; GONOGO!not(c1)] , C2?
 ]  ]    ]
```

Le résultat de branchement *gonogo* est alors transféré à l'unité PC et / ou au décodeur suivant que l'instruction est retardée ou non :

```
*[Delayed?delayed , GONOGO?gonogo;
         [ delayed=0 =>              -- Bcc : Non Delayed Branch
              GONOGO_PC!gonogo , GONOGO_DEC!gonogo
         @ delayed=1 =>              -- DBcc : Delayed Branch
              GONOGO_PC!gonogo
         ]
```

On obtient ainsi un test de branchement qui est calculé au plus tôt suivant le code condition et le signe des opérandes. Pour un surcoût en matériel minimum, puisqu'on utilise simplement les bits de signes, la valeur du branchement est calculée à temps minimum en fonction à la fois des opérandes et du code condition.

- **Conclusion**

En performance, grâce à l'utilisation des bits de signe et aux propriétés de l'addition (la latence du soustracteur varie entre 0.3 ns et 1.5 ns), on obtient une latence du simple au double pour le calcul de branchement dans l'unité, soit entre 0.4 ns et 1.9 ns. Le débit de calcul de branchement est relativement lent puisque la soustraction est toujours effectuée, il faut en effet acquitter les retenues sortantes C1 et C2 même si elles ne sont pas utilisées, soit un temps de cycle de l'ordre de 3.5 ns (ceci pourrait être amélioré avec l'utilisation du protocole PCHB).

Les deux soustracteurs et le bloc de calcul de branchement sont synthétisés en combinatoire, seuls les canaux *gonogo* en sortie vers l'unité PC et le décodeur sont pipelinés. Ceci n'est pas contraignant puisque un code exécutable contient très rarement des branchements successifs (en moyenne un branchement toutes les six instructions [HENN 96]). Ici, seul la latence nous intéresse. Pour évaluer la pénalité de branchement (c'est à dire le temps de suspension de la boucle de *fetch* après un branchement), il faut rajouter à la latence de calcul de branchement le temps de lecture des opérandes dans le banc de registres, soit 2 ns, d'où au total une suspension entre 2.4 ns et 3.9 ns, soit la perte équivalente entre une et une instruction et demi après un branchement. On voit donc ici tout l'intérêt d'implémenter des opérateurs à temps de calcul minimum en fonction des données.

Le reste de l'unité qui effectue les calculs de saut est très rapide puisqu'elle est uniquement constituée de différents processus d'envoi. Exception faite de l'instruction Lra qui utilise un additionneur supplémentaire, toutes les autres instructions ont un temps de cycle de moins de 2.5 ns pour une latence faible de l'ordre de 1.1 ns. La traversée de l'unité de part en part se fait à latence minimale suivant l'instruction. L'unité de branchement permet ainsi de synchroniser et d'échanger des données entre l'unité PC et le banc de registre avec un coût en latence minimal.

5.2.4. Unité Load-Store

L'unité Load-Store exécute les instructions qui accèdent aux différentes mémoires : mémoire programme, mémoire données ou zone périphérique (Figure 5-15). L'unité contient de plus un registre dédié 24 bits découpé en deux registres : LdSt-Reg-Low pour les 16 bits de poids faible et LdSt-Reg-High pour les 8 bits de poids fort. Des instructions spécifiques de déplacement permettent d'échanger le contenu de ce registre en une seule fois avec la mémoire programme ou par tronçon avec le banc de registres. Le but est d'obtenir un interface de lecture / écriture simple avec la mémoire programme puisque celle-ci fait 24 bits de large, tandis que le reste du microprocesseur est sur 16 bits.

L'unité Ld-St doit tout d'abord décoder l'opcode de l'instruction puis envoyer une requête de lecture ou d'écriture vers l'espace mémoire concerné ou le registre LdSt-Reg. L'unité peut recevoir jusqu'à trois opérandes en entrées : op1 pour l'adresse mémoire (programme ou données) ou une donnée vers le registre, op2 pour la donnée à écrire en mémoire données, et en dernier opérande un immédiat 8 bits comme offset pour le calcul d'adressage indirect avec déplacement ou comme adresse périphérique. Cet immédiat est reçu pour toute instruction Ld-St - En raison d'un manque de place dans les opcodes ldst, le bit src-op de l'immédiat ne peut pas être codé, le décodeur d'Aspro ne sait donc pas sans décoder tous les opcodes de ldst si l'immédiat sera nécessaire -. Enfin, après émission d'une requête pour effectuer un accès en

lecture, l'unité Ld-St reçoit en retour une donnée de l'espace mémoire correspondant qu'il doit retransmettre sur sa sortie vers le banc de registre.

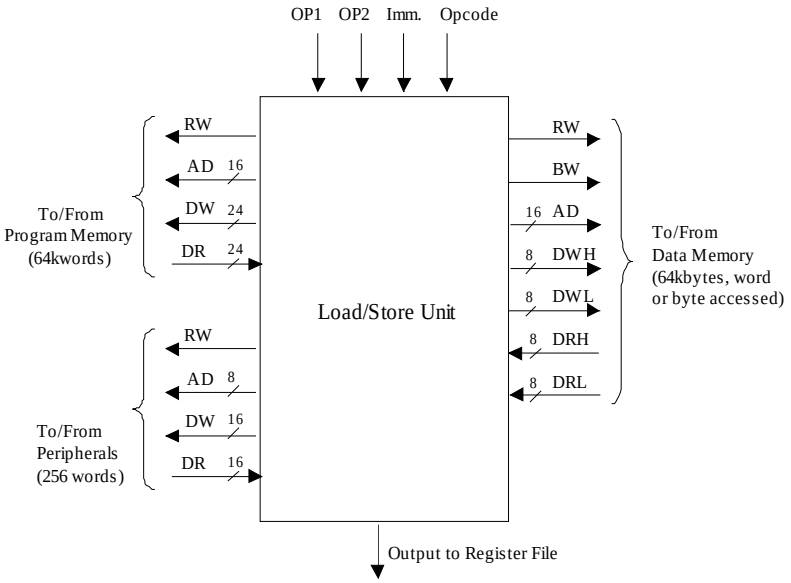


Figure 5-15 : Vue externe de l'unité Load-Store

- **Description des mémoires**

Pour exécuter ces instructions, l'unité Load-Store doit émettre les requêtes de lecture / écriture en concordance avec le fonctionnement de la mémoire correspondante : émission d'une adresse AD et d'une commande de lecture/écriture RW, émission d'une donnée DW en écriture ou réception d'une donnée DR en lecture. La mémoire programme et les périphériques ont le même type de vue externe à la taille des adresses et données près (cf. paragraphe 5.1 pour la mémoire programme, cf. paragraphe 5.6 pour les périphériques).

La mémoire données est plus complexe puisqu'elle doit gérer à la fois des accès par octet et par mot. Elle reçoit donc en supplément une commande byte/word BW et se modélise de la manière suivante en CHP (on rappelle que les accès par mot sont effectués à l'adresse paire immédiatement inférieure) :

```
Data Memory
*[ RW?rw , BW?bw , AD?ad ;
   [ rw=0 =>                                                  -- écriture
       [ bw=1 => DWL?mem-data[ad]                             -- octet
       @ bw=0 => DWL?mem-data[ad], DWH?mem-data[ad+1]         -- mot
       ]
   @ rw=1 =>                                                  -- lecture
       [ bw=1 => DRL!mem-data[ad]                             -- octet
       @ bw=0 => DRL!mem-data[ad], DRH!mem-data[ad+1]         -- mot
 ] ]   ]
```

Pour les accès par mot, les canaux DWH et DWL (ce sont des octets) sont utilisés en écriture, respectivement les octets DRH et DRL en lecture. Par contre pour les accès par octet,

seuls les canaux DWL et DRL sont utilisés. Ceci impose à l'unité Load-Store lors d'un accès par octet à la mémoire données de faire les actions suivantes : i) en lecture, faire une extension à zéro sur 16 bits de l'octet reçu ; ii) en écriture, évanouir les bits de poids forts et envoyer les bits de poids faibles à la mémoire.

La mémoire donnée est elle-même implémentée avec un schéma d'entrelacement comme celui présenté paragraphe 5.1 pour la mémoire programme. Pour cela, il est utilisé quatre fois 2 blocs de 8k octets (bancs d'octets pairs et impairs), soit 64k octets au total. Ceci permet d'obtenir un débit moyen élevé lors d'accès à des blocs consécutifs distincts (typiquement pour l'accès à des tableaux).

- **Modèle de l'unité Load-Store et décomposition**

Le mode d'adressage par défaut de la mémoire donnée est « indexé avec déplacement ». Ceci nécessite d'additionner offset et adresse de base avant d'envoyer l'adresse complète à la mémoire. Pour le reste des transferts de données à effectuer dans l'unité, les actions ne nécessitent pas de calculs supplémentaires.

Le modèle fonctionnel de l'unité Load-Store s'écrit en CHP de la manière suivante (par convention, les différents canaux sont suffixés par le nom de la mémoire, DM : Data Memory, PM : Program Memory, PH : Peripherals) :

```
Load-Store Unit
*[ OPCODE?opc , IMM?imm ;
   [ load-byte =>  RW-DM!1, BW-DM!1, [OP1?op1;AD-DM!(op1+imm)] ;
                   DRL-DM?x[7..0]; S[7..0]!x[7..0], S[15..8]!0
   @ load-word =>  RW-DM!1, BW-DM!0, [OP1?op1;AD-DM!(op1+imm)] ;
                   DRH-DM?x[15..8], DRL-DM?x[7..0]; S!x[15..0]
   @ store-byte => RW-DM!0, BW-DM!1, [OP1?op1;AD-DM!(op1+imm)],
                   [OP2?op2; DWL-DM!op2[7..0]]
   @ store-word => RW-DM!0, BW-DM!0, [OP1?op1;AD-DM!(op1+imm)],
                   [OP2?op2; DWH-DM!op2[15..8], DWL-DM!op2[7..0]]
   @ Ldpg Reg =>   RW-PM!1, [OP1?op1; AD-PM!op1] ;
                   DR-PM?x ; Reg_High:=x[23..16], Reg_Low:=x[15..0]
   @ Stpg Reg =>   x[23..16]:=Reg_High, x[15..0]:=Reg_Low ;
                   RW-PM!0, [OP1?op1; AD-PM!op1], DW-PM!x
   @ Move Reg, LdStRegL =>    S!Reg_Low
   @ Move Reg, LdStRegH =>    S[15..8]!0, S[7..0]!Reg_High
   @ Move LdStRegL, Reg =>    OP1?op1; Reg_Low:=op1[15..0]
   @ Move LdStRegH, Reg =>    OP1?op1; Reg_High:=op1[7..0]
   @ Movep @periph, Reg =>    RW-PH!0, AD-PH!imm, [OP1?op1;DW-PH!op1]
   @ Movep Reg, @periph =>    RW-PH!1, AD-PH!imm ; DR-PH?x ; S!x
]  ]
```

Par décomposition de ce processus, si on extrait les deux registres locaux LdSt-Reg-Low et LdSt-Reg-High en faisant apparaître les canaux RW et les canaux de lecture/écriture

correspondants, on observe que le processus load-store décode l'instruction reçue, génère des canaux de contrôle RW pour les mémoires ou les registres et transfert les opérandes vers/depuis les bonnes ressources mémoires. Ce modèle CHP, même s'il paraît complexe à première vue, reflète simplement le mode d'accès aux différentes mémoires et la dépendance de données entre les opérandes/canaux et les ressources mises en jeu. Ce modèle ne tient pas compte du fonctionnement interne et des performances de chaque espace mémoire.

Si on isole l'additionneur pour l'offset et les différents canaux d'entrées sorties de l'unité, l'architecture que l'on obtient est comme pour les autres unités une architecture de type "démultiplexeur / multiplexeur". Le processus initial de l'unité est réduit à un processus de décodage qui génère en parallèle les différents canaux de contrôle des processus qui font transiter les données. Le code CHP du processus de contrôle obtenu n'est pas présenté, il est similaire à celui de l'unité de branchement (paragraphe 5.2.3).

Les données circulent en empruntant un chemin le plus court suivant l'instruction. Le schéma bloc obtenu (voir Figure 5-16) est simplement l'image des dépendances fonctionnelles entre les données. Les temps de cycle des différents processus d'envoi sont élevés (2.5 ns), ce qui permet de distribuer à fréquence maximale des ordres de lecture / écriture vers les différentes ressources mémoires.

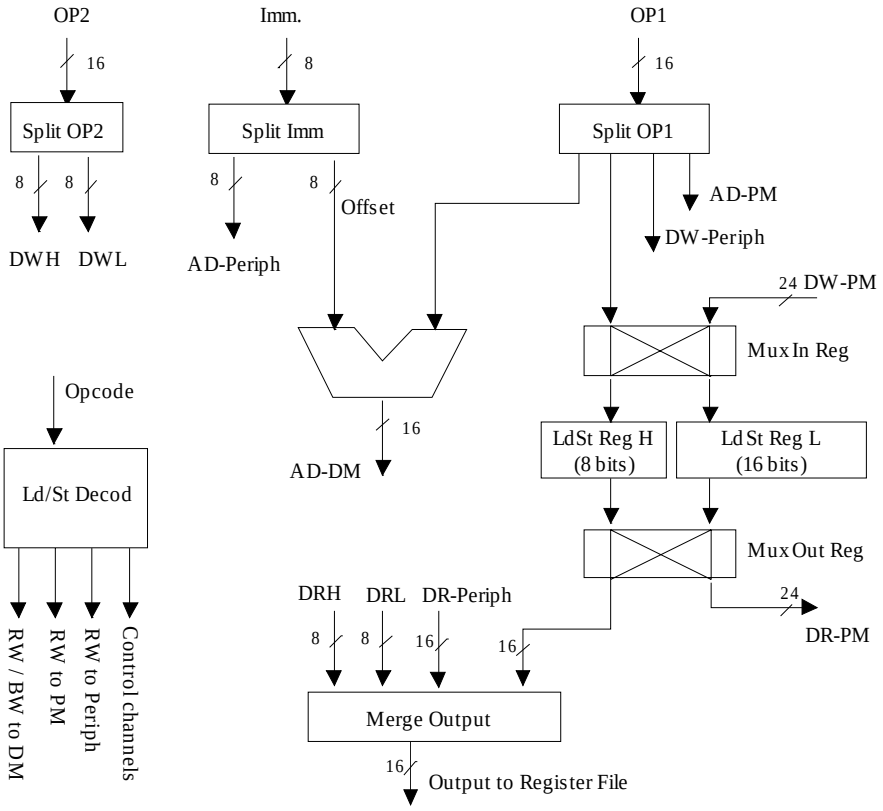


Figure 5-16 : Architecture de l'unité Load-Store

A titre de remarque, les démultiplexeur et multiplexeur utilisés ici sont plus complexes puisqu'ils doivent gérer en plus des extensions à zéro ou l'évanouissement de certains bits. Néanmoins, ces différents blocs sont synthétisables sans difficulté. Voici par exemple le modèles CHP du démultiplexeur de l'opérande OP2.

```
*[ C-OP2?cmd ;
  [ cmd=0 => OP2?op2[15..0]; DWL!op2[7..0]            -- écriture octet
  @ cmd=1 => OP2?op2[15..0]; DWL!op2[7..0],DWH!op2[15..8]  -- mot
 ] ]
```

- **Conclusion**

Par son architecture avec entrelacement, l'unité load-store autorise des dépassements entre instructions. En effet, si une instruction d'écriture suit une instruction de lecture sur une mémoire distincte (ou la même si cette mémoire l'autorise en terme de pipeline), l'écriture peut avoir lieu même si la donnée n'est pas encore disponible en sortie de la première mémoire. On obtient ainsi un entrelacement naturel des différentes requêtes tant qu'il n'y a pas de conflit sur une même ressource matérielle. C'est en particulier efficace dans le cas de l'accès en lecture à un périphérique non près : le processeur n'est pas bloqué tant que n'est pas effectuée une autre lecture ou un accès au résultat de cette lecture. En conclusion, cette unité effectue des requêtes de lecture/écriture vers les différentes ressources mémoires (données, programme, périphériques et registre local), les données circulent au plus tôt dans l'unité et les mémoires suivant leur disponibilité.

5.2.5. Conclusion sur les unités d'exécution.

Comme présenté dans les paragraphes précédents, le principe d'entrelacement utilisé dans la mémoire programme (paragraphe 5.1) est facile à implémenter en asynchrone, il est largement utilisé dans toutes les unités d'exécution d'Aspro. Dans le cas de la mémoire, l'architecture avec démultiplexeur / multiplexeur permet d'obtenir un débit moyen élevé en pipelinant les accès à des adresses successives distinctes. Dans les cas favorables, le pipeline de la mémoire a un taux d'occupation constant avec des débits et des latences équilibrées dans toutes les branches.

Pour les unités d'exécution du processeur, sans chercher plus de parallélisme en multipliant le nombre d'opérateurs, la décomposition CHP ne fait qu'isoler les différents canaux d'entrées sorties des blocs et les opérateurs nécessaires. On obtient ainsi d'une façon équivalente une architecture démultiplexeur / multiplexeur avec les opérateurs dans les différentes branches de calcul (Figure 5-17). Le schéma bloc est alors l'équivalent des dépendances de données : les données circulent entre blocs et se synchronisent par le contrôle. Pour ces structures fonctionnellement plus irrégulières, le pipeline est beaucoup plus élastique : il permet de tirer parti des variations de débit et de latence de chaque chemin ou opérateur.

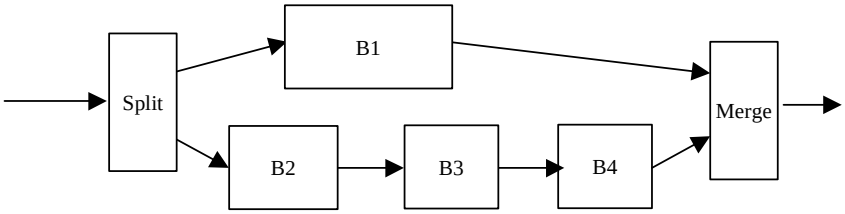


Figure 5-17 : architecture de type démultiplexeur / multiplexeur.

On obtient ainsi pour chaque unité :

- un débit élevé quand différentes branches sont successivement utilisées : il y a recouvrement.

- un débit égal à celui de l'opérateur dans le cas où la même branche est toujours utilisée.
- une latence à l'image du chemin utilisé.

Le chemin suivi par les données est donc toujours au plus court : suivant l'instruction, suivant les données elles-mêmes et suivant la disponibilité du matériel. Ce type d'architecture avec envoi au plus tôt permet d'utiliser le taux de parallélisme niveau instruction et donc de réduire les contraintes de débit par opérateur et la consommation. Le concepteur a ainsi le choix de porter ses efforts de conception sur les blocs critiques les plus couramment utilisés (additionneur, branchement, etc…).

Pour chacune de ces quatre unités d'exécutions, on a vu que l'on obtenait par décomposition un processus de décodage du code opératoire de l'instruction. Ces processus distribuent en parallèle des commandes vers les différents blocs : contrôle des démultiplexeur en entrée, des multiplexeur en sortie et contrôle des différents opérateurs. A chaque fois, ces commandes sont consommées suivant l'avancement des données dans l'unité. Le cas de l'unité utilisateur montre aussi un exemple de contrôle plus complexe avec lecture/ré-écriture des registres d'accumulation dans la même instruction. Au niveau global du microprocesseur, le décodage de ces codes opératoires est effectué dans les unités en parallèle de la lecture des opérandes dans le banc de registres car les codes opératoires sont générés dès la sortie du décodeur. Ainsi, les opérandes sont "consommés" immédiatement par l'opérateur dès leurs arrivées dans l'unité d'exécution. On observe ainsi que les données et leur contrôle associé circulent en parallèle dans le processeur et se synchronisent dans les unités.

Néanmoins, autant les démultiplexeur dispatchent les données en entrées et sont donc source de parallélisme, autant les multiplexeurs (re-)synchronisent les flux de données en sortie et donc séquencent les informations. Pour ces quatre unités, si les différentes branches ont des latences équivalentes, le pipeline se remplit correctement, il n'y a pas perte de performances. Au contraire, si les branches ont des latences très variables, une branche plus rapide doit attendre que la branche plus lente ait fini son calcul. Le multiplexeur de sortie est un point de passage obligé, il suspend le pipeline tant qu'il ne peut répondre à la première requête. En effet, la synchronisation par processus impose que les actions se déroulent dans l'ordre (et heureusement, c'est ce qui garanti le bon fonctionnement quelque soit la vitesse des opérateurs). Cette synchronisation peut donc être source de perte de performances. Par exemple dans l'unité load-store, une lecture sur un périphérique non-prêt pénalise la lecture qui suit en mémoire donnée. D'une manière générale, il y aura perte de performances lorsque la différence de latence entre les branches est plus grande que le débit du multiplexeur de sortie.

La répartition des instructions par unité tel que proposée dans Aspro offre un équilibre acceptable dans chaque unité. Cependant entre les différentes unités, on observe de fortes variations de latence. Afin de gagner en performances, le principe essentiel du processeur est d'émettre les instructions dans l'ordre, les distribuer vers les unités puis les exécuter en parallèle dans ces différentes unités et les terminer dans le désordre. Les résultats en sortie de chaque unité ne sont donc pas remis en ordre par un multiplexeur : ils sont envoyés indépendamment dans le banc de registres et donc au plus tôt et à priori dans le désordre entre unités. Le paragraphe suivant présente les conflits susceptibles de se produire et l'architecture du banc de registres choisie pour y remédier.

5.3. Banc de registres

Comme rappelé dans le paragraphe précédent, le principe essentiel du chemin de données d'Aspro est de paralléliser le plus possible les unités d'exécution. Celles-ci sont nourries d'instructions et d'opérandes en séquence par la boucle de fetch, elles s'exécutent en parallèles et écrivent indépendamment dans le banc de registres. Le but est de pouvoir tirer parti des différences de latences entre unités. En comparaison du pipeline des unités d'exécution ou de la mémoire, le pipeline entre unité est donc encore plus élastique. Il faut par contre résoudre les aléas de données [HENN 96], c'est à dire respecter les dépendances de données dans la séquence d'instructions entre numéros de registres lus et écrits.

Cas favorable	**LAE (Lecture Après Ecriture)**	**EAE (Ecriture après Ecriture)**
`Ldw reg0,(reg10)` `Add reg2, reg1, #1`	`Ldw reg0,(reg10)` `Add reg2, reg0, #1`	`Ldw reg0,(reg10)` `Add reg0, reg2, #1`

Figure 5-18 : trois cas d'enchaînement d'instructions

Dans le cas où on effectue un chargement mémoire (Ldw) dans le registre reg0, la Figure 5-18 présente trois exemples d'enchaînement d'instructions suivants les registres utilisés par l'instruction d'addition. Dans le cas favorable, comme l'instruction de chargement a une forte latence et qu'il n'y a pas de dépendances entre les registres, le but est de pouvoir lire le registre reg1, effectuer l'addition et écrire le résultat dans le registre reg2 avant la fin de la première instruction. On obtient ainsi entrelacement et dépassement entre les unités Alu et Ld/St. Par contre dans le cas d'un aléa de lecture après écriture, l'instruction d'addition doit attendre le résultat reg0 du chargement pour disposer de ses opérandes, les deux instructions doivent alors s'exécuter en séquence. Enfin, dans le cas d'un aléa d'écriture après écriture (la première instruction ne sert à rien mais ce code peut-être généré par un compilateur, le matériel doit le gérer), les écritures dans le banc de registres doivent être exécutées dans le bon ordre pour garantir la correction du code. Ces deux types d'aléas imposent donc un mécanisme de réservation des registres en écriture pour garantir la correction du code.

5.3.1.1. Vue globale du banc de registre

Pour chaque instruction[7], le banc de registres reçoit du décodeur les numéros de registres Op1, Op2, Dest, le nom Unit de l'unité concernée et un bit de Src-Op par numéro de registres pour valider ou non le numéro de celui-ci. L'objectif du Src-Op est de permettre au décodeur de toujours envoyer trois numéros de registres quel que soit le type d'accès : lecture, lecture/écriture, écriture. Ceci simplifie le décodeur : il n'a pas besoin de décoder le codeop instruction par unité, le banc de registres saura quels accès sont nécessaires.

[7] Ne sont concernées que les instructions effectuant au moins un accès à un registre. Les instructions Nop, Sleep, Dint, Eint ne sont donc pas concernées.

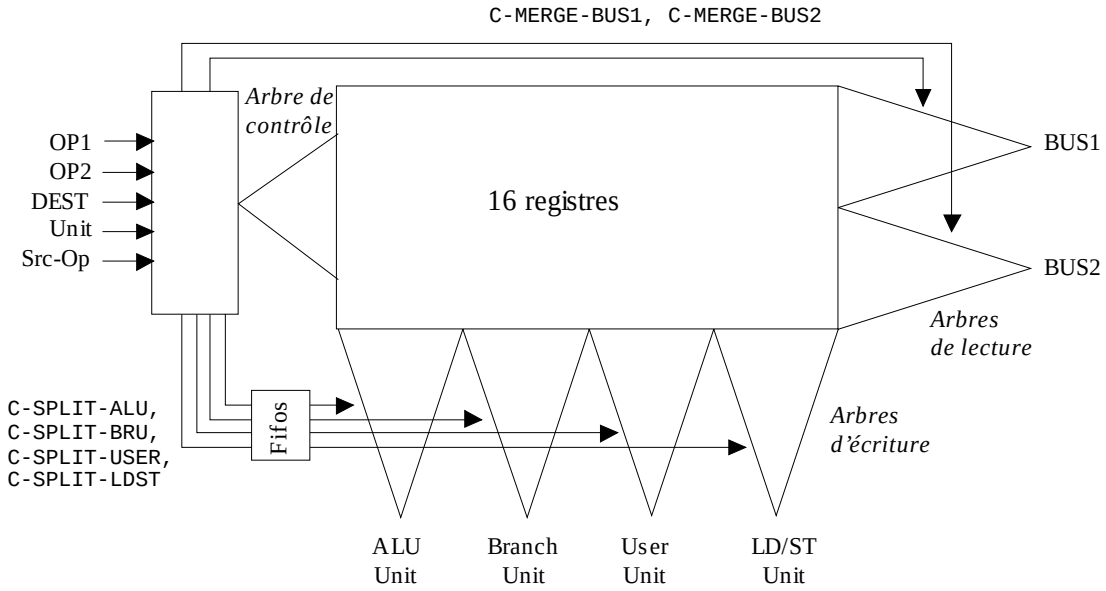


Figure 5-19 : Architecture du banc de registre.

Le banc de registres est constitué de deux ports de lecture Bus1 et Bus2 qui alimentent les unités en opérandes par l'intermédiaire de l'interface bus et de quatre ports d'écriture, un par unité (Figure 5-19). Afin de pouvoir écrire les résultats des unités dans un ordre quelconque, chaque port d'écriture est reçu par chacun des 16 registres, ce qui permet aux registres d'être écrits indépendamment les uns des autres par n'importe quelle unité.

Les arbres de lectures et d'écritures sont respectivement deux multiplexeurs 16 vers 1 et quatre démultiplexeurs 1 vers 16. La génération du contrôle de ces processus d'envoi est simple, il suffit d'envoyer le numéro de registre concerné suivant la validité du bit Src-Op correspondant et de l'unité concernée. On écrit :

Génération du contrôle des multiplexeurs de lecture (i ∈ <1:2>)

```
*[ Src-Op[i]?src, OPi?opi;
  [ src=0 => skip                          -- pas de lecture
  @ src=1 => C-MERGE-BUSi!opi              -- lecture sur BUSi
 ]]
```

Génération du contrôle des démultiplexeurs d'écriture

```
*[ Src-Op[0]?src, DEST?dest, UNIT?unit;
  [ src=0 => skip                                  -- pas d'écriture
  @ src=1 => [ unit=alu  => C-SPLIT-ALU!dest       -- écriture Alu
             @ unit=bru  => C-SPLIT-BRU!dest       -- écriture Bru
             @ unit=user => C-SPLIT-USER!dest      -- écriture User
             @ unit=ldst => C-SPLIT-LDST!dest      -- écriture LdSt
 ]]          ]
```

Afin de pouvoir écrire en continu dans le banc de registres sans suspendre le pipeline, les quatre canaux C-SPLIT-*Unit* doivent contenir suffisamment d'éléments mémoires. Il y a donc une *fifo* par contrôle de démultiplexeur en écriture (Figure 5-19), dont la profondeur est équivalente au nombre d'instructions mémorisables par l'unité concernée. Ceci permet de remplir au maximum le pipeline de toutes les unités lorsqu'il n'y a aucun conflit sur les registres concernés : pas de LAE, pas de EAE. Typiquement pour une succession de

chargement mémoire (load), les ordres d'écriture remplissent la *fifo* du canal C-SPLIT-LDST pendant que la mémoire donnée effectue les lectures.

L'interface bus se sera pas présenté en détail. Il est en fait simplement constitué de démultiplexeurs 1 vers 4 pour propager les valeurs des deux opérandes BUS1 et BUS2 vers les bonnes unités. Son contrôle provient du décodeur. Le démultiplexeur du deuxième opérande est plus complexe car il doit aussi propager ou évanouir les immédiats en provenance du décodeur, en particulier pour les immédiats de l'unité arithmétique (avec extension de signe) et les immédiats de l'unité de branchement (L'unité utilisateur n'a pas d'immédiat et l'unité LdSt reçoit son immédiat en tant que troisième opérande).

Avec ses deux ports de lecture et ses quatre ports d'écriture, l'architecture du banc de registres permet de lire et d'écrire au plus tôt, et donc d'entrelacer les écritures entre unités. Mais par ce matériel on voit aussi le conflit potentiel d'un LAE. Il se peut qu'un registre puisse répondre sur Bus1 ou Bus2 avant qu'il soit écrit par l'un des 4 ports d'écriture. Il faut donc pouvoir réserver le registre en écriture pour bloquer la lecture suivante.

5.3.1.2. Mécanisme de réservation

Contrairement à une machine RISC synchrone où un étage de pipeline est réservé pour effectuer la ré-écriture dans le banc de registres et où peut être constituée une table de réservation des registres (score-boarding [HENN 96]) pour permettre la suspension du pipeline d'exécution en cas de conflit, l'objectif ici est de rendre les informations de l'état des registres locales à chaque registre. *Aucune vue globale de l'état de réservation n'est strictement nécessaire*. En effet, grâce au principe des processus communicants, si l'accès en écriture du registre n'est pas terminé, la lecture de ce même registre ne pourra pas avoir lieu. Ce mécanisme est nécessairement déterministe. Les informations (on dira aussi les jetons) « ordre d'écriture » et « donnée à écrire » doivent se synchroniser à l'entrée du registre pour terminer l'écriture. Si il y a LAE, le jeton « ordre de lecture » ne peut pas passer avant car il a été émis après par l'étage de contrôle. Cet ordre de lecture du registre est nécessairement en attente de fin d'écriture : la lecture est naturellement suspendue.

Pour implémenter ce contrôle local à chaque registre, deux solutions étaient possibles. Soit le contrôle du banc de registres envoie uniquement des ordres de lecture/écriture aux registres concernés par l'instruction. Dans ce cas, il faut comparer les numéros de registres pour savoir combien de registres distincts sont concernés (par exemple dans l'instruction "add reg0, reg0, reg0" c'est le même registre qui reçoit trois ordres différents) et en conséquence correctement acquitter les registres ayant répondu. En terme de synthèse logique, ceci impose une logique d'acquittement complexe et lente : il faut pouvoir observer toutes les combinaisons 3 parmi 16, 2 parmi 16 et 1 parmi 16. Même avec l'aide de codage supplémentaire dans le code instruction (les comparaisons de numéros de registres seraient alors effectuées à la compilation), il faut toujours des arbres d'acquittement complexes. Cette solution est à priori la moins consommante puisque seuls les registres concernés sont activés, elle donne une bonne latence mais elle est trop lente en débit.

L'autre solution est d'envoyer des ordres de lecture/écriture à tous les registres, qui sont valides où non suivant les numéros de registres reçus. Dans ce cas tous les registres répondent : soit ils effectuent une lecture ou/et une écriture, soit ils ne font rien. La logique d'acquittement est simple puisque tous les registres sont activés. De plus, vis-à-vis du contrôle du banc de registres, il n'est plus nécessaire de comparer les numéros de registre, chaque

registre sait ce qu'il doit faire : rien, lecture (simple ou double) et/ou écriture. C'est une solution plus consommante mais plus rapide en débit pour une latence équivalente, c'est celle qui a été choisie.

5.3.1.3. Implémentation du registre

Chaque registre reçoit 3 bits de contrôle : lecture sur bus1 (r1), lecture sur bus2 (r2), écriture (w) et le numéro d'unité (unit). Chacun des 16 registres s'écrit alors de la manière suivante (*reg* est la valeur 16 bits du registre) :

```
*[ R1?r1, R2?r2, W?w, UNIT?unit;
    [ r1r2=00 => skip     -- no read
    @ r1r2=10 => Bus1!reg
    @ r1r2=01 => Bus2!reg
    @ r1r2=11 => Bus1!reg , Bus2!reg
    ] ;
    [ w=0 => skip         -- no write
    @ w=1 => [ unit=0 => Alu?reg
              @ unit=1 => Bru?reg
              @ unit=2 => User?reg
              @ unit=3 => Ldst?reg
 ]  ]         ]
```

Ainsi chaque registre possède deux ports de lecture, vers Bus1 et Bus2, et quatre port d'écriture, un par unité d'exécution. Ceci permet à chaque registre d'être totalement indépendant, chacun peut être lu ou/et écrit quelque soit le contrôle envoyé aux autres registres.

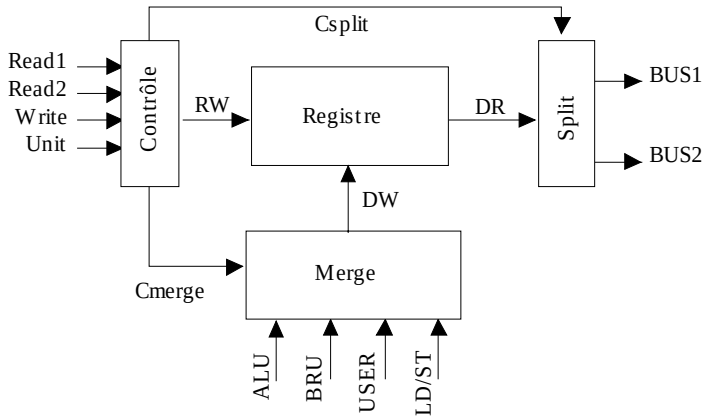
La génération de chaque bit de contrôle r1, r2 et w vers les 16 registres correspond au masque du numéro de registre décodé en 1 parmi 16 avec le bit Src-Op correspondant. L'arbre de décodage du banc de registres peut s'écrire de la manière suivante (ce processus est instancié trois fois pour Src-Op_i/Op_i/R_i respectivement égal à Src-Op2/Op2/r2, Src-Op1/Op1/r1 et Src-Op0/Dest/w) :

```
*[ Src-Op_i?src, Op_i?op_i ;
   [ src=0 => R_i[15..0]!0          -- tous les registres reçoivent un ordre nul
   @ src=1 => R_i[15..0]!(1<<op_i)  -- seul le registre op_i reçoit 1
 ] ]
```

Au niveau synthèse, la notation $1<<op_i$ est un raccourci d'écriture. La logique revient à effectuer la conversion des 4 bits du numéro de registre en un 16-rails. La logique d'acquittement est simple et performante : l'acquittement des entrées Opi / Src-Opi correspond à l'arbre d'acquittement des 16 registres car les commandes Ri sont toutes toujours activées.

Ce mécanisme semble coûteux en consommation mais simplifie énormément le matériel de décodage. Les décodages des trois numéros de registres sont donc indépendants, ils sont effectués en parallèles. Les différents bits de contrôle r1, r2, w se synchronisent alors à l'entrée de chaque registre, tous les registres répondent, ce qui permet une logique d'acquittement simple et rapide.

Il reste alors à décomposer le registre lui-même. En isolant les bus de lecture et les bus d'écriture, on obtient l'architecture suivante avec un processus de contrôle, le registre 16 bits et ses multiplexeurs en lecture / écriture (Figure 5-20).




```
Registre 16-bits
*[ RW?rw;
   [ rw=0 => DW?reg -- write
   @ rw=1 => DR!reg -- read
 ] ]
Contrôle du registre
*[ R1?r1,R2?r2,W?w,Unit?unit;
  [ r1r2w=000 => Skip
  @ r1r2w=001 => RW!0, Cmerge!unit
  @ r1r2w=010 => RW!1, Csplit!2
  @ r1r2w=011 => RW!1, Csplit!2 ; RW!0, Cmerge!unit (*)
  @ r1r2w=100 => RW!1, Csplit!1
  @ r1r2w=101 => RW!1, Csplit!1 ; RW!0, Cmerge!unit (*)
  @ r1r2w=110 => RW!1, Csplit!3
  @ r1r2w=111 => RW!1, Csplit!3 ; RW!0, Cmerge!unit (*)
]]
Split vers Bus1, Bus2
*[ Csplit?c, DR?x ;
   [ c=1 => Bus1!x
   @ c=2 => Bus2!x
   @ c=3 => Bus1!x, Bus2!x
 ] ]
Merge d'écriture
*[Cmerge?c; [ c=0 => Alu?x ; DW!x
            @ c=1 => Bru?x ; DW!x
            @ c=2 => User?x ; DW!x
            @ c=3 => Ldst?x ; DW!x
 ]          ]
```

Figure 5-20 : registre et son contrôle associé

En raison de la succession d'actions de lecture et d'écriture, le processus de contrôle obtenu ne peut-être directement synthétisé. Pour les trois lignes concernées (*), une décomposition en machine à états tel que présenté dans le chapitre 3 permet de se ramener à un modèle synthétisable. Les autres processus, y compris le registre, s'implémentent facilement.

Le processus de contrôle montre que la fonction lecture/écriture d'un même registre est totalement locale à ce registre. Que ce soit le décodeur d'Aspro, la boucle de fetch, ou même l'étage de contrôle à l'entrée du banc de registres, aucun de ces processus ne connaît l'état de ce registre enchaînant dans la même instruction une lecture/écriture. La machine à état qui permet de l'implémenter est simple et elle n'est de plus pas pénalisante car non activée dans le cas d'un accès classique : lecture (sur l'un ou les deux bus) ou écriture.

5.3.1.4. Conclusion

Entre la sortie des 4 unités et l'entrée des 16 registres, le chemin de données du processeur est donc constitué d'une architecture 4 démultiplexeurs vers tous les registres puis 16 multiplexeurs en entrée des registres. Ceci représente effectivement beaucoup de matériel. Dans l'optique d'une architecture séquentielle où les instructions seraient remises dans l'ordre à la sortie des unités, il serait nécessaire d'implémenter uniquement 1 multiplexeur des quatre unités vers le banc de registres, puis 1 démultiplexeur vers les 16 registres. Le surcoût en matériel est donc le prix du parallélisme. C'est le prix à payer pour pouvoir écrire dans le désordre à la sortie des unités et tirer parti des différences de latence entre unités.

Grâce à la sémantique des processus communicants, l'ensemble du banc de registres n'est constitué que de contrôleur locaux. Ceci permet d'implémenter la gestion des conflits d'aléas. Il n'y a aucune vue globale de l'état de réservation des registres. Tant qu'un registre réservé en écriture n'est pas lu (LAE) ou écrit (EAE), l'architecture du banc de registres permet de continuer à lire d'autres registres pour alimenter les unités d'exécution ou à écrire des résultats dans d'autres registres. Si conflit il y a, le flux d'instruction est bloqué en amont du registre concerné, l'exécution redémarre une fois l'action d'écriture terminée, et ce quelque soit la latence de cette écriture. L'exécution du programme peut ainsi se trouver naturellement suspendue si on accède à un périphérique non-prêt. C'est alors à la charge du compilateur / optimiseur de tirer parti des possibilités de performance du processeur en permettant avec un bon ré-ordonnancement des instructions le recouvrement de celles-ci. (cf. paragraphe 4.4).

5.4. Arithmétique et calcul en temps moyen

La principale caractéristique des circuits asynchrones qui est intéressante en arithmétique est leur propriété de calcul en temps moyen. Le temps de calcul d'un opérateur asynchrone est le temps pour que les sorties soient l'image fonctionnelle des entrées, c'est la latence directe. Ce temps peut varier considérablement suivant la valeur des opérandes. Par exemple, pour un additionneur à propagation de retenue, c'est la longueur de la plus longue chaîne de propagation de la retenue qui détermine le temps de calcul. Cette variabilité du temps de calcul exprime le parallélisme dynamique introduit par les données. Dans l'exemple de l'additionneur, plusieurs retenues se propagent en même temps à différentes positions. Contrairement au cas synchrone où le but est de minimiser le temps de calcul pire cas, nous allons nous intéresser à minimiser le temps de calcul moyen des opérateurs arithmétiques suivants : additionneurs dans le paragraphe 5.4.1 et multiplieurs dans le paragraphe 5.4.2.

5.4.1. Additionneur

Dans les microprocesseurs, l'addition est une opération très utilisée pour effectuer des calculs d'adresses, en tant qu'opérateur arithmétique mais aussi comme brique de base dans d'autres opérateurs. L'addition est ainsi couramment utilisée dans les opérateurs de multiplication, division [Has95], racine carrée ainsi que pour l'évaluation des fonctions élémentaires [MULL 97]. Pour le processeur Aspro, l'addition est pour l'instant utilisée en tant qu'élément de base, il serait cependant possible d'intégrer des fonctions plus complexes comme celles citées ci-dessus en tant que fonctions utilisateurs dans l'unité dédiée du processeur. Il est donc primordial de concevoir des additionneurs performants.

Dans ce paragraphe, nous allons présenter la conception des deux additionneurs asynchrones suivants : l'additionneur séquentiel (carry ripple adder) et l'additionneur à retenue bondissante (carry skip adder). A titre de comparaison, la description de ces additionneurs dans le cas synchrone est donnée dans [KORE 93], [MULL 89], [OMON 94].

Cette étude a été menée en collaboration avec Jean-Michel Muller et Arnaud Tisserand (Laboratoire d'Informatique et de Parallélisme de l'ENS Lyon) [TISS 97].

5.4.1.1. La cellule *Full-Adder*

Les additionneurs sont couramment constitués de cellules d'addition 1 bit, appelées *full-adder*. Cette cellule additionne deux bits avec une retenue entrante, elle génère en sortie un bit de somme et une retenue sortante.

Tout comme en synchrone, on utilise les propriétés de propagation/génération de la cellule full-adder. La propagation est usuellement définie en synchrone lorsque la cellule propage un signal à "1" sur la retenue. Voici comment est modélisé en CHP la dépendance de données entre retenue sortante et retenue entrante en fonction des bits d'entrées :

```
*[ A?a , B?b ;
    [ a=0 and b=0 => Cout!0, [Cin?c; S!c]      -- génération à zéro
    @ a=1 and b=1 => Cout!1, [Cin?c; S!c]      -- génération à un
    @ a /= b      => Cin?c; Cout!c, S!not(c)   -- propagation
 ]  ]
```

On parlera donc de génération (à 1 ou à 0) de la retenue sortante lorsque on peut déterminer sa valeur à la seule vue des deux bits A et B, et propagation de retenue lorsqu'il faut attendre la valeur de la retenue entrante pour déterminer sa valeur. Pour une distribution uniforme des entrées, on voit de manière évidente que la génération de la retenue sortante est possible dans la moitié des cas. Cette dépendance entre les données permet ainsi de rompre en moyenne la chaîne de propagation de retenue dans les structures d'additionneurs.

Ce modèle CHP du full-adder peut être synthétisé de différentes manières. Un schéma composé d'une porte complexe a été développé dans la bibliothèque de cellules asynchrones (voir chapitre 3). Cette porte complexe a été en particulier utilisée pour implémenter le multiplieur-accumulateur (voir paragraphe 5.4.2). De plus, ce modèle CHP peut être raffiné afin de séparer le calcul des informations de génération/propagation et le calcul des sorties. La cellule *full-adder* est alors composée des deux processus suivants :

GP: Generate/Propagate

```
*[ A?a , B?b ;
    [ a=0 and b=0 => GP!0
    @ a=1 and b=1 => GP!1
    @ a /= b      => GP!2
 ]  ]
```

SC : Compute Sum/Carry

```
*[ GP?g ;
   [ g=0 => Cout!0, [Cin?c; S!c]
   @ g=1 => Cout!1, [Cin?c; S!c]
   @ g=2 => Cin?c; Cout!c, S!not(c)
 ] ]
```

Ces deux processus sont alors synthétisables en cellules standard en utilisant la méthode présentée dans le chapitre 3.

5.4.1.2. Additionneur séquentiel

L'additionneur séquentiel (carry ripple adder) est l'additionneur le plus simple. Pour additionner deux nombres binaires de taille *n*, il est composé de *n* cellules de full-adder (Figure 5-21).

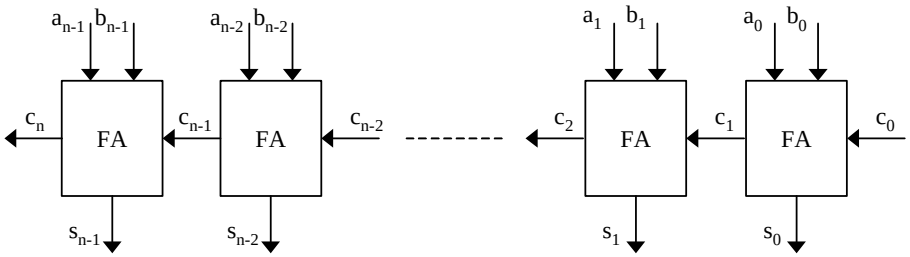


Figure 5-21 : Additionneur séquentiel

A partir d'une structure simple comme celle-ci on peut tirer les conclusions suivantes. Si on suppose le délai de la cellule d'addition unitaire et identique pour la retenue sortante et le bit de somme, on voit que :

- le temps pire cas correspond à la propagation de la retenue dans les n cellules, soit n délais.
- le temps le plus court correspond au cas où toutes les cellules génèrent et donc calculent en parallèle. Tous les bits de somme sont alors obtenus en 2 délais.
- le temps moyen de cette structure a été déterminé par [BGN46], il est en $O(\log_2 n)$ pour une distribution uniforme des entrées. Ce résultat est basé sur l'analyse de la longueur moyenne de la plus longue chaîne de propagation de la retenue qui est inférieure ou égale à $\log_2 n$.
- les bits de poids faible sont de plus toujours obtenus plus tôt : 1 ou 2 délais respectivement pour les 1er et 2ème bits.

L'écart type et le temps moyen de temps de calcul sont donnés pour des additionneurs de différentes tailles dans le Tableau 5-1 (en supposant une distribution uniforme des entrées et un délai unitaire par cellule full-adder).

Taille	Temps moyen	Ecart type
8	4.16	1.17
16	5.24	1.49
32	6.28	1.66
64	7.31	1.75

Tableau 5-1 : valeur moyenne et écart type du temps de calcul d'additionneurs séquentiels de différentes tailles [TISS 97].

La variabilité du temps de calcul dans cette structure est donc très importante (entre *2* et *n*) tout en offrant une moyenne et une dispersion relativement faible. On observe bien le parallélisme dynamique dans la structure due aux dépendances de données : les calculs s'effectuent en moyenne indépendamment les uns des autres alors qu'ils étaient à priori tous interdépendants.

Avec une structure simple comme celle-ci, la performance d'une implémentation synchrone serait très faible puisque nécessairement égale au temps pire cas, soit *n*. L'approche asynchrone permet de bénéficier d'un temps de calcul moyen très performant en $O(\log_2 n)$, ce qui est tout aussi performant, en théorie, que les meilleures implémentations d'additionneur synchrones. Brent [BREN 70] et Winograd [WINO 65] ont montré que les meilleurs additionneurs synchrones ont un temps de calcul en $O(\log_2 n)$ si on utilise un modèle de porte avec des entrances et des sortances bornées.

En conclusion, on peut considérer qu'un simple additionneur séquentiel est suffisamment performant en moyenne pour des valeurs de *n* petites (microcontrôleur 8 bits par exemple) en offrant une complexité et une consommation intéressante en comparaison d'additionneurs synchrones complexes (Carry Look Ahead, Carry Select).

5.4.1.3. Additionneur à retenue bondissante

L'additionneur à retenue bondissante (Carry Skip Adder) de deux nombres de *n* bits est basé sur un découpage de *b* blocs de *p* bits avec $n = b \times p$. Afin de rompre *les plus longues chaînes de propagation*, le principe est de propager directement la retenue entrante d'un bloc vers le bloc suivant si il y a propagation de la retenue à travers tout le bloc. Chaque bloc est alors composé d'un additionneur séquentiel de *p* bits et d'une logique qui permet de détecter si il y a propagation sur les *p* cellules d'addition de ce bloc (Figure 5-22).

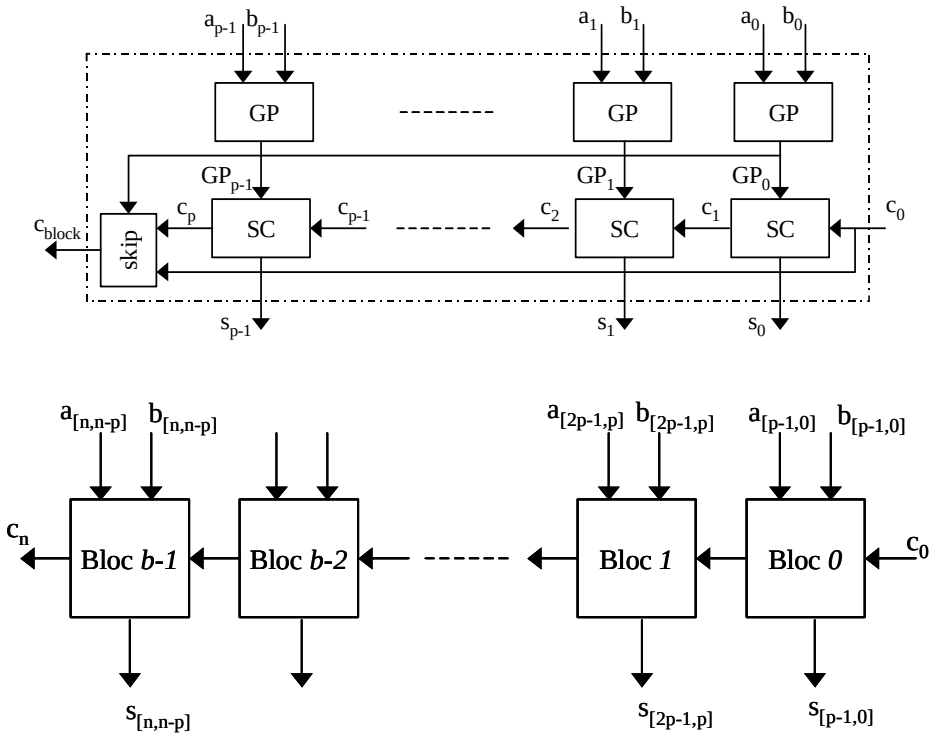


Figure 5-22 : additionneur à retenue bondissante n-bits :
description d'un bloc et de l'architecture composée de b blocs de p bits.

Afin de simplifier le matériel, comme la cellule de *full-adder* calcule déjà les informations de propagation/génération, on réutilise ces informations pour détecter si il y a ou non

propagation sur tout le bloc. La propagation de la retenue d'un bloc sur l'autre s'écrit en CHP de la manière suivante :

```
Skip: *[ GP[p..0]?g[p..0] ;
         [ g[p]=…=g[0]=2 => C[0]?c ; Cbloc!c    -- le bloc propage
         @ others        => C[p]?c ; Cbloc!c    -- le bloc génère
       ] ]
```

Chaque bloc de l'additionneur à retenue bondissante (Figure 5-22) est alors composé de *p* cellules d'addition et du processus de détection précédent, sachant que les cellules d'additions sont décomposées en un bloc de calcul des générations/propagations et un bloc de calcul des bits de somme.

L'architecture optimale correspond à un compromis entre la valeur de *b* et la valeur de *p*. En effet, plus *p* est petit, plus la probabilité qu'il y ait propagation sur le bloc est grande. Par contre, plus *p* est grand, plus les bonds de retenue sont grands et donc plus le calcul est rapide. De plus, ce compromis dépend bien sûr aussi des temps de calcul respectifs entre la cellule *full-adder* (GP et SC) et la logique de détection (Skip) qui permet d'effectuer les sauts.

En synchrone, on montre que la taille optimale des blocs est $p = \sqrt{n}$ pour un découpage régulier [LB61]. En asynchrone, le but est non d'optimiser le temps pire cas, mais le temps moyen ainsi que son écart type. L'écart type est un paramètre important puisque il permet de resserrer la distribution du temps de calcul au plus près de sa moyenne. Un opérateur avec une distribution resserrée du temps de calcul sera globalement plus performant qu'un opérateur avec un temps moyen faible et un écart type important. Pour ces raisons, et dans l'hypothèse où les temps de calcul entre le *full-adder* et la détection de propagation sur le bloc sont équivalents, [TISS 97] a montré que la taille optimale des blocs est donnée pour $p = 2$ et qu'avec ce découpage optimal, l'additionneur à retenue bondissante présente un temps de calcul moyen théorique en √(log(n)) (Tableau 5-2), soit plus performant que les meilleurs additionneurs synchrones.

Taille	Temps moyen	Ecart type
8	4.53	0.71
16	5.16	0.79
32	5.68	0.85
64	6.19	0.91

Tableau 5-2 : valeur moyenne et écart type du temps de calcul pour des additionneurs à retenue bondissante [TISS 97]

En comparant les chiffres obtenus pour l'additionneur séquentiel (Tableau 5-1) et l'additionneur à retenue bondissante (Tableau 5-2), on observe que le gain en temps de calcul moyen devient appréciable pour des additionneurs de grandes tailles (32 ou 64 bits) en raison des lois respectives en log(n) et √(log(n)). Par contre le gain en écart type est important pour toutes les tailles d'additionneurs. L'architecture de l'additionneur à retenue bondissante permet donc bien de casser les chaînes de propagation les plus longues, au prix d'une légère pénalité sur le temps de calcul moyen pour l'additionneur 8 bits.

Pour le processeur Aspro, si on mesure les performances après synthèse de la logique utilisée Figure 5-22, nous obtenons un temps de calcul légèrement plus important pour la logique de détection que pour la cellule d'addition. On observe donc une perte de latence dans la propagation d'un bloc sur l'autre dans les cas favorables (propagation sur le bloc ou génération sur le bit *p-1*). Afin de rendre ces cas moins fréquents, les additionneurs 16 bits d'Aspro sont constitués de 5 blocs de 3 bits plus une cellule d'addition. En conséquence, seules les plus longues chaînes de propagation de retenues sont coupées (pour 16 bits elles sont en moyenne de $\log_2 16=4$), les chaînes de propagations plus petites sont propagées dans le bloc par l'additionneur séquentiel.

Pour des additionneurs de plus grande taille (à priori à partir de 32 bits), il serait possible de rompre encore plus efficacement les plus longues chaînes de propagation en implémentant le saut de plusieurs blocs si tous ces blocs propagent. L'objectif est donc de diminuer le temps pire cas. La logique de propagation dans chaque bloc tiendrait compte de la propagation dans ce bloc et de un ou plusieurs blocs précédents. La retenue pourrait ainsi sauter directement plusieurs blocs en une seule fois si tous ceux-ci propagent. Néanmoins, ces cas de longues propagations sont rares et plus le nombre d'étages à sauter est important et plus la logique de détection est coûteuse. On va donc ralentir les cas les plus courants (et les plus rapides) et gagner peu sur les cas les plus rares (les plus lents). On voit que ce type d'optimisation rejoint plutôt une approche pire cas et n'est pas forcément intéressante pour des systèmes asynchrones.

5.4.1.4. Optimisation en débit des additionneurs

En niveau performance de ces additionneurs asynchrones, nous avons pour l'instant seulement présenté le temps de calcul moyen, à savoir la latence directe moyenne de la logique QDI. Pour cette logique, il faut aussi déterminer la logique d'acquittement et donc les performances en débit. Ici encore, même pour des opérateurs tel que des additionneurs, le concepteur a le choix d'optimiser ses critères de latence et débit lors du processus de synthèse.

Si les contraintes de débit sont faibles (par exemple dans l'unité arithmétique puisque intégrée dans une architecture autorisant l'entrelacement des instructions entre sous-unités), l'ensemble de la logique directe de l'additionneur est synthétisée en combinatoire. Les signaux d'acquittement des bits d'entrées de l'additionneur sont donc simplement fabriqués à partir d'un arbre d'acquittement des bits de sorties, soit une porte de Muller à 16 entrées pour un additionneur 16 bits.

Réciproquement, si les contraintes de débit sont plus fortes (par exemple les deux additionneurs de l'unité PC qui sont dans une boucle critique), l'additionneur peut-être pipeliné. Comme la structure de l'additionneur à retenue bondissante est régulière, on peut choisir de pipeliner le processus de détection de propagation (processus *skip)* par tranche ou toutes les deux tranches suivant le débit souhaité. La remise à zéro de la logique se fait en parallèle dans les différentes tranches, celle-ci commence dès que tous les bits d'une tranche ont été générés. Le temps de cycle se trouve réduit car l'arbre d'acquittement de chaque tranche est plus petit (il est de la taille du nombre de bits acquittés par tranche). Dans le cas de contraintes de débit encore plus fortes, on pourrait même imaginer pipeliner un additionneur séquentiel à chaque bit.

De plus, il faut remarquer que ces possibilités de pipeline n'influent pas ou très peu sur la latence directe et donc sur les performances en temps de calcul moyen de l'additionneur, car calcul et pipeline sont facilement combinés dans la même logique (voir chapitre 3).

Ces possibilités de pipeline des opérateurs sont donc très intéressantes. Cependant pour pouvoir tirer parti d'un tel potentiel de débit, il faut équilibrer les différents chemins du pipeline dans l'additionneur obtenu [Mar93]. Ceci consiste à mémoriser dans des étages de pipeline les bits de poids fort pendant que les bits de poids faible et la retenue du bloc suivant sont calculés, puis mémoriser les bits de poids faibles pendant que les bits de forts sont calculés à leur tour. L'additionneur présente alors un diagramme temporel déplié, qui est l'image de la dépendance de donnée de la retenue qui se propage dans celui-ci. Ce type de pipeline revient donc à décaler le calcul entre les différents bits (« *bit skew* »). Ceci peut se faire de deux manières : soit avec des étages de pipeline supplémentaires (donc une perte en surface et consommation), soit les étages de pipeline nécessaires sont distribués dans le reste de l'architecture, ce qui est malheureusement rarement le cas lorsque les bits des opérandes arrivent tous ensemble. Pour Aspro, seuls les deux additionneurs de l'unité PC seront pipelinés en raison des fortes contraintes de débit de la boucle de Fetch. Avec deux étages de pipeline, on obtient ainsi des temps de cycle de moins de 2.5 ns. D'une manière plus générale, les structures d'additionneurs très pipelinés sont particulièrement intéressantes pour des opérateurs itératifs en anneaux [ELHA 95].

5.4.1.5. Conclusion

La propriété de calcul en temps moyen des circuits asynchrones permet d'implémenter des additionneurs simples et performants. Comme constitués de simples cellules d'additions tirant parti des dépendances de données, les additionneurs séquentiels (carry ripple adder) offrent un temps de calcul moyen en $\log_2(n)$ ce qui est comparable, en théorie, aux meilleurs additionneurs synchrones. Afin d'améliorer les performances, les additionneurs à retenues bondissantes rompent les plus longues chaînes de propagation de retenue, pour un surcoût en complexité faible. Ceci permet de réduire à la fois le temps de calcul pire cas, le temps de calcul moyen qui est en $\sqrt{(\log_2(n))}$ et la distribution des temps de calcul.

Les temps de calcul que nous venons de présenter correspondent aux latences directes des additionneurs. Des compromis en terme de pipeline permettent de plus d'optimiser le débit et la consommation suivant des contraintes systèmes.

Pour conclure cette étude, il aurait été intéressant d'effectuer la conception d'additionneurs synchrones afin de comparer les architectures synchrones et asynchrones précisément en termes de vitesse / consommation / complexité. Les chiffres présentés ici, $\log_2(n)$ et $\sqrt{(\log_2(n))}$, sont purement théoriques mais donnent une idée des tendances. Cette étude doit bien sûr aussi estimer, suivant des contraintes systèmes, si les performances en temps moyen en asynchrone sont suffisantes par rapport à une garantie de temps pire cas (synchrone ou asynchrone).

5.4.2. Multiplieur - accumulateur.

L'unité utilisateur présentée paragraphe 5.2.2 intègre un opérateur de multiplication 16x16 avec accumulation sur 40 bits. Cet opérateur doit effectuer du calcul signé ou non signé et avec la contrainte suivante : l'entrée d'accumulation doit pouvoir arriver après les opérandes sans retarder le calcul, ceci afin de compenser le temps d'accès au registre d'accumulation. Le but est de commencer la multiplication suivant la disponibilité des opérandes et d'accumuler avec le contenu du registre en cours de calcul. Cet opérateur n'a à priori pas de contraintes de débit, il sera utilisé de manière sporadique. L'objectif est d'obtenir un opérateur peu complexe et peu consommant.

Nous allons tout d'abord présenter les algorithmes de multiplication signé et non signé puis différentes architectures possibles pour les implémenter.

- **Multiplication non-signée**

La multiplication de deux opérandes non-signés de tailles *n* s'écrit de la manière suivante :

$$A.B = \left(\sum_{i=0}^{n-1} a_i.2^i\right).\left(\sum_{j=0}^{n-1} b_j.2^j\right) = \sum_{(i,j)\in[0;n-1]^2} a_i b_j \cdot 2^{i+j}$$

Ceci nécessite l'addition de *n* x *n* produits partiels $a_i b_j$, soit *n* nombres de *n* bits. Le bit d'ordre *2n-1* du résultat est alors donné par la retenue sortante de l'addition. La Figure 5-23 présente en exemple la multiplication non signée de deux nombres de 4 bits.

			a_3b_0	a_2b_0	a_1b_0	a_0b_0
		a_3b_1	a_2b_1	a_1b_1	a_0b_1	
	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
a_3b_3	a_2b_3	a_1b_3	a_0b_3			

Figure 5-23 : Multiplication non-signée de deux nombres de 4 bits

- **Multiplication signée**

La multiplication de deux opérandes signés de tailles *n* codés en complément à deux s'écrit de la manière suivante :

$$A.B = \left(-a_{n-1}.2^{n-1} + \sum_{i=0}^{n-2} a_i.2^i\right).\left(-b_{n-1}.2^{n-1} + \sum_{j=0}^{n-2} b_j.2^j\right)$$

Si on distribue le deuxième membre sur le premier membre et qu'on transforme le signe négatif de b_{n-1} en utilisant les propriétés du complément à deux, on obtient :

$$A.B = \left(-\bar{a}_{n-1}.2^{n-1} + \sum_{i=0}^{n-2} \bar{a}_i.2^i + 1\right).b_{n-1}.2^{n-1} + \sum_{j=0}^{n-2}\left(-a_{n-1}.2^{n-1} + \sum_{i=0}^{n-2} a_i.2^i\right).b_j.2^j$$

On note par $\overline{a_i}$ le complément à 1 de a_i. La multiplication correspond alors à l'addition de *n* nombres signés sur *2n* bits plus le terme b_{n-1} :

a_3b_0	a_3b_0	a_3b_0	a_3b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0
a_3b_1	a_3b_1	a_3b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1	
a_3b_2	a_3b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
$\overline{a_3}\ b_3$	$\overline{a_3}\ b_3$	$\overline{a_2}\ b_3$	$\overline{a_1}\ b_3$	$\overline{a_0}\ b_3$			
				b_3			

On voit que la duplication des bits de signes sur *2n* bits nécessite à priori l'addition de termes en supplément des *n* x *n* produits partiels $a_i b_j$ initiaux. La méthode de réduction du bit de signe [ROO 86] permet de supprimer l'addition de ces termes redondants, on obtient alors l'algorithme de multiplication signée suivant (Figure 5-24) :

		$\overline{a_3 b_0}$	a_3b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0
		$\overline{a_3 b_1}$	a_3b_1	a_2b_1	a_1b_1	a_0b_1	
	$\overline{a_3 b_2}$	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
$\overline{\overline{a_3}\, b_3}$	$\overline{a_3}\, b_3$	$\overline{a_2}\, b_3$	$\overline{a_1}\, b_3$	$\overline{a_0}\, b_3$			
				b_3			

Figure 5-24 : Multiplication signée de deux nombres de 4 bits

- **Calcul des produits partiels**

Afin de pouvoir utiliser le même matériel pour effectuer la multiplication signée et non signée, la génération de certains produits partiels doit être conditionnée par le mode de calcul. Par exemple, la génération des produits partiels de la dernière ligne $i \in [0,n-1], j=(n-1)$ s'écrit de la manière suivante :

```
*[ SIGN?signed, A[i]?a , B[j]?b ;
 [ signed=0 =>   [ a=1 and b=1 => PP[i][j]!1      -- unsigned
                 @ a=0 or b=0  => PP[i][j]!0 ]
 @ signed=1 =>   [ a=0 and b=1 => PP[i][j]!1      -- signed
                 @ a=1 or b=0  => PP[i][j]!0 ]
 ]               ]
```

Dans la suite, on supposera que les 256+18 produits partiels sont disponibles en parallèles et correctement générés suivant le mode de calcul signé ou non-signé. La multiplication revient alors à additionner 16 nombres de 17 bits, soit 32 tranches de calcul, une par bit de résultat, sans oublier l'accumulation du résultat avec un nombre de 40 bits. Selon les tranches de calcul, il y a donc de 2 à 18 bits de même poids à additionner. Le nombre maximum de bits à réduire est obtenu pour la tranches 16 avec 18 bits à réduire en raison du terme b_{n-1}, et les tranches 17 et 18 avec 17 bits à réduire.

- **Multiplieur de Braun**

Le principe du multiplieur de Braun [MULL 89] est une structure cellulaire parallèle très simple à base de cellules *full-adder*, c'est la traduction directe de l'algorithme appris à l'école (Figure 5-25). Toutes les tranches verticales calculent en parallèles, elles réduisent en séquence les différents produits partiels en assimilant à chaque niveau une retenue générée dans le niveau précédent.

Dans cette structure, l'entrée d'accumulation peut être ajoutée en bas de chaque tranche. La réduction des produits partiels commence ainsi avant l'arrivée de la valeur d'accumulation et ce jusqu'à la diagonale inférieure.

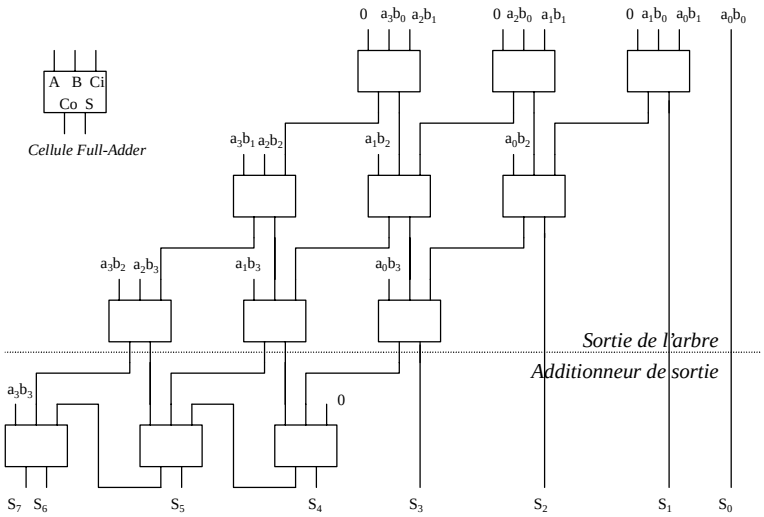


Figure 5-25 : Exemple de multiplieur de Braun 4 x 4

L'assimilation de *n* bits par tranche se fait avec *n-1* cellules de *full-adder*. Chaque tranche du multiplieur génère en bas de l'arbre un bit de somme et un bit de retenue qui peuvent ensuite être assimilés avec un additionneur rapide ou bien comme présenté dans la Figure 5-25 avec un additionneur à propagation de retenue.

En sortie de l'arbre, si on compare les temps de sortie par rapport à la tranche médiane, on observe que les bits de poids faibles sont toujours disponibles plus tôt et que les bits de poids forts peuvent sortir soit plus tôt, soit en même temps suivant les propagations/générations dans les tranches précédentes.

Si on considère le délai du *full-adder* unitaire, le multiplieur de Braun présente un délai constant de *n-1* pour la tranche médiane et un délai pire cas de *n-1* pour l'addition finale, soit un pire cas total de *2n-2*. Par simulation, [TISS 97] a montré que ce multiplieur présente un temps de calcul moyen d'environ *n+1.5* avec un écart type invariant d'environ 2.

En moyenne, on peut donc considérer que l'addition à la sortie de l'arbre ne propage quasiment pas : elle génère en moyenne tous ses bits de somme en 2.5 délais. Cette propriété du temps de calcul moyen de la dernière addition est donc propre à l'algorithme de multiplication, elle ne dépend pas de la manière de réduire les bits dans les tranches.

- **Arbre de Wallace**

Pour chaque tranche de calcul de l'arbre de multiplication, le principe d'un arbre de Wallace [Wal64] est de réduire les produits partiels en parallèles. Cette architecture permet de réduire les produits partiels avec un arbre logarithmique de cellules de *full-adder* et d'assimiler dans l'ordre les retenues entrantes de la tranche précédente.

L'assimilation de *n* bits se fait en $o(\log_2(n))$. En pratique, en raison de la gestion des retenues entre les tranches, 6 étages de *full-adder* permettent de réduire jusqu'à 19 opérandes. La Figure 5-26 représente un arbre de Wallace adapté pour 18 bits.

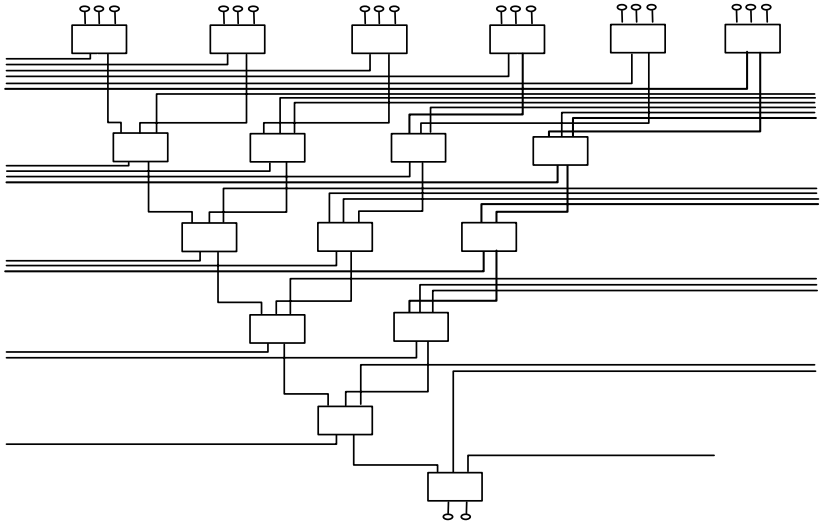


Figure 5-26 : Arbre de Wallace

Comme pour le multiplieur de Braun, chaque tranche génère un bit de somme et un bit de retenue qui doivent être ensuite additionnés. Avec un maximum de 6 délais au lieu de 17, ce multiplieur est donc plus rapide que le multiplieur de Braun. Néanmoins, il n'est pas possible d'avoir une entrée d'accumulation au plus tard dans l'arbre. Comme tous les produits partiels sont synchronisés en entrée de chaque tranche puis réduits en parallèles, on risquerait de suspendre le début du calcul dans l'arbre.

- **Arbre de Zuras Mac Allister**

L'arbre de Zuras Mac Allister [ZMA86] est un compromis entre un arbre séquentiel comme celui utilisé dans un multiplieur de Braun et un arbre parallèle comme celui de Wallace. Pour chaque tranche du multiplieur, le principe est d'effectuer en parallèle plusieurs branches d'additions séquentielles.

Les différentes branches sont équilibrées afin d'avoir entre chacune d'elles au plus un délai de cellule d'addition à leur point de rendez-vous (les cellules en noir dans la Figure 5-27). La construction de chaque tranche est effectuée de manière hiérarchique : au départ deux petites branches sont en parallèles, puis réduites et mises en parallèles d'une branche d'un délai supplémentaires, et ainsi de suite. Comme pour les deux structures précédentes, les retenues d'une tranche sur l'autre sont reparties de façon à ne pas ajouter de délais supplémentaires. Enfin, chaque tranche génère un bit de somme et de retenue qui devront être additionnés par la suite, soit par un carry-ripple adder ou par un additionneur rapide.

Grâce à la mise en parallèle de plusieurs branches, cet arbre permet de réduire plus de produits partiels qu'un multiplieur de Braun, il est donc beaucoup plus rapide. Cette architecture autorise de plus de répartir les produits partiels le long des branches et donc autorise l'entrée d'accumulation en bas de l'arbre (celle-ci peut être placée dans la cellule grisée).

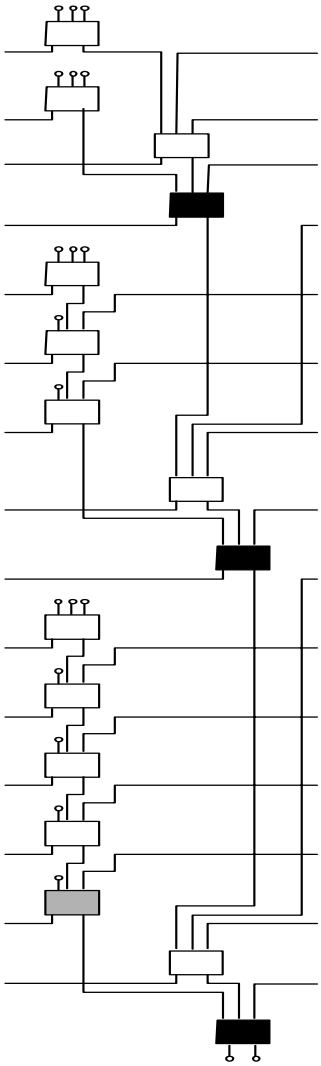


Figure 5-27 : Arbre ZM.

Pour additionner 18 bits (Figure 5-27), il faut un délai de 7 cellules de full-adder, soit un de plus que pour un arbre de Wallace. En fait, comme notre multiplieur 16x16 est carré, il y a seulement une tranche de taille maximale (la tranche 16) ce qui permet de supprimer des entrées de retenue sur celle-ci et donc un délai.

- **Implémentation du multiplieur-accumulateur**

L'arbre choisi pour le multiplieur-accumulateur est donc un arbre Zuras Mac Allister. On obtient les 32 bits de somme et de retenue en sortie de l'arbre en au plus 6 délais de full-adder, sachant que les 32 premiers bits d'accumulation ont été ajoutés au plus tard dans cet arbre. Il reste alors à effectuer l'addition des vecteurs sommes et retenues plus les 8 bits de poids forts d'accumulation, ce qui est fait avec un additionneur à retenue bondissante de 40 bits (Figure 5-28).

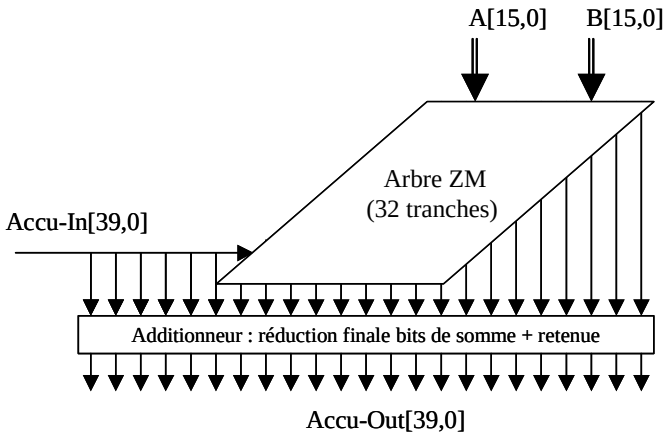


Figure 5-28 : Architecture du multiplieur-accumulateur

Comme présenté pour le multiplieur de Braun, le temps de calcul moyen de la dernière addition est de l'ordre de 2,5 délais de full-adder, soit un temps de calcul moyen du multiplieur-accumulateur de 8,5 délais, pour un temps de calcul pire cas de 6+16+8=30 délais. On voit ici tout l'intérêt d'une architecture asynchrone.

Les contraintes de débit sur cet opérateur étant faibles, cette structure est synthétisée à priori de manière toute combinatoire : l'acquittement des valeurs d'entrées (opérandes, accumulation, et contrôle de signe) est fabriqué à partir d'un unique arbre d'acquittement des 40 bits de sortie. Néanmoins, les cellules de full-adder de la bibliothèque (paragraphe 3.3) présentent des temps de propagation asymétriques entre temps de montée et temps de descente. On obtient ainsi une latence de calcul plus faible (ce qui est intéressant puisque cela détermine le débit de la boucle lecture/écriture dans l'unité utilisateur) que la latence de remise à zéro et donc un débit élevé (au moins la somme des deux). Afin de gagner en débit et de rééquilibrer débit et latence, l'architecture est pipelinée entre la sortie de l'arbre et l'additionneur rapide. Ceci permet de gagner en débit au détriment d'une légère perte de latence puisque la dernière addition ne peut commencer au plus tôt, les bits de poids faibles étant à présent synchronisés avec les bits de poids forts. Le Tableau 5-3 présente les chiffres de latence directe, latence de remise à zéro et temps de cycle pour les architectures combinatoire et pipelinée.

	Mult-Acc combinatoire	Mult-Acc pipeliné
Latence directe	2.2 ns	2.8 ns
Latence remise à zéro	3.5 ns	4.0 ns
Temps de cycle	7.7 ns	6.3 ns

Tableau 5-3 : performances du multiplieur-accumulateur (combinatoire et pipeliné)

Pour conclure, avec une structure d'arbre parallèle, l'ensemble des produits partiels d'une multiplication nécessite la traversée de tout le réseau d'addition, que ce soit pour les bits de sommes ou de retenues ; les parcours dans le réseau sont en conséquence équilibrés. Comme en synchrone, les structures de multiplieurs rapides et compacts offrent de bonnes performances. Une propriété intéressante néanmoins permet d'effectuer la dernière addition de l'arbre de réduction des produits partiels en un temps moyen fixe de 2.5 délais élémentaires alors que le pire cas est donné en *n*. Ceci est donc bénéfique à une approche asynchrone.

5.5. Boucle de Fetch : Décodeur et Unité PC.

Comme présenté dans le chapitre 4 sur le schéma d'ensemble de fonctionnement du processeur Aspro, la boucle de Fetch représente le contrôle du processeur : lecture des instructions en mémoire programme, décodage des instructions, envoi d'ordres au chemin de données puis mise à jour du compteur de programme PC. La boucle de Fetch est donc composée des trois éléments suivants : l'unité PC qui contient et met à jour le compteur programme ; la mémoire programme et son interface entre l'unité PC, l'unité Load-Store et le port A (mémoire programme externe) ; et enfin le décodeur instructions (Figure 5-29).

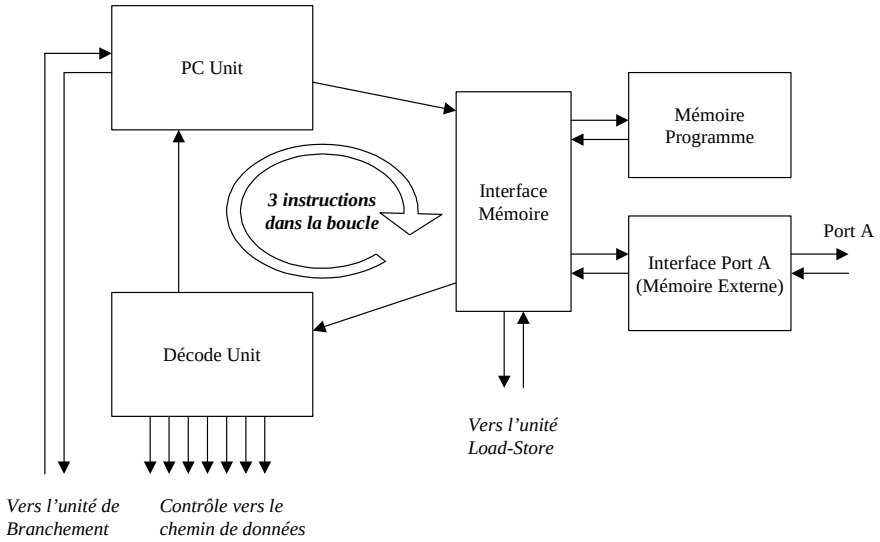


Figure 5-29 : Schéma de la boucle de Fetch du processeur.

Cette boucle auto-séquencée contient toujours trois instructions, Aspro ayant un schéma de branchement à deux « *delay slots* ». Ces trois instructions se suivent dans le pipeline de la boucle de fetch. Comme pour le chemin de données du microprocesseur, il n'y a pas d'instant privilégié où elles transitent entre mémoire, décodeur et unité PC, elles circulent les unes à la suite des autres et s'exécutent en parallèle. L'optimisation de la boucle de Fetch consiste donc à obtenir un recouvrement complet entre instructions et une utilisation maximale du matériel dans l'architecture suivant le ratio : Nombre de jetons = Latence de la boucle / Débit de la boucle [Wil94] (voir chapitre 6, paragraphe 3).

Outre le décodage des instructions et la mise à jour du compteur de programme, opérations nécessaires pour toute instruction, la boucle de Fetch doit gérer les points suivants. Tout d'abord à propos des branchements, si le branchement est retardé (DBcc), les deux instructions qui suivent sont toujours exécutées quelque soit le résultat du branchement. Au contraire, si le branchement est non-retardé (Bcc) et que le branchement est pris, les deux instructions qui suivent doivent être annulées. Le décodeur doit donc se mettre en attente de ces deux instructions, les supprimer puis continuer. On voit donc l'intérêt d'un branchement retardé, on perd moins en performances puisque deux instructions utiles sont exécutées. Dans les deux cas, l'unité PC doit mettre à jour le compteur programme suivant l'adresse et le résultat de branchement. Le décodeur doit donc savoir annuler des jetons, ce sera présenté dans le paragraphe 5.5.1.

Pour les instructions Ld-Pg et St-Pg (accès à la mémoire programme depuis l'unité Ld-St), il y a potentiellement conflit sur une ressource unique qui est la mémoire programme. En effet, à chaque instruction, l'unité PC accède la mémoire pour lire une instruction et à un moment à priori inconnu l'unité Ld-St peut accéder de son côté à la mémoire programme en lecture ou en écriture. Il y a donc nécessité de résoudre le conflit. Deux solutions sont possibles. La première est d'arbitrer au plus bas niveau les deux requêtes à la mémoire : on implémente alors des gardes indéterministes (voir chapitre 2). Cette solution a le désavantage de ralentir l'accès à la mémoire programme et donc la boucle de fetch ainsi que de ne pas garantir un ordre d'exécution des instructions ld-pg/st-pg. L'autre solution est de rendre les accès à la mémoire programme déterministes. Il faut alors ordonnancer les deux accès en donnant une priorité à l'un des deux. Ceci nécessite de savoir insérer dans la boucle de Fetch un ordre d'accès à la mémoire programme. Cela revient à créer des jetons de manière dynamique, c'est l'un des rôles de l'unité PC, ce sera présenté dans le paragraphe 5.5.2.

Il est de plus nécessaire de gérer les interruptions. Dans une architecture asynchrone, il n'y a pas à priori de moment déterminé où échantillonner un signal d'interruption. Celui-ci est totalement *asynchrone* du reste du circuit et donc correspond à l'implémentation d'une garde indéterministe. C'est le décodeur qui va gérer les interruptions : lecture du signal d'interruption, annulation des instructions en cours, sauvegarde du compteur de programme, départ effectif à l'adresse définie et acquittement de l'interruption. C'est aussi le décodeur qui exécute l'instruction Sleep (mise en veille / synchronisation sur interruption).

Le dernier problème à résoudre concerne l'initialisation du processeur pour l'exécution du programme de Boot à la levée du signal de reset. Ceci nécessite de savoir où et comment initialiser la boucle de fetch et générer correctement trois jetons lors du démarrage. Ceci sera présenté avec l'unité PC paragraphe 5.5.2.

5.5.1. Le décodeur

Vu l'encodage des instructions de type RISC (cf. paragraphe 4.2), il est facile de déterminer rapidement quel est le type de l'instruction reçue. Avec uniquement les 8 bits de poids forts de l'instruction, on peut déterminer si on a une instruction Alu, User, Ld/St, un branchement ou tout autre instruction de contrôle (Djsr, Nop, …).

A partir de cette information, il reste alors à envoyer les différents champs de l'instruction vers les unités concernées : banc de registres, interface bus et les unités d'exécution. Le décodeur est ainsi constitué (Figure 5-30) d'un étage de pré-décodage, d'un étage de contrôle pour tenir compte des interruptions et du résultat de branchement et d'un étage de *dispatch* qui envoie finalement les différents champs de l'instruction comme valeur de contrôle du chemin de données du processeur. Enfin, le décodeur doit envoyer à l'unité PC le type de l'instruction émise pour lui indiquer comment mettre le compteur de programme à jour : instruction suivante, branchement, ou interruption.

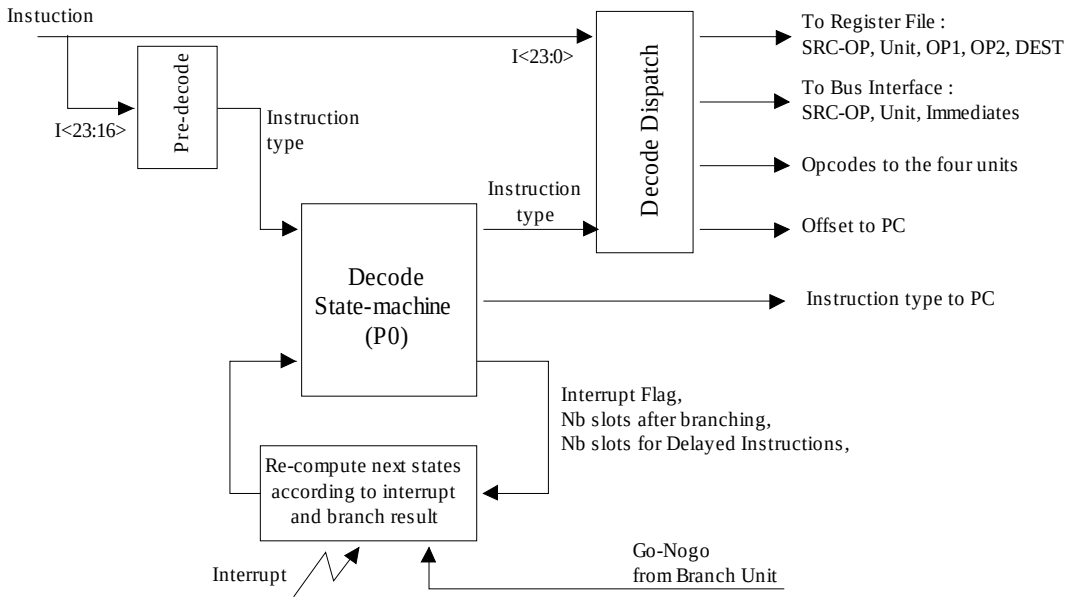


Figure 5-30 : Architecture du décodeur

5.5.1.1. Pré-décodage

L'étage de pré-décodage est un processus rapide dont la latence et le débit varient fortement suivant le type d'instruction reçue. A partir d'au plus 8 bits, ce processus génère le type instruction codé en 17-rails. Il y a effectivement beaucoup de type instruction en plus des trois unités Alu, User, Ld-St en raison des instructions Nop, Dint, Eint, Drti, Sleep et de toutes les instructions de branchements (Bru) qui doivent être décodées dès ce niveau. Les performances de ce pré-décodage sont à l'image de la complexité du codage de l'instruction. Ainsi, il suffit de 2 bits pour détecter une instruction utilisateur (unité User) tandis qu'il faut décoder 8 bits pour détecter un Nop. On voit donc ici l'influence du codage instruction sur les performances. Le codage permet ainsi de favoriser les instructions courantes en vitesse et consommation.

5.5.1.2. Contrôle du décodeur : branchements et interruptions

Une fois le pré-décodage effectué, le décodeur doit tester si il y a ou non interruption pour pouvoir autoriser l'exécution de l'instruction courante. Afin d'implémenter des interruptions précises [HENN 96] (i.e. restaurer l'état complet de la machine lors du retour d'interruption), le départ en interruption est interdit pendant l'exécution des deux instructions après un branchement retardé.

Le départ en interruption est donc effectif si : (i) il y a interruption, (ii) le flag d'interruption est positionné et (iii) l'instruction courante n'est pas l'une des deux instructions après un branchement retardé. Dans ce cas, il faut annuler l'instruction courante, sauvegarder l'adresse du PC de retour, annuler les deux instructions qui suivent dans les delay slots et enfin partir en interruption à l'adresse prédéfinie $0000.

A chaque instruction, il faut donc connaître l'état de la machine qui est fonction des instructions précédentes. Ce mécanisme d'interruption ne peut s'implémenter uniquement avec des processus combinatoires. Il est nécessaire d'utiliser différentes variables d'état : le flag d'interruption *int_flag*, le nombre d'instructions restantes à annuler *nb_slots*, le nombre d'instructions restantes après une instruction retardée *nb_slots_after_delayed*.

D'une manière similaire pour les branchements, lors d'un branchement pris non-retardé, il faut annuler les deux instructions qui suivent dans les delay slots. Ceci est implémenté en utilisant la variable *nb_slots*. En fait, l'annulation des deux instructions après un branchement pourrait s'écrire en CHP de la manière simplifiée suivante (GoNogo étant le résultat du branchement calculé et envoyé par l'unité de branchement) :

```
*[ INSTR?instr;[ instr=Bcc =>
            Bcc() ;                          --  exécute Bcc
            GoNogo?go;[ go=0 => skip         --  branch. non-pris
                      @ go=1 => INSTR?; INSTR?   --  branch. pris
          @ …         ]
```

Si on applique une décomposition en machine à état de la dernière garde (cf. chapitre 3), on voit alors que la variable *nb_slots* correspond au codage du ";" de cette garde séquentielle car il y a trois lectures sur le canal INSTR pour cette même garde.

Le contrôle du décodeur est donc décrit comme une machine à état. Il est principalement constitué du processus P0 qui transmet le type d'instruction reçue au dispatch et à l'unité PC suivant la valeur des variables d'état et qui met à jour ces variables et d'une boucle de retour pour envoyer les variables d'état d'une instruction à la suivante (Figure 5-30). Par exemple, le flag d'interruption *int_flag* est mis à 1 par l'instruction Eint ou Drti, mis à 0 par Dint ou lors du départ en interruption, ou simplement retransmis pour les autres instructions. Les deux autres variables *nb_slots* et *nb_slots_after_delayed* sont des compteurs de 2 à 0 qui sont codés en multi-rail.

Ces trois variables d'état sont traitées indépendamment les unes des autres dans la boucle de retour de P0 afin de répartir au mieux le calcul de la machine à état et donc équilibrer débit et latence dans cette boucle locale. En particulier, comme seules les instructions qui suivent un branchement sont concernées par le résultat du branchement GoNogo, on peut tenir compte de GoNogo au plus tard dans la boucle pour mettre à jour la variable *nb_slots*. Le but est que le processus P0 décrémente la valeur de *nb-slots* lors de l'annulation d'instructions ou demande sa mise à jour en fonction du résultat du branchement pour l'instruction suivante. Il suffit donc de rajouter une valeur dans le codage de la valeur de nb-slots. Le processus de la boucle de retour qui retransmet la valeur de *nb-slots* ou la met à jour en fonction du branchement va donc s'écrire ainsi :

```
*[ N-slots?nb ;
     [ nb=0 => C-slots!0
     @ nb=1 => C-slots!1
     @ nb=2 => C-slots!2
     @ nb=bcc => GoNogo?go; [ go=0 => C-slots!0  --  don't branch
                            @ go=1 => C-slots!2  --  branch
 ]   ]                      ]
```

Le processus P0 peut ainsi terminer et acquitter la génération de toutes ses sorties sans attendre le résultat du branchement GoNogo, c'est uniquement ce processus qui est synchronisé avec l'unité de branchement.

Il faut de plus noter que lors de l'annulation d'instructions (*nb_slots*>0), le processus P0 non seulement attend l'instruction, l'acquitte et décrémente *nb_slots* mais doit aussi envoyer au dispatch et à l'unité PC une commande équivalente à un Nop. Comme le pipeline de la

boucle de Fetch doit toujours contenir trois instructions, l'annulation d'une instruction impose d'en créer une autre pour continuer à incrémenter PC. Ceci permet de re-générer les instructions qui suivent le branchement ou le départ en interruption.

Le décodeur doit tester le canal d'interruption. Ceci s'écrit en CHP avec une garde indéterministe (voir chapitre 2), c'est une garde instable qui s'implémente avec une cellule synchroniseur. Cette cellule est capable de lever un état métastable de manière fiable en un temps de réponse non borné [Mar90]. Contrairement aux circuits synchrones, les circuits asynchrones peuvent utiliser ce type de cellule car la correction fonctionnelle ne dépend pas du temps de réponse du matériel et donc savent traiter les signaux indéterministes de manière fiable.

Afin de simplifier le processus P0, on isole par décomposition cette garde instable dans un processus à part qui génère une information déterministe du type : interruption/pas interruption. L'instruction Sleep est gérée dans ce même processus car dans ce cas on ne doit pas échantillonner le niveau d'interruption mais se synchroniser dessus. On écrit alors le processus suivant (noter le @@ de la garde instable) :

```
*[ Cmd-IT?cmd ;
     [ cmd=0 => [  #INT      => Read-IT!1    -- interrupt
                @@ not(#INT) => Read-IT!0 ]  –- no interrupt
     @ cmd=1 => INT? ; Read-IT!0              -- sleep
 ]   ]
```

Ainsi pour chaque instruction, le processus P0 génère la commande Cmd-IT pour demander la lecture du niveau d'interruption pour l'instruction suivante ou bien exécuter un Sleep. P0 reçoit en retour le résultat du test d'interruption dans le canal Read-IT. L'instruction Sleep s'implémente donc très naturellement en CHP. Il suffit de synchroniser le processeur sur interruption dans cet unique processus pour suspendre l'ensemble du processeur et le mettre en veille. A l'arrivée de l'interruption, le processeur redémarre alors comme si rien ne s'était passé (Read-it=0).

Le départ en interruption est ensuite effectif lorsque la condition *go-to-it=(read_it=1)&& (int_flag=1)&&(nb_slots_after_delayed=0)* est vraie. Tout comme le test d'interruption, le calcul de cette condition est effectué dans la boucle de retour de la machine à état afin de simplifier le processus P0 (Figure 5-30).

Lors du départ effectif en interruption, le processus P0 doit alors : acquitter l'instruction courante car elle n'est pas exécutée ; générer vers le dispatch une commande équivalente à l'instruction "Lra reg15, #-3" pour demander au chemin de données la sauvegarde de PC dans le registre reg15 ; générer vers l'unité PC la demande de départ en interruption ; positionner *nb_slots* à 2 pour annuler les deux instructions qui suivent ; positionner *Cmd-IT* à 1 pour acquitter l'interruption (tout comme un Sleep) et donc positionner le signal INTack[8] à 0 ; et enfin interdire les interruptions en positionnant *int_flag* à 0.

En conclusion, le processus P0 s'écrit comme une table de transition d'état. Suivant le type d'instruction reçue, le test d'interruption et les trois variables d'états, celui-ci retransmet le type d'instruction vers le dispatch et vers PC, met à jour les variables d'état et redemande une lecture d'interruption. Par convention pour les trois variables d'état, on note avec le suffixe C-

[8] L'interruption est un canal, le signal INTack permet au système de détecter si l'interruption est prise en compte.

la valeur courante (Current) et avec le suffixe N- la valeur suivante (Next). Tous les canaux seront codés en multi-rail, les constantes utilisées correspondent donc simplement au numéro du rail. On écrit :

```
*[ I-type?type, Go-to-it?go-to-it,
   C-flag?flag, C-slots?nb-slots, C-slots-delayed?nb-slots-
delayed;
   [ nb-slots=2 =>     -- annulation 1er slot
                 I-Dispatch!nop, I-PC!0, Cmd-it!0,
                 N-flag!flag, N-slots!1, N-slots-delayed!0
   @ nb-slots=1 =>     -- annulation 2ème slot
                 I-Dispatch!nop, I-PC!0, Cmd-it!0,
                 N-flag!flag, N-slots!0, N-slots-delayed!0
   @ nb-slots=0 =>
     [ go-to-it=1 =>   -- interruption
                 I-Dispatch!it, I-PC!it, Cmd-it!1,
                 N-flag!0, N-slots!2, N-slots-delayed!0
     @ go-to-it=0 =>   -- execution
  [ type=alu =>  I-Dispatch!alu, I-PC!0, Cmd-it!0,
                 N-flag!flag, N-slots!0, N-slots-delayed!(-1)
  @ type=Bcc =>  I-Dispatch!bcc,I-PC!bcc, Cmd-it!0,
                 N-flag!flag, N-slots!bcc, N-slots-delayed!(-1)
  @ type=DBCc => I-Dispatch!bcc,I-PC!bcc, Cmd-it!0,
                 N-flag!flag, N-slots!0 , N-slots-delayed!2
  @ type=user => … etc …
 ] ] ] ]
```

On peut noter ici que les types instructions sont regroupés au mieux suivant les actions à effectuer dans le dispatch ou l'unité PC. Ainsi le type d'instruction I-type issu du pré-décodage contient 17 instructions différentes tandis que les types d'instructions I-Dispatch envoyé au dispatch du décodeur et I-PC envoyé à l'unité PC contiennent respectivement 10 et 9 types d'instructions différents. Toutes ces valeurs sont codées en multi-rail pour des raisons de consommation et de performances (chapitre 3).

En conclusion, on voit ici l'intérêt des boucles auto-séquencées. Le contrôle du décodeur est une machine à états dont une partie du calcul est répartie dans la boucle retour afin d'optimiser latence et débit dans cette boucle. Ainsi, tandis que la boucle de retour permet de lire le niveau d'interruption et de prendre en compte le résultat de branchement pour l'instruction suivante, le type de l'instruction courante est transmis au plus tôt de l'étage de pré-décodage vers le décodeur instruction et l'unité PC.

5.5.1.3. Décodeur instruction

Le décodeur instruction (le processus de sortie "*dispatch*") envoie à partir du type instruction généré par le controleur les différents champs de l'instruction vers les unités concernées : numéros de registres et bits de src-op vers le banc de registres, valeurs immédiates et bits de src-op vers l'interface bus, les valeurs d'opcodes vers les quatre unités d'exécution et l'offset de branchement vers l'unité PC. De plus, pour toutes les instructions exécutées au sein même du décodeur (Nop, Dint, Eint, Drti, Sleep), il est nécessaire d'acquitter tous les bits de l'instruction même s'ils ne sont pas utilisés. Le code CHP du décodeur instruction est donné partiellement, on écrit par exemple :

```
*[ INSTR?instr[23..0] , I-Dispatch?type ;
 [ type=0 => skip           -- nop, dint, eint, drti, sleep
 @ type=1 =>                -- User
      UNIT-RF!1, SRC-OP-RF!instr[15..13],   -- to register file
      DEST!instr[3..0], OP1!instr[7..4], OP2!instr[11..8],
      UNIT-BI!1, SRC-OP-BI!instr[14..13],   -- to bus interface
      OPCODE-USER!instr[21..16]             -- to User unit
 @ type=2 =>                -- Alu
      UNIT-RF!0, SRC-OP-RF!instr[15..13],   -- to register file
      DEST!instr[3..0], OP1!instr[7..4], OP2!instr[11..8],
      UNIT-BI!0, SRC-OP-BI!instr[12],       -- to bus interface
      IMM-ALU!instr[11..8], SIGN-ALU!(instr[17] and intr[18])
      OPCODE-ALU!instr[20..16]              -- to Alu unit
 @ type=3 =>                -- Dbra
      OFFSET-to-PC!instr[15..0]             -- to PC unit
 @ type=4 =>     etc …
]     ]
```

Les champs instructions opérandes et immédiats sont donc envoyés quelque soit l'utilisation des opérandes. C'est le banc de registres et l'interface bus qui effectuent le choix des opérandes suivant les bits de Src-Op obtenus (voir paragraphe 5.3). Ceci permet de simplifier ce processus de dispatch car il y a moins de gardes à implémenter. L'envoi des différent champs est toujours effectué quelque soit l'instruction, il n'y a donc pas besoin de décoder l'opcode de l'unité à ce niveau.

Ce processus est un simple processus combinatoire mais il est relativement complexe en raison du nombre d'entrées-sorties. Nous donnons ici quelques remarques à propos de la synthèse de ce décodeur instruction. Comme les différents champs instructions sont au mieux alignés, la logique directe revient simplement à propager les valeurs des bits instructions vers les sorties concernées. Par bit instruction, il faut ainsi trouver la combinaison de gardes qui envoie ce bit instruction vers la bonne sortie. Par exemple, pour toutes les gardes sauf type 0 et 3, OP1 vaut instr[7..4].

La logique d'acquittement est relativement complexe puisque les sorties activées par garde sont très variables. Cet arbre d'acquittement risque donc de pénaliser en performance l'ensemble de la boucle de Fetch et donc le processeur en entier. Comme présenté dans le chapitre 6, paragraphe 6.1.1, soit l'acquittement est construit par garde (pour toute garde, uniquement les sorties actives sont acquittées), soit l'acquittement est régularisé (on acquitte toutes les sorties sachant qu'il faut déterminer pour quelles gardes elles ne sont pas activées). Pour le décodeur, on remarque tout de même une certaine régularité puisque certaines sorties sont toujours acquittées ensemble (contrôle du banc de registre) ou bien sont toujours exclusives (les codeops et immédiats des différentes unités). On obtient alors une logique d'acquittement intermédiaire où sont regroupés par famille les acquittements des sorties : certaines sorties sont "régularisées", d'autres non. On obtient ainsi l'arbre d'acquittement suivant : (acquittement contrôle banc de registres) et (acquittement contrôle interface bus) et (acquittement de l'un des 4 codeop) et (acquittements "régularisés" des autres sorties). Cette logique d'acquittement permet d'optimiser à la fois complexité et performances. On obtient ainsi un temps de cycle de 2.5 ns pour le décodeur instruction sachant que tous les canaux de contrôle sont pipelinés sur leurs sorties.

5.5.2. Unité PC.

L'unité PC doit mettre le compteur de programme à jour pour l'instruction suivante (deux delay slots plus tard) en fonction du type d'instruction émis par le décodeur. Pour les instructions courantes (Alu, User, …), le compteur programme PC est simplement incrémenté. Pour les instructions de branchements (Bcc ou DBcc), l'unité PC incrémente PC ou ajoute l'offset provenant du décodeur suivant le résultat de branchement GoNogo. Pour les instructions de sauts (Djsr, Djmp, Dbsr, Dbra), l'unité PC sauvegarde et/ou restaure PC dans les registres via l'unité de branchement. L'unité PC dialogue donc avec le décodeur, l'interface mémoire et l'unité de branchement (cf. Figure 5-31).

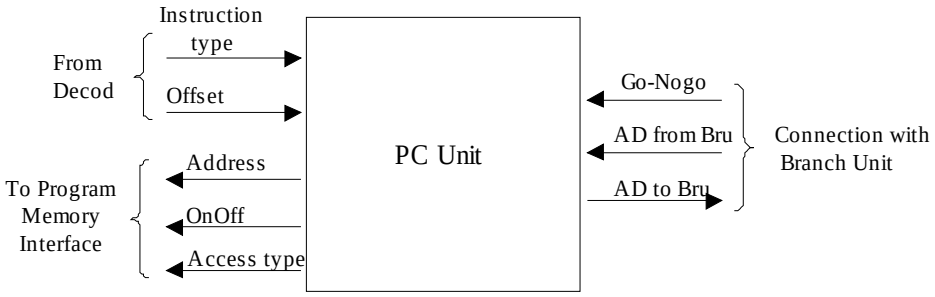


Figure 5-31 : Vue externe de l'unité PC

Une fois la valeur de PC mise à jour, elle est envoyée en tant qu'adresse à l'interface mémoire. Cet interface fait l'interface entre la mémoire interne et la mémoire externe (via le port A du processeur) et entre l'unité PC et l'unité Load-Store. Comme précisé au début du paragraphe 5.5, le but est de rendre déterministe cet interface afin de résoudre le conflit potentiel entre deux requêtes simultanées de PC et Ld/St lors d'une instruction Ld-pg ou St-pg. Pour cela, comme l'unité PC connaît le type de l'instruction, il suffit de préciser à l'interface avec un canal *Mem-Access* quel est le type d'accès à la mémoire : soit un accès depuis PC pour lire l'instruction suivante, soit un accès depuis Ld/St pour exécuter une instruction ldpg/stpg. L'interface mémoire est alors déterministe. Suivant le type d'accès demandé par PC, l'adresse provient de PC ou de Ld/St, et suivant les bits de poids fortd de cette adresse, la requête mémoire est envoyée en mémoire interne ou externe, le résultat de l'accès est alors retransmis vers PC ou Ld/St. L'interface mémoire n'est pas présenté en détail, c'est une architecture purement combinatoire composée de démultiplexeur / multiplexeur tout comme le chemin de données du processeur.

En conséquence lors d'une instruction ldpg/stpg, l'unité PC doit insérer dans la boucle de Fetch un ordre d'accès supplémentaire à la mémoire pour satisfaire la demande de Ld/St.

L'unité PC peut s'écrire ainsi de manière fonctionnelle :

```
*[ I-PC?type, pc:=pc+1;                    -- incrémente pc
   [ type = 0    =>   skip                 -- Nop, Alu, User, etc ...
   @ type = [D]Bcc => OFFSET?offset, GoNogo?go;
                              [ go=1 => pc:=pc+offset
                              @ go=0 => skip          ]
   @ type = Djmp =>   ADfromBru?ad; pc:=ad
   @ type = Dbra =>   OFFSET?offset; pc:=pc+offset
   @ type = Lra  =>   ADtoBru!pc
```

```
    @ type = Djsr =>   ADtoBru!pc, ADfromBru?ad; pc:=ad
    @ type = DBsr =>   ADtoBru!pc, OFFSET?offset; pc:=pc+offset
    @ type = Int  =>   pc:=0000                -- interruption
    @ type = LdSt-Pg => Mem-Access!1           -- demande d'accès pour Ld-St
    ] ;
    ADtoMem!pc, Mem-Access!0                   -- envoi pc à la mémoire
  ]
```

Le processus de l'unité PC est donc séquentiel de deux points de vue : mise à jour de la variable *pc* et ordres d'accès successifs à la mémoire dans le cas d'un ldpg/stpg. Par décomposition, la variable *pc* est extraite et devient une variable tournante avec une valeur *pc* courant et *pc* suivant. Afin de répartir au mieux le calcul dans cette boucle, l'incrément *pc+1* est toujours effectué et ceci dans la boucle de retour (Figure 5-32).

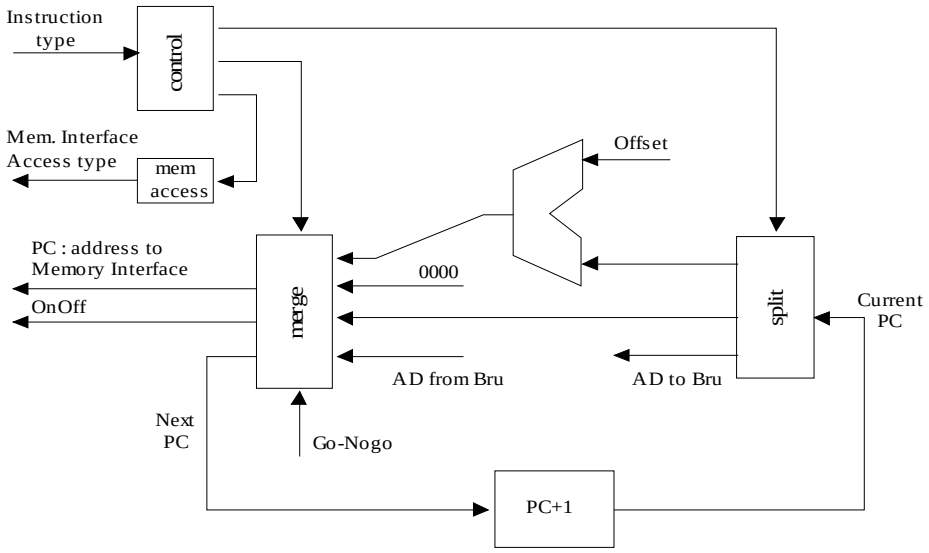


Figure 5-32 : Architecture de l'unité PC

L'unité PC est donc implémentée comme un anneau auto-séquencé qui se compose d'un processus de contrôle, d'un démultiplexeur du *pc* courant (envoi *pc+1*, envoi pour addition avec *offset*, ou envoi vers Bru), d'un multiplexeur pour générer le *pc* suivant et l'adresse vers l'interface mémoire (*pc+1* pour les instructions courantes, *pc+1* ou *pc+offset* suivant le résultat de branchement GoNogo, mise à jour depuis Bru, ou départ en interruption) et de deux additionneurs. Comme les contraintes de débit sur cette unité sont fortes, ces deux additionneurs sont pipelinés (voir paragraphe 5.4) afin d'obtenir un débit maximum.

Cette architecture permet lors d'un branchement de pré-calculer l'adresse destination *pc+offset* et de choisir au dernier moment l'envoi de *pc+1* ou de l'adresse destination suivant le résultat de branchement GoNogo. On gagne ainsi sur la pénalité de branchement. Réciproquement pour les instructions courantes, comme *pc+1* est pré-calculé dans la boucle de retour, la latence de l'unité est faible puisque réduite au temps de traversée des processus d'envoi (démultiplexeur/multiplexeur) à partir du moment où le type d'instruction est émis par le décodeur.

La gestion du type d'accès à l'interface mémoire est fait dans le processus de contrôle de l'unité PC. Celui-ci génère pour chaque instruction une commande du type instruction normale ou bien instruction ldpg-stpg. Le processus *mem-access* (Figure 5-32) permet alors de générer quand nécessaire un accès double vers la mémoire. Il s'écrit :

```
*[ Cmd?cmd; [cmd=0 => Mem-Access!0                  -- accès PC
            @cmd=1 => Mem-Access!1; Mem-Access!0    -- accès Ld/St  puis PC
 ]          ]
```

Ce processus se décompose en machine à états (voir chapitre 3) et s'implémente facilement. La gestion des accès à la mémoire programme est alors déterministe. Dans le cas courant, ce processus n'est pas pénalisant car il possède une latence très faible. Dans le cas d'un ldpg/stpg, les requêtes de PC et de Ld-St sont ordonnancés à l'entrée de l'interface quelque soit leur ordre d'arrivée respectifs. Ce processus permet donc de synchroniser la boucle de fetch et l'unité Load-Store uniquement quand c'est nécessaire.

On peut cependant noter qu'il serait très facile dans ce processus d'intervertir les ordres d'accès entre Load-Store et PC. C'est en fait ce qui est réalisé et ce pour la raison suivante. Si on observe la circulation d'une instruction ldpg-stpg, celle-ci est relativement lente (en latence) puisqu'il faut lancer l'exécution via le décodeur, traverser le banc de registres, l'unité load-store et son registre local ldst-reg avant qu'une adresse envoyée par ldst arrive à l'entrée de l'interface mémoire. Pendant tout ce temps l'adresse de l'instruction suivante est en attente à la sortie de PC. En échangeant les ordres d'accès entre PC et Load-Store à ce niveau, on autorise ainsi le début de l'exécution de l'instruction suivante pendant que le ldpg-stpg est en cours d'exécution. L'insertion dans la boucle de fetch du jeton supplémentaire pour satisfaire la requête de ldst est alors moins pénalisante pour la boucle de fetch, son schéma d'exécution d'ordinaire régulier est rompu mais moins supendu. Ainsi, le résultat des instructions ldpg-stpg est disponible une instruction plus tard, soit au troisième delay slot après l'instruction.

5.5.3. Initialisation de la boucle de Fetch

Le dernier point à mentionner sur la boucle de Fetch concerne le problème de l'initialisation de cette boucle et en particulier la génération des deux premiers delays slots. La boucle de Fetch contient toujours trois instructions, celles-ci doivent donc être générées dès le démarrage du processeur. On rappelle que la procédure de boot choisie pour Aspro consiste à venir chercher la première instruction sur le port A à l'adresse $FFF0. Les deux adresses suivantes sont donc nécessairement $FFF1 et $FFF2 quelque soit la première instruction obtenue.

Une solution élégante pourrait consister à insérer les processus suivants pour générer les trois premières adresses à l'entrée de l'interface mémoire :

```
[ CS!0 ; *[NS?s ;CS!s]]

*[[ #CS=3 => AD-MEM!FFF0, CS?, NS!2        -- première instruction
  @ #CS=2 => AD-MEM!FFF1, CS?, NS!1        -- premier delay-slot
  @ #CS=1 => AD-MEM!FFF2, CS?, NS!0        -- second delay-slot
  @ #CS=0 => AD-PC?ad ; AD-MEM!ad
 ]]
```

Ce procédé est malheureusement un peu coûteux en surface mais surtout augmente un peu la latence sur le chemin des adresses.

La boucle de Fetch est une boucle fermée qui est composée d'unités telles que le décodeur ou l'unité PC qui sont elles-mêmes rebouclées. Toutes ces boucles doivent donc être correctement initialisées afin que le processeur puissent démarrer au Reset. Il est donc plus judicieux de répartir la génération des delay slots le long de la boucle. Comme présenté dans le chapitre 6, on sait synthétiser des processus du type `[ S!0; *[E?x;S!x] ]`. Ce processus correspond en fait à un étage de half-buffer dont le reset est un peu particulier, son initialisation ne dégradant pas ou très peu sa latence. Il suffit donc d'utiliser le bon nombre de ces half-buffer avec initialisation et de les répartir le long de la boucle de Fetch.

La première adresse $FFF0 est générée par une initialisation placée à la sortie du multiplexeur de l'unite PC (Figure 5-32). Ceci permet d'initialiser du même coup la boucle locale de l'unité PC. Les deux autres delay slots sont générés par des half-buffer avec initialisation qui sont placés : i) pour le premier delay slot sur les commandes générées par le processus de contrôle de l'unité PC, ii) pour le deuxième delay slot sur le type d'instruction I-PC envoyé par le décodeur. Ces deux initialisations génèrent des commandes correspondant à une instruction NOP, ce qui permet d'incrémenter la première valeur de PC avec le matériel déjà existant.

5.5.4. Conclusion

La boucle de Fetch est l'une des parties opératives la plus importante du processeur puisque ses performances vont directement imposer les performances globales de la machine. C'est une boucle auto-séquencée dans laquelle circule en continu trois instructions. L'optimisation de cette architecture consiste alors à respecter l'équation *Latence de la boucle = Débit de la boucle * nbr de jetons*.

Dans cette boucle, on n'a pas comme pour le chemin de données du processeur une architecture avec entrelacement permettant de cacher des débits faibles. Toutes les instructions suivent le même chemin. La boucle doit donc présenter en tout point un débit maximum. Ceci a nécessité l'architecture entrelacée de la mémoire programme (paragraphe 5.1) ; un interface mémoire programme déterministe pour supprimer les arbitres ; des additionneurs pipelinés dans l'unité PC ; une simplification et une optimisation conséquente de tous les processus combinatoire, en particulier ceux du décodeur.

La latence de la boucle de son côté a aussi été minimisée le plus possible : dans la mémoire programme, au niveau de l'initialisation de la boucle, et enfin surtout par les architectures de l'unité PC et du décodeur qui utilisent des boucles de calcul avec une répartition maximale des opérateurs afin de diminuer leur contribution dans la latence directe de la boucle de Fetch et augmenter le débit des blocs correspondants. Les étages de pipeline ont donc été réparti au mieux le long de la boucle afin d'équilibrer latence et débit. Le même type d'approche a été utilisé à propos des boucles locales du décodeur et de l'unité PC.

La synchronisation entre la boucle de Fetch et le chemin de données est constituée principalement de l'étage de dispatch du décodeur. Les seules autres synchronisations qui sont fonctionnellement nécessaires concernent les branchements et les instructions ldpg/stpg. Les machines à états sont donc uniquement utilisées pour gérer les variables d'état nécessaires pour implémenter le mécanisme d'interruption. Ainsi, la boucle de Fetch ne gère aucunement le mécanisme de ré-écriture dans le banc de registres. Les écritures dans le désordre dans le banc de registres se font par contrôle local au sein même des registres (paragraphe 5.3), ce qui simplifie le matériel - et la conception - de la boucle de Fetch.

A propos des branchements, le résultat du calcul de branchement est toujours utilisé au plus tard que ce soit dans le décodeur ou l'unité PC. Ceci permet de diminuer partiellement la

pénalité de branchement. Pour les branchements retardés, le décodeur n'est pas pénalisé puisque les instructions sont toujours exécutées. Néanmoins pour les branchements non-retardés, la boucle de Fetch est en attente du résultat de branchement pour pouvoir executer les instructions des delay slots et envoyer l'adresse de l'instruction suivante. Cette pénalité existe et nécessiterait d'autres solutions architecturales tel que des mécanismes de prédiction de branchement pour les résoudre [HENN 96].

Enfin, que ce soit pour les branchements, l'instruction sleep où le décodeur est synchronisé sur l'interruption ou les instructions ldpg/stpg avec la synchronisation sur l'unité Load-Store, on observe donc globalement un schéma d'exécution de la boucle de Fetch qui peut s'avérer très élastique, montrant de brusques arrêts puis des redémarrages, tandis que le flot courant d'instructions est lui normalement relativement régulier.

5.6. Les périphériques

Les périphériques constituent l'une des unités du processeur. Cette unité est directement connectée à l'unité d'exécution Load-Store (paragraphe 5.2.4). Dans la spécification du jeu d'instructions, 256 périphériques sont normalement disponibles. Dans la version du processeur proposée, cinq périphériques sont implémentées, à savoir quatre liens séries deux-phases Nord-Est-Sud-Ouest en lecture et en écriture et un port parallèle 16-bits en entrée et en sortie (voir paragraphe 4.1). Vue du processeur, l'unité périphérique se comporte comme une mémoire, elle possède la vue externe suivante (Figure 5-33).

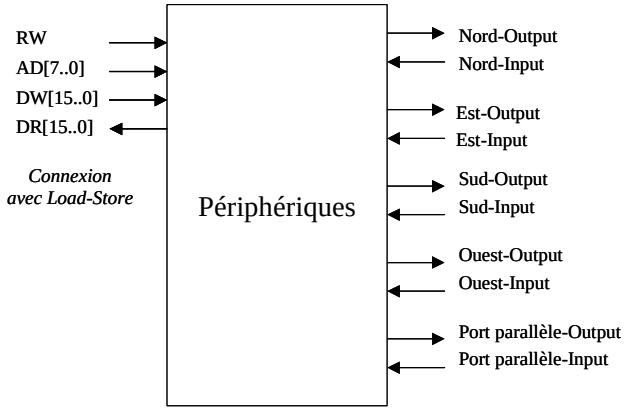


Figure 5-33 : Vue externe de l'unité périphérique.

- **Architecture**

Cette unité se comporte de la manière suivante : suivant le contrôle RW, l'adresse AD, il faut soit effectuer l'écriture de la valeur DW vers le périphérique correspondant, soit effectuer une lecture sur le périphérique et renvoyer la valeur lue dans le canal DR. Ce mécanisme peut simplement s'implémenter avec une architecture avec entrelacement, comme celle utilisée pour la mémoire programme (cf. paragraphe 5.1) où chaque bloc correspond à un espace mémoire périphérique. La Figure 5-34 présente l'architecture de l'unité périphérique.

Les périphériques imposent de plus d'effectuer des conversions de protocole entre l'extérieur du processeur et la logique QDI double-rail quatre-phases utilisée en interne (chapitre 3). Ainsi, pour les ports parallèles, il faut réaliser une conversion en protocole

quatre-phases données-groupées. Pour les ports série en écriture, il faut tout d'abord transformer le mot 16-bits en train série poids fort en tête puis effectuer la conversion du protocole double-rail quatre-phases en protocole double-rail deux-phases (et réciproquement en lecture).

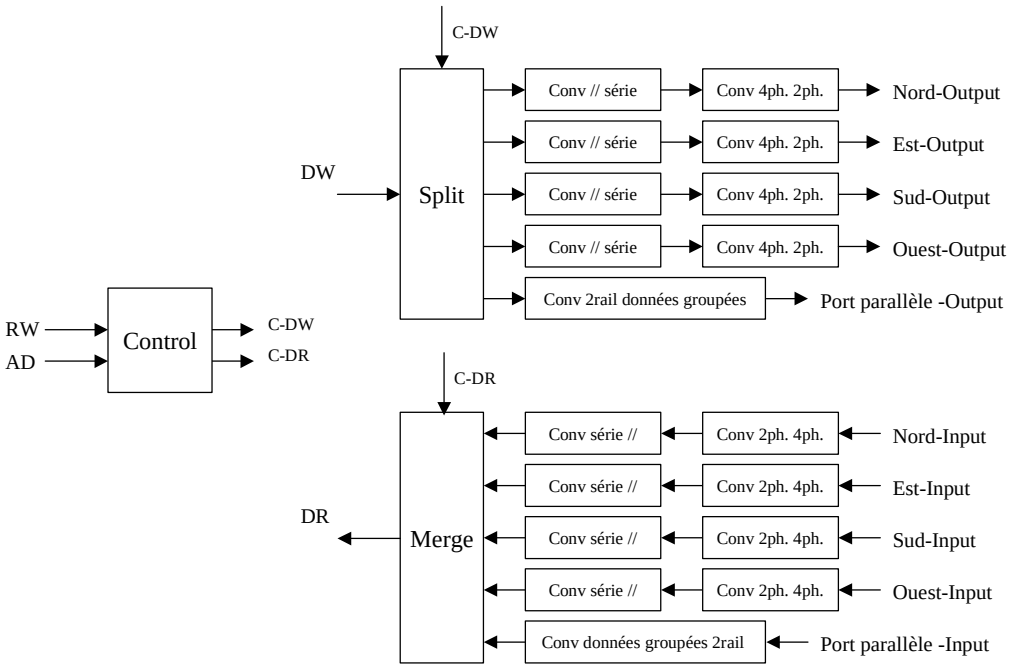


Figure 5-34 : Architecture de l'unité périphérique

Cette architecture permet ainsi de lire ou d'écrire en parallèle vers des périphériques distincts. Comme pour la mémoire programme, il est possible d'obtenir un débit moyen élevé si on lit / écrit vers des adresses différentes. Cette architecture permet même de pouvoir continuer à écrire vers des périphériques même si l'un des périphériques n'a pas répondu à une lecture précédente. Réciproquement, l'architecture se bloque si on accède successivement à des ressources non prêtes.

Cette version du processeur intègre seulement 5 périphériques distincts. Comme présenté dans le chapitre 4, grâce à l'architecture de cette unité périphérique, il serait très facile d'ajouter un périphérique dédié supplémentaire, avec ou sans connexion à l'extérieur du processeur. Cette unité serait alors directement connectée aux multiplexeurs Figure 5-34 des canaux DW et DR et pourrait disposer des bits restants d'adresse. Il suffit alors que cette unité respecte le fonctionnement de la mémoire périphérique : une adresse, une donnée en écriture, une donnée en lecture, quelque soit son schéma de fonctionnement interne.

- **Conversion parallèle-série**

Les blocs de conversion mot parallèle16-bit en train série peut être modélisé en CHP de la manière suivante :

```
*[  E[15..0]?x[15..0] ;                          -- lecture parallèle
    S!x[15] ; S!x[14]; ... ; S!x[1] ; S!x[0]     -- écriture série
 ]
```

Comme les bits d'entrées E[15..0] sont indépendants, on peut les lire (et les acquitter) au dernier moment avant de les envoyer sur la sortie S. On peut alors effectuer une décomposition en machine à états en introduisant un compteur se décrémentant de 15 à 0 sans avoir à mémoriser les bits de E non encore envoyés. La conversion parallèle série est alors composée des trois processus suivants :

```
P0 = *[  C?i ;                          --  le processus P0 transmet le bit i
         [ i=15 => E[15]?xi ; S!xi
         @ i=14 => E[14]?xi ; S!xi
         @ ……
         @ i=1  => E[1]?xi ; S!xi
         @ i=0  => E[0]?xi ; S!xi
 ]       ]
*[  Cur-C?i; C!i , Next-C!(i-1 mod 16) ]   -- décrémente i et l'envoie à P0
[Cur-C!15 ; *[ Next-C?i ; Cur-C!i ] ]      -- initialise le compteur
```

Ainsi la conversion parallèle-série est constituée d'un compteur implémenté en 16-rail et d'un multiplexeur 16 vers 1. Les bits parallèles du canal E sont donc consommés au fur et à mesure de leur envoi vers la sortie S. Réciproquement pour la conversion série parallèle, on obtient un démultiplexeur 16 vers 1 qui dispatche les bits série d'entrée pour constituer le mot 16-bits. On observe ainsi que les bits peuvent être localement désynchronisés entre eux sans pour autant gêner le reste du processeur, ils seront par exemple re-synchronisés au passage dans l'unité Load-Store.

- **Conclusion**

L'implémentation des autres blocs de conversions de protocole sera présentée dans le chapitre 6 car cet aspect concernent un aspect implémentation purement circuit et non un problème d'architecture ou de décomposition CHP. La conversion deux-quatre phases consiste en un problème de protocole, tout en gardant un encodage double-rail, tandis que la conversion en données groupées consiste à un changement de codage tout en gardant un protocole quatre phases.

D'une manière générale, on peut noter que ces périphériques, même si ils présentent une modification de protocole pour aller vers l'extérieur du processeur – moins de connectique pour les ports parallèles, plus rapides et robustes pour les ports séries -, possèdent toujours un faculté de synchronisation puisqu'ils implémentent des canaux de communications.

Ainsi, l'unité périphérique permet de synchroniser naturellement le processeur sur des périphériques extérieurs ou avec d'autres processeurs Aspro du même type. L'accès à un périphérique non-prêt, que ce soit en écriture ou en lecture, risque alors de suspendre le processeur. En lecture, le multiplexeur de sortie des périphériques et celui de l'unité load-store sont suspendus tant que le périphérique ne répond pas. Le flot d'instructions du processeur sera alors suspendu à partir du moment où il essayera de lire le résultat du périphérique dans le banc de registres (on note au passage que le mécanisme du banc de registres permet de relâcher temporairement la synchronisation). Une fois que le périphérique répond, le résultat est écrit dans le banc de registres et tout peut redémarrer normalement. Réciproquement en écriture, en raison d'un certain niveau de pipeline (démultiplexeur du canal DW et blocs de conversion de protocole), l'unité périphérique n'est pas suspendu à la première écriture sur un périphérique non prêt, mais seulement lors d'un enchaînement d'écriture à ce même périphérique. Ainsi, le flux d'instructions du processeur peut se bloquer temporairement si il envoie une série d'écritures sur un périphérique non-prêt ou trop lent.

Cette unité périphérique implémente donc des communications bloquantes avec l'extérieur du processeur. Par ce type d'architecture et d'implémentation, il n'est pas nécessaire d'écrire par programme la gestion d'un périphérique non prêt, lent, etc. Ceci est gérer naturellement par le matériel : la synchronisation du programme s'effectue avec la synchronisation de l'architecture sous jacente. La modélisation CHP par canal de communication unifie ainsi la notion de synchronisation depuis le programme jusqu'au matériel. Il est ainsi possible d'imaginer d'autres manières de programmer [SILC 98].

Tous les périphériques implémentés ici correspondent à des communications bloquantes. Il aurait aussi été possible de concevoir, selon l'adresse du périphérique, des modes de communications bloquant et non-bloquant [ROBI 97]. Ce serait particulièrement intéressant pour l'utilisation du port parallèle. Dans ce cas, le programme pourrait scruter des valeurs à l'extérieur du processeur sans interrompre son flot d'instructions. Ceci est néanmoins rendu possible par le protocole utilisé pour les ports PI et PO (chapitre 4), il suffit de connecter ensembles à l'extérieur du circuit les signaux de requêtes et d'acquittement, les ports se comportent alors comme des communications non-bloquantes.

5.7. Conclusion sur l'étude du processeur

Nous avons présenté dans ce chapitre la micro-architecture du processeur Aspro. En partant d'un modèle CHP de haut niveau, nous avons obtenu par décompositions successives un modèle d'architecture à grain fin, qui est synthétisable tel que défini dans le chapitre 3. Comme nous avons pu le montrer à travers de nombreux exemples, la décomposition CHP permet l'introduction de parallélisme et conduit à implémenter des architectures à temps de calcul et consommation minimum, à la fois en fonction des données et des instructions.

L'étude du processeur nous a tout d'abord montré qu'il est possible et aisé de découpler les paramètres de latence et de débit au sein d'une architecture asynchrone. Ceci est en faveur des performances. L'utilisation systématique d'un mécanisme d'entrelacement entre les différentes unités, sous-unités ou blocs de calcul permet d'exploiter naturellement le parallélisme niveau instruction. Ce principe d'entrelacement est implémenté avec de simples processus de multiplexage / démultiplexage qui envoient au plus tôt les requêtes et/ou données vers les blocs concernés et transmettent au plus tôt les résultats obtenus. Ceci s'effectue localement suivant la disponibilité des données et des ressources de calcul.

En conséquence, débit et latence sont tout deux minimisés : le débit car il se trouve réduit en moyenne au débit des étages d'entrées, la latence car celle-ci est l'image du plus court chemin utilisé. Ce principe a été largement mis en œuvre dans les mémoires et les unités d'exécution du processeur.

Cette étude nous a ensuite montré la flexibilité des synchronisations qu'il est possible d'implémenter dans une architecture asynchrone. Ceci est en faveur de la facilité de conception, nous y reviendrons, ainsi qu'en faveur des performances et de la consommation. La flexibilité des synchronisations offre des temps de calcul dépendant à la fois des données et des instructions. Les performances en temps moyen obtenues sont extrêmement intéressantes. Ceci est aussi bien le cas pour des calculs arithmétiques (addition) que pour du flot de contrôle (instruction *Ldpg*). Ce temps moyen correspond à l'exploitation du parallélisme dynamique offert par les dépendances de données.

Au niveau de la consommation, la spécification des synchronisations permet d'activer uniquement les unités qui sont concernées par la fonction à implémenter. La spécification intègre ainsi directement la notion de mise en veille de la logique, et ceci à tout niveau de granularité. Afin de réduire la consommation, le rôle du concepteur est donc d'étudier les synchronisations minimales à implémenter qui sont inhérentes à la spécification. L'implémentation de l'instruction *Sleep* est ainsi un très bon exemple de l'utilisation de la flexibilité des synchronisations en vue de la réduction de la consommation.

Cette flexibilité des synchronisations est supportée par la cohérence du modèle d'exécution et de la sémantique du langage. Ainsi, il est possible d'implémenter efficacement des mécanismes complexes de communication bloquants ou non-bloquants [ROBI 97] (liens série du processeur, canal d'interruption).

Aussi bien au niveau de la spécification que de la décomposition des unités du processeur, nous avons pu observer une très forte localité de la conception. Ceci est un facteur déterminant d'aide à la conception. Grâce à la sémantique de communication et de synchronisation offerte par le langage CHP, il est possible d'implémenter aisément un contrôle local.

L'exemple du mécanisme de réécriture dans le désordre au sein du banc de registres montre qu'il est aisé d'obtenir un niveau de parallélisme élevé (tirant profit des performances locales des unités et du parallélisme instruction) en ne spécifiant uniquement qu'un mécanisme de contrôle local. Aucun contrôle global n'est strictement nécessaire, c'est la sémantique de communication du langage qui assure le séquencement lorsque il est obligatoire. Vu du concepteur, ceci est bien sûr source de performance, mais il faut surtout noter la facilité de conception d'un tel mécanisme. Contrairement à une processeur scalaire synchrone, il n'a pas été nécessaire d'étudier et d'implémenter une station de réservation des registres et de mécanisme de suspension du Fetch en cas de rupture du pipeline. Le décodeur du processeur ne connaît jamais l'état du chemin de données, que ce soit pour un mode de fonctionnement « fluide », une rupture de pipeline ou un engorgement.

En conséquence de cette localité de conception, le processeur contient des machines à états qui sont distribuées dans l'architecture. Son fonctionnement correspond à un pipeline qui est fortement élastique, avec création ou suppression d'instructions de manière dynamique. L'architecture obtenue au final correspond à un modèle flot de données.

Le dernier aspect important concerne l'optimisation des performances. Tout au long de l'étude du processeur, nous avons noté et tiré partie du fait que l'optimisation des performances est indépendante de la correction fonctionnelle. Cette propriété est supportée par la sémantique du langage, celle-ci est un second facteur d'aide à la conception. Ainsi, au niveau de la décomposition et de l'implémentation de chaque bloc, le concepteur peut choisir le niveau de pipeline souhaité en fonction des performances visées sur ce bloc indépendamment de l'environnement du bloc, la correction fonctionnelle étant toujours garantie par construction. Typiquement, ceci autorise de sous-optimiser les blocs les moins utilisés ou de cacher des débits faibles dans des branches concurrentes de calculs.

Cette propriété a une conséquence importante. Non seulement, cela autorise le concepteur à optimiser en temps moyen (et non en temps pire cas), mais il peut optimiser à loisir après conception d'un bloc en ajoutant des étages de pipeline si les performances sont localement trop faibles (ou réciproquement en supprimer), sans avoir à modifier le reste de la spécification. Ceci permet donc d'optimiser localement les performances en cours ou après

conception. Nous reparlerons en toute généralité de l'analyse et de l'optimisation globale des performances dans le chapitre 6.

Pour conclure sur cette étude, nous regretterons de pas avoir eu assez le temps et les outils pour étudier plus finement l'architecture du processeur et son optimisation. Une mise à disposition d'un chaîne de compilation complète, nous aurait permis de mieux justifier nos choix architecturaux en étudiant par simulation le jeu d'instructions, la répartition de leurs exécutions dans le processeur et le taux de recouvrement entre les unités. De plus, il aurait été intéressant d'étudier l'ajout d'un mécanisme de By-Pass, de mémoires caches, ou encore d'un système de gestion d'exceptions.

A partir de cette description de la micro-architecture du processeur, le chapitre suivant présente son implémentation, son optimisation puis les performances obtenues après fabrication du prototype.

Chapitre 6 :

Le processeur ASPRO : synthèse, optimisation et résultats silicium

Introduction

Ce dernier chapitre présente l'implémentation du processeur et son optimisation, puis le test du circuit après fabrication et des mesures de performances sur silicium. Tout d'abord, le paragraphe 6.1 présente des exemples de synthèse de la micro-architecture du processeur afin de montrer le type de décomposition et d'optimisation logique qui sont possibles avec notre choix de mapping technologique. Le paragraphe 6.2 s'attache au cas particulier de la synthèse des interfaces : interfaces entre logique synchrone et logique asynchrone QDI pour les mémoires et interfaces spécifiques pour les liens série du processeur. Le paragraphe 6.3 présente un modèle théorique pour effectuer l'optimisation globale d'architectures asynchrones. Ce modèle sert de base pour l'optimisation et l'assemblage des différents blocs du processeur (paragraphe 6.4). Enfin, le paragraphe 6.5 présente la méthode de test du circuit et des mesures de performances effectuées sur le circuit après fabrication.

6.1. Conception du cœur du processeur Aspro

Nous montrons dans ce premier paragraphe différents exemples de synthèse de la micro-architecture du processeur Aspro. Cette synthèse des modèles CHP en circuit logique QDI est réalisée avec la méthode que nous avons définie et présentée dans le chapitre 3. Grâce au choix de protocole et de décomposition logique, la synthèse est effectuée en ciblant la bibliothèque standard du fondeur et la bibliothèque de cellules spécifiques qui a été présentée dans le paragraphe 3.3.

Des exemples de synthèse de processus combinatoires directement synthétisables (tel que défini dans le paragraphe 3.2) sont d'abord présentés puis des exemples de processus séquentiels. Nous présentons aussi différentes possibilités d'optimisation, à la fois au niveau de la décomposition logique et au niveau de l'encodage des données et des signaux d'acquittements. Toutefois, ce paragraphe ne se veut pas un guide complet et une méthode formelle et automatique de synthèse mais tout d'abord un reflet du travail de conception qui a été mené sur le processeur Aspro, ainsi qu'une présentation des différentes possibilités de compilation et d'optimisation de ce style de logique QDI.

6.1.1. Synthèse de processus combinatoires

Ce paragraphe présente la synthèse de quelques exemples de processus CHP «synthétisables», tel que défini dans le paragraphe 3.2. Tous ces processus ont un simple schéma de fonctionnement du type consommation / production entre des canaux d'entrées et de sorties : ce sont des processus combinatoires.

Pour ces différents exemples, les différentes phases de la méthode de synthèse ne sont pas présentées de manière systématique. L'expansion des communications (HSE) et les règles de productions ne sont pas écrites afin de simplifier la présentation, cependant le réordonnancement est donné afin de guider le lecteur. Les points délicats de l'optimisation logique seront expliqués en particulier à propos des signaux d'acquittements. Au lieu de se baser uniquement sur des équations logiques, l'objet de l'explication est d'expliciter le lien entre la logique obtenue et les dépendances de données provenant du modèle CHP.

- **Dé-multiplexeur à deux entrées.**

Soit le dé-multiplexeur suivant qui transmet une entrée E sur l'une des sorties R ou S (canal 1-bit) :

```
P1 : *[ C?c ;  [ c=0 => E?x ; R!x
               @ c=1 => E?x ; S!x
       ]       ]
```

Nous présentons la synthèse de ce processus dans sa version pipelinée et combinatoire.

Réordonnancement WCHB :

```
[C0^E0^Ra]; R0+, Ea-, Ca-; [/C0^/E0^/Ra]; R0-, Ea+, Ca+
[C0^E1^Ra]; R1+, Ea-, Ca-; [/C0^/E1^/Ra]; R1-, Ea+, Ca+
[C1^E0^Sa]; S0+, Ea-, Ca-; [/C1^/E0^/Sa]; S0-, Ea+, Ca+
[C1^E1^Sa]; S1+, Ea-, Ca-; [/C1^/E1^/Sa]; S1-, Ea+, Ca+
```

Réordonnancement combinatoire :

```
[C0^E0]; R0+; [/Ra]; Ea-,Ca-; [/C0^/E0]; R0-; [Ra]; Ea+,Ca+
[C0^E1]; R1+; [/Ra]; Ea-,Ca-; [/C0^/E1]; R1-; [Ra]; Ea+,Ca+
```

```
[C1^E0]; S0+; [/Sa]; Ea-,Ca-; [/C1^/E0]; S0-; [Sa]; Ea+,Ca+
[C1^E1]; S1+; [/Sa]; Ea-,Ca-; [/C1^/E1]; S1-; [Sa]; Ea+,Ca+
```

Les règles de production donnent de manière évidente des portes de Muller pour les rails de données R0,R1,S0,S1 : soit à trois entrées pour la version pipelinée, soit à deux entrées pour la version combinatoire. Les signaux d'acquittement Ea et Ca sont égaux, en effet les deux canaux d'entrées E et C sont toujours lus ensembles. Pour la version pipelinée, Ea est donné par le nor4 des rails R0,R1,S0,S1, ce nor4 peut être décomposé en nor2 / and2. Ainsi, les différentes portes peuvent ensuite être regroupées avec des *half-buffer commandé* (Figure 6-1). Pour la version combinatoire, Ea est simplement le and2 des signaux d'acquittement Ra et Sa. On observe ainsi que les versions combinatoire et pipelinée sont très proches en complexité.

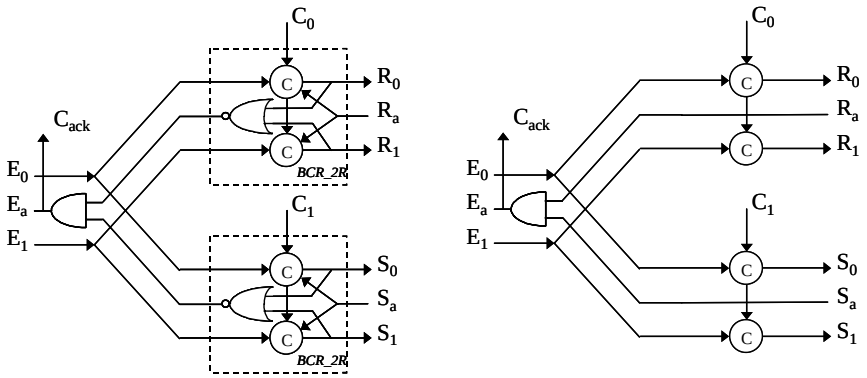


Figure 6-1 : Synthèse d'un dé-multiplexeur à deux sorties, versions pipelinée et combinatoire.

Cet exemple de synthèse peut facilement s'étendre à un dé-multiplexeur avec un nombre quelconque *n* de sorties. Dans ce cas, le canal de contrôle est codé en *n*-rail, on obtient *n* branches équivalentes en sortie et le signal d'acquittement Ea (Ca) est obtenu par une porte and à *n* entrées.

- **Multiplexeur à deux entrées**

Soit le multiplexeur suivant qui transmet une entrée A ou B sur la sortie S (1-bit) :

```
P2 : *[ C?c ;  [ c=0 => A?x ; S!x
               @ c=1 => B?x ; S!x
       ]       ]
```

Comme précédemment, nous présentons les versions pipelinée et combinatoire.
Réordonnancement WCHB :

```
[C0^A0^Sa]; S0+, Aa-, Ca-; [/C0^/A0^/Sa]; S0-, Aa+, Ca+
[C0^A1^Sa]; S1+, Aa-, Ca-; [/C0^/A1^/Sa]; S1-, Aa+, Ca+
[C1^B0^Sa]; S0+, Ba-, Ca-; [/C1^/B0^/Sa]; S0-, Ba+, Ca+
[C1^B1^Sa]; S1+, Ba-, Ca-; [/C1^/B1^/Sa]; S1-, Ba+, Ca+
```

Réordonnancement combinatoire :

```
[C0^A0]; S0+; [/Sa]; Aa-,Ca-; [/C0^/A0]; S0-; [Sa]; Aa+,Ca+
[C0^A1]; S1+; [/Sa]; Aa-,Ca-; [/C0^/A1]; S1-; [Sa]; Aa+,Ca+
```

```
[C1^B0]; S0+; [/Sa]; Ba-,Ca-; [/C1^/B0]; S0-; [Sa]; Ba+,Ca+
[C1^B1]; S1+; [/Sa]; Ba-,Ca-; [/C1^/B1]; S1-; [Sa]; Ba+,Ca+
```

Les règles de production donnent des portes de Muller et des portes or pour les rails de données S0 et S1 : le canal S est le « or commandé » entre les deux canaux A et B. Pour les signaux d'acquittements, seul le canal qui a été lu (A ou B) doit être acquitté. Cette propriété permet d'acquitter facilement le canal C, son acquittement est le « and[9] » des deux acquittement Aa et Ba.

Pour la version pipelinée, A est acquitté lorsque S0+ ou S1+ est généré pour la condition C=0, réciproquement pour B lorsque C=1. Ainsi, les différentes portes peuvent être regroupées avec des *half-buffer commandé* (Figure 6-2).

Pour la version combinatoire, lorsque l'acquittement [/Sa] arrive, Aa- est uniquement généré pour la condition C=0, réciproquement pour B. Ceci s'implémente avec une porte de Muller. Pour cet exemple, la version combinatoire est légèrement plus complexe que la version pipelinée, car il faut savoir d'où est venue la donnée pour déterminer quelle entrée acquitter lorsque le signal d'acquittement de la sortie revient.

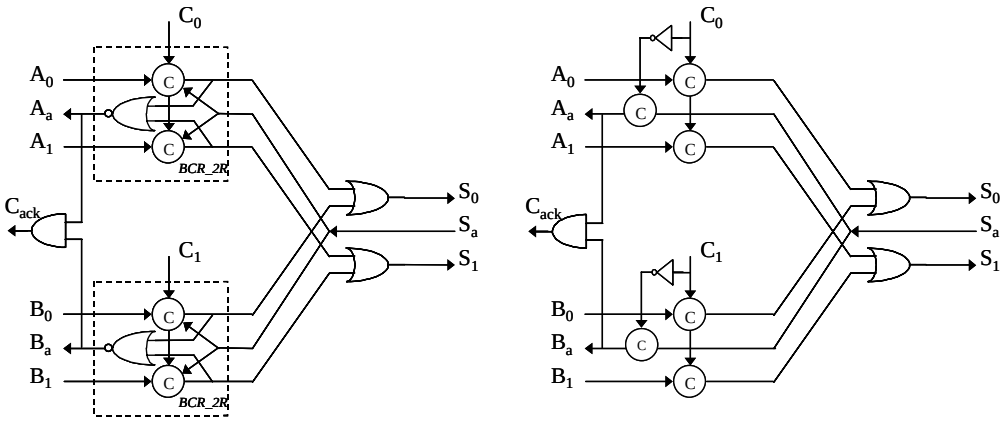


Figure 6-2 : Synthèse d'un multiplexeur à deux entrées, versions pipelinée et combinatoire.

Comme dans l'exemple précédent, cet exemple de synthèse peut facilement s'étendre à un nombre quelconque *n* d'entrées. Dans ce cas, on obtient *n* branches équivalentes en entrée, le canal de contrôle étant codé en *n*-rail et le signal d'acquittement Ca est obtenu par une porte and à *n* entrées.

- **Exemple de processus combinatoire plus complexe**

Soit le processus P3 suivant :

```
P3 = *[ C?c ;  [ c=0 => A?x ; R!x
               @ c=1 => B?x ; R!x
               @ c=2 => A?x ; S!x
               @ c=3 => B?x ; S!x
               @ c=4 => B?            -- évanouissement de la valeur B
               @ c=5 => S!1           -- envoi de la constante 1 sur S
        ]      ]
```

[9] Une porte « and » correspond effectivement au « ou » de deux signaux actifs bas.

Ce processus présente des dépendances de données plus complexe que les multiplexeurs et dé-multiplexeurs précédents. Suivant une commande C, il propage l'une des entrées A ou B sur l'une des sorties R ou S, ou évanouit une donnée sur B ou enfin génère une constante sur S. Ce processus est bien un processus combinatoire synthétisable.

Voici son réordonnancement en protocole WCHB, soit 11 gardes indépendantes :

```
1-   [C0^A0^Ra]; R0+, Aa-, Ca-; [/C0^/A0^/Ra]; R0-, Aa+, Ca+
2-   [C0^A1^Ra]; R1+, Aa-, Ca-; [/C0^/A1^/Ra]; R1-, Aa+, Ca+
3-   [C1^B0^Ra]; R0+, Ba-, Ca-; [/C1^/B0^/Ra]; R0-, Ba+, Ca+
4-   [C1^B1^Ra]; R1+, Ba-, Ca-; [/C1^/B1^/Ra]; R1-, Ba+, Ca+
5-   [C2^A0^Sa]; S0+, Aa-, Ca-; [/C2^/A0^/Sa]; S0-, Aa+, Ca+
6-   [C2^A1^Sa]; S1+, Aa-, Ca-; [/C2^/A1^/Sa]; S1-, Aa+, Ca+
7-   [C3^B0^Sa]; S0+, Ba-, Ca-; [/C3^/B0^/Sa]; S0-, Ba+, Ca+
8-   [C3^B1^Sa]; S1+, Ba-, Ca-; [/C3^/B1^/Sa]; S1-, Ba+, Ca+
9-   [C4^B0]; Ba-, Ca-; [/C4^/B0]; Ba+, Ca+
10-  [C4^B1]; Ba-, Ca-; [/C4^/B1]; Ba+, Ca+
11-  [C5^Sa]; S1+, Ca-; [/C5^/Sa]; S1-, Ca+
```

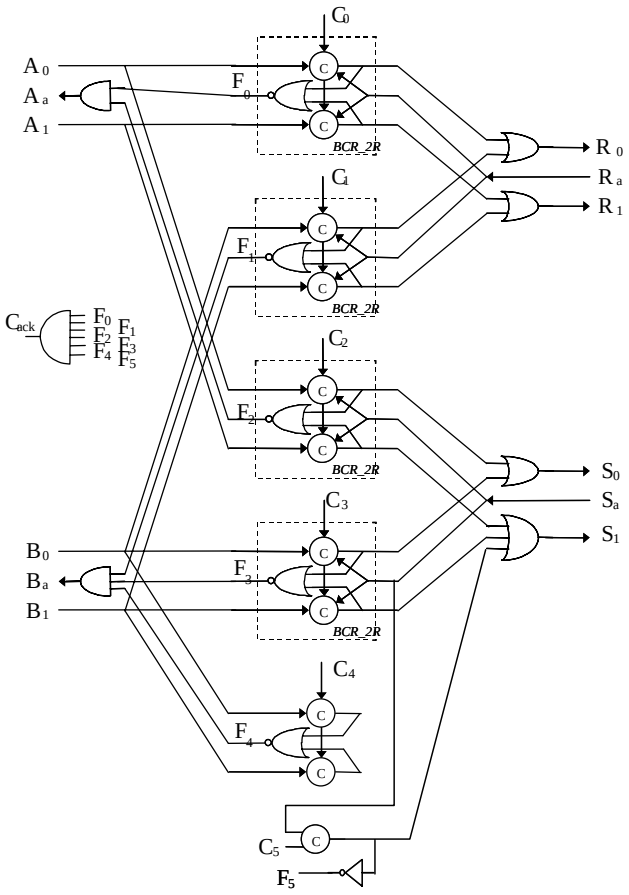


Figure 6-3 : Synthèse du processus P3

Le schéma final est donné Figure 6-3. On obtient ainsi directement 11 portes de Muller. Les rails de sortie sont composés de or2 (comme pour le multiplexeur) des portes de Muller associées deux par deux. Il y a une irrégularité pour le rail S1 avec la génération de la valeur 1 pour C=5, ce qui impose un or3. Pour les signaux d'acquittements, Aa est le nor4 des portes

de Muller 1,2,5,6, Ba est le nor6 des portes de Muller 3,4,7,8,9,10, et Ca est le nor11 des 11 portes de Muller. Tous ces nor peuvent être décomposés en nor/and ce qui permet de factoriser les termes en communs. Cette décomposition logique permet de faire un mapping avec des buffers commandés. Les signaux d'acquittements sont alors composés (comme pour le dé-multiplexeur) de and de signaux d'acquittements intermédiaires Fi.

Cet exemple montre qu'il est possible de synthétiser avec notre style de décomposition logique des processus combinatoires complexes avec des entrées-sorties conditionnelles comme ceux qui ont été présentés chapitre 5, y compris avec évanouissement et génération de données. Cependant, cette dépendance de donnée reste relativement simple puisque que l'envoi d'une donnée d'une entrée vers une sortie correspond toujours à des gardes exclusives. Le paragraphe suivant montre un contre exemple et les possibilités d'optimisations logique qui en découle.

- **Dé-multiplexeur et duplication**

Soit le processus P4 suivant :

```
P4 = *[ C?c ;  [ c=0 => E?x ; R!x
               @ c=1 => E?x ; S!x
               @ c=2 => E?x ; R!x , S!x
       ]       ]
```

Ce processus permet de propager une entrée E sur l'une des sorties R ou S ou de la propager sur les deux sorties. Ainsi il y a deux gardes (c=0 et c=2) qui exhibent une même dépendance E=>R, ce qui n'était pas le cas dans les exemples précédents. Si on effectue le même style de décomposition logique que précédemment on obtient le circuit a) Figure 6-4. Sur ce circuit, on remarque une duplication du matériel qui implémente les transferts entre E / R et E / S. L'idée d'optimisation consiste à regrouper les cas qui effectue les mêmes transferts.

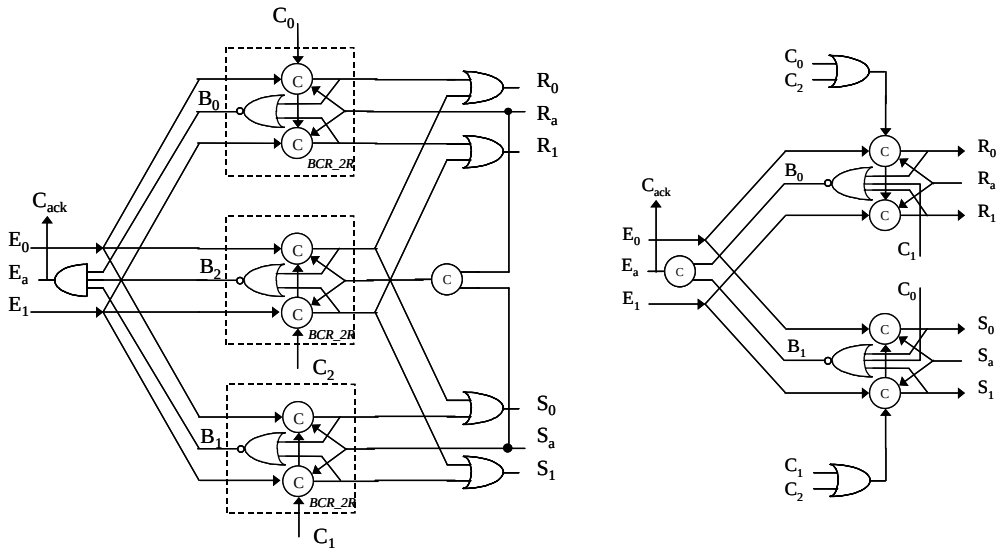


Figure 6-4 : Synthèse de P4 : a) forme normale et b) forme optimisée.

L'optimisation logique en ne se basant que sur les règles de production de P4 est possible mais n'est pas évidente à effectuer, on obtient le circuit b) Figure 6-4. Il est en effet plus facile

de récrire le modèle CHP de manière à regrouper chaque canal de sortie dans des gardes distinctes concurrentes. On écrit :

```
P4' = *[ C?c , E?x ; [ c=0 or c=2 => R!x @ c=1 => skip ] ,
                     [ c=1 or c=2 => S!x @ c=0 => skip ]
       ]
```

Ce modèle CHP ne présente plus uniquement la fonction d'origine du processus P4, on peut remarquer les deux points suivants :

- Le modèle spécifie explicitement quand générer une sortie. Ceci explique les deux portes or :
  (C0 or C2) et (C1 or C2).
- Le modèle spécifie aussi quand ne pas générer une sortie. Les signaux associés au mot skip permettent de calculer un signal d'acquittement correspondant à chaque sous-garde (signaux B0 et B1 Figure 6-4). Ainsi, l'acquittement des entrées C et E est donnée par la Muller du nombre de sorties - que la sortie soit générée ou non – et non plus comme un and3 qui correspondait auparavant au nombre de gardes du processus.

Ce style de décomposition et d'optimisation logique est particulièrement efficace lorsque le nombre de gardes et de canaux de sorties distincts activés par garde est important. Cette optimisation a été utilisée par exemple pour synthétiser le décodeur d'instruction du processeur ASPRO.

- **Conclusion**

Nous venons de montrer des exemples de synthèse de processus combinatoires. L'ensemble du cœur du processeur a été synthétisé de cette manière. Ces exemples montrent des dépendances de données complexes avec des entrées-sorties distinctes. Tous ces exemples peuvent se décomposer et s'implémenter avec de simples portes de Muller. On peut aussi bien obtenir des versions pipelinées avec le protocole WCHB que des versions combinatoires. De plus, pour les cas plus irréguliers, des optimisations logiques sont possibles afin d'optimiser la vitesse et la surface. Cette logique est donc bien plus riche que la logique DIMS où les signaux de données et d'acquittements sont logiquement séparés [DAVI 92], [SPAR 93].

6.1.2. Synthèse de processus séquentiels

Dans ce paragraphe, nous montrons des exemples de synthèse de processus séquentiels, soit en raison d'une variable de mémorisation, soit en raison d'un opérateur de séquencement. Nous présentons en particulier un processus registre et une machine à état avec son processus d'initialisation.

- **Processus registre**

Un registre est défini par le processus suivant :

```
*[ RW?rw;  [ rw=0 => E?x
           @ rw=1 => S!x
 ]         ]
```

Sa synthèse en cellules standard est donnée Figure 6-5. Cette synthèse n'est pas directe : elle utilise une bascule RS pour implémenter l'élément mémoire x. L'étage d'entrée permet d'écrire dans la RS suivant le signal de contrôle en écriture RW0. L'étage de sortie lit le

contenu de la RS sur demande du signal RW1 si la sortie est prête (signal d'acquittement Sa). Le signal d'acquittement Ea doit vérifier la fin de l'écriture dans la bascule RS afin que le registre soit parfaitement insensible aux délais.

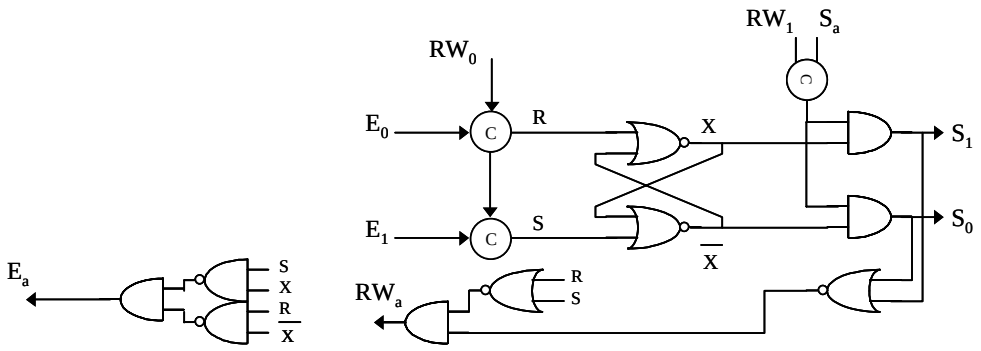


Figure 6-5 : Synthèse du processus registre.

Il est vrai que ce schéma est complexe en surface. Une optimisation avec une cellule complexe serait nécessaire [LINE 95]. Cependant, comparativement à une flip-flop synchrone, ce registre implémente totalement le protocole de communication sur les trois canaux RW, E et S. Cette logique permet ainsi d'utiliser directement cette cellule registre comme simple et unique mécanisme de réservation au sein du banc de registre du processeur (paragraphe 5.3).

- **Exemple de machine à état**

Soit le processus P5 suivant :

```
P5 = *[ C?c ;  [ c=0 => T!0
               @ c=1 => T!1 ; T!0
       ]       ]
```

Cet exemple provient de l'unité PC du processeur (paragraphe 5.5.2), il permet d'insérer une demande d'accès mémoire programme supplémentaire dans la boucle de Fetch dans le cas d'une instruction ldpg/stpg. Ce processus est décomposé en machine à état afin d'extraire l'opérateur séquentiel « ; » (cf. paragraphe 3.2). En introduisant deux canaux CS et NS, on obtient les deux processus suivants :

```
P5' = *[  [ #C=0 => T!0 , C?
          @ #C=1 => CS?cs ; [ cs=0 => T!1 , NS!1
                            @ cs=1 => T!0 , NS!0 , C?
       ]  ]                 ]

Pcs = [ CS!0 ; *[ NS?cs ; CS!cs ]]     -- Pcs initialise et propage l'état
```

Voici le réordonnancement de P5' en utilisant un protocole pipeliné :

```
[Ta^C0]; T0+; Ca-;                    [/Ta^/C0]; T0-; Ca+
[Ta^NSa^C1^CS0]; T1+,NS1+; CSa-;      [/Ta^/NSa^/CS0]; T1-,NS1-; CSa+
[Ta^NSa^C1^CS1]; T0+,NS0+; CSa-,Ca-;[/Ta^/NSa^/C1^/CS1]; T0-,NS0-; CSa+,Ca+
```

Le processus P5' présente donc un schéma de consommation / production sauf pour la deuxième garde (#C=1 & cs=0). Dans ce cas, la commande d'entrée C n'est pas acquittée : sa

remise à zéro n'est donc pas vérifiée. Ceci s'implémente avec une porte de Muller asymétrique, par exemple avec une porte de Muller généralisée à quatre-entrées dont l'une des entrées PMOS est câblée à zéro. Le schéma du processus P5' est donné en (a) Figure 6-6.

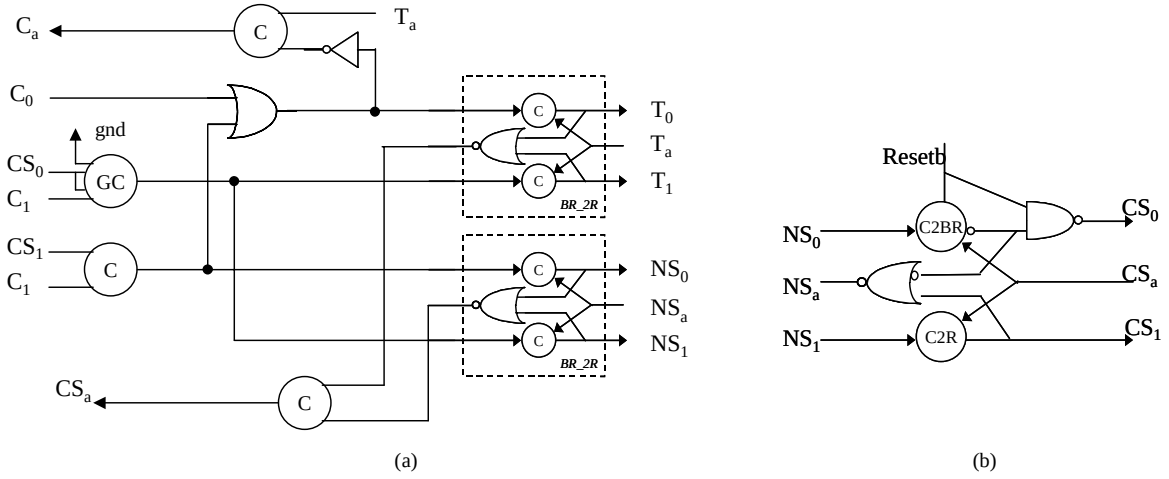


Figure 6-6 : Synthèse de FSM : (a) processus P5' et (b) processus d'initialisation Pcs

Quant à lui, le processus Pcs qui initialise la boucle et propage la variable d'état se synthétise comme un half-buffer mais avec une politique de reset différente. A l'initialisation, celui-ci doit générer un 1 sur le rail CS0 et aussi interdire le passage d'une éventuelle donnée à son entrée (NSa=0). De son côté, le rail CS1 présente un reset normal. Le schéma (b) Figure 6-6 montre une solution utilisant une porte C2BR (porte de Muller inverseuse avec reset).

Enfin, afin que l'association des processus P5' et PCS ne présente pas d'inter-blocage (cf. paragraphe 6.3), il est nécessaire d'avoir suffisamment d'éléments mémoires dans la boucle de retour NS / CS. En raison du protocole quatre phase, il faut en effet au minimum 3 half-buffers dans une boucle afin qu'un jeton puisse y circuler. Comme le processus P5' est pipeliné, il est nécessaire d'ajouter un half-buffer supplémentaire dans la boucle de retour NS/CS en sus du processus Pcs.

Ce type de décomposition des opérateurs séquentiels en machine à état à été utilisé dans différents endroits du processeur, comme dans l'unité utilisateur pour effectuer la réécriture des registres d'accumulation, ou encore la conversion parallèle-série dans l'unité périphérique.

6.1.3. Synthèse de choix non-déterministe

Dans ce paragraphe, nous montrons un exemple de synthèse de processus comportant un choix non-déterministe. Nous présentons l'exemple du test du signal d'interruption utilisé dans le décodeur du processeur (cf. paragraphe 5.5.1). Ce processus s'écrit avec une garde non-déterministe car il échantillonne un signal asynchrone du reste du système. Ce processus peut aussi se synchroniser sur ce même signal d'interruption afin d'implémenter l'instruction Sleep, il est modéliser de la manière suivante en CHP :

```
*[ Cmd-IT?cmd ;
    [ cmd=0 =>  [  #INT      => Read-IT!1       -- interruption
```

```
                    @@ not(#INT) => Read-IT!0 ]     -- pas d'interruption
    @ cmd=1 =>   INT? ; Read-IT!0                   -- instruction sleep
  ]  ]
```

Comme pour tous les autres exemples précédents, ce processus peut être synthétisé sous une forme pipelinée. Voici son réordonnancement en utilisant la version pipelinée (afin de faciliter la lecture, nous notons en abrégé C pour CMD-IT, I pour INT et R pour READ-IT) :

```
[ [Ra^C0]; [  [I] ;  R1+;Ca-;     [/Ra^/C0]; R1-;Ca+
           @@ [/I] ; R0+;Ca-;     [/Ra^/C0]; R0-;Ca+  ]
@ [Ra^C1^I]; R0+;Ia-,Ca-;         [/Ra^/C1^/I]; R0-; Ia+,Ca+
]
```

La cellule de synchronisation SYNC présentée paragraphe 3.3 permet d'implémenter cette expansion des communications. Le test d'interruption est uniquement effectué pour la condition `[Ra^C0]`. La porte de Muller correspondante active la commande de la cellule SYNC (Figure 6-7). Comme la cellule SYNC respecte le protocole quatre phase, la remise à zéro de cette porte de Muller remet ensuite à zéro ses deux sorties u et v. Le reste du schéma est classique. La synchronisation pour l'instruction sleep est implémentée avec une porte de Muller à trois entrées. L'interruption est acquittée (signal INTack) uniquement pour cette dernière condition.

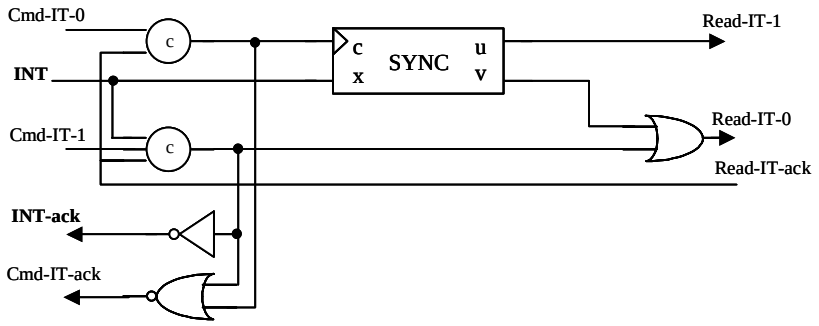


Figure 6-7 : Synthèse de choix non déterministe : test d'interruption du processeur ASPRO.

La cellule SYNC est donc utilisée au milieu même du schéma afin d'arbitrer sur la valeur de l'interruption : le test d'indétermination s'intègre aisément au sein du protocole quatre phase. On observe ainsi qu'il n'y a pas de précaution supplémentaire à suivre au niveau logique et électrique pour implémenter des gardes indéterministes. La cellule SYNC peut répondre avec un temps non borné dans le cas d'une indétermination, ce processus reste fonctionnellement correct.

6.1.4. Codage des signaux d'acquittement

Tous les exemples qui ont été présentés jusqu'à présent sont des processus qui ne porte que sur un unique bit de donnée. Ce paragraphe présente la politique utilisée pour coder les signaux d'acquittements d'un champ de bit, puis comment implémenter les arbres d'acquittements et enfin des possibilités d'optimisation.

La question de la synchronisation des informations se pose effectivement jusqu'au plus bas niveau. Le codage présenté paragraphe 3.1 est un codage double rail utilisant un signal d'acquittement par bit de donnée. Ceci parait effectivement coûteux en matériel mais est

intéressant et ce pour plusieurs raisons. Tout d'abord au niveau architectural, la désynchronisation possible entre bits peut permettre de gagner en performance à chaque fois que des bits peuvent être consommés à des instants différents. Par exemple, lorsqu'on pipeline les différentes retenues qui se propagent dans un additionneur (paragraphe 5.4), si on veut tirer parti de cette architecture, il est nécessaire de désynchroniser les bits d'entrée et de sortie de l'additionneur afin que les sous blocs puissent fonctionner en parallèle. Un autre exemple encore avec la mémoire programme du processeur (paragraphe 5.1), comme la mémoire est pipelinée par blocs, il est nécessaire de désynchroniser les bits de poids faible de l'adresse pour tirer parti de l'architecture de la mémoire. Au niveau électrique et consommation ensuite, le fait d'avoir un signal d'acquittements par bit permet d'avoir des ratios fan-in/fan-out plus faible. En effet, un unique signal d'acquittements par champ de bit devrait attaqué toutes les cellules du champ de bit de l'étage précédent. Enfin au niveau performances, la réalisation d'un arbre d'acquittement à chaque champs de bit serait pénalisant. En effet, le temps de traversée de cet arbre d'acquittement serait inclus à chaque fois dans le temps de cycle du canal de la donnée.

Cependant, il est parfois nécessaire d'implémenter des arbres d'acquittements, mais seulement lorsque la dépendance de donnée du modèle CHP le nécessite. Nous montrons une décomposition CHP sur l'exemple suivant avec un canal de contrôle C (en multi-rail par exemple), une fonction f() et des canaux d'entrées sorties E et S de *n* bits :

```
P = *[ C?c , E(0..n)?x ; S(0..n)!f(c,x) ]
```

En utilisant un acquittement par bit, si chaque bit est indépendant, on peut écrire :

```
P = *[ C?c ,   E(i)?x(i) , E(i+1)?x(i+1) , … ;
               S(i)!f(c,x(i)), S(i+1)!f(c,x(i+1)) , …   ]
```

Par décomposition, on crée *n* canaux C(i) qui sont une copie du canal C d'origine et *n* processus indépendants traitant un unique bit :

```
PC = *[ C?c ; C(i)!c , C(i+1)!c , … ]    // duplication de C
Pi = *[ C(i)?ci , E(i)?x(i) ; S(i)!f(ci,x(i)) ]    // n fois
```

Le processus Pi est alors aussi simple que ceux présentés paragraphe 6.1. Quant à lui, le processus PC se synthétise de manière évidente en utilisant un protocole combinatoire (Figure 6-8). Chaque rail de sortie de Ci est une copie du rail d'entrée de C, l'acquittement du canal C est constitué d'une porte de Muller à *n* entrées : c'est l'arbre d'acquittement (*completion tree*).

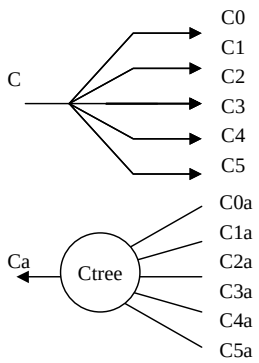


Figure 6-8 : duplication d'un canal et son arbre d'acquittement

Ainsi, la pénalité en performance de l'arbre d'acquittement n'apparaît que sur le canal de contrôle C du processus P d'origine et non sur les canaux de données dont les bits restent indépendants.

Une optimisation est possible : elle consiste à pipeliner cet arbre d'acquittement. Les *n* canaux de duplication peuvent être pipelinés (version WCHB du processus PC) en ajoutant un half-buffer sur chaque canal Ci. Dans ce cas, la pénalité de l'arbre d'acquittement n'apparaît plus dans le temps de cycle de chaque processus Pi, mais seulement dans le temps de cycle du processus PC. Cette solution est coûteuse en matérielle mais offre de très bon niveaux de performances. Enfin, on peut même imaginer décomposer le processus PC (une duplication vers *n*) en *p* processus de duplication vers *n / p*. Dans ce cas, l'arbre d'acquittement est réparti sur plusieurs étages. Cette solution est appelée pipeline 2D [MART 97]. Le pipeline est effectué sur deux dimensions, dans le sens des données et entre chaque bit de donnée, ce qui permet d'atteindre des débits très élevés.

Ainsi, on a montré qu'il est possible de gérer la pénalité des arbres d'acquittements au niveau de l'optimisation globale de l'architecture. Suivant le niveau de performance nécessaire dans tel ou tel bloc, il est possible de synchroniser tous les bits (opérateur tout combinatoire), de désynchroniser les bits et avoir un arbre d'acquittement si nécessaire, ou enfin pipeliner la duplication du contrôle. De manière générale pour le processeur Aspro, seules les deux premières solutions ont été utilisées puisque les temps de cycle obtenus était toujours suffisants pour le niveau de performance visé (cf. paragraphe 6.3).

6.1.5. Conclusion

Les exemples de synthèse du processeur présentés dans les paragraphes précédents montrent plusieurs choses. D'une manière générale, la logique QDI obtenue par la méthode proposée dans le chapitre 3 implémente à la fois la logique de calcul, le protocole de communication et les étages de mémorisation. Contrairement à la logique synchrone classique ou à la logique asynchrone de type micro-pipeline bundle-data, le contrôle n'est pas séparé des données. Le circuit obtenu est un circuit qu'on peut qualifier de « *super-pipeliné* ». En effet, chaque étage de logique, que ce soit du contrôle ou du chemin de donnée, peut-être synthétisé comme un étage de pipeline. La notion de pipeline s'étend du niveau architectural au niveau circuit. Il y a une convergence parfaite entre l'asynchronisme fonctionnel modélisé par le langage CHP de haut niveau et l'asynchronisme niveau circuit.

Par rapport à notre choix de synthèse afin de cibler une bibliothèque de cellules standard, le choix de réordonnancement et de décomposition logique en porte de Muller s'est révélé suffisant pour pouvoir implémenter tout style de circuit. Les circuits peuvent être indifféremment synthétisés sous une forme pipelinée avec le protocole symétrique WCHB ou sous une forme purement combinatoire. Les circuits obtenus ne sont sans doute pas les plus performants en surface / vitesse, mais la décomposition logique est toujours possible, y compris avec certaines optimisations.

Enfin, le langage CHP synthétisable que nous avons défini a permis de modéliser et d'implémenter de manière la plus systématique possible la plus grande partie du processeur. Ceci a été un grand gain de productivité. Les rares cas particuliers, machines à états ou registres ont été traités par des décompositions sur le langage CHP afin de simplifier la procédure de synthèse. On se ramenait alors à des processus combinatoires classiques.

6.2. Conception des interfaces

Dans ce paragraphe, nous présentons les deux types d'interfaces qu'il a été nécessaire de réaliser autour du cœur du processeur Aspro. Le paragraphe 6.2.1 présente les interfaces synchrone-asynchrone avec les différentes mémoires : mémoire programme et mémoire données. Le paragraphe 6.2.2 présente les interfaces pour réaliser les liens série du processeur.

6.2.1. Interface avec des mémoires synchrones

Les différentes mémoires du processeur utilisent une architecture avec entrelacement afin de présenter un débit moyen élevé (paragraphe 5.1). Cette architecture mémoire est purement asynchrone. Cependant, afin de garder une approche cellules standard et surtout de réduire les temps de développement, chaque bloc mémoire est implémenté avec des mémoires SRAM qui sont fournies par le fondeur (STMicroelectronics). Comme ces mémoires sont des blocs synchrones, des interfaces spécifiques entre la logique QDI du cœur du processeur et les mémoires sont nécessaires [RENA 99a]. Le même type d'interface a été développé pour la mémoire programme (4 blocs de 4K instructions de 24 bits) et la mémoire donnée du processeur (8 blocs de 8Koctets).

- **Modèle**

Chaque bloc mémoire asynchrone que l'on veut implémenter possède les canaux suivants : un canal de contrôle Read/Write RW, des adresses AD, la valeur lue en mémoire DR, et la valeur à écrire en mémoire DW (Figure 6-9). De son côté, le bloc de mémoire synchrone utilise un protocole classique avec les signaux suivants : une horloge CK, un signal de contrôle Wen (Write enable) et les bus d'adresses A, de donnée en écriture D et de lecture Q. Cette mémoire synchrone doit respecter différentes contraintes de timings : temps de pré-positionnement (Tsetup), temps de maintien (Thold), temps d'accès (Taccess) et enfin un temps de cycle (Tcycle).

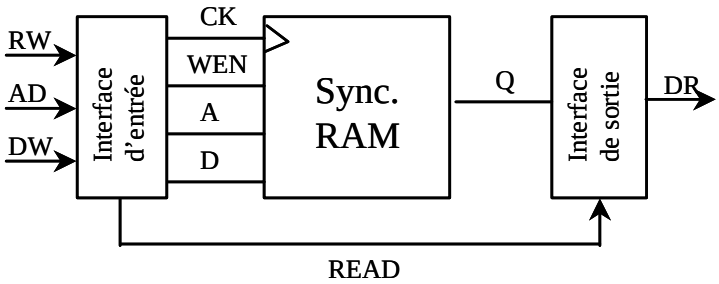


Figure 6-9 : Structure de l'interface mémoire synchrone/asynchrone

Dans la Figure 6-9, les traits fléchés correspondent à des canaux, les traits non-fléchés à des signaux. L'interface d'entrée génère le signal d'horloge de la mémoire synchrone. Ce signal d'horloge est local, il est généré à la demande des canaux d'entrées. Ceci permet d'activer la mémoire seulement lorsque nécessaire, ce qui réduit la consommation. Ceci évite de plus de se synchroniser sur une horloge externe, ce qui serait pénalisant en performance. En lecture, un canal READ est utilisé afin d'activer l'interface de sortie qui transmet la donnée lue sur le canal DR. Ceci peut se modéliser de la manière suivante en CHP :

Interface d'entrée :

```
*[[ #RW=0 and #AD and #DW =>                    -- Ecriture
      WEN=0, A=AD, D=DW; [Tsetup]; CK+; [TholdW]; RW?,AD?,DW?; CK-
  @ #RW=1 and #AD =>                            -- Lecture
      WEN=1, A=AD; [Tsetup]; CK+; [TholdR]; RW?,AD?,[Taccess]; READ!,CK-
 ]]
```

Interface de sortie :

```
*[ READ? ; [ Q=0 => DR!0
           @ Q=1 => DR!1
 ]         ]
```

La contrainte de temps de cycle n'apparaît pas directement dans ce modèle. Celle-ci sera facilement vérifiée en lecture en raison d'un temps d'accès déjà important. Par contre, en écriture il faut ralentir l'interface d'entrée afin de respecter le temps de cycle Tcycle de la mémoire. Ceci est réalisé en utilisant un temps de maintien plus long en écriture (TholdW) qu'en lecture (TholdR).

- ## Implémentation des interfaces

La synthèse de ces interfaces s'effectue avec la méthode de synthèse présentée dans le chapitre 3 en écrivant l'expansion des communications et un réordonnancement adapté. On utilise un protocole WCHB pour l'interface d'entrée et un protocole combinatoire pour l'interface de sortie. Comme la mémoire synchrone n'est pas pipelinée (elle possède un étage de *latch* en entrée pour mémoriser les adresses), on obtient finalement une mémoire asynchrone pipelinée (i.e. équivalent à un étage de half-buffer). Le circuit de ces deux interfaces est donné Figure 6-10.

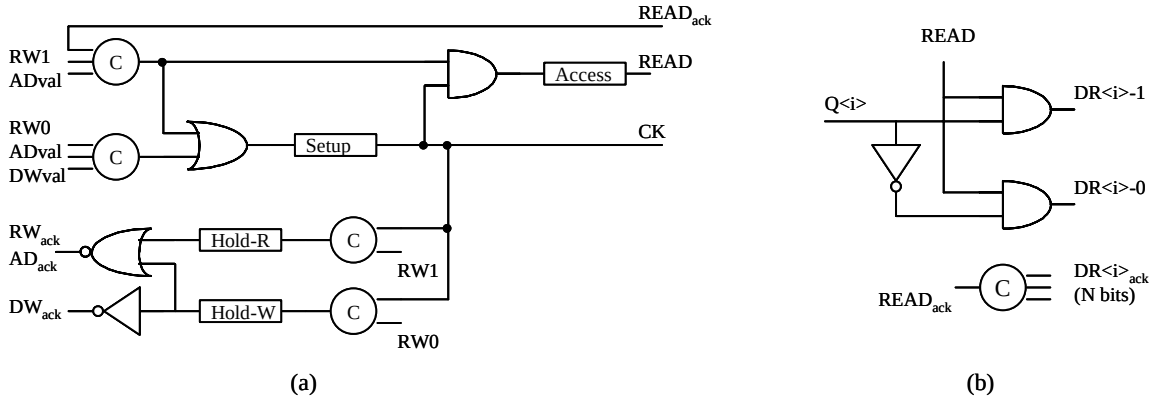


Figure 6-10 : Schéma des interfaces (a) d'entrée et (b) de sortie

Les signaux ADval et DWval indique la validité des canaux AD et DW, ils sont obtenus à partir d'un test de validité de ces bus double-rail (constitué de or2 et d'un arbre de Muller). Les bus d'adresse A et de données D de la SRAM sont simplement connectés aux rails 1 des canaux correspondants (propriété évidente du codage double rail lorsque le canal est valide). Pour l'interface de sortie, l'acquittement du canal READ est obtenu par un arbre d'acquittement de tous les bits du canal DR.

- **Implémentation des délais**

Les éléments de délais sont réalisés individuellement avec des cellules standard (inverseurs). Ces délais sont caractérisés par des simulations électriques, ils sont calibrés afin d'obtenir une marge de 50% par rapport au « *datasheet* » en valeur typique des mémoires SRAM.

Une optimisation par rapport à ces délais est nécessaire afin de ne pas pénalisé le temps de cycle de l'interface en raison du protocole quatre phase utilisé côté asynchrone. Le but est d'avoir un délai uniquement dans la phase montante du calcul et non lors de la remise à zéro du protocole. On peut écrire un délai asymétrique en CHP de la manière suivante :

```
*[[#E => [Delay] ; S! ; E? ]]
```

Soit l'expansion des communications suivante en introduisant deux signaux E0 et E1 :

```
[E];E0+;D;S+;[/Sa];Ea-;[/E];    [ S-;[Sa];Ea+ ],
                                [ E0-;D;E1- ]
```

Ce qui s'implémente avec le circuit de la Figure 6-11.

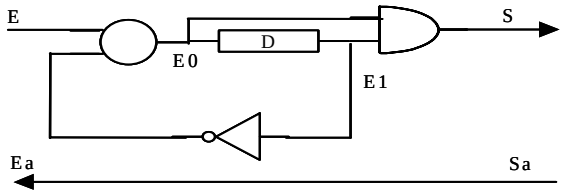


Figure 6-11 : Délai asymétrique

Ce circuit est bien QDI car il vérifie bien que le délai d'un cycle précédent est redescendu (E1-) avant d'en autoriser un nouveau (E0+). Au temps de traversée des portes près, ce circuit présente donc un délai D pour la transition E+,S+ et un délai nul pour la transition E-,S-. Cependant, il peut être pénalisant si la remise à zéro du délai n'est pas finie avant l'accès suivant.

Cette même architecture peut être optimisée en sub-divisant le délai D en n tronçons et en ayant une porte de Muller à n entrées vérifiant la fin de la remise à zéro de chaque tronçon (Figure 6-12). Afin de ne pas être pénalisé, il faut que D soit inférieur à d1+d2 pour que la remise à zéro du délai soit finie avant la requête suivante. De manière générale, on peut donc jouer habilement entre le ratio des temps de cycle de l'environnement et le délai D élémentaire.

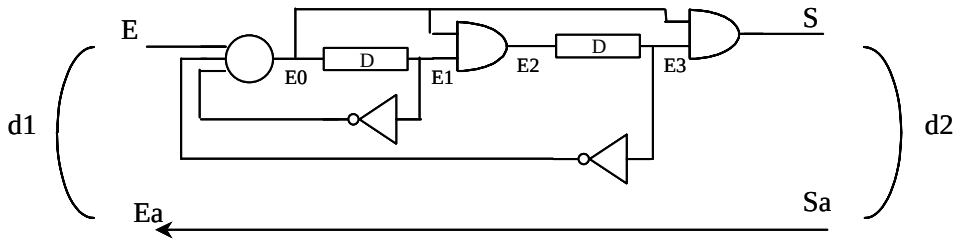


Figure 6-12 : Délai asymétrique avec remise à zéro deux fois plus rapide.

- **Conclusion**

L'interface synchrone-asynchrone présenté a permis d'utiliser des mémoires synchrones au sein du processeur. La conception de ces interfaces au niveau logique est relativement simple avec la même méthode de synthèse. Cependant, elle nécessite une certaine attention afin de respecter les contraintes de timings du mode synchrone et d'optimiser les performances. La synthèse de délais asymétrique QDI est la preuve des optimisations possibles au bas niveau afin d'adapter latence et temps de cycle.

Cet interface a été utilisé à la fois pour les blocs de mémoires programmes et les blocs de mémoires données du processeur. Cet interface correspond en fait à une conversion de protocole entre le protocole double-rail quatre-phase et le protocole données-groupées quatre-phase. Le même type d'interface a été développé pour le port A du microprocesseur (accès mémoire programme externe) ainsi que pour les ports parallèles PO et PI. En introduisant une hypothèse de délai, ces interfaces réduisent le nombre de plots du processeur puisqu'ils implémentent un signal de requête et d'acquittement pour un bus de données (soir N+2 signaux) au lieu du codage double rail classique (2N+1 signaux).

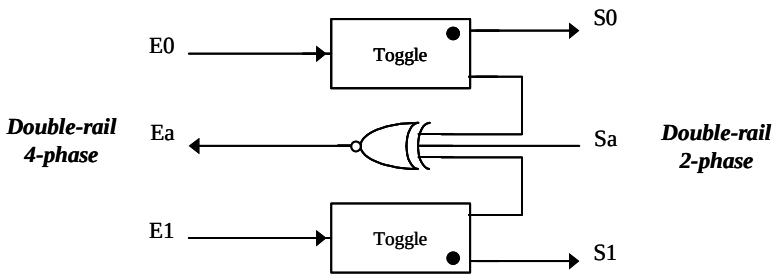
6.2.2. Interface liens série : conversion 2 phases / 4 phases.

Afin de concevoir les liens série du processeur, il est nécessaire d'implémenter des blocs de conversions de protocole entre la logique double-rail quatre-phase du cœur du processeur et le protocole double-rail double-phase des liens série (cf. paragraphe 4.2 pour la spécification de ces liens série et paragraphe 5.6 pour l'architecture de l'unité périphérique et la conversion parallèle-série).

Cette conversion de protocole correspond à un codage double rail différent. Le même nombre de signaux est utilisés (signaux double-rail et un signal d'acquittement) mais avec une table de transition d'état différente. Il faut ainsi effectuer la conversion du protocole quatre-phase qui correspond à un codage trois états par niveaux en un protocole deux-phase qui correspond à un codage quatre états par transition.

Ainsi en protocole quatre-phase, l'envoi de la valeur '1' est effectué avec l'émission de '1' sur le rail qui code la valeur '1', puis à sa remise à zéro. Tandis qu'en protocole deux-phase, l'envoi de la valeur '1' est effectué en générant uniquement une transition (montante ou descendante) sur le rail qui code '1'. Ce dernier protocole est performant car nécessite moins de transitions, c'est pourquoi il est utilisé aux interfaces afin de réduire la pénalité du temps de traversée vers l'extérieur du circuit.

La Figure 6-13 présente respectivement les blocs de conversion 4-phase vers 2-phase utilisés en sortie du processeur et de conversion 2-phase vers 4-phase utilisés en entrée du processeur. Ces blocs sont constitués principalement de portes XOR et d'éléments Toggle [SUTH 89]. La cellule Toggle transmet une transition d'entrée alternativement d'une sortie sur l'autre. Le point indique la sortie correspondant à la première transition.



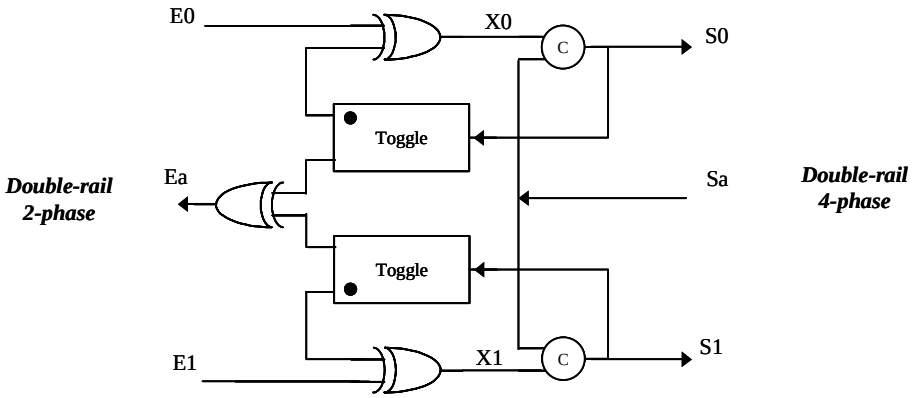


Figure 6-13 : Conversion de protocole : 4-phase / 2-phase et 2-phase / 4-phase

Dans ces schémas, E et S sont des canaux un-bit. Pour la conversion 4-2, une transition sur l'un des rails d'entrée Ei est directement transmise par la cellule Toggle vers le rail de sortie Si correspondant. Ensuite, après réception de l'acquittement de sortie Sa, l'entrée est acquittée grâce à la porte XOR3. La remise à zéro de l'entrée et de son acquittement Ea est effectuée immédiatement via la deuxième sortie du Toggle et le XOR3 sans réaliser de communication avec l'extérieur. L'envoi de la valeur Ei suivante effectuera une transition opposée sur le même rail de sortie (valeur identique) ou une transition identique sur l'autre rail de sortie (valeur différente). Voici par exemple, la suite de transition pour l'envoi de la valeur '1' :

```
[E1] ; S1+ ; [Sa] ; Ea- ; [/E1] ; Ea-
```

Pour la conversion 2-4, une transition sur un rail d'entrée Ei est transmise via la porte XOR2 (signal Xi) et la porte de Muller sur le rail de sortie Si si le signal d'acquittement Sa de la sortie le permet. Le Toggle permet de remettre à zéro tout d'abord l'entrée Xi de la porte de Muller afin d'autoriser la remise à zéro du protocole quatre-phase sur S. Lorsque la sortie est acquittée (Sa-), le rail Si est alors remis à zéro par la porte de Muller correspondante, ce qui acquitte pour finir l'entrée (Ea) via la deuxième sortie du Toggle et le XOR2. L'envoi de la valeur suivante peut s'effectuer sur le même rail ou l'opposé. Voici par exemple la suite de transition pour l'envoi de la valeur '1' :

```
[E1] ; Xi+ ; [Sa] ; S1+ ; Xi- ; [/Sa] ; S1- ; Ea+
```

Cette implémentation correspond donc à un entrelacement particulier des deux protocoles. La remise à zéro du protocole quatre-phase se fait en parallèle de la transmission / réception du bit vers / provenant de l'extérieur du processeur. Ces circuits sont relativement rapides, l'utilisation du protocole deux-phases offre alors de bonnes performances car n'utilise que deux transitions vers l'extérieur du processeur par bit transmis. Les bits sont alors envoyées en train série sur ces canaux, la conversion parallèle-série étant effectué dans l'unité périphérique avec la logique quatre-phase classique. Les liens série sont d'autant plus rapide que ces circuits de conversions de protocoles sont parfaitement QDI. Il n'y a donc aucune hypothèse de délai à effectuer aux interfaces du circuit, que ce soit au niveau des plots du circuits ou au niveau d'une carte système. Les bits seront toujours transmis au plus vite suivant la conception du système et les temps de réponse du périphérique distant (par exemple un processeur Aspro identique).

6.3. Analyse des performances et optimisation

La logique asynchrone possède un mode de fonctionnement radicalement différent du mode synchrone. L'objet de ce paragraphe est de présenter le comportement du pipeline asynchrone, l'analyse et l'optimisation de ses performances. Ce paragraphe est largement inspiré d'un article de F. Robin [ROBI 01]. La théorie des anneaux asynchrones a été étudiée de manière approfondie par T. Williams [WILL 91], [WILL 94]. Ce principe d'optimisation des architectures asynchrones est aussi appelé « *slack-matching* » par A. Martin [MART 93], [BURN 91] - (littéralement *appariement de l'élasticité*) -.

• Comportement du pipeline asynchrone

Afin d'illustrer ce paragraphe, prenons en exemple un simple étage de logique combinatoire (tel que présenté paragraphe 6.1) séparé par des éléments de mémorisation (comme des half-buffers) (Figure 6-14).

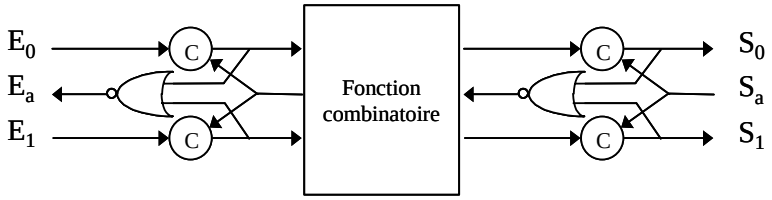


Figure 6-14 : exemple de structure de pipeline asynchrone

Les caractéristiques d'une telle structure sont très différentes d'un pipeline synchrone. Le but de ce paragraphe est de les analyser à partir d'un modèle simplifié. On définit les paramètres suivants :

Lf la latence directe par étage d'un jeton
Lr la latence retour par étage d'une bulle (i.e. la remise à zéro du jeton précédent)
P le temps de cycle local par étage
N le nombre total d'étages dans le pipeline
K le nombre de jetons (i.e. de données) circulant dans le pipeline ou l'anneau.
λ la latence totale du pipeline ou de l'anneau
T le débit global du pipeline ou de l'anneau

Le principe général est de considérer la structure comme un ensemble de boucles interconnectées. Pour un unique étage de pipeline comme dans la Figure 6-14, on observe deux boucles distinctes. Une première boucle partant du rail E0, traversant la porte de Muller correspondante, la fonction, puis la porte de Muller du rail S0, pour ensuite revenir sous forme d'un signal d'acquittement à travers la porte NOR du *half-buffer* de sortie puis à travers la fonction. La deuxième boucle correspond à l'autre rail de donnée. Dans cette structure, une donnée, tout comme un jeton circulant dans un graphe de flot de donnée, peut s'étendre dynamiquement sur plus ou moins d'étages successifs, d'où la notion de pipeline élastique.

En effet, considérons un pipeline initialement vide. Tous les rails de données et d'acquittement sont à leur valeur d'initialisation du protocole. Lorsqu'une nouvelle donnée se présente à l'entrée, celle-ci se propage sans attente, comme dans une structure totalement combinatoire, aussi bien à travers la fonction qu'au travers des différents étages de mémorisation car tous les signaux d'acquittements autorisent le passage de cette nouvelle

donnée. Le temps de traversée d'une donnée dans un pipeline vide est appelée latence directe. On définit une latence directe moyenne par étage, notée Lf. C'est la vitesse maximale de propagation de la donnée.

Le temps de cycle d'un étage, noté P, est le temps minimum qui sépare la prise en compte de deux données successives dans ce même étage. Il est égal à la latence de la boucle activée dans l'étage (il faudrait considérer en fait le maximum des latences des différentes boucles possibles par étage). En raison du protocole quatre phase, il faut inclure le temps de remise à zéro de la boucle, ce qui correspond au parcours du jeton (la donnée) puis de la bulle associée (la remise à zéro du protocole). Comme on utilise le protocole WCHB, le temps de cycle P est alors égal à 2 * (Lf + Lr) où Lr est la latence de calcul du signal d'acquittement. Ce temps de cycle correspond au temps de cycle maximum local de l'étage.

On voit immédiatement que la vitesse de propagation d'une donnée (liée à la latence directe) peut être bien supérieure au débit local (lié au temps de cycle), alors que ces quantités sont égales en mode synchrone. Dans un pipeline vide, pendant que le front d'une donnée avance, les étages qui précèdent le front sont à des stades différents du protocole : phase d'acquittement ou de remise à zéro. On observe ainsi un « effet de traîne » du jeton pendant son avancement. L'extension (moyenne) d'un jeton correspond à la distance parcourue pendant une durée égale au temps de cycle (moyen), à la vitesse de propagation (moyenne) liée à la latence directe des étages. Le nombre d'étages sur lequel s'étale le jeton est donc égal au ratio P / Lf (cf. Figure 6-15).

En se basant sur la structure de la Figure 6-14, si on remplace la fonction combinatoire par la fonction identité (simplement trois fils), on obtient une simple FIFO composée de half-buffers (Figure 6-15). En supposant que le temps de traversée de la NOR est égal à une unité et le temps de traversée de la porte de Muller égal à deux unités, on obtient :

une latence de Lf = t(C) = 2 ;

un temps de cycle local de P = 2 * (t(C) + t(C) + t(NOR)) = 10 ;

soit une extension de la propagation du jeton de P / Lf = 5.

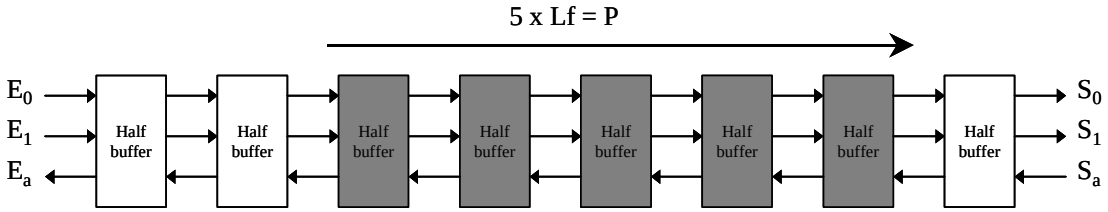


Figure 6-15 : Elasticité du pipeline asynchrone.

• Modélisation des anneaux asynchrones

Dans la plupart des architectures, en particulier dans les microprocesseurs, les unités de calcul utilisent des données qui dépendent de leurs propres résultats précédents. Le pipeline n'est donc pas seulement un pipeline linéaire, mais il comporte des rebouclages sur lui-même. On parle couramment d'anneaux asynchrones [WILL 94]. C'est bien sûr le cas des différentes boucles de calcul du processeur Aspro (chapitre 4 et 5) : boucle de fetch, boucle du data-path, boucle de branchement, boucle de l'unité PC, ainsi que des boucles plus locales comme celles des différentes machines à états.

Considérons la structure simplifiée de la Figure 6-16. Elle représente un anneau composé d'étages de pipeline tous identiques avec leur signalisation bidirectionnelle (rails de données

dans un sens et signaux d'acquittements dans l'autre), ainsi que trois jetons qui y circulent. Ceci représente par exemple, de manière extrêmement simplifiée, le fonctionnement de la boucle de Fetch avec ses trois instructions.

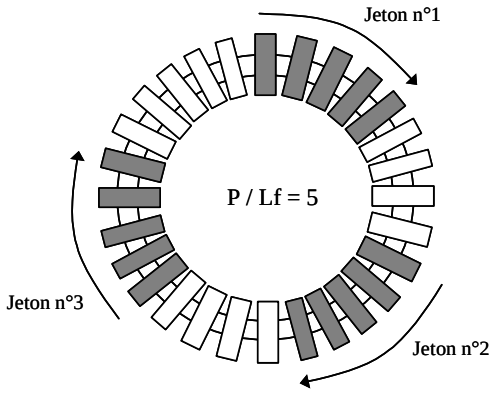


Figure 6-16 : Circulation de données dans un anneau asynchrone (dans le cas « limité par les données »)

Dans cet exemple, la caractéristique de propagation de chaque jeton est identique au cas d'un pipeline vide car il y a suffisamment d'étages vides pour que chacun des trois jetons avance, sans « mordre la queue » du jeton qui le précède. Calculons dans ce cas le débit global de l'anneau :

La latence totale de l'anneau est de : $\lambda = N . Lf$.

Le débit global de l'anneau correspond au nombre de jetons K qu'il contient divisé par le temps de parcours λ de chaque jeton, soit :

$$T = K / \lambda = K / (N . Lf) \qquad (1)$$

Quel est alors l'optimal de cet anneau ? Comme le débit global de l'anneau ne pourra jamais dépasser le débit local de chaque étage, le débit maximum de l'anneau est donné par $T_{max} = 1 / P$. Ainsi, pour un nombre K donné de jetons, le nombre optimal d'étages pour que le débit global soit maximal est donné par :

$$N_{opt.} = K . P / Lf \qquad (2)$$

Cet optimal correspond à la limite d'application de la formule (1). C'est effectivement le nombre minimal d'étages permettant aux jetons de se propager à la vitesse maximale correspondant à la latence directe Lf, tout en évitant de se rencontrer et donc de se ralentir en raison de contrainte de synchronisation locale.

Si le nombre d'étages est supérieur à cet optimum K.P/Lf, les données se propagent comme dans une structure purement combinatoire. On dit que l'anneau est limité par les données : il reste des étages inoccupés, l'anneau est sous-alimenté en jetons. C'est le cas de la Figure 6-16.

A l'opposé, si le nombre d'étages est inférieur à 2K+1, l'anneau ne peut pas fonctionner car il y a inter-blocage. En effet, comme chaque jeton s'étend au minimum sur 2 étages en raison du protocole quatre-phase, il faut au moins un étage vide pour que la circulation des données soit possible.

Dans le cas où le nombre d'étages est compris entre 2K+1 et K.P/Lf, les jetons ne peuvent se propager à leur vitesse maximale, leur extension est limité par la place disponible dans

l'anneau. On dit que l'anneau est limité par les bulles. La vitesse de circulation n'est plus déterminée par le nombre de jetons mais par le nombre de bulles. Le nombre d'étages disponibles pour la circulation des acquittements est égal au nombre total d'étages moins le nombre d'étages minimum nécessaire au stockage des données, soit N-2K. Les bulles se propagent dans le sens inverse des données, avec une latence égale à la latence retour Lr. En raison de la phase de remise à zéro, il faut deux circulations complètes autour de l'anneau à une bulle pour faire avancer un jeton complet. Le débit global de l'anneau est alors donné par :

$$T = (N - 2.K) / (2 . N . Lr) \quad (3)$$

La Figure 6-17 présente le débit global de l'anneau en fonction du nombre d'étages pour un nombre de jeton donné et en fonction du nombre de jetons pour un nombre d'étages donné.

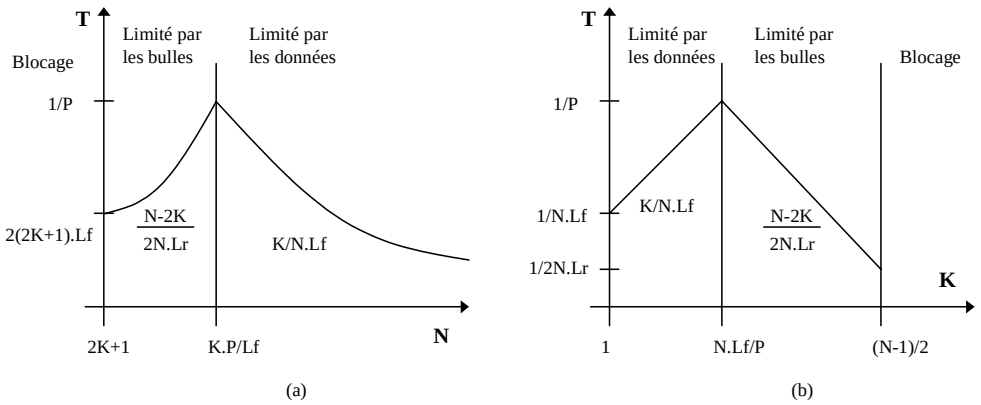


Figure 6-17 : modélisation du débit d'un anneau asynchrone :
(a) en fonction du nombre d'étages, (b) en fonction du nombre de jetons.

Ce modèle, bien que très simplifié, permet d'analyser et d'optimiser correctement les structures pipelinées dans les architectures asynchrones. En réalité, chaque étage de pipeline est différent, les synchronisations entre boucles dépendent des données et ce en interagissant avec de nombreuses autres boucles de calcul (chapitre 5). Il faudrait de plus tenir compte de paramètres différents pour chaque étage de pipeline : latence directe, latence retour et temps de cycle local. A ce jour, il n'existe pas de modèle théorique complet afin de modéliser et d'optimiser ce type d'architecture asynchrone. Ceci est un domaine de recherche totalement ouvert.

Il faut cependant retenir l'importance du paramètre P / Lf qui fait toute la différence avec les architectures synchrones. Ce paramètre offre un nouveau paradigme en terme d'architecture, en découplant latence et débit (paragraphe 5.1).

Enfin il faut noter que si l'optimal N=K.P/Lf n'est pas atteint, il sera plus optimal de travailler dans une zone de fonctionnement « limité par les données » que « limité par les bulles ». En effet, l'équation (3) montre un débit qui décroît plus rapidement que l'équation (1). Ceci est d'autant plus vrai que la latence retour Lr est souvent supérieure à la latence directe Lf en raison des arbres d'acquittements (cf. paragraphe 6.1.4).

- **Conclusion et comparaison avec le pipeline synchrone.**

L'optimisation globale du pipeline asynchrone est une chose délicate puisqu'il faut jouer sur quatre paramètres de l'architecture : latence, débit, nombre de jetons, nombre d'étages. Il

n'existe pas actuellement d'outils afin d'optimiser dans le cas général les performances des anneaux asynchrones. Le seul outil à la disposition du concepteur est la vérification et l'estimation de performance par simulation. Le schéma de fonctionnement asynchrone est effectivement plus complexe que le mode synchrone, où toutes les opérations sont cadencées par un unique cycle horloge. Cependant, le problème d'optimisation globale du pipeline asynchrone n'est *à priori* pas plus complexe que le choix d'architecture d'un pipeline synchrone. En effet pour ce dernier, il faut décider de la répartition des fonctions entre les étages de pipeline et concevoir le contrôle associé à ce découpage, afin que tous les étages soient les plus réguliers possibles, avec des chaînes critiques entre étages non disparates afin de profiter au mieux de chaque cycle horloge dans chaque étage.

De son côté, la méthode de conception asynchrone offre une approche descendante où la conception de la fonction est nettement séparée de l'optimisation de ses performances. Si dans une première étape de la conception, il est nécessaire d'estimer correctement le nombre de jetons devant circuler dans l'architecture (comme le nombre d'instructions dans la boucle de Fetch pour notre architecture), l'optimisation en débit est une tâche qui est effectuée tard dans le design.

En effet, contrairement au cas synchrone où les étages de pipeline sont figés dès le début du design, par décomposition de processus, il est possible de subdiviser tout étage de pipeline (ou bloc de calcul, de contrôle, etc.) afin de réduire son temps de cycle local car la correction fonctionnelle de l'ensemble reste toujours préservée. La décomposition est toujours possible localement sans avoir à modifier le reste des blocs. Le concepteur peut ainsi s'appliquer à l'optimisation des blocs critiques, c'est à dire ceux qui sont sur les chemins potentiellement les plus utilisés, et relâcher la contrainte d'optimisation sur d'autres blocs. Ceci est d'autant plus pertinent que l'architecture asynchrone aide à cacher les débits faibles grâce à l'entrelacement dynamique entre les différentes branches matérielles (architecture des mémoires et des différentes unités du processeur, chapitre 5).

La décomposition permet surtout un bon niveau de performance car le pipeline du mode asynchrone ne coûte pas cher. Grâce à la logique « super-pipelinée » présentée dans les exemples du paragraphe 6.1, il n'y a pas de perte de latence directe, les éléments de mémorisation du pipeline sont intégrés avec la logique de calcul, ce qui optimise le ratio P / Lf. Comme tout processus peut être synthétisé sous une forme combinatoire ou pipelinée, le concepteur peut choisir la forme nécessaire suivant le niveau de performance voulu localement.

De plus, il est toujours possible d'ajouter des étages de pipeline en insérant des half-buffers supplémentaires afin de rajouter de la latence directe, ou de découpler des chemins entre différents blocs : on ajoute de l'élasticité afin d'optimiser les performances. Ceci est parfois nécessaire pour éviter tout inter-blocage. En effet, pour faire circuler K jeton, il faut au minimum 2K+1 étages de pipeline (soit 3 étages pour un jeton). L'ajout de half-buffer permet surtout de découpler les différents chemins. Cela permet de construire des Fifo qui stockent temporairement des données avant qu'elle ne soient consommées (par exemple les Fifos sur les opcodes dans l'interface bus, paragraphe 4.3).

En conclusion, que ce soit en synchrone ou en asynchrone, la notion de pipeline permet toujours de diminuer le temps de cycle local. Cependant en synchrone, trop de pipeline devient pénalisant car ajoute autant de latence que de nombre d'étages. Au contraire, en asynchrone, grâce au ratio élevé temps de cycle sur latence P / Lf, on peut se permettre de pipeliner plus avec une moindre pénalité. On évite en particulier des pertes de performances trop élevées lorsque un pipeline profond est suspendu.

6.4. Optimisation du processeur Aspro

6.4.1. Implémentation, optimisation

La micro-architecture du processeur a été présentée dans le chapitre 5. La synthèse des différents blocs du processeur a été effectuée avec la méthode de synthèse et le flot de conception présentés dans le chapitre 3. Différents exemples de synthèse ont été présentés dans le paragraphe 6.1. La Figure 6-18 rappelle l'architecture du processeur avec ses principaux blocs fonctionnels et boucles de synchronisation (chapitre 4).

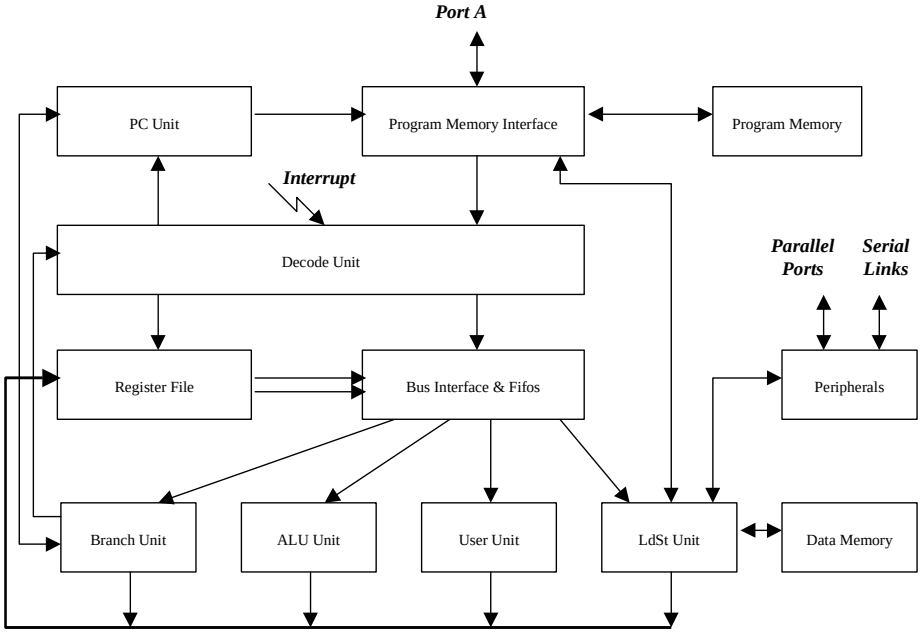


Figure 6-18 : Architecture du processeur Aspro.

Afin d'optimiser les performances, la majeure partie des blocs du processeur a été conçue pour respecter un temps de cycle de 2.5 ns avant placement & routage. En supposant que l'architecture du processeur soit optimale (paragraphe 6.3) en terme de ratio temps de cycle / latence des boucles et nombre d'instructions dans le pipeline, les performances des blocs pourraient offrir une vitesse globale de 2.5 ns avant routage. Avec la technologie considérée (CMOS 0.25µm, STMicroelectronics), un rapport d'environ deux entre les performances avant et après placement & routage a été notre hypothèse de travail. Nous visions ainsi un temps de cycle de 5 ns, soit 200 Mips après fabrication.

Tous les blocs critiques du processeur ont été conçus afin de respecter ce temps de cycle local de 2.5 ns. Tout d'abord, la boucle de Fetch avec l'unité PC, l'interface mémoire, la mémoire programme et le décodeur instruction doit impérativement respecter cette contrainte car ces blocs sont utilisés pour chaque instruction. En raison de sa latence importante (en particulier en raison du temps d'accès mémoire), cette boucle de Fetch contient en continu trois instructions. Ainsi, en dehors des instructions de branchement qui sont spécifiques, la boucle de Fetch se comporte de manière équivalente à l'exemple présenté dans le paragraphe

6.3. La latence globale de la boucle ne doit pas dépasser 7.5 ns afin que la boucle ne soit pas dans un mode de fonctionnement limité par les données. Dans ce cas, on obtient un mode de fonctionnement du type P / Lf = 7.5 ns / 2.5 ns = 3 instructions.

L'architecture mémoire est critique car représente une part importante de la latence dans la boucle de Fetch. L'entrelacement de requêtes mémoires à des sous blocs consécutifs (paragraphe 5.1) permet en limitant la latence à 3.5 ns avant placement & routage de respecter le temps de cycle local de 2.5 ns en moyenne. Dans ce cas, la boucle de Fetch est optimale, elle distribue des instructions au chemin de données via le décodeur instruction avec un temps de cycle de 2.5 ns.

Cependant, lors d'un branchement, l'unité PC est suspendue, elle se synchronise et attend un résultat de l'unité de branchement (résultat de branchement ou adresse de branchement) : la boucle de Fetch est suspendue le temps de résoudre le branchement. La latence de branchement doit être la plus faible possible afin de réduire cette pénalité. Un temps d'accès des registres de 2 ns cumulé avec la latence de branchement comprise entre 0.4 et 1.9 ns (en fonction des opérandes) représente une latence de branchement entre 2.4 ns et 3.9 ns. Ainsi, la boucle de Fetch montre une pénalité de une à une instruction et demi lors d'un branchement. On voit ici l'intérêt d'effectuer du calcul en temps moyen (paragraphe 5.2.3) en fonction des opérandes. Par rapport au cas synchrone, cette suspension est faible en comparaison du nombre d'étages de pipeline pour calculer ce même branchement.

Ensuite, le chemin de données a été optimisé afin de respecter de la même manière un temps de cycle moyen de 2.5 ns. L'ensemble du banc de registres et de l'interface bus respecte cette contrainte puisque ces blocs sont utilisés pour toute instruction. Les unités d'exécution présentent des temps de cycle très variables, compris entre 2 ns et 6 ns, cependant les étages d'entrées de ces unités offrent toujours des débits rapides. Ceci permet de distribuer à vitesse maximale les opérandes vers les différentes unités d'exécution, et de les stocker le temps de l'exécution. Les unités peuvent présenter en interne des débits faibles comme dans l'unité de multiplication-accumulation (6.5 ns) ou l'unité arithmétique et logique (5 ns). Pour cette dernière, grâce à l'entrelacement de différents sous-blocs (paragraphe 5.2.1), un débit moyen élevé entre les unités de décalage / comparaison / addition / logique peut être obtenu même si chaque bloc présente un débit faible.

Une fois les opérandes distribués à vitesse maximale dans le chemin de données, le mécanisme de réécriture dans le désordre tire parti des différences de latence entre les unités d'exécution. La latence des unités varie entre 1.5 ns pour un Ldi (load immediate, via l'unité de branchement) et 8 ns pour un Ldb/w (load byte/word en mémoire donnée via l'unité load-store). Ainsi, le pipeline d'exécution sera rempli de manière optimale, avec un temps de cycle de 2.5 ns, tant qu'un aléa de lecture n'oblige pas une synchronisation dans le banc de registres.

Vu la disparité de latence entre les unités, il est alors nécessaire d'introduire de 0 à 2 instructions entre les instructions qui présentent une dépendance pour éviter toute suspension du pipeline. Dans le cas contraire, le banc de registres impose localement la synchronisation, le registre attend la fin de l'exécution de l'instruction pour écrire son résultat avant de pouvoir envoyer l'opérande de l'instruction suivante. Comme précisé dans le paragraphe 4.4, il est donc nécessaire d'avoir une bonne technologie de compilation pour analyser le parallélisme instruction afin de tirer parti de l'architecture du processeur et du parallélisme potentiel entre les unités.

6.4.2. Implémentation physique du processeur

Une fois les différents blocs du processeur assemblés, des validations fonctionnelles et des estimations de performances ont été effectuées. Comme présenté dans le chapitre 3, ceci a été effectué en simulant en VHDL la netlist niveau porte du processeur. Le placement & routage de l'ensemble du circuit a alors été réalisé ainsi que les optimisations électriques nécessaires afin d'ajuster les ratio fan-in/fan-out des portes logiques aux capacités de routage. Après placement-routage, le circuit a été validé en simulant en VHDL la netlist back-annotée par les capacités de routage.

D'une manière générale, nous nous sommes rendu compte que le facteur de perte lié au routage était plus élevé que ce que nous avions estimé au préalable. De plus, nous ne disposions pas de cellules standard avec des « *drive* » suffisamment élevés. Tandis que la bibliothèque du fondeur dispose classiquement de cellules standard avec des « drive » de 1 à 4, nous ne disposions seulement de portes de Muller avec des « drive » de 1 à 2. L'insertion de buffer sur les signaux a permis de rendre le circuit correct et fonctionnel au niveau électrique mais a dégradé les performances. En particulier, le routage de certains blocs, comme le banc de registres, a posé problème et a dégradé fortement les performances localement sur ces blocs critiques et donc globalement. De la même manière, nous avons observé une perte de latence dans la boucle de Fetch, celle-ci est alors apparue sous optimale car limitée par les données.

En raison du manque de temps et de ressources lors de la finalisation du circuit, nous n'avons pas pu effectuer toutes les estimations et optimisations de performances nécessaires sur l'ensemble du processeur après placement & routage. Des estimations de performances précises sur les différents blocs et sur les principales boucles fonctionnelles auraient été nécessaires pour savoir identifier et optimiser les chaînes critiques.

Pour optimiser le circuit, nous aurions dû effectuer un réel placement & routage par bloc et non un placement par zone, et aussi disposer de cellules dans notre bibliothèque avec des « *drive* » plus élevés. Certains blocs critiques auraient pu être reconçu (décomposition avec ajout d'étages de pipeline) afin d'optimiser localement le débit de certaines boucles. Afin de ne pas manquer un « départ de masque », il a été décidé d'envoyer le circuit au plus tôt en l'état, sachant qu'il était validé et fonctionnel.

Le Tableau 6-1 et la Figure 6-19 présentent respectivement le nombre de transistors par bloc et le placement des différentes zones hiérarchiques du processeur. La technologie est une technologie CMOS 0.25µm, 6 niveaux de métal, 2.5V, de STMicroelectronics.

Décodeur	13300
Unité PC	15600
Interface mémoire programme – port A	11800
Mémoire programme	24400
Banc de registre + Interface bus	205000
Unité Alu	44000
Unité Mac	83000
Unité Branchement	13600
Unité Load-Store	19000
Périphériques	29100
Mémoire Donnée	27400
Total ASPRO	**486 200**

144 plots (57 entrées, 48 sorties, 39 supply)	15000
Blocs SRAM (4*4k-24bits+8*8k-8bits)	**5 783 600**
Total : ASPRO + SRAM + Plots	**6 284 800**

Tableau 6-1 : Nombre de transistors (total et par bloc) du processeur

Le processeur est constitué d'environ 6.3 Millions de transistors, dont 500 000 pour le cœur logique et 5.8 Millions pour les mémoires. Le circuit contient au total 112 koctets de mémoire avec 12 blocs de mémoires SRAM différents (4 blocs de 4k mots 24-bit pour la mémoire programme et 8 blocs de 8koctets pour la mémoire donnée).

La surface totale du circuit est de 40 mm^2. Les mémoires représentent une surface de 17 mm^2, tandis que le cœur du processeur représente une surface de 8 mm^2 pour environ 44 000 portes logiques. La densité de la logique est donc de 62 500 transistors/mm^2, loin de ce qu'il est possible d'obtenir avec un routage optimisé dans cette même technologie (environ 100 000 transistors/mm^2).

Le circuit a été réalisé avec trois réseaux d'alimentation distincts, un pour le cœur logique, un pour les mémoires et un pour les entrées-sorties, des cellules de conversion de tension ayant été insérées sur les signaux d'interfaces entre ces blocs. Le nombre d'entrées-sorties est de 144 plots, soit 57 plots d'entrées, 48 plots de sortie et 39 plots d'alimentation. Le processeur sera monté dans un boîtier de type PGA144.

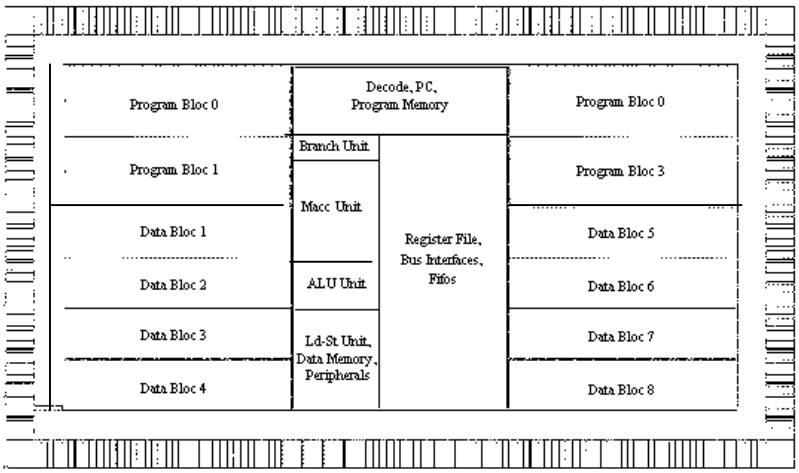


Figure 6-19 : Placement bloc du processeur

Les dernières vérifications physiques effectuées (DRC, LVS), le fichier « *GDSII* » a été généré et envoyé au cours du mois de décembre 1998. Le circuit est revenu après fabrication en avril 1999. La Figure 6-20 montre une photographie du processeur après fabrication.

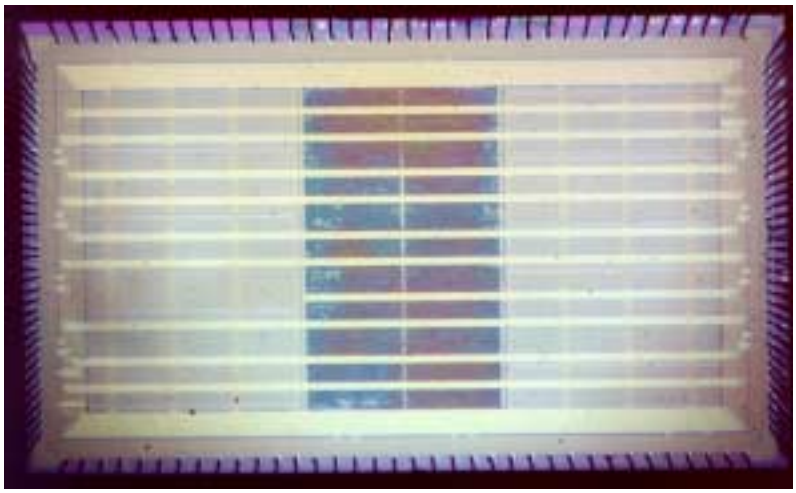


Figure 6-20 : Photographie du processeur.

6.5. Test et mesures de performances sur le circuit

6.5.1. Test du circuit

Afin de tester le circuit, plusieurs questions se posaient : tout d'abord quel modèle de faute et quels vecteurs de test appliqués et comment tester ce circuit avec un testeur synchrone.

- **Test de la logique QDI**

Le test des circuits asynchrones dépend fortement du type de circuit asynchrone considéré. La logique quasi-insensible aux délais se prête bien au test en raison de l'absence d'hypothèse de délais en dehors des fourches isochrones [DAVI 95], [HULG 95], [MART 91]. Si on choisit un modèle de test avec collage sur les entrées, grâce à l'encodage double-rail et au protocole quatre-phase que nous avons implémentés, le circuit est auto-testable.

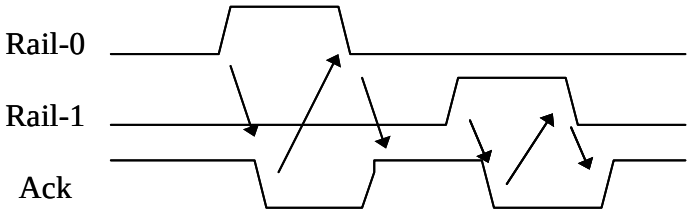


Figure 6-21 : Protocole quatre phase et encodage double rail : envoi de '0' puis '1'.

Il suffit pour valider les entrées-sorties de chaque porte logique d'exciter le chemin électrique correspondant avec un vecteur de test adapté. La phase montante du protocole (Figure 6-21) permet de vérifier le non-collage à zéro de l'entrée de la porte, la phase de remise à zéro du protocole permet de vérifier le non-collage à un de cette même entrée. Si on effectue ceci pour les valeurs '0' et '1' de chaque bit, on vérifie le non collage de chaque rail

des signaux double-rail des canaux. Comme le signal d'acquittement effectue aussi une transition montante et descendante dans chaque cycle, si le protocole se déroule correctement pour chaque bit de donnée, on vérifie de plus le non-collage des signaux d'acquittement.

L'insensibilité aux délais permet de s'affranchir du test de fautes temporelles. Pour que le circuit soit fonctionnel, il faut vérifier et observer un enchaînement correct des transitions du protocole pour chaque canal de communication [HULG 95]. Ainsi, pour un additionneur à propagation de retenue uniquement constitué de cellules de Full-Adder, il suffit de pouvoir observer que la somme d'opérandes égaux à $0000 et d'une retenue entrante à 0 produit la valeur $0000 et respectivement que la somme de deux opérandes égaux à $FFFF et d'une retenue entrante à 1 produit la valeur $FFFF. Pour une structure aussi simple que celle-ci, il suffit de deux vecteurs de test pour valider le non collage des signaux de tous les canaux de communications.

Le test du protocole vérifie que les entrées-sorties des cellules logiques ne sont pas collées (ni à '1', ni à '0'), mais ceci ne teste pas nécessairement l'ensemble de la table de vérité de chaque cellule. Comme la cellule d'addition est une cellule complexe (paragraphe 3.3), toutes les branches CMOS de la cellule ne sont pas testées par ce type de test. Pour cela, il faudrait plus de vecteurs de test pour couvrir tous les nœuds internes de la cellule. Cependant, pour toutes les autres portes, y compris les portes de Muller, il n'y a pas de problème, une transition à 1 et à 0 de sa sortie valide complètement les transitions de ses entrées et toute sa table de vérité. Les autres portes logiques sont uniquement des portes and/or à *n* entrées. Celles-ci sont toujours utilisées de manière monotone, une seule entrée parmi *n* monte et redescend (comme dans les processus multiplexeur / démultiplexeur, paragraphe 6.1), ce qui permet de tester une par une toutes les branches CMOS de ces éléments. Par exemple, dans le schéma du processus « and double rail deux entrées» (paragraphe 3.1.2), il faut trouver les vecteurs qui excitent tous les min-termes de la logique, soit quatre vecteurs de test (que ce soit pour la version pipelinée WCHB ou combinatoire). Ainsi, en dehors du cas de porte complexe comme le Full Adder, grâce au protocole et à notre choix de décomposition logique, l'utilisation d'un vecteur de test qui excite chaque porte permet de tester l'ensemble de la logique.

- **Génération des vecteurs de test**

Après synthèse du circuit, nous avons recherché par bloc les stimulis qui permettent d'exciter toute la logique. Même si cela semble fastidieux, cette tâche s'est révélée relativement facile en s'y appliquant de manière systématique. Dans une très grande majorité des cas, l'utilisation des vecteurs $0000 et $FFFF pour chaque donnée d'instruction permet de tester la logique correspondante. Par exemple pour le multiplieur, il suffit de 16 vecteurs de test pour couvrir toute la structure (les 4 tests 00*00, 00*FF, FF*00, FF*FF couvrent en très grande partie toute la logique, les 12 autres vecteurs permettent de couvrir les cas particuliers dus au calcul non signé). Pour des structures plus régulières, la génération des vecteurs de test est plus facile. Par exemple dans le cas du banc de registres et de l'interface bus, il faut vérifier par registre que l'écriture et la lecture des deux valeurs 0000 et FFFF s'effectuent correctement. Il faut aussi vérifier que le contrôle associé à chaque registre est couvert, c'est à dire que les commandes lecture / écriture / lecture-puis-écriture sont aussi testées. On peut alors associer les tests entre eux pour couvrir le plus de matériel et réduire le nombre de vecteurs de test.

Nous avons ainsi écrit un programme de test en assembleur d'environ 350 instructions. Afin de pouvoir vérifier de l'extérieur le bon déroulement du programme de test, ce dernier écrit une signature sur le port parallèle PO. Même si ce programme de test a été développé à la main, nous pouvons considérer que la couverture du programme de test est quasi-complète.

- **Utilisation du testeur**

Lorsque le signal de reset devient inactif, le processeur exécute le programme à l'adresse $FFF0, c'est à dire sur le port A du processeur. Nous utilisons donc le port A avec le testeur pour alimenter le processeur en instructions. Au niveau du testeur, il suffit d'utiliser une horloge suffisamment lente pour pouvoir écrire des valeurs sur ce port et contrôler les signaux de requête-acquittement correspondants. Le programme de test exécuté par le processeur est fournie par le testeur via le port A. Cependant, pour pouvoir effectuer des mesures de performances, il est nécessaire d'exécuter le programme à tester en interne. Pour cela au démarrage, une routine copie tout d'abord le programme de test en mémoire programme interne, puis l'exécute. Afin de mesurer les performances depuis l'extérieur, ces programmes effectuent des boucles qui incrémentent le port parallèle PO. Il suffit alors de mesurer avec le testeur la fréquence d'écriture.

6.5.2. Mesures de performances

Sur banc de test, nous avons observé que le processeur était parfaitement fonctionnel au premier silicium [RENA 99c]. Il tolère une plage de fonctionnement de 0.65V à 2.5V. Ceci est la preuve que les circuits QDI sont parfaitement robustes aux variations de tensions.

Néanmoins, nous avons rencontrés des difficultés par rapport au système d'alimentation, il ne nous a pas été possible de diminuer la tension d'alimentation du cœur du processeur indépendamment de celle des mémoires en raison sans doute de problème électrique avec les cellules de conversion de niveau de tension. Cependant, les mémoires SRAM se comportant bien ainsi que les interfaces synchrone / asynchrone (l'hypothèse de délai reste respectée dans toute la gamme de tension), nous avons pu observé que le circuit était fonctionnel jusqu'à 0.65V, mémoires comprises. Les mesures que nous présentons par la suite sont uniquement effectuées jusqu'à 1V en raison de problème avec le testeur.

Tension d'alimentation	Temps de cycle (ns)	Débit (Mips)	Courant coeur (mA)	Courant SRAM (mA)	Courant total (mA)	Puissance (mW)	Coeff. mW/MIPS
1	30,4	32,9	8	11	19	19,0	0,58
1,2	18,8	53,2	18	18	36	43,2	0,81
1,4	13,6	73,5	26	24	50	70,0	0,95
1,6	10,1	99,2	35	30	65	104,0	1,05
1,8	9,0	110,6	46	35	81	145,8	1,32
2	7,8	127,6	57	39	96	192,0	1,51
2,3	6,8	147,1	77	45	122	280,6	1,91
2,5	6,2	160,3	90	50	140	350,0	2,18

Tableau 6-2 : Mips et consommation en fonction de la tension d'alimentation pour une série de NOP.

Le Tableau 6-2 présente les mesures effectuées pour un programme assembleur consistant en une série de NOP. Cette routine nous permet de mesurer les performances de la boucle de Fetch du processeur. En effet, pour un NOP, aucune synchronisation n'est effectuée avec le reste du processeur. Pour la tension nominale de 2.5V, le processeur effectue les NOPs à 160 Mips pour une consommation de 350 mW. La consommation du cœur est un peu moins du double de celle des mémoires SRAM. A 1 Volt, le processeur réalise encore 33 Mips pour une consommation de 20 mW.

	Temps de cycle (ns)	Mips
Test sous-unité addition	6,90	144,9
Test sous-unité décalage	6,98	143,3
Test sous-unité logique	6,98	143,3
Test sous-unité comparaison	8,71	114,9
Test unité ALU (instr. mixtes)	7,02	142,4
Test unité load-store : chargements mémoire	6,90	144,9
Test unité load-store : rangements mémoire	6,90	144,9

Tableau 6-3 : Performances de différentes unités du processeur.

Le Tableau 6-3 présente les mesures de performances de quelques instructions du processeur. Ces mesures sont effectuées pour des instructions successives sans aucune dépendance sur les noms de registres concernés. Nous avons mesuré les performances des différentes sous-unités de l'unité arithmétique et logique (cf. paragraphe 5.2.1). Les trois sous-unités d'addition, de décalage et logique fonctionnent chacune à environ 140 Mips. La sous-unité de comparaison est un peu plus lente et fonctionne seulement à 115 Mips. Lorsqu'on entrelace les instructions dans l'unité arithmétique et logique en envoyant en séquence les instructions dans les quatre sous-unités concernées, on obtient une vitesse de 140 Mips. Ceci prouve que le mécanisme d'entrelacement utilisé au sein de l'unité arithmétique et logique fonctionne, car il permet de cacher le débit de la sous-unité la plus lente. Cependant il ne permet pas d'améliorer au delà les performances de l'unité. Les mesures effectuées sur l'unité Load-Store avec l'exécution de chargement ou rangement successifs montrent que les performances du processeur sont aussi égales à 140 Mips.

Ceci nous amène à conclure que les performances du processeur sont limitées par le banc de registres, ce qui coïncide avec les problèmes que nous avons rencontrés lors du placement & routage. Ainsi, même dans le cas où la succession d'instructions ne présente aucune dépendance sur les noms de registres, nous n'obtenons pas les 160 Mips correspondant à l'exécution de NOPs. Le processeur n'est pas limité par un aléa avec suspension du pipeline et blocage sur un registre ou par un débit local trop faible dans une unité mais plutôt par le débit local du banc de registres. Le débit crête offert par le processeur est donc de 140 Mips, aussi bien pour des instructions arithmétiques que pour des rangements ou chargements en mémoire données.

Nous présentons ensuite dans le Tableau 6-4 les mesures effectuées pour un programme consistant en une routine de filtrage de type RIF (*Réponse Impulsionnelle Finie*). Cette routine assembleur effectue une boucle avec itération contenant une multiplication-accumulation, des chargements mémoires, l'incrémentation d'un pointeur et d'un compteur puis un branchement retardé. Comme présenté dans le paragraphe 4.4, cette routine

assembleur a été optimisée afin de supprimer tout aléa entre les registres : les instructions ont été ordonnancées afin d'obtenir un recouvrement maximum entre les unités d'exécutions.

Tension d'alimentation	Temps de cycle (ns)	Débit (Mips)	Courant cœur (mA)	Courant SRAM (mA)	Courant total (mA)	Puissance (mW)	Coeff. mW/MIPS
1	42,3	23,7	16	11	27	27,0	1,14
1,2	25,9	38,6	28	18	46	55,2	1,43
1,4	18,6	53,8	42	24	66	92,4	1,72
1,6	14,9	67,0	60	29	89	142,4	2,13
1,8	12,5	79,8	80	34	114	205,2	2,57
2	11,0	91,0	100	40	140	280,0	3,08
2,3	9,4	106,0	135	45	180	414,0	3,91
2,5	8,7	115,0	160	50	210	525,0	4,58

Tableau 6-4 : Mips et consommation en fonction de la tension d'alimentation pour une routine de type RIF.

A 2.5V, les performances obtenues sont de 115 Mips pour une consommation de 525 mW. Nous notons une dégradation de performance par rapport au 140 Mips précédent. Ceci s'explique non pas en raison de suspension de pipeline du à des aléas sur les registres, mais en raison du branchement. Cette routine exécute un branchement toutes les 7 instructions, nous observons donc une suspension de la boucle de Fetch d'environ une instruction et demi (7*(140/115-1)=1.52). Ce résultat correspond à ce que nous avions estimé au niveau de l'architecture (paragraphe 6.4.1).

Outre la pénalité de branchement observée, les performances de cette boucle prouve que le mécanisme de réécriture dans le désordre au sein du banc de registres est efficace : les instructions s'entrelacent correctement entre les unités concernées. Il serait nécessaire de mesurer les performances de cette même boucle mais sans optimisation.

La Figure 6-22 présente les Mips, la consommation du cœur du processeur et des mémoires pour différentes tensions. A 2.5V, les mémoires SRAMs consomment environ un tiers de moins que le cœur. A une tension de 1V, le processeur montre encore une performance de 23 Mips pour 27 mW.

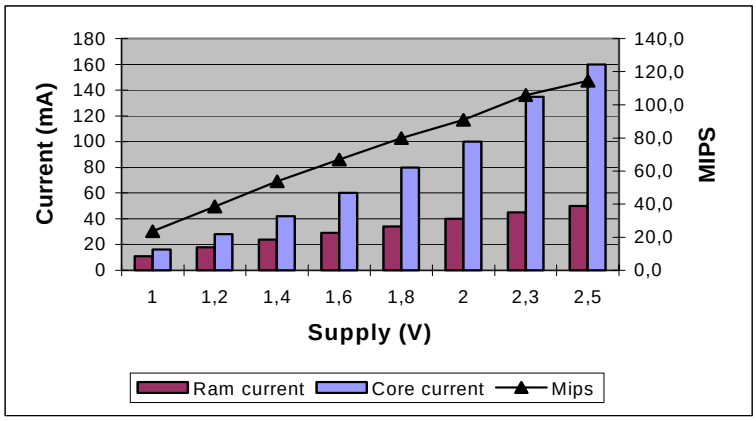


Figure 6-22 : Mips, consommation du cœur et des mémoire du processeur en fonction de la tension pour une routine de type RIF.

Par rapport à l'évolution en tension, nous pouvons noter que la courbe de Mips est légèrement sous-linéaire, tandis que celle de la consommation est sur-linéaire. Il est donc possible de profiter de cette propriété en ajustant la tension d'alimentation pour obtenir le meilleur rapport Mips/mW en fonction des performances voulues. Le meilleur rapport Mips/mW sera donc obtenu à basse tension. Le compromis vitesse / consommation offert par les circuits asynchrones peut être réellement intéressant en fonction de l'application visée, ce compromis avec transfert d'énergie est bien sûr rendu possible par la grande robustesse de ces circuits.

Enfin, nous avons aussi validé et mesuré les performances des liens séries du processeur. Afin de mesurer le débit maximum au testeur, le test a consisté à connecter l'un des ports d'entrée sur un port de sortie (par exemple le port série NI avec NO). Les liens séries sont fonctionnels et offrent à 2.5V un débit de 50 Mbit/s. Ce niveau de performance montre que le protocole deux phase quasi-insensible aux délais utilisé est tout a fait adapté à ce type de périphérique, ceux-ci permettront de mettre en œuvre des applications multi-processeurs performantes.

6.6. Conclusion

Le processeur Aspro a été testé avec succès, il a été fonctionnel au premier silicium. Les mesures que nous avons présentées montrent des performances plus faibles que celles auxquelles nous nous attendions. Nous avons en effet préféré conclure le projet et ne pas manquer un départ en fonderie plutôt que de continuer des estimations et optimisations après placement & routage. Par la suite, il serait peut être envisageable d'effectuer une reprise du circuit afin d'optimiser ses performances. En particulier, il serait nécessaire de disposer de cellules complexes pour implémenter le banc de registres, ceci diminuerait sa complexité en terme de nombre de transistors et faciliterait le placement & routage. De plus, comme la structure du banc de registres est régulière, il serait aussi possible d'effectuer un placement manuel de celui-ci.

Les mesures que nous avons présentées dans le paragraphe précédent sont relativement succinctes. Nous regrettons de ne pas avoir eu assez de temps pour effectuer plus de mesures sur le processeur. Il aurait été intéressant de mesurer les performances de chaque unité, chaque instruction, en débit et en latence, afin de mesurer les suspensions en cas d'aléas dans le banc de registres ou en cas de branchements. Ces mesures et informations sont en effet nécessaires si on veut par la suite optimiser la génération de code afin de profiter de l'architecture et des performances du processeur.

Les résultats que nous avons présentés précédemment sont basés sur l'évaluation de performances de quelques routines assembleurs, ils sont donc à nuancer. En effet, nous ne disposions pas à l'époque de compilateur C pour le processeur, nous ne pouvions pas exécuter des programmes de référence. Il est donc délicat de comparer directement nos résultats avec des résultats basés sur des « benchmark » standard de type « Dhrystone ». Néanmoins, nous pouvons essayer de comparer notre processeur avec d'autres processeurs asynchrones existants (Tableau 6-5). Ces processeurs ont été présentés dans le chapitre 1 du manuscrit.

	mW	Mips	Volt	mw/Mips	Techno.	Style de circuit asynchrone
Aspro (2.5v)	500	140	2,5	3,6	0,25	16-bit, QDI, std-cell
Aspro (1v)	27	24	1	1,1		
Caltech 1st	225	18	5	12,5	1,6	16-bit, QDI, full custom
MiniMips (3.3v)	4000	170	3,3	23,5	0,6	32-bit, QDI, full custom
MiniMips (1,6v)	220	60	1,6	3,7		
Phillips 80c51	9	4	3,3	2,3	0,5	8-bit, µ–pipe., std-cell
Amulet2e	150	42	3,3	3,6	0,5	32-bit , µ–pipe., std-cell + full custom
Titac2	2100	54	3,3	38,9	0,5	32-bit, SDI, std-cell + full custom

Tableau 6-5 : Performances des différents processeurs asynchrones existants

La comparaison de ces résultats montre que la logique quasi-insensible aux délais permet d'atteindre des performances en vitesse élevées. Le processeur MiniMips [MART 97] conçu en circuit QDI full-custom en technologie 0.6µm est le plus rapide avec une performance de 170 Mips. Aspro est le deuxième processeur asynchrone le plus rapide avec 140 Mips crête. Les autres processeurs comme l'Amulet2e [FURB 97] ou Titac2 [TAKA 97], conçus partiellement en standard-cell et en full-custom, sont nettement plus lents. Même avec une technologie d'écart (0.5µm au lieu de 0.25µm), on peut observer que le principe du micro-pipeline ne permet pas d'atteindre des performances très élevées. Au niveau consommation, Aspro montre un ratio en mW/Mips équivalent à celui de l'Amulet2e et bien supérieur à celui de Titac2, pour une vitesse beaucoup plus élevée. Il est donc difficile de conclure, sachant que les technologies sont différentes, que le processeur Aspro est un processeur 16-bit et non 32-bit, et que l'Amulet2e a été partiellement optimisé avec un chemin de données en full-custom. Le microcontrôleur 80c51 de Philips [GAGE 98] est difficilement comparable avec les autres processeurs puisqu'il ne correspond pas à une architecture optimisée pour la vitesse.

Le processeur Aspro montre donc des résultats très prometteurs par rapport aux autres processeurs asynchrones existants avec 140 Mips crête pour une consommation de 500 mW. Ces résultats sont la preuve que notre méthodologie de conception et que la logique quasi-insensible aux délais implémentée en cellules standard montre des performances intéressantes à la fois en vitesse et en consommation.

Pour conclure sur le niveau de performance de ce premier prototype, il serait aujourd'hui nécessaire de terminer les outils de compilation et d'effectuer plus de mesures de performances et ce sur des benchmarks existants. La conception d'une carte test pour le processeur Aspro a été commencée à France Telecom R&D, le développement est en cours de finition au laboratoire TIMA [RENA 00a]. Cette carte de test contient un processeur Aspro, des mémoires Flash connectées au port A pour le boot, des connexions avec des ports parallèles pour faire de l'acquisition et du traitement d'images, une connexion aux liens séries du processeur pour télécharger des programmes et des données via un FPGA et le port parallèle d'un compatible PC (Figure 6-23).

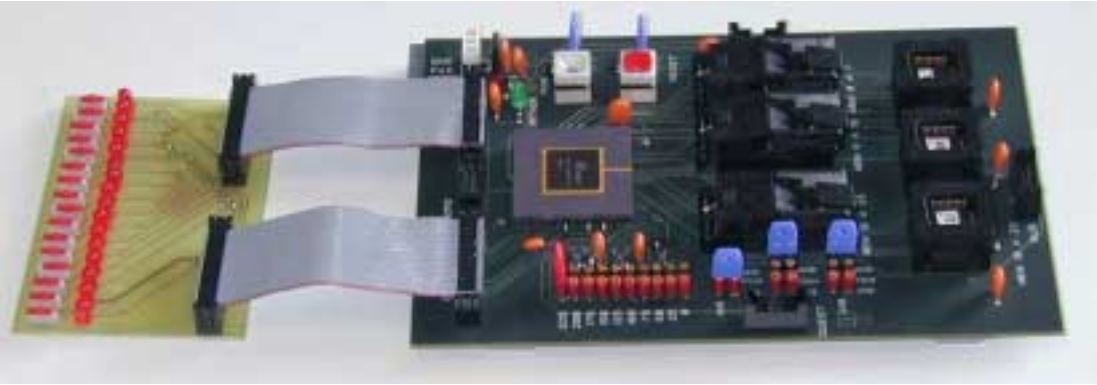


Figure 6-23 : carte de test pour le processeur Aspro.

Conclusion

- **Une méthodologie de conception de circuits asynchrones quasi-insensibles aux délais**

Dans la première partie du manuscrit, nous avons présenté une méthodologie de conception de circuits asynchrones qui s'insère dans les flots de conception existants. A partir d'une description en langage CHP, cette méthodologie permet de concevoir des circuits quasi-insensibles aux délais implémentés en cellules standard, le langage CHP étant parfaitement adapté à la spécification de systèmes asynchrones, le langage VHDL étant utilisé comme plate-forme de simulation / validation.

En terme d'outil et de méthodologie, nos contributions sont les suivantes. Nous avons développé et présenté dans le chapitre 2 un traducteur CHP vers VHDL qui offre le moyen de simuler des modèles CHP en utilisant les outils de simulation VHDL. Il est alors possible de simuler dans un même langage différents niveaux d'abstractions : modèle CHP asynchrone, netlist niveau cellules standard, blocs synchrones ou toute autre bibliothèque HDL. Cet outil aide le concepteur dans toutes les étapes de validation, que ce soit avant ou après synthèse.

D'une manière plus générale, cet outil s'inscrit dans le cadre d'un flot de conception plus large : non seulement cela offre aux concepteurs de circuits synchrones le choix d'intégrer des circuits asynchrones et réciproquement, mais aussi il devient possible de simuler dans un environnement de simulation unique des circuits synchrones et asynchrones, de les étudier et comparer conjointement leurs différentes architectures. Cet outil est ainsi un premier pas vers des environnements de conception / simulation qui vise la conception de systèmes mixtes synchrones-asynchrones en cherchant à tirer parti des avantages de chacun des styles. C'est en particulier adapté à l'étude de systèmes de type Globalement Asynchrone – Localement Synchrone (GALS), solution actuellement envisagée par nombre d'industriels pour la conception de « *System-on-Chip* ».

Au niveau de la synthèse logique, nous avons effectué différents choix au niveau du langage CHP afin qu'il soit synthétisable, des protocoles de communication (protocole symétrique) et de la décomposition logique (décomposition de type DIMS) qui conduisent à

implémenter des circuits en cellules standard. La bibliothèque de cellules standard que nous avons définie contient principalement des portes de Muller ainsi que quelques portes plus spécifiques. Les circuits asynchrones quasi-insensibles aux délais obtenus sont alors uniquement réalisés avec des portes de Muller et des portes logiques de type and / or.

Nous disposons pour chaque cellule des vues transistor, layout et VHDL, ce qui offre l'accès aux flots de conception usuels : placement & routage et simulation back-annotée. Deux types de protocoles sont disponibles en utilisant de la logique combinatoire ou pipelinée, ce qui offre au concepteur un choix d'optimisation en performance. La méthode de synthèse présentée reste cependant manuelle à ce jour, avec saisie de schéma dans un éditeur graphique.

- **Etude et conception d'un processeur RISC 16-bit**

Parallèlement au développement du flot de conception, nous avons étudié et conçu un processeur RISC 16-bits scalaire. Le processeur Aspro est un processeur d'usage général qui intègre une mémoire programme et une mémoire données distinctes, un banc de registres, un niveau d'interruption, des unités classiques de traitement arithmétique, branchement et chargement mémoire ainsi qu'une unité spécialisée de multiplication-accumulation et des périphériques dédiés (liens séries implémentant une communication quasi-insensible aux délais).

L'étude du processeur a consisté à exploiter au maximum les bénéfices que le mode de fonctionnement asynchrone peut apporter à une architecture de processeur. L'architecture originale du processeur présentée dans le chapitre 4 autorise l'envoi des instructions dans l'ordre et leur terminaison dans le désordre. Grâce à quatre unités d'exécutions indépendantes et à un simple mécanisme de réservation au sein du banc de registres, les instructions sont envoyées dans les unités, s'exécutent à leur propre rythme en fonction de l'unité et des données concernées puis se terminent dans le désordre dans le banc de registres suivant la dépendance entre instructions.

L'étude détaillée de la micro-architecture du processeur dans le chapitre 5 nous a montré qu'il est possible par décomposition de programmes CHP d'introduire du parallélisme et d'obtenir des unités d'exécutions à temps de calcul et consommation minimum en fonction à la fois des données et des instructions. De nombreux exemples au sein du processeur montrent qu'il est aisé de décorréler les paramètres de latence et de débit au sein d'une architecture asynchrone. Ainsi, l'utilisation d'un mécanisme d'entrelacement, effectué localement suivant la disponibilité des données et des ressources de calcul, permet de profiter du parallélisme instruction, ce qui est source de performance.

D'une manière générale, au cours de l'étude du processeur nous avons observé une grande flexibilité des communications et synchronisations au sein de l'architecture ainsi qu'une importante localité de la conception. Ces propriétés sont supportées par la sémantique de communication du langage CHP, ce sont des facteurs déterminants d'aide à la conception. La localité de la conception permet d'implémenter efficacement un contrôle local et d'optimiser localement les performances tout en conservant la correction fonctionnelle du système. Le concepteur obtient au final une architecture fonctionnant avec un modèle flot de données.

Cette étude d'architecture de processeur RISC illustre qu'il est possible d'obtenir une architecture originale grâce à une implémentation asynchrone. Ceci prouve que le spectre d'architecture asynchrone est beaucoup plus large qu'en mode synchrone en raison de l'absence de contrainte de synchronisation globale, hypothèse extrêmement réductrice. Ceci

est bien sûr en faveur du choix de l'asynchrone pour l'intégration des systèmes complexes dans les technologies avancées. D'autres études seraient envisageables afin d'analyser d'autres types d'architectures : processeur super-scalaire, processeur VLIW, bus de communication totalement asynchrone, etc.… et les bénéfices que le mode de fonctionnement asynchrone pourrait apporter.

L'implémentation du processeur en logique quasi-insensible aux délais a été effectuée avec le flot de conception proposé. Le circuit fabriqué dans une technologie CMOS 0,25µm de STMicroelectronics a été testé et mesuré avec succès. Comme présenté dans le chapitre 6, le processeur Aspro est le deuxième processeur asynchrone le plus rapide avec 140 Mips crête pour une consommation de 500mW.

Les performances obtenues montrent que les choix de « *mapping* » technologique (cellules de la bibliothèque), de protocole de communication (logique combinatoire ou logique pipelinée) et de CHP synthétisable, ont conduit à une implémentation efficace de tous les blocs du processeur (blocs de contrôle, de calcul ou interfaces avec l'extérieur).

Ces résultats sont la preuve que notre méthodologie de conception est pertinente : il est possible d'atteindre de haut niveau de performances avec des circuits asynchrones quasi-insensibles aux délais réalisés en cellules standard. Le travail de conception du processeur représente quatre homme.ans (deux personnes sur deux ans), ce qui est relativement peu sachant que la méthode de conception était en cours de définition.

Par ailleurs, les différents exemples de synthèse illustrent la continuité sémantique entre la spécification des blocs en langage CHP et le matériel qui les implémente : l'asynchronisme architectural est supporté par l'asynchronisme niveau circuit. La logique quasi-insensible aux délais utilisée permet d'implémenter un pipeline avec un niveau de granularité extrêmement fin. Comme la logique intègre à la fois le calcul, le protocole de communication et les éléments de mémorisation, celle-ci peut être qualifiée de « super-pipelinée ». L'architecture asynchrone obtenue se comporte comme un ensemble d'anneaux dans lesquels les paramètres de latence et de débit peuvent être optimisés de manière plus ou moins corrélée. Dans un cadre général, l'analyse théorique des anneaux asynchrones et leur optimisation sous contrainte reste un domaine de recherche à explorer même si des principes simples peuvent être appliqués.

Les résultats obtenus pour le processeur sont très prometteurs, toutefois ils sont à nuancer. Ceux-ci ne correspondent pas à des mesures de performances effectuées sur des « benchmark » standard de type « Dhrystone ». Comme nous ne disposons pas de compilateur, il n'est pas possible d'exécuter des programmes de référence pour effectuer une comparaison entre différentes architectures de processeurs (ce qui évalue à la fois compilateur / jeu d'instructions / architecture / implémentation). Il est donc difficile de comparer l'architecture proposée et ses résultats avec d'autres processeurs synchrones ou asynchrones.

De plus, comme nous avons conçu une machine spécifique et non un processeur existant, il n'est pas envisageable de quantifier précisément la méthode de conception en comparant à architectures équivalentes les performances (vitesse / consommation) de deux versions, l'une synchrone, l'autre asynchrone. Ce choix nous a bien sûr offert une grande liberté au niveau conception et étude d'architecture mais est en effet regrettable au niveau de l'évaluation de la méthodologie de conception. Vu cette première expérience de conception, il semble aujourd'hui nécessaire d'effectuer une comparaison à architectures équivalentes pour pouvoir intéresser les concepteurs de circuits synchrones dans un contexte plus industriel.

Enfin, une carte de test pour le processeur Aspro et un ensemble logiciel (assembleur, compilateur, éditeur de liens) sont en cours de développement au laboratoire TIMA. Ces outils permettront prochainement d'exécuter des applications complètes sur le processeur et sa carte de développement. Ceci permettra tout d'abord d'effectuer des mesures de performances plus précises sur le processeur. Ensuite, il sera envisageable de mettre en œuvre des applications plus complexes afin de montrer la flexibilité de synchronisation et de programmation d'une telle architecture. On pourra par exemple concevoir pour des applications de traitement d'images un système multi-processeurs communicant via les liens série de chaque processeur.

- **Une autre expérience de conception**

Dans la suite de l'expérience de conception du processeur Aspro, nous avons étudié et conçu en 1998 avec Marc Renaudin une architecture de microcontrôleur pour une application de carte sans contact. Un système complet a été réalisé au laboratoire de France Telecom R&D Meylan en 1999 [ABRI 00b], [ABRI 01]. Ce système est composé d'une antenne intégrée, d'un module d'émission / réception radio-fréquence qui récupère une tension d'alimentation à travers l'antenne et effectue le transfert des données, d'un interface synchrone / asynchrone et enfin du microcontrôleur asynchrone. Pour cette application télé-alimentée, le style de conception asynchrone offre plusieurs avantages importants. La logique asynchrone offre une consommation faible et n'est pas sensible aux variations de tension d'alimentation, ce qui simplifie la conception de la régulation de tension du système. De plus, le faible bruit émit par le contrôleur l'autorise à fonctionner pendant les phases d'émission / réception de message sans nuire au bloc analogique. Dans le même temps, la société Philips a réalisé un système équivalent à partir de leur microcontrôleur 80c51 asynchrone [KESS 00].

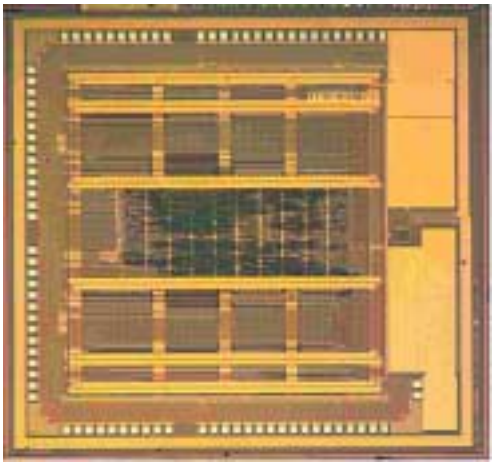


Figure 7-1 : MICABI, une carte sans contact à base de microcontrôleur asynchrone 8-bit.

Le microcontrôleur asynchrone, appelé Mica, est une architecture CISC 8-bit relativement « cossue ». Celui-ci intègre 8 registres données 8-bits , 8 registres d'adresses 16-bits, un registre d'état, 16 koctets de mémoires RAM, 2 koctets de ROM incluant un BIST (« *Built-In-Self-Test* »), des instructions classiques d'arithmétique entière, des ports parallèles pour se

connecter avec des mémoires Flash externes et enfin des liens séries compatibles à ceux du processeur Aspro.

Grâce à l'aide de deux stagiaires [DALL 98], ce microcontrôleur asynchrone a été conçu avec succès en utilisant la méthodologie de conception proposée (Figure 7-1). Afin de réduire la consommation, une optimisation a consisté à implémenter la majeure partie du cœur du microcontrôleur en base 4 (encodé en 4-rail) [ABRI 00a]. Le travail de conception a représenté un homme*an pour le cœur de contrôleur et un homme*an pour les parties analogiques et l'intégration système. Conçu en cellules standard dans la même technologie CMOS 0.25 µm de STMicroelectronics, le cœur du microcontrôleur offre les performances suivantes : 25 Mips pour une consommation de 25 mW à tension nominale de 2.5 Volts et 4.5 Mips pour 800 µW à 1 Volt. L'application complète carte sans-contact est fonctionnelle et a été mise en œuvre dans un démonstrateur dans l'année 2000.

Cette deuxième expérience de conception nous a montré que notre méthode de conception est performante et que le style de conception asynchrone est parfaitement adapté pour ce type d'application. Il est ainsi possible de concevoir un système complexe mixte numérique-analogique, synchrone - asynchrone et de tirer parti des avantages de robustesse, faible consommation, faible bruit liés au mode de fonctionnement asynchrone. Enfin au niveau architectural, le style de conception asynchrone que nous utilisons est non seulement adapté pour concevoir des architectures rapides et parallèles comme le processeur Aspro mais est tout aussi bien adapté pour concevoir des architectures fortement séquentielles tel qu'un microcontrôleur.

• Perspectives

Le principal enjeu au niveau du flot de conception réside aujourd'hui dans la définition et le développement d'un outil de synthèse pour générer automatiquement ce type de circuits. La mise à disposition d'un outil de synthèse est sûrement la condition sine qua non pour que les industriels puissent réellement s'intéresser aux circuits quasi-insensibles aux délais.

Suite aux travaux présentés dans cette thèse, des outils de synthèse sont en cours d'étude et de développement au laboratoire TIMA. La difficulté d'un tel outil de synthèse consiste à savoir formaliser clairement les différents choix de synthèse bas niveau et de savoir développer des algorithmes d'optimisation particuliers à ce type de logique. En effet, les optimisations courantes de l'algèbre de Boole ne sont pas directement applicables car risquerait de violer le modèle de circuit QDI (insertion de portes dans les fourches isochrones).

Dans un tel outil de synthèse, il serait possible d'imaginer des optimisations en vitesse, surface, consommation. La première difficulté consiste à savoir analyser les performances de la logique. Il serait alors envisageable de choisir automatiquement le taux de pipeline à implémenter afin de respecter des contraintes en temps de cycle ou en latence, en fonction des performances locales du bloc considéré ainsi que suivant l'influence des blocs adjacents. Ces outils fourniraient alors des estimations de performances sur la logique générée, ce qui éviterait d'effectuer des simulations post-synthèse.

Afin de tirer parti du mode de fonctionnement en temps moyen, il serait aussi possible d'optimiser la logique générée en fonction de connaissances sur l'application. Ce type d'optimisation n'est pas envisageable dans le cas de la génération de circuits synchrones, par contre c'est une solution qui est utilisée à plus haut niveau pour optimiser la génération de

code sur une machine cible suivant l'application exécutée (« *code profiling* »). Au niveau de la synthèse de circuits logiques asynchrones, ceci pourrait de la même manière s'effectuer en deux étapes : simulation du modèle pour obtenir des probabilités de transition par garde, par couple entrée-sortie, puis rétro-annotation dans la spécification des probabilités obtenues afin que l'outil de synthèse optimise la logique pour les transitions les plus probables.

Au niveau performance de la logique générée, des améliorations seraient envisageables. L'expérience du processeur Aspro nous a montré que la décomposition en portes de Muller pouvait s'avérer coûteuse en surface (on peut penser en particulier au banc de registres). L'utilisation d'un protocole plus optimal tel que le protocole PCHB (cf. paragraphe 3.1) et la définition d'une bibliothèque de cellules étendue pourraient permettre d'obtenir des circuits à priori plus rapides et plus compacts. Une bibliothèque constituée d'un sous-ensemble de portes complexes et de portes de Muller serait une solution pertinente à étudier.

D'autres enjeux existent au niveau du flot de conception. Le premier point consiste dans la vérification des fourches isochrones. La seule hypothèse temporelle des circuits que nous avons générés n'a pour l'instant pas été vérifiée. Il serait nécessaire de savoir identifier les fourches isochrones dans les outils de synthèse afin de garantir au concepteur la correction du circuit généré. Le deuxième point concerne la testabilité des circuits obtenus. Par rapport à ce qui a été présenté dans le paragraphe 6.5, il serait nécessaire d'analyser plus précisément le modèle de faute à adopter. On pourrait alors envisager de générer automatiquement les vecteurs de test, au moins localement par bloc. La problématique du test est un facteur primordial dans un contexte industriel qui doit absolument être adressée. Il faudrait aussi s'intéresser à des solutions de conception pour le test : chaîne de « *scan* » adaptée à un mode de fonctionnement asynchrone, etc…

Au niveau modélisation, le langage CHP et le traducteur CHP_2VHDL se sont avérés de très bons outils de modélisation / simulation. Cependant, afin d'aider le concepteur, il serait nécessaire d'étendre le langage CHP pour offrir des types plus étendus (types énumérés multi-rail), ainsi qu'étendre la simulation de l'opérateur de concurrence « , ». Une solution pertinente serait de traduire vers le langage Verilog puisque celui-ci contient les instructions de parallélisme « fork » et « join ». Par ailleurs, de nouveaux langages et outils de modélisation apparaissent aujourd'hui. Le langage SystemC offre des possibilités de modélisation avec des mécanismes de communication évolués qu'il serait intéressant d'évaluer.

A plus haut niveau, il serait extrêmement intéressant d'adresser le problème de la décomposition architecturale ainsi que l'analyse de performance des systèmes asynchrones et leur optimisation. Ce sont bien sûr des problèmes d'une réelle complexité. Les règles de transformation pour obtenir un modèle CHP synthétisable que nous avons présentées dans le chapitre 3 sont sans aucun doute des règles qui seraient généralisables et automatisables dans un outil de synthèse comportemental. Le principal problème réside dans la définition d'un estimateur de performance au niveau architecture (pour éviter d'avoir à effectuer la synthèse bas niveau) pour pouvoir contraindre et orienter l'outil dans les différents choix de décomposition possibles.

Pour conclure, l'expérience de conception du processeur Aspro ainsi que celle de la carte sans contact montrent des performances très prometteuses. La logique asynchrone quasi-

insensible aux délais semble tout à fait adaptée pour adresser le problème de l'intégration des systèmes complexes dans les technologies avancées. Les avantages intrinsèques du mode de fonctionnement asynchrone tels que robustesse (variations de délais dus aux variations technologiques, de tension, température, couplage entre les interconnexions), modularité, migration, consommation, faible bruit restent actuellement encore relativement qualitatifs. Il sera possible d'en tirer parti à plus grande échelle le jour où les méthodologies de conception seront encore plus étendues et matures.

Publications

Revues d'audience internationale avec comité de rédaction

[ABRI 01] A. Abrial, J. Bouvier, P.Senn, M. Renaudin, P. Vivet, “A New Contactless Smartcard IC using an On-Chip Antenna and an Asynchronous Micro-controller”, à paraître dans IEEE Journal of Solid State Circuit, july 2001.

Revues d'audience nationale avec comité de rédaction

[RENA 97] M. Renaudin, F. Robin, P. Vivet, "AAAA : asynchronisme et adéquation algorithme architecture", Traitement du Signal, numéro spécial, vol. 14, n° 6, Adéquation Algorithme Architecture, pp. 589-604, 1997.

Communications effectuées à des manifestations d'audience internationale avec comité de sélection et actes

[RENA 98a] M. Renaudin, P. Vivet, "CHP2VHDL, a CHP to VHDL translator, towards asynchronous design simulation", Second ACiD-WG Workshop, Torino, January 1998.

[RENA 98b] M. Renaudin, P. Vivet, F. Robin, "ASPRO-216 : A Standard-Cell Q.D.I. 16-Bit RISC Asynchronous Microprocessor", in "International Symposium on Advanced Research in Asynchronous Circuits and Systems", ASYNC’98, San Diego, USA, 30 March- 2 April, 1998.

[RENA 99a] M. Renaudin, P. Vivet, "A Design Experiment : The ASPRO Program Memory", Third ACiD-WG Workshop, New-Castle Upon Tyne, January 1999.

[RENA 99b] M. Renaudin, P. Vivet , F. Robin, "A Design Frame Work for Asynchronous/ Synchronous Circuit Based on CHP to HDL Transaction", In “International Symposium on Advanced Research in Asynchronous Circuits and Systems”, ASYNC'99, Barcelona, Spain, April 19-21, pp 135-144, 1999.

[RENA 99c] M. Renaudin, P. Vivet , F. Robin, “ASPRO : an Asynchronous 16-Bit RISC Microprocessor with DSP Capabilities”, Proceedings of the 25th European Solid-State Circuits Conference, ESSCIRC’99, Duisburg, Germany, 21-23 September 1999, pp 428-431.

[ABRI 00a] A. Abrial, J. Bouvier, M. Renaudin, P. Vivet, "A Contactless Smart-Card Chip based on an Asynchronous 8-bit Microcontroller", 4th AciD Workshop, Grenoble, France, 31st – 1st February, 2000.

[RENA 00a] M. Renaudin, P. Vivet, Ph. Geoffroy, "ASPRO : a toy demo", 4th ACiD Workshop, Grenoble, France, 31st – 1st February, 2000.

[ABRI 00b] A. Abrial, J. Bouvier, P.Senn, M. Renaudin, P. Vivet, “A New Contactless Smartcard IC using an On-Chip Antenna and an Asynchronous Micro-controller”, Proceedings of the 26th European Solid-State Circuits Conference ESSCIRC’00, Stockholm, Sweden, 19-21 September 2000.

Bibliographie

[ABRI 00a] A. Abrial, J. Bouvier, M. Renaudin, P. Vivet, "A Contactless Smart-Card Chip based on an Asynchronous 8-bit Microcontroller", 4th AciD Workshop, Grenoble, France, 31st – 1st February, 2000.

[ABRI 00b] A. Abrial, J. Bouvier, P.Senn, M. Renaudin, P. Vivet, "A New Contactless Smartcard IC using an On-Chip Antenna and an Asynchronous Micro-controller", Proceedings of the 26th European Solid-State Circuits Conference, ESSCIRC'00, Stockholm, Sweden, 19-21 September 2000.

[ABRI 01] A. Abrial, J. Bouvier, P.Senn, M. Renaudin, P. Vivet, "A New Contactless Smartcard IC using an On-Chip Antenna and an Asynchronous Micro-controller", à paraître dans Journal of Solid State Circuit, 2001.

[AIRI 90] R. Airiau, J-M Bergé, V. Olive, J. Rouillard, « VHDL, du langage à la modélisation », Collection Technique et Scientifique des Télécommunications, Presses Polytechniques et Universitaires Romandes,1990.

[AKEL 92] V. Akella, G. Gopalakrishnan, "SHILPA : A high level synthesis system for self-timed circuits", in International Conference on Computer Aided Design (ICCAD), Santa Clara, pp. 587-594, November, 1992.

[ALAD 98] D. Aladenise, « Génération automatique de circuits sans horloge », rapport de stage effectué au laboratoire de France Telecom R&D, Meylan, 1998.

[BAIN 98] Bainbridge W.J., Furber S, « Asynchronous Macrocell Interconnect using Marble », Proceedings Async'98, San Diego, April 1998, pp. 122-132.

[BARD 97] A. Bardsley, D. Edwards, "Compiling the language Balsa to delay-insensitive hardware", in C.D. Kloos and E. Cerny, editors, Hardware description languages and their applications (CHDL), pp. 89-91, April, 1997.

[BAUE 97] J.C. Bauer, E. Closse, E. Flamand, M. Poize, J. Pulou, P. Venier, « SAXO : A retargetable optimized compiler for DSPs », Proceedings of the 8th International Conferance on Signal Processing Applications and Technology, San Diego, CA, Sept. 97, pp. 1032-1036.

[BERK 91] K. van Berkel, J. Keyssels, M. Ronken, R. Saeijs and F. Chalij, "The VLSI programming language Tangram and its translation into handshakes circuits",

Proceedings of the European Conference on Design Automation, Amsterdam, pp. 384-389, 1991.

[BERK 92] K. Van Berkel, "Beware the isochronic fork", Integration, the VLSI journal, N° 13, pp. 103-128, 1992.

[BERK 93] K. Van Berkel, "Handshake Circuits - An Asynchronous Architecture for VLSI Programming", Cambridge University Press, 1993.

[BERK 94] K. Van Berkel, R. Burgess, J. Kessels, M. Roncken, F. Schalij et A. Peeters, "Asynchronous Circuits for Low Power : A DCC Error Corrector", IEEE design & test of computers, Vol. 11, N°. 2, pp. 22-32, 1994.

[BIRT 95] G. Birtwistle and A. Davis Editors, "Asynchronous Digital Circuit Design", Springer, 1995.

[BLUN 00] I. Blunno, L. Lavagno, "Automated synthesis of micro-pipelines from behavioral Verilog HDL", In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, Eilat, Israel, April 1-6, 2000, pp 84-92.

[BREN 70] R.P. Brent. "On the addition of binary numbers". IEEE Transaction on Computers, pages 758-759, Août 1970.

[BRUN 89] E. Brundvand, R. Sproull, "Translating Concurrent Programs into Delay-Insensitive Circuits", in Proc. ICCAD, pp.262-265, 1989.

[BRZO 95] J. A. Brzozowski, C.J.H. Seger, "Asynchronous Circuits", Springer Verlag, Monograph in computer science, 1995, ISBN : 0-387-94420-6.

[BURK 46] A.W. Burks, H.H. Goldstine et J. Von Newmann. "Preliminary discussion of the logical design of an electronic instrument". Rapport technique, The Institute of Advanced Study, Princeton, N.J., 1946. (reprinted in Bell and Newell, Computer structures : readings and examples, Computer Science series, Mc Graw-Hill, 1971).

[BURN 91] S. M. Burns, « Performance Analysis and Optimization of Asynchronous Circuits », PhD Thesis, Caltech-CS-TR-91-01, California Institute of Technology, Pasadena, 1991.

[CHAP 84] D.M. Chapiro, « Globally-Asynchronous Locally-Synchronous Systems », PhD Thesis, Stanford University, Oct. 1984.

[CHRI 98] K. T. Christensen, P. Jensen, P. Korger, J. Sparso, "The design of an Asynchronous Tiny RISC TR4101 Microprocessor Core", ASYNC'98, pp. 108-119, San Diego, USA, April, 1998.

[CHU 85] T. A. Chu, C. K. C. Leung et T. S. Wanuga, "A Design Methodology for Concurrent VLSI Systems", ICCD 85, pp. 407-410, 1985.

[CHU 87a] T. A. Chu, "Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications", MIT Tech. Report MIT/LCS/TR-393, June, 1987.

[CHU 87b] K.M. Chu et D.L. Pulfrey, "A Comparison of CMOS Techniques : Differential Cascode Voltage Switch Logic Versus Conventional Logic", IEEE journal of solid state circuits, Vol. sc-22, N°. 4,pp 528 - 532, Aug. 987.

[CHU 93] T.A. Chu, "CLASS: a CAD system for automatic synthesis and verification of asynchronous finite state machines", in Integration, The VLSI Journal, Vol 15, N° 3, October 1993, pp. 263-289.

[CLAR 67] W.A. Clark, "Macromodular Computer Systems", In Proc. of the Spring Joint Computer Conference, AFIPS, April, 1967.

[DALL 98] E. Dallard, C. Raymond, “Etude et implémentation d’un microcontrôleur asynchrone pour carte sans contact”, rapport de stage de fin d’étude effectué au laboratoire de France Telecom R&D, Meylan, 1998.

[DAVI 92] L. David, R. Ginosar et M. Yoeli, "An Efficient Implementation of Boolean Functions as Self-Timed Circuits", IEEE transactions on computers, Vol. 41, N°. 1, pp 2 - 11, Jan. 1992.

[DAVI 95] I. David, R. Ginosar, and M. Yoeli, "Self-timed is self-checking", Journal of Electronic Testing : Theory and Applications, Vol. 6(2), pp. 219-228, April 1995.

[DIJK 75] E. Dijsktra. "Guarded Commands, Nondeterminancy and formal derivations of programs", Communications of the ACM, vol. 18, August 1975.

[EBER 91] J. Ebergen, "A formal approach to designing delay-insensitive circuits", Distributed Computing, Vol. 5, N°. 3, pp. 107-119, July, 1991.

[EDWA 99] D. Edwards, A. Bardsley, "Synthesizing Asynchronous Systems Using Balsa : A Tutorial and Case Study", 3rd ACiD Workshop, Newcastle Upon Tyne, January, 1999.

[ELHA 95] B. El Hassan, "Architecture VLSI Asynchrone utilisant la logique différentielle à précharge : Application aux opérateurs arithmétiques", Thèse de l'Institut National Polytechnique de Grenoble (INPG), spécialité Microélectronique, soutenue le 26 Septembre 1995 à Grenoble.

[ENDE 98] P. Endecott, S. Furber, "Behavioral Modeling of Asynchronous Systems for Power and Performance Analysis", PATMOS'98 international workshop, Denmark, pp. 137-146, October, 1998.

[FURB 94] S. B. Furber, P. Day, J.D. Garside, N.C. Paver, S. Temple et J.V. Woods, "The design and evaluation of an asynchronous microprocessor", International Conf. Computer Design (ICCD), IEEE Computer Society Press, Oct. 1994.

[FURB 96a] S.B. Burber , J. Lui, "Dynamic Logic Four Phase Micropipelines", In International Symposium on Advanced Research in Asynchronous Circuits and Systems", ASYNC'96, Aizu-Wakamatsu, Japan, 18 - 21 April, pp. 11-16, 1996.

[FURB 96b] B. Furber , Paul Day, "Four Phase Micropipeline Latch Control Circuits", IEEE transactions on VLSI systems, Vol. 4, N°2, june, 1996.

[FURB 97] S.B. Furber, J. Garside, S. Temple, J. Liu, "AMULET2e : An Asynchronous Embedded Controller", in Proceedings of the third international symposium on Advanced Research in Asynchronous Circuits and Systems, April 7-10, 1997, Eindhoven, The Netherlands.

[FURB 99] S. Furber, J. Garside, P. Riocreux, S. Temple, P. Day, J. Liu, N. Paver, “AMULET2e : an asynchronous embedded controller”, Proceedings of the IEEE, vol. 87, n° 2, February 1999, p.243-256.

[GAGE 98] Van Gageldonk H., Van Berkel K., Peeters A., Baumann D., Gloor D., Stegmann G., « An asynchronous low-power 80C51 microcontroller », Proc. of the Int. Symp. on Advanced Research in Async. Circuits and Systems, 1998, IEEE, p. 96-107.

[GARS 00] Garside J., et al., « AMULET3i – an Asynchronous System-On-Chip », Proceedings of the Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems, 2000, IEEE Computer Society, p. 162-175.

[GARS 99] Garside J., Furber S., Chung S.-H., « AMULET3 revealed », Proceedings of the Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems, 1999, IEEE Computer Society, p. 51-59.

[GOPA 92] G. Gopalakrishnan et V. Akella, "VLSI asynchronous systems : specification and synthesis", Microprocessors and Microsystems, Vol. 16, N°. 10, pp 517 - 527, 1992.

[HAUC 95] S. Hauck, "Asynchronous Design Methodologies : An Overview", Proceeding of the IEEE, Vol. 83, N° 1, pp. 69-93, January, 1995.

[HEMA 99] A. Hemani et al, « Lowering Power Consumption in Clock by Using Globally-Asynchronous Locally-Synchronous Design Style », In proc. ACM/IEEE Design Automation Conference, 1999.

[HENN 96] J.L. Hennessy, D.A. Patterson. "Computer Architecture, a Quantitative Approach". Morgan Kaufmann, 1996. Second edition.

[HOAR 78] C.A.R. Hoare, "Communicating Sequential Processes", Communications of the ACM 21, vol. 8, pp. 666-677, Aug 1978.

[HOLL 82] L.A. Hollar, "Direct Implementation of Asynchronous Control Units", IEEE Transaction on Computers, Vol. C-31, pp 1133-1141, Dec. 1982.

[HULG 95] H. Hulgaard, S.M. Burns, and G. Borriello, "Testing asynchronous circuits : A survey", Integration the VLSI Journal, Vol. 19, pp.111-131, 1995.

[JACO 00] H. Jacobson, E. Brunvand, G. Gopalakrishnan, P. Kudva, « High-Level Asynchronous System Design using the ACK Framework », In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, Eilat, Israel, April 1-6, 2000, pp 93-103.

[KESS 00] J. Kessels, G. den Besten, T. Kramer, V. Timm, "Applying Asynchronous Circuits in Contactless Smart Cards", 4th AciD Workshop, Grenoble, France, 31st – 1st February, 2000.

[KESS 00] J. Kessels, T. Kramer, G. den Besten, V. Timm, « Appying Asynchronous Circuits in Contactless Smart Cards », In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, Eilat, Israel, April 1-6, 2000, pp 36-44.

[KISH 92] M. A. Kishinevsky, A. K. Kondratyev, A. R. Taubin et V. I. Varshavsky, "On Self Timed Behavior Verification", TAU 92, Mar.1992.

[KISH 94] M. A. Kishinevsky, A. K. Kondratyev, A. R. Taubin, V. I. Varshavsky, "Concurrent Hardware, The Theory and Practice of Self-Timed Design", Wiley Series in Parallel Computing, Chichester, 1994.

[KLEE 87] L. Kleeman, A. Cantoni, "Metastable behavior in digital systems", IEEE Design & Test of Computers, December 1987, pp. 4-19.

[KOL 96] R. Kol, R. Ginosar, G. Samuel, "Statechart Methodology for the Design, Validation, and Synthesis of Large Scale asynchronous systems", in "International Symposium on Advanced Research in Asynchronous Circuits and Systems", Aizu, Japan, pp.164-174, March 18-21, 1996.

[KOND 98] A. Kondratyev, M. Kishinevsky, A. Yakovlev, "Hazard-Free Implementation of Speed-Independent Circuits", IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, Vol. 17, N°. 9, September, 1998.

[KORE 93] I. Koren. "Computer arithmetic algorithms". Prentice Hall, Englewood Cliffs, NJ, 1993.

[KUDV 96] P. Kudva, G. Gopalakrishnan, H. Jacobson, S.M. Nowick, "Synthesis of hazard-free customized CMOS complex-gate networks under multiple-input changes", in Proc. Of the 33rd ACM/IEEE Design Automation Conference, pp.77-82, June, 1996.

[LAVA 93] L. Lavagno et A. Sangiovanni-Vincentelli, "Algorithms for Synthesis and Testing of Asynchronous Circuits", Kluwer Academic Publishers, 1993.

[LEHM 61] M. Lehman et N. Bural. "Skip techniques for high-speed carry propagation in binary arithmetic units". IRE Transactions on Electronic Computers, page 691, décembre 1961.

[LINE 95] A. M. Lines, « Pipelined Asynchronous Circuits », Master Thesis, Caltech Internal Report CS-TR-95-21, California Institute of Technology, Pasadena, June 1995, Revised June 1998.

[LIS 89] J.S. Lis, D.D. Gajski, « Synthesis from VHDL », Proceedings ICCD, IEEE, New York, October 3-5, 1988, pp. 378-381.

[MALL 99] W. Mallon, J.T. Udding, T. Verhoeff, "Analysis and Application of the XDI model", In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, Barcelona, April 18-22, pp. 231-242, 1999.

[MARS 94] A. Marshall, B. Coates, P. Siegel, "Designing an Asynchronous Communications Chip", in IEEE Design and Test of Computers, Volume 11, Number 2, Summer 1994, pp. 8-21.

[MART 86] A.J. Martin, "Compiling communication processes into delay insensitive VLSI circuits", Distributed computing, vol. 1, pp. 226 - 234, 1986.

[MART 89] A. J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, and Pieter J. Hazewindus, "The Design of an Asynchronous Microprocessor", in Charles L.

Seitz editor, Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference in VLSI, pp. 351-373, MIT Press, 1989.

[MART 90a] A.J. Martin, "Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits", in "Developments in Concurrency and Communication", edited by C.A.R. Hoare, Addison Wesley, pp. 1-64, 1990.

[MART 90b] A.J. Martin, "The limitations to delay-insensitivity in asynchronous circuits", in W.J. Dally, editor, Proceedings of the Sixth MIT Conference on Advanced Research in VLSI, 1990, MIT Press, p. 263-278.

[MART 91] Alain J. Martin and Pieter J. Hazewindus, "Testing delay-insensitive circuits", In Carlo H. Séquin, editor, Advanced Research in VLSI, MIT Press, pp. 118-132, 1991.

[MART 93] A.J. Martin, "Synthesis of Asynchronous VLSI Circuits", Internal Report, Caltech-CS-TR-93-28, California Institute of Technology, Pasadena, 1993.

[MART 97] Martin A., Lines A., Manohar R., Nyström M., Penzes P., Southworth R., Cummings U., Lee T., « The design of an asynchronous MIPS R3000 microprocessor », Proceedings of the 17th Conference on Advanced Research in VLSI, p. 164-181, 1997.

[MATS 97] G. Matsubara, N. Ide, "A low power zero-overhead self-timed division and square root unit combining a single-rail circuit with a dual rail dynamic circuit", in Proceedings of the third international symposium on Advanced Research in Asynchronous Circuits and Systems, April 7-10, 1997, Eindhoven, The Netherlands.

[MAY 90] D. May, "Compiling OCCAM into Silicon", in "Developments in Concurrency and Communication", edited by C.A.R. Hoare, Addison Wesley, pp. 87-106, 1990.

[MCAU 92] A.J. McAuley, "Four State Asynchronous Architectures", IEEE Transactions on Computers, Vol. 41, N°. 2, pp 129 -142, February, 1992.

[MEYE 89] E. Meyer, « VHDL opens the road to top-down design », Computer Design, February 1, 1989, pp. 57-62.

[MOLN 85] C. E. Molnar, T. P. Fang, F. U. Rosenberg, "Synthesis of delay-insensitive modules"Chapel Hill conference on VLSI, computer science press , pp 67 -85, 1985.

[MULL 65] R.E. Muller, "Sequential circuits", Chapter 10, Switching theory, Vol 2, N.Y. Wiley, 1965

[MULL 89] J.M. Muller. "Arithmétique des ordinateurs". Masson, 1989.

[MULL 97] J.M. Muller, A. Tisserand, et J.M. Vincent. "Asynchronous sub-logarithmic adders". IEEE Pacific Rim Conference on Communication, Computers and Signal Processing (PACRIM97), vol.2, pg. 515-518, Août 97. Victoria, Canada.

[MULL 97] J.M. Muller. "Elementary Functions, Algorithms and Implementation". Birkhauser, Boston, 1997.

[MUTT 00] J. Muttersbach, T. Villiger, W. Fichtner, « Practical Design of Globally-Asynchronous Locally-Synchronous Systems », In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, Eilat, Israel, April 1-6, 2000, pp 52-59.

[NANY 94] Nanya T., Ueno Y., Kagotani H., Kuwako M., Takamura A., « TITAC: design of a quasi-delay-insensitive microprocessor », IEEE Design and Test of Computers, vol. 11, n° 2, 1994, p. 50-63.

[NIEL 94] C.D.Nielsen, "Evaluation of function blocks for Asynchronous design", EURODAC'94, Grenoble, France, pp 454 - 459, Sept.1994.

[NIEL 94] L.S. Nielsen, C. Niessen, J. Sparso, J. Van Berkel, "Low Power Operation Using Self-Timed Circuits and Adaptive Scaling of the Supply Voltage", IEEE Transaction on Very Large Scale Integration (VLSI) Systems, Vol. 2, N° 4, December, 1994.

[NOWI 91] S.M. Nowick et D.L. Dill, "Automatic Synthesis of Locally-Clocked Asynchronous State Machines", ICCAD, pp. 318-321, 1991.

[NOWI 95] S.M. Nowick, D.L. Dill, "Exact two level minimization of hazard-free logic with multiple input changes", IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, vol. 14(8), pp. 986-997, August 1995.

[OMON 94] A.R. Omondi. "Computer Arithmetic Systems, Algorithms, Architecture, and Implementations". Prentice Hall International Series in Computer Science, Englewood Cliffs, NJ, 1994.

[PAST 98] E. Pastor, J. Cortadella, A. Kondratyev, O. Roig, "Structural Methods for Synthesis of Speed-Independent Circuits", IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, Vol. 17, N°. 11, November, 1998.

[PAVE 98] N.Paver et al, "A Low-Power, Low-Noise, Configurable Self-Timed DSP", ASYNC'98, pp.32-42, San Diego, March, 1998.

[PIGU 99] C. Piguet, J. Zahnd, "Electrical Design of Dynamic and Static Speed-Indemendent CMOS Circuits from STGs", 3rd ACiD Workshop, Newcastle Upon Tyne, January 1999.

[PROC 95] Proceedings of the second working conference on asynchronous Design Methodologies, May 30-31, 1995, London, England.

[PUTZ 98] W. Putzke-Röming, M. Radetski, W. Nebel, "A Flexible Message Passing Mechanism for Objective VHDL", DATE'98, Paris, France, February 23-26, pp.242-249, 1998.

[RENA 94a] M. Renaudin, B. El Hassan, "The Design of Fast Asynchronous Adder Structures and Their Implementation Using D.C.V.S. Logic", in Proceedings ISCAS, London, May, 1994.

[RENA 94b] M. Renaudin and B. El Hassan, "A minimum power, 100 MHz , 12x18+30-b Multiplier-Accumulator operating in asynchronous and synchronous mode", ESSCIRC 94, Ulm , Germany, Sept. 1994.

[RENA 96] M. Renaudin, B. El Hassan, A. Guyot, "A New Asynchronous Pipeline Scheme : Application to the Design of a Self-Timed Ring Divider", in IEEE Journal of Solid-State Circuits, Vol. 31, N° 7, pp. 1001-1013, July, 1996.

[RENA 97] M. Renaudin, F. Robin, P. Vivet, "AAAA : asynchronisme et adéquation algorithme architecture", Traitement du Signal, numéro spécial, vol. 14, n° 6, Adéquation Algorithme Architecture, pp. 589-604, 1997.

[RENA 98a] M. Renaudin, P. Vivet, "CHP2VHDL, a CHP to VHDL translator, towards asynchronous design simulation", Second ACiD-WG Workshop, Torino, January 1998.

[RENA 98b] M. Renaudin, P. Vivet, F. Robin, "ASPRO-216 : A Standard-Cell Q.D.I. 16-Bit RISC Asynchronous Microprocessor", in "International Symposium on Advanced Research in Asynchronous Circuits and Systems", ASYNC'98, San Diego, USA, 30 March- 2 April, 1998.

[RENA 99a] M. Renaudin, P. Vivet, "A Design Experiment : The ASPRO Program Memory", Third ACiD-WG Workshop, New-Castle Upon Tyne, January 1999.

[RENA 99b] M. Renaudin, P. Vivet , F. Robin, "A Design Frame Work for Asynchronous/ Synchronous Circuit Based on CHP to HDL Transaction", In International Symposium on Advanced Research in Asynchronous Circuits and Systems" ASYNC'99, Barcelona, Spain, April 19-21, pp 135-144, 1999.

[RENA 99c] M. Renaudin, P. Vivet , F. Robin, “ASPRO : an Asynchronous 16-Bit RISC Microprocessor with DSP Capabilities”, Proceedings of the 25th European Solid-State Circuits Conference, ESSCIRC’99, Duisburg, Germany, 21-23 September 1999, pp 428-431.

[RENA 00a] M. Renaudin, P. Vivet, Ph. Geoffroy, "ASPRO : a toy demo", 4th AciD Workshop, Grenoble, France, 31st – 1st February, 2000.

[RENA 00b] M. Renaudin, « Etat de l’art sur la conception des circuits asynchrones, perspectives pour l’intégration des systèmes complexes », rapport de recherche interne TIMA / STMicroelectronics, Fev. 2000.

[RENA 00c] M. Renaudin, « Asynchronous circuits and systems : a promising design alternative », Edition Elsevier, Microelectronic Engineering, Vol. 54, pp. 133-149, Dec. 2000.

[RICH 96] W.F. Richardson, E Brunvand, « Fred : an Architecture for a Self-Timed Decoupled Computer », Proceedings of Second Internationnal Symposium on Advanced Research in Asynchrnous Circuits and Systems, Aizu-Wakamatsu, Japon, March 1996, pp. 60-68.

[ROBI 01] F. Robin, "Introduction aux Microprocesseurs Asynchrones", Technique et Science Informatique, Hermès Science Publications, Paris, Volume 20, no. 1/2001, pp. 9-30.

[ROBI 96] F. Robin, M. Renaudin, G. Privat, N. Van Den Bossche, " A Functionally Asynchronous Array-Processor for Morphological Filtering of Greyscale Images ", IEE Computers and Digital Techniques, special section on Asynchronous Architecture, Vol. 143, N°. 5, September, 1996.

[ROBI 97a] F. Robin, "Etude d'architectures VLSI numériques parallèles et asynchrones pour la mise en œuvre de nouveaux algorithmes d'analyse et rendu d'images", Thèse de doctorat de l'ENST Paris, spécialité Electronique et Communications, soutenue à Grenoble le 27 octobre 1997.

[ROBI 97b] F. Robin, M. Renaudin, G. Privat, N. Van Den Bossche, "Un réseau cellulaire VLSI fonctionnellement asynchrone pour le filtrage morphologique d'images", Traitement du Signal, numéro spécial, vol. 14, n° 6, Adéquation Algorithme Architecture, pp. 655-664, 1997.

[ROO 86] M. Roorda. "Method to reduce the sign bit extension in a multiplier that uses the modified Booth algorithm", Electronics letters, Vol. 22, No. 20, pp. 1061-1062, 25th September 1986.

[ROSE 88] F. U. Rosenberg, C. E. Molnar, T. J. Chaney et T. P. Fang, "Q-Modules: Internaly clocked delay-insensitive modules", IEEE transactions on computers, Vol. 37, pp. 1005-1018, Sept. 1988.

[SAKU 88] T. Sakurai, "optimization of CMOS arbiter and synchronizer circuit with submicrometer MOSFET'S", IEEE journal of solid-state circuits, Vol. 23, N° 24, pp. 901 - 906, Aug. 1991.

[SIA 97] Semiconductor Industry Association, "The National Technology Roadmap for Semiconductors", 4300 Stevens Creek Blvd. Suite 271, San Jose, CA 95129, 1997

[SILC 98] J. Silc, B. Robic, and T. Ungerer, « Asynchrony in Parallel Computing : from Dataflow to Multithreading », Parallel and Distributed Computing Practices, Vol.1, No.1, March 1998.

[SPAR 93a] J. Sparo et J. Staunstup, "Delay insensitive multi-ring structures", Integration, the VLSI journal, pp. 313 - 340, 1993.

[SPAR 93b] J. Sparo, C.D. Nielsen,L.Nielsen, et J. Staunstup, "Design of self timed multipliers : a comparaison", IFIP Working conférence on asynchronous design methodologies, England, Apr.1993.

[SPROU 94] R.F Sproull, I. Sutherland, C. Molnar, "The Counterfow Pipeline Processor Architecture", in IEEE Design and Test of Computers, Volume 11, Number 3, Fall 1994, pp. 48-59.

[STEV 99] K. Stevens, R. Ginosar, S. Rotem, "Relative Timings", In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, Barcelona, April 18-22, pp. 208-218, 1999.

[SUTH 89] Ivan E. Sutherland, “Micropipelines”, Communication of the ACM Volume 32, N°6, June 1989.

[SWAM 95] S. Swamy, A. Molin, B. Convot, "OO-VHDL Object Oriented Extensions of VHDL", IEEE Computer, October, pp.18-26, 1995.

[TAKA 97] Takamura A., Kuwako M., Imai M., Fujii T., Ozawa M., Fukasaku I., Ueno Y., Nanya T., « TITAC-2: an asynchronous 32-bit microprocessor based on scalable-delay-insensitive model », Proceedings of the ICCD, 1997, IEEE, p. 288-294.

[TAN 98] S.Y. Tan, S. Furber, W.F. Yen, "The design of an asynchronous VHDL Synthetizer", DATE'98, Paris, France, February 23-26, pp.44-51, 1998.

[TERA 99] H. Terada, S. Miyata, M. Iwata, "DDMP's : Self-Timed Super-Pipelined Data-Driven Multimedia Processors", Proceedings of the IEEE, Vol. 87, N°. 2, pp.282-296, February, 1999.

[TIER 94] J. A. Tierno, A.J. Martin, D. Borkovic et T. Lee, "A 100 MIPS GaAs asynchronous microprocessor", IEEE Design & Test of Computers, Vol. 11, n°. 2, pp. 43-49, 1994.

[TISS 97] A. Tisserand. "Adéquation Arithmétique Architecture : Problèmes et études de cas". Thèse Ecole Normale Supérieure de Lyon, septembre 1997.

[VARS 86] V.I. Varshavsky, "Self-timed Control of Concurrent Processes", Kluwer Academic Publishers, 1986, ISBN : 0-7923-0525-6.

[VERH 88] Verhoeff T., « Delay-insensitive codes – an overview », Distributed Computing, vol. 3, p. 1-8, Springer-Verlag, 1988.

[WALL 64] C.S. Wallace, "A suggestion for a fast multiplier", IEEE Transaction on Electronic Computers, pp. 14-17, February 1964.

[WILL 91] T.E. Williams, "Self timed rings and their application to division", Ph.D. dissertation, Stanford University, California, May1991.

[WILL 94] T.E. Williams, "Performance of iterative computation in self timed rings", in J. VLSI Signal Processing, no. 7, pp 17-31, Février 1994.

[WINO 65] S. Winograd. "On the time required to perform addition". Journal of the association for computing Machinery, 12(2):277-285, Avril 1965.

[WOOD 97] Woods J., Day P., Furber S., Garside J., Paver N., Temple S., « AMULET1: an asynchronous ARM microprocessor », IEEE Transactions on Computers, vol. 46, n° 4, April 1997, p. 385-398.

[YUN 92] K. Yun et D. Dill, "Automatic Synthesis of 3D Asynchronous State Machines", ICCAD, pp. 576-580, 1992.

[YUN 96] K. Yun, P. Beerel, J. Arceo, "High Performance Asynchronous Pipeline", In "International Symposium on Advanced Research in Asynchronous Circuits and Systems", ASYNC'96 , Aizu-Wakamatsu, Japan, 18 - 21 April, pp 17-28, 1996.

[ZURA 86] D. Zuras, W. Mc Allister, "Balanced delay trees and combinatorial division in VLSI", IEEE J. Solid State Circuits, vol. SC 21, No. 5, pp. 814-819, October 1986

RESUME

Au contraire des circuits synchrones, les circuits asynchrones se caractérisent par leur absence de signal d'horloge. Ces circuits sont séquencés par un mécanisme de signalisation et de communication locale. Les circuits asynchrones offrent ainsi des perspectives intéressantes pour l'intégration de systèmes dans les technologies submicroniques telles que : robustesse, faible bruit, faible consommation, bonne modularité. Cependant, le manque de méthodes et outils de conception est un frein à leur développement. Les travaux présentés dans cette thèse portent sur la définition d'une méthodologie de conception de circuits intégrés asynchrones quasi-insensibles aux délais. Les circuits quasi-insensibles aux délais font partie de la classe des circuits asynchrones les plus robustes, propriété avantageuse pour les technologies à venir. La méthode de conception proposée permet d'une part la modélisation dans un langage de haut-niveau et la simulation dans un environnement standard et d'autre part la génération de circuits uniquement constitués de cellules standard. Cette méthodologie a été appliquée à l'étude et à la réalisation d'un processeur RISC 16-bit. Son architecture originale permet d'effectuer l'envoi des instructions dans l'ordre et de les terminer dans le désordre. L'étude d'architecture montre qu'il est possible de tirer parti du mode de fonctionnement asynchrone afin de relâcher les synchronisations et d'implémenter des unités à temps de calcul et consommation minimum, à la fois en fonction des données et des instructions. Le prototype fabriqué dans la technologie CMOS 0.25um de STMicroelectronics est l'un des processeurs asynchrones le plus rapide réalisé à ce jour.

TITLE

A quasi-delay insensitive integrated circuit design methodology :
application to the study and design of a 16-bit asynchronous RISC microprocessor.

ABSTRACT

Unlike synchronous circuits, asynchronous circuits are not sequenced using a global clock signal. These circuits are locally synchronized using an adapted signaling and a local communication mechanism. Thus, asynchronous circuits offer interesting perspectives for system integration in deep submicron technologies such as : robustness, low noise, low power, good modularity. Nevertheless, the lack of associated design methodologies and tools prevent them from being widely spread. This thesis first specifies a design methodology for quasi-delay insensitive asynchronous circuits. Quasi delay insensitive circuits are the most robust asynchronous circuits, which is a major advantage for upcoming technologies. The proposed design methodology allows to model circuits using a high level language, to simulate them in a standard environment, and to generate circuits using only standard cells. This methodology was then applied to the study and design of a 16-bit RISC processor. Its original architecture enables in-order issue of instructions and out-of-order completion of their execution. It is shown that the asynchronous execution mode can weaken synchronizations, leading to the implementation of units exhibiting minimal execution-time and power-consumption. The prototype fabricated with the 0.25um CMOS technology from STMicroelectronics, is one of the fastest asynchronous processor designed so far.

SPECIALITE : Microélectronique

MOTS-CLES :

Circuits intégrés, CAO, Méthodologie de conception, Langage CHP,
Circuits asynchrones, Circuits quasi-insensibles aux délais, Microprocesseur, Architecture RISC.

INTITULE ET ADRESSE DE l' U.F.R OU DU LABORATOIRE :

France Telecom R&D, 28 chemin du Vieux Chêne, 38240 Meylan Cedex, France

ISBN 2-913329-68-3 broché
ISBN 2-913329-69-0 électronique