



HAL
open science

Une Méthodologie de Conception de Circuits Asynchrones à Faible Consommation d'Énergie: Application au Microprocesseur MIPS

K. Slimani

► **To cite this version:**

K. Slimani. Une Méthodologie de Conception de Circuits Asynchrones à Faible Consommation d'Énergie: Application au Microprocesseur MIPS. Micro et nanotechnologies/Microélectronique. Institut National Polytechnique de Grenoble - INPG, 2004. Français. NNT: . tel-00008328

HAL Id: tel-00008328

<https://theses.hal.science/tel-00008328>

Submitted on 2 Feb 2005

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

| / / / / / / / / / / |

T H E S E

Pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Micro et Nano Electronique

Préparée au laboratoire **TIMA** dans le cadre de
**L'Ecole Doctorale d'« Electronique, Electrotechnique, Automatique,
Télécommunications, Signal »**

Présentée et Soutenue publiquement

Par

Kamel SLIMANI

Le 16 décembre 2004

Titre :

**UNE METHODOLOGIE DE CONCEPTION DE CIRCUITS
ASYNCHRONES A FAIBLE CONSOMMATION D'ENERGIE:
APPLICATION AU MICROPROCESSEUR MIPS**

Directeur de Thèse : Marc Renaudin
Codirecteur : Gilles Sicard

JURY

Mme. Nathalie Julien	, Présidente
M. Christian Piguet	, Rapporteur
M. Jean-Didier Legat	, Rapporteur
M. Marc Renaudin	, Directeur
M. Gilles Sicard	, Codirecteur

*À la Mémoire d'Amé Youcef
Et de Moez Ouni*

Remerciements

La thèse a été effectuée au laboratoire TIMA (Techniques de Informatique et de la Microélectronique pour l'Architecture des Ordinateurs) sur le site Viallet de l'Institut National Polytechnique de Grenoble. Je remercie le directeur du laboratoire Bernard Courtois pour m'y avoir accueilli.

Je remercie mon directeur de thèse Marc Renaudin professeur à l'ENSERG pour m'avoir accueilli au sein de son groupe de recherche Concurrent Integrated Circuit (CIS). Je le remercie pour m'avoir permis de côtoyer le monde passionnant de la conception asynchrone. Je voudrais aussi sincèrement le remercier pour son appui technique, pour sa gentillesse et son humour.

Je tiens à remercier les membres du jury:

Nathalie Julien professeur à l'Université de Bretagne Sud pour m'avoir fait l'honneur de présider le jury de ma thèse.

Christian Piguet professeur au Swiss Center for Electronics and Microtechnology (CSEM) et Jean-Didier Legat Professeur à l'Université Catholique de Louvain pour avoir accepté d'être les rapporteurs de ma thèse. Je leur suis très reconnaissant de la qualité et de l'analyse intéressante de leurs rapports.

J'exprime ma gratitude à mon codirecteur Gilles Sicard maître de conférence à l'Université Joseph Fourier pour sa disponibilité, pour son retour très intéressant et pour son aide à la rédaction du manuscrit. Je voudrais aussi saluer sa gentillesse, sa simplicité et sa passion pour la Formule 1. Je n'oublierais jamais son agréable compagnie à Santorin.

Je tiens à remercier chaleureusement les membres du groupe CIS pour leur sympathie quotidienne, pour leur convivialité et pour les nombreuses activités qu'on a pu faire ensemble (ordre alphabétique) :

Alain (Mr opérateur arithmétique), Amine (mon fidèle compagnon de bureau), Anh Vu, Antoine, Arnaud, Aurélien, Bertrand, Cédric, David, Dhanistha, Estelle, Fabien (le roi des blagues), Fraidy (soirée au D'ombolo ça te dit), Gautier, Isabelle, JB (sa cheminée c'est la caverne d'Ali Baba), Jérôme, Joao & Marianna (alors ce bowling !!!), Julien, Laurent, Livier, Manu (toujours partant pour un verre, il est pas encore arrivé le train de Paris!!), Momo (alors marocain, tac tac tac), Philippe le montpelliérain, Salim (le kabyle), Sophie, Thibaut, Vivien, Yann (le photographe de choc), Yannick ('a' 'o' 'i' tout doux).

Je voudrais aussi sincèrement remercier toutes les personnes que j'ai connues à TIMA et qui ont fait que la vie au laboratoire était si sympathique. Je ne peux malheureusement pas toutes les nommer, je citerais quelques noms seulement :

Aimen, Amel, Amer (merci le gadgeot), Aziz, Damien (le néo canadien), Faiza, Férid, Greg, Iuliana (Ibiza c'est pour quand?) et Kati, Karim, Khirdine, Lobna, Marius, Menouer, Nacer, Nadir (fais le maximum!), Wassim, Yanick.

Je voudrais exprimer par la même occasion ma gratitude au personnel fort sympathique de TIMA, CMP et du CIME : les professeurs, les maîtres de conférence, les administrateurs réseaux et les très souriantes secrétaires.

J'adresse en particulier mes remerciements aux personnes qui m'ont très gentiment et très efficacement apporté leur aide lorsque je les sollicitais. Qu'Alexandre Chagoya et que Kholdoun reçoivent ma sincère gratitude.

Je remercie mes amis d'enfance et du football que je ne peux pas me risquer à nommer de peur d'en oublier. Je remercie également mes amis des virées nocturnes du week-end (Fabrice, Caro, Clarisse).

Enfin je terminerai par remercier de tout mon cœur mes parents et toute ma famille pour l'amour qu'ils me donnent chaque jour et pour la confiance qu'ils m'accordent.

Résumé

L'évolution effrénée de la complexité des appareils portables s'accompagne d'une augmentation de la consommation d'énergie qui rend l'autonomie de ces appareils très limitée. Pour accroître l'autonomie des systèmes embarqués tels que les téléphones ou les ordinateurs portables, de nombreuses recherches sur la conception de circuits intégrés ont été réalisées en vue de réduire la consommation d'énergie.

Ces travaux de thèse ont pour but de proposer au concepteur des moyens de concevoir des circuits intégrés numériques à faible consommation d'énergie. Trois étapes importantes vers la réduction de la consommation d'énergie ont été proposées. L'utilisation de la logique asynchrone représente le premier pas vers la réduction de la consommation d'énergie. En effet, de nombreux travaux réalisés ces dernières années ont montré que les circuits asynchrones présentent la propriété intrinsèque de consommer moins d'énergie que les circuits implémentés en logique synchrone. Le second pas important est d'offrir au concepteur des outils lui permettant d'obtenir des informations sur l'activité et la consommation d'énergie du circuit lors de la conception de celui-ci. Nous avons spécifié un estimateur d'activité et un estimateur de la consommation d'énergie qui permettent au concepteur de collecter des informations pertinentes sur la répartition de l'activité et de la consommation d'énergie d'un circuit lors d'une simulation donnée. L'avantage de ces estimateurs est que les informations sont obtenues très tôt dans le flot de conception, au niveau de la spécification CHP d'un circuit. Enfin, des techniques d'optimisation sont proposées pour réduire la consommation d'énergie des circuits. Certaines techniques sont utiles à l'outil de synthèse (pour la synthèse automatique de circuits à faible consommation d'énergie) et d'autres sont utiles au concepteur pour le guider dans ses choix d'implémentation.

La méthodologie d'estimation et les techniques d'optimisation de la consommation d'énergie ont été appliquées à la réalisation d'un mini processeur, les résultats ont montré une réduction de la consommation d'énergie de 24%.

Mots-clés : Circuits asynchrones, faible consommation d'énergie, estimation d'activité, estimation de la consommation d'énergie, techniques d'optimisation, synthèse faible consommation d'énergie, implémentation faible consommation.

Abstract

The steady evolution of mobile device complexity comes with an increase of the energy consumption which dramatically limits the autonomy of these devices. To increase the autonomy of embedded systems such as portable phones or computers, several researches on integrated circuit design have been performed in order to reduce the energy consumption.

This Phd work aims at proposing to the designer the means to design low energy digital integrated circuits. Three important steps have been proposed in this work to reduce the energy consumption of circuits. The first step is to use asynchronous logic to design circuit. Indeed, several works achieved these last years have shown that asynchronous logic can significantly reduce the energy consumption of circuits. It is an intrinsic property of asynchronous logic. The second step is to offer tools that allow the designer to get information on the activity and the energy consumption of a circuit during the design flow. An activity estimator and an energy estimator have been specified that allow the designer to get relevant information on the distribution of the activity and the energy within a circuit during a specific simulation. The inherent advantage of these estimators is to provide information on the activity and the energy very early in the design flow, at the CHP specification. Finally, optimisation techniques have been defined to reduce the energy consumption of circuits. Some techniques can be integrated in the synthesis tool (for automatic low energy-oriented synthesis) and other techniques can be useful to guide the designer in his implementation choices.

The estimation methodology and the energy consumption optimisation techniques have been applied to the implementation of a mini processor. The results have shown an energy consumption reduction of 24%.

Keywords: Asynchronous circuits, low energy consumption, activity estimation, energy consumption estimation, optimisation techniques, low energy consumption synthesis, low consumption design.

Table des Matières

Remerciements	vii
Résumé	ix
Abstract	x
Table des Matières	xi
Liste des Figures	xv
Liste des Tableaux	xix
Introduction	- 1 -
Chapitre 1 Etat de l'art : Logique asynchrone, Consommation d'Energie et Microprocesseurs Asynchrones	- 5 -
1.1 Logique Asynchrone	- 5 -
1.1.1 Asynchrone versus Synchrone	- 5 -
1.1.2 Les principes de base de l'asynchrone	- 8 -
1.1.2.1 Codage donnée groupée du canal	- 8 -
1.1.2.2 Codage Double Rail du canal	- 10 -
1.1.2.3 La porte de Muller	- 12 -
1.1.3 Les logiques asynchrones	- 13 -
1.1.3.1 Circuits insensibles aux délais (DI)	- 13 -
1.1.3.2 Circuits indépendant de la vitesse et quasi-insensible aux délais	- 13 -
1.1.3.3 Circuits micropipelines	- 14 -
1.1.3.4 Circuits de Huffman	- 15 -
1.1.4 Les Bufferisations	- 15 -
1.1.4.1 Sans bufferisation : le protocole séquentiel	- 16 -
1.1.4.2 La bufferisation WCHB	- 16 -
1.1.4.3 La bufferisation PCHB	- 17 -
1.1.4.4 La bufferisation PCFB	- 18 -
1.2 La consommation d'énergie	- 18 -
1.2.1 Les techniques d'estimation de la consommation d'énergie	- 19 -
1.2.2 Les techniques d'optimisation de la consommation d'énergie	- 20 -
1.2.2.1 Pipeline asynchrone	- 20 -
1.2.2.2 Structure de contrôle	- 22 -
1.2.2.3 Optimisation des communications	- 22 -
1.2.2.4 Paramètre $E\tau^2$	- 23 -
1.2.2.5 Techniques de réduction au niveau système	- 24 -
1.2.3 Autres techniques	- 25 -
1.3 Les Microprocesseurs asynchrones	- 26 -

1.3.1	Caltech Asynchronous Processeur (CAP).....	- 26 -
1.3.2	Philips 80C51 asynchrone.....	- 27 -
1.3.3	AMULET.....	- 29 -
1.3.4	ASPRO.....	- 33 -
1.3.5	MICA.....	- 34 -
1.3.6	TITAC.....	- 35 -
1.3.7	MiniMips.....	- 38 -
1.3.8	Asynchronous TinyRISC (ARISC).....	- 39 -
1.3.9	Lutonium.....	- 41 -
1.3.10	Autres Microprocesseurs asynchrones.....	- 42 -
1.3.11	Récapitulatif des microprocesseurs asynchrones.....	- 45 -
1.4	Conclusion.....	- 46 -
Chapitre 2 Estimation de la Consommation d'Énergie d'un Programme CHP		- 47 -
2.1	Méthodologie d'estimation de la consommation d'énergie.....	- 47 -
2.2	Estimation de la consommation d'énergie structurelle.....	- 49 -
2.2.1	Coût intrinsèque des commandes gardées.....	- 52 -
2.2.1.1	Coût associé à l'évaluation de la garde.....	- 53 -
2.2.1.2	Coût d'activation de la garde.....	- 59 -
2.2.1.3	Coût de la bufferisation.....	- 60 -
2.2.1.4	Coût de l'initialisation.....	- 62 -
2.2.1.5	Coût de la fonction.....	- 62 -
2.2.2	Coût d'intersection des branches.....	- 70 -
2.2.3	Consommation des portes.....	- 72 -
2.2.4	Consommation des architectures de base.....	- 77 -
2.2.4.1	Latch.....	- 77 -
2.2.4.2	Fourche ("Fork").....	- 78 -
2.2.4.3	Jonction ("Join").....	- 80 -
2.2.4.4	Merge.....	- 81 -
2.2.4.5	Multiplexeur.....	- 83 -
2.2.4.6	Démultiplexeur.....	- 85 -
2.3	Estimation de la consommation d'énergie dynamique.....	- 86 -
2.3.1	Labellisation du réseau de pétri.....	- 87 -
2.3.2	Calcul du coût dynamique.....	- 87 -
2.4	Application à une architecture hétérogène.....	- 88 -
2.5	Conclusion.....	- 94 -
Chapitre 3 Spécification des Outils d'Estimation.....		- 95 -
3.1	Présentation du flot TAST.....	- 95 -
3.1.1	Le compilateur CHP.....	- 96 -
3.1.2	Le simulateur de Pétri Net.....	- 97 -
3.1.3	Le convertisseur CHP vers VHDL fonctionnel.....	- 97 -
3.1.4	L'outil de synthèse Quasi Insensible aux Délais.....	- 98 -
3.1.5	L'outil de synthèse Micropipeline.....	- 98 -
3.2	Estimateur d'activité.....	- 99 -

3.2.1	Format du fichier Trace.....	- 99 -
3.2.2	Exemple.....	- 103 -
3.2.3	Caractéristiques de l'estimateur d'activité	- 106 -
3.2.3.1	Couverture de code.....	- 106 -
3.2.3.2	Trace.....	- 106 -
3.2.3.3	Statistiques	- 108 -
3.3	Estimateur de la consommation d'énergie.....	- 110 -
3.3.1	Principe de l'estimateur de la consommation d'énergie.....	- 111 -
3.3.2	Caractéristiques de l'estimateur de la consommation d'énergie	- 111 -
3.4	Conclusion.....	- 113 -
Chapitre 4 Exploration Architecturale et Optimisation de la Consommation		
d'Énergie - 115 -		
4.1	Synthèse orientée faible consommation d'énergie	- 115 -
4.1.1	Protocole de communication.....	- 115 -
4.1.2	Bufferisation entrée/sortie.....	- 117 -
4.1.3	Bufferisation mixte.....	- 125 -
4.1.4	Désynchronisation intra canal.....	- 126 -
4.1.5	Ajout de synchronisation.....	- 127 -
4.1.6	Etude des opérateurs arithmétiques et logiques	- 129 -
4.2	Implémentation CHP orientée faible consommation d'énergie.....	- 130 -
4.2.1	Optimisation micro architecturale.....	- 130 -
4.2.2	Codage des données	- 131 -
4.2.3	Adaptation de la taille des canaux de donnée	- 132 -
4.2.4	Désynchronisation inter canaux	- 133 -
4.2.5	Codage du contrôle.....	- 135 -
4.2.6	Fusion des opérateurs	- 136 -
4.2.7	Réduction de la synchronisation en cas d'exclusion mutuelle.....	- 139 -
4.2.8	Structures de choix en fonction des probabilités.....	- 139 -
4.2.8.1	Structure de choix hiérarchique.....	- 139 -
4.2.8.2	Structure de choix hiérarchique avec commandes découpées	- 145 -
4.3	Conclusion.....	- 148 -
Chapitre 5 Application : Réalisation d'un Mini Processeur Asynchrone à Faible		
Consommation d'Énergie		
- 149 -		
5.1	Architecture du Mini Processeur.....	- 149 -
5.1.1	Le compteur de programme (PC).....	- 150 -
5.1.2	Le décodeur (decode).....	- 151 -
5.1.3	L'interface d'exécution (Exec Interface).....	- 151 -
5.1.4	L'unité d'exécution (Exec Unit).....	- 152 -
5.1.5	L'unité de multiplication et division (MDU).....	- 152 -
5.1.6	Le banc de registres.....	- 153 -
5.1.7	Les mémoires	- 154 -
5.1.7.1	Mémoire instruction (Instr Memory)	- 154 -
5.1.7.2	Mémoire donnée (Data Memory).....	- 154 -
5.2	Jeu d'instructions	- 154 -
5.3	Étapes d'optimisation du mini processeur	- 155 -

5.3.1	Version V0 : Forme primaire du mini processeur.....	- 155 -
5.3.2	Version V1 : Mini processeur implémenté en codage 1 parmi 4.....	- 156 -
5.3.3	Version V2 : Optimisation en fonction des probabilités d'exécution.....	- 156 -
5.3.4	Version V3 : Optimisation du contrôle	- 163 -
5.4	Résultats	- 163 -
5.4.1	Consommation de l'ensemble du cœur du mini processeur	- 164 -
5.4.2	PC	- 168 -
5.4.3	Décodeur	- 169 -
5.4.4	Interface d'exécution	- 169 -
5.4.5	Unité d'exécution.....	- 170 -
5.4.6	La MDU	- 171 -
5.5	Implémentation finale du mini processeur	- 172 -
5.5.1	Implémentation du cœur.....	- 172 -
5.5.2	Implémentation du banc de registres.....	- 173 -
5.5.3	Implémentation des mémoires	- 174 -
5.5.3.1	Mémoire instruction	- 174 -
5.5.3.2	Mémoire donnée.....	- 175 -
5.5.4	Implémentation des interfaces.....	- 175 -
5.5.4.1	Interface banc de registres.....	- 175 -
5.5.4.2	Interface mémoire instruction	- 176 -
5.5.4.3	Interface mémoire donnée	- 177 -
5.5.5	Simulation du mini processeur	- 179 -
5.6	Conclusion.....	- 179 -
Conclusion et Perspectives.....		- 181 -
Publications et Réalisations Personnelles.....		- 185 -
Bibliographie.....		- 187 -
Annexe		- 195 -

Liste des Figures

Figure 1.1	Structure de base d'un circuit synchrone.....	- 5 -
Figure 1.2	Structure de base d'un circuit asynchrone.....	- 6 -
Figure 1.3	Représentation d'un circuit asynchrone	- 8 -
Figure 1.4	Canal codé en donnée groupée	- 9 -
Figure 1.5	Codage donnée groupée associé au protocole 4 phases.....	- 9 -
Figure 1.6	Codage donnée groupée associé au protocole 2 phases.....	- 10 -
Figure 1.7	Canal codé en double rail.....	- 11 -
Figure 1.8	Codage Double Rail associé au protocole 4 phases.....	- 11 -
Figure 1.9	Codage Double Rail associé à un protocole 2 phases.....	- 12 -
Figure 1.10	Porte de Muller	- 13 -
Figure 1.11	Hypothèse circuit QDI (gauche) et SI (droite).....	- 14 -
Figure 1.12	Circuit micropipeline	- 14 -
Figure 1.13	Une table de flot (gauche) et la machine à état correspondante (droite)	- 15 -
Figure 1.14	Schéma général d'une bufferisation	- 16 -
Figure 1.15	Buffer séquentiel.....	- 16 -
Figure 1.16	Buffer WCHB	- 17 -
Figure 1.17	Buffer PCHB.....	- 18 -
Figure 1.18	Buffer PCFB	- 18 -
Figure 1.19	Anneau de buffers	- 21 -
Figure 1.20	Débit et Consommation en fonction du nombre de buffers.....	- 21 -
Figure 1.21	Pipeline utilisant un latch reconfigurable	- 22 -
Figure 1.22	Multiplexeur régulier et décomposé	- 22 -
Figure 1.23	Architecture du CAP.....	- 27 -
Figure 1.24	Architecture du 80C51 asynchrone.....	- 28 -
Figure 1.25	Layout du 80C51 asynchrone	- 29 -
Figure 1.26	Architecture de AMULET1	- 30 -
Figure 1.27	Architecture de AMULET2e	- 31 -
Figure 1.28	Architecture de AMULET3	- 32 -
Figure 1.29	Système de contrôle DRACO	- 33 -
Figure 1.30	Architecture d'ASPRO	- 34 -
Figure 1.31	ASPRO.....	- 34 -
Figure 1.32	Architecture du processeur TITAC.....	- 36 -
Figure 1.33	Architecture du processeur TITAC-2	- 37 -
Figure 1.34	Performances de TITAC-2 en fonction de la tension d'alimentation.....	- 37 -
Figure 1.35	Layout de TITAC-2	- 38 -
Figure 1.36	Architecture du MiniMIPS	- 39 -
Figure 1.37	Architecture du processeur Asynchronous TinyRISC.....	- 40 -
Figure 1.38	Layout du processeur Asynchronous TinyRISC	- 40 -
Figure 1.39	Architecture de Lutonium.....	- 41 -
Figure 1.40	Répartition de la consommation d'énergie dans Lutonium	- 42 -
Figure 1.41	Microprocesseur SPA	- 42 -
Figure 1.42	Layout du processeur MSL16A	- 44 -
Figure 2.1	Estimation de la consommation d'énergie dans le flot de synthèse TAST	- 48 -
Figure 2.2	Réseau de pétri de la forme générale	- 50 -
Figure 2.3	Evaluation de l'égalité (gauche) et de l'inégalité (droite) avec une constante ..	- 54 -
Figure 2.4	Evaluation de l'égalité (gauche) et de l'inégalité (droite) avec une variable.....	- 55 -

Figure 2.5 Exemple de test incluant les opérateurs and et nand	- 56 -
Figure 2.6 Exemple de test incluant les opérateurs or et nor	- 57 -
Figure 2.7 Exemple de test incluant les opérateurs xor et xnor	- 58 -
Figure 2.8 Contrôle d'activation	- 59 -
Figure 2.9 Initialisation d'une sortie double rail.....	- 62 -
Figure 2.10 Circuit d'émission d'une constante sur un canal	- 64 -
Figure 2.11 Synthèse MR[2] de la porte AND.....	- 65 -
Figure 2.12 Synthèse MR[4] de la porte AND.....	- 65 -
Figure 2.13 Synthèse MR[2] de la porte OR.....	- 66 -
Figure 2.14 Synthèse MR[2] de la porte XOR.....	- 67 -
Figure 2.15 Synthèse MR[4] de la porte XOR.....	- 67 -
Figure 2.16 Full Adder (FA) MR2	- 68 -
Figure 2.17 Full Adder (FA) MR4	- 68 -
Figure 2.18 Multiplieur MR[2]	- 69 -
Figure 2.19 Acquittement de la commande	- 70 -
Figure 2.20 Fourche des données en entrée	- 70 -
Figure 2.21 Acquittement des données entrantes.....	- 71 -
Figure 2.22 Réunion des sorties	- 71 -
Figure 2.23 Bufferisation en sortie d'une structure de choix à plusieurs gardes.....	- 72 -
Figure 2.24 Décomposition d'une porte OR à n entrées.....	- 73 -
Figure 2.25 Décomposition d'une porte AND à n entrées.....	- 74 -
Figure 2.26 Décomposition d'une porte de muller à 8 entrées.....	- 75 -
Figure 2.27 Fourche non bufferisée (gauche) et bufferisée (droite)	- 75 -
Figure 2.28 Synthèse du Latch.....	- 78 -
Figure 2.29 Fork bufferisé en entrée	- 79 -
Figure 2.30 Fork bufférisé en sortie	- 79 -
Figure 2.31 Synthèse du Join bufferisé en entrée.....	- 80 -
Figure 2.32 Synthèse du Join bufferisé en sortie	- 81 -
Figure 2.33 Merge bufferisé en entrée	- 82 -
Figure 2.34 Merge bufferisé en sortie	- 83 -
Figure 2.35 Multiplexeur bufferisé en entrée.....	- 84 -
Figure 2.36 Multiplexeur bufferisé en sortie finale.....	- 84 -
Figure 2.37 Demultiplexeur bufferisé en entrée.....	- 86 -
Figure 2.38 Démultiplexeur bufferisé en sortie	- 86 -
Figure 2.39 Disposition du coût structurel sur le réseau de Pétri.....	- 87 -
Figure 2.40 Réseau de Pétri du composant hétérogène.....	- 90 -
Figure 2.41 Synthèse du composant hétérogène	- 93 -
Figure 2.42 Labellisation du réseau de pétri avec le coût structurel	- 94 -
Figure 3.1 Flot TAST	- 96 -
Figure 3.2 Interface TAST	- 96 -
Figure 3.3 Compilateur CHP	- 97 -
Figure 3.4 Simulateur de réseau de pétri.....	- 97 -
Figure 3.5 Convertisseur Chp2vhdl	- 98 -
Figure 3.6 Synthèse QDI.....	- 98 -
Figure 3.7 Synthèse Micropipeline	- 99 -
Figure 3.8 Trace des canaux ("Trace –channels").....	- 107 -
Figure 3.9 Trace variables.....	- 108 -
Figure 3.10 Statistiques sur les instructions	- 109 -
Figure 3.11 Statistiques sur les structures de choix	- 110 -
Figure 3.12 Consommation d'énergie des labels du process_1	- 112 -

Figure 3.13	Consommation d'énergie des processus du composant_1	- 113 -
Figure 3.14	Consommation des composants	- 113 -
Figure 4.1	Comparaison de bufferisation WCHB, PCHB, PCFB	- 116 -
Figure 4.2	Consommation d'un Fork bufferisé en entrée et en sortie	- 118 -
Figure 4.3	Consommation d'un Join bufferisé en entrée et en sortie	- 119 -
Figure 4.4	Merge bufferisé en entrée	- 120 -
Figure 4.5	Merge bufferisé en sortie	- 120 -
Figure 4.6	Consommation d'un Merge bufferisé en entrée et en sortie	- 121 -
Figure 4.7	Multiplexeur bufferisé en entrée	- 122 -
Figure 4.8	Multiplexeur bufferisé en sortie finale	- 123 -
Figure 4.9	Consommation d'un Multiplexeur bufferisé en entrée et en sortie	- 123 -
Figure 4.10	Demultiplexeur bufferisé en entrée	- 124 -
Figure 4.11	Demultiplexeur bufferisé en sortie finale	- 124 -
Figure 4.12	Consommation d'un démultiplexeur bufferisé en entrée et en sortie	- 125 -
Figure 4.13	Bufferisation normale (gauche) et désynchronisation intra canal (droite) ...	- 126 -
Figure 4.14	Consommation d'énergie d'un Latch acquitté digit à digit	- 127 -
Figure 4.15	Multiplexeur non optimisé	- 128 -
Figure 4.16	Multiplexeur avec ajout de synchronisation	- 128 -
Figure 4.17	Consommation du multiplexeur synchronisé	- 129 -
Figure 4.18	Décomposition d'une ALU	- 131 -
Figure 4.19	Consommation d'énergie d'un Latch en fonction de la base	- 132 -
Figure 4.20	Statistique dans un canal	- 133 -
Figure 4.21	Découpage d'un canal de donnée	- 133 -
Figure 4.22	Synchronisation inter canaux	- 134 -
Figure 4.23	Désynchronisation inter canaux	- 134 -
Figure 4.24	Consommation en fonction du codage de la commande	- 135 -
Figure 4.25	Consommation d'un multiplexeur codé en SR	- 136 -
Figure 4.26	Portes logiques décrites à partir d'une porte AND	- 137 -
Figure 4.27	Portes Egal et non Egal	- 137 -
Figure 4.28	Portes inférieure, supérieure et inférieure ou égale	- 137 -
Figure 4.29	Opérateurs avant fusion	- 138 -
Figure 4.30	Opérateurs après fusion	- 138 -
Figure 4.31	Relâchement de la synchronisation en sortie	- 139 -
Figure 4.32	Statistiques sur les gardes exécutées	- 140 -
Figure 4.33	Synthèse du multiplexeur régulier (gauche) hiérarchique (droite)	- 141 -
Figure 4.34	Multiplexeur hiérarchique vs multiplexeur régulier	- 143 -
Figure 4.35	Synthèse démultiplexeur hiérarchique	- 144 -
Figure 4.36	Démultiplexeur hiérarchique vs démultiplexeur régulier	- 144 -
Figure 4.37	Canal de commande découpé	- 145 -
Figure 4.38	Multiplexeur hiérarchique avec commande découpée	- 146 -
Figure 4.39	Structure hiérarchique avec commande découpée	- 146 -
Figure 4.40	Structure hiérarchique avec commande découpée avec la seconde commande codée en SR	- 147 -
Figure 5.1	Architecture du mini processeur	- 150 -
Figure 5.2	Probabilités du composant PC	- 157 -
Figure 5.3	Probabilités du décodeur	- 158 -
Figure 5.4	Probabilités de l'interface d'exécution	- 160 -
Figure 5.5	Probabilités de l'unité d'exécution	- 161 -
Figure 5.6	Probabilités d'une sous partie de l'unité d'exécution	- 162 -
Figure 5.7	Probabilités de la MDU	- 163 -

Figure 5.8 Taille des composants du Mini Processeur.....	- 164 -
Figure 5.9 Taille du Mini Processeur (V0, V1, V2 et V3).....	- 165 -
Figure 5.10 Répartition de la consommation obtenue par Nanosim (gauche) et par notre estimateur (droite)	- 166 -
Figure 5.11 Consommation du Mini Processeur.....	- 167 -
Figure 5.12 Consommation du composant PC.....	- 168 -
Figure 5.13 Consommation du Décodeur	- 169 -
Figure 5.14 Consommation de l'interface d'exécution	- 170 -
Figure 5.15 Consommation de l'unité d'exécution	- 171 -
Figure 5.16 Consommation de la MDU	- 172 -
Figure 5.17 Buffer WCHB MR[4]	- 173 -
Figure 5.18 Convertisseur MR[2] -> MR[4].....	- 173 -
Figure 5.19 Convertisseur MR[4] -> MR[2].....	- 173 -
Figure 5.20 Registre MR[4]	- 174 -
Figure 5.21 Mémoire ROM synchrone	- 175 -
Figure 5.22 Mémoire RAM synchrone	- 175 -
Figure 5.23 Interface registre	- 176 -
Figure 5.24 Interface mémoire instruction.....	- 176 -
Figure 5.25 Interface mémoire instruction d'entrée	- 177 -
Figure 5.26 Convertisseur MR[4] -> Bit.....	- 177 -
Figure 5.27 Interface mémoire instruction de sortie	- 177 -
Figure 5.28 Interface mémoire donnée	- 178 -
Figure 5.29 Interface mémoire donnée d'entrée.....	- 178 -
Figure 5.30 Interface mémoire donnée de sortie.....	- 179 -

Liste des Tableaux

Tableau 1-1 Codage Double Rail	- 11 -
Tableau 1-2 Performances du 80C51 asynchrone.....	- 28 -
Tableau 1-3 Performances de AMULET1 et de ARM6	- 30 -
Tableau 1-4 Performances du processeur AMULET2e	- 31 -
Tableau 1-5 Performances du processeur ASPRO.....	- 34 -
Tableau 1-6 Consommation d'énergie de MICA	- 35 -
Tableau 1-7 Performances du processeur TITAC.....	- 36 -
Tableau 1-8 Performances du processeur MiniMIPS	- 39 -
Tableau 1-9 Performances de Lutonium	- 41 -
Tableau 1-10 Performances du processeur DINAMIK.....	- 45 -
Tableau 1-11 Microprocesseurs asynchrones	- 46 -
Tableau 2-1 Coût de la bufferisation.....	- 61 -
Tableau 2-2 Consommation d'un AND pour les bases 2 et 4	- 66 -
Tableau 2-3 Consommation du XOR pour les bases 2 et 4	- 67 -
Tableau 2-4 Consommation de l'additionneur pour les bases 2 et 4	- 69 -
Tableau 2-5 Consommation du multiplieur pour les bases 2 et 4	- 69 -
Tableau 2-6 Estimation du nombre de transitions des portes de base.....	- 77 -
Tableau 5-1 Jeu d'instructions du mini processeur	- 155 -
Tableau 5-2 Evolution de la Taille du Mini Processeur.....	- 165 -
Tableau 5-3 Consommation obtenue par Nanosim et par la Méthodologie.....	- 166 -
Tableau 5-4 Evolution de la consommation du Mini Processeur.....	- 167 -
Tableau 5-5 Temps de simulation des quatre versions de processeur.....	- 167 -
Tableau 5-6 Evolution du temps d'exécution en pourcentage.....	- 167 -
Tableau 5-7 Evolution de la consommation du PC.....	- 168 -
Tableau 5-8 Evolution de la consommation du Décodeur	- 169 -
Tableau 5-9 Evolution de la consommation de l'Interface d'Exécution	- 170 -
Tableau 5-10 Evolution de la consommation de l'Unité d'Exécution	- 171 -
Tableau 5-11 Evolution de la consommation de la MDU.....	- 172 -

Introduction

Contexte

La fin du XX^{ième} siècle a vu l'apparition d'une variété importante d'appareils électroniques portables (téléphones cellulaires, PC bloc-notes, mini-PC) intégrant des fonctionnalités de plus en plus complexes et diverses à des fréquences d'horloge de plus en plus élevées. La complexité grandissante de ces appareils s'est accompagnée d'une augmentation préjudiciable de la consommation d'énergie à tel point que l'autonomie de ces appareils est devenue un facteur limitant pour la viabilité de ces circuits.

La recherche réalisée pour l'amélioration des batteries d'alimentation est déjà très avancée. La densité d'énergie des accumulateurs NiMH (nickel métal hybride), Li-ion (lithium ion) et NiCd (nickel cadmium) ne devrait plus changer notablement et les autres idées émises dans ce domaine sont encore très loin d'être commercialisables.

C'est donc aux concepteurs de circuits intégrés de se pencher sur de nouvelles méthodes pour réduire la consommation d'énergie. C'est devenu ces dernières années un axe de recherche prioritaire des laboratoires de conception de circuits intégrés dans le monde.

Réduire la consommation d'énergie des circuits intégrés qui composent un appareil portable se traduit naturellement par un accroissement de l'autonomie de ces appareils, cela permet en plus une diminution de la taille de ces appareils car la taille des sources d'alimentation se voit réduite.

Les appareils d'aujourd'hui utilisent des circuits intégrés synchrones. C'est-à-dire, qu'il s'agit de circuits dont le contrôle est effectué à l'aide d'une horloge. Cependant, on sait pertinemment que la logique synchrone dissipe beaucoup d'énergie (pour ne pas dire la plupart) dû à l'activité ininterrompue de ce signal d'horloge lors du fonctionnement d'un appareil. A cette dissipation propre à l'horloge s'ajoute la consommation sans cesse de l'ensemble du circuit qui dans la plupart des cas ne fait aucun traitement effectif.

Des idées intéressantes ont ainsi émergé afin d'atténuer cette dissipation excessive. Une solution qui consiste à inhiber l'activité de l'horloge à l'aide du "gated clock" est théoriquement intéressante [BENI-97]. Cette technique consiste à coupler certaines branches de l'arbre d'horloge à des portes logiques AND, supprimant ainsi l'activité de l'horloge en aval. Mais celle-ci se confronte en pratique au problème de la consommation induite par le contrôle propre du "gated clock". Le compromis entre la consommation de l'horloge et la consommation induite par le contrôle du "gated clock" ne pèse pas toujours en faveur du "gated clock" (surtout que la surface est augmentée pour assurer ce contrôle).

Une autre solution est apparue, elle utilise de la logique permettant l'utilisation des deux fronts de l'horloge [STRO-00]. Cette solution est à première vue très avantageuse puisqu'elle permet de réduire de moitié le nombre de commutations de l'horloge. Cependant, elle nécessite l'utilisation d'une logique qui est plus complexe pour permettre la détection des deux fronts. La réduction de la consommation d'énergie de l'activité de l'horloge se voit rapidement surpasser par la consommation induite par ces cellules complexes.

Objectifs

On constate donc que réduire la consommation d'énergie d'un circuit intégré n'est pas une mince affaire. L'objectif de cette thèse est de permettre la réalisation des circuits intégrés numériques à faible consommation d'énergie. Nous proposons trois étapes importantes que nous présentons succinctement ici, elles sont plus longuement développées dans la suite du manuscrit de thèse.

1^{ère} étape : Passage à la logique asynchrone

Le regain en popularité de la conception asynchrone est révélateur de la volonté des chercheurs à apporter des solutions innovantes en terme de réduction de la consommation d'énergie. La logique asynchrone règle définitivement les nombreux problèmes liés à l'horloge dont sa consommation intrinsèque excessive, puisque celle-ci est remplacée par un contrôle local du type "poignée de mains" (contrôle mutuel entre composants) [RENA-00]. Dans un second temps, la logique asynchrone permet de ne plus dissiper de l'énergie inutilement. En effet, elle présente l'avantage de dissiper de l'énergie seulement lorsqu'un travail effectif est réalisé par le circuit. Lorsque les parties du circuit n'ont pas de travail à faire, elles sont naturellement au repos.

Ces raisons intéressantes et qui ont été vérifiées lors de nombreuses réalisations de circuits intégrés asynchrones nous ont convaincus de choisir la logique asynchrone dans notre quête d'implémentation de microprocesseurs à faible consommation d'énergie.

2^{ème} étape : Réalisation d'outils d'estimation de l'activité et de la consommation d'énergie

Le deuxième pas important est de pouvoir offrir aux concepteurs la possibilité de disposer d'outils permettant d'obtenir des informations sur la répartition de l'activité et de la consommation d'énergie dans un circuit asynchrone Quasi Insensible aux Délais (QDI) lors d'une simulation donnée. En asynchrone, on a relevé très peu de recherches sur la mise en place d'outils d'estimation de l'activité et de la consommation d'énergie. Il s'agit pourtant d'outils très importants pour la réalisation de circuits intégrés asynchrones à faible consommation d'énergie. Le concepteur peut en effet exploiter ces informations pour être guidé dans ses choix d'implémentation et ce très tôt dans le flot de conception.

3^{ème} étape : Proposition de techniques d'optimisation des circuits

Le troisième pas important est de proposer des techniques judicieuses d'optimisation des circuits asynchrones. Ce sont des techniques d'optimisation consécutives à l'analyse des résultats fournis par la méthodologie d'estimation d'énergie d'une part et par l'outil d'estimation d'activité (exploitation des probabilités d'exécution) d'autre part. Ces techniques d'optimisation peuvent être des informations utiles directement à l'outil de synthèse (pour la synthèse automatique de circuits asynchrones à faible consommation d'énergie) ou au concepteur (pour lui donner des pistes d'optimisation en fonction des probabilités d'exécution par exemple).

Contribution

Dans le cadre de cette thèse les contributions sont les suivantes.

- Participation à la spécification d'un estimateur d'activité

L'estimateur d'activité est un outil qui, comme son nom l'indique, permet d'estimer et plus précisément de fournir des informations sur la répartition de l'activité d'un circuit lors d'une simulation donnée. Il donne des statistiques intéressantes sur l'exécution des sous parties d'un circuit asynchrone [SLIM-04].

Cet outil a été réalisé et intégré dans le flot TAST [TAST-02] par des informaticiens du groupe CIS. Il est aujourd'hui opérationnel.

- Formalisation d'une méthodologie d'estimation de la consommation d'énergie

Il s'agit d'une méthodologie permettant d'estimer en terme de nombre de commutations de porte l'énergie d'un circuit asynchrone Quasi Insensible aux Délais (QDI) lors d'une simulation donnée [SLIM-03]. Cette estimation est réalisée au niveau de la spécification CHP d'un circuit asynchrone à partir des équations de dépendance [DINH-02b]. Ces informations sur la consommation d'énergie d'un circuit sont offertes très tôt au concepteur dans le flot de conception.

La méthodologie d'estimation de la consommation d'énergie a été complètement formalisée, l'outil permettant d'utiliser cette méthodologie a été spécifié. Il reste à l'intégrer dans le flot de conception TAST.

- Définition de techniques d'optimisation de la consommation d'énergie

Un ensemble de techniques d'optimisation a été proposé pour réduire la consommation d'énergie. Ces techniques d'optimisation sont appliquées en fonctions des estimations de la consommation d'énergie et de l'activité.

- Evaluation de la méthode sur un processeur MIPS asynchrone

Il s'agit d'un processeur RISC 32 bits de la famille MIPS. Ce processeur a été implémenté en quatre versions correspondant à trois étapes d'optimisation. Ces versions existent sous forme de netlists VHDL mappées sur une technologie ST 0.12µm.

Plan du manuscrit

Le Chapitre 1 est un état de l'art sur les concepts fondamentaux de la logique asynchrone, de la consommation d'énergie et des microprocesseurs asynchrones.

Dans une première partie nous présentons les caractéristiques de base de la logique asynchrone, les avantages et les inconvénients. Une brève comparaison est réalisée entre la logique synchrone et la logique asynchrone pour justifier la réapparition de celle-ci. Nous présentons les notions élémentaires permettant de comprendre le concept de l'asynchrone : les types de codage des données (données groupées et double rail) et des protocoles qui peuvent leur être associés (protocole 2 phases et 4 phases), la porte de muller (qui est la porte de l'asynchrone par excellence), les différentes architectures asynchrones existantes en fonction de leurs hypothèses temporelles (DI, QDI, SI, micropipeline et Huffman) et les bufferisations possibles (WCHB, PCHB, PCFB)

Dans une seconde partie de ce chapitre, nous présentons les techniques d'estimation et d'optimisation de la consommation d'énergie. Quelques rares techniques ont été proposées par certains laboratoires de recherche pour estimer la consommation d'énergie des circuits intégrés asynchrones (car l'utilisation des outils synchrones n'est pas possible). Ensuite nous détaillons certaines techniques permettant l'optimisation des circuits intégrés asynchrones

(pipeline asynchrone, les structures de contrôle, optimisation des communications, utilisation du paramètre $E\tau^2$, les techniques de réduction au niveau système, les bibliothèques dédiées etc.). Une troisième partie dresse une liste exhaustive des microprocesseurs asynchrones réalisés ces 15 dernières années. Les plus importants et ceux qui ont été à la base d'une commercialisation sont développés. Un tableau final permet de faire la synthèse de tous les processeurs asynchrones et de leurs performances.

Le Chapitre 2 présente et développe la formalisation de la méthodologie d'estimation de la consommation d'énergie des circuits asynchrones Quasi Insensible aux Délais. Il s'agit d'une estimation du nombre de commutations des portes d'un circuit lors d'une simulation donnée. Cette estimation a la particularité d'être obtenue très tôt dans le flot de conception, au niveau de la spécification CHP du circuit. Cette méthodologie se présente sous forme de deux estimations que nous nommons estimation structurelle et estimation dynamique. Ces deux estimations sont détaillées tout au long du chapitre. Un exemple que nous qualifions d'hétérogène illustre notre méthodologie d'estimation de la consommation d'énergie.

Le Chapitre 3 est dédié aux outils. Il présente l'estimateur d'activité et l'estimateur de la consommation d'énergie. L'estimateur d'activité a été spécifié et intégré dans le flot de conception TAST [TAST-02]. Toutes les options de cet outil telles qu'elles existent à l'heure de la rédaction de ce manuscrit de thèse sont détaillées. La description de cet outil met en avant les possibilités riches et variées de cet estimateur d'activité.

La spécification de l'estimateur de la consommation d'énergie telle qu'on souhaite l'intégrer dans le flot de conception TAST est présentée. Cet estimateur utilise la méthodologie d'estimation de la consommation d'énergie qui est présentée au chapitre 2.

Le Chapitre 4 propose une exploration architecturale afin d'identifier les implémentations qui consomment le moins d'énergie. Cette étude approfondie utilise les estimations de la consommation d'énergie et d'activité permettant d'observer les différences de comportement entre les différentes implémentations. Ce chapitre présente deux axes principaux : fournir des informations utiles à l'outil de synthèse (pour la synthèse automatique de circuits à faible consommation d'énergie) et des informations au concepteur pour le guider dans ses choix d'optimisation.

Le dernier chapitre (Chapitre 5) est une application pour évaluer les théories d'estimation (activité et consommation d'énergie) et d'optimisation des chapitres précédents. Pour ce faire, nous choisissons un exemple de taille et de complexité assez importantes puisqu'il s'agit d'un mini processeur de la famille MIPS.

Une première partie est consacrée à la description des blocs et du jeu d'instructions du mini processeur. Ensuite nous proposons trois étapes d'optimisation (le processeur original est noté V0, et les versions optimisées sont notées V1, V2 et V3). Des estimations sont réalisées sur ces versions de processeur pour montrer les réductions de la consommation d'énergie apportées par ces optimisations.

Enfin nous présentons l'architecture de certaines parties du circuit qui ont permis la réalisation du mini processeur final.

Nous proposons en fin de manuscrit une conclusion qui fait la synthèse du travail qui a été réalisé tout au long de la thèse et nous ajoutons une partie perspective qui présente les travaux en cours et les travaux connexes à réaliser.

Chapitre 1

Etat de l'art :

Logique asynchrone, Consommation d'Energie et Microprocesseurs Asynchrones

1.1 Logique Asynchrone

La logique asynchrone est apparue au milieu des années 1950 avec l'avènement des circuits intégrés numériques. Elle fût très rapidement abandonnée au profit de la logique synchrone à cause d'une plus grande difficulté et complexité de conception. Pendant près d'un demi siècle, la logique synchrone s'est imposée dans la conception de circuits intégrés. Les évolutions technologiques et les besoins récents en terme de performance ont accéléré le retour de la logique asynchrone [RENA-00]. Nous présentons dans ce premier paragraphe les concepts fondamentaux de la logique asynchrone.

1.1.1 Asynchrone versus Synchrone

De nos jours, la plupart des circuits réalisés sont conçus en logique synchrone. Les circuits synchrones sont caractérisés par l'utilisation de la logique binaire et d'une notion de temps discret via la distribution d'un signal global qu'on appelle l'horloge. L'horloge permet de commander et de synchroniser l'ensemble des blocs présents dans un circuit (Figure 1.1).

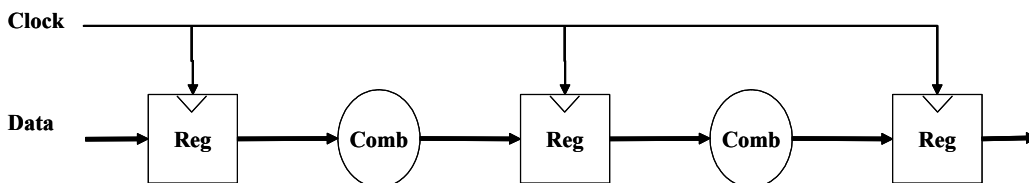


Figure 1.1 Structure de base d'un circuit synchrone

Les circuits asynchrones sont fondamentalement différents, ils continuent d'utiliser la notion de logique binaire mais n'ont plus recours à un signal de synchronisation globale. L'horloge est remplacée par un système de communication de type poignée de main (ou "handshaking" en anglais) entre les différents composants du circuit. Cette communication particulière permet d'assurer le transfert et la synchronisation de données entre les composants. La communication poignée de main nécessite l'ajout de signaux de contrôle qui sont le signal de requête et le signal d'acquiescement. La Figure 1.2 représente un circuit asynchrone.

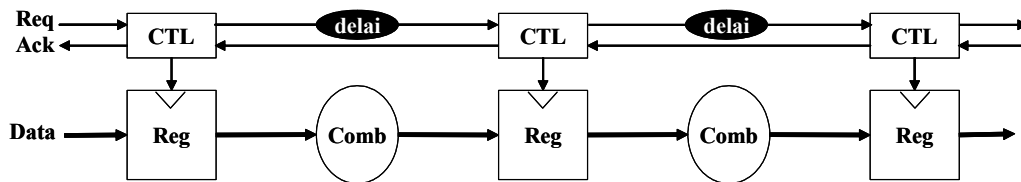


Figure 1.2 Structure de base d'un circuit asynchrone

Cette délocalisation du contrôle apporte un certain nombre d'avantages et d'inconvénients. Les avantages de la conception asynchrone se présentent à différents niveaux :

- Suppression du "clock skew"

Un effet préjudiciable directement attribuer à l'utilisation d'une horloge dans les circuits synchrones est le "clock skew". Le "clock skew" correspond à la différence de temps d'arrivée de l'horloge (qui part d'une source unique) aux composants qui reçoivent ce signal. Le routage de l'arbre d'horloge représente ainsi une étape à part entière lors du Placement&Routage d'un circuit, celui-ci doit être méticuleusement équilibré pour faire face à ce problème de décalage de temps d'arrivée du signal d'horloge. Ce problème devient de plus en plus critique avec l'évolution technologique car le délai des fils tend à devenir supérieur au délai dans les portes logiques.

Les circuits asynchrones ne rencontrent pas ce problème car le contrôle dans les circuits asynchrones n'est plus centralisé mais local. De plus, le contrôle entre différents blocs est fait point à point, il n'y a donc aucune distribution du signal de contrôle.

- Le pipeline

La conception asynchrone présente une propriété intéressante qui est l'élasticité. Cela signifie que la taille d'un pipeline peut très facilement être modifiée. On peut ajouter du pipeline pour accroître le débit d'un circuit ou en ôter pour privilégier la surface et la consommation d'énergie en toute simplicité sans introduire d'erreurs d'exécution ni avoir à re-synthétiser l'ensemble du circuit [YUN-96]. Pour les circuits synchrones, introduire du pipeline dans un circuit est beaucoup plus difficile à concevoir. En effet, cela nécessite d'une part de synthétiser l'ensemble du circuit pour satisfaire les nouvelles contraintes temporelles introduites par l'ajout d'un pipeline (ou par la suppression d'un étage de pipeline); d'autre part, il faut s'assurer que le nombre de pipeline dans des chemins qui se joignent (convergence en "ET") est identique dans chaque branche afin de ne pas créer d'erreurs fonctionnelles.

- Modularité

Un point appréciable, qui est de plus en plus reconnu par les concepteurs de circuits, est la réutilisation des circuits asynchrones grâce à la propriété inhérente de modularité. Grâce au contrôle local qui ne nécessite absolument aucune contrainte de temps global, les parties d'un circuit asynchrone peuvent très facilement être réutilisées dans d'autres circuits. Cette propriété permet de rendre les circuits asynchrones très portables, qui plus est, lors d'une migration technologique. Une synthèse générale n'est pas nécessaire contrairement aux circuits synchrones où il est indispensable de synthétiser l'ensemble du circuit pour déterminer les contraintes de temps de fonctionnement du circuit (fréquence de l'horloge).

- Consommation d'énergie

Les circuits asynchrones sont des candidats intéressants à la faible consommation d'énergie en raison du fait qu'ils cessent leur activité lorsque aucune tâche n'est accomplie. Les parties qui ne participent pas à l'exécution d'une tâche ne consomment aucune énergie dynamique. De plus, ces circuits ont la possibilité de passer instantanément du mode endormi (aucune consommation d'énergie) à un mode de pleine activité (maximum de débit).

- Vitesse

Les circuits synchrones présentent le handicap de fonctionner à la vitesse de l'horloge. Celle-ci est déterminée par le chemin critique du circuit, c'est-à-dire le chemin le plus lent. Même si l'exécution d'un programme utilise un chemin autre que le chemin critique, ce dernier est contraint d'aller à la vitesse fixée par le chemin critique. De plus, si on désire optimiser la vitesse d'un circuit synchrone, le seul champ d'action est le chemin critique puisque la vitesse du circuit est fixée par ce dernier. En revanche, les circuits asynchrones gèrent localement le transfert des données et l'exécution des instructions. Il n'y a plus de contrainte forte liée à un contrôle global comme il est question dans les circuits synchrones. L'avantage immédiat est qu'à chaque chemin du circuit correspond un temps d'exécution, et lorsque ce chemin est emprunté, le temps exécuté correspond au temps propre du chemin emprunté. Si bien que lorsqu'on emprunte souvent des chemins qui ne sont pas le chemin critique, la vitesse moyenne d'un circuit se voit nettement améliorée par rapport aux circuits synchrones [RENA-00]. De surcroît, si on optimise en vitesse des chemins du circuit qui ne sont pas le chemin critique, la vitesse du circuit est quand même améliorée. Donc, toutes optimisations sur l'ensemble du circuit participent à l'optimisation de la vitesse.

- Emission électromagnétique

L'avantage qu'on peut mettre en avant des circuits asynchrones est la faible émission électromagnétique (point crucial dans le domaine des télécommunications notamment). Dans les circuits asynchrones, les nombreuses parties du circuit fonctionnent à des instants différents et à des vitesses différentes plutôt que de commuter simultanément au signal de l'horloge comme dans les circuits synchrones. Il en résulte une diminution de l'émission électromagnétique et une répartition dans le temps de l'activité du circuit [PANY-02] [BOUE-04a]. De plus, l'architecture d'un circuit asynchrone peut être optimisée pour minimiser les émissions électromagnétiques [PANY-04].

- Sécurité

Ces dernières années, une propriété intéressante a émergé des circuits asynchrones. Il s'agit de la sécurité contre les attaques DPA (Differential Power Analysis) que présentent les circuits asynchrones, notamment les circuits Quasi Insensibles aux Délais [BOUE-04b] [MONN-04]. Les attaques DPA correspondent à l'étude des courants d'alimentation dans un circuit; les fluctuations de courants dans un circuit permettent de dévoiler les données qui sont manipulées. Les circuits synchrones sont des circuits qui signent beaucoup, c'est-à-dire qu'ils dévoilent très facilement les variations de la consommation en fonction des données manipulées. De plus, la consommation est bien localisée dans le temps puisqu'elle se produit aux fronts d'horloge. Il est très facile pour les "hackers" de circuits de lier la consommation d'énergie et les données manipulées dans les circuits synchrones.

Nous avons récemment implémenté une unité de multiplication et de division (MDU) asynchrone qui a été intégrée dans un cœur de processeur MIPS4Ksc [G3CARD-02].

L'intégration asynchrone permet de sécuriser cette partie importante du processeur contre les analyses de courant.

De plus, l'optimisation des circuits asynchrones contre les attaques DPA permet aussi de réduire les émissions électromagnétiques [BOUE-04a]. Réduire les émissions électromagnétiques d'un circuit permet de sécuriser davantage un circuit asynchrone.

Les nombreux avantages qu'on attribue à la logique asynchrone sont sérieusement pris en compte par les concepteurs de circuits. Il faut cependant préciser que la logique asynchrone présente certains inconvénients :

- Surface

La substitution de l'horloge par un contrôle local asynchrone représente un surcoût en terme de surface. L'exemple du processeur 80c51 asynchrone [GAGE-98] a montré que l'implémentation en asynchrone a augmenté la surface du processeur d'un facteur 2. Néanmoins, de nombreux travaux sont menés pour faire face à cet inconvénient, notamment la réalisation de bibliothèques spécifiques [MAUR-03] permettant de réduire considérablement la taille des portes asynchrones.

- Outils CAD

La logique asynchrone est réapparue lors de ces 15 dernières années, elle manque donc considérablement d'outils CAD et de test qui font sérieusement obstacle à son utilisation dans le milieu industriel. La mise au point d'outils de conception asynchrone est réalisée essentiellement par des universités. L'université de Manchester développe l'outil de conception Balsa [BARD-02] qui se base sur son prédécesseur Tangram [BERK-91] de l'université d'Eindhoven (en collaboration avec Philips). Tangram est le seul outil industriel propriétaire des handshakes solutions. Un flot très complet est actuellement développé par le laboratoire TIMA, il s'agit de TAST [TAST-02]. On verra dans la suite de ce chapitre, que ces outils ont permis la réalisation de circuits asynchrones très performants.

1.1.2 Les principes de base de l'asynchrone

La structure de base des circuits asynchrones telle qu'elle a été présentée à la Figure 1.2 peut être représentée d'une manière beaucoup plus générale comme nous le montre la Figure 1.3.

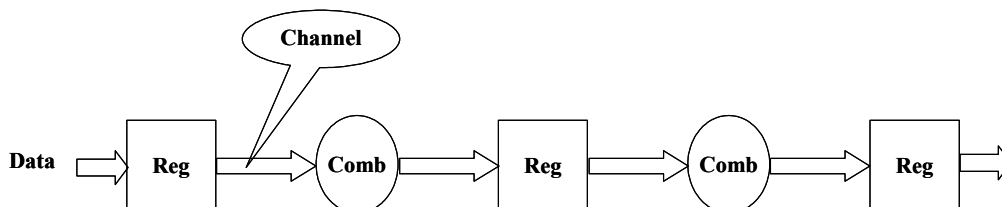


Figure 1.3 Représentation d'un circuit asynchrone

Le contrôle et les données ont été regroupés dans un type qui est propre à la conception asynchrone et qu'on appelle canal ("channel" en anglais). Ce canal intègre la donnée et les signaux de contrôle (la requête et l'acquiescement). Les sous parties qui suivent expliquent plus en détail le canal et les formes que celui-ci peut avoir.

1.1.2.1 Codage donnée groupée du canal

Le codage donnée groupée ("Bundled data" en anglais) utilise un bus de donnée pour coder l'information, celui-ci est identique aux bus utilisés dans la conception synchrone (bus de taille n pour coder n bits) auquel on associe un fil de requête et un fil d'acquittement pour le contrôle de la validité de la donnée (Figure 1.4).

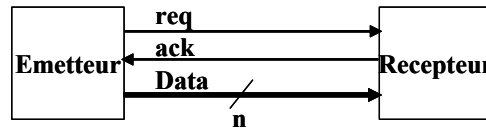


Figure 1.4 Canal codé en donnée groupée

On associe deux types de protocole de communication à ce codage du type donnée groupée:

- Codage donnée groupée utilisant un protocole 4 phases

Le terme 4 phases indique qu'il y a quatre actions de communication pour le transfert d'une donnée d'un émetteur vers un récepteur (Figure 1.5). Ce protocole est communément appelé le protocole Retour à Zéro (RTZ pour "Return To Zero"). On trouve aussi le terme de codage par niveau ("level encoding") pour caractériser le protocole 4 phases.

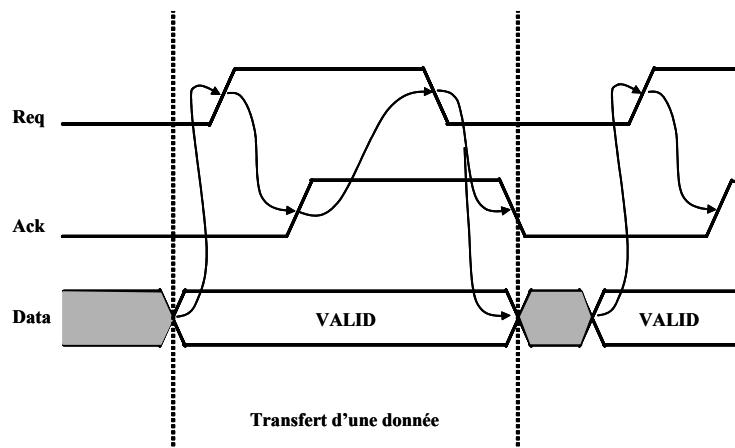


Figure 1.5 Codage donnée groupée associé au protocole 4 phases

Il existe quatre étapes principales pour décrire ce protocole de communication:

- (1) L'émetteur place une nouvelle donnée sur le bus de donnée (Data) et met le signal de requête (req) à 1.
- (2) Le récepteur reçoit la donnée et la requête, il indique à l'émetteur qu'il a bien reçu la donnée et la requête en mettant le signal d'acquittement (ack) à 1.
- (3) L'émetteur reçoit le signal d'acquittement et répond en mettant la requête à 0 pour indiquer qu'il invalide la donnée.
- (4) Le récepteur finalise la communication en mettant l'acquittement à 0 pour signifier qu'il est prêt à débiter un nouveau transfert de donnée.

Une fois les 4 actions de communication réalisées, l'émetteur peut initier une nouvelle communication en mettant de nouveau une donnée sur le bus de donnée et en activant la requête.

- Codage donnée groupée utilisant un protocole 2 phases

Le protocole 2 phases, ou protocole Non Retour à zéro (NRZ pour "No Return to Zero"), utilise deux fois moins d'actions de communication pour transférer une donnée (Figure 1.6). Il est à priori moins consommant et plus rapide que le protocole de communication 4 phases. Les informations sur les signaux de requête et d'acquittement sont codées en tant que transitions de signaux, il n'y a aucune différence entre la commutation 0->1 et 1->0.

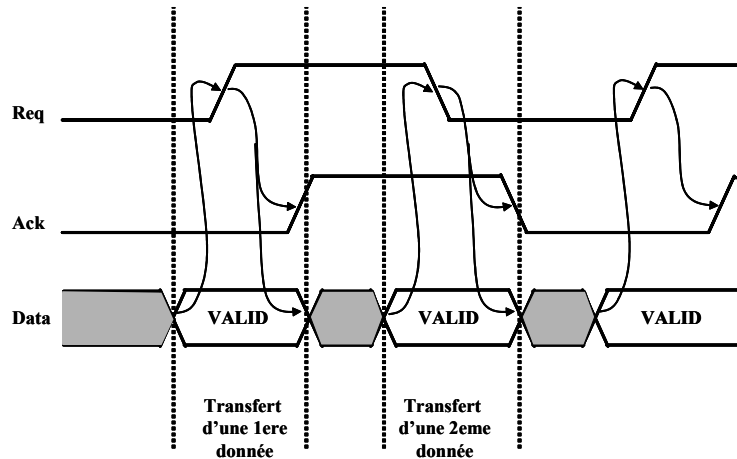


Figure 1.6 Codage donnée groupée associé au protocole 2 phases

Les deux étapes de ce protocole de communication sont les suivantes:

- (1) L'émetteur place une nouvelle donnée sur le bus de donnée (Data) et fait commuter le signal de requête (0->1 ou 1->0).
- (2) Le récepteur reçoit la donnée et la requête, il indique à l'émetteur qu'il a bien reçu la donnée et la requête en faisant commuter le signal d'acquittement (0->1 ou 1->0). La donnée sur le bus de donnée devient alors invalide.

Le transfert de la donnée se termine et l'émetteur peut initier une nouvelle communication en plaçant une nouvelle donnée sur le bus de donnée et en faisant commuter le signal de requête.

La communication utilisant un protocole 2 phases est en théorie plus rapide et présente deux fois moins de transitions que le protocole de communication 4 phases. On devrait a priori gagner en vitesse et en consommation d'énergie en utilisant ce protocole. En pratique, le protocole de communication 2 phases est plus complexe car il nécessite l'utilisation d'une logique sensible aux deux fronts. Cette logique est plus coûteuse en énergie et moins rapide en vitesse. De ce fait, le protocole de communication 4 phases est privilégié dans la conception des circuits asynchrones actuels.

1.1.2.2 Codage Double Rail du canal

Le codage Double Rail du canal diffère du codage donnée groupée. Le signal de requête n'est plus un fil séparé, il est intégré dans le bus de donnée qui présente désormais une taille double, c'est-à-dire que pour transférer une donnée de n bits on utilise un bus de taille $2n$. On utilise une taille double car chaque bit de la donnée est représenté par deux fils: un fil qui code la valeur 0 du bit et le second fil la valeur 1. Le fil d'acquittement du codage Double Rail est de même nature que pour le codage donnée groupée (il s'agit d'un fil séparé). Le schéma général d'un canal Double Rail est présenté à la Figure 1.7.

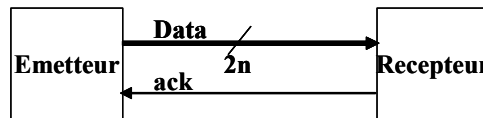


Figure 1.7 Canal codé en double rail

Etant donné que la requête est intégrée dans la donnée, il y a un codage double rail particulier pour chaque protocole de communication :

- Codage Double Rail utilisant un protocole 4 phases

Le codage double rail associé au protocole 4 phases est présenté au Tableau 1-1.

1 Bit		Valeur
Fil 1	Fil 0	
0	0	Aucune Donnée
0	1	0
1	0	1
1	1	Incorrect

Tableau 1-1 Codage Double Rail

Le protocole 4 phases utilise quatre actions de communication pour transférer une donnée d'un émetteur vers un récepteur. Le déclenchement du transfert se fait par la donnée elle-même et non plus par un signal de requête comme il était question pour le codage donnée groupée. La Figure 1.8 illustre le transfert d'une donnée d'un seul bit codée en Double Rail.

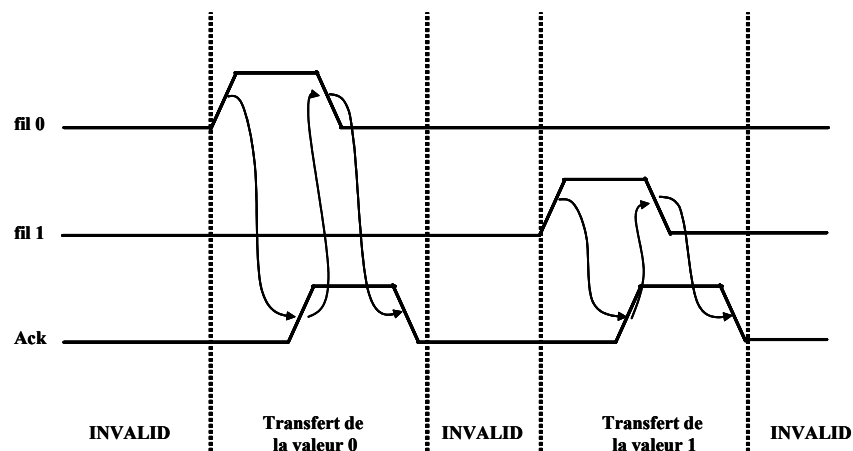


Figure 1.8 Codage Double Rail associé au protocole 4 phases

- (1) L'émetteur met une donnée valide sur le bus de donnée, c'est-à-dire que le fil 0 passe à 1 si la valeur 0 est transférée ou le fil 1 passe à 1 s'il veut transférer la valeur 1. Un seul fil est autorisé à passer à 1.
- (2) Le récepteur reçoit la donnée et le signale à l'émetteur en mettant le signal d'acquittement à 1.
- (3) À la réception du signal d'acquittement, l'émetteur invalide la donnée en faisant passer à 0 le fil du bit qui était à 1.
- (4) Le récepteur termine la communication en mettant à 0 le signal d'acquittement, signalant ainsi qu'il est libre, et qu'il est prêt à recevoir une nouvelle donnée.

Le codage double rail associé au protocole de communication 4 phases est le plus populaire et le plus utilisé. La robustesse des circuits que cette association produit est très appréciée des concepteurs de circuits asynchrones.

- Codage Double Rail utilisant un protocole 2 phases

Le codage double rail associé au protocole 2 phases correspond aux transitions des fils 0 et 1. La transition du fil 0 permet d'émettre la valeur 0; la transition du fil 1 permet d'émettre la valeur 1. Le transfert d'une donnée d'un bit est proposé à la Figure 1.9.

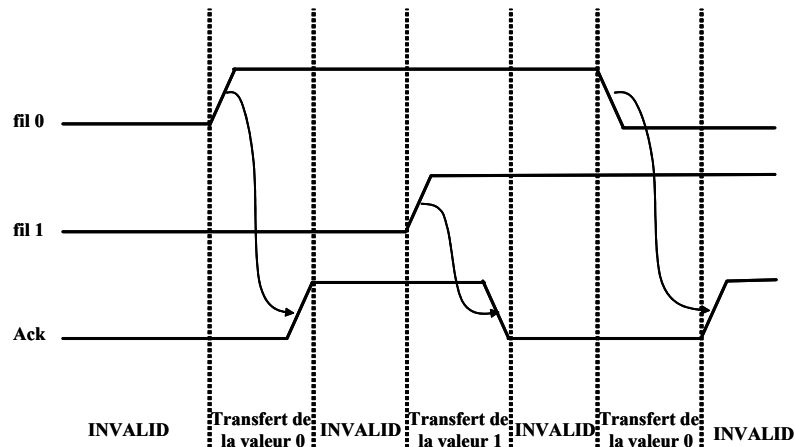


Figure 1.9 Codage Double Rail associé à un protocole 2 phases

(1) L'émetteur met une donnée valide sur le bus de donnée, c'est-à-dire que le fil 0 commute (0->1 ou 1->0) pour l'émission de la valeur 0 ou le fil 1 commute (0->1 ou 1->0) s'il veut envoyer la valeur 1. Un seul fil peut commuter à la fois.

(2) Le récepteur reçoit la donnée et le signale à l'émetteur en faisant commuter le signal d'acquiescement (0->1 ou 1->0).

La donnée est ainsi transférée et une nouvelle communication peut se faire.

Le codage Double Rail associé au protocole 2 phases est rarement utilisé (on le retrouve dans les liens séries par exemple [RENA-98]). La complexité est importante, car toute la logique du chemin de donnée doit être sensible aux deux fronts pour pouvoir satisfaire ce protocole. La complexité est beaucoup plus importante que celle du codage donnée groupée associé au protocole 2 phases, car pour ce dernier seuls les latches doivent satisfaire la sensibilité aux deux fronts, la manipulation des données restant identique au calcul qu'on connaît en logique synchrone.

1.1.2.3 La porte de Muller

Dans les circuits synchrones, l'horloge définit le moment où les signaux sont stables et valides. Entre deux fronts d'horloge, les signaux peuvent passer par un certain nombre d'états intermédiaires avant de se stabiliser à une valeur. Ces états intermédiaires sont des aléas, c'est-à-dire des valeurs de signaux qui dépendent de la logique et des délais dans cette logique. Ces états intermédiaires ne posent pas de problème puisque la stabilité des résultats est garantie aux fronts d'horloge. Dans les circuits asynchrones, nous sommes face à une situation qui est différente puisqu'il n'y a plus d'horloge, par conséquent il n'existe pas un signal global qui échantillonne les signaux. La conception asynchrone utilise des techniques d'implémentation qui permettent d'éviter que des signaux changent intempestivement de valeur (conception de

circuits sans aléas). La cellule qui caractérise les circuits asynchrones est la porte de Muller (ou "C-element"). Cette cellule propre aux circuits asynchrones permet de créer la synchronisation entre plusieurs signaux. Lorsque toutes les entrées sont à 0 la sortie passe à 0; lorsque toutes les entrées sont à 1 la sortie passe à 1; enfin, pour toutes les autres combinaisons des entrées la sortie reste inchangée.

La Figure 1.10 montre le symbole et la fonction logique d'une porte de muller à 2 entrées.

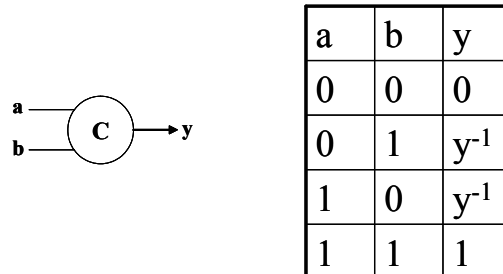


Figure 1.10 Porte de Muller

1.1.3 Les logiques asynchrones

Il existe plusieurs styles de logique asynchrone qu'on peut classer le plus simplement possible en considérant les hypothèses temporelles.

1.1.3.1 Circuits insensibles aux délais (DI)

Un circuit est insensible aux délais (DI pour "Delay Insensitive") lorsque les manipulations et les calculs qu'il réalise sont indépendants des délais des opérateurs et des fils [EBER-91]. Il s'agit d'un modèle temporel non borné, c'est-à-dire que le délai des fils et des portes est non borné (les résultats obtenus sont corrects sans introduire une notion de délai). Le récepteur du signal informe systématiquement l'émetteur lorsqu'il reçoit l'information sous forme d'un signal de "completion detection", l'émetteur attend la réception de ce signal avant d'émettre une nouvelle donnée. L'absence d'hypothèse temporelle apporte de nombreux bénéfices en terme de performance, notamment la vitesse. La communication se fait de manière optimale, aucune borne supérieure n'est introduite. Cependant, l'expérience dans la recherche des types de circuits asynchrones a montré qu'il est très difficile de concevoir des circuits insensibles aux délais (DI). La taille et la consommation d'énergie de ces circuits représentent un handicap majeur. De surcroît, la complexité croît plus rapidement que la fonctionnalité du circuit. La synthèse de ce type de circuit est difficile et leur utilisation impraticable pour de grands circuits. On peut même ajouter qu'il n'existe à ce jour aucun circuit DI de taille importante.

1.1.3.2 Circuits indépendant de la vitesse et quasi-insensible aux délais

Les circuits indépendants de la vitesse (SI pour "Speed Independent") utilisent un modèle de délai similaire aux circuits DI présentés précédemment. En revanche, ils supposent que les délais des fils, au lieu d'être non borné, sont tous nuls.

Les circuits Quasi Insensible aux délais (QDI pour "Quasi Delay insensitive") utilisent un modèle qui se positionne entre les circuits DI et SI. Les délais des fils sont supposés non bornés (comme dans le cas du DI) mais ils obéissent au principe de fourche isochrone [MART-90a]. Le principe de fourche isochrone se résume à: étant donné que le délai des fils est supposé non borné, les différences de temps de réception d'un signal par plusieurs récepteurs sont négligeables.

Nous avons regroupés les circuits QDI et SI car ils sont en théorie très proches, et en pratique il n'y a pas de réelles différences. La Figure 1.11 montre comment l'hypothèse des circuits QDI est interprétée par la théorie des circuits SI.



Figure 1.11 Hypothèse circuit QDI (gauche) et SI (droite)

On voit sur cette figure que les délais dans le cas QDI sont bornés et différents de zéro. Dans la fourche, chaque branche a un délai borné différent de zéro; cependant le principe de fourche isochrone impose que la différence de délai entre les branches est négligeable (inférieur à ϵ). On peut obtenir le circuit SI équivalent en déplaçant les délais des fils à l'intérieur de la porte: les fils ont désormais un délai négligeable (propriété SI) et la différence de délais (inférieur à ϵ) des branches de la fourche est négligeable (propriété des fourches isochrones).

Les circuits SI et QDI ont un large avantage sur les circuits DI. Ils permettent, grâce à l'hypothèse de fourches isochrones, l'implémentation de blocs complexes de façon beaucoup plus efficace que l'implémentation DI. On utilise pour les circuits SI et QDI des portes de base à une seule sortie au lieu de l'utilisation de portes complexes à plusieurs sorties pour les circuits DI.

1.1.3.3 Circuits micropipeline

Le style de conception micropipeline ("Bundled Data") a été proposé par Sutherland [SUTH-89], il représente une alternative aux pipelines élastiques synchrones. Le micropipeline correspond à l'association d'un chemin de donnée (qui utilise des techniques de délais bornés) et d'une partie contrôle (qui utilise des techniques d'implémentation DI). L'hypothèse temporelle utilisée s'applique à toutes les sous-parties du chemin de donnée où on suppose que les délais sont connus. Ces sous-parties du circuit sont ainsi séparées des autres sous-parties du circuit par des latches contrôlés par des réseaux de composants du chemin de contrôle. Ces réseaux de composants (réalisés à l'aide de portes de muller) permettent la synchronisation de toutes les parties du circuit. Pour avoir un comportement cohérent et fonctionnellement juste, des délais sont ajoutés sur le chemin de contrôle pour qu'il y ait correspondance entre le délai du chemin de donnée et le délai du chemin de contrôle.

On qualifie souvent les circuits micropipeline de circuits pseudo-synchrones. Au lieu d'avoir une horloge de fréquence fixe qui commande l'ensemble des latches du circuit, on a des contrôleurs locaux (partie contrôle) qui commandent les latches à des fréquences locales (correspondant aux délais du chemin de donnée). La structure générale d'un circuit micropipeline est montrée à la Figure 1.12.

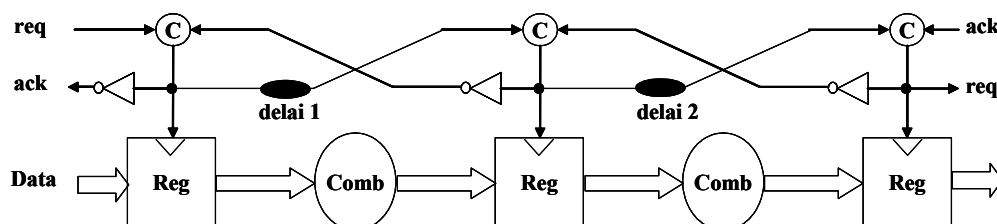


Figure 1.12 Circuit micropipeline

1.1.3.4 Circuits de Huffman

L'hypothèse temporelle des circuits de Huffman est essentiellement celle des circuits synchrones, à savoir les délais associés à tous les éléments et les interconnexions du circuit sont connus, ou du moins, leurs bornes supérieures sont connues. La synthèse des circuits de Huffman est faite à partir d'une table de flot qui s'apparente aux tables de vérité. La Figure 1.13 montre une table de flot et la machine à état associée. La table de flot présente des lignes qui correspondent aux états internes, et des colonnes qui correspondent aux combinaisons possibles des entrées. Dans le tableau figurent les états stables et les transitions ainsi que les sorties générées par ces transitions.



Figure 1.13 Une table de flot (gauche) et la machine à état correspondante (droite)

La synthèse est réalisée par des méthodes similaires utilisées par les machines à états obtenues à partir du tableau de Karnaugh. On suppose que le changement de valeur d'une entrée entraîne un passage du circuit par de nombreux états instables avant d'atteindre un état stable en un temps fini et connu. Dans le but de garantir l'absence de glitch sur les sorties, des cubes redondants (depuis le tableau de karnaugh) sont implémentés. Cette redondance ne permet pas d'éviter les glitches si plusieurs entrées changent en même temps. Les entrées doivent changer une à la fois, le circuit doit donc atteindre un état stable avant qu'un changement sur une autre entrée puisse survenir (mode fondamental). Les états sont obtenus par rétroactions de signaux via des délais pour garantir qu'aucun changement ne survient avant d'atteindre l'état stable. Lors de la conception de circuits séquentiels de Huffman, des méthodes synchrones sont utilisées.

L'approche de la conception des circuits de Huffman présente cependant de nombreuses limites. La première limite est liée au fait qu'une seule entrée peut changer à la fois, il est par conséquent très difficile d'utiliser ce mécanisme sachant que dans les circuits traditionnels plusieurs entrées sont amenées à changer en même temps. Ensuite, la méthodologie de conception de ces circuits ne permet pas de combiner de petits circuits dans un circuit hiérarchique. Enfin, un tout autre problème est le besoin d'ajouter des délais, ce qui peut dégrader les performances des circuits.

Les circuits "burst-mode" [DAVI-93] [FUHR-99] sont associés aux circuits de Huffman car ils présentent des similarités conceptuelles. La particularité des circuits burst-mode est que certaines combinaisons d'entrée ou sorties peuvent changer (qu'on nomme entrées ou sorties "bursts") contrairement aux circuits de Huffman où une seule entrée est autorisée à changer.

1.1.4 Les Bufferisations

La bufferisation correspond à l'ajout d'un buffer régit par un séquençement d'actions de communication (c'est équivalent à l'ajout d'étages de pipeline). Les buffers peuvent être de plusieurs natures en fonction du séquençement des actions.

Dans un circuit asynchrone, la bufferisation est essentielle pour assurer le bon fonctionnement d'un circuit, elle permet aussi d'accroître les performances des circuits. L'insertion de buffers dans les parties combinatoires permet d'accroître le pipeline. Le schéma classique d'une bufferisation est montré à la Figure 1.14.

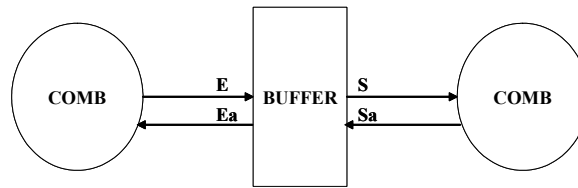


Figure 1.14 Schéma général d'une bufferisation

L'ordonnancement des entrées et des sorties donne lieu à différents styles de bufferisation. Il existe quatre bufferisations principales en asynchrone; ces bufferisations ont été initialement proposées par A. Lines [LINE-95], elles sont explicitées en détail dans [VIVE-01] et sont succinctement présentées dans la suite de ce paragraphe.

1.1.4.1 Sans bufferisation : le protocole séquentiel

Le protocole séquentiel présente l'expansion de communication (HSE) suivante :

$$P = * [[E]; S+; [\overline{Sa}]; Ea-; [\overline{E}]; S-; [Sa]; Ea+]$$

L'expansion des communications est une représentation qui détaille l'ordonnancement des signaux à l'intérieur d'un canal. C'est la représentation intermédiaire des règles de production permettant la synthèse des circuits asynchrones selon la méthode d'Alain Martin [MART-91]. E est la donnée entrante, Ea l'acquittement correspondant ; S est la donnée sortante, Sa l'acquittement correspondant.

- 1) À l'arrivée d'une requête [E], on envoie une requête sur la sortie S+
- 2) La sortie S répond par un acquittement $[\overline{sa}]$ qui permet d'acquitter l'entrée Ea-
- 3) L'entrée est invalidée $[\overline{E}]$ permettant ainsi d'invalider la sortie S-
- 4) La sortie devient de nouveau disponible grâce à [Sa], et le buffer se libère en mettant Ea+ , autorisant ainsi une nouvelle communication.

Cet ordonnancement montre que les entrées sont envoyées en sortie. La donnée entrante E est envoyée sur la sortie S et l'acquittement de sortie Sa est envoyée sur l'acquittement d'entrée Ea. En réalité la bufferisation séquentielle n'est pas une bufferisation à proprement parler puisque aucune mémorisation n'est réalisée. Les données sont propagées de l'entrée vers la sortie et les acquittements sont propagés de la sortie vers l'entrée comme l'illustre la Figure 1.15.

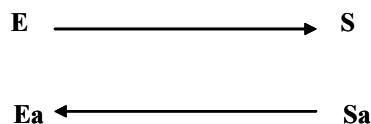


Figure 1.15 Buffer séquentiel

1.1.4.2 La bufferisation WCHB

La bufferisation WCHB qui signifie "Weak Condition Half Buffer" est la bufferisation communément utilisée. L'expansion des communications s'écrit pour cette bufferisation :

$$P = * [[E \wedge Sa] ; S+; Ea-; [\bar{E} \wedge \bar{Sa}] ; S-; Ea+]$$

1) Lorsqu'il y a une donnée valide E et que l'acquittement indique que l'étage suivant est libre ($[E \wedge Sa]$) alors une requête est émise en sortie S+ et la donnée d'entrée est acquittée Ea-.

2) Lorsque la donnée d'entrée est invalidée et que la sortie a été acquittée ($[\bar{E} \wedge \bar{Sa}]$) alors la sortie est invalidée (S-) et l'étage précédent est libéré (Ea+).

Cette bufferisation permet de synchroniser la phase de remise à zéro de l'entrée et de la sortie. La sortie est remise à zéro (S-) en même temps que le retour à l'état initial de l'acquittement d'entrée (Ea+); cette remise à zéro est synchronisée par l'acquittement de sortie (\bar{Sa}) et la donnée entrante (\bar{E}). Un tel ordonnancement est implémenté sous forme d'un buffer illustré à la Figure 1.16.

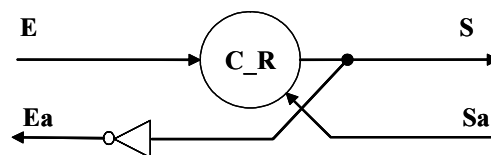


Figure 1.16 Buffer WCHB

C_R est une porte de muller avec reset qui permet de synchroniser la donnée entrante et l'acquittement de sortie (cf. Annexe A pour les portes asynchrones de base). Les buffers asynchrones sont longuement explicités dans [RIGA-02].

1.1.4.3 La bufferisation PCHB

La bufferisation PCHB ("PreCharge logic Half Buffer") est qualifié de "Precharge" car il existe une étape de pré-charge de la sortie S. Elle présente l'ordonnancement suivant :

$$P = * [[E \wedge Sa] ; S+; Ea-; [\bar{Sa}] ; S-; [\bar{E}] ; Ea+]$$

1) La première étape est similaire à la bufferisation WCHB

2) On attend que la sortie soit acquittée (et seulement la sortie $[\bar{Sa}]$) pour invalider la sortie (S-)

3) On attend enfin l'invalidité de la donnée entrante ($[\bar{E}]$) pour libérer l'étage précédent (Ea+).

La remise à zéro se fait sur la sortie, une fois que celle-ci est terminée on procède à la remise à zéro de l'entrée. Cet ordonnancement est plus performant que celui du protocole WCHB lorsque la sortie est plus rapide que l'entrée. Il n'est en effet pas nécessaire d'attendre le retour à zéro de la donnée d'entrée pour remettre la sortie à zéro. Le buffer généré par un tel ordonnancement est illustré à la Figure 1.17.

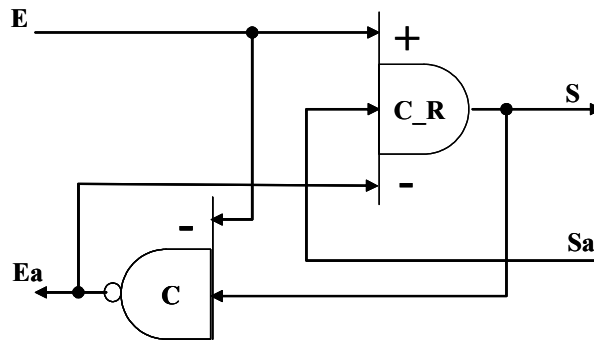


Figure 1.17 Buffer PCHB

Le buffer PCHB intègre des portes de muller dissymétriques à deux et trois entrées (cf. Annexe A).

1.1.4.4 La bufferisation PCFB

La bufferisation PCFB ("PreCharge logic Full Buffer") est très proche de la bufferisation PCHB. La seule différence est que les remis à zéro de l'entrée et de la sortie se font en parallèle au lieu de se faire séquentiellement (virgule au lieu du point virgule). L'ordonnancement s'écrit:

$$P = * [[E \wedge Sa] ; S+; Ea-; [[\overline{Sa}]; S-], [[\overline{E}]; Ea+]]$$

La bufferisation PCFB est une variante de la bufferisation PCHB. Il y a, dans cette bufferisation, une décorrélation de la remise à zéro de l'entrée et de la remise à zéro de la sortie, ceci grâce à l'introduction d'une variable interne au buffer. Cette bufferisation présente l'avantage d'être plus rapide que la bufferisation PCHB. Le circuit représentant un buffer PCFB [RIGA-02] est décrit à la Figure 1.18.

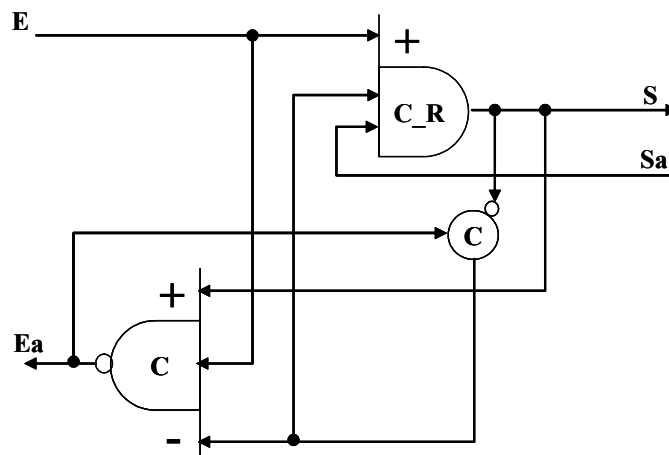


Figure 1.18 Buffer PCFB

1.2 La consommation d'énergie

La volonté d'apporter des solutions en vue de réduire la consommation d'énergie des circuits asynchrones se manifeste à différents niveaux du flot de conception. Ce paragraphe présente les techniques d'estimation de la consommation d'énergie d'une part et les techniques

d'optimisations de la consommation d'énergie d'autre part qui ont été développées ces dernières années.

1.2.1 Les techniques d'estimation de la consommation d'énergie

L'absence d'horloge dans les circuits asynchrones rend inutilisable les moyens communs de quantification de la dissipation d'énergie qui existent pour les circuits synchrones. Il a fallu, par conséquent mettre en place des techniques et des outils d'estimation propres à la logique asynchrone.

Les pionniers dans la mise au point d'outils permettant l'estimation de la consommation d'énergie sont l'université d'Utah en 1994. Une technique d'estimation de la consommation d'énergie de la représentation Pétri Net des circuits asynchrones micropipelines 2 phases a été proposée [PRAB-94]. Il semblait évident à cette date que pour concevoir des circuits à faible consommation d'énergie, un outil permettant d'estimer la consommation d'énergie était indispensable. Cet outil se devait de produire une estimation rapide et assez précise de l'énergie consommée par un circuit asynchrone. Les instigateurs de cette méthodologie partent d'une forme décomposée du circuit global, celui-ci est partitionné en 3 éléments : le bloc de contrôle (CB), le bloc datapath (DB) et enfin un sélecteur de donnée (PB). Ce partitionnement judicieux permet d'appliquer des techniques d'estimation adaptées à chaque type de blocs. L'estimation de la consommation d'énergie de l'élément de contrôle CB est réalisée en fonction de la topologie du circuit. Il s'agit d'une librairie de macromodules (CALL, XOR etc.) qui ont été simulées et estimées (pour une capacité de charge de sortie d'un buffer) à l'aide de HSPICE. L'estimation des blocs DB et PB est faite à l'aide des techniques d'estimation des blocs combinatoires de Ghosh-Devadas [GHOS-92] où le signal d'horloge est remplacé par une combinaison de requête/acquittement.

Certaines places du réseau de pétri initial sont rétro-annotées avec les estimations obtenues. Lors de la simulation du réseau de pétri, les consommations des places sont ajoutées pour calculer l'énergie dynamique du circuit. Cette méthodologie a été appliquée à un certain nombre de circuits, et a montré des résultats intéressants.

D'autres méthodes ont depuis émergé, notamment une estimation de la consommation d'énergie des circuits asynchrones insensibles à la vitesse (SI) [BEER-95]. Contrairement à la technique précédente, l'estimation est faite pour une classe plus générale de circuits de contrôle. De plus, l'estimation est faite sur une netlist de portes plutôt que sur des macro cellules prédéfinies (estimées à l'aide de HSPICE). Une énergie moyenne par transition de signal externe est établie à partir d'une forme dérivée du graphe de transition de signal (STG) qui est appelé SSTG (graphe de transition séquentielle de signal).

Un modèle d'énergie a été proposé dans [TIER-96]. Ce modèle est basé sur la spécification CSP [HOAR-78] des circuits asynchrones. Connaissant la procédure de synthèse, il a été possible d'établir les coûts de la structure du langage CSP, en particulier le coût des synchronisations. Il y a une transparence entre la spécification en CSP et le circuit synthétisé. Une affectation décrite en CSP correspond à un circuit CMOS.

Le langage CHP utilisé à Caltech est issu de CSP, on peut de ce fait établir un coût à la syntaxe de la spécification CHP. Un coût est attribué à chaque primitive du langage CHP telles que la composition parallèle ("||"), la composition séquentielle (";"), le choix gardé ("[]"), la répétition ("*"). A partir de ces coûts de base, des coûts plus élaborés peuvent être établis. Pour la séquentialité par exemple, le coût est défini selon la décomposition suivante :

$$C(\langle\langle i : 1..n : P_i \rangle\rangle; \langle\langle j : 1..m : Q_j \rangle\rangle) = C(\text{join}(n)) + C(\text{fork}(m)) + C(;) + C(\langle\langle i : 1..n : P_i \rangle\rangle) + C(\langle\langle j : 1..m : Q_j \rangle\rangle)$$

$C()$ est la fonction de coût d'un certain programme; P_i et Q_i sont deux processus du programme; $C(\text{join}(n))$ est le coût d'une jonction de n branches vers une branche et $C(\text{fork}(m))$ est le coût d'une fourche d'une branche vers m branches.

Lors d'une simulation, on peut calculer le coût dynamique à partir de ces coûts de synchronisation. Ce modèle peut facilement s'adapter à d'autres langages de spécification de circuits asynchrones, le circuit doit être fini et déterministe.

Des techniques d'estimation ont été proposées pour les circuits asynchrones quasi insensibles aux délais (QDI) [PENZ-02a]. Il s'agit d'un simulateur Esim qui se base sur la représentation logique du circuit asynchrone QDI appelé les règles de production (PR) [MART-91]. Ces règles de production ont été développées à Caltech et représentent la forme intermédiaire de la synthèse des circuits QDI. Les circuits QDI sont intéressants dans le contexte de l'estimation de la consommation d'énergie grâce à la localité de l'activité (absence d'une horloge globale) et à la mise au repos automatique des parties qui n'ont aucun traitement à faire.

Une estimation dynamique est faite grâce au simulateur qui comptabilise le nombre de transition des portes (qui sont pondérées) lors de la simulation des PR. Les capacités de charge sont déterminées à l'aide de statistiques. Ce simulateur a été utilisé pour estimer la consommation d'énergie dans différentes parties du processeur MiniMIPS [MART-97], une erreur de l'ordre de 10% par rapport à une simulation en HSPICE a été enregistrée. La précision de cette méthodologie est meilleure que celle des outils synchrones.

1.2.2 Les techniques d'optimisation de la consommation d'énergie

Des techniques d'optimisation des circuits asynchrones existent, elles sont de nature logicielle et matérielle. Quelques techniques sont proposées dans ce paragraphe.

1.2.2.1 Pipeline asynchrone

De nombreuses techniques ont été proposées et introduites pour améliorer les performances des circuits. Le pipeline est un moyen commun fréquent pour accroître le débit d'un circuit. Dans la conception synchrone, le pipeline est réalisé en insérant des registres à l'intérieur des parties combinatoires et en ajustant l'horloge principale en conséquence. La synthèse globale du circuit doit à nouveau être faite pour satisfaire les nouvelles contraintes de temps. En revanche, le pipeline dans les circuits asynchrones est réalisé en insérant des buffers/latches entre les étages sans procéder à la synthèse de l'ensemble du circuit. Il est beaucoup plus aisé d'accroître le pipeline en asynchrone qu'en synchrone. Cependant, la facilité à augmenter la profondeur du pipeline dans un circuit asynchrone doit être faite avec précaution. En effet, l'insertion de latches dans le but d'améliorer le débit d'un circuit induit l'ajout de logique supplémentaire et par conséquent une augmentation de la consommation d'énergie. Il s'agit de l'incontournable compromis vitesse/consommation d'énergie. Néanmoins, il existe une limite supérieure de buffer à ajouter dans un circuit pour atteindre le débit maximal. Au-delà de ce nombre de buffers, ajouter davantage de buffer augmente la consommation d'énergie sans pour autant augmenter la performance du circuit. Plus grave encore, l'ajout d'un nombre excessif de buffer se traduit par une baisse de performance du circuit due aux délais de l'excès de buffers qui peuvent ralentir le circuit [WILL-94]. Des modèles ont été proposés pour déterminer précisément le nombre optimal de buffer à ajouter dans un pipeline asynchrone afin d'obtenir le débit maximal. Une expérience intéressante a été proposée par [WILL-94] dans laquelle il utilise un anneau de buffers (Figure 1.19) où il fait varier le nombre de buffers et analyse les performances.

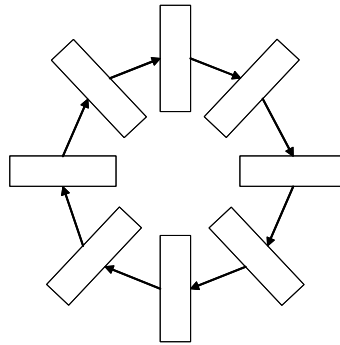


Figure 1.19 Anneau de buffers

Il a montré que les performances d'un anneau asynchrone peuvent varier entre un état bloqué lorsque le nombre de buffers est en dessous du nombre minimal, à un maximum de débit lorsque le nombre de buffers est optimal. Au-delà de ce nombre optimal de buffers, les performances diminuent et la dissipation d'énergie augmente (Figure 1.20).

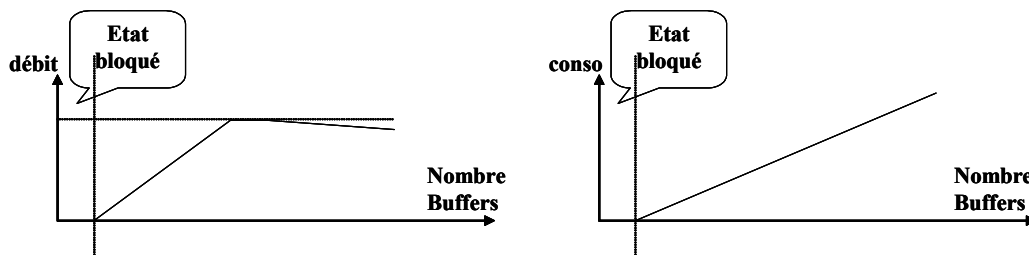


Figure 1.20 Débit et Consommation en fonction du nombre de buffers

D'autres modèles tel que le paramètre temps/énergie $E\tau^2$ proposé dans [MART-01] permet de déterminer la borne supérieure de buffers à ajouter dans un pipeline asynchrone pour obtenir le plus grand débit. Le nombre optimal de buffers qui sont à ajouter dans un circuit est appelé "slack matching" [MART-97]. Le "slack matching" n'affecte pas le fonctionnement correct d'un circuit lorsqu'on l'applique aux parties d'un circuit ne présentant pas d'arbitrage. Les parties du circuit ne présentant pas d'arbitrage sont appelées "slack elastic".

Une autre source de dissipation d'énergie est observée lorsque les instructions de branchement sont exécutées et réalisées. De manière générale, plus le pipeline est profond et plus la quantité d'énergie dissipée est grande lorsque le branchement est réalisé. Ceci s'explique par le fait qu'il y a autant d'instructions que d'étages de pipeline, par conséquent la dissipation d'énergie est directement proportionnelle au nombre d'instructions qui doivent être supprimées dans le pipeline. Pour remédier à cette dissipation excessive d'énergie, des solutions basées sur les prédictions de branchement sont utilisées. Lorsque la prédiction de branchement est proche de la réalité alors la consommation d'énergie se voit ainsi considérablement réduite. Cependant, la prédiction de branchement consomme de l'énergie si bien qu'il est intéressant de se demander si la consommation d'énergie de l'exécution totale se voit réduite. Une solution élégante à ce problème, qui exploite l'élasticité du pipeline asynchrone, est de contrôler la profondeur du pipeline en supprimant au besoin certains étages du pipeline. Ceci est réalisé dynamiquement, le processeur peut donc passer d'une configuration hautement pipelinée à une configuration présentant une profondeur de pipeline très faible, et ceci très rapidement lors de l'exécution d'un programme [EFTH-03]. La configuration dynamique du pipeline a le potentiel de réduire considérablement la consommation d'énergie. L'inconvénient de cela est qu'on a tendance à ralentir le processeur

mais cela permet tout de même d'économiser de l'énergie parce que peu d'instructions spéculées sont prises en mémoire et décodées. Cette configuration dynamique est réalisée au moyen de latches reconfigurables [LEWI-99] qui peuvent être transparents ("collapsed") ou normaux (Figure 1.21). Les étages de pipelines sont soit en mode de fonctionnement normal soit en mode regroupé. Cette nouvelle génération de latch a été utilisée dans AMULET3 [FURB-00].

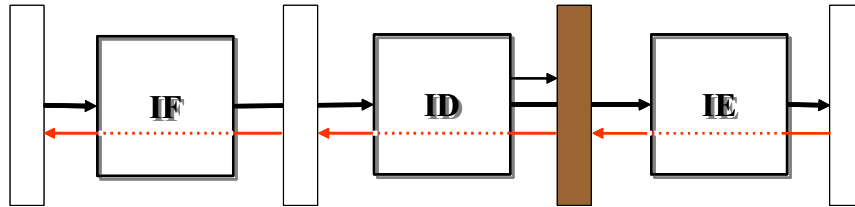


Figure 1.21 Pipeline utilisant un latch reconfigurable

1.2.2.2 Structure de contrôle

Quand le chemin de donnée est non linéaire, c'est-à-dire qu'il présente des structures de contrôle (multiplexeur), la complexité du circuit est plus grande car plusieurs entrées peuvent se propager sur une seule sortie. Ces architectures ont une consommation d'énergie élevée si aucune attention n'est prêtée lors de leurs implémentations. Des stratégies sont proposées pour réduire la consommation d'énergie de ces composants en déséquilibrant la structure de choix en faveur des cas qui présentent les plus grandes probabilités [TIER-96]. Les probabilités sont établies par simulation et dépendent directement de l'application. Les multiplexeurs de grande taille sont ainsi décomposés en une succession de multiplexeurs de plus petite taille permettant par conséquent aux cas présentant les plus grandes probabilités de traverser un ensemble restreint de multiplexeur de plus petite taille (Figure 1.22).

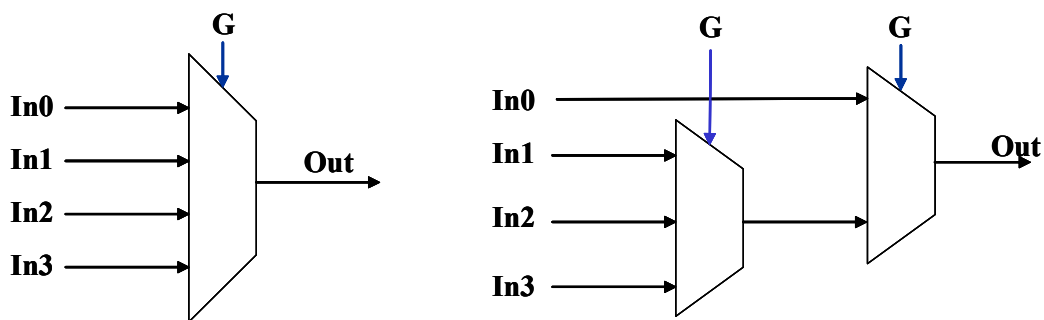


Figure 1.22 Multiplexeur régulier et décomposé

La figure montre que l'entrée In0 est privilégiée dans le second cas puisqu'elle ne traverse qu'un multiplexeur à deux entrées au lieu d'un multiplexeur à quatre entrées. Il y a donc un gain en consommation d'énergie à chaque exécution de cette entrée. Sur l'ensemble d'une simulation, si celle-ci présente la plus forte probabilité, l'économie faite sur la consommation d'énergie peut être significatif.

La même décomposition peut être réalisée pour des démultiplexeurs.

1.2.2.3 Optimisation des communications

Nous présentons trois optimisations possibles des communications:

- Communication 2 phases

L'expérience de la conception de circuits asynchrones a montré que le protocole de communication 4 phases est plus simple et beaucoup plus performant que le protocole 2 phases. Il est logiquement le plus utilisé dans les circuits asynchrones actuels. Cependant, des travaux ont montré que pour des communications entre composants éloignés, il est possible de réduire la consommation d'énergie en utilisant un protocole de communication 2 phases [APPL-97]. Le protocole de communication 2 phases permet de réduire d'un facteur deux le nombre de commutations des fils de communication. Il est intéressant d'implémenter seulement la communication en utilisant un protocole 2 phases, les composants du circuit conservent leurs implémentations 4 phases. Des interfaces 2 phases vers 4 phases, et 4 phases vers 2 phases sont nécessaires pour adapter les protocoles de communication. Ce mixage des communications permet d'apporter un gain en consommation d'énergie lorsqu'un circuit présente de nombreux composants communicants.

- Codage 1 parmi 4

Le passage du codage dual rail (1 parmi 2) au codage 1 parmi 4 permet de réduire le nombre de commutations d'un facteur deux. Le codage 1 parmi 4 est utilisé dans l'implémentation des bus de communication [BAIN-01]. Récemment, l'utilisation du codage 1 parmi 4 a montré des intérêts en terme de consommation d'énergie dans l'implémentation d'opérateurs arithmétiques et logiques [FRAG-03]

- Longueur adaptable

Il est aussi possible de réduire la consommation d'énergie des communications en adaptant dynamiquement la longueur des données lors d'une communication. Une dissipation superflue d'énergie est observée lorsque la taille des données est figée dès l'implémentation. Il existe certains cas où la donnée n'est pas utilisée dans son ensemble, certains bits seulement sont utilisés par des unités. Les travaux qui ont été réalisés par [MANO-01] permettent d'émettre d'une unité à une autre les bits indispensables uniquement. Il y a une sélection des bits à émettre depuis l'unité source (sélection des bits de poids forts ou faibles par exemple), les bits non utilisés ne sont pas émis, économisant ainsi de l'énergie.

1.2.2.4 Paramètre $E\tau^2$

Dans le but de comparer différentes implémentations d'un même circuit, il a été nécessaire de développer une métrique qui combine le temps de cycle et la consommation d'énergie : ce paramètre est $E\tau^2$ [MART-00]. Ce paramètre permet de trouver le meilleur compromis entre le temps d'exécution et la consommation d'énergie de manière à obtenir la performance la meilleure. Le terme E de la consommation d'énergie ne représente que l'énergie dynamique de charge et de décharge des capacités, les énergies dues au courant de fuite et au court-circuit sont supposées nulles. L'énergie vaut $E = C * V^2$ où C représente toutes les capacités de charge durant l'exécution. Le délai d dans les transistors et dans les fils est inversement proportionnel à la tension d'alimentation: $d = \frac{C}{k * V}$. Le temps de cycle est la somme des

délais de toutes les transitions du chemin critique, le temps de cycle τ vaut $\tau = \frac{K}{V}$.

Etant donné que l'énergie E et le temps de cycle t sont corrélés et sont tributaires de la tension d'alimentation, il est impossible de dissocier l'ensemble. Il est donc nécessaire de combiner

l'énergie et le temps de cycle et d'obtenir une seule équation. Dans un premier temps, la métrique $E * \tau$ a été utilisée pour comparer des circuits, mais il s'est avéré qu'il ne s'agissait pas d'une relation pertinente puisque cette métrique dépendait directement de la tension d'alimentation V .

$$E * \tau = (C * V^2) * \left(\frac{K}{V}\right) = K * C * V$$

Il n'était donc pas possible de comparer deux implémentations différentes sans faire abstraction de la tension d'alimentation. Une meilleure métrique est le produit $E * \tau^2$ qui permet d'avoir une relation indépendante de la tension d'alimentation dans une première approximation.

$$E * \tau^2 = (C * V^2) * \left(\frac{K}{V}\right)^2 = K^2 * C$$

Une équation de la sorte permet de trouver le compromis entre l'énergie E et le temps de cycle τ sans connaissance de la tension d'alimentation. Lorsque le circuit est réalisé en considérant ce paramètre, la tension d'alimentation peut varier pour faire de l'adaptation dynamique de la tension d'alimentation par exemple.

Certains circuits ont été réalisés à Caltech en utilisant cette métrique, notamment le CAP [MART-89] et le MiniMIPS [MART-97]. Le but était de minimiser autant que possible cette métrique de manière à obtenir la meilleure performance possible.

1.2.2.5 Techniques de réduction au niveau système

Il existe des techniques logicielles intéressantes qui participent à la réduction de la consommation des circuits intégrés. Lorsqu'on couple ces techniques logicielles avec les techniques matérielles, la consommation d'énergie du circuit sur l'ensemble d'une exécution est significativement optimisée. Nous présentons quelques techniques logicielles, gérées au niveau du système d'exploitation, qui permettent de contribuer à la réduction de la consommation d'énergie.

- Mode endormi

Le mode endormi correspond à l'état d'un circuit où aucune activité n'est enregistrée. La consommation d'énergie est ainsi réduite à un minimum absolu. Les systèmes synchrones utilisent le mode endormi pour réduire la consommation d'énergie lorsque le circuit ne fait aucun traitement pendant un temps donné. Cependant, il est bien connu que les circuits synchrones nécessitent un temps relativement long associé à une dissipation lorsque le système entre en mode endormi ou lorsqu'il se réveille. Cet inconvénient majeur nécessite d'utiliser le mode endormi à bon escient. Des algorithmes élaborés sont utilisés par le système d'exploitation pour passer du mode normal au mode endormi, et vice versa. En revanche, les circuits asynchrones sont intrinsèquement adaptés au mode endormi. Ceci est lié aux propriétés de la logique asynchrone qui présente une activité seulement lors d'un travail effectif. Les parties du circuit qui ne participent pas à la tâche sont naturellement au repos (donc en mode endormi). Aucun algorithme n'est nécessaire, le système d'exploitation est par conséquent dispensé de cette tâche. De plus, le mode endormi est visible à un niveau de granularité très fin. L'exemple d'une application vidéo sur le processeur ASPRO [RENA-98]

montre que le mode endormi est présent à chaque fois qu'une partie du circuit ne fonctionne pas [ESSA-02].

- Adaptation dynamique de la tension d'alimentation (DVS)

Contrairement au mode endormi, l'adaptation dynamique de la tension d'alimentation [ESSA-02] permet de répartir dans le temps l'exécution d'un programme. Au lieu, d'avoir un mode normal hyper actif suivi d'un mode endormi où l'activité est réduite à un minimum, il est souvent préférable d'utiliser différentes tensions d'alimentation de manière à étaler l'exécution d'un programme dans le temps. Cela permet de réduire la consommation d'énergie aux dépens de la vitesse d'exécution. La vitesse d'un circuit est directement proportionnelle à la tension d'alimentation, elle est d'autant plus grande que la tension d'alimentation est importante.

Il existe une différence de comportement entre l'adaptation dynamique en logique synchrone et en logique asynchrone. En synchrone, la variation de la vitesse est tributaire de la tension d'alimentation et de la fréquence de l'horloge. Par conséquent, on a besoin de deux étapes : dans un premier temps il faut réduire la fréquence de l'horloge puis dans un second temps diminuer la tension d'alimentation. Dans le cas où on repasse à la tension d'alimentation initiale qui est supérieure, il faut dans un premier temps augmenter la tension d'alimentation et dans un second augmenter la fréquence de l'horloge. Le passage d'un cas à l'autre, et vice et versa, nécessite un temps d'adaptation qui correspond aux paramètres du convertisseur DC-DC et de la PLL. La fréquence des circuits synchrones ne varie pas de manière continue mais par pas. Le processeur par conséquent stoppe les traitements en cours jusqu'à ce que la fréquence de l'horloge se stabilise à la fréquence à atteindre. En revanche, pour le cas asynchrone les tâches du circuit ne sont pas stoppées pendant la variation de la tension d'alimentation. De plus, la vitesse varie de manière continue et est finement ajustée pendant l'exécution de la tâche [FESQ-04].

1.2.3 Autres techniques

Les recherches sur la réduction de la consommation d'énergie ne cessent de se faire. De nombreuses autres techniques développées à travers le monde existent, celles-ci sont présentées succinctement dans ce paragraphe :

- Réduction des glitches

Les circuits asynchrones doivent être dépourvus de glitch pour un fonctionnement correct du circuit. Cependant, certaines techniques de synthèse garantissent l'absence de glitch sur les sorties principales mais ne dispensent pas de glitches sur certains signaux intermédiaires. La synthèse burst-mode est de cette nature. Les travaux réalisés permettent de limiter le nombre de glitch des signaux intermédiaires afin de réduire la consommation d'énergie [NOWI-97].

- Librairie à faible consommation d'énergie

L'utilisation de bibliothèques dédiées à la faible consommation d'énergie est d'un intérêt certain dans la conception de circuits à faible consommation d'énergie. Les bibliothèques "full-custom" sont privilégiées par rapport aux bibliothèques de cellules standard qui consomment davantage. La bibliothèque de "handshake component"[BERK-95] de l'outil de synthèse TANGRAM, a été faite de manière à réduire autant que possible la consommation de ces cellules spéciales. Des bibliothèques telles que présentées dans [PIGU-98] ont été réalisées spécialement pour la faible consommation d'énergie. Enfin, plus récemment la bibliothèque TAL [MAUR-03] de l'outil TAST

[TAST-02] a été réalisée pour réduire la taille et la consommation d'énergie des circuits asynchrones Quasi Insensible aux Délais.

- Dimensionnement des transistors

Enfin, à un niveau d'abstraction très bas, à savoir au niveau transistor, on peut noter des travaux qui permettent de redimensionner la taille des transistors de manière à optimiser la consommation d'énergie [PENZ-02b].

1.3 Les Microprocesseurs asynchrones

De nombreuses recherches sont faites sur la conception de circuits asynchrones, des méthodologies ont ainsi été développées pour concevoir des circuits asynchrones. Ces quinze dernières années ont vu la réalisation de nombreux microprocesseurs asynchrones qui montrent clairement la faisabilité de circuits asynchrones de taille importante. Une étude exhaustive faite sur les microprocesseurs asynchrones est présentée dans ce paragraphe.

1.3.1 Caltech Asynchronous Processeur (CAP)

CAP [MAR-89] a été réalisé à la fin des années 80 par le groupe d'Alain Martin à California Institute of Technology, il s'agit du premier microprocesseur entièrement implémenté en logique asynchrone (Figure 1.23). Le but d'une telle réalisation était d'expérimenter la conception de circuits asynchrones et de tester les méthodes de transformation de programmes développées à Caltech [MART-90b]. Initialement, il s'agissait non pas d'un processeur mais de huit processus concurrents et indépendants qui ont été synthétisés séparément en circuits. C'est seulement à partir de ce moment que les modules ont été assemblés pour former un processeur. Le processeur est du type RISC de 16 bits avec un pipeline à 3 étages. Il intègre 4 bus, une unité arithmétique et logique (ALU), 2 additionneurs et un banc de registres. Il existe deux versions : une première version implémentée en technologie MOSIS SCMOS 2 μ m qui intègre un banc de 12 registres de 16 bits, et une seconde version en technologie MOSIS SCMOS 1.6 μ m intégrant un banc de 16 registres de 16 bits. Le processeur présente une architecture de Harvard où les mémoires données et instructions sont séparées. Le processeur implémente ni mécanismes d'interruption ni ports de communication. CAP implémente une architecture double rails utilisant un protocole de communication 4 phases. A l'exception des interfaces mémoires, le circuit est du type insensible aux délais (DI). Le circuit a été synthétisé en utilisant une librairie de cellules standard (portes complexes statiques), il contient 20000 transistors (aucune optimisation électrique n'a été faite sur le dimensionnement des transistors).

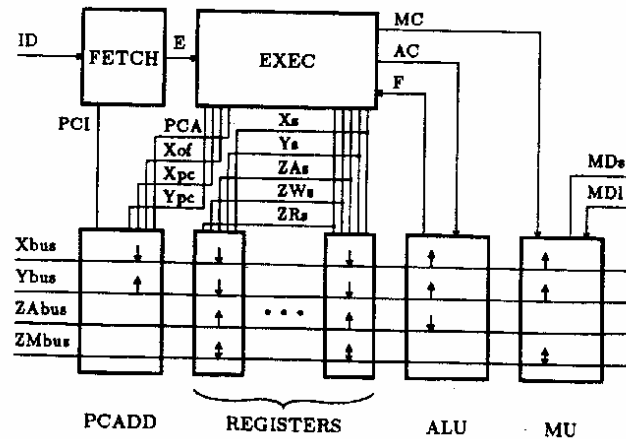


Figure 1.23 Architecture du CAP

Le circuit a été simulé fonctionnellement utilisant COSMOS et les performances obtenues sont, pour la version 1.6 μ m, de 26 MIPS pour une consommation de 1.5W à 10V; 18 MIPS pour une consommation de 225mW à 5V et 5 MIPS pour une consommation de 10.4mW à 2V.

1.3.2 Philips 80C51 asynchrone

Le microcontrôleur 80C51 asynchrone [GAGE-98] a été réalisé à Eindhoven University of Technology en collaboration avec Philips Research Laboratories en 1995. Le circuit a été implémenté en un temps relativement court de 6 mois. Le microcontrôleur 80C51 (CPU et périphériques) a une architecture du type CISC contenant un jeu de 255 instructions et supportant différents modes d'adressage. Le jeu d'instructions peut être décomposé en 5 classes : arithmétique, logique, transfert de données, booléens et branchements. La version asynchrone est complètement compatible avec le processeur original. Le but de cette duplication était de montrer les bénéfices de l'implémentation asynchrone en comparant directement la version originale synchrone et la version asynchrone. Le circuit a été spécifié en langage de haut niveau à l'aide de Tangram [BERK-91], et compilé en une netlist à l'aide de l'outil de compilation Tangram. La compilation d'un programme Tangram en une netlist se fait en 2 étapes dans lesquels les handshakes circuits représentent la forme intermédiaire. Les circuits handshake sont implémentés utilisant le protocole 4 phases et une architecture du type simple rail. La transparence du compilateur Tangram permet d'apporter des optimisations au niveau de la spécification Tangram. De plus, une combinaison adaptée de structure de bus et de communication point à point contribue à réduire la consommation d'énergie du processeur. Il intègre aussi un convertisseur DC/DC pour l'adaptation dynamique de la tension d'alimentation (DVS). Le processeur a été conçu en technologie CMOS 0.5 μ m utilisant des cellules standard. L'architecture du processeur est proposée Figure 1.24.

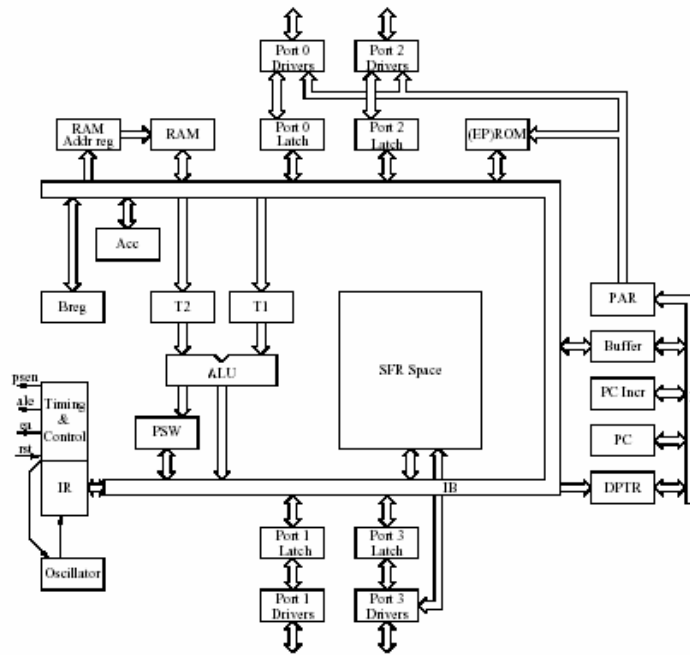


Figure 1.24 Architecture du 80C51 asynchrone

Les performances obtenues pour une tension d'alimentation de 3.3V sont dressées au Tableau 1-2.

Version	MIPS	Power (mW)	MIPS/W
Sync 80C51	4	40	100
Async 80C51 (V1)	4	9	444

Tableau 1-2 Performances du 80C51 asynchrone

La consommation d'énergie a été réduite d'un facteur 4 par rapport à la version originale synchrone. Cela prouve les bénéfices que la logique asynchrone apporte en terme de consommation d'énergie. Une réduction des émissions électromagnétiques a aussi été observée. L'inconvénient rencontré lors de l'implémentation de la version asynchrone se situe au niveau de la surface occupée, en effet celle-ci est 2 fois supérieure à l'originale. Ce surcoût en surface est attribué au contrôle des circuits asynchrones qui est plus complexe que celui des circuits synchrones. Le layout du circuit est montré à la Figure 1.25.

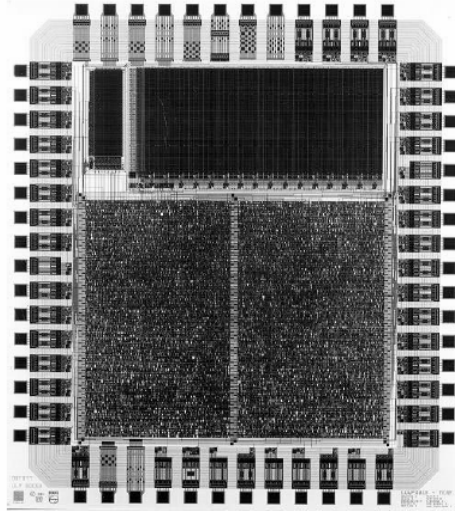


Figure 1.25 Layout du 80C51 asynchrone

1.3.3 AMULET

AMULET (Asynchronous Microprocessor Using Low Energy and Technology) est un projet né à la fin de l'année 1990 à l'université de Manchester. Il avait pour but de montrer le potentiel à réduire la consommation d'énergie grâce à l'utilisation de la logique asynchrone. Le but était de développer un microprocesseur ARM (Advanced RISC Machine) totalement asynchrone; cette œuvre devait démontrer la faisabilité à appliquer les techniques de conception asynchrone à un processeur commercial.

La première version réalisée fut le processeur AMULET1 [FURB-94] au début de l'année 1993 ; la réalisation de ce processeur a nécessité 5 personnes*an. Bien que le circuit soit complètement fonctionnel, il doit être considéré comme une première tentative à l'implémentation d'un processeur commercial utilisant les techniques de conception asynchrone. AMULET1 est une réplique asynchrone du processeur ARM6 RISC 32 bits de la compagnie ARM Limited. L'architecture globale du processeur asynchrone est illustrée à la Figure 1.26. Le style de conception asynchrone utilisé dans AMULET1 est basé sur la technique de conception micropipeline élaborée par Sutherland [SUTH-89]. AMULET1 utilise un protocole de communication 2 phases entre les différentes unités.

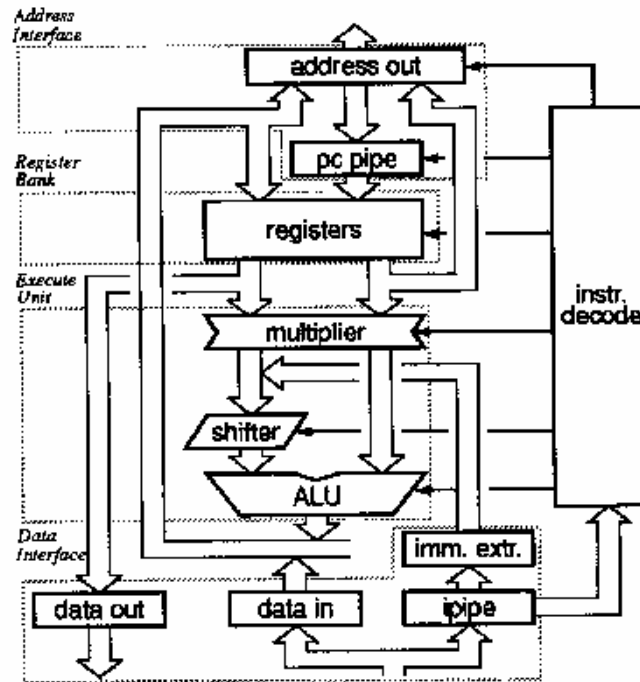


Figure 1.26 Architecture de AMULET1

Les performances de AMULET1 sont présentées dans le Tableau 1-3.

	AMULET1	ARM6
Process	1 μ m DLM CMOS	1 μ m DLM CMOS
Aire	5.5mm x 4.1mm	4.1mm x 2.7mm
Nombre de transistors	58 374	33 494
Performance	9 K dhrystones	14K dhrystones @ 10 MHz
Power	83mW	75mW @ 10 MHz

Tableau 1-3 Performances de AMULET1 et de ARM6

Les caractéristiques enregistrées par AMULET1 montrent qu'il présente deux fois plus de transistors que l'ARM6 original, qu'il est aussi moins performant et consomme plus d'énergie. Les résultats obtenus ne reflètent pas ce qui était initialement prévu en terme de consommation d'énergie et de performance. En effet, on s'attendait à obtenir une consommation moindre pour une performance plus élevée que le processeur ARM6 synchrone. AMULET1 a un pipeline beaucoup plus profond que le processeur ARM6, d'où un nombre de transistors plus élevé, on devrait cependant observer un débit plus important que l'ARM6 original. Malheureusement, il fut tout autre, ceci peut être attribué au décodeur d'instructions qui ne peut pas émettre des instructions à une vitesse qui permettrait de garder le pipeline rempli. De plus, les dépendances de données entre les successions d'instructions causent des blocages fréquents (le compilateur n'est pas optimisé pour AMULET1). De plus, l'utilisation d'un protocole 2 phases qui devait en théorie réduire la consommation d'énergie voit en pratique une augmentation de la consommation due à l'utilisation d'une logique plus complexe pour la détection des 2 fronts.

Une seconde version AMULET2e [FURB-98] a été réalisée afin de remédier aux problèmes rencontrés dans AMULET1 et de faire évoluer l'architecture en intégrant entre autre un système de cache. AMULET2e implémente un cœur AMULET2 avec 4Kbytes de mémoire, qui peut être configuré comme mémoire cache où comme RAM. Un « Jump Trace Buffer » est incorporé dans l'interface d'adresse d'AMULET2 pour la prédiction de branchements.

L'architecture intègre un mécanisme de blocage de registres sous forme de FIFO et un mécanisme de forwarding (bypass register). Pour réduire la dissipation d'énergie, le registre à décalage et le multiplieur sont contournés quand ils ne sont pas nécessaires. De plus, un mécanisme "halt" permet de stopper toute activité dans le circuit. Contrairement au processeur AMULET1, AMULET2 utilise un protocole de communication 4 phases qui présente l'avantage d'être plus rapide et moins consommant. L'architecture est, comme pour AMULET1, du type micropipeline. L'architecture de AMULET2e est proposée à la Figure 1.27.

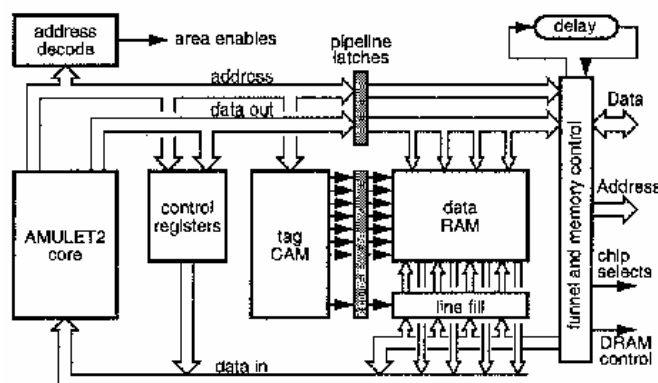


Figure 1.27 Architecture de AMULET2e

Le circuit a été implémenté en technologie CMOS 0.5µm, il utilise 454 000 transistors dont 93 000 dans le cœur du processeur pour une taille de 6.4mm². Il est empaqueté dans un boîtier de 128 pins et utilise une tension d'alimentation de 3.3V. Les performances sont illustrées au Tableau 1-4, elles sont comparées aux processeurs synchrones ARM710 et ARM810.

	ARM710	AMULET2e	ARM810
Process	0.6µm 2LM	0.5µm 3LM	0.5µm 3LM
Surface mm ²	32	41	76
Transistors	570 295	454 000	836 022
MIPS	23	40	86
Power mW	120	150	500
MIPS/W	192	250	172

Tableau 1-4 Performances du processeur AMULET2e

La troisième version AMULET3 [FURB-00] est un processeur RISC asynchrone de 32 bits qui est complètement compatible avec le cœur ARM9TDMI synchrone. Il représente le sommet d'une recherche de 10 ans sur le développement de microprocesseurs asynchrones à l'université de Manchester. C'est la première tentative pour une application commerciale. Il présente une architecture de Harvard : un port est utilisé pour la mémoire d'instruction et un second pour la mémoire donnée. Il supporte l'architecture ARM version 4T, incluant le jeu d'instructions 16 bits compressé Thumb. La possibilité d'étendre le code d'instruction Thumb localement est exploitée par des instructions multi-cycles (multi-registres PUSH/POP). Une instruction qui ne produit pas de résultats est supprimée sans traverser tout le pipeline. Une innovation architecturale de AMULET3 est l'implémentation d'un mécanisme de "register forwarding" (registre qui permet de stocker temporairement les résultats avant de les envoyer au banc de registres). AMULET3 a montré que la technologie asynchrone est commercialement viable, elle est compétitive en terme de performance, surface et

consommation d'énergie par rapport aux circuits synchrones. Le chemin de donnée utilise un layout full custom fait à la main, la librairie standard a été fournie par ARM Limited. Une partie du contrôle a été synthétisée à l'aide de Petrify [CORT-96]. AMULET3 utilise des contrôleurs de latches dynamiques pour permettre de varier la profondeur du pipeline pendant une exécution [LEWI-99]. Le circuit a été simulé à l'aide du simulateur LARD [ENDE-98] développé à l'université de Manchester. Le cœur AMULET3 compte 113 000 transistors, il occupe une surface de 3mm² pour une technologie de 0.35μm et une tension VDD de 3.3V. Il est capable d'exécuter jusqu'à 120 MIPS (Dhrystone 2.1), le débit maximal consomme environ 155mW, soit 780 MIPS/W. Ces performances sont très similaires au cœur ARM9TDMI synchrone. La Figure 1.28 montre l'architecture du cœur AMULET3.

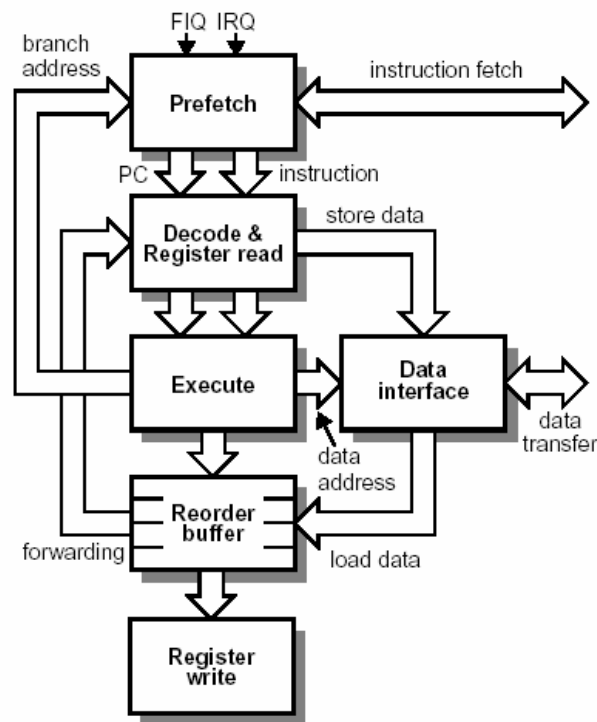


Figure 1.28 Architecture de AMULET3

Le cœur AMULET3 est intégré dans AMULET3i [GARS-00] avec un bus asynchrone interne MARBLE [BAIN-98], un contrôleur DMA (synthétisé à l'aide de Balsa [BARD-02]) et des mémoires RAM et ROM. Le premier AMULET3i est planifié pour une production commerciale au début de l'année 2000. Sa première application est pour le système de contrôle DRACO (Dect Radio COmmunication controller) qui est un appareil de communication sans fil (Figure 1.29).

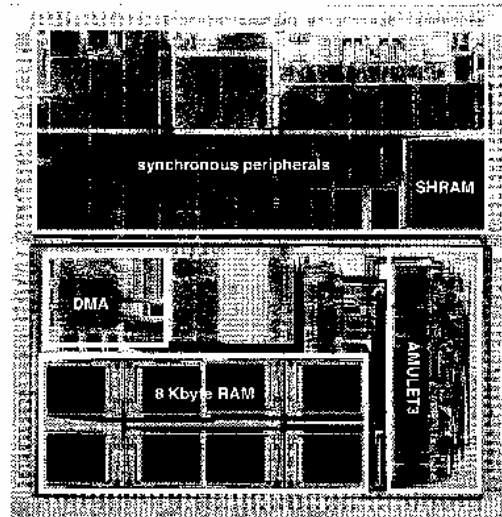


Figure 1.29 Système de contrôle DRACO

1.3.4 ASPRO

ASPRO [RENA-98] est un microprocesseur asynchrone de 16 bits à 3 étages de pipeline développé par l'ENST de Bretagne en collaboration avec France Telecom-CNET Grenoble et STMicroelectronics en 1998. L'équipe a depuis rejoint TIMA en 1999. C'est un microprocesseur RISC régulier (pas de registre de statut, pas de jeu d'instructions réduits), il contient 16 registres de 16 bits. Il s'agit d'un processeur scalaire qui décode les instructions dans l'ordre et les exécute dans le désordre. Le flot de conception et le style de circuit sont une application originale de la méthode d'Alain Martin [MART-90b]. La motivation était de montrer que les techniques asynchrones peuvent améliorer efficacement les systèmes VLSI et peuvent permettre d'optimiser la vitesse et la consommation d'énergie. En plus, cette réalisation met en évidence les propriétés bénéfiques de l'asynchrone telles que la modularité et la réutilisabilité. Il a été décidé de ne pas dupliquer une version synchrone existante dans le but de ne pas être trop contraint, ainsi il a été possible d'exploiter autant que possible les avantages de la conception asynchrone. ASPRO a été implémenté en une architecture multi-rails quasi insensible aux délais (QDI) en utilisant un protocole de communication 4 phases. La synthèse a été partiellement faite à la main et mappée sur une librairie de cellules standard fournie par le fondeur STMicroelectronics. Le processeur présente 2 mémoires distinctes sur la puce, une pour le programme et l'autre pour les données. La mémoire programme interne a une largeur de 24 bits. Un convertisseur 4 phases double rails vers 4 phases micropipeline est nécessaire pour accéder aux mémoires synchrones.

Le port parallèle A permet d'accéder à la mémoire centrale, il est implémenté en micropipeline et utilise un protocole de communication 4 phases. Le double rail n'est pas utilisé au niveau de l'interface afin de minimiser le nombre de pins et pour une compatibilité avec les mémoires standard. Les ports séries Nx, Ex, Wx et Sx sont implémentés en double rails associé à un protocole 2 phases, ce type d'implémentation a été adopté afin d'augmenter la vitesse et la robustesse des communications.

Le microprocesseur a été fabriqué en technologie CMOS 0.25 μm de CNET/SGS-Thomson. Le processeur, sans les mémoires, contient 400 000 transistors ; la surface est de 4mm². L'architecture de ASPRO est proposée à la Figure 1.30.

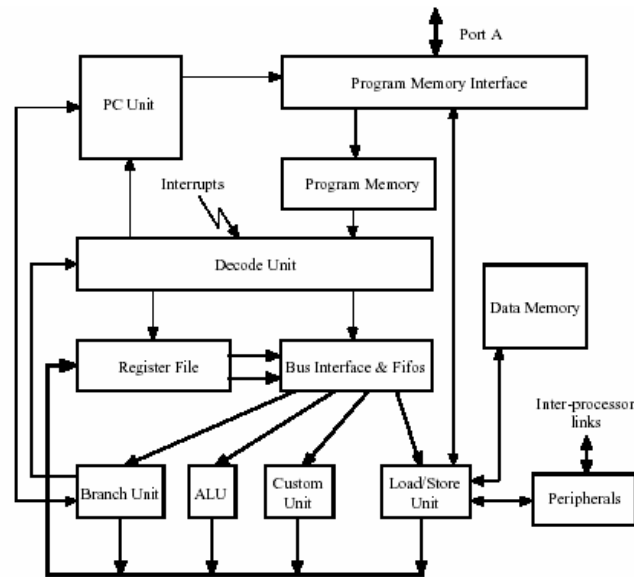


Figure 1.30 Architecture d'ASPRO

Les performances sont montrées au Tableau 1-5.

Tension alimentation	Power	MIPS	Mw/MIPS
VDD = 1V	27mW	24	1.1
VDD = 2.5V	500mW	140	3.6

Tableau 1-5 Performances du processeur ASPRO

Le CPU présente 3 sources d'alimentations indépendantes qui peuvent varier de 0.9V à 3.0V. Cela permet au processeur d'ajuster dynamiquement sa tension d'alimentation en fonction des besoins de l'application. Le circuit physique final est proposé à la Figure 1.31.

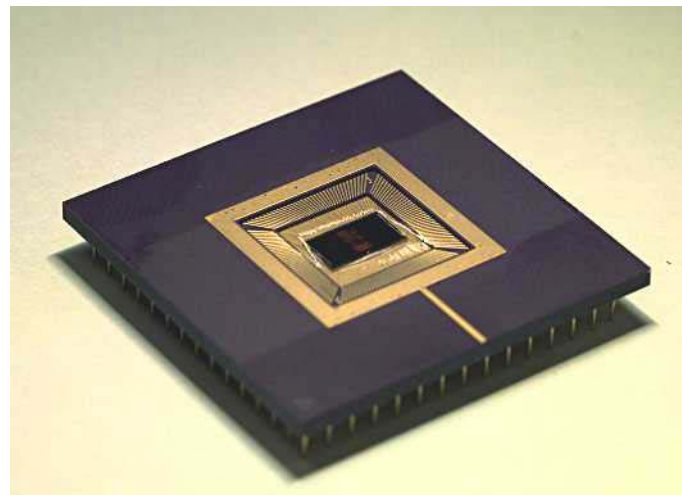


Figure 1.31 ASPRO

1.3.5 MICA

MICA est un microcontrôleur asynchrone CISC de 8 bits à 3 étages de pipeline réalisé par le groupe CIS du laboratoire TIMA en collaboration avec France Telecom R&D et STMicroelectronics en 2000 [ABRI-01]. Il intègre 2 différents bancs de registres: 8 registres de donnée de 8 bits et 8 registres de 16 bits d'adresse (incluant le compteur de programme PC

et le pointeur de pile). Des unités spécifiques sont associées à chaque banc de registres favorisant ainsi le calcul en parallèle des données et des adresses. Une unité périphérique est intégrée, elle contient 6 ports parallèles de 8 bits et 4 ports séries (implémenté à l'aide d'un protocole 2 phases insensible aux délais et compatible avec le processeur ASPRO [RENA-98]). De plus, le microcontrôleur intègre 16 Ko de RAM et 2 Ko de ROM. La mémoire ROM inclut un BIST ("*Built-In-Self-Test*") qui est exécuté à l'initialisation en fonction du mode sélectionné (8 modes sont disponibles).

Le cœur du microcontrôleur a été implémenté en logique QDI (codage n-rail) associé au protocole 4 phases. L'architecture de MICA a été implémentée sous forme d'un système distribué. Cette conception permet de limiter les actions de communications et de réduire la complexité de la machine à états principale en décomposant celle-ci. Le but était de réduire autant que possible la consommation d'énergie. La consommation d'énergie des communications a été réduite en utilisant un certain nombre de codages allant de 5 rails à 12 rails (1 parmi 5 et 1 parmi 12) pour les canaux de contrôle. Cela pour réduire le nombre de commutation des fils par action de communication (donc la consommation dynamique). Les canaux de donnée sont implémentés à l'aide d'un codage 1 parmi 4; une ALU et un banc de registres en base 4 ont été réalisés pour convenir aux communications 1 parmi 4 du chemin de donnée. Les performances enregistrées sont illustrées au Tableau 1-6. MICA est utilisée dans une carte à puce qui s'appelle MICABI [ABRI-01].

Alimentation	MIPS	power	MIPS/Watt
1V	4.3	0.8mW	5503.6
1.5V	11.9	4.7mW	2560.2
2V	18.6	13.3mW	1398.0
2.5V	23.8	28.0mW	850.3
3V	27.8	48.9mW	568.1
3.5V	31.3	77.0mW	405.8

Tableau 1-6 Consommation d'énergie de MICA

1.3.6 TITAC

TITAC [NANY-94] est un microprocesseur asynchrone de 8 bits conçu à Tokyo Institute of Technology. Il s'agit d'un processeur à un seul accumulateur. Il est caractérisé par un accès mémoire selon une architecture de von Neumann. Ce processeur a été implémenté en 2 versions asynchrones et une version synchrone afin de pouvoir comparer leurs performances. TITAC n'implémente aucun mécanisme d'interruption nécessitant l'utilisation d'arbitre ou de synchroniseur. Il n'y a pas de bus interne. La version asynchrone est basée sur un modèle d'implémentation QDI. Il présente un protocole 4 phases, une implémentation double rails et des portes complexes de plusieurs niveaux AND-OR avec une structure BDD pour la génération efficace de signaux. Les signaux de contrôle utilisent un codage M parmi N où $M < N$. On a recourt à un convertisseur 2 rails vers 1 rail et 1 rail vers 2 rails pour l'accès à la mémoire RAM. L'architecture de TITAC est illustrée Figure 1.32.

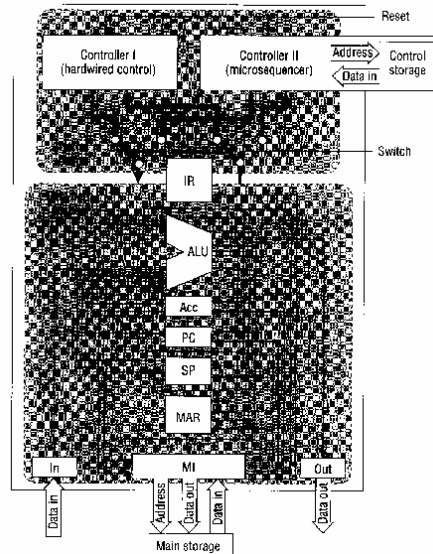


Figure 1.32 Architecture du processeur TITAC

TITAC a été implémenté en technologie 1.0µm. Il existe 2 versions asynchrones : une version implémentant un type de contrôleur ("hardwired control") et une seconde version implémentant un autre type de contrôleur ("microprogrammed control"). Les performances pour une tension d'alimentation de 5V sont présentées au Tableau 1-7. La version asynchrone présente 2.8 fois plus de portes que son homologue synchrone.

	TITAC asynchrone (hardwired control)	TITAC asynchrone (microprogrammed control)
Performance	11.2 MIPS	1.8 MIPS
Power	212mW	70mW

Tableau 1-7 Performances du processeur TITAC

La réalisation d'un processeur asynchrone TITAC-2 [TAKA-97] a été faite par ce groupe de recherche en 1997. Il s'agit d'une version asynchrone du processeur MIPS R2000. La reproduction de ce processeur n'est pas complètement fidèle puisque certaines instructions n'ont pas été implémentées et d'autres instructions ont été modifiées. Le processeur est du type RISC 32 bits à 5 étages de pipeline. Le processeur intègre une mémoire cache interne, il manipule les exceptions et les interruptions externes. TITAC-2 est implémenté à l'aide d'une architecture SDI ("Scalable-Delay-Insensitive"). Il s'agit d'un modèle de délai non borné, aucune hypothèse sur les bornes supérieures des fils et des portes n'est faite. Les bénéfices de ce modèle sont visibles en terme de vitesse. Les circuits SDI sont plus rapides que les circuits QDI. Les données sont codées en double rails et la communication est réalisée à l'aide d'un protocole 4 phases. La mémoire cache interne et l'interface du bus externe utilisent une implémentation micropipeline. L'organisation de TITAC-2 est proposée à la Figure 1.33.

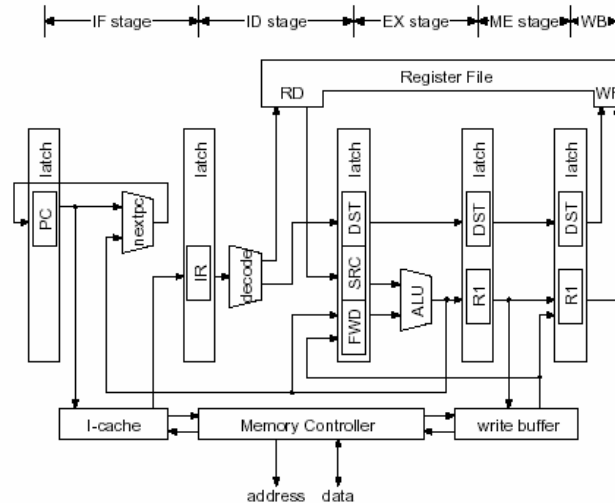


Figure 1.33 Architecture du processeur TITAC-2

TITAC-2 a été synthétisé en utilisant des cellules standard et a été fabriqué en technologie $0.5\mu\text{m}$. Il présente 496 367 transistors MOS, 8.6Kbytes de mémoire cache. Le tout a été intégré sur une surface de $12.15\text{mm} \times 12.15\text{mm}$. Les performances obtenues sont de 52.3 VAX MIPS pour un test dhrystone V2.1 et consomme 2.11W à 3.3V. Pour une tension d'alimentation variant de 1.5V à 6.0V, la consommation d'énergie est montrée sur la Figure 1.34. La Figure 1.35 propose le layout du circuit.

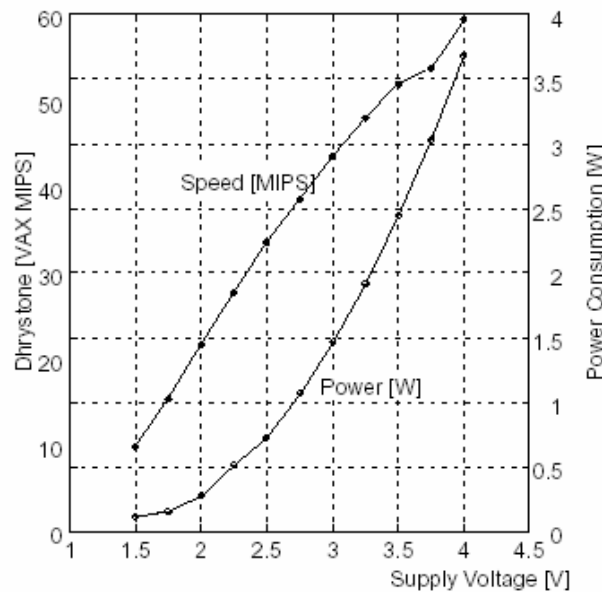


Figure 1.34 Performances de TITAC-2 en fonction de la tension d'alimentation

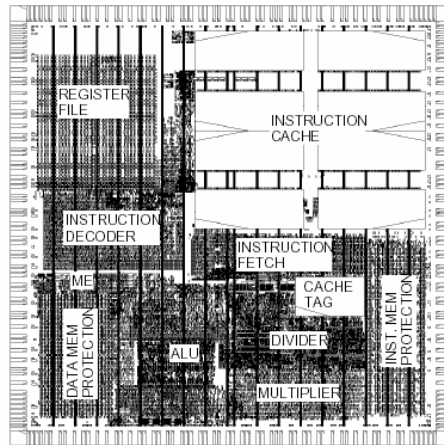


Figure 1.35 Layout de TITAC-2

1.3.7 MiniMips

Le processeur MiniMIPS [MART-97] a été réalisé par California Institute of Technology. Il s'agit d'une implémentation asynchrone du microprocesseur MIPS R3000. Le but de la réalisation de ce processeur était dans un premier temps de minimiser le paramètre $E\tau^2$ [MART-00] indépendamment de la tension d'alimentation VDD; une fois l'obtention d'un $E\tau^2$ minimal, il s'agissait d'exploiter la robustesse des circuits asynchrones en fonction de la variation de la tension d'alimentation VDD. On voulait observer les bénéfices d'une telle implémentation sur la vitesse à haute tension VDD, et la faible consommation d'énergie à faible VDD. Le MIPS R3000 se présente sous forme de 2 coprocesseurs sur une seule puce : un CPU RISC 32 bits et une MMU (CP0). Le CPU présente 32 registres de 32 bits, un pc et deux registres spéciaux pour la multiplication et la division. Le premier prototype asynchrone est une architecture simple scalaire (une instruction est décodée à la fois). MiniMIPS final est très finement pipeliné afin d'avoir un débit important. La structure du MIPS se présente comme 3 étages séquentiels : calcul de l'adresse pc, fetch l'instruction en mémoire, exécute l'instruction. MiniMIPS implémente un mécanisme de bypass des résultats pour améliorer les performances. L'implémentation des exceptions est l'une des principales innovations de ce projet. Cependant, le traitement des exceptions en asynchrone est beaucoup plus délicat qu'en synchrone. Le circuit utilise une architecture QDI (codage 1 parmi N) et les canaux sont implémentés en protocole 4 phases. Des optimisations ont été réalisées dans le mécanisme d'acquittement du chemin de donnée. Le chemin de donnée est décomposé en petites unités (largeur de 4 bits), chaque unité génère son propre signal d'acquittement. L'arbre de d'acquittement est plus petit, réduisant ainsi les délais et la consommation d'énergie du processeur. Le nombre de transistors dans le contrôle et le chemin de donnée est de 250K, pour les caches il est de 1.25M. L'architecture de MiniMIPS est proposée Figure 1.36.

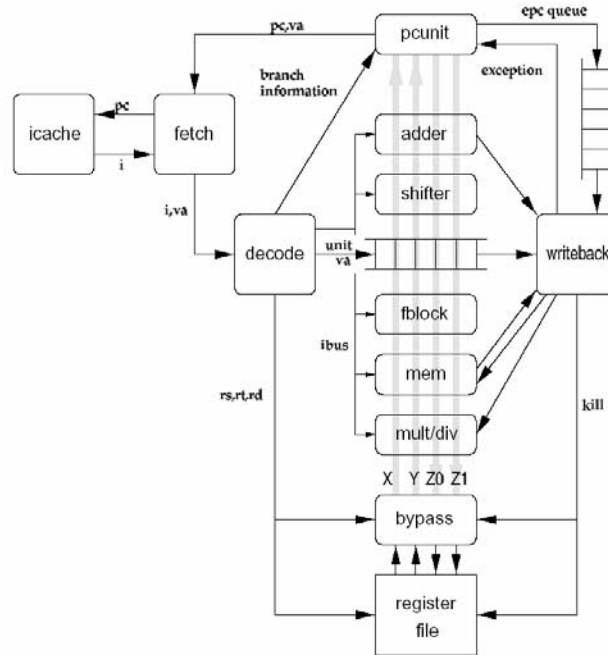


Figure 1.36 Architecture du MiniMIPS

Le processeur a été fabriqué en technologie 0.6µm MOSIS SCMOS, les performances sont montrées au Tableau 1-8.

Tension d'alimentation	performance	Consommation
VDD = 1.5V	100 MIPS	0.5W
VDD = 2 V	150 MIPS	1W
VDD = 3.3V	280 MIPS	7W

Tableau 1-8 Performances du processeur MiniMIPS

MiniMIPS est un processeur très innovant pour l'année où il a été réalisé.

1.3.8 Asynchronous TinyRISC (ARISC)

Asynchronous TinyRISC [CHRI-98] est l'implémentation asynchrone du TinyRISC TR4101. Il a été fait pour un effort d'un homme*an pendant une période de 5 mois dans un projet conjoint entre Technical University of Denmark et LSI Logic Denmark. (Utilisation des CAD-tools et des bibliothèques standard). Le TR4101 implémente les instructions MIPS-II et MIPS16, c'est un processeur RISC 32 bits. Le TR4101 n'inclut pas de MMU, de caches ou d'unité de multiplication/Division (mais peuvent être ajoutés par l'utilisateur sous forme de macro cellules). Il a 3 étages de pipeline. L'implémentation du ARISC a été faite en faveur de la faible consommation d'énergie et de la simplicité architecturale au dépens de la performance (sans trop la compromettre). Cette simplification a conduit à une architecture présentant un certain nombre d'unités d'exécution et une forme restreinte d'exécution dans le désordre. ARISC n'implémente pas de MMU ni de traitement des exceptions (même si les exceptions sont signalées dans le circuit). L'accès mémoire implémente une architecture de Harvard où les ports mémoires sont séparés pour l'accès aux données et aux instructions. Quand ARISC travaille en mode MIPS16, 2 instructions sont décodées en même temps (bande passante plus grande). De plus, on optimise la consommation d'énergie en invoquant le PC et le cache d'instructions une seule fois pour 2 instructions. Ce processeur utilise une

architecture du type micropipeline et un protocole 4 phases. Il est mappé sur des cellules standard statiques. La spécification du processeur a été faite à l'aide de Verilog et les outils de synthèse de Synopsys. Ces outils ont été utilisés pour décrire et synthétiser la logique du chemin de données. La logique de contrôle asynchrone SI (Speed Independent) a été synthétisée partiellement à la main et partiellement à l'aide de l'outil Petrify. L'architecture du ARISC est présentée à la Figure 1.37.

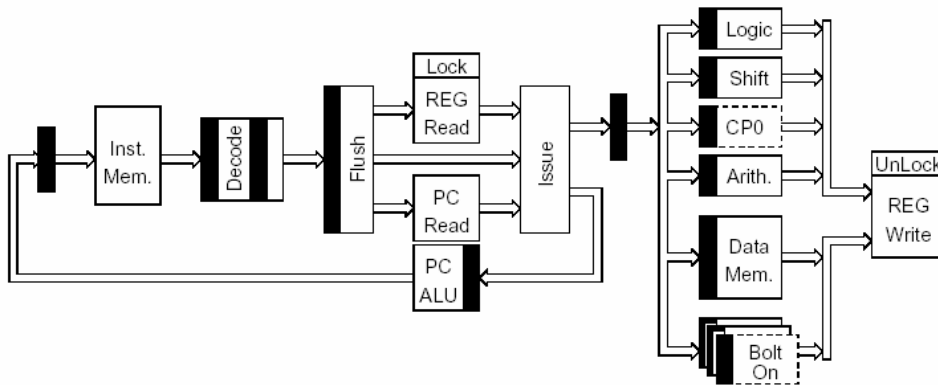


Figure 1.37 Architecture du processeur Asynchrone TinyRISC

La technologie CMOS 0.35 μ m est utilisée, la surface du cœur ARISC est de 2.2mm² (le banc de registre couvre 0.4 mm²). Les performances observées sont de 74-123 MIPS en fonction des instructions exécutées. A 74 MIPS et pour une tension d'alimentation de 3.3V (architecture avec des caches asynchrones ne nécessitant aucune synchronisation), la consommation d'énergie est de 117mW soit 635 MIPS pour un watt. Les performances obtenues par le TR4101 synchrone original fabriqué en technologie CMOS 0.35 μ m de LSI Logic et fonctionnant à une tension VDD de 3.3V, sont de 81 MHz et de 62 MIPS. En fonction du benchmark exécuté, les performances du ARISC sont 50-90 % meilleures que le TR4101. Le rapport MIPS/Watt du ARISC est 17 % meilleur que le TR4101. L'activité du ARISC est inférieure à celle du TR4101 grâce à la propriété asynchrone. Le layout du circuit est proposé Figure 1.38.

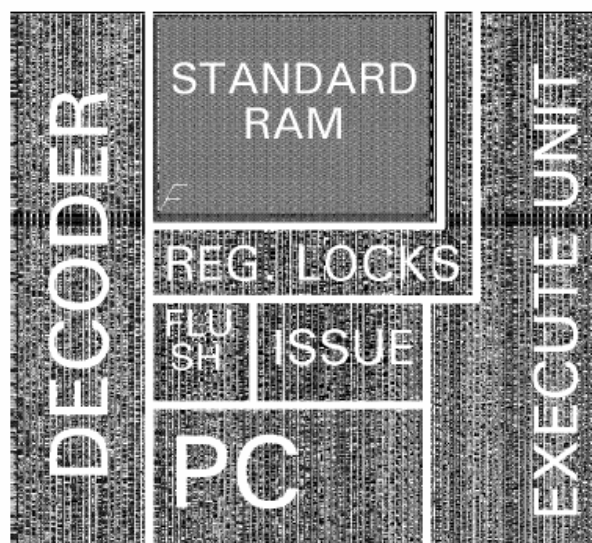


Figure 1.38 Layout du processeur Asynchrone TinyRISC

1.3.9 Lutonium

Lutonium [MART-03] est un microcontrôleur 8051 asynchrone développé par California Institute of Technology en 2003. Ce tout récent microcontrôleur a été implémenté pour un paramètre $E\tau^2$ optimal. Il est difficile de justifier le choix du 8051 sur la base de la faible consommation d'énergie sachant que ce dernier présente un jeu d'instructions complexe et irrégulier. La seule véritable raison est que le 8051 est un microcontrôleur très populaire que l'on trouve dans de nombreuses applications dédiées à la faible consommation d'énergie. Lutonium est hautement pipeliné pour augmenter la vitesse. Aucune prédiction de branchement n'est réalisée. Le décodeur d'instructions est beaucoup plus complexe que celui du MiniMIPS, il est implémenté dans une seule unité afin de réduire la taille et le nombre de communications des canaux dans le circuit. Afin de ne pas augmenter la consommation d'énergie, rien n'est calculé ou manipulé en dehors d'un traitement effectif. Lutonium implémente une architecture QDI; les circuits implémentés antérieurement ont montré l'efficacité et la robustesse des circuits QDI à de faibles tensions d'alimentation. La Figure 1.39 montre l'architecture de Lutonium.

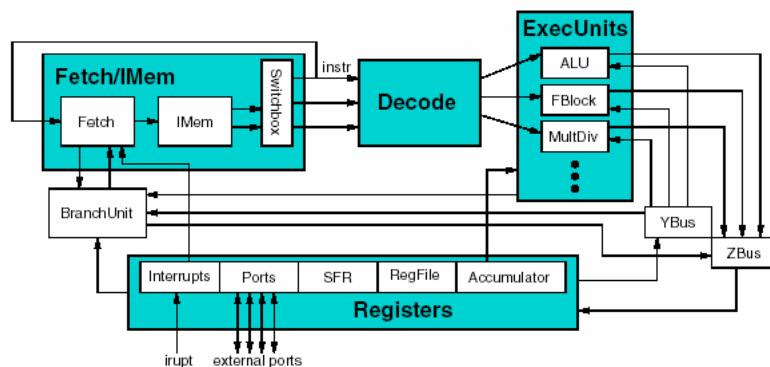


Figure 1.39 Architecture de Lutonium

Lutonium est fabriqué en technologie TSMC SCN018 offert par MOSIS : CMOS 0.18 μ m pour une tension d'alimentation VDD de 1.8V. Les performances enregistrées sont montrées sur le Tableau 1-9.

1.8V	200 MIPS	100.0mW	500pJ/inst	1800 MIPS/W
1.1V	100 MIPS	20.7mW	207pJ/inst	4830 MIPS/W
0.9V	66 MIPS	9.2mW	139pJ/inst	7200 MIPS/W
0.8V	48 MIPS	4.4mW	92pJ/inst	10900 MIPS/W
0.5V	4 MIPS	170 μ W	43pJ/inst	23000 MIPS/W

Tableau 1-9 Performances de Lutonium

La répartition de la consommation d'énergie dans Lutonium est illustrée à la Figure 1.40.

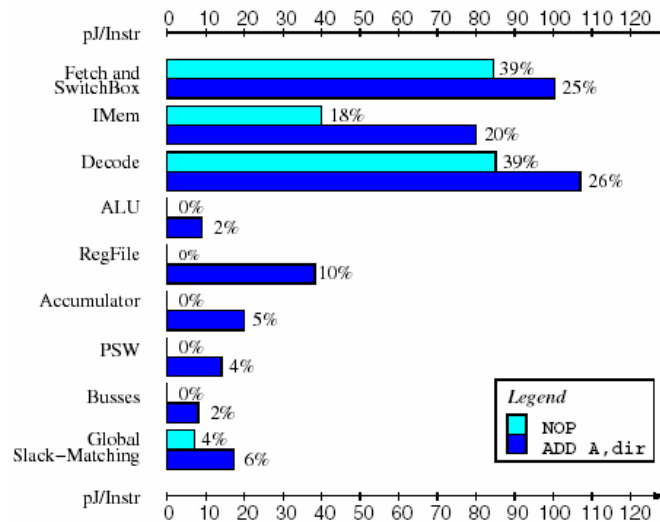


Figure 1.40 Répartition de la consommation d'énergie dans Lutonium

1.3.10 Autres Microprocesseurs asynchrones

Il existe d'autres microprocesseurs asynchrones. Nous en présentons quelques uns:

- SPA

SPA [PLAN-02] est un cœur de processeur ARM asynchrone entièrement synthétisé à l'aide de l'outil de synthèse Balsa [BARD-02] par l'université de Manchester. Balsa a permis de réaliser ce processeur dans un délai relativement court et nécessitant peu de personnes. SPA est un microprocesseur capable de résister aux attaques basées sur l'analyse de la consommation d'énergie (DPA) et de l'analyse électromagnétique. Il est implémenté en double rails associé à un protocole 4 phases. L'architecture est présentée à la Figure 1.41.

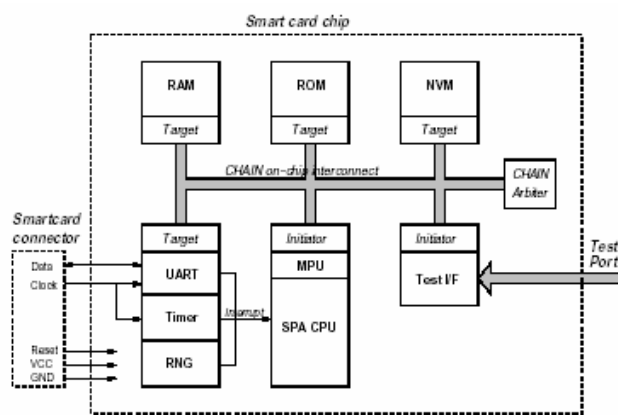


Figure 1.41 Microprocesseur SPA

- FAM : Fully Asynchronous Microprocessor

FAM [CHO-92] a été développé par Korean Institute of Science and Technology et Tokyo Institute of Technology. C'est un processeur RISC 32 bits asynchrone qui présente 4 étages de pipeline. FAM, comme CAP, a été réalisé dans un but purement expérimental. Il présente un banc de 32 registres de 32 bits, une ALU de 32 bits, un multiplieur et un registre à décalage de 32 bits. Il intègre seulement 18 instructions. FAM a été implémenté en dual rail et

utilise un protocole de communication 4 phases. Le développement de FAM a été basé sur la logique 2-AND au lieu de portes de muller. Ce style d'implémentation utilise au plus 2 transistors en série pour mettre à 0 ou à 1 les bascules. Les avantages de cette implémentation par rapport à la porte de muller sont la vitesse et la compacité. FAM a été fabriqué en technologie CMOS 0.5 μ m, il contient 71 000 transistors pour une surface de 6.4mmx4.9mm. Les performances sont de 300 MIPS.

- SCALP

SCALP [ENDE-96] est un microprocesseur superscalaire asynchrone développé à l'université de Manchester. Il prend 5 instructions en mémoire à la fois et les envoie dans le désordre à 5 unités parallèles. Il a été observé que le pipeline exploite moins le potentiel de l'implémentation asynchrone que le parallélisme superscalaire. En effet, pour une structure implémentant un pipeline asynchrone, la latence est déterminée par la somme des délais. Toutes les données doivent passer à travers tous les blocs incluant le bloc le plus lent, par conséquent le débit est toujours limité par le bloc le plus lent. Pour une structure superscalaire, une opération lente dans un bloc peut être dépassée par une opération rapide dans un autre bloc. La latence et le débit sont donc déterminés par une performance typique. Dans SCALP, les signaux de contrôle sont implémentés en DI, et les macromodules sont implémentés en SI.

- NSR : NonSynchronous RISC

NSR [BRUN-93] a été développé à l'université d'Utah, c'est principalement un assemblage de blocs asynchrones (5 au total). Il a été mis au point dans un but expérimental. C'est un processeur RISC asynchrone de 16 bits, contenant 16 registres de 16 bits. Il intègre seulement 16 instructions. NSR est implémenté utilisant une architecture micropipeline et un protocole de communication 2 phases. Ce processeur a été intégré dans un FPGA, les performances observées sont de 1.3 MIPS pour la meilleure configuration de la simulation.

- CFPP : CounterFlow Pipeline Processor

CFPP [SPRO-94] est un processeur RISC développé en 1994 à Sun Microsystems. Le nom est dérivé de la caractéristique principale de son architecture, il utilise un pipeline bi-directionnel dans lequel les instructions vont dans un sens et les résultats dans le sens opposé. Le PC et le banc de registres sont placés aux extrémités opposées du pipeline.

- Strip : Self-Timed RISC Processor

Strip [DEAN-92] est basé sur le processeur synchrone MIPS-X, il présente 5 étages de pipeline. Ce processeur est implémenté en micropipeline, les interfaces externes utilisent une architecture double rails et un protocole de communication 4 phases. Cette implémentation permet une communication efficace entre des modules de vitesses différentes. Les performances obtenues à partir d'une simulation Spice sont de 62.5 MIPS pour une technologie CMOS 2 μ m.

- ASYNMPU

ASYNMPU [TSE-97] est un microprocesseur asynchrone de 8/16 bits développé par Hong Kong Polytech. L'objet de ce travail est de montrer les avantages de l'implémentation

architecturale asynchrone pour une petite application faible consommation d'énergie. Il s'agit du premier microprocesseur CISC implémenté utilisant les techniques de conception asynchrone. C'est un processeur qui manipule des longueurs d'instruction variables. Ce microprocesseur est compatible avec le microprocesseur Intel 8/16 bits. Il peut remplacer celui-ci dans une application grâce à une synchronisation adaptée.

- MSL16A

MSL16A [TSAN-99] est un processeur asynchrone de 16 bits développé par The Chinese University of Hong Kong. Il s'agit du clone du processeur synchrone MSL16 (Minimal instruction, Small and Low power 16-bit microprocessor) qui permet d'exécuter le langage Forth. Le jeu d'instructions est petit et simple, les instructions sont codées sur 4 bits. Les chemins de donnée et de contrôle sont très simples, et la densité de code est importante. L'implémentation asynchrone du MSL16 a été réalisée dans le but de montrer les avantages de la technique de conception asynchrone en terme de consommation d'énergie. MSL16 est une machine à double pile contenant des données codées sur 16 bits et des bus mémoires. Il emploie un pipeline à 2 étages (Fetch/Execute). La vitesse d'exécution du MSL16A est rapide grâce à son jeu d'instructions simple et son chemin critique court. Un fetch en mémoire permet de récupérer 4 instructions de 4 bits en un seul coup. La logique de contrôle asynchrone a été implémentée en utilisant la méthode de transformation de Caltech [MART-90b]; les composants du chemin de donnée ont été conçus à la main. Le circuit asynchrone est du type quasi insensible aux délais. Les données sont codées en double rail et utilise un protocole de communication 4 phases. Des convertisseurs double rail vers un rail et un rail vers double rail sont nécessaires pour interfacer avec le monde extérieur. La technologie est du type CMOS AMI 1.2 μ m, la surface est de 4.335mmx4.671mm.

Le système a été simulé en utilisant HSPICE pour une tension d'alimentation de 5V; les performances sont de l'ordre de 33 MIPS pour une consommation de 200mW à pleine vitesse. Le layout du circuit est montré à la Figure 1.42.

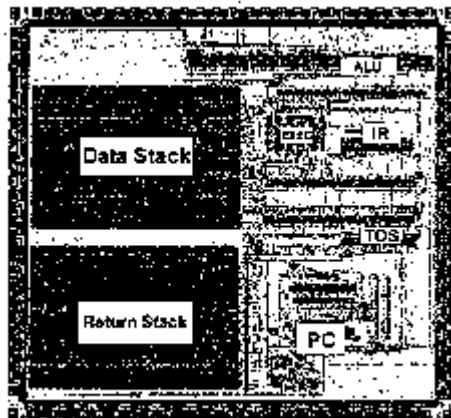


Figure 1.42 Layout du processeur MSL16A

- DINAMIK

DINAMIK [CHOI-99] (Delay INSensitive Asynchronous Microprocessor In K-JIST) est un microprocesseur asynchrone de 16 bits réalisé par Kwang-Ju Institute of Science and Technology (K-JIST). L'implémentation a été faite en VHDL, les outils de conception ont été COMPASS ASIC Synthesizer pour la synthèse et Gate-level Simulator pour la simulation. La

synthèse a été faite par SYNOPSIS en utilisant la librairie standard CMOS KG75000 0.6µm de Samsung Electronics. Avec CADENCE, VerilogXL a été utilisé pour la simulation. L'architecture utilisée pour la conception de ce processeur a été le codage double rail et le protocole de communication 4 phases. La surface est de 5mmx5mm et le nombre de transistors est de 52 168. DINAMIK fonctionne correctement pour une tension d'alimentation variant de 2.0V à 8.0V. Les performances sont illustrées au Tableau 1-10.

VDD (V)	MIPS	W(mW)	MIPS/W
2.0	5	60	83.33
3.0	8	180	44.44
4.0	10	440	22.73
5.0	11.76	750	15.68
6.0	13.33	1200	11.11
7.0	13.33	1820	7.32
8.0	13.33	2480	5.37

Tableau 1-10 Performances du processeur DINAMIK

1.3.11 Récapitulatif des microprocesseurs asynchrones

Le Tableau 1-11 résume les caractéristiques et les performances de tous ces processeurs.

Processeur asynchrone	Version synchrone	Type	architecture	VDD	Techno (µm)	W(mW)	MIPS	Taille
CAP	-	RISC 16 bits	DI (DR) 4 phases	5	1.6	-	16	20K tr
Philips 80C51 asynchrone	80C51	CISC	BD 4 phases	3.3	0.5	40	4	-
AMULET1	ARM6	RISC 32 bits	BD 2 phases		1.0	83		22.55 mm ²
AMULET2	ARM7	RISC 32 bits	BD 4 phases	3.3	0.5	150	40	6.4 mm ²
AMULET3	ARM9	RISC 32 bits	BD 4 phases	3.3	0.35	155	120	3.0 mm ²
ASPRO	-	RISC 16 bits	QDI (DR) 4 phases	2.5	0.25	500	140	4.0 mm ²
TITAC	-	8 bits	QDI (DR) 4 phases	5	1.0	70	1.8	-
TITAC-2	MIPS R2000	RISC 32 bits	SDI (DR) 4 phases	3.3	0.5	2100	52.3	147 mm ²
MiniMIPS	MIPS R3000	RISC 32 bits	QDI (DR) 4 phases	3.3	0.6	7000	280	250K tr
Asynchronous TinyRISC	TR4101	RISC 32 bits	BD 4 phases	3.3	0.35	117	74	2.2 mm ²
Lutonium	80C51	CISC	QDI	1.8	0.18	100	200	-

			4 phases					
MICA	-	8 bits	QDI (MR4) 4 phases	-	-	-	-	-
SPA	ARM9	RISC 32 bits	DR 4 phases	-	-	-	6	737K tr
FAM	-	RISC 32 bits	DR 4 phases	3.3	0.5	-	300	31.36 mm ²
SCALP	-	-	DI et SI	-	-	-	-	-
NSR	-	RISC 16 bits	BD 2 phases	-	-	-	1.3	-
CFPP	-	-	-	-	-	-	-	-
Strip	MIPS-X	-	BD 4 phases	-	2.0	-	62.5	-
ASYNMPU	Intel 8/16 bits	CISC 8/16 bits		-	-	-	-	-
MSL16A	Forth MSL16	16 bits	QDI (DR) 4 phases	5	1.2	200	33	20.22 mm ²
DINAMIK	-	16 bits	DR 4 phases	3	0.6	180	8	25 mm ²

Tableau 1-11 Microprocesseurs asynchrones

1.4 Conclusion

La logique asynchrone se dresse aujourd'hui comme une solution intéressante contre les nombreux problèmes que rencontre la logique utilisant une synchronisation globale. Les nombreux avantages que celle-ci apporte en terme de modularité, de faible émission électromagnétique et de faible consommation d'énergie sont des plus appréciables. De plus, la diversité d'architectures et de protocoles de communication permet de réaliser un certain nombre de circuits très performants. Nous utilisons la logique asynchrone pour sa propriété de réduction de la consommation d'énergie plus précisément. Nous souhaitons associer des techniques d'estimation et d'optimisations de la consommation d'énergie afin de concevoir un microprocesseur à faible consommation d'énergie. Une méthodologie d'estimation de la consommation d'énergie est particulièrement importante pour observer le comportement d'un circuit. Cette méthodologie est l'objet du prochain chapitre.

Chapitre 2

Estimation de la Consommation d'Énergie d'un Programme CHP

L'état de l'art du Chapitre 1 a présenté la conception asynchrone et a mis en évidence les bénéfices intéressants que celle-ci apporte en terme de modularité, de faible émission électromagnétique, de vitesse moyenne et enfin de faible consommation d'énergie. La réduction de la consommation d'énergie est la propriété de la conception asynchrone qui nous intéresse particulièrement dans notre quête de conception de microprocesseurs et circuits au sens large. Le premier pas vers la réduction de la consommation d'énergie est le passage à la logique asynchrone. On souhaite tout de même réduire davantage la consommation d'énergie. Pour ce faire, dans le cadre de la thèse nous avons spécifié une méthodologie qui permet d'estimer la consommation d'énergie des circuits asynchrones Quasi Insensible aux Délais (QDI). La consommation d'énergie s'exprime en terme de nombre de commutations des portes. Cette estimation présente l'avantage de fournir au concepteur de circuits asynchrones des informations sur la consommation d'énergie d'un circuit très tôt dans le flot de conception. Le concepteur a ainsi un regard très avisé sur la répartition de la consommation d'énergie du circuit qu'il conçoit. Ces informations lui permettent d'identifier les parties du circuit qui consomment le plus et qui nécessitent une optimisation. Il est ainsi mieux armé pour optimiser judicieusement le circuit.

Le premier paragraphe introduit l'organisation générale de la méthodologie d'estimation de la consommation d'énergie, et précise comment cette méthodologie s'intègre dans le flot de synthèse des circuits asynchrones Quasi Insensible aux Délais (QDI). Une description succincte du flot de synthèse est faite par la même occasion au paragraphe 2.1. L'estimation est faite en 2 étapes: dans un premier temps une estimation dite structurelle est élaborée, cette estimation est présentée au paragraphe 2.2; puis une estimation dite dynamique est faite, celle-ci est expliquée au paragraphe 2.3. Une application que nous nommons hétérogène est proposée au paragraphe 2.4 et enfin une conclusion vient clôturer ce chapitre sur la formalisation de la consommation d'énergie des circuits asynchrones.

2.1 Méthodologie d'estimation de la consommation d'énergie

La méthodologie d'estimation de la consommation d'énergie en terme de nombre de commutations de portes que nous proposons dans ce chapitre s'intègre dans l'outil TAST [TAST-02] (Tima Asynchronous Synthesis Tools). TAST est un flot de conception de circuits numériques asynchrones développé au sein du groupe de recherche CIS du laboratoire TIMA. La méthodologie d'estimation de la consommation d'énergie s'inscrit plus précisément dans la partie synthèse des circuits asynchrones Quasi Insensibles aux Délais du flot TAST (Figure 2.1). Cette méthodologie est indissociable des techniques de synthèse que nous utilisons (elle n'est pas portable sur d'autres méthodologies de conception asynchrone).

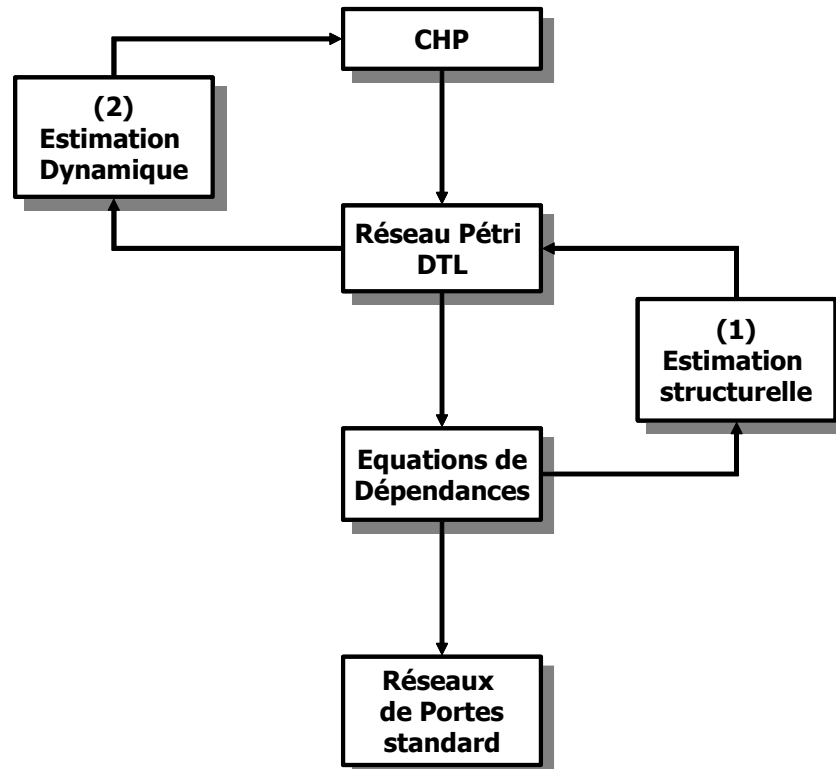


Figure 2.1 Estimation de la consommation d'énergie dans le flot de synthèse TAST

La synthèse des circuits Quasi Insensible aux Délais se réalise en quatre étapes principales. Nous exposons brièvement ces étapes :

- Spécification CHP

Les circuits asynchrones que nous concevons à TIMA sont spécifiés à l'aide d'un langage de conception matérielle de haut niveau nommé CHP qui signifie "Concurrent Hardware Processes". Ce langage que nous utilisons et développons est basé sur le langage CHP original né à Caltech [MART-90a] et sur le langage commercial VHDL. CHP est un langage de conception asynchrone, il intègre par conséquent des aspects propres à la conception asynchrone: utilisation de canaux, communication par passage de messages entre processus grâce aux canaux, type multi-rail, codage insensible aux délais.

Les nouveautés qui sont apportées au langage CHP original sont une plus grande efficacité, une plus grande lisibilité et une plus grande puissance puisque les avantages du langage commercial VHDL y sont intégrés (modélisation structurelle hiérarchique) (cf. Annexe B pour la syntaxe et la sémantique CHP).

- Réseau de Pétri DTL

La première étape de la synthèse des circuits asynchrones est la compilation de la spécification CHP de départ en une représentation intermédiaire standard sous forme d'un réseau de pétri. La forme intermédiaire obtenue lors de la compilation est une forme restreinte qu'on nomme réseau de Pétri DTL ("Data Transfer level"). La forme DTL pour les circuits asynchrones est de même nature que la forme RTL ("Register Transfer Level") pour les circuits synchrones. Il s'agit d'une restriction d'écriture de la spécification CHP, cette restriction se résume à : une spécification est dite DTL si et seulement si elle est composée de

processus combinatoires et de processus ayant des initialisations de canal de communication. Ces règles DTL permettent de limiter l'utilisation de variable d'état mémorisante [DINH-02b].

- Les équations de dépendance

Le réseau de Pétri DTL obtenu est ensuite parcouru et des équations de dépendances sont établies [DINH-02b]. Une équation de dépendance permet de formaliser le fonctionnement d'un circuit. Elle exprime les dépendances des sorties en fonction des entrées du circuit et contient absolument toutes les informations qui permettent aux sorties de commuter. L'expression générale d'une équation de dépendance est présentée au paragraphe 2.2.

- Réseau de portes standard

Enfin, les équations de dépendance sont compilées en une netlist de portes. Il s'agit d'une netlist de même type que celle obtenue par les outils de synthèses de la conception synchrone. Néanmoins, la netlist que nous obtenons intègre des portes utilisées dans la conception asynchrone (muller par exemple). Ces portes asynchrones font parties d'une librairie de portes asynchrones développée à TIMA qui se nomme TAL (Tima Asynchronous Library) [MAUR-03] (cf. Annexe A).

A ce flot principal de synthèse des circuits Quasi Insensible aux Délais viennent se greffer les 2 étapes d'estimation de la consommation de notre processus d'estimation de la consommation d'énergie des circuits asynchrones QDI :

- L'estimation structurelle

L'estimation structurelle est la première étape de l'estimation de la consommation d'énergie. Elle intervient au niveau des équations de dépendance du flot de synthèse asynchrone. L'estimation de la consommation d'énergie utilise les équations de dépendance et permet d'obtenir une équation booléenne par processus. Les spécificités de l'estimation structurelle de la consommation d'énergie sont expliquées au paragraphe 2.2.

- L'estimation dynamique

L'estimation dynamique est la seconde étape de l'estimation de la consommation d'énergie. Elle se greffe au niveau de la simulation du réseau de pétri et utilise l'estimation structurelle. Les détails concernant la méthodologie de l'estimation dynamique sont présentés au paragraphe 2.3.

2.2 Estimation de la consommation d'énergie structurelle

L'estimation structurelle consiste à établir les consommations en terme de nombre de commutations de portes en fonction de la structure du circuit décrit en langage CHP, d'où le nom d'estimation structurelle. La forme de départ est l'ensemble des équations de dépendances, le résultat produit est une équation booléenne qui recense le nombre de transitions d'un processus. Cette équation intègre des opérateurs booléens lorsque le processus présente des structures de choix.

La forme générale de l'équation de dépendance (Équation 2.1) permet de décrire tout type de processus d'un circuit asynchrone.

Équation 2.1

$$\text{process} = F_{\text{dep}} \left(\sum_{i=0}^{I-1} \text{In}(i); \sum_{q=0}^{Q-1} G(q); \sum_{j=0}^{J-1} \text{Out}(j) \right) = F_S \left(\sum_{i=0}^{I-1} \text{In}(i) \right)$$

- F_{dep} : fonction de dépendance (exprime les dépendances des sorties en fonction des entrées d'un processus)
- $\text{In}(i)$: les canaux d'entrées (In pour Input)
- $G(q)$: les gardes des structures de choix
- $\text{Out}(j)$: les canaux de sorties (Out pour Output)
- F_S : fonction de calcul des sorties

Pour résumer, on peut dire que cette équation établit les dépendances de toutes les sorties du processus $\left(\sum_{j=0}^{J-1} \text{Out}(j) \right)$ en fonction de toutes les entrées $\left(\sum_{i=0}^{I-1} \text{In}(i) \right)$ du processus considéré. Les processus peuvent présenter des structures de choix avec un certain nombre de gardes $\left(\sum_{q=0}^{Q-1} G(q) \right)$. Cette équation de dépendance est la forme la plus générale, elle regroupe absolument toutes les formes pouvant être décrite en CHP DTL. On verra par la suite que les structures de base telles que le latch, la fourche ("fork"), la jonction ("join"), le multiplexeur ("mux") et le démultiplexeur ("demux") sont facilement modélisées par cette équation. Le réseau de pétri de cette forme générale est représenté à la Figure 2.2:

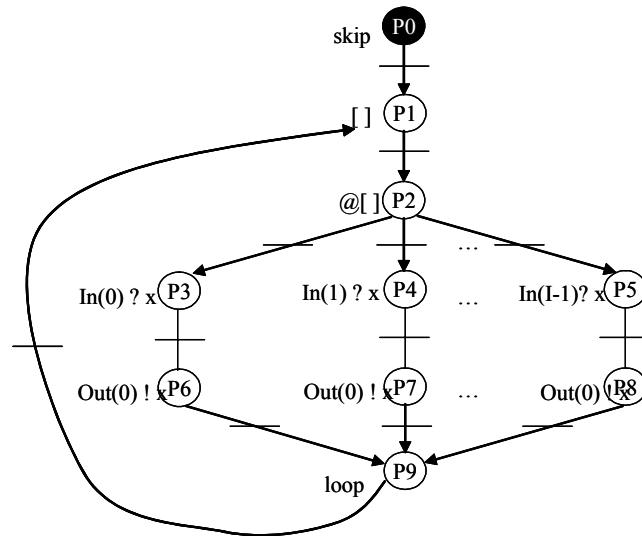


Figure 2.2 Réseau de pétri de la forme générale

Nous allons établir le coût en terme de nombre de commutations de cette structure très générale, et les coûts pour les structures plus simples seront déduits très logiquement. Le coût d'un circuit asynchrone décrit à l'aide d'équations de dépendance du type Équation 2.1 peut être écrit à l'aide d'une somme (Équation 2.2):

Équation 2.2

$$\text{Coût}_{\text{structurel}} \{ \text{process} \} = \sum_{q=0}^{Q-1} G(q) \cdot \left[\text{Coût}_{\text{structurel}} \{ F_{\text{dep}}(G(q)) \} \right]$$

Cette équation se présente comme une somme de coûts structurels des commandes gardées. Il s'agit plus précisément d'une somme qui intègre des opérateurs booléens. On verra par la suite, notamment à la section 2.3 sur l'estimation dynamique, que lors d'une exécution donnée, seule la garde qui est exécutée contribue à la consommation globale du processus (propriété d'exclusion mutuelle entre les gardes dans les circuits asynchrones). L'opérateur booléen permet de ne comptabiliser que le coût de la commande gardée qui est exécutée.

Le coût structurel correspond donc à une seule et unique équation booléenne par processus. Cette équation contient les coûts de toutes les branches du processus (si celui-ci présente des structures de choix).

L'exemple ci-après illustre le pseudo code CHP d'un processus présentant une structure avec un choix à trois branches:

```

Process P
begin
[
@[ G0 => I0
    G1 => I1
    G2 => I2
]
]
end process ;
    
```

Où G0, G1, G2 sont les gardes de la structure de choix et I0, I1, I2 sont les instructions associées à ces gardes.

Le coût structurel du processus P s'écrit :

$$\text{Coût}_{\text{structurel}}\{P\} = G(0) \bullet \text{Coût}_{\text{structurel}}\{F_{\text{dep}}(G(0))\} + G(1) \bullet \text{Coût}_{\text{structurel}}\{F_{\text{dep}}(G(1))\} + G(2) \bullet \text{Coût}_{\text{structurel}}\{F_{\text{dep}}(G(2))\}$$

Si la garde G(0) est exécutée par exemple, les produits scalaires des autres gardes sont nuls, par conséquent leurs coûts ne sont pas comptabilisés dans le coût structurel du processus lors de cette exécution. En utilisant ce principe, il nous suffit d'établir une seule équation de coût structurel par processus. Par sa nature booléenne, celle-ci recense toutes les informations concernant le coût des gardes. Pour obtenir cette équation finale du coût structurel du processus, il est indispensable d'établir les coûts structurels des fonctions de dépendance de toutes les commandes gardées. Nous allons montrer dans la suite de cette section comment obtenir le coût structurel à partir de la fonction de dépendance d'une commande gardée: $\text{Coût}_{\text{structurel}}\{F_{\text{dep}}(G(q))\}$.

Le coût structurel de la commande gardée qu'on cherche à établir peut être décomposé en une somme de deux termes (Équation 2.3):

Équation 2.3

$$\text{Coût}_{\text{structurel}}\{F_{\text{dep}}(G(q))\} = \text{Coût}_{\text{structurel}}\{F_{\text{dep}}(G(q))_0\} + \text{Coût}_{\text{structurel}}\left\{F_{\text{dep}}(G(q)) \cap \sum_{q=0}^{Q-1} \overline{F_{\text{dep}}(G(q))}\right\}$$

Le premier coût est le coût structurel intrinsèque de la garde qu'on étudie. Ce coût est directement lié aux instructions présentes dans la commande gardée. Le second terme correspond au coût qu'on appelle coût d'intersection de la garde. Il existe dans une structure de choix des connexions entre les différentes commandes gardées: ce sont des partages de canaux (commandes, entrées et/ou de sorties). Cela induit de la logique supplémentaire et donc une consommation d'énergie supplémentaire qu'il ne faut pas omettre dans le coût final du processus. Le terme $\overline{G(q)}$ signifie toutes les gardes autres que celle qui est considérée.

2.2.1 Coût intrinsèque des commandes gardées

Dans un premier temps, il nous faut établir le coût intrinsèque de chacune des gardes. Nous présentons toutes les étapes de calcul du coût pour une seule commande gardée. Un coût est bien entendu établi pour chacune des commandes gardées présentes dans une structure de choix d'un processus. Le coût intrinsèque d'une commande gardée est directement lié aux actions effectuées dans la garde considérée. Il est établi en fonction du nombre de données entrantes, des opérations effectuées et du nombre de données sortantes.

Ce coût peut être décomposé en plusieurs termes comme le montre l'équation suivante :

$$\begin{aligned} \text{Coût}_{\text{structurel}} \{F_{\text{dep}}(G(q))_0\} &= \text{Coût}_{\text{structurel}} \{G(q)\} + \text{Coût}_{\text{structurel}} \left\{ G(q), \sum_{i=0}^{I-1} \text{In}(i) \right\} \\ &+ \text{Coût}_{\text{structurel}} \left\{ \sum_{j=0}^{J-1} [\text{Out}(j) = F_s(\sum_{i=0}^{I-1} \text{In}(i))] \right\} + \text{Coût}_{\text{structurel}} \{\text{buff}\} + \text{Coût}_{\text{structurel}} \{\text{init}\} \end{aligned}$$

- Le coût d'évaluation de la garde: $\text{Coût}_{\text{structurel}} \{G(q)\}$

Le premier terme du coût intrinsèque correspond au coût de l'évaluation de la garde (voir section 2.2.1.1). Il s'agit d'attribuer un coût à la logique permettant de tester et de valider les conditions booléennes qui composent la garde. Ce coût est directement lié à la forme de la garde. La forme de la garde peut aller d'un simple test d'égalité à une forme beaucoup plus complexe intégrant des opérateurs booléens. Toutes les expressions possibles des gardes que l'on rencontre dans TAST sont listées et les coûts associés sont établis.

- Le coût d'activation de la garde: $\text{Coût}_{\text{structurel}} \left\{ G(q), \sum_{i=0}^{I-1} \text{In}(i) \right\}$

Lorsque la garde est évaluée, le test de la garde fournit une valeur booléenne du type test vrai ou test faux. Si le test est vrai, cela signifie qu'on autorise les instructions à l'intérieur de la commande gardée à être exécutées. Il y a par conséquent un coût correspondant à l'envoi de cette valeur booléenne aux différentes données entrantes, les autorisant ainsi à être consommées. Il s'agit du coût de l'activation de la garde que nous présentons au paragraphe 2.2.1.2.

- Coût de la fonction: $\text{Coût}_{\text{structurel}} \left\{ \sum_{j=0}^{J-1} [\text{Out}(j) = F_s(\sum_{i=0}^{I-1} \text{In}(i))] \right\}$

Il y a ensuite un coût qu'on attribue aux fonctions qui manipulent les entrées et qui produisent les sorties. Ce coût dépend des fonctions qui sont réalisées. Ces opérations peuvent être des opérateurs logiques, arithmétiques, d'émission de données etc. Toutes les fonctions existantes, et pouvant être synthétisées par TAST sont présentées à la section 2.2.1.5.

- Coût de la bufferisation : $\text{Coût}\{\text{buff}\}$

La bufferisation est le moyen de découpler une succession d'actions entre elles (pipeline asynchrone). Elle permet de produire un acquittement aux données entrantes, et ainsi libérer les étages en amont. On utilise la bufferisation pour augmenter le pipeline d'un circuit. Cette bufferisation permet d'accroître le débit d'un circuit et se fait au dépend de la consommation d'énergie. La consommation est pénalisée car de la logique supplémentaire est ajoutée. Cette logique supplémentaire présente un coût qui est décrit au paragraphe 2.2.1.3.

- Coût des sorties initialisantes : $\text{Coût}\{\text{init}\}$

Certaines sorties envoient une valeur valide constante dès l'étape d'initialisation du circuit. De la logique est ajoutée sur le chemin de donnée pour envoyer cette valeur constante à l'initialisation. Par conséquent même en fonctionnement normal, ces portes consomment de l'énergie, il est donc important de compter ce coût dans le coût global du processus (2.2.1.4).

Les prochaines sections développent tous les coûts précédemment cités en fonction des portes de la librairie TAL [MAUR-03], c'est-à-dire les portes de base des circuits asynchrones (Muller2, Muller2R, etc.) (cf. Annexe A). La section 2.2.3 détaille le coût des portes de la librairie TAL en terme de nombre de transitions. Ces portes sont étalonnées en référence à la commutation d'un inverseur.

Par abus de langage, on désigne le coût d'une porte logique en indiquant seulement le nom de la porte. Par exemple, on désigne le coût d'une porte de muller à 2 entrées comme suit:

$$\text{Coût}\{\text{muller}(2)\} = \text{muller}(2)$$

Où $\text{muller}(2)$ désigne le coût de la commutation d'une porte de muller à 2 entrées.

Il existe, en plus des coûts des portes de la librairie TAL, un coût lié aux fourches. Ces fourches permettent d'acheminer un fil vers plusieurs destinations. Ce coût est compté à l'aide de la fonction $\text{Fourche}(X)$ où X est le nombre de destination du fil.

D'autres fonctions apparaissent dans la suite du chapitre: les fonctions $B(X)$ et $L(X)$ désignent respectivement la base et la longueur d'un canal X .

2.2.1.1 Coût associé à l'évaluation de la garde

TAST permet l'implémentation d'un ensemble de tests dans les gardes. On présente les formes de test de bases, des formes beaucoup plus complexes peuvent aussi être implémentées. On montrera que ces formes complexes ne sont qu'une combinaison des formes de bases que nous présentons. Il est bien sûr possible d'établir le coût de n'importe quelle garde aussi complexe soit elle.

- Test d'une variable avec une constante

Les premiers tests de garde qu'on estime sont les tests d'égalité et d'inégalité d'une variable avec une constante (tests les plus simples et les plus utilisés). Le CHP correspondant est le suivant :

```
@[ X = cste => I1
   X /= cste => I2
   ]
```

Il s'agit d'une structure de choix à deux branches. X est une variable de base B et de longueur L; cste est une constante et enfin I1 et I2 sont les instructions associées aux deux gardes de la structure de choix.

Si c'est la branche égalité qui est vraie, seule cette branche consomme de l'énergie, seule son évaluation contribue à la consommation. Son coût est le suivant :

$$\text{Coût}\{X[B][L] = \text{cste}\} = \text{muller}(L)$$

Si c'est la seconde branche qui est vraie, on a alors toutes les combinaisons autres que la valeur de la constante, son coût est le suivant :

$$\text{Coût}\{X[B][L] \neq \text{cste}\} = \text{muller}(L) + \text{OR}(B^L - 1)$$

On considère l'exemple de l'égalité d'une variable de base 2 et de longueur 2 :

```
@[ X = "0.0"[2] => I1
   X /= "0.0"[2] => I2
   ]
```

Le résultat de la synthèse est celui de la Figure 2.3.

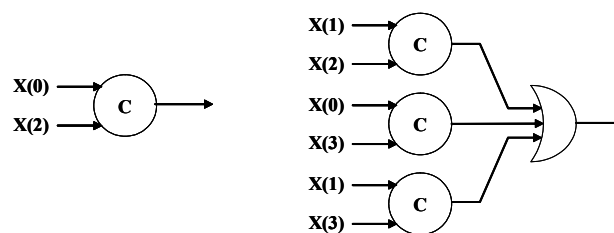


Figure 2.3 Evaluation de l'égalité (gauche) et de l'inégalité (droite) avec une constante

- Test d'une variable avec une autre variable

La seconde forme possible de test d'égalité et d'inégalité qu'on peut rencontrer correspond au test d'une variable avec une autre variable. La spécification CHP est la suivante :

```
@[ X = Y => I1
   X /= Y => I2
   ]
```

Dans le cas où la condition booléenne de l'égalité est vraie, le coût est :

$$\text{Coût}\{X[B][L] = Y[B][L]\} = L * \text{muller}(2) + L * \text{OR}(B) + \text{muller}(L)$$

Autrement, le coût correspond au coût de l'inégalité, il est de :

$$\text{Coût}\{X[B][L] \neq Y[B][L]\} = L * \text{muller}(2) + L * \text{OR}(B) + \text{muller}(L) + \text{OR}(B^L - 1)$$

Choisissons l'exemple de deux variables X et Y ayant un codage du type multi rail MR[2][2], la synthèse de l'évaluation de la garde est décrite à la Figure 2.4.

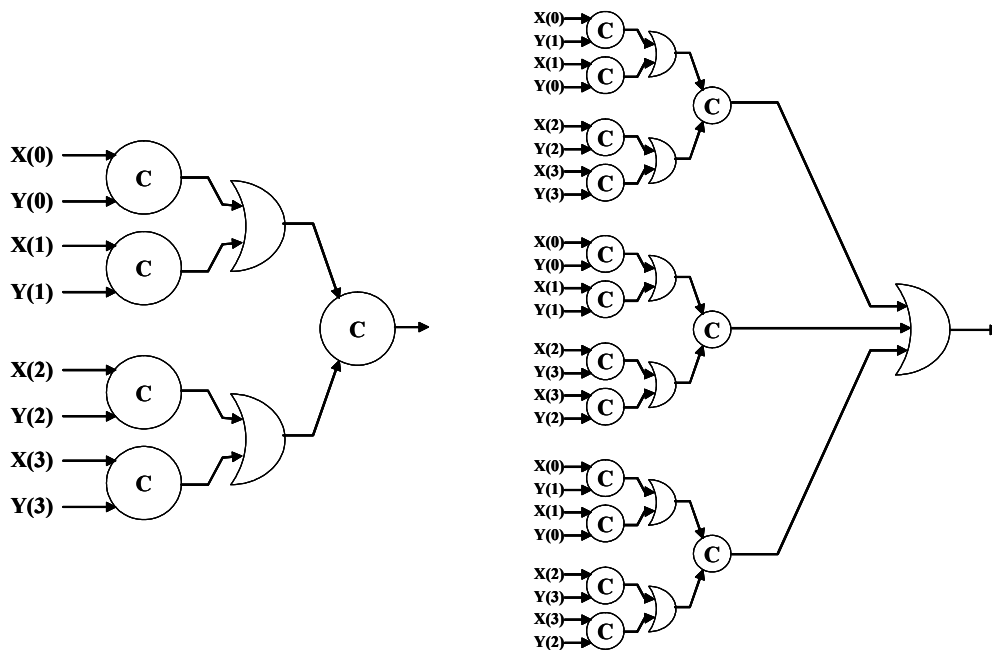


Figure 2.4 Evaluation de l'égalité (gauche) et de l'inégalité (droite) avec une variable

- Test utilisant les opérateurs booléens and et nand

On peut utiliser des tests un peu plus sophistiqués en intégrant des opérateurs logiques "and" et "nand" par exemple. Le code CHP suivant propose l'implémentation de gardes intégrant ces deux opérateurs.

```

@ [Cond1 and Cond2 =>I1
   Cond1 nand Cond2 =>I2
]
```

Où Cond1 et Cond2 sont des conditions d'égalité (de constantes ou de variables) ou d'inégalité (de constantes ou de variables) que nous avons présentés précédemment. Le coût de la garde utilisant l'opérateur "and" est :

$$\text{Coût}\{\text{Cond1 and Cond 2}\} = \text{Coût}\{\text{Cond 1}\} + \text{Coût}\{\text{Cond 2}\} + \text{muller}(2)$$

Le coût de la garde utilisant l'opérateur booléen "nand" est le suivant :

$$\text{Coût}\{\overline{\text{Cond1}} \text{ nand } \overline{\text{Cond2}}\} = \sup \left\{ \begin{array}{l} \text{Coût}\{\overline{\text{Cond1}}\} + \text{Coût}\{\overline{\text{Cond2}}\}, \text{Coût}\{\overline{\text{Cond1}}\} + \text{Coût}\{\text{Cond2}\}, \\ \text{Coût}\{\text{Cond1}\} + \text{Coût}\{\overline{\text{Cond2}}\} \end{array} \right\}$$

+ muller(2) + OR(3)

Où $\overline{\text{Cond1}}$ et $\overline{\text{Cond2}}$ correspondent aux conditions duales de Cond1 et Cond2: par exemple si Cond1 est un test d'égalité, alors $\overline{\text{Cond1}}$ correspond au test d'inégalité, et inversement. On utilise un supérieur pour majorer le coût car il peut y avoir plusieurs combinaisons possibles. Illustrons ce test par l'exemple suivant:

```
@ [X="00"[2] and Y="00"[2] =>I1
  X="00"[2] nand Y="00"[2] =>I2
]
```

La synthèse de l'évaluation des ces gardes complexes est présentée à la Figure 2.5. Cond1 correspond au test X="00"[2] et Cond2 à Y="00"[2].

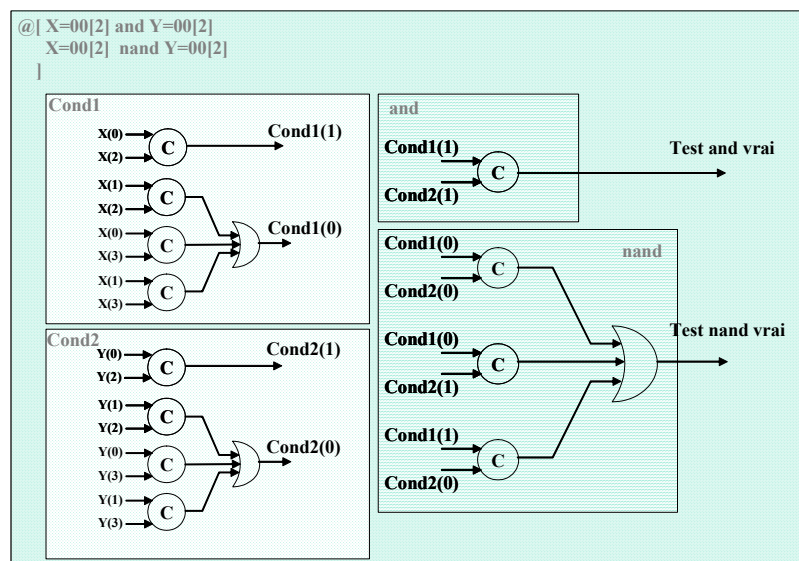


Figure 2.5 Exemple de test incluant les opérateurs and et nand

- Test utilisant les opérateurs "or" et "nor"

On peut rencontrer le cas où la garde est une combinaison en "or" ou en "nor" de deux conditions Cond1 et Cond2. Le code CHP est le suivant :

```
@ [Cond1 or Cond2 =>I1
  Cond1 nor Cond2 =>I2
]
```

Le coût de la combinaison en OR est le suivant :

$$\text{Coût}\{\text{Cond1 or Cond2}\} = \sup \left\{ \begin{array}{l} \text{Coût}\{\overline{\text{Cond1}}\} + \text{Coût}\{\text{Cond2}\}, \text{Coût}\{\text{Cond1}\} + \text{Coût}\{\overline{\text{Cond2}}\} \\ \text{Coût}\{\text{Cond1}\} + \text{Coût}\{\text{Cond2}\} \end{array} \right\}$$

+ muller(2) + OR(3)

Le coût de la combinaison en NOR est le suivant :

$$\text{Coût}\{\text{Cond1 nor Cond2}\} = \text{Coût}\{\overline{\text{Cond1}}\} + \text{Coût}\{\overline{\text{Cond2}}\} + \text{muller}(2)$$

L'exemple suivant est un test incluant les opérateurs or et nor:

```
@ [X="00"[2] or Y="00"[2] =>I1
  X="00"[2] nor Y="00"[2] =>I2
]
```

La synthèse est présentée à la Figure 2.6 (Cond1 correspond au test X="00"[2] et Cond2 à Y="00"[2]).

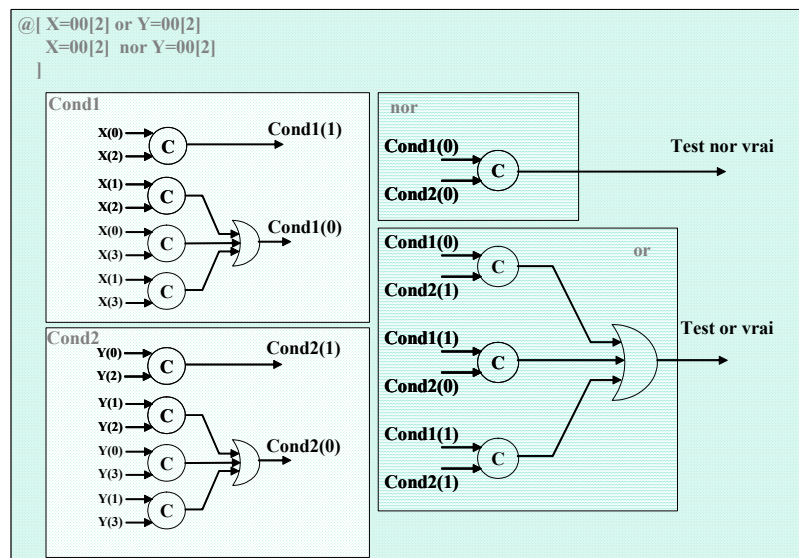


Figure 2.6 Exemple de test incluant les opérateurs or et nor

- Test utilisant les opérateurs "xor" et "xnor"

Enfin, les deux derniers opérateurs qu'on peut rencontrer sont le "xor" et le "xnor". Le CHP est de la forme suivante :

```
@ [Cond1 xor Cond2 =>I1
  Cond1 xnor Cond2 =>I2
]
```

Le coût de condition avec le "xor" s'écrit :

$$\text{Coût}\{\text{Cond1 xor Cond2}\} = \sup\{\text{Coût}\{\overline{\text{Cond1}}\} + \text{Coût}\{\text{Cond2}\}, \text{Coût}\{\text{Cond1}\} + \text{Coût}\{\overline{\text{Cond2}}\}\} + \text{muller}(2) + \text{OR}(2)$$

Le coût de condition avec le "xnor" s'écrit :

$$\text{Coût}\{\text{Cond1 xnor Cond2}\} = \sup\{\text{Coût}\{\overline{\text{Cond1}}\} + \text{Coût}\{\overline{\text{Cond2}}\}, \text{Coût}\{\text{Cond1}\} + \text{Coût}\{\text{Cond2}\}\} + \text{muller}(2) + \text{OR}(2)$$

L'exemple suivant est un test incluant les opérateurs xor et xnor:

```
@ [X="00"[2] xor Y="00"[2] =>I1
  X="00"[2] xnor Y="00"[2] =>I2
]
```

La synthèse est présentée à la Figure 2.7 (Cond1 correspond au test X="00"[2] et Cond2 à Y="00"[2]).

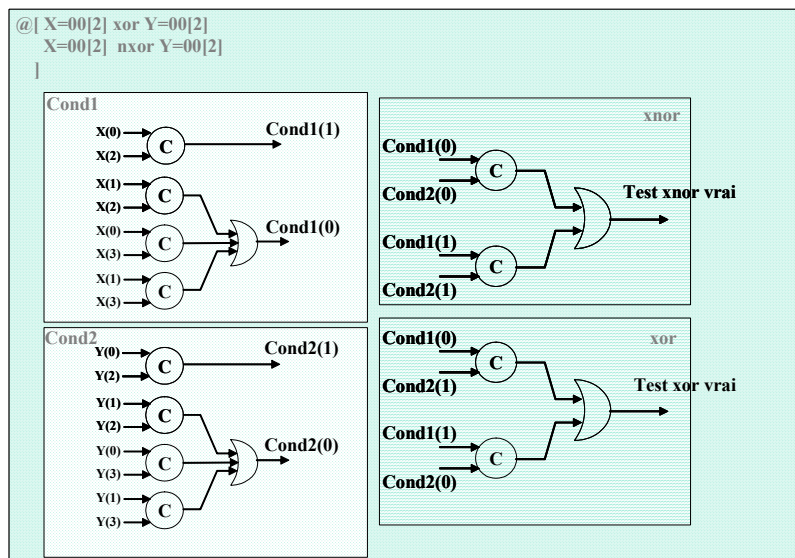


Figure 2.7 Exemple de test incluant les opérateurs xor et xnor

Il est possible de réaliser tout type de structures avec tous les cas cités précédemment, et d'établir très rapidement les coûts associés à l'évaluation de ces gardes.

2.2.1.2 Coût d'activation de la garde

Lorsque le test de la garde est vrai, une valeur booléenne vraie est produite. En asynchrone, il s'agit de la mise à 1 d'un fil. Ce fil autorise le traitement des instructions à l'intérieur de la commande gardée. En fait, ce fil autorise l'arrivée des données (par la suite celles-ci déclencheront le traitement et la production des sorties). Le coût est directement lié au nombre et au type des données entrantes qui sont activées (Figure 2.8).

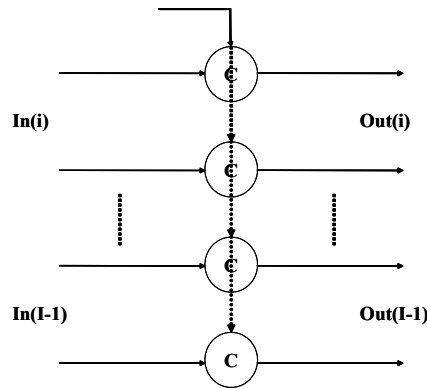


Figure 2.8 Contrôle d'activation

Il y a un rendez-vous qui se crée entre le fil de validité (produit par l'évaluation de la garde) et tous les digits de toutes les données entrantes. Ce rendez-vous se fait à l'aide de portes de muller à 2 entrées. Le nombre de portes de muller est directement proportionnel au nombre et au type des données qui entrent. Le coût de l'activation de la garde est établi selon l'équation suivante :

$$\text{Coût}_{\text{structurel}} \left\{ G(q), \sum_{i=0}^{I-1} \text{In}(i) \right\} = \text{muller}(2) * \sum_{i=0}^{I-1} G(q) \bullet L(\text{In}(i)) + \text{Fourche} \left(\sum_{i=0}^{I-1} G(q) \bullet L(\text{In}(i)) * B(\text{In}(i)) \right)$$

Où $B(\text{In}(i))$ et $L(\text{In}(i))$ représentent respectivement la base et la longueur du canal d'entrée $\text{In}(i)$. $G(q) \bullet L(\text{In}(i))$ est le produit scalaire de la garde $G(q)$ avec la longueur de la donnée $L(\text{In}(i))$. Ce produit scalaire vaut $L(\text{In}(i))$ si la donnée $\text{In}(i)$ est une donnée entrante qui appartient à la commande gardée $G(q)$. En effet, toutes les données entrantes du processus global n'entrent pas forcément dans toutes les commandes gardées de la structure de choix. Le coût d'activation des données d'une commande gardée n'est effectif que si la donnée est une donnée de la commande gardée en question. Il faut aussi remarquer qu'on ne comptabilise que les portes qui commutent, car effectivement toutes les portes de muller ne commutent pas. Seule une porte de muller par digit commute car on est en présence d'une implémentation 1 parmi N (one-hot).

On note la présence d'un second terme qui correspond au coût d'une fourche. Il s'agit en fait de la fourche qu'on crée lorsqu'on envoie le fil de validité vers toutes les portes de muller. La aussi il y a un produit scalaire, le coût est ajouté au coût global si et seulement si la donnée entrante $\text{In}(i)$ appartient à la garde $G(q)$.

Dans la suite, ce type de produit scalaire apparaît souvent, il a toujours la même signification. A savoir, le coût vaut zéro si la donnée entrante (ou sortante) n'appartient pas à la commande gardée qu'on est en train d'estimer, il vaut une valeur non nulle si la donnée existe dans la garde considérée.

2.2.1.3 Coût de la bufferisation

La bufferisation, comme son nom l'indique, correspond à l'introduction de buffers à l'intérieur d'un circuit (étages de pipeline asynchrone). On introduit des buffers à des fins de découplage d'étages. Cela permet de relâcher les contraintes de synchronisation entre des étages successifs. Ce relâchement des contraintes entre les étages permet d'accroître le débit d'un circuit. Cela revient à augmenter le pipeline. La bufferisation peut être réalisée de plusieurs manières, chaque bufferisation présente des caractéristiques qui apportent certains avantages au circuit.

- Sans Bufferisation (COMB)

Les entrées sont manipulées par des blocs logiques, et les résultats sont envoyés vers les sorties. Les acquittements des sorties sont propagés sur les entrées. En fonction du nombre d'entrées et du nombre de sortie, de la logique supplémentaire est ajoutée pour combiner et propager rigoureusement les acquittements des sorties vers les entrées.

- Bufferisation WCHB

Le protocole de bufferisation WCHB est le type de bufferisation le plus utilisé dans les circuits asynchrones que nous implémentons avec TAST. Il permet l'utilisation de l'acquittement de sortie et la production des acquittements des entrées. On réalise une bufferisation WCHB à l'aide de porte de muller avec reset.

- Bufferisation PCHB

PCHB est un second type de bufferisation possible. L'utilisation de l'acquittement de sortie et la production de l'acquittement d'entrée sont gérées différemment par rapport à la bufferisation WCHB. La remise à zéro de la sortie (dès que celle-ci est acquittée) survient plus tôt pour le modèle PCHB. Cette pré-charge s'effectue sans attendre l'invalidité du canal d'entrée.

- Bufferisation PCFB

La bufferisation PCFB est plus concurrente que le modèle PCHB. L'introduction d'une variable est nécessaire pour la mise à zéro des signaux d'acquittement en parallèle. Ce protocole permet de mémoriser une donnée complète dans le pipeline.

Le Tableau 2-1 présente les coûts des différents modèles de bufferisation. Pour chaque type de bufferisation, on distingue les coûts de la bufferisation effectuée sur les données en entrée de la bufferisation des données en sortie. Le coût peut effectivement être différent car le nombre et le type des données en entrée peuvent être différents du nombre et du type des données en sortie. On remarque que les sorties ne correspondent pas aux sorties du processus global, on parle plutôt de sorties intermédiaires. Ces sorties intermédiaires sont les sorties des commandes gardées.

Buff	Entrée sortie	Coût
COMB	Entrée	$\text{Coût}_{\text{COMB_entrée}} = \text{muller} \left(\sum_{j=0}^{J-1} G(q) \bullet \text{Out}(j) \right) + \text{Fourche} \left(\sum_{i=0}^{I-1} G(q) \bullet \text{In}(i) \right)$
	Sortie	$\text{Coût}_{\text{COMB_sortie}} = \text{muller} \left(\sum_{j=0}^{J-1} G(q) \bullet \text{Out}(j) \right) + \text{Fourche} \left(\sum_{i=0}^{I-1} G(q) \bullet \text{In}(i) \right)$
WCHB	Entrée	$\text{coût}_{\text{WCHB_entrée}} = \text{muller} \left(\sum_{j=0}^{J-1} G(q) \bullet \text{Out}(j) \right) + \text{Fourche} \left(\sum_{i=0}^{I-1} G(q) \bullet B(\text{In}(i)) \bullet L(\text{In}(i)) \right) + \sum_{i=0}^{I-1} G(q) \bullet \left[L(\text{In}(i)) \bullet \text{muller } 2R + L(\text{In}(i)) \bullet \text{NOR}(B(\text{In}(i))) + \text{muller}(L(\text{In}(i))) \right]$
	Sortie	$\text{coût}_{\text{WCHB_sortie}} = \text{muller} \left(\sum_{j=0}^{J-1} G(q) \bullet \text{Out}(j) \right) + \text{Fourche} \left(\sum_{j=0}^{J-1} G(q) \bullet B(\text{Out}(j)) \bullet L(\text{Out}(j)) \right) + \sum_{j=0}^{J-1} G(q) \bullet \left[L(\text{Out}(j)) \bullet \text{muller } 2R + L(\text{Out}(j)) \bullet \text{NOR}(B(\text{Out}(j))) + \text{muller}(L(\text{Out}(j))) \right] + \text{muller} \left(\sum_{j=0}^{J-1} G(q) \bullet \text{Out}(j) \right) + \text{Fourche} \left(\sum_{i=0}^{I-1} G(q) \bullet \text{In}(i) \right)$
PCHB	Entrée	$\text{coût}_{\text{PCHB_entrée}} = \text{muller} \left(\sum_{j=0}^{J-1} G(q) \bullet \text{Out}(j) \right) + \text{Fourche} \left(\sum_{i=0}^{I-1} G(q) \bullet B(\text{In}(i)) \bullet L(\text{In}(i)) \right) + \sum_{i=0}^{I-1} G(q) \bullet \left[L(\text{In}(i)) \bullet \left(\text{OR}(B(\text{In}(i))) + \text{muller } 3D1P1NR + \text{OR}(B(\text{In}(i))) + \text{muller } 2D1NB \right) + \text{muller}(L(\text{In}(i))) \right]$
	Sortie	$\text{coût}_{\text{PCHB_sortie}} = \text{muller} \left(\sum_{j=0}^{J-1} G(q) \bullet \text{Out}(j) \right) + \text{Fourche} \left(\sum_{j=0}^{J-1} G(q) \bullet B(\text{Out}(i)) \bullet L(\text{Out}(i)) \right) + \sum_{j=0}^{J-1} G(q) \bullet \left[L(\text{Out}(j)) \bullet \left(\text{OR}(B(\text{Out}(j))) + \text{muller } 3D1P1NR + \text{OR}(B(\text{Out}(j))) + \text{muller } 2D1NB \right) + \text{muller}(L(\text{Out}(j))) \right] + \text{muller} \left(\sum_{j=0}^{J-1} G(q) \bullet \text{Out}(j) \right) + \text{Fourche} \left(\sum_{i=0}^{I-1} G(q) \bullet \text{In}(i) \right)$
PCFB	Entrée	$\text{coût}_{\text{PCFB_entrée}} = \text{muller} \left(\sum_{j=0}^{J-1} G(q) \bullet \text{Out}(j) \right) + \text{Fourche} \left(\sum_{i=0}^{I-1} G(q) \bullet B(\text{In}(i)) \bullet L(\text{In}(i)) \right) + \sum_{i=0}^{I-1} G(q) \bullet \left[L(\text{In}(i)) \bullet \left(\text{OR}(B(\text{In}(i))) + \text{muller}(2) + \text{muller } 2D1PR + \text{OR}(B(\text{In}(i))) + \text{muller } 3D1P1NB + \text{INV} + \text{muller}(2) \right) + \text{muller}(L(\text{In}(i))) \right]$
	Sortie	$\text{coût}_{\text{PCFB_sortie}} = \text{muller} \left(\sum_{j=0}^{J-1} G(q) \bullet \text{Out}(j) \right) + \text{Fourche} \left(\sum_{j=0}^{J-1} G(q) \bullet B(\text{Out}(i)) \bullet L(\text{Out}(i)) \right) + \sum_{j=0}^{J-1} G(q) \bullet \left[L(\text{Out}(j)) \bullet \left(\text{OR}(B(\text{Out}(j))) + \text{muller}(2) + \text{muller } 2D1PR + \text{OR}(B(\text{Out}(j))) + \text{muller } 3D1P1NB + \text{INV} + \text{muller}(2) \right) + \text{muller}(L(\text{Out}(j))) \right] + \text{muller} \left(\sum_{j=0}^{J-1} G(q) \bullet \text{Out}(j) \right) + \text{Fourche} \left(\sum_{i=0}^{I-1} G(q) \bullet \text{In}(i) \right)$

Tableau 2-1 Coût de la bufferisation

Les portes de muller Muller3D1N1PR, muller2D1PR correspondent à des portes de muller dissymétriques introduites dans [RIGA-02]. Ces portes sont longuement explicitées dans la bibliothèque TAL [MAUR-03]. Les coûts associés à ces portes sont donnés au paragraphe 2.2.3.

On s'aperçoit que les équations diffèrent en fonction du choix de la bufferisation. Le Chapitre 4 fera une analyse de ces types de bufferisation pour essayer de mettre en exergue la

bufferisation la mieux adaptée à la faible consommation d'énergie. L'étude sera étendue à l'analyse des bufferisations entrée/sortie, cette étude permettra d'identifier les contextes dans lesquels ces bufferisations sont les mieux adaptées.

2.2.1.4 Coût de l'initialisation

Certaines sorties sont parfois initialisées à une valeur constante dans la partie initialisation du programme CHP. Un signal initialisé à une constante (ici 0) s'écrit en CHP :

$[S!0'; *[\dots S ! In(i)]$

A la synthèse, un nouveau bloc est inséré dans le chemin de cette sortie pour implémenter l'initialisation à une valeur constante (Figure 2.9). Ce nouveau bloc est ajouté après l'étage de bufferisation et ne concerne que les sorties initialisantes.

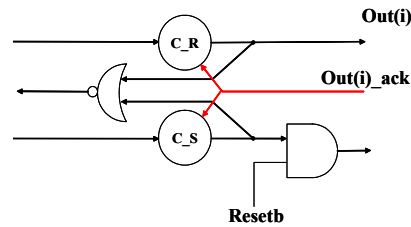


Figure 2.9 Initialisation d'une sortie double rail

Il y a de la logique supplémentaire qui est insérée dans le chemin de donnée. Ce qui signifie qu'il y a un coût lors de l'étape d'initialisation, mais ce coût ne disparaît pas lorsqu'on est en fonctionnement normal. En effet, à chaque fois qu'une donnée est émise sur ce canal, elle traverse cette logique supplémentaire, il y a donc une contribution de cette logique au coût global lors du fonctionnement normal. On s'aperçoit que le chemin propageant la valeur 0 est différent de celui propageant la valeur 1 (traversée d'une porte AND), la consommation est donc différente selon qu'on envoie un 0 ou un 1 en sortie. Par conséquent, le coût d'une telle structure dépend de la valeur de la donnée qui est propagée. On établit de ce fait le coût moyen $Coût(avg)$ en considérant une équiprobabilité de traversée des deux chemins et le coût critique $Coût(wc)$ en considérant le chemin qui consomme le plus (en l'occurrence le chemin avec la porte AND) de cette structure.

$$Coût(wc) = \sum_{j=0}^{J-1} G(q) \bullet \left[\begin{array}{l} L(Out(j)) \cdot (muller2S + AND(2)) \\ + L(Out(j)) \cdot NOR(B(Out(j)) + muller(L(Out(j)))) \end{array} \right]$$

$$Coût(avg) = \sum_{j=0}^{J-1} G(q) \bullet \left[\begin{array}{l} \frac{1}{B(Out(j))} L(Out(j)) \cdot (muller2S + AND(2)) \\ + \frac{B(Out(j)) - 1}{B(Out(j))} L(Out(j)) \cdot muller2R + L(Out(j)) \cdot NOR(B(Out(j))) \\ + muller(L(Out(j))) \end{array} \right]$$

2.2.1.5 Coût de la fonction

Le coût de la fonction réalisée dépend bien évidemment de l'opérateur mis en jeu. Il faut donc lister toutes les fonctions qui peuvent être synthétisées par l'outil TAST. A chacune de ces fonctions est attribué un coût.

- Emission d'une entrée vers la sortie

```
In(i) ? x; Out(j) ! x
```

L'émission d'une entrée vers la sortie sans manipulation de la donnée ne coûte rien car il s'agit d'un prolongement de fils.

- Une entrée envoyée en parallèle sur plusieurs sorties différentes

```
[In(i) ? x;  
Out(0) ! x,  
...,  
Out(j) ! x,  
...,  
Out(J-1) ! x  
]
```

Lorsqu'une donnée est envoyée sur plusieurs canaux de sortie, cela crée des fourches. Le coût vaut:

$$\text{Coût} = L(\text{In}(i) * \text{Fourche}(J))$$

- Le décalage d'une donnée :

```
In(i) ? x; Out(j)[31..0] ! x[63..32]
```

Le décalage de donnée ne coûte rien car, la aussi, il s'agit d'un prolongement de fils de l'entrée vers la sortie (même si l'indice n'est pas respecté).

- Envoi d'une valeur constante sur un canal

```
... S ! cste ...
```

L'envoi d'une constante sur un canal se synthétise en QDI comme suit:

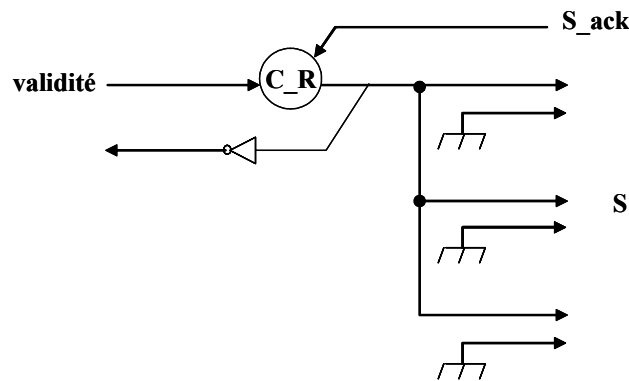


Figure 2.10 Circuit d'émission d'une constante sur un canal

Le canal de validité correspond à la validité des données en entrée, ce signal déclenche l'émission de la constante sur le canal S.

Le coût de cette émission est :

$$\text{Coût} = \sum_{j=0}^{J-1} \text{muller2R} + \text{INV} + \text{Fourche}(L(\text{Out}(j)))$$

- Conversion MR[2] vers MR[4]

Le circuit implémentant la conversion MR[2] vers MR[4] est présenté à la Figure 5.18. Le coût d'une conversion de ce type est égal à:

$$\text{Coût} = L(\text{Out}(j)) * \text{muller}(2)$$

- Conversion MR[4] vers MR[2]

Le circuit implémentant la conversion MR[2] vers MR[4] est présenté à la Figure 5.19. Le coût de cette conversion vaut :

$$\text{Coût} = L(\text{Out}(j)) * \text{OR}(2)$$

- L'opérateur logique unaire : **not**

Complémenter une valeur en multi-rail est très simple, il s'agit d'une simple permutation de fils. En effet, pour un dual rail par exemple, un fil code le 0 et un second fil code le 1. Si une entrée vaut 0 et si on veut complémenter cette entrée, il suffit d'invertir les fils pour obtenir un 1. Ceci peut s'appliquer pour tout type de codage multi-rail.

- L'opérateur logique AND : X_{BL} **and** Y_{BL}

L'estimation de la consommation d'énergie des opérateurs arithmétiques et logiques ne peut être établie à l'aide d'une équation générale qui dépend seulement de la base et de longueur. En effet, les architectures de ces opérateurs ne présentent pas de structures régulières qui nous permettent d'établir une équation unique. Par contre, ce sont des opérateurs digit à digit, par conséquent il est possible d'écrire les formules par digit pour un certain nombre de bases. Pour des opérateurs d'une longueur L, il suffit simplement de multiplier la consommation du digit de base B par la longueur L de la donnée. Pour les opérateurs arithmétiques et logiques,

on a établi des coûts pour les bases 2 et 4. De plus, on propose des estimations de consommation d'énergie soit en moyenne soit critique. On fait cela car on s'aperçoit que le chemin traversé dépend de la valeur des données en entrée.

L'opérateur logique AND synthétisé en base 2 pour un seul digit est présenté à la Figure 2.11.

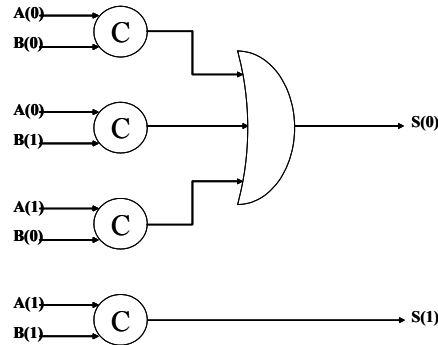


Figure 2.11 Synthèse MR[2] de la porte AND

Un opérateur AND implémenté en MR[4] est présenté à la Figure 2.12.

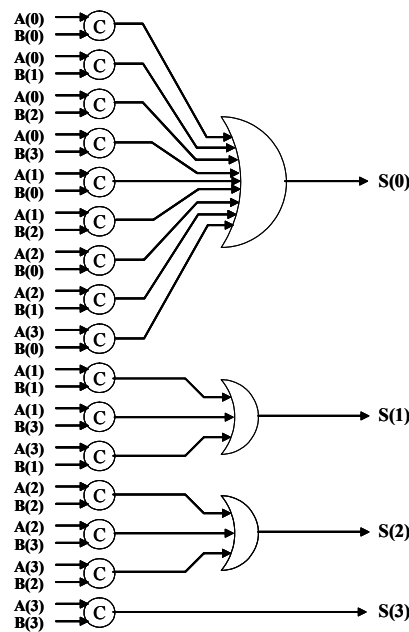


Figure 2.12 Synthèse MR[4] de la porte AND

On constate sur ces deux circuits que la consommation réelle est différente selon la valeur de la sortie obtenue. La sortie à 0 dans les deux circuits consomme plus que les autres. Cela montre bien que la consommation d'énergie dépend du chemin qui est emprunté (et donc de la valeur des données en entrée). L'estimation du nombre de commutations de l'opérateur logique AND est donnée au Tableau 2-2 pour les bases 2 et 4.

Base	Coût
2	$\text{Coût}(\text{avg}) = L * (\text{muller}(2) + \frac{3}{4} * \text{OR}(3))$
	$\text{Coût}(\text{wc}) = L * (\text{muller}(2) + \text{OR}(3))$
4	$\text{Coût}(\text{avg}) = L * (\text{muller}(2) + \frac{6}{16} * \text{OR}(3) + \frac{9}{16} * \text{OR}(9))$
	$\text{Coût}(\text{wc}) = L * (\text{muller}(2) + \text{OR}(9))$

Tableau 2-2 Consommation d'un AND pour les bases 2 et 4

- L'opérateur logique NAND : $X_{BL} \text{ nand } Y_{BL}$

La porte NAND est pratiquement identique à la porte AND. Du point de vue logique rien n'est ajouté, en synthèse on intervertit les fils de sortie (un inverseur en codage 1 parmi N ne coûte rien, il s'agit seulement d'une permutation de fils en sortie). Le fil S(0) devient S(1) et inversement pour le circuit MR[2], pour le circuit MR[4] on retourne les fils du digit (le fil de poids fort devient le fil de poids faible). Les consommations moyenne et critique restent strictement les mêmes.

- L'opérateur logique OR : $X_{BL} \text{ or } Y_{BL}$

L'opérateur OR utilise les mêmes portes que l'opérateur AND. Les combinaisons sont établies différemment (Figure 2.13) mais les consommations moyenne et critique restent identiques.

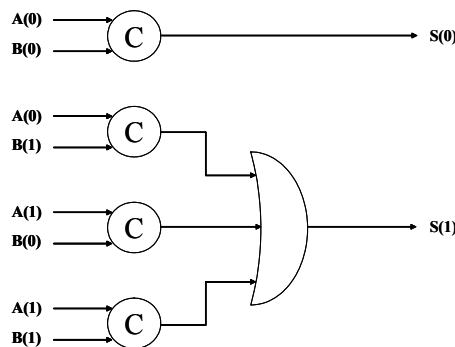


Figure 2.13 Synthèse MR[2] de la porte OR

- L'opérateur logique NOR : $X_{BL} \text{ nor } Y_{BL}$

Pour la porte NOR, on fait la même remarque que pour la porte NAND. La porte NOR se différencie de la porte OR seulement par la permutation des fils en sortie. Le coût reste le même.

- L'opérateur logique XOR : $X_{BL} \text{ xor } Y_{BL}$

Pour la porte XOR, la logique implémentée diffère des portes présentées précédemment. Le circuit synthétisé en base 2 pour un digit est présenté à la Figure 2.14.

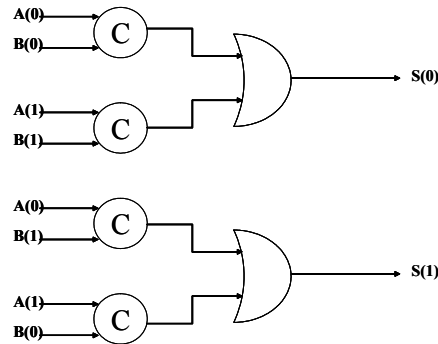


Figure 2.14 Synthèse MR[2] de la porte XOR

Le circuit du XOR implémenté en MR[4] est proposé à la Figure 2.15.

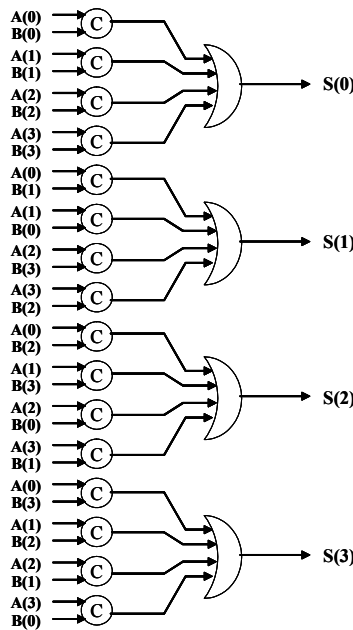


Figure 2.15 Synthèse MR[4] de la porte XOR

La porte XOR en base 2 et en base 4 est une structure symétrique, le coût est absolument identique qu'on produise un 0 ou un 1 en sortie. Pour d'autres bases la symétrie n'existe plus, et les consommations dépendent des données entrantes. Les coûts critique et moyen de l'opérateur XOR sont ceux du Tableau 2-3.

Base	Coût
2	Coût(avg) = $L * (\text{muller}(2) + \text{OR}(2))$
	Coût(wc) = Coût(avg)
4	Coût(avg) = $L * (\text{muller}(2) + \text{OR}(4))$
	Coût(wc) = Coût(avg)

Tableau 2-3 Consommation du XOR pour les bases 2 et 4

- L'opérateur logique XNOR : $X_{BL} \text{ xnor } Y_{BL}$

Le coût pour la porte XNOR est identique au coût d'une porte XOR, on réalise ici aussi une simple permutation des fils de sortie pour réaliser cette porte.

- L'opérateur arithmétique "+" : $X_{BL} + Y_{BL}$

TAST intègre l'opérateur arithmétique d'addition qui est implémenté à l'aide d'une architecture Carry Ripple Adder [FRAG-03]. La cellule de base est un Full Adder.

Un Full Adder implémenté en MR[2] est illustré à la Figure 2.16.

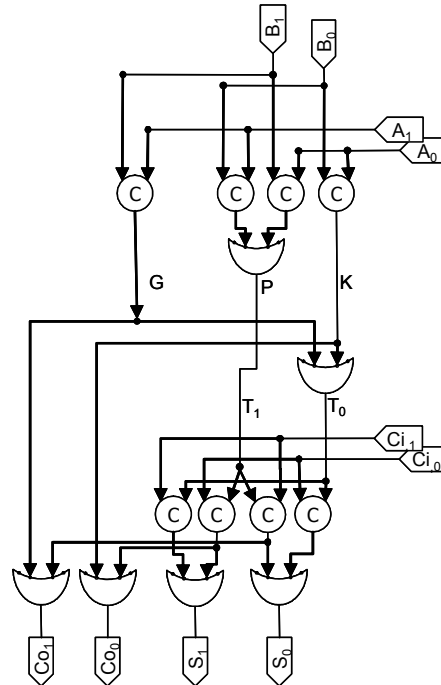


Figure 2.16 Full Adder (FA) MR2

Le Full Adder implémenté en MR[4] est celui de la Figure 2.17.

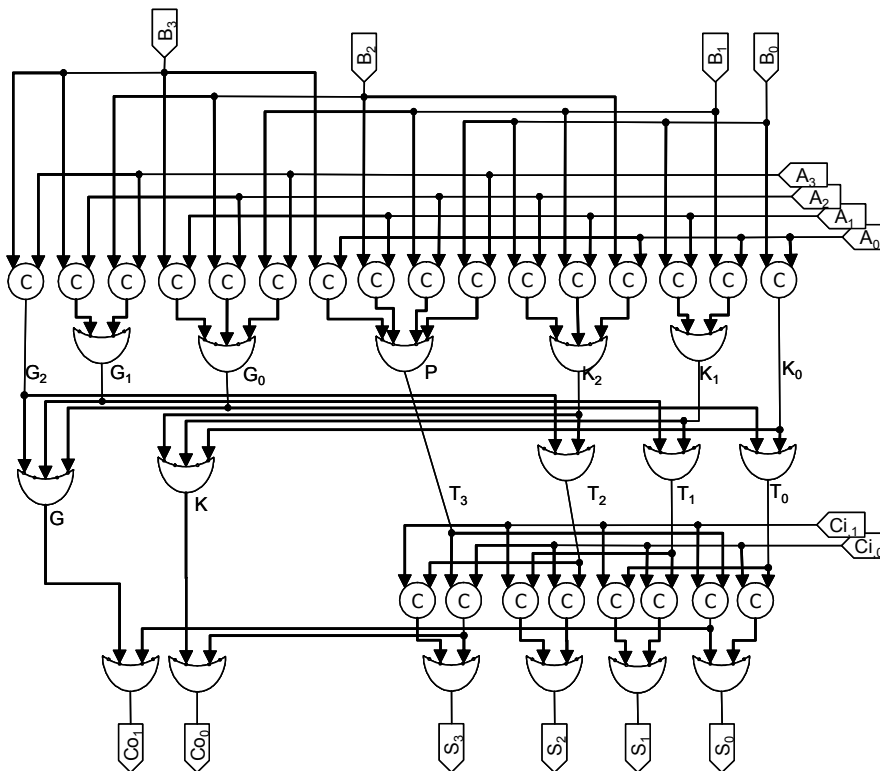


Figure 2.17 Full Adder (FA) MR4

Les coûts moyen et critique sont illustrés au Tableau 2-4.

Base	Coût
2	$\text{Coût}(\text{avg}) = L \cdot (2 \cdot \text{muller}(2) + 2 \cdot \text{OR}(2))$
	$\text{Coût}(\text{wc}) = \text{Coût}(\text{avg})$
4	$\text{Coût}(\text{avg}) = L \cdot (2 \cdot \text{muller}(2) + 3 \cdot \text{OR}(2) + 1,125 \cdot \text{OR}(3) + \frac{1}{4} \cdot \text{OR}(4))$
	$\text{Coût}(\text{wc}) = L \cdot (2 \cdot \text{muller}(2) + 3 \cdot \text{OR}(2) + 2 \cdot \text{OR}(3))$

Tableau 2-4 Consommation de l'additionneur pour les bases 2 et 4

- L'opérateur arithmétique "*" : $X_{BL} * Y_{BL}$

De la même manière que pour l'opération d'addition, on établit les coûts de l'opérateur de multiplication pour les bases 2 et 4 (Tableau 2-5). Il s'agit d'un "array multiplieur" utilisant l'opérateur d'addition précédent (noté FA pour Full Adder) et l'opérateur logique AND. Le circuit du multiplieur MR[2] est présenté à la Figure 2.18.

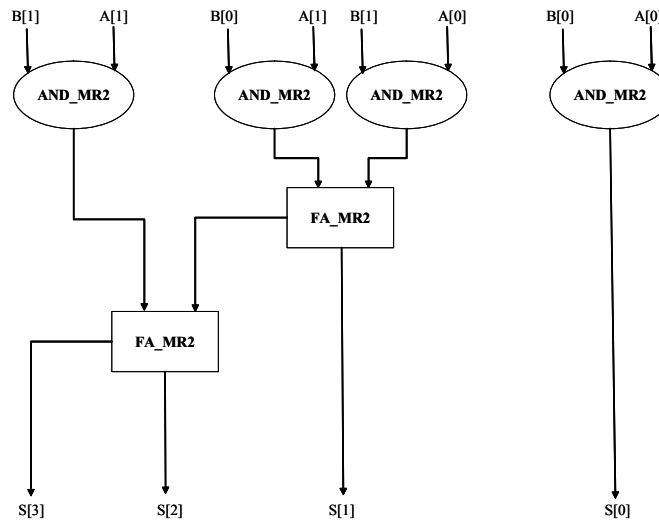


Figure 2.18 Multiplieur MR[2]

Pour le multiplieur en base 4, il est longuement présenté dans les travaux de thèse de Joao Frago do laboratoire TIMA.

Base	Coût
2	$\text{Coût}(\text{avg}) = L^2 \cdot (\text{muller}(2) + \frac{3}{4} \cdot \text{OR}(3)) + (L^2 - L) \cdot \text{FA}(2)$
	$\text{Coût}(\text{wc}) = L^2 \cdot (\text{muller}(2) + \text{OR}(3)) + (L^2 - L) \cdot \text{FA}(2)$
4	$\text{Coût}(\text{avg}) = L^2 \cdot (\text{muller}(2) + 0,125 \cdot \text{OR}(2) + \frac{3}{16} \cdot \text{OR}(3) + \frac{1}{2} \cdot \text{OR}(8) + \frac{1}{4} \cdot \text{OR}(4) + \frac{3}{4} \cdot \text{OR}(12)) + (2 \cdot L^2 - L) \cdot \text{FA}(4)$
	$\text{Coût}(\text{wc}) = L^2 \cdot (\text{muller}(2) + \text{OR}(12)) + (2 \cdot L^2 - L) \cdot \text{FA}(4)$

Tableau 2-5 Consommation du multiplieur pour les bases 2 et 4

2.2.2 Coût d'intersection des branches

Le coût d'intersection des branches correspond au coût des canaux d'entrée et de sortie qui sont partagés par plusieurs gardes. L'équation de coût de l'intersection des branches peut s'écrire selon la décomposition suivante:

$$\text{Coût}_{\text{structurel}} \left\{ F_{\text{dep}}(G(q)) \cap \sum_{q=0}^{Q-1} F_{\text{dep}}(\overline{G(q)}) \right\} = \text{Coût}_{\text{structurel}} \left\{ \sum_{q=0}^{Q-1} G(q) \right\} + \text{Coût}_{\text{structurel}} \left\{ \sum_{q=0}^{Q-1} G(q), \sum_{i=0}^{I-1} \text{In}(i) \right\} \\ + \text{Coût}_{\text{structurel}} \left\{ \sum_{q=0}^{Q-1} G(q), \sum_{j=0}^{J-1} \text{Out}(j) \right\} + \text{Coût}_{\text{structurel}} \{ \text{Buff_sortie} \}$$

- Coût du canal de commande $\text{Coût}_{\text{structurel}} \left\{ \sum_{q=0}^{Q-1} G(q) \right\}$

Il s'agit du coût de la commande principale de la structure de choix qui est utilisée par les commandes gardées. Ce coût vaut:

$$\text{Coût}_{\text{structurel}} \left\{ \sum_{q=0}^{Q-1} G(q) \right\} = \text{AND} \left(\sum_{q=0}^{Q-1} G(q) \right)$$

Ce coût correspond à l'acquittement de la commande qui dépend de la taille de la structure de choix. Il dépend directement du nombre de gardes de la structure, il augmente si le nombre de gardes augmente. Seule une branche est exécutée à un moment donné (propriété de l'exclusion mutuelle). Il faut une porte AND pour regrouper les acquittements de toutes les branches (Figure 2.19) (rappelons que les acquittements sont actifs au niveau bas).

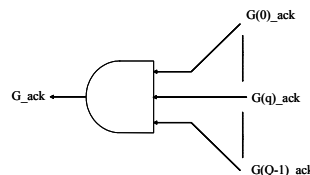


Figure 2.19 Acquittement de la commande

- Coût des fourches de données $\text{Coût}_{\text{structurel}} \left\{ \sum_{q=0}^{Q-1} G(q), \sum_{i=0}^{I-1} \text{In}(i) \right\}$

Il y a ensuite un coût supplémentaire lié aux canaux d'entrée qui sont utilisés par plusieurs commandes gardées. Des fourches sont nécessaires pour acheminer les fils aux différentes commandes gardées de la structure de choix (Figure 2.20).

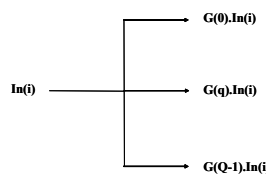


Figure 2.20 Fourche des données en entrée

De plus, si une entrée est utilisée dans plusieurs commandes gardées, l'acquittement de la donnée doit tenir compte de l'utilisation de cette donnée. Il faut ajouter de la logique supplémentaire pour acquitter convenablement les entrées utilisées dans plusieurs commandes gardées de la structure de choix (Figure 2.21).

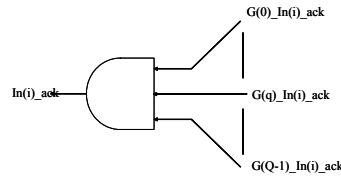


Figure 2.21 Acquittement des données entrantes

Le coût de partage des données entrantes correspond à la somme de ces deux contributions, il s'écrit donc :

$$\text{Coût}_{\text{structurel}} \left\{ \sum_{q=0}^{Q-1} G(q), \sum_{i=0}^{I-1} \text{In}(i) \right\} = \sum_{i=0}^{I-1} \left[L(\text{In}(i)) * \text{Fourche} \left(\sum_{q=0}^{Q-1} G(q) \bullet \text{In}(i) \right) + \text{AND} \left(\sum_{q=0}^{Q-1} G(q) \bullet \text{In}(i) \right) \right]$$

Le terme $G(q) \bullet \text{In}(i)$ représente le produit scalaire de l'entrée $\text{In}(i)$ avec la garde $G(q)$. Ce produit vaut 1 si l'entrée $\text{In}(i)$ est utilisée dans la garde $G(q)$, 0 sinon.

- Coût des données sortantes $\text{Coût}_{\text{structurel}} \left\{ \sum_{q=0}^{Q-1} G(q), \sum_{j=0}^{J-1} \text{Out}(j) \right\}$

On a discuté jusqu'à présent des données qui entraînent, l'interconnexion et la logique que cela engendrait. On s'intéresse maintenant aux sorties. En effet, une même sortie peut parfois être générée par plusieurs branches de la structure de choix. Il faut donc pouvoir regrouper toutes ces sorties intermédiaires en une seule sortie (correspondant à la sortie finale du processus). Il s'agit de les réunir sous forme d'une porte OR en sortie (Figure 2.22).

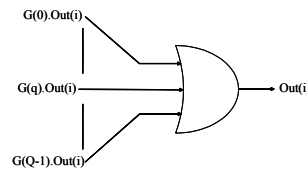


Figure 2.22 Réunion des sorties

Le coût de cette réunion en sortie vaut :

$$\text{Coût}\{\text{réunion}\} = \sum_{j=0}^{J-1} \left[L(\text{Out}(j)) * \text{OR} \left(\sum_{q=0}^{Q-1} G(q) \bullet \text{Out}(j) \right) \right]$$

Ici aussi, $G(q) \bullet \text{Out}(j)$ correspond au produit scalaire. Si la sortie $\text{Out}(j)$ est présente dans la garde $G(q)$, alors ce produit vaut 1, et la somme est augmentée. On obtiendra au final une porte OR avec un nombre d'entrée qui correspond au nombre de fois que la sortie est présente dans les commandes gardées. De plus, les acquittements des sorties qui sont présentes dans plusieurs gardes doivent pouvoir envoyer leurs acquittements aux commandes gardées qui les utilisent. Il faut donc acheminer les acquittements des sorties vers les commandes gardées qui

utilisent ces acquittements, ce qui induit de nouvelles fourches. Le coût de ces fourches est le suivant:

Si le composant est bufferisé en entrée :

$$\text{Coût} = \sum_{j=0}^{J-1} \text{Fourche} \left(\sum_{q=0}^{Q-1} \overline{G(q)} \bullet \text{Out}(j) \right) * \sum_{i=0}^{I-1} B(\text{In}(i)) * L(\text{In}(i))$$

Si le composant est bufferisé en sortie (on parle ici de sorties intermédiaires) :

$$\text{Coût} = \sum_{j=0}^{J-1} \text{Fourche} \left(\sum_{q=0}^{Q-1} \overline{G(q)} \bullet \text{Out}(j) \right) * B(\text{Out}(j)) * L(\text{Out}(j))$$

On utilise $\overline{G(q)}$ pour exclure la garde qu'on traite. En effet, la fourche correspondant à la garde qu'on traite a déjà été comptabilisée dans la bufferisation du coût intrinsèque. On compte ici toutes les autres gardes qui contiennent des canaux de sorties partagés.

- Coût de la bufferisation de la sortie finale du processus $\text{Coût}\{\text{Buff_sortie}\}$

Enfin, on comptabilise un dernier coût qui est une bufferisation supplémentaire qui est proposée aux structures de choix de plusieurs gardes. En effet, la bufferisation peut se faire dans la commande gardée elle-même (bufferisation en entrée ou bufferisation sur les sorties intermédiaires) ou encore pour les structures de choix à plusieurs gardes, la bufferisation peut s'opérer sur les sorties finales (la Figure 2.23 montre une bufferisation WCHB sur une sortie finale).

NB: Bien entendu, si on décide de bufferiser la sortie finale de la structure de choix, il ne faut pas avoir de bufferisation à l'intérieur de la commande gardée.

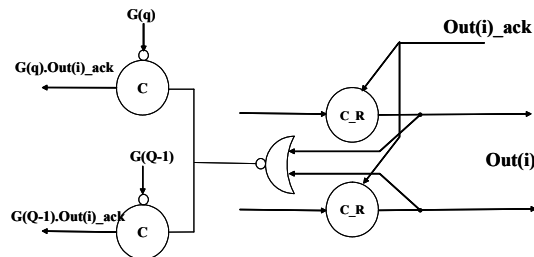


Figure 2.23 Bufferisation en sortie d'une structure de choix à plusieurs gardes

Le coût de la bufferisation réalisée de la sorte est :

$$\text{Coût}\{\text{buff_sortie}\} = \text{Coût}\{\text{buff}\} + \text{INV} + \text{muller} \left(1 + \sum_{j=0}^{J-1} G(q) \bullet \text{Out}(j) \right) + \text{Fourche} \left(\sum_{i=0}^{I-1} G(q) \bullet \text{In}(i) \right)$$

2.2.3 Consommation des portes

Les estimations de la consommation d'énergie sont établies en fonction des portes présentes dans le circuit. Ces portes sont de nature et de taille différentes, et par conséquent ont des consommations d'énergie différentes. Dans l'estimation de la consommation d'énergie, on veut pouvoir comptabiliser les commutations survenues lors d'une simulation donnée. Pour obtenir une valeur de transition qui est en adéquation avec la consommation d'énergie réelle

d'un circuit, il est intéressant de pondérer les commutations en fonction de la nature des portes, de manière à ce qu'une porte complexe ait un coût plus important qu'une porte simple. On procède dans un premier temps à la décomposition des portes AND, OR, NAND et NOR ayant un nombre d'entrée supérieur à 2 en un arbre de portes NAND et NOR à 2 entrées.

Les portes OR à n entrées sont décomposées en une alternance de portes NOR/NAND à 2 entrées. Il est parfois nécessaire d'ajouter un inverseur en sortie de la décomposition pour ne pas altérer la fonctionnalité de la porte initiale. Si le nombre d'étages de décomposition est impair (à savoir $\text{sup}|\log_2 n|$ est impair), on doit ajouter un inverseur en sortie pour respecter la fonctionnalité de la porte initiale. La Figure 2.24-a montre la représentation d'une porte OR à huit entrées à l'aide d'une alternance de portes NOR/NAND à deux entrées. On note la présence d'un inverseur en sortie pour respecter la fonctionnalité de la porte OR à huit entrées (car $\text{sup}|\log_2 8| = 3$ est impair).

Pour un nombre d'étages pair (à savoir $\text{sup}|\log_2 n|$ est pair), aucun inverseur n'est ajouté car la fonctionnalité de la porte initiale est respectée. La Figure 2.24-b montre la décomposition d'une porte OR à 4 entrées à l'aide de l'alternance NOR/NAND à deux entrées ($\text{sup}|\log_2 4| = 2$ est pair, il ne faut donc pas ajouter d'inverseur en sortie).

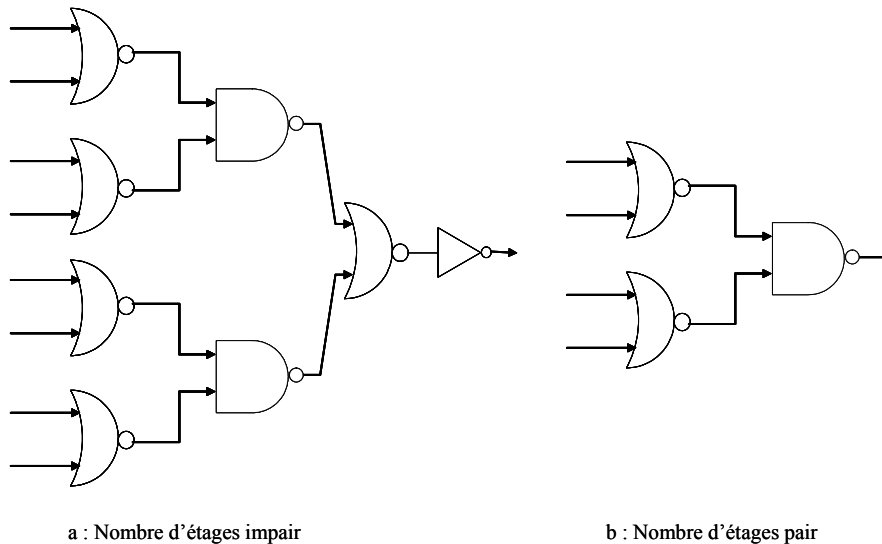


Figure 2.24 Décomposition d'une porte OR à n entrées

Dans le cas d'une porte AND à n entrées, la décomposition est basée sur le même principe. L'alternance est cette fois-ci du type NAND/ NOR. Les mêmes remarques sont faites quant à la parité du nombre d'étages (Figure 2.25).

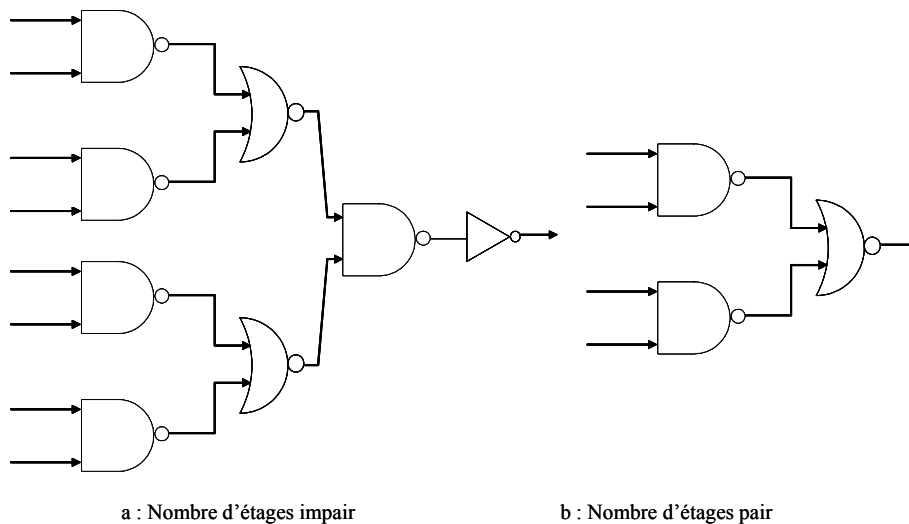


Figure 2.25 Décomposition d'une porte AND à n entrées

Pour une porte NAND à n entrées, on utilise la décomposition de la porte AND à n entrées à laquelle on ajoute un inverseur. S'il s'agit d'un nombre impair d'étages, on observe deux inverseurs à la suite qu'on peut supprimer. Donc finalement, on ajoute un inverseur seulement dans le cas où on a un nombre d'étage pair.

Pour une porte NOR à n entrées, on fait la même remarque que pour la porte NAND à n entrées. Le raisonnement est identique.

On peut établir ainsi une formule unique pour la décomposition de toutes ces portes :

$$\text{OR}(n) = \text{NOR}(n) = \text{AND}(n) = \text{NAND}(n) = \sup \lfloor \log_2 n \rfloor * \left(\frac{\text{NAND}(2) + \text{NOR}(2)}{2} \right)$$

Il s'agit d'un logarithme car toutes les portes de l'arbre ne commutent pas. Il y a seulement une branche qui commute. En effet, dans les circuits asynchrones, les portes AND, OR, NAND, NOR autorisent le changement d'une seule entrée afin de respecter les propriétés des circuits QDI.

Etant donné qu'il y a toujours une alternance NAND/ NOR ou NOR/NAND, on considère la moyenne du coût de ces deux portes. De plus, on ne comptabilise pas la consommation de l'inverseur en fonction de la parité du nombre d'étages de la décomposition. Cela nous permet d'obtenir une formule unique pour ces portes.

On établit maintenant la relation de décomposition des portes de muller à n entrées en fonction de portes de muller à deux entrées (la Figure 2.26 montre l'exemple d'une porte de muller à huit entrées). Une porte de muller commute à 1 lorsque toutes ses entrées sont à 1, elle repasse à 0 lorsque toutes ses entrées reprennent la valeur 0. Par conséquent le coût d'une porte de muller à n entrées correspond à la commutation de l'ensemble des portes de muller à 2 entrées dans l'arbre de décomposition. On écrit le coût:

$$\text{muller}(n) = (n - 1) * \text{muller}(2)$$

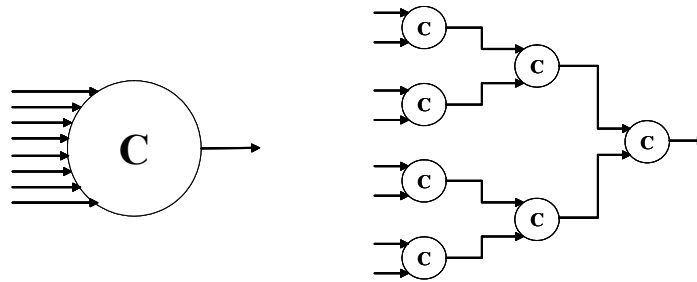


Figure 2.26 Décomposition d'une porte de muller à 8 entrées

On souhaite enfin établir le coût de la fonction fourche qu'on a souvent vu apparaître dans les équations de coûts. Une fourche est à la base un fil qu'on connecte à un certain nombre de destinataires. Le problème est qu'on ne peut pas connecter un fil à un nombre infini de destinataire. Il faut amplifier pour produire assez de puissance pour faire commuter les portes auxquelles il est relié. Pour ce faire, on crée un arbre d'inverseurs de manière à ce que chaque inverseur soit connecté à un nombre borné de destinataires pour que la puissance soit suffisante pour faire commuter les portes destinataires. La Figure 2.27 montre comment la fourche est modélisée.

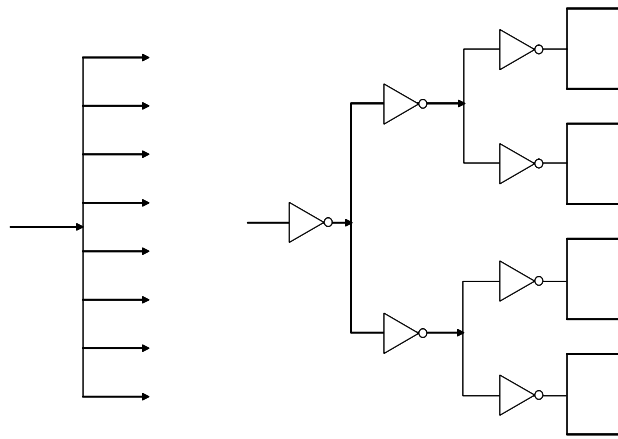


Figure 2.27 Fourche non bufferisée (gauche) et bufferisée (droite)

Donc à chaque fois qu'on émet une valeur sur la fourche, il y a finalement des inverseurs qui commutent. Un fil peut généralement être connecté à un certain nombre de portes sans dégrader la valeur du signal et le délai de commutation des portes auxquelles il est relié. On estime qu'une fourche est correctement bufferisée si la sortie d'une porte est connectée à huit destinataires au maximum. Les circuits que nous avons réalisés ces dernières années suivent ce principe. Par conséquent le coût d'une fourche à n destinataires vaut :

$$\text{Fourche}(n) = (n \text{ div } 8) * \text{INV}$$

Enfin, on calibre la consommation d'énergie en terme de nombre de transitions des portes de base à 2 entrées qu'on rencontre dans la conception de circuits asynchrones. On fixe le nombre de commutation d'un inverseur à une transition. L'inverseur est notre étalon ; les autres portes sont calibrées en fonction de la consommation d'un inverseur. Pour calibrer les portes en fonction de l'inverseur, on utilise l'effort logique [SUTH-99] qui permet de pondérer une porte logique de façon à ce que cette porte produise le même courant de sortie que ce que produirait un inverseur avec la même capacité d'entrée. Le Tableau 2-6 liste les efforts logiques de toutes les portes qu'on rencontre en asynchrone.

porte	intitulé	symbole	Nombre de transitions
INV	inverseur		1
NAND	Porte nand à 2 entrées		4/3
NOR	Porte nor à 2 entrées		5/3
Muller(2)	Muller à 2 entrées		2
Muller2R	Muller à 2 entrées avec reset		2
Muller2S	Muller à 2 entrées avec set		2
Muller2B	Muller à 2 entrées avec sortie complémentée		2
Muller2BR	Muller à 2 entrées avec sortie complémentée avec reset		2
Muller2BS	Muller à 2 entrées avec sortie complémentée avec set		2
Muller2D1N	Muller dissymétrique négative		2
Muller2D1NR	Muller dissymétrique négative avec reset		2
Muller2D1NS	Muller dissymétrique négative avec set		2
Muller2D1P	Muller dissymétrique positive		2
Muller2D1PR	Muller dissymétrique positive avec reset		2
Muller2D1PS	Muller dissymétrique positive avec set		2
Muller2D1NB	Muller dissymétrique négative avec sortie complémentée		2
Muller2D1NBR	Muller dissymétrique négative avec sortie complémentée avec reset		2
Muller2D1NBS	Muller dissymétrique négative avec sortie complémentée avec set		2
Muller2D1PB	Muller dissymétrique positive avec sortie complémentée		2

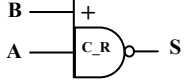
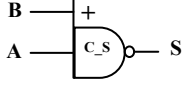
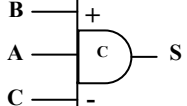
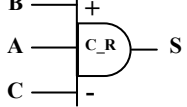
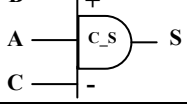
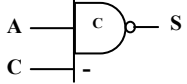
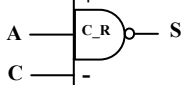
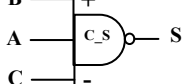
Muller2D1PBR	Muller dissymétrique positive avec sortie complémentée avec reset		2
Muller2D1PBS	Muller dissymétrique positive avec sortie complémentée avec set		2
Muller3D1P1N	Muller dissymétrique positive négative		2
Muller3D1P1NR	Muller dissymétrique positive négative avec reset		2
Muller3D1P1NS	Muller dissymétrique positive négative avec set		2
Muller3D1P1NB	Muller dissymétrique positive négative avec sortie complémentée		2
Muller3D1P1NBR	Muller dissymétrique positive négative avec sortie complémentée avec reset		2
Muller3D1P1NBS	Muller dissymétrique positive négative avec sortie complémentée avec set		2

Tableau 2-6 Estimation du nombre de transitions des portes de base

2.2.4 Consommation des architectures de base

Nous allons dans ce paragraphe définir certaines architectures de base qu'on rencontre dans les circuits asynchrones. Ces architectures se déduisent de l'architecture générale que nous avons définie précédemment. Les architectures de base sont des cas particuliers de l'architecture générale, elles correspondent à des valeurs particulières des paramètres Q, I, J définissant respectivement le nombre de gardes, le nombre d'entrées et le nombre de sorties. On établit les équations de la consommation des architectures de base implémentées à l'aide du protocole WCHB. On établit les coûts de ces architectures de base pour une bufferisation en entrée et pour une bufferisation en sortie.

2.2.4.1 Latch

Le latch (aussi appelé buffer) est un composant qui permet de découpler deux étages successifs. Il permet de propager l'entrée vers la sortie en retournant un acquittement à l'étage qui le précède (le libérant ainsi). C'est un composant souvent utilisé pour augmenter le pipeline d'un circuit (il correspond à un registre de stockage en synchrone). Le latch peut être défini à partir de l'équation de dépendance générale Équation 2.1 en fixant les paramètres Q, I, J définissant respectivement les gardes, les entrées et les sorties.

- Q = 1 : il y a une garde (elle est toujours vraie).
- I = 1 : il y a une seule entrée.

- $J = 1$: il y a une sortie.

L'équation de dépendance générale se réduit donc à l'équation suivante :

$$\text{Latch} = F_{\text{dep}}(\text{In}(0); G(0) = \text{TRUE}; \text{Out}(0) = \text{In}(0))$$

Le code CHP du latch est le suivant :

```
[In(0) ? X; Out(0) ! X; loop
]
```

Etant donné qu'on a une seule entrée et une seule sortie de même type et de même longueur, la bufferisation ne diffère pas qu'on choisisse de bufferiser l'entrée ou la sortie. La synthèse du latch est la suivante (Figure 2.28).

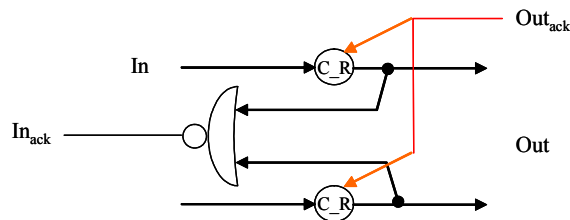


Figure 2.28 Synthèse du Latch

Le coût vaut :

$$\begin{aligned} \text{Coût}_{\text{WCHB_entree}}\{\text{latch}\} &= \text{Coût}_{\text{WCHB_sortie}}\{\text{latch}\} = \\ & \text{Fourche}(\text{B}(\text{In}(0))*\text{L}(\text{In}(0))) + \text{L}(\text{In}(0))*\text{muller2R} + \text{L}(\text{In}(0))*\text{NOR}(\text{B}(\text{In}(0))) \\ & + \text{muller}(\text{L}(\text{In}(0))) \end{aligned}$$

2.2.4.2 Fourche ("Fork")

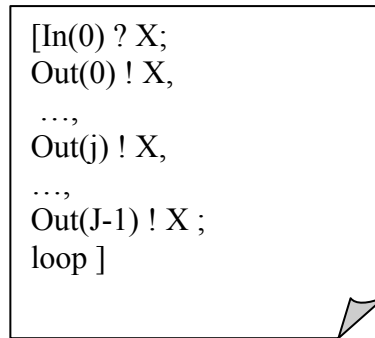
Le composant "Fork" (fourche) permet d'envoyer une donnée sur plusieurs bus différents (la base et la longueur des canaux de sorties sont du même type que la donnée entrante). Les bus de sorties doivent être correctement synchronisés. Les paramètres Q, I et J sont fixés comme suit:

- $Q = 1$: il y a une seule garde qui est toujours vraie.
- $I = 1$: il y a une seule donnée entrante
- $0 \leq j < J - 1$: il y a en tout J données sortantes

L'équation de dépendance du "fork" déduite de l'équation générale Équation 2.1 s'écrit :

$$\text{Fork} = F_{\text{dep}}(\text{In}(0); G(0) = \text{TRUE}; \sum_{j=0}^{J-1} \text{Out}(j) = \text{In}(0))$$

Le code CHP du "fork" est :



Etant donné que le nombre de données en entrée diffère du nombre de données en sortie, on obtient par conséquent deux circuits différents en fonction de la bufferisation en entrée (Figure 2.29) et en sortie (Figure 2.30).

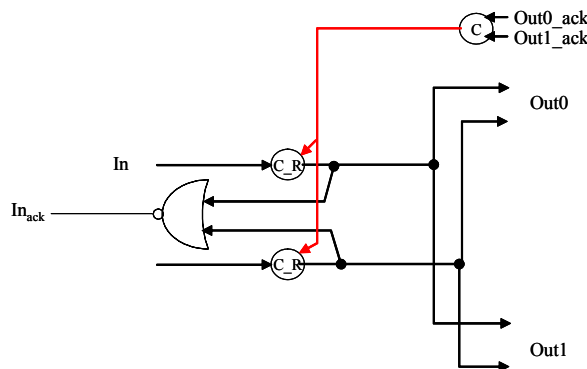


Figure 2.29 Fork bufferisé en entrée

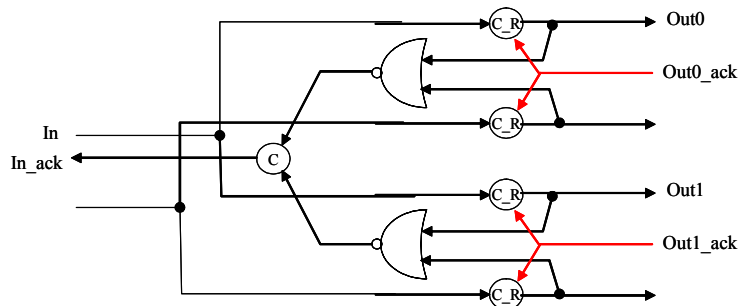


Figure 2.30 Fork bufferisé en sortie

Il existe donc deux équations de la consommation pour ce composant correspondant à une bufferisation WCHB en entrée et à une bufferisation WCHB en sortie.

Pour une bufferisation WCHB en entrée, le coût est :

$$\text{Coût}_{\text{WCHB_entree}} \{\text{Fork}\} = L(\text{In}(0)) * \text{Fourche}(J) + \text{muller}(J) + \text{Fourche}(B(\text{In}(0)) * L(\text{In}(0))) + L(\text{In}(0)) * \text{muller}2R + L(\text{In}(0)) * \text{NOR}(B(\text{In}(0))) + \text{muller}(L(\text{In}(0)))$$

Pour une bufferisation WCHB en sortie, le coût s'écrit :

$$\text{Coût}_{\text{WCHB_sortie}} \{\text{Fork}\} = L(\text{In}(0)) * \text{Fourche}(J) + \text{muller}(J) + \text{Fourche}(J * B(\text{Out}(0)) * L(\text{Out}(0))) + J * L(\text{Out}(0)) * \text{muller}2R + J * L(\text{Out}(0)) * \text{NOR}(B(\text{Out}(0))) + J * \text{muller}(L(\text{Out}(0)))$$

On choisit arbitrairement la sortie Out(0) ici car toutes les sorties ont la même base et la même longueur.

2.2.4.3 Jonction ("Join")

Le composant "Join" permet de synchroniser un certain nombre de bus d'entrée, d'utiliser les valeurs contenues dans les bus pour réaliser une fonction logique ou arithmétique et d'émettre le résultat sur un bus de sortie. Les valeurs de Q, I, J sont les suivantes:

- $Q = 1$: il y a une seule garde qui est toujours vraie.
- $0 \leq i < I - 1$: il y a en tout I données entrantes
- $J = 1$: il y a en seule donnée sortante

L'équation de dépendance du "Join" s'écrit:

$$\text{Join} = F_{\text{dep}}\left(\sum_{i=0}^{I-1} \text{In}(i); G(0) = \text{TRUE}; \text{Out}(0) = F_s\left(\sum_{i=0}^{I-1} \text{In}(i)\right)\right)$$

Le code CHP du "join" s'écrit :

```
[In(0) ? X,
..., In(i) ? Y,
..., In(I-1) ? Z;
Out(0) ! fct((X,Y,...,Z) ;
loop ]
```

Le nombre d'entrées étant supérieur au nombre de sorties, on aboutit logiquement à deux équations de coût suivant qu'on choisisse de bufferiser les entrées ou la sortie. Pour une bufferisation en entrée, le circuit est le suivant (Figure 2.31).

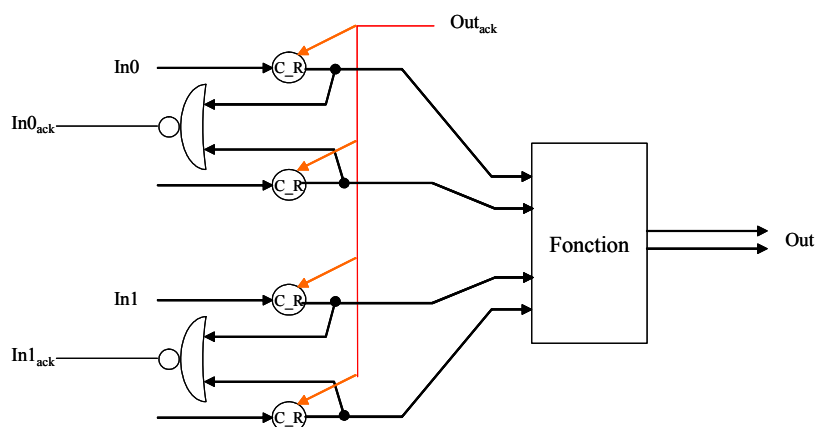


Figure 2.31 Synthèse du Join bufferisé en entrée

Le coût de la bufferisation en entrée vaut:

$$\text{Coût}_{\text{WCHB_entrée}}\{\text{Join}\} = \text{Coût}\{F_s\} + \text{Fourche}(I * B(\text{In}(0)) * L(\text{In}(0))) \\ + I * L(\text{In}(0)) * \text{muller2R} + I * L(\text{In}(0)) * \text{NOR}(B(\text{In}(0))) + I * \text{muller}(L(\text{In}(0)))$$

Pour une bufferisation en sortie, on obtient le circuit de la Figure 2.32.

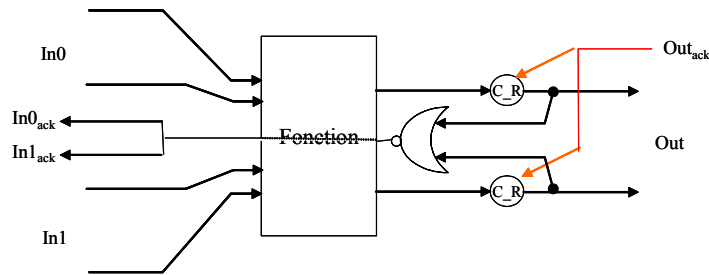


Figure 2.32 Synthèse du Join bufferisé en sortie

Le coût associé est:

$$\begin{aligned} \text{Coût}_{\text{WCHB_sortie}}\{\text{Join}\} &= \text{Coût}\{F_s\} + \text{Fourche}(B(\text{Out}(0)) * L(\text{Out}(0))) \\ &+ L(\text{Out}(0)) * \text{muller}2R + L(\text{Out}(0)) * \text{NOR}(B(\text{Out}(0))) + \text{muller}(L(\text{Out}(0))) \\ &+ \text{Fourche}(I) \end{aligned}$$

2.2.4.4 Merge

Le composant "merge" correspond à un multiplexage (avec exclusion mutuelle) de plusieurs entrées vers une sortie sans avoir recours à une commande. Il y a une détection qui est faite sur les canaux d'entrée: l'entrée qui présente une valeur valide active le composant, celui-ci envoie la valeur du canal activé vers la sortie. Contrairement au multiplexeur que nous allons présenter dans le paragraphe suivant, ce composant n'est pas commandé par un canal particulier. L'exclusion mutuelle doit être garantie en amont.

- $0 \leq q \leq Q-1$: il y a plusieurs gardes (ces gardes ne sont toutefois pas commandées).
- $0 \leq i < I-1$: il y a en tout I données entrantes
- $J = 1$: il y a une seule donnée sortante

L'équation de dépendance du composant "merge" s'écrit :

$$\text{Merge} = F_{\text{dep}} \left(\sum_{i=0}^{I-1} \text{In}(i); \sum_{q=0}^{Q-1} G(q); \text{Out}(0) = \sum_{i=0}^{I-1} \text{In}(i) \right)$$

La spécification CHP du "merge" est décrite comme suit:

```

[ @[ #In(0) => In(0) ? X ; Out(0) ! X ; break
  ...
  #In(i) => In(i) ? X ; Out(0) ! X ; break
  ...
  #In(I-1) => In(I-1) ? X ; Out(0) ! X ; break
  ]
loop ]
    
```

Les branches de la structure de choix sont homogènes, c'est-à-dire qu'elles réalisent la même opération (toutes les branches reçoivent une donnée et l'émettent en sortie). Par conséquent, le coût de chacune des branches est identique.

$$\text{Coût}\{\text{Merge}\} = \sum_{q=0}^{Q-1} G(q) \cdot \text{Coût}\{F_{\text{dep}}(G(q))\} = \text{Coût}\{F_{\text{dep}}(G(0))\}$$

Le composant "merge" est une structure de choix non commandée mais qui présente la propriété d'exclusion mutuelle. Il n'y a qu'une seule garde qui est valide à la fois. Pour ce genre de structure, il existe en tout trois possibilités de bufferisation. La première étant de bufferiser les entrées, la seconde les sorties intermédiaires (les sorties à l'intérieur des gardes) et enfin la bufferisation des sorties finales, c'est-à-dire à la sortie de la structure de choix. Les bufferisations en entrée et intermédiaires (Figure 2.33) sont identiques en raison du fait que la donnée entrante et la sortie intermédiaire sont de mêmes types.

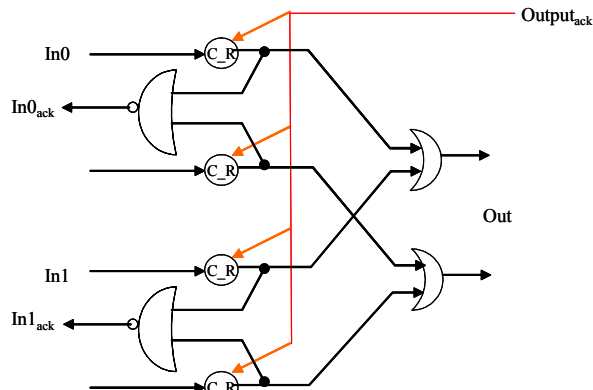


Figure 2.33 Merge bufferisé en entrée

Le coût de la bufferisation en entrée et intermédiaire s'écrit:

$$\begin{aligned} \text{Coût}_{\text{WCHB_entrée}}\{\text{Merge}\} &= \text{Coût}_{\text{WCHB_sortie_inter}}\{\text{Merge}\} = \\ & \text{Fourche}(Q * B(\text{In}(0)) * L(\text{In}(0))) + L(\text{In}(0)) * \text{muller}2R + L(\text{In}(0)) * \text{NOR}(B(\text{In}(0))) \\ & + \text{muller}(L(\text{In}(0))) + L(\text{Out}(0)) * \text{OR}(Q) \end{aligned}$$

Pour une bufferisation de la sortie finale du composant "merge", le circuit est présenté à la Figure 2.34.

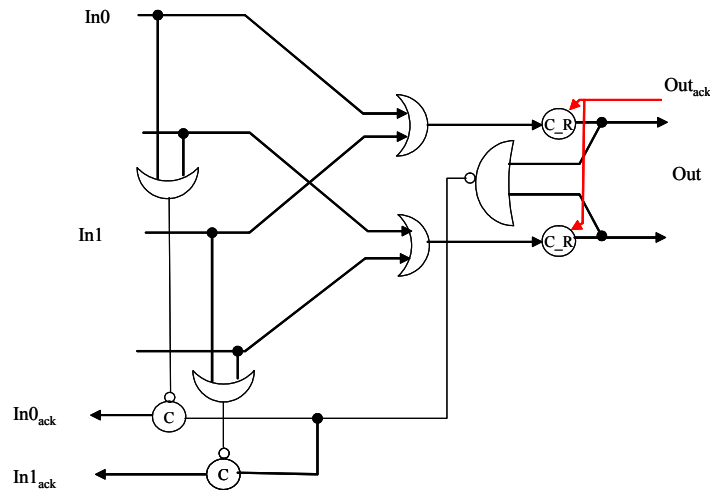


Figure 2.34 Merge bufferisé en sortie

L'équation de coût s'écrit alors :

$$\begin{aligned} \text{Coût}_{\text{WCH_sortie_finale}} \{\text{Merge}\} = & \\ & \text{Fourche}(B(\text{Out}(0))*L(\text{Out}(0))) + L(\text{Out}(0))*\text{muller}2R + L(\text{Out}(0))*\text{NOR}(B(\text{Out}(0))) \\ & + \text{muller}(L(\text{Out}(0))) + L(\text{Out}(0))*\text{OR}(Q) + \text{INV} + \text{muller}(2) + \text{Fourche}(Q) \\ & + L(\text{In}(0))*\text{OR}(B(\text{In}(0))) + \text{muller}(L(\text{Out}(0))) \end{aligned}$$

2.2.4.5 Multiplexeur

Le multiplexeur est une structure de choix qui est cette fois-ci contrôlée par un canal de commande. Le multiplexeur permet l'envoi d'une entrée parmi un certain nombre d'entrées vers une sortie unique. Une seule donnée à la fois peut être envoyée sur la sortie selon le principe d'exclusion mutuelle. La donnée qui est envoyée vers la sortie est commandée par le signal de contrôle de la structure de choix.

Le paramètres Q, I, J sont définis comme suit :

- $0 \leq q \leq Q - 1$: il y a en tout Q gardes commandées
- $0 \leq i < I - 1$: il y a en tout I données entrantes
- $J = 1$: il y a une seule donnée sortante

L'équation de dépendance du multiplexeur s'écrit :

$$\text{Mux} = F_{\text{dep}} \left(\sum_{i=0}^{I-1} \text{In}(i); \sum_{q=0}^{Q-1} G(q); \text{Out}(0) = \sum_{i=0}^{I-1} \text{In}(i) \right)$$

Le code CHP est :


```
[command? G;
@[ G(0) => In(0) ? X ; Out(0) ! X ; break
...
  G(q) => In(i) ? X ; Out(0) ! X ; break
...
  G(Q-1) => In(I-1) ? X ; Out(0) ! X ; break
]
loop ]
```

Comme pour le composant "merge", le multiplexeur peut être bufferisé de trois manières différentes : on peut le faire sur les données en entrée, sur les sorties intermédiaires (c'est-à-dire les sorties au niveau des branches) (Figure 2.35) ou enfin sur la sortie finale (Figure 2.36).

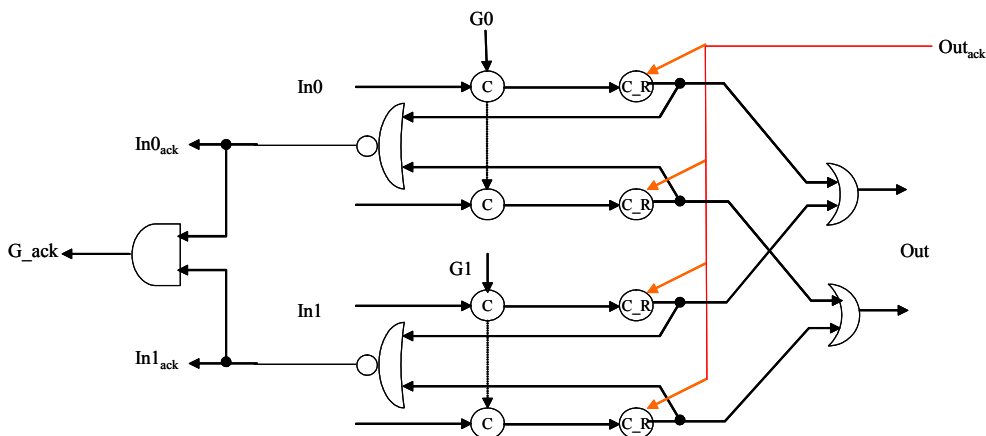


Figure 2.35 Multiplexeur bufferisé en entrée

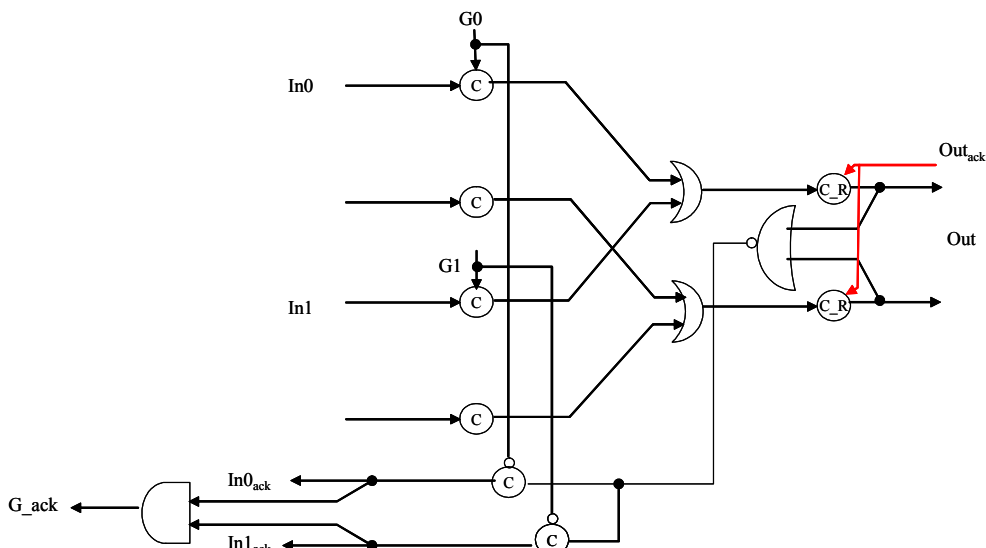


Figure 2.36 Multiplexeur bufferisé en sortie finale

Pour une bufferisation WCHB en entrée ou sur la sortie intermédiaire, le coût est identique, il vaut :

$$\begin{aligned} \text{Coût}_{\text{WCHB_entrée}}\{\text{Mux}\} &= \text{Coût}_{\text{WCHB_sortie_inter}}\{\text{Mux}\} = \\ & \text{muller}(\text{L}(\text{G})) + \text{L}(\text{In}(0)) * \text{muller}(2) + \text{Fourche}(\text{L}(\text{In}(0))*\text{B}(\text{In}(0))) \\ & + \text{Fourche}(\text{B}(\text{In}(0))*\text{L}(\text{In}(0))) + \text{L}(\text{In}(0))*\text{muller}2\text{R} + \text{L}(\text{In}(0))*\text{NOR}(\text{B}(\text{In}(0))) \\ & + \text{muller}(\text{L}(\text{In}(0))) + \text{AND}(\text{Q}) + \text{L}(\text{Out}(0))*\text{OR}(\text{Q}) \\ & + \text{Fourche}(\text{B}(\text{In}(0))*\text{L}(\text{In}(0))*(\text{Q} - 1)) \end{aligned}$$

Si on décide de bufferiser la sortie finale :

$$\begin{aligned} \text{Coût}_{\text{WCHB_sortie}}\{\text{Mux}\} &= \text{muller}(\text{L}(\text{G})) + \text{L}(\text{In}(0)) * \text{muller}(2) + \text{Fourche}(\text{L}(\text{In}(0))*\text{B}(\text{In}(0))) \\ & + \text{AND}(\text{Q}) + \text{L}(\text{Out}(0))*\text{OR}(\text{Q}) \\ & + \text{Fourche}(\text{B}(\text{Out}(0))*\text{L}(\text{Out}(0))) + \text{L}(\text{Out}(0))*\text{muller}2\text{R} + \text{L}(\text{Out}(0))*\text{NOR}(\text{B}(\text{Out}(0))) \\ & + \text{muller}(\text{L}(\text{Out}(0))) + \text{INV} + \text{muller}(2) + \text{Fourche}(\text{Q}) \end{aligned}$$

2.2.4.6 Démultiplexeur

Le Démultiplexeur est en quelque sorte l'architecture duale du multiplexeur. Il s'agit d'une structure de choix contrôlée par une commande qui reçoit une donnée unique et permet la propagation de celle-ci sur une sortie parmi un certain nombre de sorties. Une seule sortie peut émettre à la fois, l'émission sur la sortie est contrôlée par une commande qui garantit l'exclusion mutuelle.

- $0 \leq q \leq Q - 1$: il y a en tout Q commandes
- $I = 1$: il y a une seule donnée entrante
- $0 \leq j \leq J - 1$: il y a J données sortantes

L'équation de dépendance se simplifie :

$$\text{Demux} = F_{\text{dep}}(\text{In}(0); \sum_{q=0}^{Q-1} \text{G}(q); \sum_{j=0}^{J-1} \text{Out}(j) = \text{In}(0))$$

Le code CHP s'écrit :

```
[command? G;
@[ G(0) => In(0) ? X ; Out(0) ! X ; break
...
  G(q) => In(0) ? X ; Out(j) ! X ; break
...
  G(Q-1) => In(0) ? X ; Out(J-1) ! X ; break
]
loop ]
```

La bufferisation est là aussi possible en entrée (Figure 2.37) et sur les sortie finales (Figure 2.38).

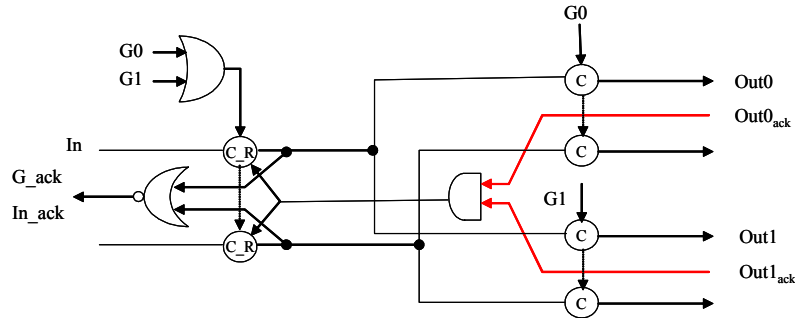


Figure 2.37 Demultiplexeur bufferisé en entrée

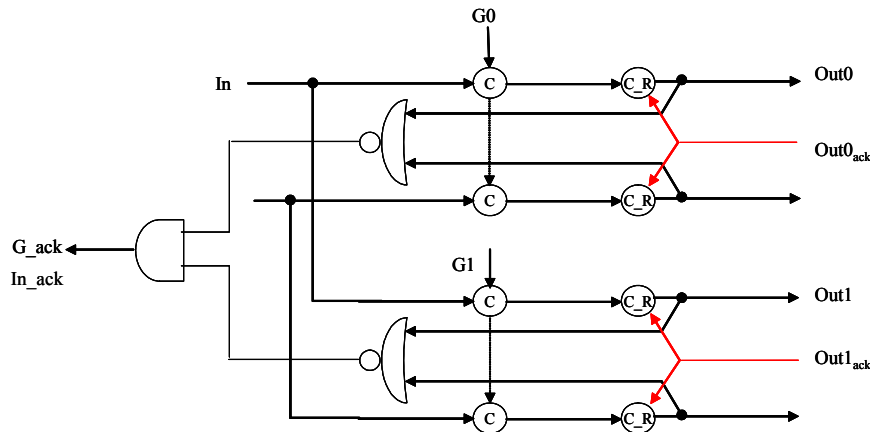


Figure 2.38 Démultiplexeur bufferisé en sortie

Pour une bufferisation en entrée, le coût vaut :

$$\begin{aligned} \text{Coût}_{\text{WCHB_sortie_entree}} \{ \text{Demux} \} = & \\ & \text{muller}(L(G)) + \text{OR}(Q) + \text{Fourche}(L(\text{In}(0)) * B(\text{In}(0))) + \text{Fourche}(L(\text{In}(0)) * B(\text{In}(0))) \\ & + L(\text{In}(0)) * \text{muller}3R + L(\text{In}(0)) * \text{NOR}(B(\text{In}(0))) + \text{muller}(L(\text{In}(0))) + L(\text{In}(0)) * \text{muller}(2) \\ & + \text{AND}(Q) + L(\text{In}(0)) * \text{Fourche}(Q) \end{aligned}$$

Dans le cas d'une bufferisation des sorties finales :

$$\begin{aligned} \text{Coût}_{\text{WCHB_sortie}} \{ \text{Demux} \} = & \\ & \text{muller}(L(G)) + L(\text{In}(0)) * \text{muller}(2) + \text{Fourche}(L(\text{In}(0)) * B(\text{In}(0))) \\ & + \text{Fourche}(B(\text{In}(0)) * L(\text{In}(0))) + L(\text{In}(0)) * \text{muller}2R + L(\text{In}(0)) * \text{NOR}(B(\text{In}(0))) \\ & + \text{muller}(L(\text{In}(0))) + \text{AND}(Q) + L(\text{In}(0)) * \text{Fourche}(Q) + \text{AND}(Q) \end{aligned}$$

2.3 Estimation de la consommation d'énergie dynamique

L'estimation structurelle telle qu'elle a été présentée dans le paragraphe précédent permet d'attribuer un coût aux sous parties d'un circuit, notamment aux processus. A chaque processus d'un circuit asynchrone est attribué une équation de coût. Cette équation de coût utilise des opérateurs booléens pour distinguer les différentes gardes de la structure.

L'estimation dynamique consiste à utiliser ces estimations structurelles, plus précisément à les sommer, lors de l'exécution d'un programme donné.

Dans un premier temps, on indique comment ces équations de coût structurel sont associées au réseau de pétri d'un processus puis on explique comment le coût dynamique est calculé en utilisant cette information.

2.3.1 Labellisation du réseau de pétri

Le coût structurel est établi pour un processus donné. Il existe au final une équation de coût structurel par processus. Cette équation de coût structurel introduit des opérateurs booléens si le processus présente des structures de choix.

Il existe un réseau de pétri par processus. A chaque réseau de pétri représentant un processus est associé une équation de coût structurel. Cette équation est disposée à la dernière place du réseau de pétri représentant le processus. Cette place peut contenir deux types d'instructions : l'instruction "break" si on termine ou l'instruction "loop" si on reboucle (Figure 2.39).

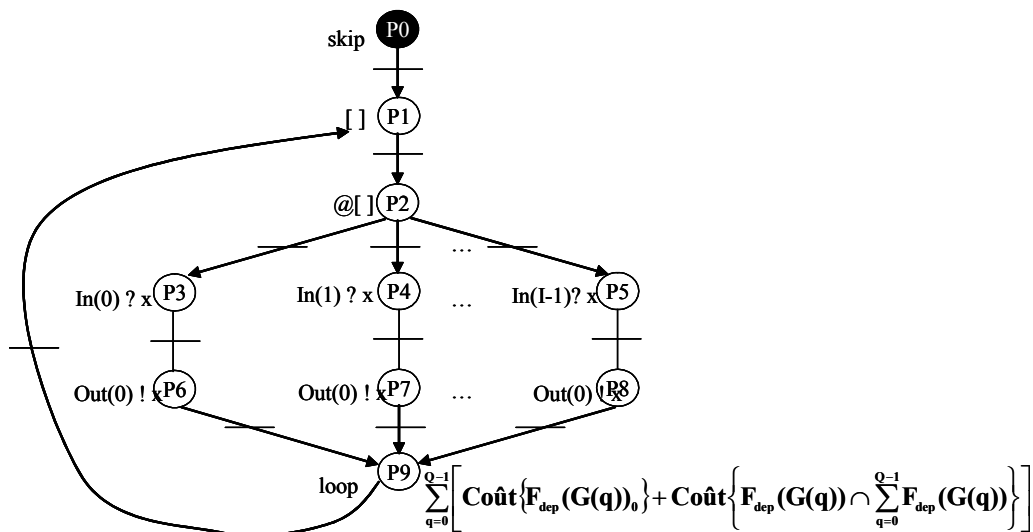


Figure 2.39 Disposition du coût structurel sur le réseau de Pétri

On choisit de mettre l'équation de coût à cette place car c'est la place qui est toujours atteinte lorsque le réseau de pétri est simulé. Par ailleurs, lorsqu'on atteint cette place pendant une simulation, on a à disposition toutes les informations nécessaires sur le chemin qui a été emprunté, les canaux d'entrée et les canaux de sortie utilisés. Cette équation relate toutes les informations dans toutes les branches de la structure de choix. La présence d'opérateurs booléens permet à terme de comptabiliser seulement le coût de la séquence de marquages qui définit l'exécution qui vient d'avoir lieu dans le processus.

2.3.2 Calcul du coût dynamique

Une fois que les coûts structurels de tous les processus ont été estimés puis disposés sur les réseaux de pétri aux endroits qui leur sont réservés, il ne reste alors plus qu'à simuler les réseaux de pétri. Le coût dynamique est finalement obtenu en sommant les coûts structurels des processus dans une variable globale. Il est néanmoins possible de sommer méthodiquement ces coûts structurels de sorte à obtenir un degré de granularité plus ou moins fin en fonction des informations que le concepteur souhaite recueillir. Par exemple en utilisant des variables locales à des processus et composants.

La granularité la plus fine de l'estimation de la consommation d'énergie que le concepteur puisse observer est l'estimation au niveau de chaque commande gardée de la structure de choix d'un processus.

On peut établir le coût dynamique de chaque commande gardée d'un processus donné:

$$\text{Coût}_{\text{dynamique}} \{F_{\text{dep}}(G(q))\} = \sum_{s=0}^{S-1} \text{Coût}_{\text{structurel}} \{F_{\text{dep}}(G(q))\}$$

Où la variable "s" correspond au nombre d'exécutions de la commande gardée G(q).

On peut noter qu'en présence d'un processus ayant une seule garde TRUE (Q=1), c'est-à-dire qu'il ne s'agit pas d'une structure de choix traditionnelle avec plusieurs gardes mutuellement exclusives, l'estimation dynamique de cette unique garde correspond à l'estimation dynamique du processus entier.

Il est ensuite possible d'augmenter la granularité de l'estimation de la consommation d'énergie. Le niveau directement au dessus du niveau des commandes gardées est l'estimation dynamique des processus. Pour chaque processus, on peut obtenir des informations sur sa consommation d'énergie. Le coût dynamique d'un processus s'écrit :

$$\text{Coût}_{\text{dynamique}} \{\text{process}\} = \sum_{q=0}^{Q-1} \text{Coût}_{\text{dynamique}} \{F_{\text{dep}}(G(q))\} = \sum_{q=0}^{Q-1} \sum_{s=0}^{S-1} \text{Coût}_{\text{structurel}} \{F_{\text{dep}}(G(q))\}$$

Le coût d'un processus est finalement la somme des coûts de toutes les commandes gardées exécutées lors d'une simulation donnée.

Le niveau hiérarchique qui vient juste au dessus du processus est le composant. Un composant en CHP est un ensemble de processus qui communiquent les uns avec les autres à l'aide d'une connexion de canaux point à point. L'estimation de la consommation dynamique d'un composant est finalement la somme de la consommation de tous les processus qu'il contient. Ceci est vrai car on utilise les propriétés propres à l'asynchrone, à savoir qu'il n'y a pas de dissipation d'énergie en dehors d'un traitement effectif d'un processus. Si un processus n'est pas sollicité, alors sa consommation est nulle.

Le coût d'un composant se calcule très facilement à l'aide de la formule suivante :

$$\text{Coût}_{\text{dynamique}} \{\text{component}\} = \sum_{\text{Nb process}} \text{Coût}_{\text{dynamique}} \{\text{process}\} = \sum_{\text{Nb process}} \sum_{q=0}^{Q-1} \text{Coût}_{\text{dynamique}} \{F_{\text{dep}}(G(q))\}$$

On a établi ainsi l'estimation de la consommation d'énergie dynamique d'un circuit asynchrone.

2.4 Application à une architecture hétérogène

Nous proposons dans ce paragraphe d'appliquer le formalisme de l'estimation de la consommation d'énergie à un exemple qu'on qualifie d'hétérogène. Le code CHP correspondant s'écrit:

```

[Cmd ? G;
  @[ G= "00"[2] =>
    In(0) ? X; Out(0) ! X;
    break
  G= "01"[2] =>
    In(1) ? X; Out(0) ! X, Out(1) ! X;
    break
  G="10"[2] or G = "11"[2] =>
    In(0) ? X, In(1) ? Y; Out(1) ! (X and Y) ;
    break
  ]
loop ]
    
```

On parle d'une architecture hétérogène parce qu'elle présente une structure de choix à plusieurs branches avec des instructions et des dépendances de données variées. Effectivement, dans la première branche l'entrée lue est envoyée sur une sortie (cela reproduit le fonctionnement du "latch"), dans la seconde branche l'entrée lue est envoyée sur deux sorties différentes (cela reproduit le comportement d'une fourche) et enfin dans la troisième branche deux entrées sont lues, combinées par une fonction "and" logique et le résultat est envoyé sur une sortie (comportement du "join"). De plus, une entrée figure dans plusieurs branches, de même qu'une donnée sortante apparaît dans plusieurs branches. On a donc des intersections aussi bien en entrée qu'en sortie (comme il est question dans le multiplexeur et le démultiplexeur).

Il y a en tout trois gardes G(0), G(1) et G(2) qui correspondent aux trois branches de la structure de choix. La garde G(2) est une garde composée (G="10"[2] or G="11"[2]). On est donc en présence d'un exemple qui regroupe la complexité de toutes les architectures de base. L'équation de dépendance de cette architecture déduite de l'équation générale Équation 2.1 est:

$$\text{hétérogène} = F_{\text{dep}} \left(\sum_{i=0}^1 \text{In}(i); \sum_{q=0}^2 G(q); \sum_{j=0}^1 \text{Out}(j) = F \left(\sum_{i=0}^1 \text{In}(i) \right) \right)$$

Il y a en tout :

- 3 gardes correspondant aux trois branches de la structure de choix : $0 \leq q \leq 2$
- 2 entrées In(0) et In(1) : $0 \leq i \leq 1$
- 2 sorties Out(0) et Out(1) : $0 \leq j \leq 1$

Toutes les données sont codées en MR[2][1], c'est-à-dire un seul digit double rails. Le réseau de Pétri du composant hétérogène est présenté à la Figure 2.40.

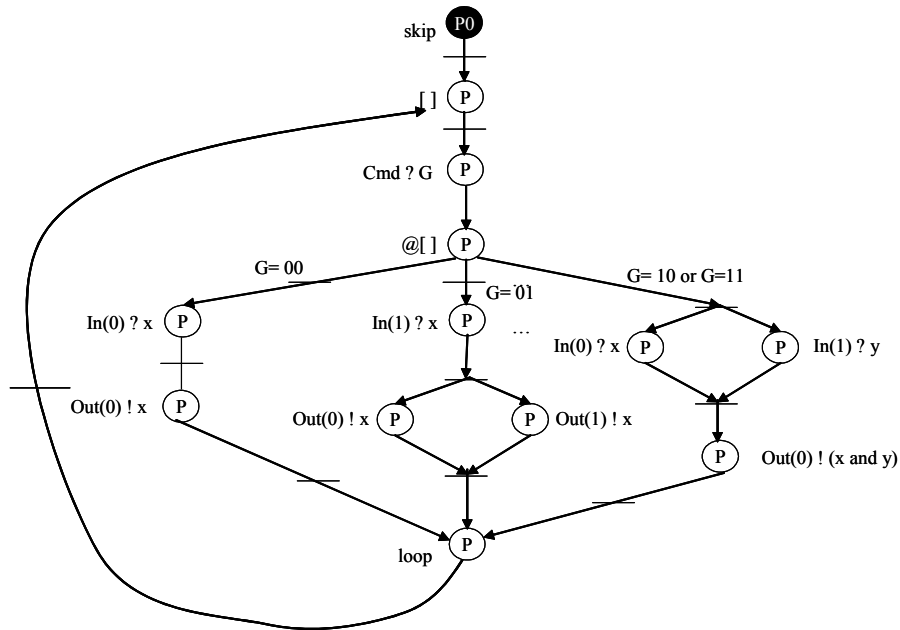


Figure 2.40 Réseau de Pétri du composant hétérogène

Etablissons les coûts structurels des trois gardes en utilisant l'Équation 2.2.

Coût structurel de la garde G(0)

Le coût structurel de la commande gardée G(0) est égal au coût intrinsèque de la garde (ici il s'agit du coût d'un latch) (Coût{I} de la Figure 2.41), additionné au coût d'intersection de la garde G(0) avec les gardes G(1) et G(2). Ce coût d'intersection correspond au partage de la commande G avec les 2 autres gardes (Coût{IV}; partage de l'entrée In(0) avec la garde G(2) (Coût{V}+Coût{VII})) et enfin il y a la sortie Out(0) qui est partagée avec la garde G(1) (Coût{IX}).

$$\text{Coût}_{\text{structurel}}\{F_{\text{dep}}(G(0))\} = \text{Coût}_{\text{structurel}}\{F_{\text{dep}}(G(0))_0\} + \text{Coût}_{\text{structurel}}\{F_{\text{dep}}(G(0)) \cap (F_{\text{dep}}(G(1)), F_{\text{dep}}(G(2)))\}$$

$$\text{Coût}_{\text{structurel}}\{F_{\text{dep}}(G(0))\} = (\text{Coût}\{I\}) + (\text{Coût}\{IV\} + \text{Coût}\{V\} + \text{Coût}\{VII\} + \text{Coût}\{IX\})$$

$$\text{Coût}\{I\} = \text{muller}(L(G)) + \text{Fourche}(B(\text{In}(0))*L(\text{In}(0))) +$$

$$+ \text{Fourche}(B(\text{In}(0))*L(\text{In}(0))) +$$

$$L(\text{In}(0))*\text{muller}(2) + L(\text{In}(0))*\text{muller}2R + L(\text{In}(0))*\text{NOR}(B(\text{In}(0)))$$

$$\text{Coût}\{I\} = \text{muller}(2) + \text{Fourche}(2) + \text{Fourche}(2) + \text{muller}(2) + \text{muller}2R + \text{NOR}(2)$$

$$\text{Coût}\{IV\} = \text{AND}(3)$$

$$\text{Coût}\{V\} = \text{AND}(2)$$

$$\text{Coût}\{VII\} = L(\text{In}(0))*\text{Fourche}(2) = \text{Fourche}(2)$$

$$\text{Coût}\{IX\} = L(\text{Out}(0))*\text{OR}(2) = \text{OR}(2)$$

$$\begin{aligned}
 \text{Coût}_{\text{structurel}} \{F_{\text{dep}}(G(0))\} &= 2 * \text{muller}(2) + 3 * \text{Fourche}(2) + \text{muller}2R + \text{NOR}(2) \\
 &+ \text{AND}(3) + \text{OR}(2) + \text{AND}(2) \\
 &= 2 * 2 + 3 * (2 \text{ div } 8) + 2 + 5/3 + \text{sup}|\log_2 3| * \left(\frac{5/3 + 4/3}{2}\right) + 5/3 + 4/3 \\
 &= 41/3 \text{ Transitions}
 \end{aligned}$$

Coût structurel de la garde G(1)

Le coût structurel de la commande gardée G(1) est établi de manière identique au coût de la garde G(0). Il y a le coût intrinsèque (identique au composant "fork") (Coût{II} de la Figure 2.41) et le coût d'intersection avec les autres gardes. On note le partage de la commande globale avec les 2 autres gardes (Coût{IV}), il y a l'entrée In(1) (Coût{VI}+Coût{VIII}) et la sortie Out(1) (Coût{X}) qui sont partagées avec la garde G(2) et enfin la sortie Out(0) qui est utilisée dans la garde G(0) (Coût{IX}).

$$\begin{aligned}
 \text{Coût}_{\text{structurel}} \{F_{\text{dep}}(G(1))\} &= \text{Coût}_{\text{structurel}} \{F_{\text{dep}}(G(1))_0\} + \text{Coût}_{\text{structurel}} \{F_{\text{dep}}(G(1)) \cap (F_{\text{dep}}(G(0)), F_{\text{dep}}(G(2)))\} \\
 \text{Coût}_{\text{structurel}} \{F_{\text{dep}}(G(1))\} &= (\text{Coût}\{\text{II}\}) + (\text{Coût}\{\text{IV}\}) + \text{Coût}\{\text{VI}\} + \text{Coût}\{\text{VIII}\} + \text{Coût}\{\text{IX}\} + \text{Coût}\{\text{X}\}
 \end{aligned}$$

$$\begin{aligned}
 \text{Coût}\{\text{II}\} &= \text{muller}(L(G)) + \text{Fourche}(B(\text{In}(1)) * L(\text{In}(1))) \\
 &+ \text{Fourche}(B(\text{In}(1)) * L(\text{In}(1))) \\
 &+ L(\text{In}(1)) * \text{muller}(2) + L(\text{In}(1)) * \text{muller}2R + L(\text{In}(1)) * \text{NOR}(B(\text{In}(1))) \\
 &+ \text{Fourche}(2) \quad /* \text{fourche de la sortie qui va vers IX et vers X} */ \\
 \text{Coût}\{\text{II}\} &= \text{muller}(2) + \text{Fourche}(2) + \text{Fourche}(2) + \text{muller}(2) + \text{muller}2R + \text{NOR}(2) + \text{Fourche}(2) \\
 \text{Coût}\{\text{IV}\} &= \text{AND}(3) \\
 \text{Coût}\{\text{VI}\} &= \text{AND}(2) \\
 \text{Coût}\{\text{VIII}\} &= L(\text{In}(1)) * \text{Fourche}(2) = \text{Fourche}(2) \\
 \text{Coût}\{\text{IX}\} &= L(\text{Out}(0)) * \text{OR}(2) = \text{OR}(2) \\
 \text{Coût}\{\text{X}\} &= L(\text{Out}(1)) * \text{OR}(2) = \text{OR}(2)
 \end{aligned}$$

$$\begin{aligned}
 \text{Coût}_{\text{structurel}} \{F_{\text{ep}}(G(1))\} &= 2 * \text{muller}(2) + 4 * \text{Fourche}(2) + \text{muller}2R + \text{NOR}(2) \\
 &+ \text{AND}(3) + \text{AND}(2) + 2 * \text{OR}(2) \\
 &= 2 * 2 + 4 * (2 \text{ div } 8) + 2 + 5/3 + \text{sup}|\log_2 3| * \left(\frac{5/3 + 4/3}{2}\right) + 4/3 + 2 * 5/3 \\
 &= 46/3 \text{ transitions}
 \end{aligned}$$

Coût structurel de la garde G(2)

Pour cette dernière commande gardée, le coût intrinsèque correspond au coût d'un "join" utilisant l'opérateur AND_MR2 (porte AND asynchrone double rail) (Coût{III} de la Figure 2.41). Le coût d'intersection avec les autres gardes correspond au partage de la commande globale avec les autres gardes (Coût{IV}), l'entrée In(0) est partagée avec la garde G(0) (Coût{V}+Coût{VII}), l'entrée In(1) (Coût{VI}+Coût{VIII}) et la sortie Out(1) (Coût{X}) sont utilisées dans G(1).

$$\text{Coût}_{\text{structurel}}\{F_{\text{dep}}(G(2))\} = \text{Coût}_{\text{structurel}}\{F_{\text{dep}}(G(2))_0\} + \text{Coût}_{\text{structurel}}\{F_{\text{dep}}(G(2)) \cap (F_{\text{dep}}(G(0)), F_{\text{dep}}(G(1)))\}$$

$$\text{Coût}_{\text{structurel}}\{F_{\text{dep}}(G(2))\} = (\text{Coût}\{\text{III}\}) + \left(\begin{array}{l} \text{Coût}\{\text{IV}\} + \text{Coût}\{\text{V}\} + \text{Coût}\{\text{VII}\} + \text{Coût}\{\text{VI}\} \\ + \text{Coût}\{\text{VIII}\} + \text{Coût}\{\text{X}\} \end{array} \right)$$

$$\text{Coût}\{\text{III}\} = \text{muller}(L(G)) + \text{OR}(2) + \text{Fourche}(B(\text{In}(0))*L(\text{In}(0)) + B(\text{In}(1))*L(\text{In}(1)))$$

$$+ \text{Fourche}(B(\text{In}(0))*L(\text{In}(0)) + B(\text{In}(1))*L(\text{In}(1)))$$

$$+ L(\text{In}(0))*\text{muller}(2) + L(\text{In}(0))*\text{muller}2R + L(\text{In}(0))*\text{NOR}(2)$$

$$+ L(\text{In}(1))*\text{muller}(2) + L(\text{In}(1))*\text{muller}2R + L(\text{In}(1))*\text{NOR}(2)$$

$$+ \text{muller}(2)$$

$$+ \text{Coût}\{\text{AND_MR2}\}$$

$$\text{Coût}\{\text{III}\} = \text{muller}(2) + \text{OR}(2) + \text{Fourche}(4) + \text{Fourche}(4)$$

$$+ \text{muller}(2) + \text{muller}2R + \text{NOR}(2)$$

$$+ \text{muller}(2) + \text{muller}2R + \text{NOR}(2)$$

$$+ \text{muller}(2)$$

$$+ \text{muller}(2) + \text{OR}(3) \quad /* \text{Coût}(wc) \text{ de la porte AND_MR2} */$$

$$\text{Coût}\{\text{IV}\} = \text{AND}(3)$$

$$\text{Coût}\{\text{V}\} = \text{AND}(2)$$

$$\text{Coût}\{\text{VII}\} = L(\text{In}(0))*\text{Fourche}(2) = \text{Fourche}(2)$$

$$\text{Coût}\{\text{VI}\} = \text{AND}(2)$$

$$\text{Coût}\{\text{VIII}\} = L(\text{In}(1))*\text{Fourche}(2) = \text{Fourche}(2)$$

$$\text{Coût}\{\text{X}\} = \text{OR}(2)$$

$$\text{Coût}_{\text{structurel}}\{F_{\text{dep}}(G(2))\} = 5*\text{muller}(2) + 2*\text{Fourche}(4) + 2*\text{muller}2R + 2*\text{NOR}(2)$$

$$+ \text{AND}(3) + 2 * \text{AND}(2) + 2*\text{Fourche}(2) + 2 * \text{OR}(2) + \text{OR}(3)$$

$$= 5 * 2 + 2 * (4\text{div}8) + 2 * 2 + 2 * 5/3 + \text{sup}|\log_2 3| * \left(\frac{5/3 + 4/3}{2}\right) + 2 * 4/3 + 2 * (2\text{div}8)$$

$$+ 2 * 5/3 + \text{sup}|\log_2 3| * \left(\frac{5/3 + 4/3}{2}\right)$$

$$= 88/3 \text{ transitions}$$

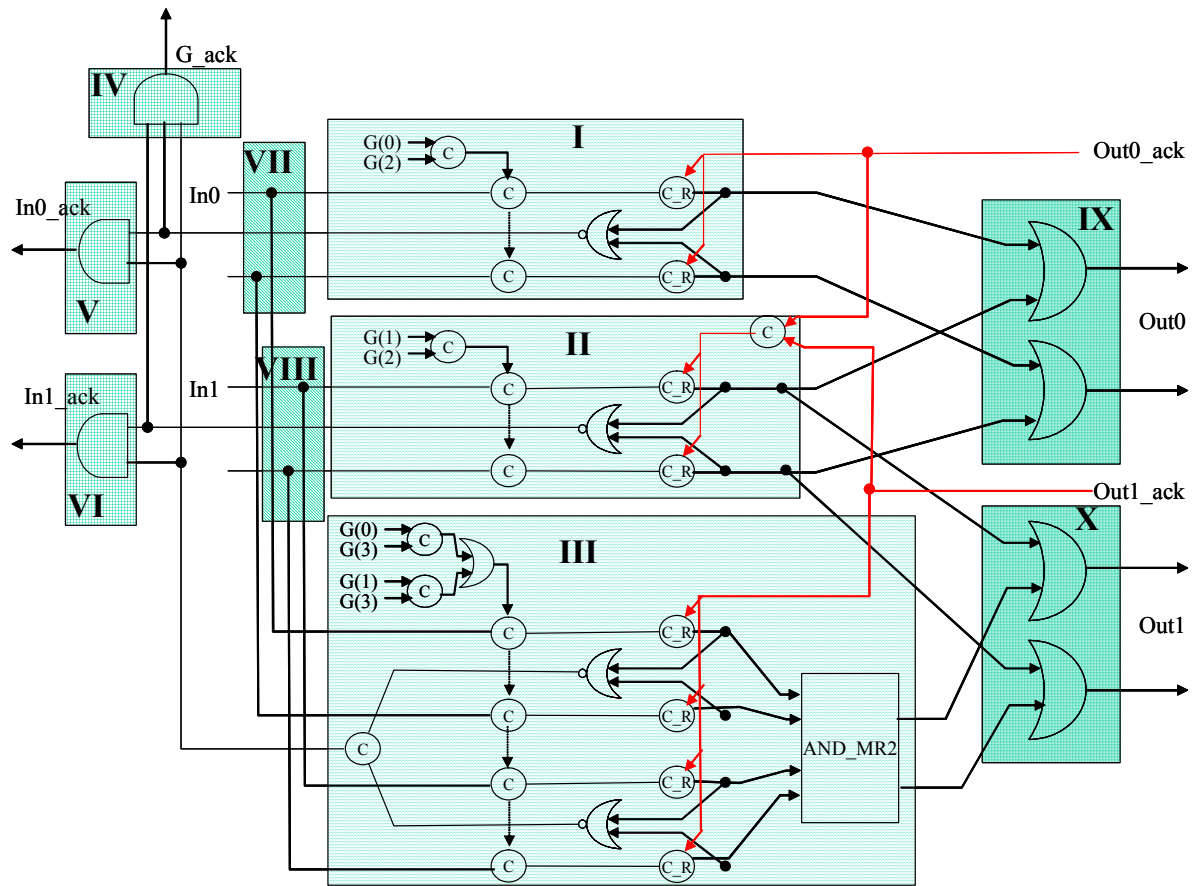


Figure 2.41 Synthèse du composant hétérogène

Le coût structurel global de cette architecture hétérogène s'écrit donc :

$$\text{Coût}_{\text{structurel}}\{\text{hétérogène}\} = G(0) \cdot \frac{41}{3} + G(1) \cdot \frac{46}{3} + G(2) \cdot \frac{88}{3}$$

Cette équation de coût est placée dans le réseau de pétri représentant le processus à la dernière place atteinte (Figure 2.42).

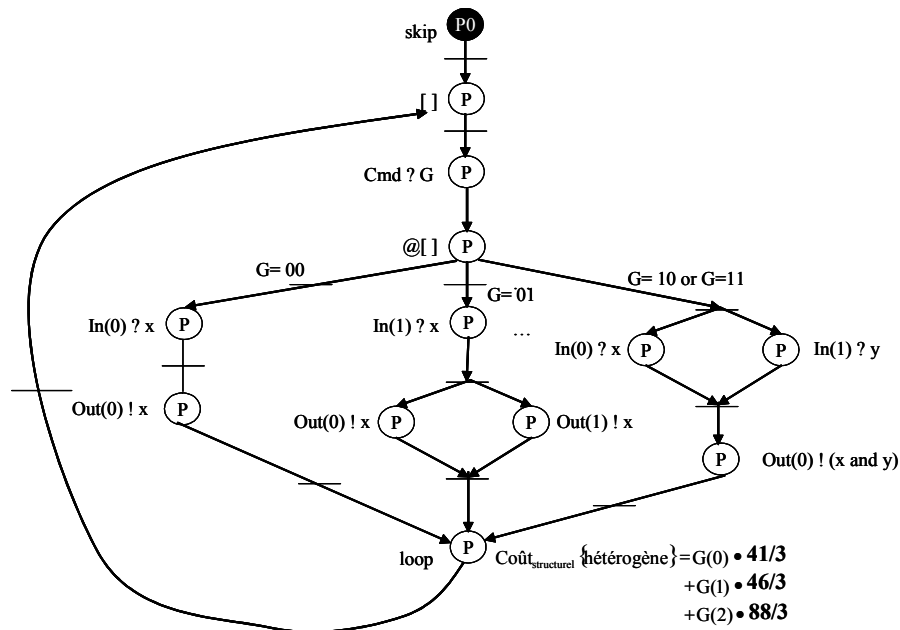


Figure 2.42 Labellisation du réseau de pétri avec le coût structurel

On veut maintenant estimer la consommation dynamique de cette architecture pour une simulation donnée. Faisons l'hypothèse que la simulation consiste en l'exécution de 100 fois G(0), 50 fois G(1) et 15 fois G(2). Le nombre d'exécution de ce processus est de 165 fois (la variable "s" définie dans le paragraphe précédent).

Etablissons le coût dynamique pour cette simulation:

$$\text{Coût}_{\text{dynamique}} \{\text{heterogene}\} = \sum_{s=0}^{164} G(0) \cdot \frac{41}{3} + G(1) \cdot \frac{46}{3} + G(2) \cdot \frac{88}{3}$$

$$\text{Coût}_{\text{dynamique}} \{\text{heterogene}\} = 100 \cdot \frac{41}{3} + 50 \cdot \frac{46}{3} + 15 \cdot \frac{88}{3}$$

On a donc un coût dynamique de la structure après simulation qui vaut :

$$\text{Coût}_{\text{dynamique}} \{\text{heterogene}\} = \frac{7720}{3} \approx 2573 \text{ Transitions}$$

2.5 Conclusion

Une description exhaustive de notre méthodologie d'estimation de la consommation d'énergie en terme de nombre de commutations a été proposée dans ce chapitre. Celle-ci est réalisée en deux étapes principales que nous avons définies par les appellations estimation structurelle et estimation dynamique. La présentation de cette méthodologie a été accompagnée d'exemples pour expliquer les calculs d'obtention de la consommation d'énergie en terme de nombre de commutations. Un exemple significatif que nous avons qualifié d'hétérogène a permis d'illustrer la méthodologie d'estimation de la consommation d'énergie.

La méthodologie d'estimation de la consommation d'énergie sera à terme intégrée dans le flot de conception TAST. Les spécifications de cet outil sont présentées au chapitre suivant.

Chapitre 3

Spécification des Outils d'Estimation

La réalisation de circuits intégrés asynchrones à faible consommation d'énergie requiert le développement d'outils adaptés. La volonté de concevoir des circuits à faible consommation d'énergie nécessite la mise en place d'outils permettant d'obtenir des informations sur la consommation d'énergie des circuits tout au long du processus de conception. Dans le cadre des travaux de recherche de cette thèse, nous avons développé un outil qui permet d'estimer l'activité d'un circuit et nous avons spécifié un second outil permettant d'estimer la consommation d'énergie d'un circuit en terme de nombre de transitions.

Nous présentons au paragraphe 3.1 de manière succincte le flot complet de conception TAST. Cette présentation permet de situer les estimateurs d'activité et de consommation d'énergie dans le flot de conception. Les caractéristiques de l'estimateur d'activité telles qu'elles ont été spécifiées et implémentées dans l'outil TAST sont explicitées à la section 3.2. La section 3.3 présente les spécifications de l'estimateur de la consommation d'énergie. Ce dernier n'a pas été implémenté pour l'instant.

3.1 Présentation du flot TAST

Les outils d'estimation d'activité et de consommation d'énergie s'inscrivent dans le cadre du développement de TAST [TAST-02] (Tima Asynchronous Synthesis Tools) : outil de synthèse des circuits asynchrones développé par le groupe CIS du laboratoire TIMA. Les possibilités importantes et variées proposées par TAST pour la conception de circuits asynchrones, tant sur l'aspect simulation (simulation fonctionnelle en VHDL ou en C), que synthèse (synthèse QDI et micropipeline), intègrent désormais la possibilité d'estimer l'activité d'un circuit ainsi que la consommation d'énergie (Figure 3.1).

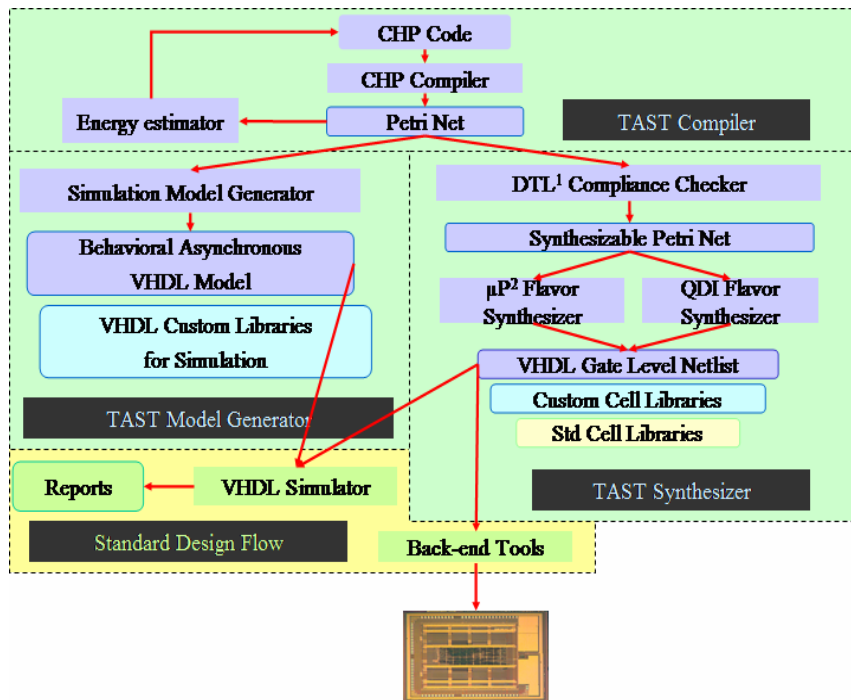


Figure 3.1 Flot TAST

L'outil TAST se présente comme une interface principale (Figure 3.2) depuis laquelle les outils qu'il intègre peuvent être appelés. L'interface est composée de 3 cadrans : le cadran en haut à gauche correspond au répertoire de projet courant, le cadran en haut à droite est un éditeur de texte des programmes CHP et enfin en bas on trouve les résultats des divers compilations. Une barre de menu permet d'ouvrir ou de créer des projets, elle permet aussi d'appeler les divers outils de TAST que nous présentons dans la suite de cette section.

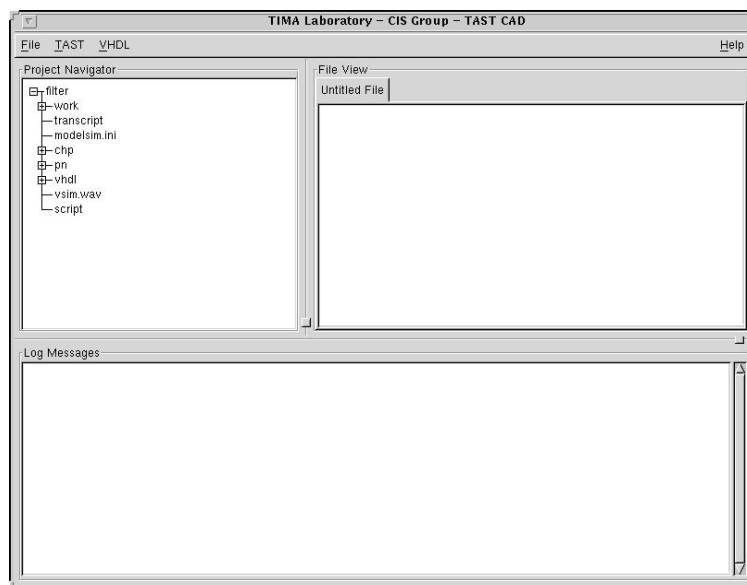


Figure 3.2 Interface TAST

3.1.1 Le compilateur CHP

Une première étape du flot de conception consiste à générer la représentation intermédiaire de la spécification CHP du circuit à concevoir. Cette représentation intermédiaire est un réseau

de pétri. Pour ce faire, on utilise un compilateur CHP qui se nomme Chp2Pn qui est utilisable depuis l'interface principale de l'outil (Figure 3.3).

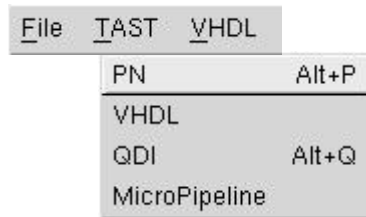


Figure 3.3 Compilateur CHP

Cet arbre intermédiaire est une représentation indispensable. Tous les outils que nous développons partent de cette forme intermédiaire.

3.1.2 Le simulateur de Pétri Net

Le simulateur est composé d'une fenêtre principale (à gauche sur la Figure 3.4): cette fenêtre permet d'ouvrir les réseaux de pétri des processus du circuit (les 2 fenêtres de droite de la Figure 3.4), elle permet aussi de lancer ou d'arrêter une simulation.

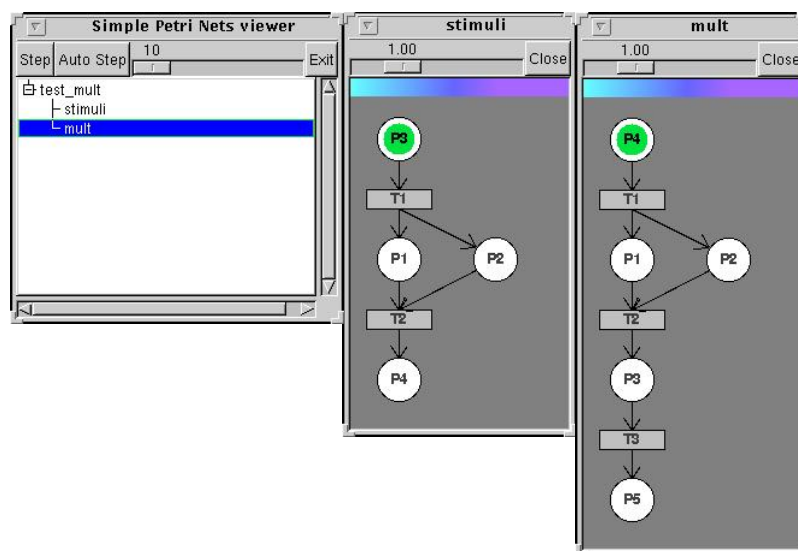


Figure 3.4 Simulateur de réseau de pétri

La simulation du réseau de pétri consiste au parcours du réseau à l'aide de jetons. Ces jetons parcourent les places du réseau lorsque les transitions autorisent le passage d'une place à la suivante. Un jeton peut prendre différentes couleurs (vert, rouge) en fonction de l'état de la place (autorisation de communication ou état bloqué).

Les estimations d'activité et de consommation d'énergie sont réalisées lors de la simulation du réseau de pétri.

3.1.3 Le convertisseur CHP vers VHDL fonctionnel

TAST permet de convertir une spécification CHP d'un circuit en une description VHDL (Figure 3.5). Cette conversion se fait à l'aide de l'outil Chp2vhdl. Le VHDL ainsi obtenu est une description purement fonctionnelle, elle ne peut bien entendu pas être synthétisée à l'aide

d'outils de synthèse commerciaux. Par contre, cette description VHDL peut être simulée à l'aide d'outils de simulation commerciaux (synopsys ou modelsim par exemple). C'est un outil important qui permet de tester le fonctionnement correct d'un circuit, dans un environnement standard de conception de circuits synchrones.

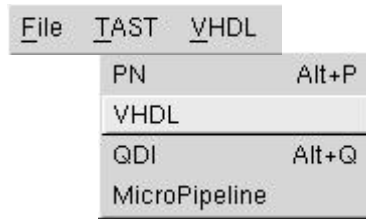


Figure 3.5 Convertisseur Chp2vhdl

3.1.4 L'outil de synthèse Quasi Insensible aux Délais

TAST inclut un outil de synthèse de circuits asynchrones Quasi Insensible aux Délais (Figure 3.6). Cet outil permet, à partir d'une spécification CHP, de générer une netlist composée de portes de mullers TAL [MAUR-03], de portes standard traditionnelles etc. Cette netlist est de même nature que les netlist obtenues par les outils de synthèse de circuits synchrones (Design Analyzer de synopsys ou Leonardo de mentor graphics).

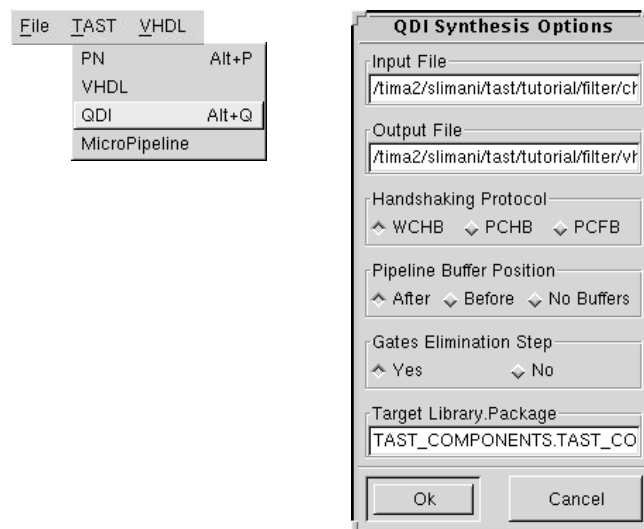


Figure 3.6 Synthèse QDI

Une interface de l'outil de synthèse QDI propose au concepteur certaines options de synthèse: le protocole de communication (WCHB, PCHB, PCFB), la disposition des buffers (en entrée ou en sortie) etc.

3.1.5 L'outil de synthèse Micropipeline

TAST propose aussi un outil qui permet de synthétiser une spécification CHP en circuits asynchrones micropipelines (Figure 3.7). Les architectures micropipelines sont explicitées dans le Chapitre 1.

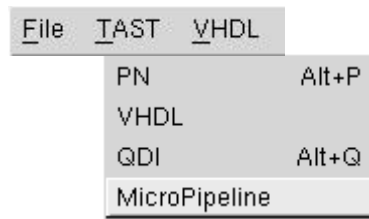


Figure 3.7 Synthèse Micropipeline

3.2 Estimateur d'activité

L'estimateur d'activité est un outil qui permet d'observer l'activité d'un circuit lors d'une simulation donnée. Il permet plus précisément d'analyser la répartition de l'activité dans un circuit. Les parties du circuit les plus sollicitées pourront être ainsi identifiées et seront optimisées au besoin par le concepteur. L'estimateur d'activité est l'une des nouveautés que nous proposons dans le cadre de cette thèse. C'est un outil indispensable à la conception de circuits asynchrones à faible consommation d'énergie. Il s'apparente fortement aux outils de couverture de code et de profiling des outils commerciaux. Mentor Graphics propose dans Modelsim une option *-coverage* qui permet de faire de la couverture de code. Il s'agit tout simplement de compter l'exécution des lignes de code d'un circuit décrit en langage de haut niveau (VHDL ou Verilog) lors d'une simulation donnée. A chaque ligne est associée un compteur, celui-ci est incrémenté lorsque la ligne de code est exécutée. Une démonstration est proposée en [MODE].

Synopsys propose un outil similaire permettant de faire de la couverture de code, il s'agit de CoverMeter[COVE]. Il existe enfin d'autres outils qui réalisent de la couverture de code mais basée cette fois-ci sur des métriques de calcul et non pas sur un compteur de lignes [FALL-01].

Nous verrons un peu plus loin dans cette section que notre outil est plus précis que les outils précédemment cités. Il permet en effet de décomposer les lignes de codes en instructions et en opérations. On peut ainsi recenser séparément les opérateurs et les instructions pour une plus grande souplesse d'analyse. L'estimation de l'activité d'un circuit est réalisée lors de la simulation du réseau de pétri. Celui-ci génère dans un premier temps un fichier qui recense toutes les instructions exécutées lors de la simulation du réseau de pétri. Ce fichier se présente sous un format particulier qu'on explicite très en détail dans la partie 3.2.1, un exemple complet est ensuite présenté (partie 3.2.2) et enfin ce fichier est analysé et des estimations (ainsi que des probabilités) en sont déduites (partie 3.2.3) .

3.2.1 Format du fichier Trace

La simulation d'un circuit se traduit par l'évolution du marquage du réseau de pétri associé au circuit. A chaque place correspond une instruction, cette instruction est imprimée dans un fichier qu'on nomme fichier Trace. Ainsi, toutes instructions de toutes les places parcourues sont recensées dans ce fichier Trace.

Le fichier Trace est de la forme :

NOM	: top_component.trf
TYPE	: texte
CONTENT	: blocs

Le fichier généré porte le nom du composant top auquel est ajouté l'extension ".trf". Il contient une succession de blocs au format texte qui recensent les opérations exécutées. Chaque bloc se présente sous la forme suivante :

```
BLOCK: Time Path Instruction|Operateur
          {instr_param}
          ...
          {expr : type=valeur}
```

Le bloc est composé d'une ligne d'entête qui identifie précisément l'instruction ou l'opérateur recensé (la barre verticale signifie qu'il s'agit soit d'une instruction ou soit d'un opérateur). A cette ligne d'entête s'ajoute des sous-blocs identifiables aux accolades. Ces sous-blocs sont présents pour certaines instructions seulement.

Nous explicitons maintenant les différents champs de la ligne d'entête :

- Time

Le premier terme correspond à "Time" qui est l'instant auquel est exécutée l'opération considérée. Il s'agit d'un temps relatif à chaque processus. C'est un paramètre intéressant qui sert à observer la concurrence ou la séquentialité de plusieurs instructions ou opérations.

- Path

Le second champs est le "path": chemin qui permet d'identifier précisément l'instruction ou l'opération. Ce chemin est unique pour chaque instruction ou opération, car il inclut toute la hiérarchie de composants et processus identifiant l'instruction ou l'opération exécutée.

La hiérarchie se présente comme suit :

```
path   : Component/[Component_instance]*/Process.[Label.gardes]*
```

Component est le nom du composant top.

Component_instance est le nom de l'instance du composant qui contient l'opération (identification hiérarchique). « []* » signifie qu'il peut s'agir d'un composant récursif.

Process est le nom du process qui contient l'opération.

Label est le nom de l'identificateur de l'instruction. Ce label est précisé dans le code CHP. Un label ne peut être associé aux opérateurs de concurrence « , » et de séquentialité « ; ». Un label peut être associé aux instructions d'affectation, de sélection, de répétition et de communications.

Ces labels sont ajoutés dans le code CHP de la façon suivante :

- . Pour une instruction d'affectation :

```
label_name : S := a + b
```

. Pour une instruction de sélection :

```
label_name : @[ G0 => ... break| loop
                G1 => ... break| loop
                ...
                Gn => ... break| loop
                ]
```

. Pour une instruction d'écriture sur un canal :

```
label_name : A ! A_var
```

. Pour une instruction de lecture sur un canal :

```
label_name : A ? A_var
```

Enfin le terme *gardes* correspond aux gardes des structures de choix. Les gardes sont par défaut labellisées par un numéro qui caractérise l'ordre de description des gardes dans le code CHP (G_1, \dots, G_n). Le label des gardes est généré automatiquement par l'outil de génération de traces. « $[]^*$ » signifie qu'il peut y avoir des combinaisons de labels et de gardes (choix imbriqués).

- Instructions : $[]$ $@[]$ $@@[]$ $?$ $!$ $:=$ skip

Le champ *Instruction* correspond aux instructions du langage CHP:

" $[]$ " est une structure d'une seule garde qui est toujours vraie. Cette instruction délimite traditionnellement un processus.

" $@[]$ " est une structure de choix déterministe. Cette structure doit présenter plus d'une garde, et seulement une garde peut être valide (propriété d'exclusion mutuelle).

" $@@[]$ " est un choix non déterministe. C'est une structure qui inclut un nombre de commandes gardées égal ou supérieur à 2, ici plusieurs gardes peuvent être vraies. La structure doit prendre une décision pour choisir une garde parmi les gardes vraies. Il n'y a pas de propriété d'exclusion mutuelle à respecter pour cette instruction.

" $!$ " est l'instruction d'émission. Elle permet d'émettre une donnée sur un canal.

" $?$ " est l'instruction duale de la précédente, elle représente la réception d'une donnée d'un canal.

" $:=$ " est l'instruction d'affectation d'une variable à l'intérieur d'un processus.

Enfin "skip" est l'instruction qui ne fait rien, elle passe seulement.

- Opérateur : « ; » «,div » «,conv »

Le dernier champ qui apparaît dans la ligne d'entête est l'*Opérateur*. Il s'agit des opérateurs de séquentialité et de concurrence. On note que l'opérateur de concurrence « , » se décline en 2 traces : « ,div » et « ,conv » qui correspondent à la divergence et à la convergence du réseau de pétri pour la concurrence.

Dans le cas où le code CHP recensé est une instruction, alors des sous-blocs sont ajoutés pour apporter des informations supplémentaires et nécessaires à la compréhension de l'instruction. Ces sous-blocs sont de deux types :

- {instr_param}

Ce sous-bloc est utilisé seulement dans le cas des sélections et des répétitions. Ce sous-bloc apporte des précisions sur le nombre de gardes dans la sélection, le nombre de gardes vraies et la garde sélectionnée.

{nombre de gardes *n* nombre de gardes vraies *Gv* garde sélectionnée *i*}

- {expr : type=valeur} :

Ce second type de sous-bloc est présent dans deux cas de figure, le premier cas concerne les instructions d'affectation, de skip ou de communication (émission ou réception d'une donnée). Le champ *expr* contient l'expression intégrale de l'instruction. Le champs *type* énumère le type des données manipulées : le nom de la donnée est précisé suivi de son type dont le format est de la forme [Base][Longueur][Dimension], suivi du signe de la donnée ([U] pour non signée et [S] pour signée). La valeur de la variable figure juste après son type. Lorsqu'une constante apparaît dans l'expression, elle est explicitée dans la trace. L'exemple suivant illustre la génération du fichier Trace à partir du CHP (la flèche → montre comment se présente le fichier Trace à partir du code CHP) :

Ex :

```
label_name : S := a + b - 10    -- ( S, a et b sont des vecteurs de 8 bits)
                                  -- a =5 et b=6

→ time component/component_instance/process_name.label_name :=
{S := a + b - 10 : S[2][8][1][S]="0.0.0.0.0.0.1" a[2][8][1][S]="0.0.0.0.1.0.1"
                  b[2][8][1][S]="0.0.0.0.1.1.0" "0.0.0.1.0.1.0"[2][8][1][S] }
```

NB :

L'exemple ci-dessous illustre un cas où une variable apparaît de part et d'autre de l'instruction d'affectation :

Ex :

```

label_name : a := a + 2      -- ( a est un vecteur de 8 bits)
                        -- a =1
→ time component/component_instance/process_name.label_name :=
{a := a + 2 : a[2][8][1][S]="0.0.0.0.0.1.1" a[2][8][1][S]="0.0.0.0.0.0.1"
  "0.0.0.0.0.1.0"[2][8][1][S] }

```

On note les valeurs de "a" avant et après l'affectation.

Ce second sous-bloc est aussi présent dans le cas de sélections ou de répétitions. Le champ *expr* représente la condition de la garde, *type* précise le type de la variable de la garde et *valeur* apporte des informations sur la donnée (variable ou constante) présente dans la condition.

L'exemple ci-dessous illustre un choix de 3 gardes, une seule garde est vraie et la deuxième garde est sélectionnée.

```

label_name : @[ a = "0001"[4] => I1 ; break
                a = "0010"[4] => I2 ; break
                a = "0100"[4] or a = "1000"[4] => I3 ; break
                ]
→ time component/component_instance/process_name.label_name @[
{3 1 2}
{a="0001" : a[4][1][1][U] ="0001"[4][1][1][U] }
{a="0010" : a[4][1][1][U] ="0010"[4][1][1][U] }
{a="0100" or a="1000" : a[4][1][1][U]="0100"[4][1][1][U]
a[4][1][1][U]="1000"[4][1][1][U] }

```

On note qu'une garde composée « *a="0100" or a="1000"* » apparaît sous sa forme complète dans le sous-bloc.

Le fichier généré par l'outil Trace est donc une source d'information complète caractéristique de l'exécution d'un programme CHP. Pour comprendre les résultats obtenus, un exemple complet est présenté dans le prochain paragraphe.

3.2.2 Exemple

Nous présentons un fichier Trace généré à partir d'une simulation d'un circuit décrit en CHP. On considère un composant, qu'on nomme "exemple", spécifié en CHP sous la forme suivante :

```

Component exemple
  Port ( cmd   : in DI MR[4][1][1] ;
        A     : in DI MR[2][8][1] ;
        B     : in DI MR[2][8][1] ;
        C     : out DI MR[2][8][1]
        );

Begin

Process P1
  Port ( cmd   : in DI MR[4][1][1] ;
        A     : in DI MR[2][8][1] ;
        B     : in DI MR[2][8][1] ;
        C     : out DI MR[2][8][1]
        );

Begin

Variable cmd_var : MR[4][1][1] ;
Variable A_var   : MR[2][8][1] ;
Variable B_var   : MR[2][8][1] ;
Variable C_var   : MR[2][8][1] ;

boucle_1 : [ cmd ? cmd_var ;
            @[ cmd_var = "0001"[4] => A ? A_var , B ? B_var ;
              expr_1 : c_var := (A_var + B_var + 2) ;
              C ! C_var ; Break
            cmd_var = "0010"[4] => A ? A_var ;
              C ! A_var ; break
            cmd_var = "0100"[4] or cmd_var = "1000"[4] =>
              B ? B_var ;
              C ! B_var ; break
            ] ;
          loop
        ]
End; -- process
End; -- component

```

Le fichier Trace généré lors de la simulation du réseau de pétri de ce circuit se nomme "exemple.trf" et se présente sous la forme suivante :

```

time exemple/P1.boucle_1 []
  {1 1 1}
  {TRUE : }
time exemple/P1.boucle_1.G1 ?
  { cmd ? cmd_var : cmd[4][1][1][U]="0001" cmd_var[4][1][1][U]="0001" }
time exemple/P1.boucle_1.G1 ;
time exemple/P1.boucle_1.G1 @[]
  {3 1 1}
  { cmd_var = "0001"[4] : cmd_var[4][1][1][U]="0001"[4][1][1][U]}
  { cmd_var = "0010"[4] : cmd_var[4][1][1][U]="0010"[4][1][1][U]}
  { cmd_var = "0100"[4] or cmd_var = "1000"[4] :
    cmd_var[4][1][1][U]="0100"[4][1][1][U]
    cmd_var[4][1][1][U]="1000"[4][1][1][U]}
time exemple/P1.boucle_1.G1.G1 ,div
time exemple/P1.boucle_1.G1.G1.no_label ?
  { A ? A_var : A [2][8][1][U]= "0.0.0.0.0.0.1"
    A_var[2][8][1][U] = "0.0.0.0.0.0.1" }
time exemple/P1.boucle_1.G1.G1.no_label ?
  { B ? B_var : B[2][8][1][U]= "0.0.0.0.0.1.0"
    B_var[2][8][1][U]= "0.0.0.0.0.1.0" }
time exemple/P1.boucle_1.G1.G1 ,conv
time exemple/P1.boucle_1.G1.G1 ;
time exemple/P1.boucle_1.G1.G1.expr_1 :=
  {c_var :=(A_var+B_var+2) : c_var[2][8][1][U]="0.0.0.0.1.0.1"
    A_var[2][8][1][U]= "0.0.0.0.0.0.1"
    B_var[2][8][1][U]="0.0.0.0.0.1.0"
    "0.0.0.0.0.1.0"[2][8][1][U]}
time exemple/P1.boucle_1.G1.G1 ;
time exemple/P1.boucle_1.G1.G1.no_label !
  { C ! C_var : C[2][8][1][U]="0.0.0.0.1.0.1"
    C_var[2][8][1][U]= "0.0.0.0.1.0.1"}
time exemple/P1.boucle_1 []
  {1 1 1}
  {TRUE : }

```

On considère dans cet exemple que le composant est simulé une seule fois et que la première garde est sélectionnée (cmd_var = "0001"[4]). On observe qu'un grand nombre d'information est offert au concepteur sur l'exécution de ce composant. Il peut de ce fait avoir un regard très avisé sur le comportement du circuit lors de la simulation. Cependant, pour de longues simulations, la quantité d'information recueillie peut être très fastidieuse et pas forcément d'une grande utilité. Plus grave encore, l'information utile peut se noyer dans un amas d'informations. Par conséquent, il est proposé au concepteur des options permettant de ne collecter que les informations dont il a réellement besoin. De surcroît, une représentation graphique permet d'illustrer les résultats de l'exécution et d'estimer l'activité du circuit. Cela est présenté dans le prochain paragraphe.

3.2.3 Caractéristiques de l'estimateur d'activité

Au regard de l'exemple et des remarques faites précédemment, cette partie est consacrée aux possibilités variées proposées par l'outil d'estimation de l'activité. Il est possible de réduire la quantité d'information que le concepteur désire recueillir pour cibler des points très précis de son analyse. Ceci est possible grâce à la présence de commandes et d'options que l'outil d'estimation d'activité offre au concepteur. Nous présentons dans ce paragraphe ces possibilités.

3.2.3.1 Couverture de code

L'outil d'estimation de l'activité propose au concepteur de réaliser l'analyse de la couverture de code d'une spécification CHP. Le but de cette couverture de code est d'identifier les parties d'une description qui ne sont jamais utilisées lors d'une simulation donnée. Les informations concernant le code non exécuté sont imprimées dans un fichier ("code_coverage.log") qui est directement visible par le concepteur. Une ligne de code jamais exécutée produit un fichier au format suivant:

```
path.process_name instr ref
```

Où path.process_name correspond au chemin qui identifie précisément un processus dans lequel l'instruction n'est jamais exécutée. "Instr" est l'instruction jamais exécutée, et "ref" fournit les informations sur la colonne et la ligne de l'instruction jamais exécutée.

Dans la simulation du composant "exemple" présenté dans la section précédente, les commandes gardées G2 et G3 de la structure de choix ne sont jamais exécutées, le fichier généré par la commande de couverture de code est le suivant:

```
exemple.p1.boucle_1.G1.G2 a ? a_var[7..0][0] (L=47, C=11)
exemple.p1.boucle_1.G1.G2 c ! a_var[7..0][0] (L=48, C=11)

exemple.p1.boucle_1.G1.G3 b ? b_var[7..0][0] (L=50, C=11)
exemple.p1.boucle_1.G1.G3 c ! b_var[7..0][0] (L=51, C=11)
```

Comme il a été dit plus tôt dans le chapitre, les gardes dans une structure de choix sont par défaut labellisées. Ce label est le numéro de la garde; la première garde est labellisé G1, la seconde G2 etc. Si l'instruction jamais exécutée se trouve dans une garde de la structure de choix, l'identification de l'instruction inclut le label de la garde pour identifier précisément l'instruction. Dans le composant "exemple", les gardes G2 et G3 de la structure de choix ne sont jamais exécutées, les instructions contenues dans ces deux commandes gardées sont écrites dans le fichier de couverture de code.

3.2.3.2 Trace

La commande "Trace" proposée par l'outil d'estimation d'activité permet d'observer le comportement d'un ou plusieurs canaux et/ou variables d'une spécification CHP.

- Canaux

La première option de cette commande "Trace" permet au concepteur de visualiser précisément le comportement d'un ou plusieurs canaux d'un circuit. Il peut ainsi observer le nombre de fois qu'un canal est activé et les valeurs qu'il reçoit ou qu'il émet. Toutes les informations sont imprimées dans le fichier Trace.

La commande "Trace" va plus loin en proposant au concepteur d'observer les statistiques sur le comportement des digits dans le canal. Les résultats des statistiques sont présentés à la Figure 3.8 pour le canal "cmd" qui est la commande de la structure de choix du composant exemple. Il s'agit d'un multirail MR[4], c'est un dire d'un seul digit codé sur 4 rails. Les statistiques obtenues par l'outil montre que le fil 0 de la commande est le seul utilisé (il présente une activité de 100%); les 3 autres rails ne sont jamais sollicités.

C'est une information importante pour de futures optimisations (cf. Chapitre 4).

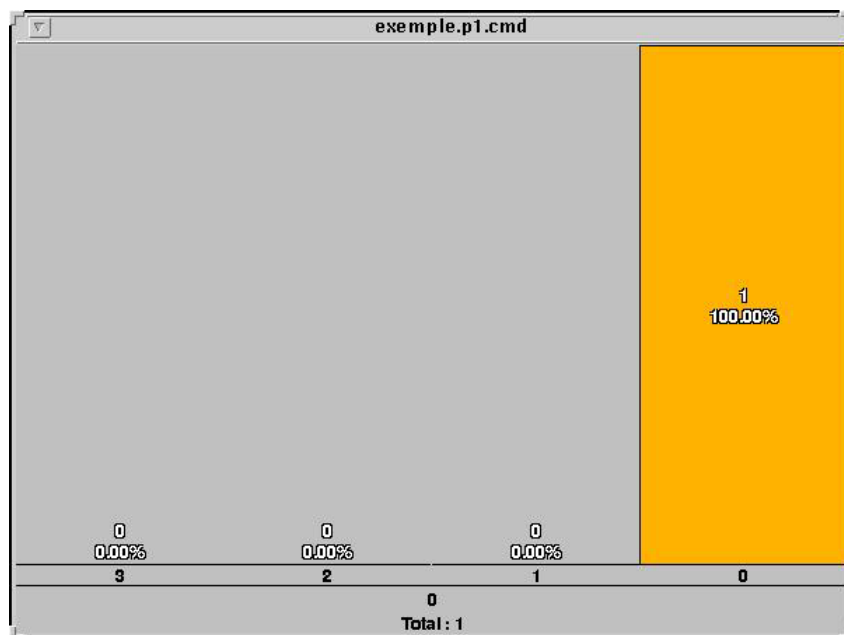


Figure 3.8 Trace des canaux ("Trace -channels")

- Variables

L'option variable de la commande Trace suit le même raisonnement que ce que nous avons présenté dans le cas des canaux. Ici, le traitement est fait sur les variables au lieu des canaux, mais les résultats obtenus par l'outil sont de même nature. A savoir, l'outil génère un fichier Trace qui détaille le comportement d'une ou plusieurs variables lors d'une simulation. Ce fichier s'accompagne de statistiques de l'activité des digits dans une variable comme cela est montré à la Figure 3.9. La figure montre le comportement de la variable a_var qui est une variable de 8 digits. Cette variable reçoit la valeur 1 (seul le digit 0 est mis à 1).

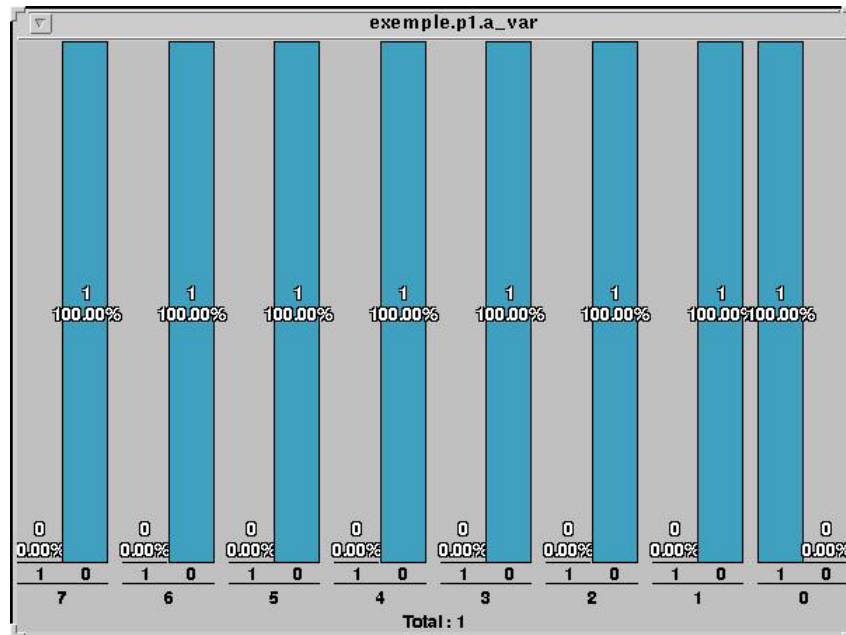


Figure 3.9 Trace variables

3.2.3.3 Statistiques

L'estimation de l'activité d'un circuit est une information pertinente pour observer la répartition de l'activité d'un circuit. Le but de ces informations est d'analyser comment l'activité est répartie lors d'une simulation donnée (représentant l'application du circuit) et d'identifier les parties du circuit qui sont le plus fréquemment utilisées. Les résultats sont donnés sous forme de statistiques. Pour faire des comparaisons pertinentes, l'outil propose de calculer des statistiques de différentes natures que nous expliquons dans ce paragraphe.

- Instructions

Le premier type de statistiques que le concepteur peut obtenir concerne les statistiques sur les instructions. Dans un processus, des statistiques sont établies pour analyser quelles instructions sont exécutées. Les résultats sont présentés par un histogramme qui établit le nombre de fois que les instructions sont exécutées et donne les statistiques sur l'exécution de ces instructions parmi toutes les instructions existantes. La Figure 3.10 montre les résultats des statistiques données par la simulation du composant exemple.

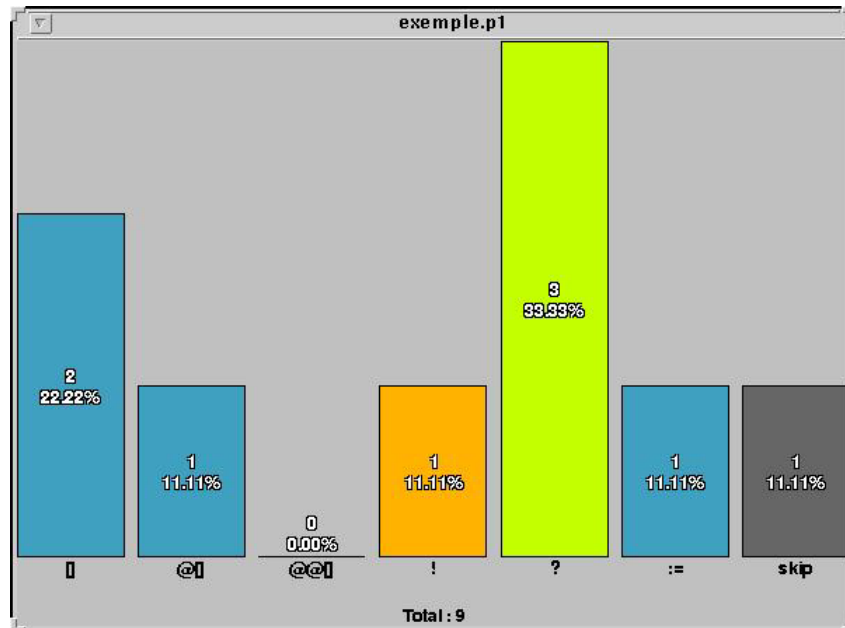


Figure 3.10 Statistiques sur les instructions

On voit que le composant ne contient aucun choix non déterministe "@@[]". De plus, l'instruction de réception "?" est l'instruction la plus exécutée.

- Structures de choix

Des statistiques sur les structures de choix peuvent être établies par l'estimateur d'activité. Dans une structure de choix incluant plusieurs commandes gardées, il est intéressant de connaître les statistiques sur l'exécution de chaque garde. Le composant exemple présente une structure de trois choix (G1, G2 et G3). Les statistiques obtenues par l'outil Trace sur l'exécution des gardes sont présentées à la Figure 3.11.

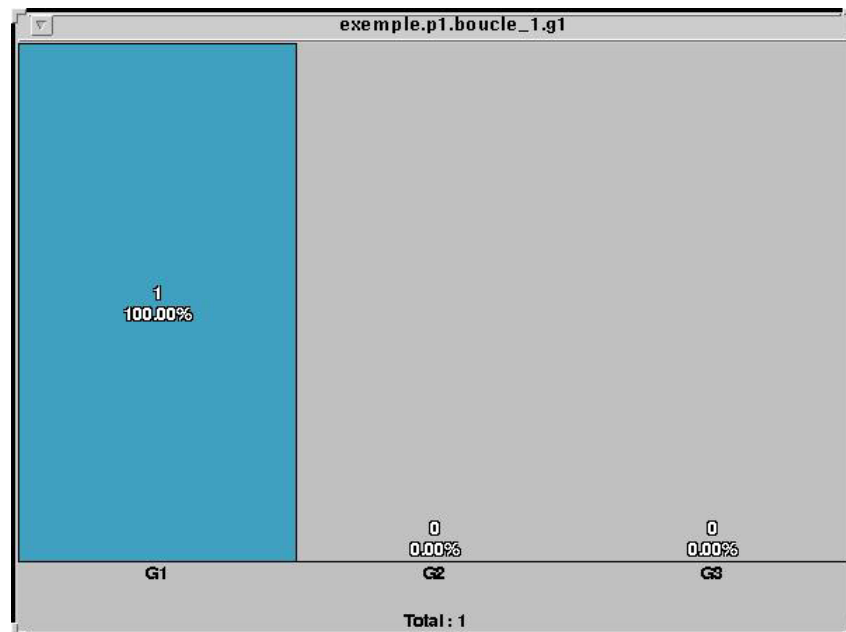


Figure 3.11 Statistiques sur les structures de choix

L'histogramme fournit des informations précieuses sur l'exécution des commandes gardées. On peut voir que seule la garde G1 est exécutée (100%). Les deux autres ayant une activité nulle.

On verra par la suite, notamment au Chapitre 4, comment ces probabilités peuvent être utilisées pour déséquilibrer les structures de choix afin d'optimiser la consommation d'énergie. Il est possible d'obtenir des résultats plus élaborés que ceux présentés dans ce paragraphe. Le CHP développé à TIMA autorise le concepteur à ajouter des labels dans un programme CHP, il peut ainsi analyser des opérations et/ou instructions bien définies dans sa spécification CHP.

L'estimation d'activité d'un circuit est une information intéressante pour concevoir des circuits à faible consommation d'énergie, cependant ces informations sont dans certains cas incomplètes. Pour rendre l'information pertinente, il est intéressant de coupler l'activité du circuit avec l'estimation de la consommation d'énergie (en terme de nombre de transitions) pour avoir un regard précis sur les parties qui sont le plus sollicitées et celles qui consomment le plus. L'estimateur de la consommation d'énergie fait l'objet du prochain paragraphe.

3.3 Estimateur de la consommation d'énergie

Une spécification de l'estimateur de la consommation d'énergie a été réalisée dans le cadre de la thèse. Celui-ci n'est pas encore intégré dans le flot TAST, mais son intégration représente la perspective principale des travaux de cette thèse. L'estimation de la consommation d'énergie que nous présentons dans ce paragraphe est une information complémentaire, qui n'est pas forcément redondante de l'estimation d'activité. En effet, une sous partie d'un circuit qui présente une forte activité n'est pas forcément la partie qui consomme le plus, car la consommation d'énergie dépend de l'instruction qui est exécutée. Pour l'activité, il s'agit d'un compteur qui est incrémenté à chaque exécution d'un composant, d'un processus ou d'une instruction, tandis que pour l'estimation de la consommation d'énergie il s'agit d'un calcul qui tient compte de la complexité de l'instruction exécutée. La complémentarité de ces deux estimateurs apporte aux concepteurs des informations très intéressantes pour la réalisation de circuits à faible consommation d'énergie.

3.3.1 Principe de l'estimateur de la consommation d'énergie

Des outils commerciaux existent et permettent de faire de l'estimation de la consommation d'énergie, cependant cette estimation est faite soit au niveau porte logique [POWE] soit au niveau électrique [SPEC]. Ces évaluations sont plus précises que celle que nous proposons, mais elles présentent l'inconvénient d'être longues (particulièrement la simulation électrique). De plus, ces estimations surviennent assez tardivement dans le flot de conception pénalisant ainsi le temps de conception. L'estimation que nous proposons survient au niveau de la description CHP fonctionnelle. Contrairement à l'estimateur d'activité, l'estimateur de la consommation d'énergie des circuits ne se base pas sur le fichier de Trace. L'estimation est faite directement lors de la simulation du réseau de Pétri ; elle est basée sur la méthodologie d'estimation de la consommation d'énergie établie au chapitre précédent. Nous avons vu que l'estimation de la consommation d'énergie est réalisée en deux étapes. La première étape consiste à établir tous les coûts structurels de tous les processus du circuit à concevoir. Cela permet d'établir une équation booléenne par processus du circuit. Puis une seconde étape du calcul de la consommation d'énergie consiste à simuler le circuit et à calculer le coût dynamique de la structure lors de cette exécution. Le calcul de ce coût dynamique correspond à une somme de coûts structurels lorsque la place du réseau de pétri contenant une équation de coût structurel est atteinte. L'estimateur de la consommation d'énergie établit les coûts dynamiques des parties du circuit simulées.

3.3.2 Caractéristiques de l'estimateur de la consommation d'énergie

On rappelle que l'outil d'estimation de la consommation d'énergie n'est pas encore intégré dans le flot de conception asynchrone TAST. La méthodologie d'estimation de la consommation d'énergie a été entièrement spécifiée, celle-ci a été longuement présentée dans le chapitre précédent. Nous présentons dans ce paragraphe les caractéristiques de l'estimateur de la consommation d'énergie tel qu'il sera intégré dans TAST. L'estimation est faite lors de la simulation du réseau de pétri annoté des équations de coûts structurels. L'estimation de la consommation d'énergie est réalisée au même niveau que l'estimation de l'activité du circuit présentée dans les paragraphes précédents.

Nous présentons les différentes informations sur la consommation que le concepteur peut recueillir en fonction de l'option qu'il choisit. Pour ce faire, on considère le pseudo code CHP suivant :

```

Component Composant_1
  process Process_1
    @ [ G0 => I0
      G1 => I1
    ]
  process Process_2
    Label_2 : instruction
Component Composant_2
  Process Process_3
    instruction
  
```

- Consommation des commandes gardées labellisées

La première option possible de l'estimateur de la consommation d'énergie est de recueillir des informations sur la consommation d'énergie des instructions labellisées. Cette option représente la plus fine granularité de l'estimation de la consommation d'énergie. Il existe une différence sur l'appellation de "label" dans le cadre du calcul de la consommation d'énergie par rapport à l'estimation de l'activité du circuit. Le terme label que nous utilisons pour le calcul de la consommation d'énergie est plus restreint que le label introduit dans le cadre de l'estimation de l'activité. Le label dans le cas de la consommation d'énergie ne concerne que les commandes gardées des structures de choix. On a vu précédemment que les gardes ont par défaut un label correspondant au numéro de la garde. On estime la consommation d'énergie des commandes gardées seulement et non de chaque instruction. Les autres labels du programme CHP sont tout simplement ignorés lors du calcul de l'estimation de la consommation d'énergie. Le processus Process_1 du composant Component_1 présente une structure à deux choix. L'estimateur permet dans ce cas d'établir la consommation d'énergie des deux gardes (Figure 3.12).

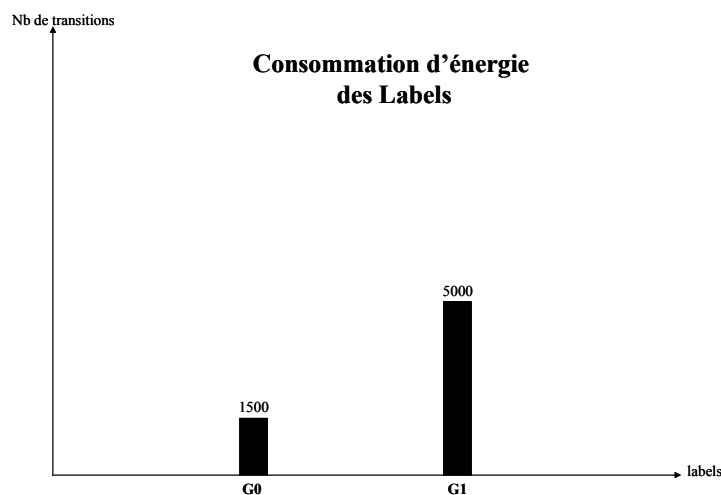


Figure 3.12 Consommation d'énergie des labels du process_1

L'histogramme signifie que la commande gardée G0 présente une consommation d'énergie de 1500 transitions lors d'une simulation donnée alors que la commande gardée G1 a une consommation de 5000 transitions.

- Consommation des processus

La seconde option est l'estimation de la consommation d'énergie des processus dans un composant. L'estimateur de la consommation d'énergie permet de connaître le coût de chaque processus d'un composant (Figure 3.13).

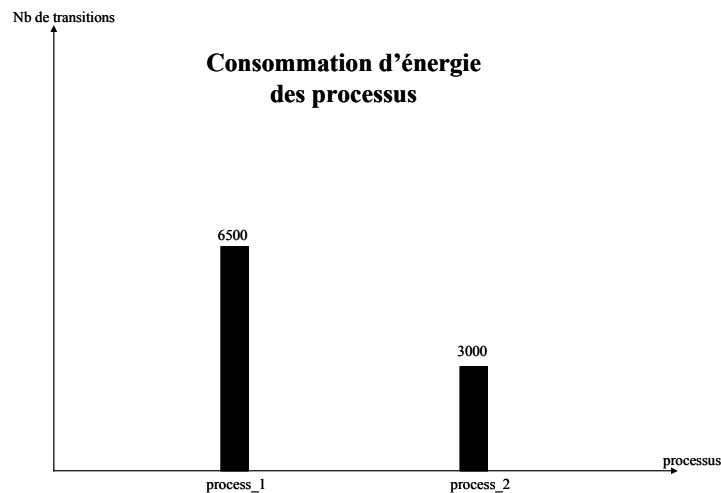


Figure 3.13 Consommation d'énergie des processus du composant_1

Les processus process_1 et process_2 du composant composant_1 présentent les coûts respectifs de 6500 et de 3000 transitions lors de la simulation. On note que la valeur 6500 correspond à la somme des consommations d'énergie des commandes gardées G0 et G1 exécutées.

- Consommation des composants

Enfin, le plus haut niveau de la hiérarchie représente le composant. Il est possible d'estimer la consommation d'énergie globale de chaque composant d'un circuit (Figure 3.14).

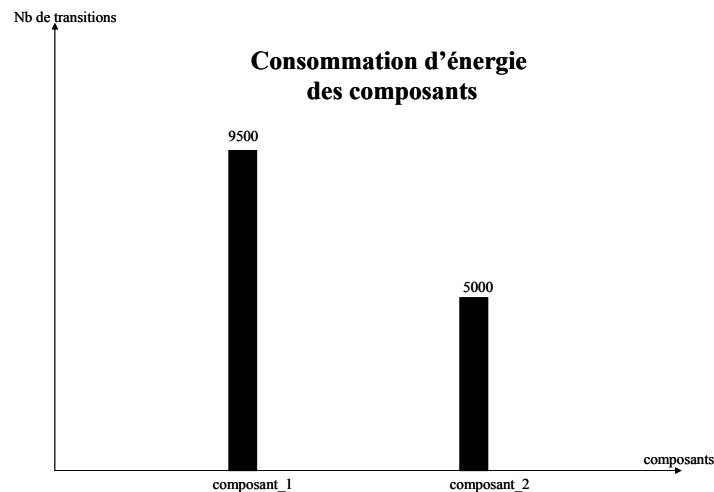


Figure 3.14 Consommation des composants

Le composant composant_1 a une consommation de 9500 transitions (qui est la somme des consommations des 2 processus qu'il contient) et le composant composant_2 à une consommation de 5000 transitions.

3.4 Conclusion

Ce chapitre a permis de détailler les fonctionnalités diverses et variées de l'estimateur d'activité telles qu'elles ont été intégrées dans le flot de conception TAST. Le mode de

fonctionnement de cet estimateur a été exposé; cet outil est aujourd'hui opérationnel. Par ailleurs, on a spécifié un second outil qui est un estimateur de la consommation d'énergie tel qu'on souhaite qu'il soit intégré à terme dans TAST. Cet estimateur de la consommation d'énergie utilise la méthodologie d'estimation de la consommation d'énergie que nous avons établie au chapitre précédent.

Ce chapitre a très clairement montré la capacité de ces outils à fournir des informations importantes pour la réalisation de circuits asynchrones à faible consommation d'énergie. Le chapitre suivant montre comment utiliser ces outils pour concevoir des circuits asynchrones à faible consommation d'énergie.

Chapitre 4

Exploration Architecturale et Optimisation de la Consommation d'Énergie

La volonté de concevoir des circuits à très faible consommation d'énergie s'est en tout premier lieu manifestée par l'utilisation de la logique asynchrone. Dans un second temps, un estimateur d'activité et une méthodologie d'estimation de la consommation d'énergie à un niveau de conception très élevé ont été développés afin d'observer la répartition de la consommation d'énergie et d'identifier les parties du circuit qui consomment le plus. Enfin nous proposons dans ce chapitre les techniques d'optimisation de la consommation d'énergie. Ce chapitre se veut une exploration architecturale assez large qui s'attache à identifier les optimisations judicieuses pouvant être apportées à un circuit afin de réduire à un minimum la consommation d'énergie. Il s'articule autour de deux parties principales. La section 4.1 permet l'étude et la comparaison de certaines implémentations architecturales afin d'identifier celles qui consomment le moins. Ces informations seront à terme utilisées dans l'outil de synthèse pour permettre la synthèse automatique de circuits à faible consommation d'énergie. La section 4.2 présente au concepteur certaines techniques de spécifications CHP pour obtenir des circuits à faible consommation d'énergie.

4.1 Synthèse orientée faible consommation d'énergie

L'équation du coût structurel de la consommation d'énergie établie au Chapitre 2 constitue un modèle qui permet d'étudier la consommation d'énergie en fonction des différents paramètres tels que le nombre d'entrées, le nombre de sorties, la base des données, la longueur des données et le nombre de branches d'une structure de choix. En analysant le comportement de la consommation d'énergie en fonction de ces différents paramètres, il est possible de comparer différentes implémentations. Ces informations pourront être à terme utilisées pour la synthèse automatique à faible consommation d'énergie.

4.1.1 Protocole de communication

Le premier choix important, que tout concepteur est amené à faire, concerne la bufferisation qu'il doit utiliser dans le circuit qu'il conçoit. Nous avons présenté dans l'état de l'art les protocoles de communication existants (WCHB, PCHB, PCFB). Nous avons par la suite, notamment au paragraphe 2.2.1.3, établi le coût de ces trois types de bufferisation. Nous allons à présent analyser la bufferisation la mieux adaptée à la conception de circuit à faible consommation d'énergie.

Nous considérons le cas le plus simple d'un latch qui est un composant permettant d'envoyer une entrée sur une sortie en retournant un acquittement au bloc qui le précède. Dans le cas d'un latch, nous avons une entrée d'un certain type qui est envoyée sur une sortie du même

type que l'entrée, par conséquent la bufferisation est identique qu'on choisisse de bufferiser l'entrée où la sortie, et ce pour les trois types de protocole. Les équations de coût du latch pour les 3 protocoles sont présentées.

Dans le cas d'une bufferisation WCHB, le coût structurel est le suivant :

$$\text{Coût}_{\text{WCHB_entree}} \{\text{latch}\} = \text{Coût}_{\text{WCHB_sortie}} \{\text{latch}\} = \text{Fourche}(\text{B}(\text{In}(0)) * \text{L}(\text{In}(0))) + \text{L}(\text{In}(0)) * \text{muller2R} + \text{L}(\text{In}(0)) * \text{NOR}(\text{B}(\text{In}(0))) + \text{muller}(\text{L}(\text{In}(0)))$$

Pour une bufferisation PCHB, il vaut :

$$\begin{aligned} \text{Coût}_{\text{PCHB_entree}} \{\text{latch}\} &= \text{Coût}_{\text{PCHB_sortie}} \{\text{latch}\} = \text{Fourche}(\text{B}(\text{In}(0)) * \text{L}(\text{In}(0))) \\ &+ \text{L}(\text{In}(0)) * (\text{OR}(\text{B}(\text{In}(0))) + \text{muller3DIP1NR} + \text{OR}(\text{B}(\text{In}(0))) + \text{muller2D1NB}) \\ &+ \text{muller}(\text{L}(\text{In}(0))) \end{aligned}$$

Enfin pour une bufferisation PCFB, le coût structurel est :

$$\begin{aligned} \text{Coût}_{\text{PCFB_entree}} \{\text{latch}\} &= \text{Coût}_{\text{PCFB_sortie}} \{\text{latch}\} = \text{Fourche}(\text{B}(\text{In}(0)) * \text{L}(\text{In}(0))) \\ &+ \text{L}(\text{In}(0)) * \left(\begin{array}{l} \text{OR}(\text{B}(\text{In}(0))) + \text{muller}(2) + \text{muller2DIP1R} + \text{OR}(\text{B}(\text{In}(0))) + \text{muller3DIP1NB} \\ + \text{INV} + \text{muller}(2) \end{array} \right) \\ &+ \text{muller}(\text{L}(\text{In}(0))) \end{aligned}$$

Observons le comportement de ces coûts en fonction du nombre de digits de la donnée manipulée (c'est-à-dire la longueur L), les résultats obtenus sont illustrés à la Figure 4.1 (on choisit une donnée de base B=2).

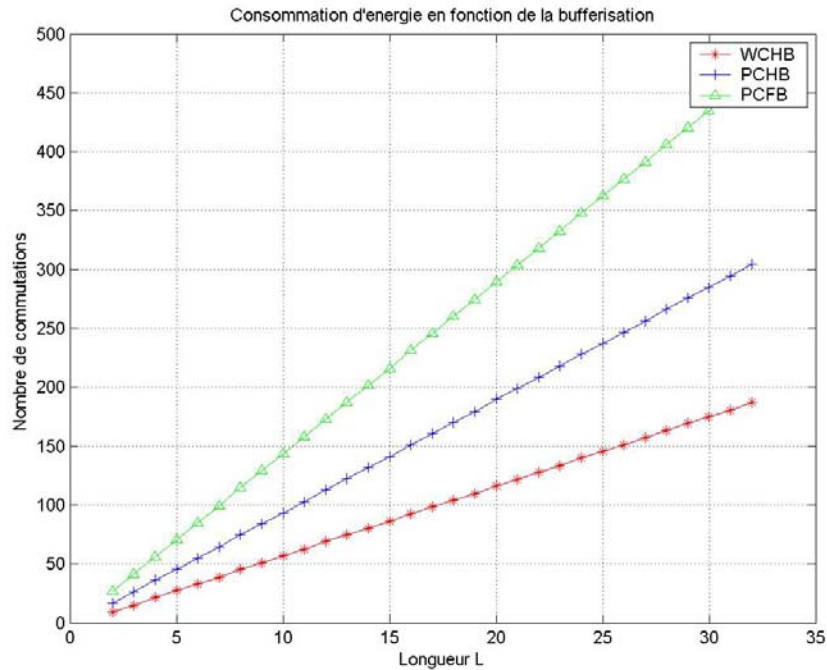


Figure 4.1 Comparaison de bufferisation WCHB, PCHB, PCFB

La Figure 4.1 montre très clairement que la bufferisation WCHB est celle qui consomme le moins pour une tâche donnée (c'est-à-dire pour la réception d'une donnée sur le canal d'entrée et l'émission de cette donnée sur le canal de sortie indépendamment de la vitesse d'exécution de cette tâche). C'est un résultat qui était assez prévisible pour les concepteurs asynchrones. En effet, il est connu que les protocoles de communication PCHB et PCFB sont plus adaptés à l'optimisation de la vitesse des circuits asynchrones, avec une remise à zéro des sorties qui survient plus tôt que pour le protocole WCHB. Par ailleurs, le protocole PCFB permet d'inclure un buffer complet dans le but de découpler deux étages successifs (utilisation d'une variable).

L'option "faible consommation d'énergie" de l'outil de synthèse générera par défaut une bufferisation WCHB.

4.1.2 Bufferisation entrée/sortie

Il est intéressant d'étudier la consommation d'énergie de certains composants, et d'essayer de trouver la meilleure configuration possible en vue de réduire la consommation d'énergie.

Certains composants présentent un nombre de données en entrée qui n'est pas forcément le même qu'en sortie. Il est souvent très difficile de savoir avec exactitude quelle est la bufferisation adéquate pour une consommation minimale de l'énergie. Dans cette partie nous nous attachons à étudier la bufferisation des composants de base et de dire avec précision quelle bufferisation entrée/sortie est la mieux adaptée.

- Le latch

Pour le cas d'un latch (Figure 2.28), étant donné qu'on a une seule donnée en entrée et une seule donnée en sortie, la question de la bufferisation n'a pas besoin d'être traitée. En effet, un latch présente une seule entrée qui est envoyée sur une seule sortie. De plus, l'entrée est du même type que la sortie, par conséquent la bufferisation est identique qu'elle soit réalisée en entrée ou en sortie.

- Le Fork

Le "fork" est un composant qui présente une donnée en entrée qui est propagée sur plusieurs sorties, les bufferisations en entrée (Figure 2.29) et en sortie (Figure 2.30) devraient a priori entraîner une consommation d'énergie différente.

Les équations de coût établies au paragraphe 2.2.4.2 pour une bufferisation en entrée et en sortie sont les suivantes.

Pour une bufferisation en entrée du "fork", la consommation est:

$$\text{Coût}_{\text{WCHB_entrée}}\{\text{Fork}\} = L(\text{In}(0)) * \text{Fourche}(J) + \text{muller}(J) + \text{Fourche}(B(\text{In}(0)) * L(\text{In}(0))) + L(\text{In}(0)) * \text{muller}2R + L(\text{In}(0)) * \text{NOR}(B(\text{In}(0))) + \text{muller}(L(\text{In}(0)))$$

Si on décide de bufferiser les sorties, alors la consommation est :

$$\text{Coût}_{\text{WCHB_sortie}}\{\text{Fork}\} = L(\text{In}(0)) * \text{Fourche}(J) + \text{muller}(J) + \text{Fourche}(J * B(\text{Out}(0)) * L(\text{Out}(0))) + J * L(\text{Out}(0)) * \text{muller}2R + J * L(\text{Out}(0)) * \text{NOR}(B(\text{Out}(0))) + J * \text{muller}(L(\text{Out}(0)))$$

Nous comparons les bufferisations en entrée et en sortie en fonction du nombre de sorties J. On fixe la base et la longueur de l'entrée In(0) et des sorties Out(j), elles valent B=2 et L=32.

On veut observer comment évolue la consommation d'énergie en fonction du nombre de sorties J (Figure 4.2).

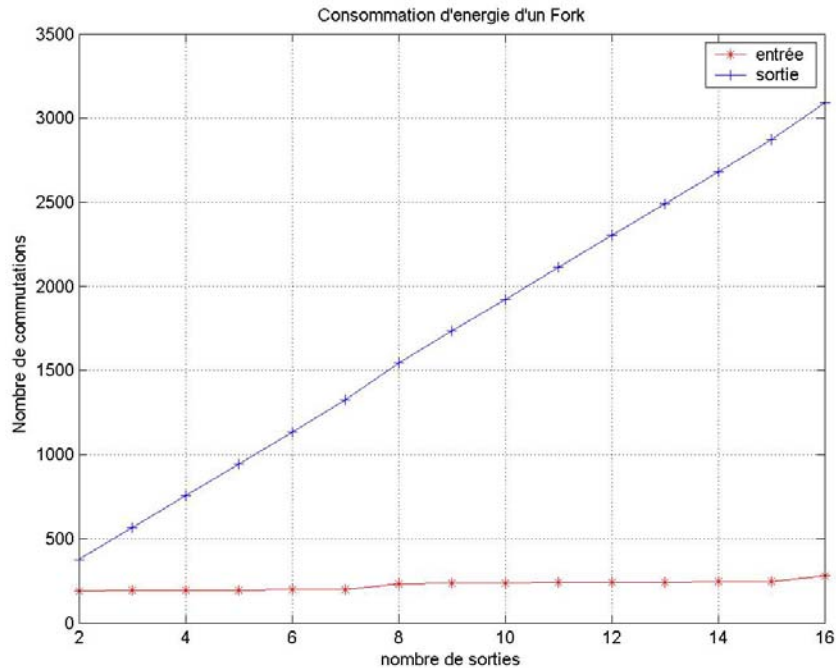


Figure 4.2 Consommation d'un Fork bufferisé en entrée et en sortie

La consommation d'énergie est nettement inférieure lorsqu'on bufferise l'entrée. Un résultat qui était à priori envisageable sachant qu'on a une seule donnée en entrée et J données en sortie. On constate aussi que la consommation est d'autant plus importante en sortie que le nombre de sorties augmente.

De plus, bufferiser l'entrée permet de retourner plus tôt et plus rapidement l'acquiescement d'entrée, améliorant ainsi la vitesse du circuit.

La synthèse automatique de ce composant se fera toujours en bufferisant l'entrée.

- Le Join

Le composant "join" correspond à la jonction de plusieurs données d'entrée, ces données sont manipulées, un résultat est généré puis envoyé sur un bus de sortie. Les bufferisations en entrée et en sortie de ce composant sont proposées respectivement à la Figure 2.31 et à la Figure 2.32.

La bufferisation réalisée en entrée présente le coût suivant :

$$\text{Coût}_{\text{WCHB_entrée}}\{\text{Join}\} = \text{Coût}\{F_S\} + \text{Fourche}(I * B(\text{In}(0)) * L(\text{In}(0))) \\ + I * L(\text{In}(0)) * \text{muller}2R + I * L(\text{In}(0)) * \text{NOR}(B(\text{In}(0))) + I * \text{muller}(L(\text{In}(0)))$$

Pour une bufferisation en sortie, le coût est différent et vaut :

$$\text{Coût}_{\text{WCHB_sortie}}\{\text{Join}\} = \text{Coût}\{F_S\} + \text{Fourche}(B(\text{Out}(0)) * L(\text{Out}(0))) \\ + L(\text{Out}(0)) * \text{muller}2R + L(\text{Out}(0)) * \text{NOR}(B(\text{Out}(0))) + \text{muller}(L(\text{Out}(0))) \\ + \text{Fourche}(I)$$

Comme pour le composant "fork", nous fixons la base et la longueur des données $In(i)$ et $Out(0)$ ($B=2$ et $L=32$). On regarde l'évolution de la consommation d'énergie de ces deux bufferisations en fonction du nombre d'entrées I (Figure 4.3). On choisit $Coût\{F_s\} = cste = 0$, le coût de la fonction est le même qu'on bufferise en entrée ou en sortie, on veut seulement visualiser l'effet de la bufferisation sur la consommation d'énergie.

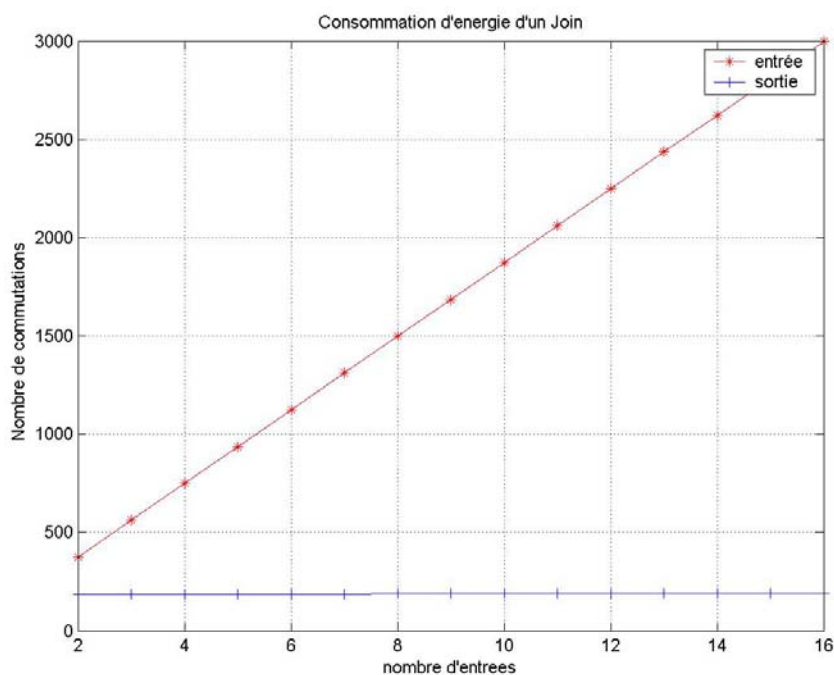


Figure 4.3 Consommation d'un Join bufferisé en entrée et en sortie

On constate sur la figure que la consommation d'énergie du composant "join" bufferisé en sortie est inférieure à la consommation de celui bufferisé en entrée. Par ailleurs, la bufferisation en sortie est indépendante du nombre de données entrantes. Inversement, la bufferisation en entrée est fortement tributaire du nombre d'entrées. Il est en somme plus judicieux de bufferiser la sortie d'un "join" sachant qu'on a une seule sortie à bufferiser plutôt que les entrées.

Néanmoins, le fait de bufferiser la sortie présente un impact sur la vitesse du circuit. La bufferisation en sortie implique un acquittement qui est produit plus tard (voir beaucoup plus tard si on est en présence d'un opérateur qui nécessite un temps relativement long pour produire un résultat), par conséquent les étages qui précèdent le "join" sont libérés plus tard. La vitesse du circuit se voit ainsi pénalisée.

- Le merge

Un "merge" est un composant qui prend une donnée en entrée parmi un certain nombre, et envoie cette donnée en sortie. C'est une structure qui n'est pas commandée, par conséquent le concepteur qui utilise un merge doit s'assurer de l'exclusion mutuelle des données en entrée. La Figure 4.4 présente une bufferisation en entrée et en sortie intermédiaire.

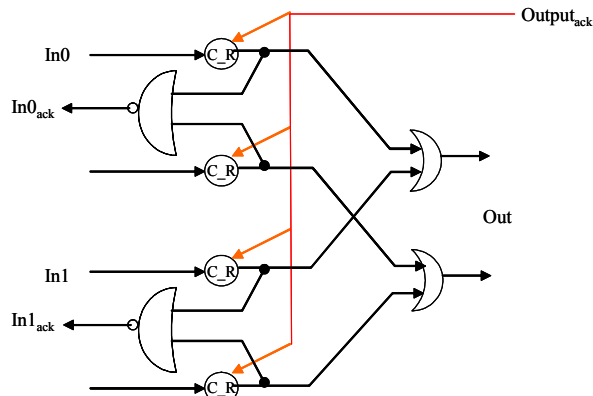


Figure 4.4 Merge bufferisé en entrée

Le coût est:

$$\text{Coût}_{\text{WCHB_entrée}} \{\text{Merge}\} = \text{Coût}_{\text{WCHB_sortie_inter}} \{\text{Merge}\} =$$

$$\text{Fourche}(Q * B(\text{In}(0)) * L(\text{In}(0))) + L(\text{In}(0)) * \text{muller}2R + L(\text{In}(0)) * \text{NOR}(B(\text{In}(0)))$$

$$+ \text{muller}(L(\text{In}(0))) + L(\text{Out}(0)) * \text{OR}(Q)$$

Si la bufferisation est faite en sortie finale (Figure 4.5).

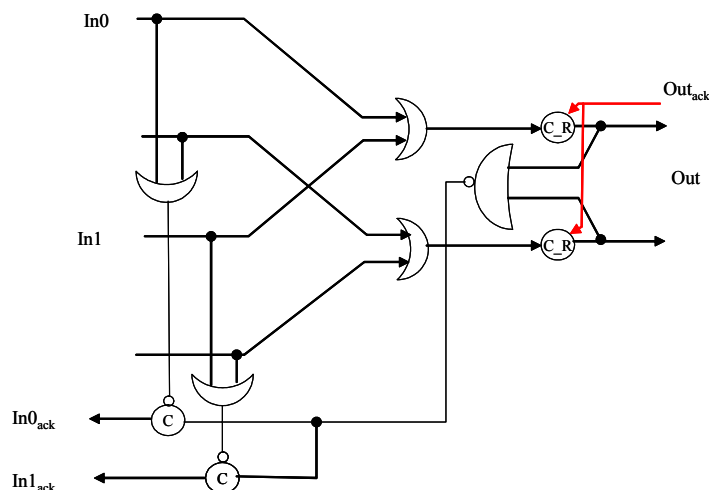


Figure 4.5 Merge bufferisé en sortie

Alors l'équation de la consommation est la suivante:

$$\text{Coût}_{\text{WCH_sortie_finale}} \{\text{Merge}\} =$$

$$\text{Fourche}(B(\text{Out}(0)) * L(\text{Out}(0))) + L(\text{Out}(0)) * \text{muller}2R + L(\text{Out}(0)) * \text{NOR}(B(\text{Out}(0)))$$

$$+ \text{muller}(L(\text{Out}(0))) + L(\text{Out}(0)) * \text{OR}(Q) + \text{INV} + \text{muller}(2) + \text{Fourche}(Q)$$

$$+ L(\text{In}(0)) * \text{OR}(B(\text{In}(0))) + \text{muller}(L(\text{Out}(0)))$$

Comparons les consommations d'énergie d'un "merge" bufferisé en entrée et d'un "merge" bufferisé en sortie en fonction du nombre de gardes Q présentes dans le "merge" (à noter que le nombre de garde est égal au nombre d'entrée I) (Figure 4.6). On fixe B=2 et L=32 des données utilisées dans ce composant.

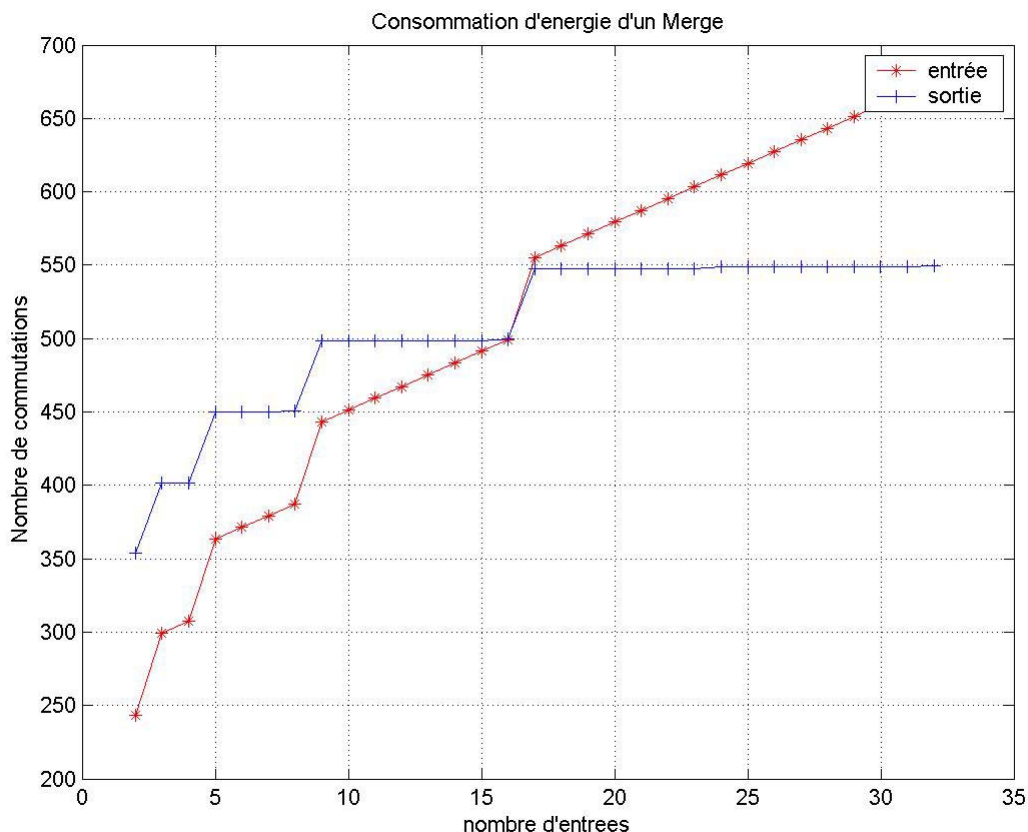


Figure 4.6 Consommation d'un Merge bufferisé en entrée et en sortie

Le graphique montre que pour un nombre d'entrée inférieur à 16, il est préférable de bufferiser les entrées plutôt que la sortie. Lorsque le "merge" est bufferisé en entrée, cela coûte la consommation d'un buffer sur le chemin de donnée de l'entrée qui est activée. Par contre, lorsqu'on bufferise la sortie, cela coûte la consommation d'un buffer sur la sortie, il faut en plus détecter quelle donnée a été activée pour n'acquitter que celle-ci. La logique ajoutée pour détecter la validité de la donnée explique le surcoût de la bufferisation en sortie. Au delà de 16 entrées, la bufferisation en sortie devient moins coûteuse que la bufferisation en entrée. Le coût de la fourche d'acquittement qui est envoyé à toutes les entrées lorsque le "merge" est bufferisé en entrée devient supérieur au coût de la logique de détection de la donnée lorsque le "merge" est bufferisé en sortie.

On remarque que les courbes sont en dent de scie. Pour la bufferisation en entrée, on voit que la courbe augmente progressivement puis un saut assez important apparaît. L'augmentation progressive est liée au terme $Fourche(Q * B(In(0)) * L(In(0)))$ qui est le coût correspondant à l'envoi de l'acquittement de sortie sur toutes les données entrantes. Tout naturellement, plus le nombre de données en entrée augmente, plus le coût est important. Le saut qu'on observe par moment est lié au terme $L(Out(0)) * OR(Q)$ qui correspond à la réunion de toutes les sorties intermédiaires de chaque branche du "merge" pour n'envoyer qu'une seule sortie. Le coût du terme $OR(Q)$ est établi à l'aide de la formule $\sup |\log_2(Q)|$, ce terme reste constant lorsque le nombre d'entrées I est compris dans l'intervalle: $2^n < I \leq 2^{n+1}$. On voit bien sur la courbe que le saut se produit lorsqu'on franchit une valeur 2^n .

La bufferisation en sortie présente les mêmes sauts que la bufferisation en entrée pour des raisons similaires. Cependant, on remarque que la consommation est constante entre deux sauts. Cela s'explique par le fait que le coût de la fourche d'acquittement n'est plus dépendant

du nombre d'entrées car la bufferisation est réalisée sur le canal de sortie. Le nombre de sortie est constant : $Fourche(B(In(0)) * L(In(0)))$.

La vitesse du circuit est plus grande lorsque le "merge" est bufferisée en entrée pour des raisons similaires à ce qui a été dit pour les précédents composants, à savoir l'étage précédent est relaxé plus tôt.

- Le multiplexeur

Le multiplexeur est un composant qui prend une entrée parmi un certain nombre et l'envoie sur sa sortie. Contrairement au "merge", le multiplexeur est contrôlé par une commande qui sélectionne l'entrée à envoyer sur la sortie.

Le coût du multiplexeur bufferisé en entrée et en sortie intermédiaire (Figure 4.7) est :

$$\begin{aligned} \text{Coût}_{\text{WCHB_entrée}}\{\text{Mux}\} &= \text{Coût}_{\text{WCHB_sortie_inter}}\{\text{Mux}\} = \\ & \text{muller}(L(G)) + L(In(0)) * \text{muller}(2) + \text{Fourche}(L(In(0))*B(In(0))) \\ & + \text{Fourche}(B(In(0))*L(In(0))) + L(In(0))*\text{muller}2R + L(In(0))*\text{NOR}(B(In(0))) + \text{muller}(L(In(0))) \\ & + \text{AND}(Q) + L(Out(0))*\text{OR}(Q) + \text{Fourche}(B(In(0))*L(In(0))*(Q-1)) \end{aligned}$$

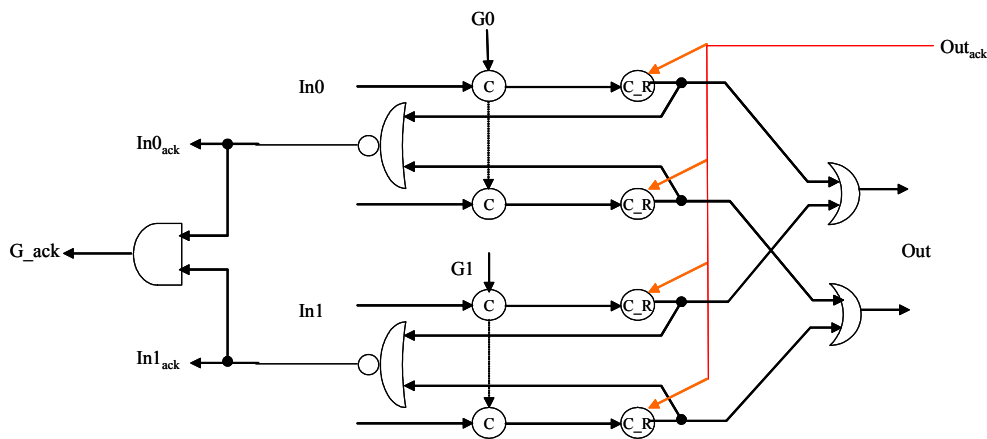


Figure 4.7 Multiplexeur bufferisé en entrée

Pour une bufferisation en sortie (Figure 4.8) le coût devient :

$$\begin{aligned} \text{Coût}_{\text{WCHB_sortie}}\{\text{Mux}\} &= \\ & \text{muller}(L(G)) + L(In(0)) * \text{muller}(2) + \text{Fourche}(L(In(0))*B(In(0))) \\ & + \text{Fourche}(B(Out(0))*L(Out(0))) + L(Out(0))*\text{muller}2R + L(Out(0))*\text{NOR}(B(Out(0))) \\ & + \text{muller}(L(Out(0))) + \text{AND}(Q) + L(Out(0))*\text{OR}(Q) + \text{INV} + \text{muller}(2) + \text{Fourche}(Q) \end{aligned}$$

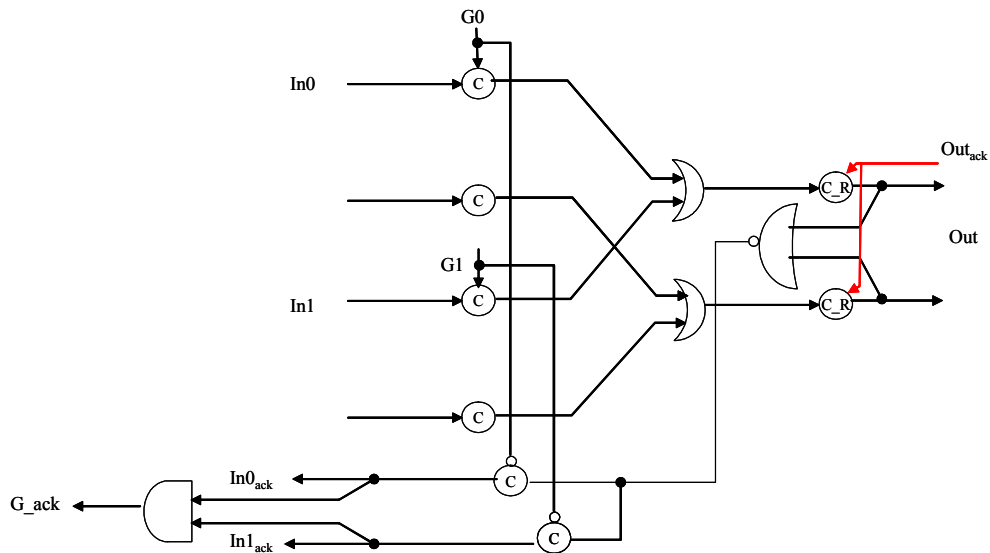


Figure 4.8 Multiplexeur bufferisé en sortie finale

Nous allons comparer les consommations d'énergie d'un multiplexeur bufferisé en entrée et d'un multiplexeur bufferisé en sortie en fonction du nombre de gardes Q (Figure 4.9). Comme dans le cas du composant "merge", le nombre de gardes Q est identique au nombre d'entrées I (il s'agit d'un multiplexeur traditionnel, c'est-à-dire avec des branches homogènes). Une garde correspond à la sélection d'une entrée qui est envoyée vers la sortie.

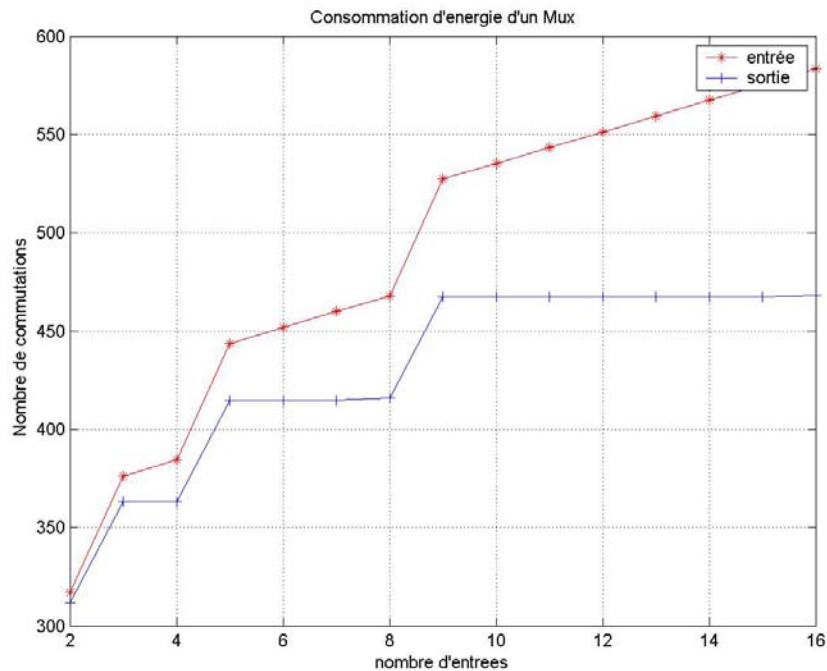


Figure 4.9 Consommation d'un Multiplexeur bufferisé en entrée et en sortie

La figure montre que la bufferisation en sortie finale du multiplexeur est mieux adaptée à la faible consommation d'énergie. De plus, l'influence du nombre d'entrée est très limitée sur l'augmentation de la consommation d'énergie. Il est avantageux pour la consommation d'énergie de bufferiser le multiplexeur en sortie plutôt qu'en entrée.

La différence qu'on note par rapport au "merge" est que le multiplexeur utilise un canal de commande pour garantir l'exclusion mutuelle. On utilise aussi ce canal pour acquitter l'entrée qui est activée.

On remarque des courbes en dent de scie, les mêmes raisons que pour le "merge" expliquent cette forme.

- Le démultiplexeur

Enfin, le démultiplexeur est un composant permettant d'envoyer une entrée sur une sortie parmi un certain nombre de sorties (la sélection est commandée).

Le coût d'un démultiplexeur bufferisé en entrée (Figure 4.10) est :

$$\begin{aligned} \text{Coût}_{\text{WCHB_sortie_entree}} \{ \text{Demux} \} = & \\ & \text{muller}(L(G)) + \text{OR}(Q) + \text{Fourche}(L(\text{In}(0)) * B(\text{In}(0))) + \text{Fourche}(L(\text{In}(0)) * B(\text{In}(0))) \\ & + L(\text{In}(0)) * \text{muller}3R + L(\text{In}(0)) * \text{NOR}(B(\text{In}(0))) + \text{muller}(L(\text{In}(0))) + L(\text{In}(0)) * \text{muller}(2) \\ & + \text{AND}(Q) + L(\text{In}(0)) * \text{Fourche}(Q) \end{aligned}$$

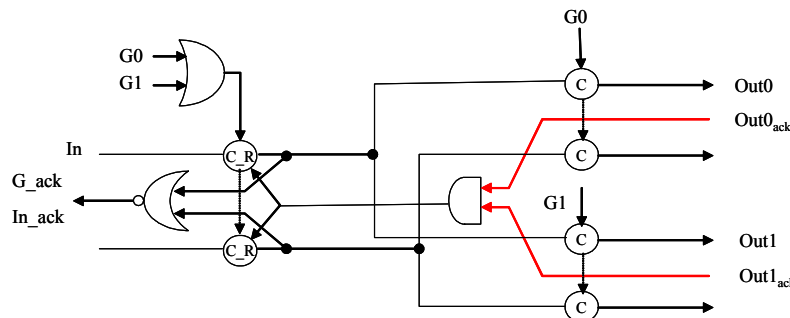


Figure 4.10 Demultiplexeur bufferisé en entrée

Le coût du démultiplexeur bufferisé en sortie finale (Figure 4.11) devient :

$$\begin{aligned} \text{Coût}_{\text{WCHB_sortie}} \{ \text{Demux} \} = & \\ & \text{muller}(L(G)) + L(\text{In}(0)) * \text{muller}(2) + \text{Fourche}(L(\text{In}(0)) * B(\text{In}(0))) \\ & + \text{Fourche}(B(\text{In}(0)) * L(\text{In}(0))) + L(\text{In}(0)) * \text{muller}2R + L(\text{In}(0)) * \text{NOR}(B(\text{In}(0))) \\ & + \text{muller}(L(\text{In}(0))) + \text{AND}(Q) + L(\text{In}(0)) * \text{Fourche}(Q) + \text{AND}(Q) \end{aligned}$$

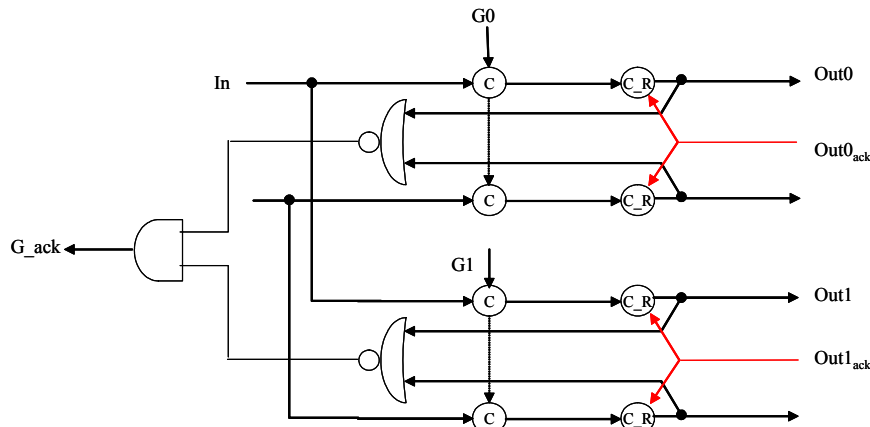


Figure 4.11 Demultiplexeur bufferisé en sortie finale

Comparons les consommations d'énergie d'un démultiplexeur bufferisé en entrée et d'un démultiplexeur bufferisé en sortie (Figure 4.12) en fonction du nombre de sorties.

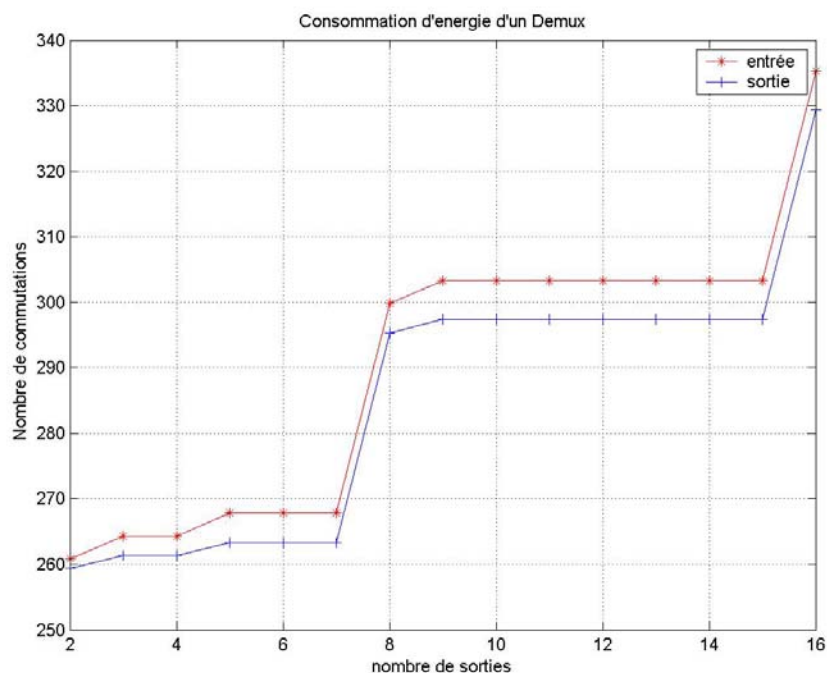


Figure 4.12 Consommation d'un démultiplexeur bufferisé en entrée et en sortie

La consommation d'énergie lorsque le démultiplexeur est bufferisé en sortie est légèrement inférieure à la bufferisation en entrée. Une double évaluation de la commande est nécessaire pour respecter les propriétés QDI pour la bufferisation en entrée (Figure 4.10). Ce double test entraîne un surcoût en consommation.

4.1.3 Bufferisation mixte

Dans les structures de choix présentant des branches hétérogènes, c'est-à-dire des branches contenant des instructions variées. Il est intéressant d'adapter la bufferisation aux branches et non plus à l'ensemble du composant. Cela se traduit par une bufferisation qui peut se faire sur les entrées d'une branche ou sur les sorties intermédiaires de celle-ci.

Considérons une structure de choix à l'image de la structure de choix présentée dans la section 2.4. Dans cette structure, la garde 0 se comporte comme un "latch", la garde 1 réalise un "fork" et la garde 2 est un "join". Pour ce cas de figure, il est intéressant de bufferiser cette structure dans son contexte. La garde 0 pourrait être bufferisée en entrée ou en sortie (le coût est identique), la garde 1 serait bufferisée en entrée (car la bufferisation en entrée d'un "fork" consomme moins qu'une bufferisation en sortie) et enfin la garde 2 aurait une bufferisation en sortie intermédiaire (car la bufferisation en sortie d'un "join" coûte moins qu'une bufferisation en entrée).

Ces optimisations apportent le plus souvent des bénéfices pour la surface, car la tendance est de bufferiser la partie du composant qui présente le moins de canaux (donc moins de logique pour la bufferisation). En revanche pour la vitesse, la bufferisation en sortie pénalise la vitesse d'un circuit car l'acquittement d'entrée est produit plus tard que la bufferisation en sortie. On libère moins vite le composant précédent.

4.1.4 Désynchronisation intra canal

Les bus de données ne présentent conventionnellement qu'un seul fil d'acquiescement. Dans un premier temps, chaque digit est acquiescé séparément lorsqu'il y a une valeur valide sur le bus, puis tous les fils d'acquiescement de tous les digits du bus sont réunis pour ne former qu'un fil d'acquiescement. Ce fil représente le fil d'acquiescement de la donnée, il est envoyé au composant qui le précède pour acquiescer celui-ci. Si maintenant on observe le composant qui reçoit ce fil d'acquiescement, on constate que ce fil est distribué à l'ensemble des digits du composant qui utilise cet acquiescement. En somme, on réalise d'une part la réunion des acquiescements de tous les digits en un seul fil d'acquiescement dans le composant qui émet l'acquiescement, puis cet acquiescement est distribué à l'ensemble des digits du composant qui reçoit l'acquiescement. On réalise donc deux manipulations qui semblent inutiles puisqu'elles sont duales, on pourrait effectivement les supprimer en acquiescant digit à digit. On propose par conséquent une nouvelle méthode d'acquiescement qu'on nomme désynchronisation intra canal qui permet l'acquiescement digit à digit (Figure 4.13). Cette désynchronisation intra canal n'a pas d'incidence sur la correction de la donnée, on laisse chaque digit de chaque canal se propager à sa propre vitesse sans être synchronisé avec les autres digits du canal. La synchronisation de ces digits se fait aux endroits où elle est indispensable (par exemple les parties combinatoires qui manipulent les digits pour produire un résultat).

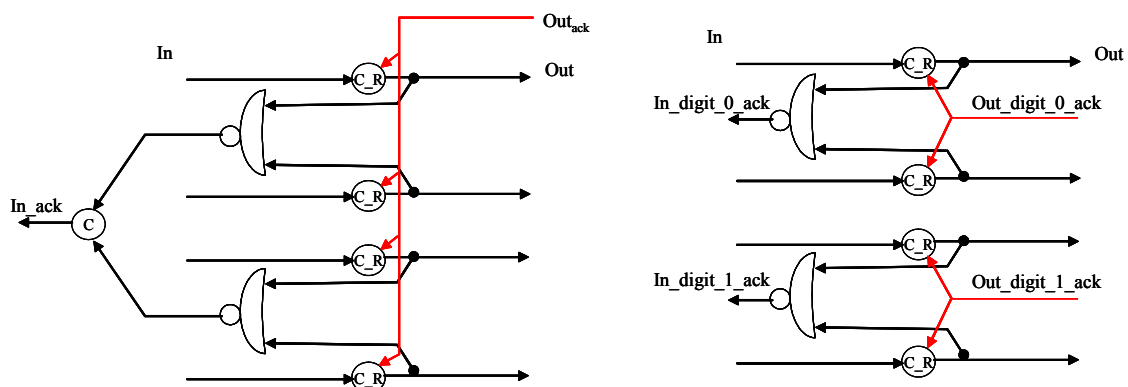


Figure 4.13 Bufferisation normale (gauche) et désynchronisation intra canal (droite)

L'équation de la consommation avec cette désynchronisation devient :

$$\text{Coût}_{\text{WCHB_entree}} \{ \text{latch} \} = \text{Coût}_{\text{WCHB_sortie}} \{ \text{latch} \} =$$

$$\text{Fourche}(\text{B}(\text{In}(0)) * \text{L}(\text{In}(0))) + \text{L}(\text{In}(0)) * \text{muller} 2 \text{R} + \text{L}(\text{In}(0)) * \text{NOR}(\text{B}(\text{In}(0)))$$

Regardons l'effet de cette optimisation sur la consommation d'énergie (Figure 4.14).

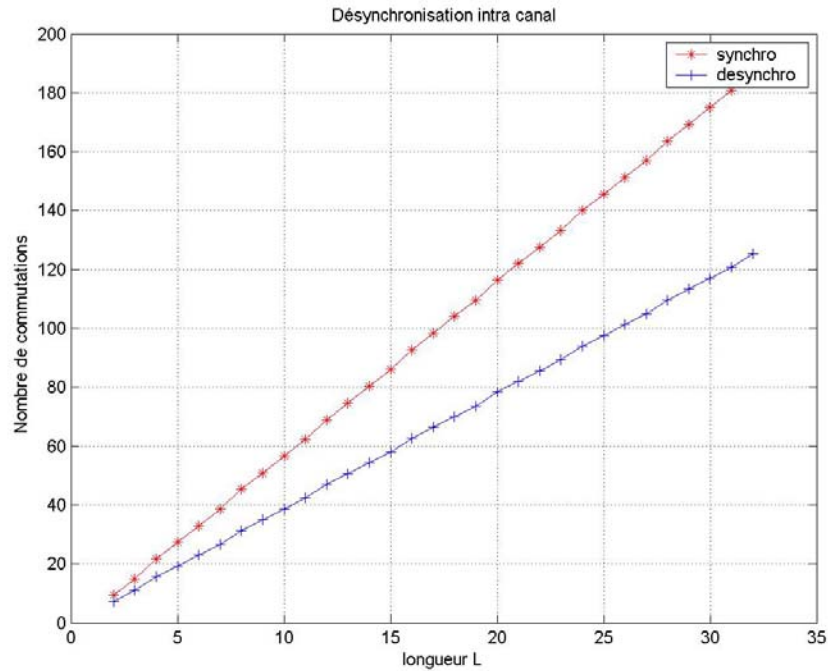


Figure 4.14 Consommation d'énergie d'un Latch acquitté digit à digit

On observe un gain en consommation d'énergie lorsqu'on désynchronise les digits d'un même canal. L'écart de consommation croît naturellement lorsqu'on augmente la taille de la donnée (donc le nombre de digits). Le transfert de données est très fréquent dans un circuit, acquitter de cette manière contribue à réduire considérablement la consommation d'énergie. On supprime une porte de muller de synchronisation des digits donc la surface se voit réduite. De plus, on laisse évoluer les digits à leurs propres vitesses, cela entraîne une optimisation de la vitesse de transfert des données. A titre d'exemple, pour un canal de 16 digits la consommation passe de 92 à 62 transitions soit une réduction d'environ 30%.

4.1.5 Ajout de synchronisation

Contrairement au cas précédent où il est intéressant de limiter autant que possible la synchronisation des données afin de réduire la consommation d'énergie, il existe certains cas où ajouter de la synchronisation permet une réduction de la consommation d'énergie. L'insertion d'une synchronisation à des endroits clés du circuit permet d'atténuer l'activité de certaines fourches. C'est le cas du multiplexeur pour lequel ajouter de la synchronisation entre la commande et les acquittements de sorties permet de réduire l'activité des fourches d'acquitterment.

Traditionnellement, l'acquitterment des sorties est envoyé à l'ensemble des commandes gardées qui utilisent cet acquitterment, de ce fait l'activité de la fourche est observée pour l'ensemble de ces gardes (Figure 4.15).

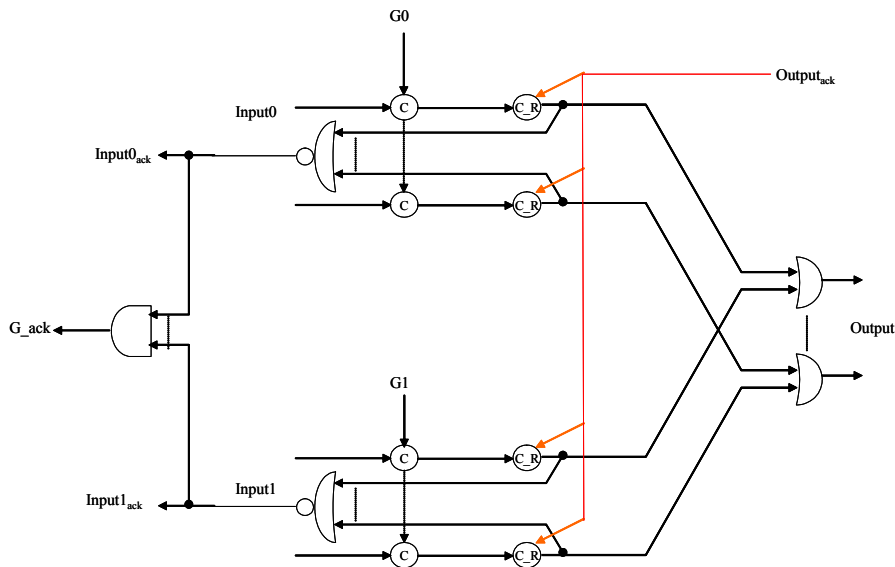


Figure 4.15 Multiplexeur non optimisé

On cherche à introduire de la logique supplémentaire de manière à limiter l'activité de ces fourches très gourmande en énergie. L'insertion de portes de muller à des endroits précis, notamment entre la commande et l'acquittement de la sortie (Figure 4.16), permet de stopper l'activité de la fourche d'acquittement des gardes qui ne sont pas exécutées.

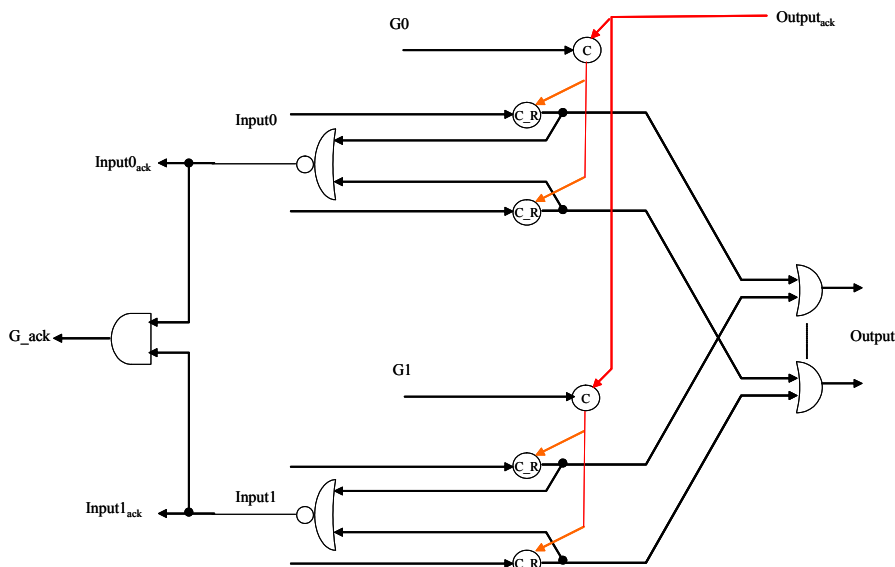


Figure 4.16 Multiplexeur avec ajout de synchronisation

Cette optimisation se résume à remplacer le terme $(L * muller(2) + Fourche(B(In(0)) * L(In(0)) * (Q - 1)))$ dans l'équation de coût du multiplexeur par le terme $(muller(2) + Fourche(Q))$. Cette substitution qu'on réalise dans l'équation du multiplexeur montre le gain substantiel d'une telle optimisation. Ce gain est d'autant plus important que les données sont grandes, et que le nombre de garde devient important. La Figure 4.17 montre le gain en consommation d'un multiplexeur dont la fourche d'acquittement de sortie est synchronisée avec la commande.

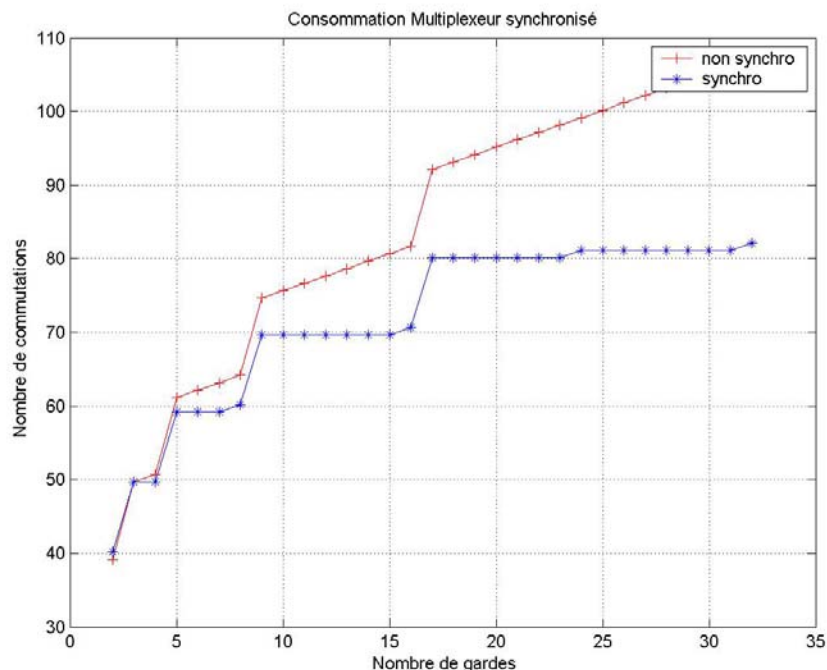


Figure 4.17 Consommation du multiplexeur synchronisé

On voit sur la figure que l'ajout de la porte de muller de synchronisation permet d'annihiler l'effet de la fourche d'acquiescement. On note une consommation d'énergie constante entre deux sauts (voir explication du "merge" 4.1.2 concernant ces sauts de consommation).

Une nette économie de la consommation d'énergie est observable. Plus le nombre de gardes est important et plus la réduction de la consommation est visible. De plus, si on augmente la taille des données manipulées alors le gain devient encore plus important.

4.1.6 Etude des opérateurs arithmétiques et logiques

Il est souvent intéressant d'analyser les opérateurs arithmétiques et logiques afin de choisir l'architecture qui consomme le moins d'énergie. De nombreuses études sont faites sur la conception de nouvelles architectures d'opérateurs arithmétiques et logiques afin de réduire la consommation d'énergie [FRAG-03]. Les outils de synthèse peuvent cibler de telles architectures de manière à optimiser la consommation d'énergie des circuits.

Par ailleurs, une étude intéressante consiste à analyser le contexte d'utilisation des opérateurs arithmétiques et logiques dans un circuit. En effet, selon l'utilisation faite d'un opérateur arithmétique ou logique, on peut implémenter l'architecture la mieux adaptée. Dans des circuits complexes tels que les microprocesseurs, on réalise souvent des opérations utilisant une constante, on peut citer l'exemple d'un compteur de programme (PC) qui additionne une adresse avec une constante. On peut aussi observer des opérateurs logiques de comparaison tels que les branchements avec condition qui comparent, dans certains cas, une donnée avec une constante (branchement si égal à 0 etc.). Il est intéressant en terme de réduction de la consommation d'énergie (et aussi de surface) d'implémenter un opérateur adapté au contexte d'utilisation au lieu d'avoir recours à un opérateur général.

Le cas d'opérateurs arithmétiques telles que l'addition et la multiplication est longuement étudié par [FRAG-03], nous ne les traitons donc pas dans le cadre de cette thèse, nous nous penchons surtout sur les opérateurs logiques. Nous allons illustrer les bénéfices apportés à la consommation d'énergie d'une implémentation adaptée au contexte, pour cela nous considérons le cas de l'opérateur logique d'égalité. Prenons le cas d'un opérateur de

branchement avec condition d'égalité entre 2 données (BEQ). Cet opérateur génère un 1 si les deux données sont égales, il génère un 0 autrement.

Le coût d'un opérateur général qui teste l'égalité de deux données est le suivant :

$$\text{Coût}\{X[B][L] = Y[B][L]\} = L * \text{muller}(2) + L * \text{OR}(B) + \text{muller}(L)$$

Où X et Y sont deux données. L et B sont respectivement la longueur et la base des données X et Y.

Si cet opérateur est utilisé pour tester l'égalité non pas de deux variables mais d'une variable avec une constante, il est plus judicieux d'utiliser une architecture appropriée. Un opérateur qui compare une donnée avec une constante présente le coût suivant :

$$\text{Coût}\{X[B][L] = \text{cste}\} = \text{muller}(L)$$

L'utilisation de cet opérateur adapté permet une économie de $(L * \text{muller}(2) + L * \text{OR}(B))$ à chaque fois que l'opérateur est sollicité. Il s'agit donc d'une optimisation judicieuse.

En somme on peut dire qu'il est possible d'optimiser les opérateurs arithmétiques et logiques en fonction du contexte dans lequel ces opérateurs sont utilisés. Cette théorie est envisageable pour tous les opérateurs arithmétiques et logiques qui prennent une valeur constante en entrée. Dans les microprocesseurs, il y a des incréments (qui correspondent à des additions entre une valeur et une constante), des compteurs, il y a aussi des opérateurs booléens tels que branchement si égal zéro, branchement si différent de zéro, branchement si supérieur à zéro, branchement si inférieur à zéro, des trap si égal à zéro etc.

4.2 Implémentation CHP orientée faible consommation d'énergie

La partie précédente a présenté les techniques d'optimisation pouvant être directement implémentées dans l'outil de synthèse de TAST de manière à synthétiser automatiquement des circuits asynchrones à faible consommation d'énergie. La partie, que nous entamons à présent, s'adresse au concepteur de circuits asynchrones. On souhaite proposer au concepteur certaines techniques de spécification CHP pour la conception de circuits asynchrones à faible consommation d'énergie. Ces conseils ont pour but de guider le concepteur dans ses choix d'implémentation, ils portent sur le codage des données et du contrôle, sur la rigueur d'écriture d'un code CHP, sur la fusion des opérateurs et enfin sur l'utilisation judicieuse des probabilités pour déséquilibrer les branches d'une structure de choix.

4.2.1 Optimisation micro architecturale

Lorsque l'estimateur d'activité montre qu'un composant présente une forte activité lors d'une simulation, il est intéressant d'apporter des optimisations à ce composant (ou ce processus) afin de réduire l'énergie dynamique. Une méthode pour optimiser un processus est de le décomposer en un ensemble de sous-processus dans le but de réduire la complexité. Seuls les sous-processus qui participent à l'exécution d'une tâche sont activés, les sous-processus qui ne font aucun travail effectif sont naturellement au repos et ne consomment pas d'énergie dynamique. On considère l'exemple d'une unité arithmétique et logique, la décomposition est celle de la Figure 4.18.

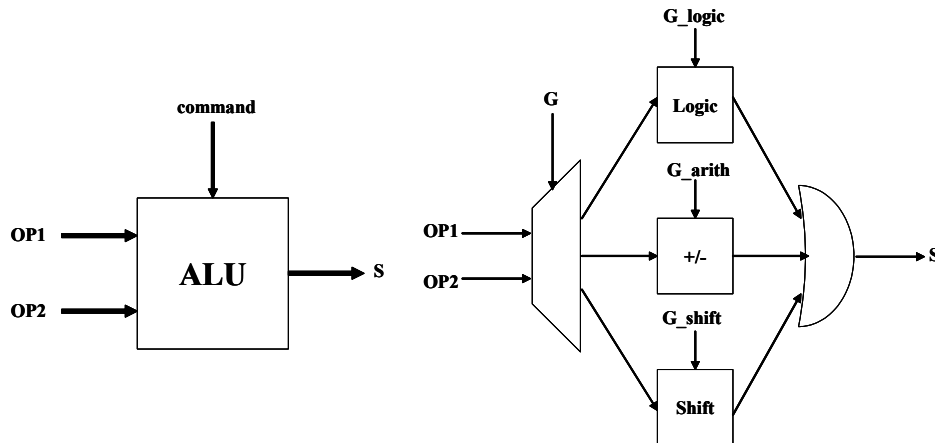


Figure 4.18 Décomposition d'une ALU

Chaque bloc de l'ALU décomposée représente un processus. Chaque processus est réduit à la plus petite complexité. Si une opération logique est réalisée, seul le chemin impliqué dans le traitement de l'opération logique correspondante présente une activité.

4.2.2 Codage des données

En asynchrone, le codage des données (qu'on note base 2, base 4 etc.) n'a pas la même signification qu'en synchrone. En effet, en asynchrone le type de codage a un effet sur la disposition des fils et sur leurs commutations. C'est la raison pour laquelle il est intéressant de trouver le codage le mieux adapté à une implémentation afin de minimiser la consommation d'énergie du circuit.

Prenons le cas d'un Latch où la consommation d'énergie est la suivante :

$$\text{Coût}_{\text{WCHB_entree}} \{\text{latch}\} = \text{Coût}_{\text{WCHB_sortie}} \{\text{latch}\} =$$

$$\text{Fourche}(B(\ln(0)) * L(\ln(0))) + L(\ln(0)) * \text{muller}2R + L(\ln(0)) * \text{NOR}(B(\ln(0))) + \text{muller}(L(\ln(0)))$$

L'équation montre que la consommation d'énergie dépend directement de la base et de la longueur de la donnée utilisée. Par conséquent, il est intéressant de connaître le meilleur compromis entre la base et la longueur afin de réduire autant que possible la consommation d'énergie. Choisissons une donnée codée sur 32 bits. En double rail, cette donnée est codée sous forme d'un multi rail MR[2][32]; si on décide de coder cette même donnée en base 4, le codage devient un multi rail MR[4][16] (la Figure 4.13 de gauche montre une donnée de 2 bits codée en MR[2] et la Figure 5.17 décrit une donnée de 2 bits codée sous forme d'un digit MR[4]). Le fait d'augmenter la base des données entraîne une réduction de la longueur des données (c'est-à-dire du nombre de digits). Essayons maintenant de voir l'évolution de la consommation d'énergie pour un certain nombre de combinaisons Figure 4.19.

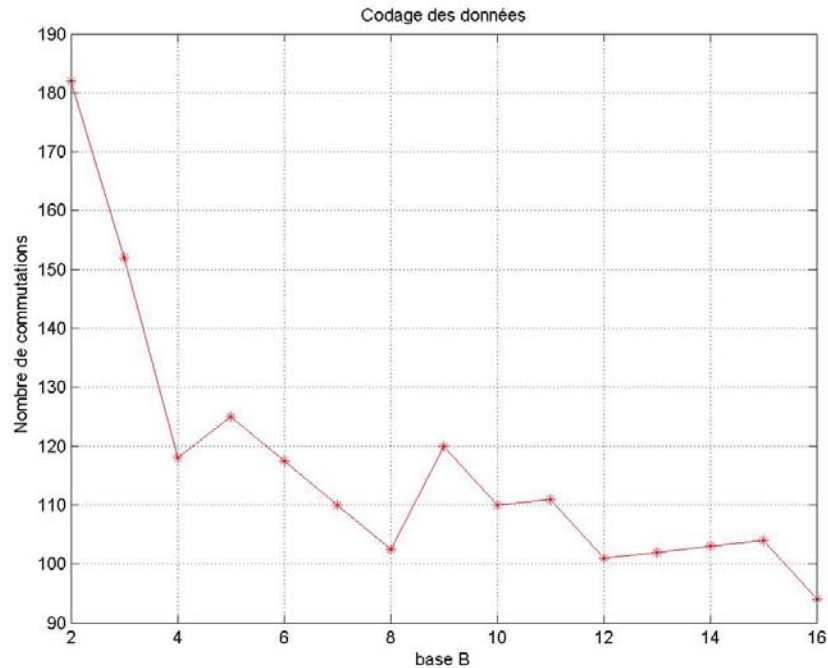


Figure 4.19 Consommation d'énergie d'un Latch en fonction de la base

Ce graphique illustre la variation de la consommation d'énergie en fonction du type de codage de la donnée. Il s'agit de la même donnée, qui contient la même information mais codée de manière différente. Il est intéressant de comprendre à quoi sont dues les fluctuations de la consommation. La tendance générale de la courbe est vue comme une forte diminution de la consommation d'énergie de la base 2 à la base 4, puis une diminution progressive à partir de la base 4. La forte diminution s'explique par le fait qu'on réduit considérablement la longueur de la donnée (32 à 16 digits) quand on passe d'une base 2 à une base 4. Au delà de la base 4, le fait d'augmenter la base se traduit par une diminution de la longueur qui est beaucoup moins prononcée.

On observe de petites fluctuations. Ceci s'explique par le fait que, par moment, le fait d'augmenter la base ne fait pas diminuer de beaucoup la longueur de la donnée, cette diminution de longueur est inférieure à la logique qu'on ajoute lorsque la base augmente (notamment la porte NOR de l'acquittement qui dépend de la base).

On note que le passage de la base 2 à la base 4 n'introduit aucun surplus de fils. On a dans les deux cas un total de 64 fils (2×32 et 4×16). Par contre au delà de la base 4, le fait d'augmenter la base se traduit par une augmentation importante du nombre de fils. On pénalise ainsi la surface du circuit si on utilise des données ayant des bases très élevées.

4.2.3 Adaptation de la taille des canaux de donnée

Les informations que nous donne l'estimateur d'activité sur les canaux de données sont utiles pour choisir la meilleure organisation d'un canal de donnée. L'activité des digits dans un canal est une information intéressante afin d'observer quels digits sont rarement ou jamais utilisés (Figure 4.20).

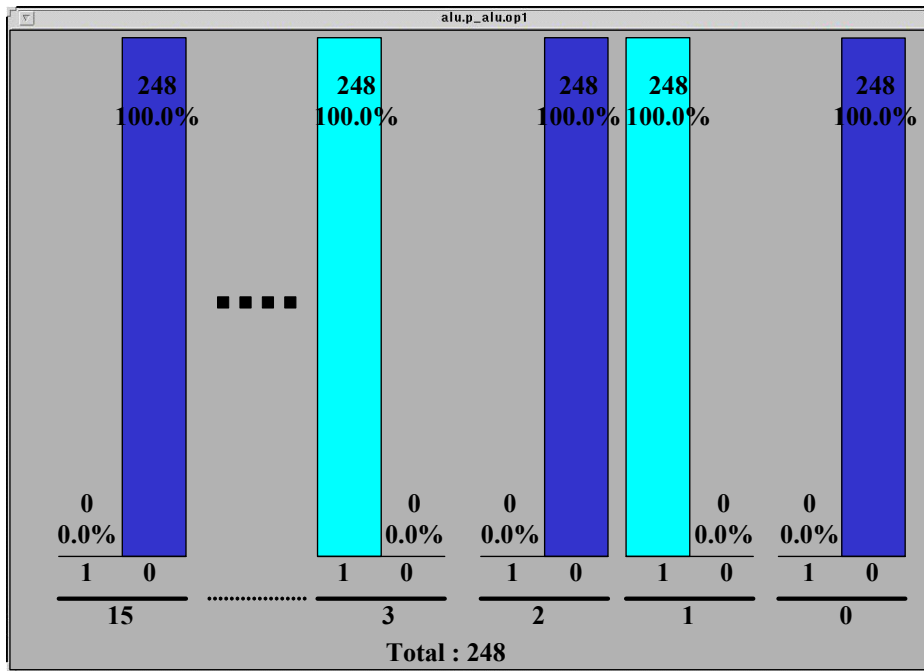


Figure 4.20 Statistique dans un canal

Grâce à cette analyse, un canal de donnée peut être découpé en un ensemble de sous canaux qui peuvent être activés conditionnellement (adaptation de la taille du canal). Par conséquent, au lieu d'envoyer tout le temps n digits dans le canal, on envoie seulement les digits nécessaires à travers le bon sous canal ($n1$ et $n2$ par exemple) (Figure 4.21).

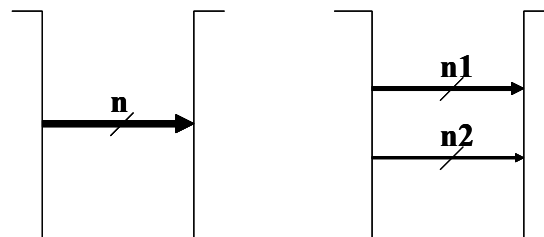


Figure 4.21 Découpage d'un canal de donnée

4.2.4 Désynchronisation inter canaux

En CHP, comme dans d'autres langages, il n'existe pas une façon unique de coder un circuit. Or, la synthèse d'un composant est directement liée à sa spécification CHP. Le circuit synthétisé obtenu est parfois plus complexe que ce qu'il ne pourrait être. Cette complexité s'explique souvent par l'ajout de synchronisation entre les canaux qui n'est pas tout le temps nécessaire. Ce paragraphe s'attache à illustrer un exemple fréquent de codage CHP qui ajoute de la synchronisation inutile lors de la synthèse. :

```
[[ A ? a , B ? b ; S0 ! a , S1 ! b ]
loop ]
```

La synthèse de cette spécification CHP assure la validité simultanée des 2 canaux en entrée ainsi que la disponibilité simultanée des 2 canaux en sortie. Ceci est réalisé par l'ajout d'une porte de muller qui crée le rendez-vous des deux acquittements de sortie (Figure 4.22).

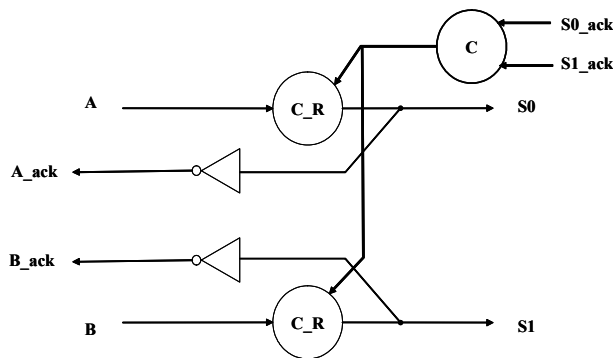
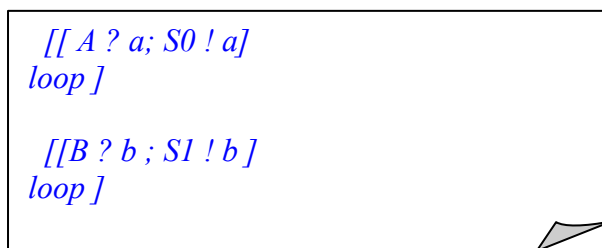


Figure 4.22 Synchronisation inter canaux

Ce circuit peut être optimisé en supprimant la synchronisation qui est tout simplement superflue. Dans cet exemple, l'entrée A est envoyée sur la sortie S0 et l'entrée B est envoyée sur la sortie S1, par conséquent il n'y a nullement besoin d'introduire une synchronisation entre l'émission de ces deux données. Une écriture plus rigoureuse serait de scinder ce processus en deux processus indépendants. Ces deux processus indépendants s'écrivent comme suit :



La synthèse de cette spécification CHP est décrite à la Figure 4.23.

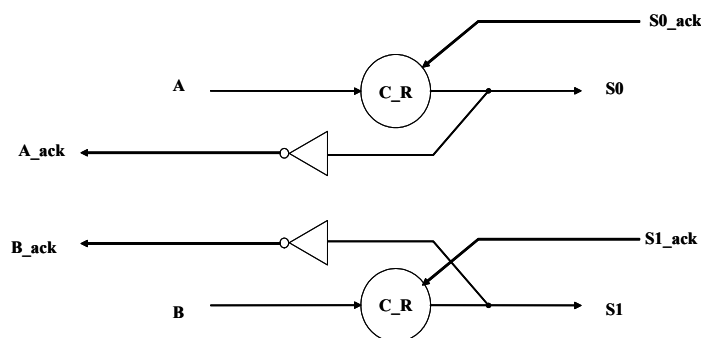


Figure 4.23 Désynchronisation inter canaux

On voit que la porte de muller (qui permet la synchronisation des acquittements de sortie) et la fourche des acquittements de sortie ont disparu. Cette façon de procéder est possible; on laisse les canaux évoluer à leurs propres rythmes sans les synchroniser inutilement. Ils seront synchronisés, s'il y a lieu d'être, dans les parties combinatoires nécessitant l'utilisation des deux canaux.

Cet exemple montre la rigueur dont le concepteur doit faire preuve lorsqu'il spécifie un circuit en CHP. Il peut réaliser un circuit mais parfois à un coût beaucoup plus important que ce qu'il

pourrait être. Une mauvaise spécification peut introduire de la synchronisation entre des données, ce qui coûte beaucoup en énergie.

4.2.5 Codage du contrôle

Les paragraphes précédents ont présenté des optimisations possibles à réaliser sur le chemin de données. Maintenant, nous allons voir l'optimisation de la consommation d'énergie du chemin de contrôle. Les structures qui nécessitent un contrôle sont les multiplexeurs commandés et les démultiplexeurs commandés. Le choix du codage du contrôle est tout aussi important que le codage des données. On cherche à savoir quel est le meilleur compromis entre la base et la longueur du codage du contrôle pour obtenir un minimum de consommation d'énergie.

Si on prend le cas d'un multiplexeur, l'équation de la consommation d'énergie est la suivante :

$$\begin{aligned} \text{Coût}_{\text{WCHB_entrée}}\{\text{Mux}\} &= \text{Coût}_{\text{WCHB_sortie_inter}}\{\text{Mux}\} = \\ &= \text{muller}(L(G)) + L(\text{In}(0)) * \text{muller}(2) + \text{Fourche}(L(\text{In}(0))*B(\text{In}(0))) \\ &+ \text{Fourche}(B(\text{In}(0))*L(\text{In}(0))) + L(\text{In}(0))*\text{muller}2R + L(\text{In}(0))*\text{NOR}(B(\text{In}(0))) + \text{muller}(L(\text{In}(0))) \\ &+ \text{AND}(Q) + L(\text{Out}(0))*\text{OR}(Q) + \text{Fourche}(B(\text{In}(0))*L(\text{In}(0))*(Q-1)) \end{aligned}$$

Analysons le comportement de la consommation d'énergie en fonction du codage de la commande (Figure 4.24). On choisit l'exemple d'un multiplexeur à 16 entrées, par conséquent la commande en entrée est codée de manière à pouvoir représenter 16 cas possibles.

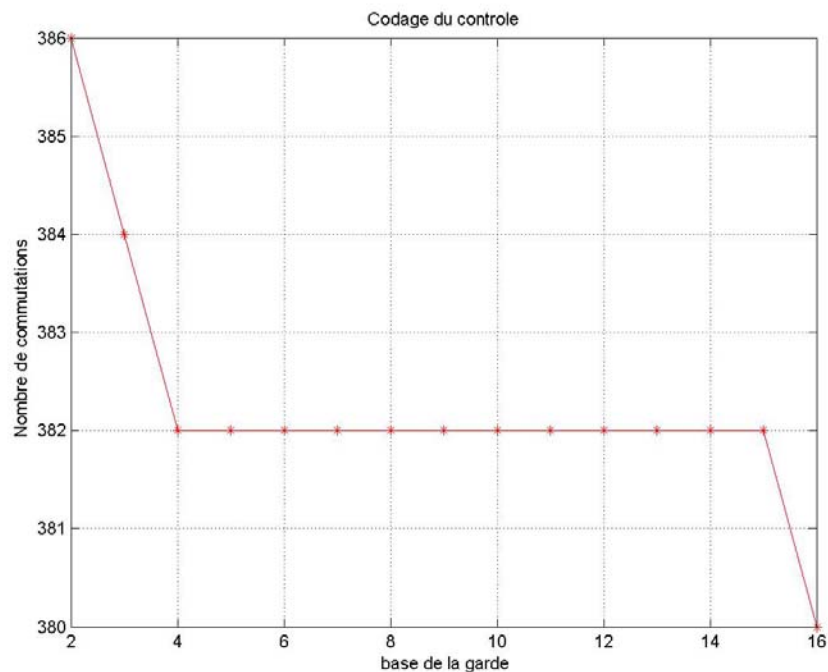


Figure 4.24 Consommation en fonction du codage de la commande

L'allure générale de la courbe montre que la consommation d'énergie décroît lorsqu'on augmente la base du canal de commande. On explique cette diminution de la consommation par la diminution de la longueur de la commande (par conséquent, l'évaluation de la commande est moins coûteuse), notamment lors du passage de la base 2 à la base 4, ou la

longueur est réduite de moitié. Il existe un pallier entre la base 4 et la base 15, ceci est dû au fait que malgré l'augmentation de la base, la longueur du canal de commande ne diminue pas. La consommation est minimale pour la base 16 car nous analysons un multiplexeur à 16 gardes, le codage le mieux adapté est le codage MR[16][1]. Il y a un fil par cas, on supprime par conséquent la logique d'évaluation de la garde.

Les circuits asynchrones, qui ont été réalisés ces dernières années [RENA-98][ABRI-01], ont le plus souvent implémenté une commande codée en MR[Gmax][1] où Gmax correspond au nombre maximum de gardes du multiplexeur. L'étude qu'on fait ici légitime ce choix de codage.

Il est tout de même possible de réduire davantage la consommation d'énergie en utilisant un type de codage single rail. Au lieu de coder la commande sous forme d'une seule commande codée en MR[Gmax][1], il est plus intéressant d'avoir plusieurs commandes, notamment Gmax commandes, codées en MR[1] (ou SR pour Single Rail). De ce fait, il n'y a plus une seule et unique commande pour toutes les gardes de la structure de choix, mais une commande par garde. De cette manière, on décorrèle les branches, on gagne ainsi sur l'évaluation de la commande et sur l'acquittement de celle-ci.

Observons le gain qu'on obtient en fonction du nombre de garde (Figure 4.25).

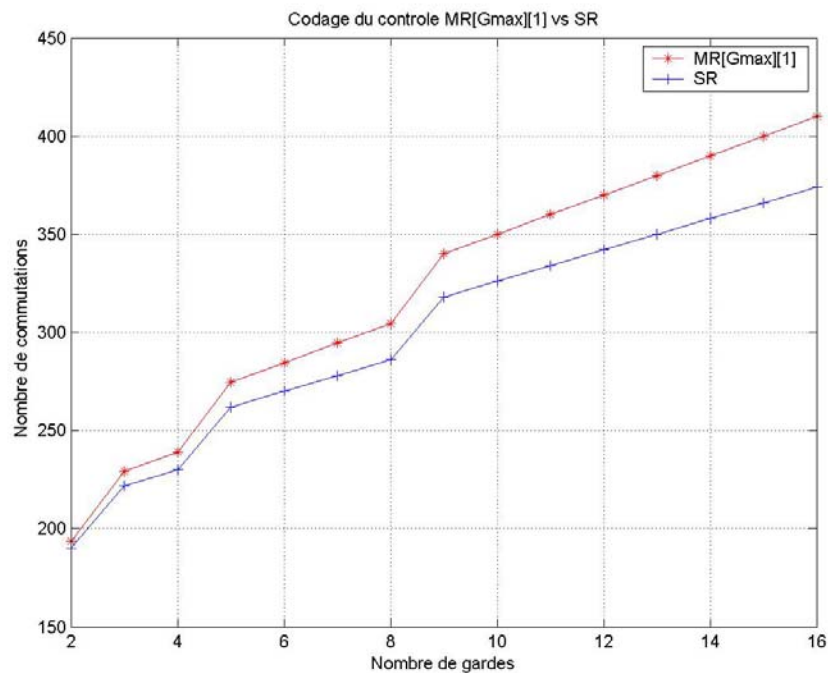


Figure 4.25 Consommation d'un multiplexeur codé en SR

La même étude pourrait être menée dans le cas d'un démultiplexeur, on aboutirait à des résultats tout à fait similaires.

4.2.6 Fusion des opérateurs

L'idée de cette fusion d'opérateur est de se dire qu'un inverseur en double rail (voir en multi rail) ne coûte rien, c'est juste une permutation de fils. Grâce à cette propriété propre au codage, on peut utiliser un opérateur (AND par exemple) pour réaliser d'autres opérateurs (OR, NOR, NAND) en inversant au besoin les entrées et/ou les sorties (Figure 4.26).

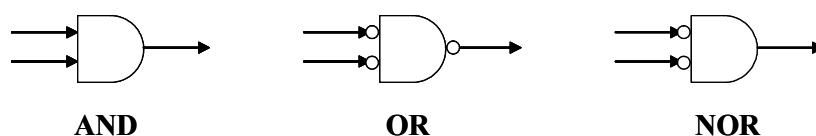


Figure 4.26 Portes logiques décrites à partir d'une porte AND

Cette théorie peut s'appliquer à l'unité d'exécution d'un microprocesseur, où plusieurs opérations peuvent être implémentées à l'aide d'un opérateur unique en complémentant correctement les entrées et/ou la sortie.

Pour les opérateurs les plus courants, on peut citer l'opérateur d'égalité et d'inégalité (Branchement si égal ou si non égal par exemple) Figure 4.27. On choisit d'implémenter un seul de ces opérateurs, par exemple l'opérateur d'égalité. A partir de cet opérateur, on réalise l'opération d'inégalité en inversant la sortie (en permutant les fils de sortie).



Figure 4.27 Portes Egal et non Egal

Un autre exemple d'opérateur souvent présent dans un microprocesseur est le test d'infériorité, de supériorité et de supériorité ou égalité. On choisit d'implémenter le test d'infériorité, on peut utiliser ce dernier pour réaliser le test de supériorité en intervertissant les données en entrée, on peut aussi effectuer le test de supériorité ou d'égalité en inversant la sortie (Figure 4.28).

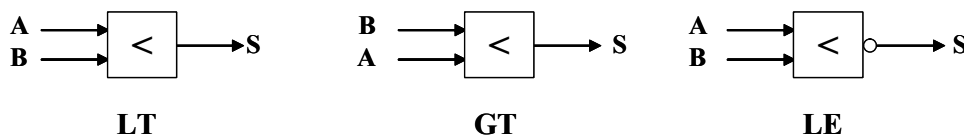


Figure 4.28 Portes inférieure, supérieure et inférieure ou égale

De plus, fusionner les opérateurs de la sorte permet de réduire la consommation d'énergie. En effet, la réduction du nombre d'opérateurs implique une réduction de la taille des fourches qui permettent d'acheminer les opérandes aux opérateurs. Les réunions en sortie qui permettent l'envoi du résultat sont de la même manière réduites. De la logique supplémentaire est cependant nécessaire pour complémenter les opérandes et/ou le résultat.

Nous traitons le cas d'un unité arithmétique et logique (Figure 4.29).

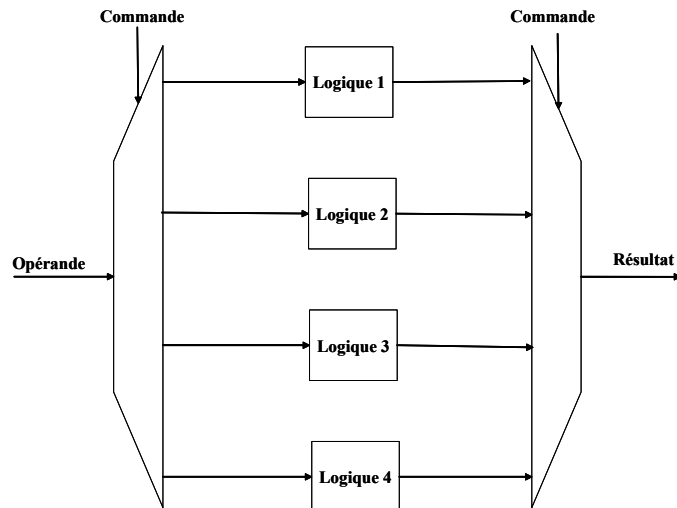


Figure 4.29 Opérateurs avant fusion

Sur la Figure 4.29, on voit bien qu'en entrée on a un démultiplexeur avec autant de branches qu'il y a d'opérateurs; ces branches servent à distribuer les opérandes aux opérateurs qui les sollicitent. De même en sortie, le multiplexeur présente autant de branches que d'opérateurs, et ceci pour pouvoir envoyer le résultat final vers une destination unique (généralement le banc de registres). Les nombreuses fourches que cela induit participe à la consommation d'énergie. La fusion des opérateurs permet de réduire ces fourches en ajoutant un minimum de logique.

Supposons qu'il soit possible de réaliser le bloc de logique 2 de la figure précédente à l'aide du bloc de logique 1, de même on peut réaliser le bloc de logique 4 à l'aide du bloc de logique 3 (en complétant l'opérande ou le résultat au besoin). On peut donc supprimer les blocs de logique 2 et 4 comme l'illustre la Figure 4.30.

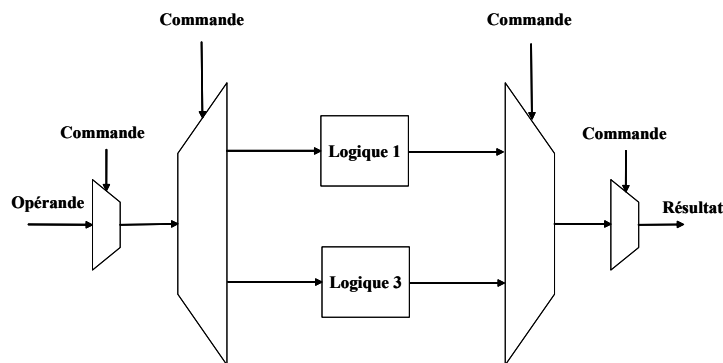


Figure 4.30 Opérateurs après fusion

On s'aperçoit que le démultiplexeur en entrée et le multiplexeur en sortie ont un nombre de branches qui a été réduit d'un facteur 2. Par conséquent leurs consommations respectives ont été diminuées. La logique ajoutée en entrée et en sortie représente de petits blocs de conversion. Leurs consommations ne sont pas importantes par rapport aux consommations des multiplexeurs et démultiplexeurs. Cette réduction peut être accentuée si on est en présence d'un nombre important de blocs logiques pouvant être fusionnés. Ce qui est le cas pour le processeur MIPS [MIPS-01] que nous considérons au Chapitre 5.

4.2.7 Réduction de la synchronisation en cas d'exclusion mutuelle

Il est possible d'aller encore plus loin dans l'optimisation, ceci en relaxant les contraintes de synchronisation si le concepteur garantit l'exclusion mutuelle entre certains blocs. En effet, en asynchrone la synchronisation (parfois inutile) déjà observée au niveau de la synchronisation systématique des canaux se manifeste également au niveau du traitement de blocs. Le concepteur a souvent tendance à commander les blocs de traitement en entrée et en sortie (Figure 4.29, Figure 4.30). Par moment, quand l'exclusion mutuelle est garantie en amont, il est possible d'assouplir la contrainte de synchronisation en sortie de ces blocs en remplaçant le multiplexeur (qui présente une architecture assez coûteuse en énergie et en surface) par un "merge" en sortie (qui est finalement une porte OR si celui-ci n'est pas bufferisé) (Figure 4.31).

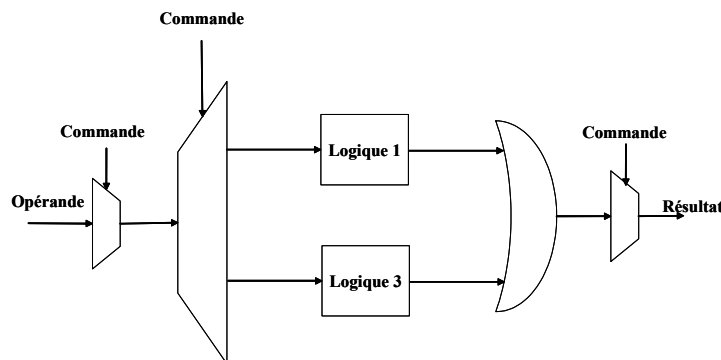


Figure 4.31 Relâchement de la synchronisation en sortie

Le résultat, produit par un des blocs logiques, active à lui seul la porte OR, contrairement au multiplexeur qui synchronise le résultat avec la commande. La consommation d'un "merge" étant inférieure à la consommation d'un multiplexeur commandé, il est préférable d'utiliser le composant "merge" en lieu et place du multiplexeur chaque fois que c'est possible (cf. §4.1.2).

4.2.8 Structures de choix en fonction des probabilités

Toutes les optimisations précédemment citées ne concernent que l'étude des équations de coût structurel. Il s'agit par conséquent d'une étude qui ne tient pas compte de l'activité relative des parties du circuit. Nous allons maintenant montrer comment exploiter les résultats obtenus à l'aide de l'estimateur d'activité et de l'estimateur de la consommation d'énergie. Le principe est de déséquilibrer les structures de choix en fonction des gardes les plus probables.

4.2.8.1 Structure de choix hiérarchique

Les structures de choix sont des composants où il est très intéressant de connaître les probabilités d'exécution des gardes. Il est possible de déséquilibrer les structures de choix en favorisant les gardes les plus utilisées. Considérons une structure de choix qui se présente sous la forme suivante :


```
@[ G = G0 => I0
  G = G1 => I1
  G = G2 => I2
  G = G3 => I3
]
```

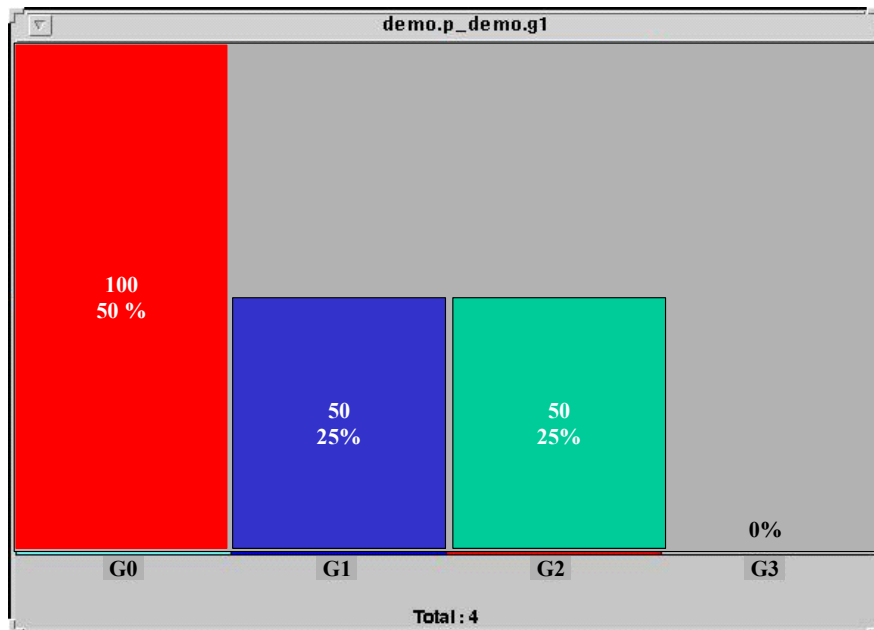


Figure 4.32 Statistiques sur les gardes exécutées

Si les résultats obtenus par l'estimateur d'activité (statistiques sur les choix) montrent que la garde G0 présente une probabilité plus élevée que les autres gardes (par exemple les statistiques de la Figure 4.32), alors il est intéressant de réorganiser la spécification CHP de la structure de choix en privilégiant la garde G0 (il est aussi possible de supprimer la garde G3). La nouvelle écriture CHP se présente sous la forme suivante :

```
@[ G=G0 => I0
  (G=G1 or G=G2 or G=G3) => @[ G=G1 => I1
                               G=G2 => I2
                               G=G3 => I3
                               ]
]
```

Cette nouvelle spécification CHP de la structure de choix entraîne une synthèse qui est différente de la spécification régulière originale. En somme, on réduit la logique du chemin de donnée de l'exécution de la garde G0 (donc la consommation d'énergie de celle-ci), on augmente par contre la complexité du chemin de donnée des autres commandes gardées. Cela signifie, que la garde G0 consomme moins dans cette configuration par rapport à la structure originale, mais que l'exécution des autres gardes coûtent plus en énergie par rapport à

l'implémentation originale. Il faut par conséquent être prudent et ne pas systématiquement transformer ce code.

Nous allons établir la condition nécessaire pour que cette transformation soit bénéfique. Cette condition est établie à l'aide de l'estimation dynamique de la consommation d'énergie. Pour ce faire, nous allons traiter le multiplexeur et le démultiplexeur séparément.

- Multiplexeur

Dans le cas du multiplexeur traditionnel, c'est-à-dire présentant des branches homogènes, si on souhaite réorganiser le code CHP en privilégiant la garde G_0 , on transforme le code CHP de la manière suivante :

```

@[ G=G0 => In0 ? x ; S ! x
  (G=G1 or G=G2 or G=G3) => @[ G=G1 => In1 ? x ; S ! x
                               G=G2 => In2 ? x ; S ! x
                               G=G3 => In3 ? x ; S ! x
                               ]
]
    
```

La synthèse qui résulte de cette modification de code (Figure 4.33) permet de réduire la consommation d'énergie de l'exécution de la garde G_0 . On observe, sur le chemin de donnée qui permet l'exécution de cette garde, une réduction de la taille de la porte AND d'acquiescement de la commande, on passe d'une taille Q (où Q est le nombre total de gardes) à une porte AND à 2 entrées. De même, la porte OR en sortie passe d'une taille Q à une taille 2.

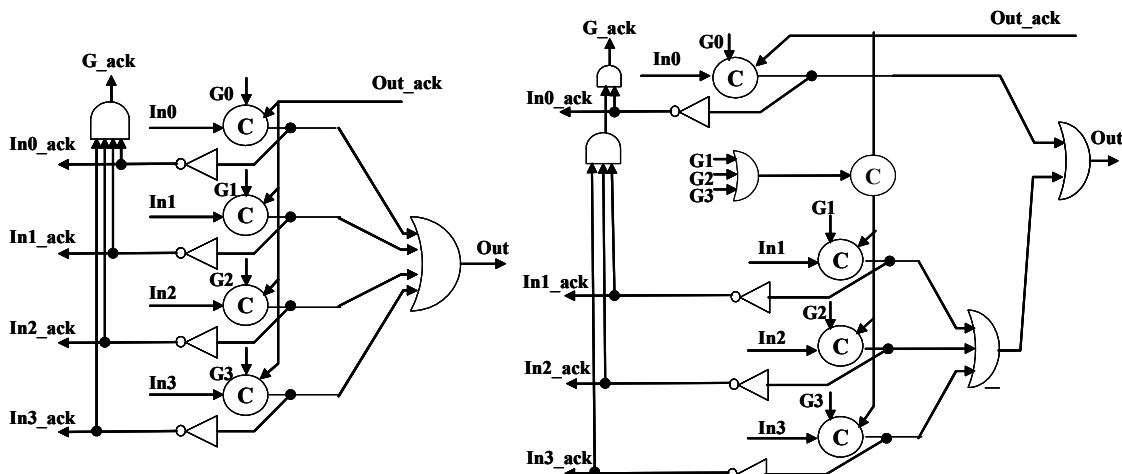


Figure 4.33 Synthèse du multiplexeur régulier (gauche) hiérarchique (droite)

On s'aperçoit néanmoins que de la logique supplémentaire est ajoutée sur le chemin de donnée des autres gardes. Une porte OR à $(Q-1)$ entrées est ajoutée pour évaluer la validité des autres gardes; une porte de muller à 2 entrées est implémentée pour réaliser le rendez-vous entre la porte OR précédemment citée et l'acquiescement de sortie; ensuite la porte AND(Q) d'acquiescement de la commande est scindée en une porte AND à $(Q-1)$ entrées et une porte AND à 2 entrées; enfin la porte OR(Q) en sortie est elle aussi fractionnée en deux, elle est devenue une porte OR à $(Q-1)$ entrées associée à une porte OR à 2 entrées.

Le coût dynamique du multiplexeur régulier est le suivant :

$$\text{Coût}_{\text{dynamique}}(\text{Mux Régulier}) = \sum_{\text{NbExec}} (G_0 \text{ or } G_1 \cdots \text{ or } G_{Q-1}) \bullet \text{Coût}(G(q)) \quad \text{où } q = 0, \dots, Q-1$$

Le coût du multiplexeur est identique quelque soit la garde exécutée. Ceci s'explique par le fait qu'on a une structure régulière et homogène.

Dans le cas d'un multiplexeur déséquilibré, le coût dynamique dépend de la garde qui est exécutée :

$$\text{Coût}_{\text{dynamique}}(\text{Mux hiérarchique}) = \sum_{\text{NbExec}} G_0 \bullet \text{Coût}(G(0)) + (G_1 \text{ or } \cdots G_{Q-1}) \bullet \text{Coût}(G(q))$$

où $q=1, \dots, Q-1$

Le coût du multiplexeur vaut le coût de la garde G_0 si celle-ci est exécutée, il vaut le coût des autres gardes si une autre garde est exécutée (sachant que les commandes gardées différentes de G_0 ont toutes le même coût).

On a vu que privilégier une garde se fait toujours au détriment des autres gardes. En effet, on réduit la logique sur le chemin de donnée de la garde G_0 mais en contre partie de la logique supplémentaire est ajoutée sur le chemin de donnée des autres gardes. Il existe, par conséquent, une condition sur le nombre d'exécution de la garde G_0 par rapport aux autres gardes pour qu'au final on observe une réduction de la consommation d'énergie par rapport à la structure de choix régulière. Ceci se traduit par l'équation suivante :

$$\text{Coût}_{\text{dynamique}}(\text{Mux hiérarchique}) \leq \text{Coût}_{\text{dynamique}}(\text{Mux Régulier})$$

\Leftrightarrow

$$X \cdot \text{Coût}(G_0) + Y \cdot \text{Coût}(\overline{G_0}) \leq (X + Y) \cdot \text{Coût}(G)$$

Où X est le nombre d'exécution de la garde G_0 , Y le nombre d'exécution d'une garde autre que la garde G_0 , représentée par $\overline{G_0}$ (G_1, G_2, \dots), et enfin G est l'exécution d'une garde quelconque (G_0, G_1, \dots).

La condition nécessaire et suffisante pour réduire la consommation en déséquilibrant le multiplexeur régulier en multiplexeur hiérarchique repose sur le nombre d'exécution X par rapport au nombre d'exécution Y . Cette condition se traduit par l'inégalité suivante :

$$X \geq \frac{(1 + L) \cdot \text{OR}(Q - 1) + L \cdot \text{OR}(2) + \text{muller}(2) + \text{AND}(Q - 1) + \text{AND}(2) - \text{AND}(Q) - L \cdot \text{OR}(Q)}{\text{AND}(Q) + L \cdot \text{OR}(Q) + \text{Fourche}(QBL) - \text{AND}(2) - L \cdot \text{OR}(2) - \text{Fourche}(BL)} \cdot Y$$

Où B et L sont respectivement la base et la longueur des données (sachant que pour un multiplexeur régulier homogène, toutes les données ont la même base et la même longueur L). Q correspond au nombre de branches de la structure de choix.

Illustrons ceci par un exemple de multiplexeur à 16 branches ($Q=16$) manipulant des données de base $B=2$ et de longueur $L=4$. On fixe le nombre d'exécution Y à 50 et on essaie de trouver le nombre d'exécution minimal X pour que la structure de choix hiérarchique soit bénéfique en terme de consommation d'énergie par rapport au multiplexeur régulier (Figure 4.34).

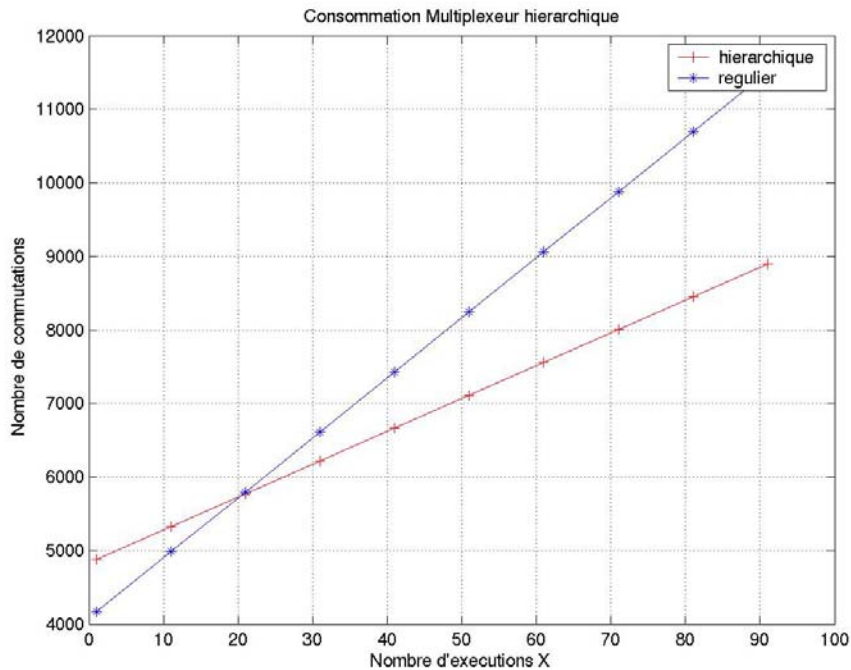


Figure 4.34 Multiplexeur hiérarchique vs multiplexeur régulier

Cette figure montre qu'il y a bénéfice à déséquilibrer le multiplexeur si le nombre d'exécution de la garde G0 est supérieur à 20 (sachant qu'on avait fixé l'exécution des autres gardes à 50) soit 28% d'exécution de la garde privilégiée par rapport aux autres gardes. On note que pour 90 exécutions de la garde G0 (soit 64% d'exécution de la garde la plus exécutée sur l'exécution totale), on obtient une réduction de la consommation d'énergie de 21,5% en utilisant un multiplexeur hiérarchique.

- Démultiplexeur

Si on considère le cas d'un démultiplexeur, on souhaite aussi établir la condition nécessaire et suffisante pour permettre une réduction de la consommation en déséquilibrant le code CHP. Le code CHP du démultiplexeur après déséquilibre s'écrit comme suit :

```

@[ G=G0 => In ? x ; S0 ! x
  (G=G1 or G=G2 or G=G3) => @[ G=G1 => In ? x ; S1 ! x
                               G=G2 => In ? x ; S2 ! x
                               G=G3 => In ? x ; S3 ! x
                               ]
  ]
    
```

Le résultat de la synthèse de la spécification CHP déséquilibrée du démultiplexeur est montré à la Figure 4.35.

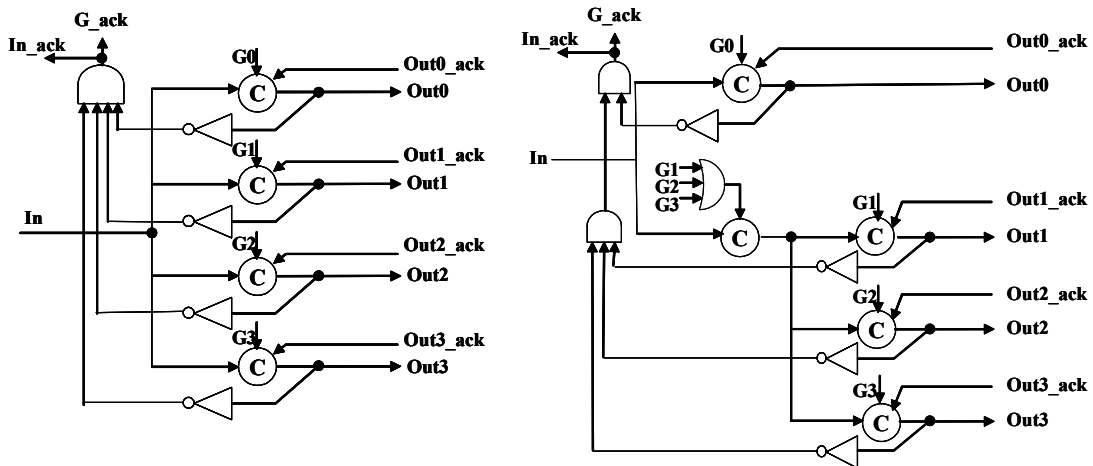


Figure 4.35 Synthese demultiplexeur hierarchique

La condition nécessaire et suffisante sur le nombre d'exécutions de la garde G0 (le nombre X) par rapport au nombre d'exécutions d'une autre garde (le nombre Y) est la suivante :

$$X \geq \frac{OR(Q-1) + Fourche(BL) + L.muller(2) + AND(Q-1) + AND(2) + L.Fourche(Q-1) - AND(Q - L.Fourche(Q))}{AND(Q) + L.Fourche(Q) - AND(2)} \cdot Y$$

Si on considère l'exemple d'un demultiplexeur à 16 branches (Q=16) qui utilise des données d'une base B=2 et de longueur L=4, alors la condition pour qu'une telle optimisation puisse se faire est montrée à la Figure 4.36.

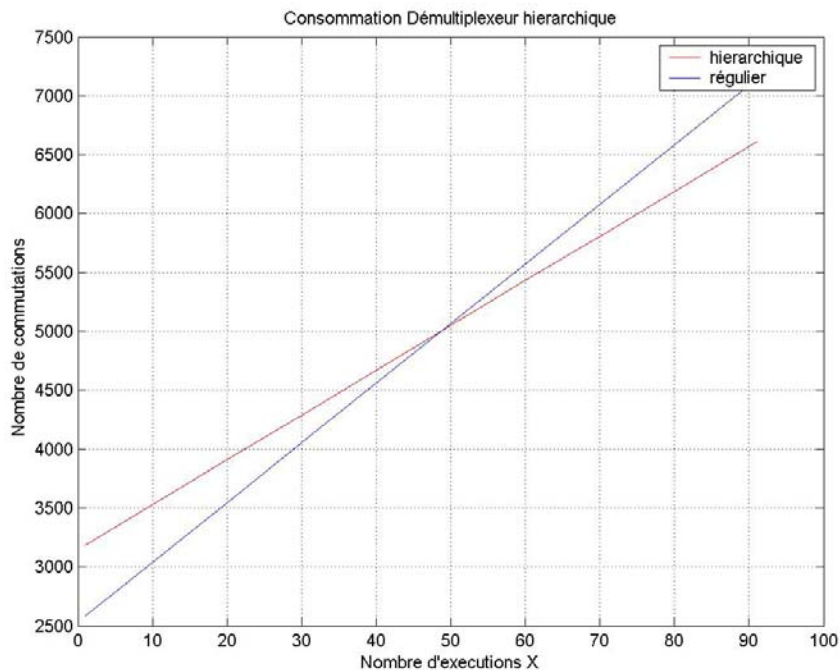


Figure 4.36 Demultiplexeur hierarchique vs demultiplexeur régulier

La figure montre qu'on peut envisager de déséquilibrer la structure du demultiplexeur pour un nombre d'exécution X supérieur à 48 (soit à peu près 50% d'exécution de la garde la plus exécutée). On note que pour 90 exécutions de la garde G0 (soit 64% d'exécution de la garde la

plus exécutée sur l'exécution totale), on obtient une réduction de la consommation d'énergie de 8% en utilisant un démultiplexeur hiérarchique.

On remarque que la consommation d'un multiplexeur est supérieure à la consommation d'un démultiplexeur pour des probabilités d'exécution identiques. Cela s'explique par le fait que la réunion des sorties intermédiaires en une seule sortie (à l'aide de portes OR) dans le multiplexeur coûte plus que les fourches du démultiplexeur qui permettent d'acheminer les données.

On peut donc dire qu'il faut être vigilant et ne pas adopter systématiquement la structure hiérarchique. Il faut s'assurer qu'on a un nombre d'exécution minimal de la garde la plus exécutée par rapport à l'exécution des autres gardes pour que cette architecture soit rentable.

4.2.8.2 Structure de choix hiérarchique avec commandes découpées

On peut optimiser davantage la consommation d'énergie en découpant le canal de commande en plusieurs sous canaux (informations obtenues par l'estimateur d'activité en considérant les statistiques des digits à l'intérieur d'un canal) (Figure 4.37). Seuls les digits nécessaires sont envoyés en fonction de la garde qui est sélectionnée. C'est une méthode qui permet de réduire le nombre de digits à envoyer et de réduire la complexité de l'acquittement.

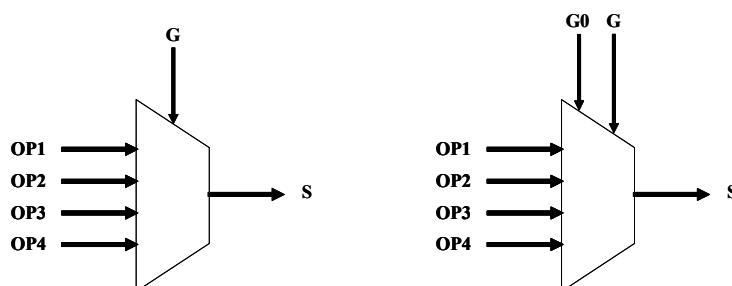


Figure 4.37 Canal de commande découpé

Le code CHP s'écrit:

```
@[ G0 => I0
  G0' => @[ G = G1 => I1
            G=G2 => I2
            G=G3 => I3
          ]
]
```

Où G_0 est une commande séparée de la garde G , et G_0' correspond au complément de la garde G_0 .

La synthèse de cette nouvelle spécification CHP du multiplexeur est illustrée à la Figure 4.38.

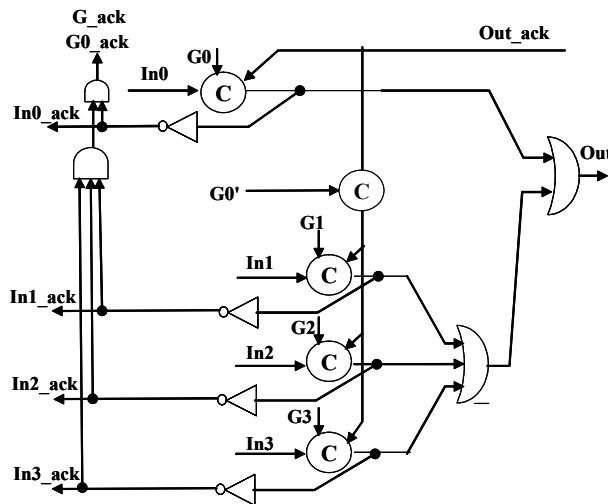


Figure 4.38 Multiplexeur hiérarchique avec commande découpée

Découper la commande en deux commandes permet de réduire la consommation d'énergie de l'exécution de la garde G0 ainsi que l'exécution des autres gardes. En ce qui concerne la garde G0, on envoie seulement une commande codée en MR[2] (ou dual rail) au lieu d'une commande qui permet le codage de toutes les branches de la structure de choix; on simplifie aussi l'évaluation de la garde (il y a un seul fil). Pour ce qui est des autres commandes gardées, on simplifie l'évaluation de la garde (G1 or G2 ...) en un seul fil (ce fil est le second fil du Double Rail de la commande G0).

Si on prend le cas du multiplexeur hiérarchique, et qu'on découpe la commande alors le résultat de cette optimisation est le suivant (Figure 4.39).

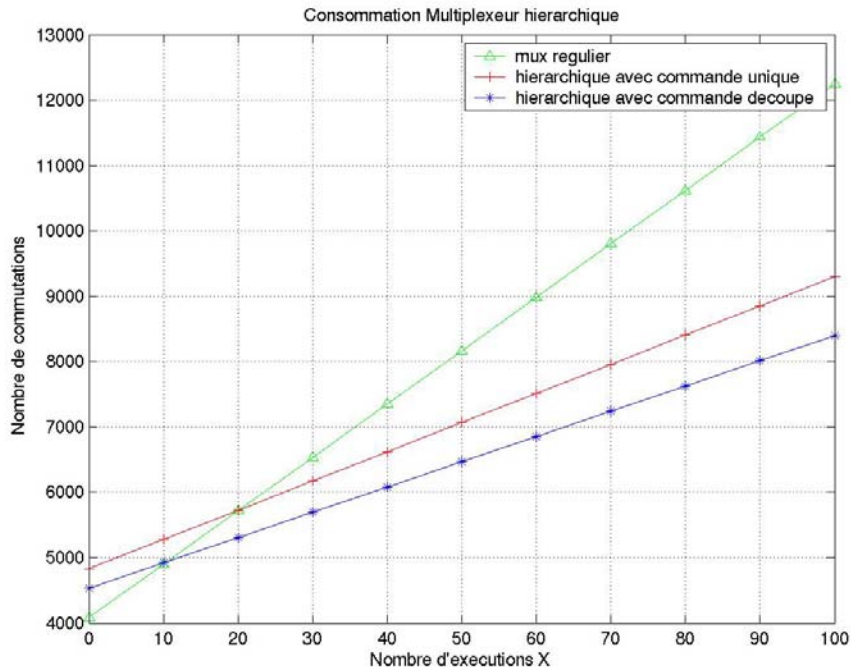


Figure 4.39 Structure hiérarchique avec commande découpée

On voit bien sur la figure que la consommation d'énergie dans le cas d'un multiplexeur hiérarchique avec commande découpée est réduite par rapport à un multiplexeur avec commande hiérarchique non découpée. Grâce au découpage de la commande, la réduction de

la consommation d'énergie a diminué de 30% par rapport au multiplexeur original régulier (pour un nombre d'exécution de G0 de 90 soit 64% d'exécution de la garde la plus exécutée). Il faut aussi remarquer qu'on ne considère que le composant qui reçoit la commande découpée. A la réduction de la consommation d'énergie du multiplexeur grâce au découpage de la commande s'ajoute une réduction de la consommation d'énergie du composant qui émet la commande (en l'occurrence le décodeur). Etant donnée que le canal de commande est découpé en sous canaux, l'émission d'un sous canal coûte moins que l'émission d'un canal de grande taille. Par conséquent on gagne en consommation sur le composant qui émet la commande.

On peut optimiser davantage la consommation d'énergie en utilisant un codage single rail (SR) comme on l'a d'ailleurs montré dans la partie 4.2.5, partie qui traitait le codage du contrôle. Il est intéressant de coder la seconde commande (c'est-à-dire celle ne concernant pas G0) en SR. On réduit la complexité d'évaluation de celle-ci et la porte AND à 2 entrées d'acquittement de la seconde commande. Il faut tout de même conserver le codage Double Rail pour la commande G0 pour ne pas réintroduire la complexité d'évaluation de la garde (G1 or G2 ...) qu'on a justement supprimée en utilisant un codage double rail pour la commande G0 (Figure 4.40).

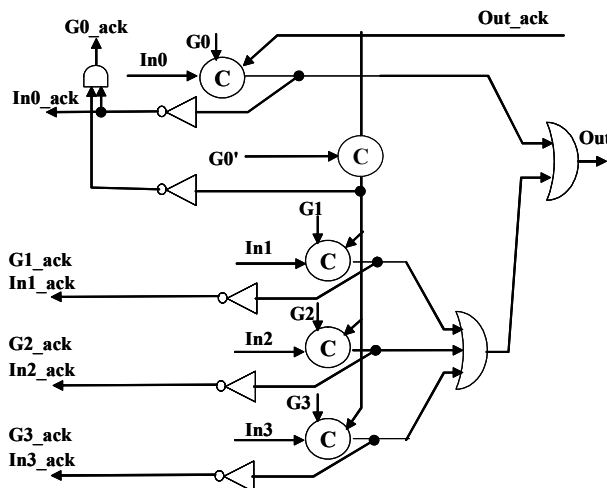


Figure 4.40 Structure hiérarchique avec commande découpée avec la seconde commande codée en SR

On vient de montrer l'utilité d'observer les probabilités d'exécution des gardes lors d'une simulation donnée (à l'aide de l'estimateur d'activité) et la consommation d'énergie (à l'aide de la méthodologie d'estimation de la consommation d'énergie). Celles-ci nous permettent de réorganiser le code CHP en fonction de la garde présentant la plus forte probabilité. Ce déséquilibre permet un gain substantiel en consommation d'énergie si la condition sur le nombre d'exécution de la garde privilégiée est satisfaite. Ce déséquilibre a été réalisé pour une seule garde, mais on peut déséquilibrer davantage la structure de choix en agissant de la même manière sur la sous structure de choix comme l'illustre la spécification ci-dessous :


```
@[ G0 => I0
  G0' => @[ G1 => I1
    G1' => @[ G=G2 => I2
      G=G3 => I3
    ]
  ]
]
```

On peut ainsi récursivement établir une hiérarchie aussi importante que les probabilités et la condition de déséquilibre sont satisfaites.

4.3 Conclusion

Nous avons, dans ce chapitre montré l'utilité des équations de coûts établies au Chapitre 2. Ces dernières nous ont permis de comparer les consommations d'énergie de différentes implémentations possibles. Ces analyses et les résultats obtenus ont permis de fournir des informations à l'outil de synthèse d'une part et au concepteur de circuits asynchrones d'autre part. De plus, nous avons montré comment exploiter les résultats obtenus par l'estimateur d'activité (statistiques) et par la méthodologie d'estimation de la consommation d'énergie pour réorganiser une spécification CHP en vue d'obtenir un circuit final présentant la plus petite consommation d'énergie. Les estimations et les techniques d'optimisation pour concevoir des circuits à faible consommation d'énergie ont été, jusqu'à maintenant, appliquées à de petites structures. Le prochain chapitre montre l'utilité de ces estimations et des techniques d'optimisation pour la réalisation d'un microprocesseur asynchrone.

Chapitre 5

Application :

Réalisation d'un Mini Processeur Asynchrone à Faible Consommation d'Energie

Les chapitres précédents ont formalisé les techniques d'estimation et d'optimisation de la consommation d'énergie des circuits asynchrones QDI. Nous allons dans cette partie mettre en application ces théories à la réalisation d'un exemple complet : un mini processeur. Le mini processeur que nous avons implémenté est un sous ensemble du processeur MIPS 4Ksc [MIPS-01].

Le paragraphe 5.1 présente les différents blocs du mini processeur et la forme générale du code CHP associé à chaque bloc. Le paragraphe 5.2 liste le jeu d'instructions de ce mini processeur. Les étapes d'optimisations sont présentées au paragraphe 5.3 et les résultats de l'estimation de la consommation d'énergie sont présentés à la section 5.4. Enfin le paragraphe 5.5 présente l'implémentation finale du processeur.

5.1 Architecture du Mini Processeur

Le mini processeur est un processeur RISC de 32 bits. Il s'agit d'un sous ensemble du processeur MIPS 4Ksc de la famille MIPS [MIPS-01]. Nous avons choisi un processeur MIPS car il s'agit d'une famille de processeur très connue et souvent utilisée pour être implémentée en asynchrone [MART-97], [TAKA-97]. De plus, nous avons à notre disposition toute la documentation concernant ce processeur. Il a donc été plus facile pour nous de choisir cette famille de processeur.

Le mini processeur est composé d'un cœur de processeur et d'éléments mémorisants. Le cœur de processeur est constitué de 5 composants principaux : le compteur de programme (PC), le décodeur (Decode), l'interface d'exécution (Exec interface), l'unité d'exécution (Exec unit) et l'unité de multiplication et division (MDU); les éléments mémorisants sont le banc de registres (Register), la mémoire instruction (Instr Memory) et la mémoire donnée (Data Memory). L'architecture du mini processeur est décrite à la Figure 5.1.

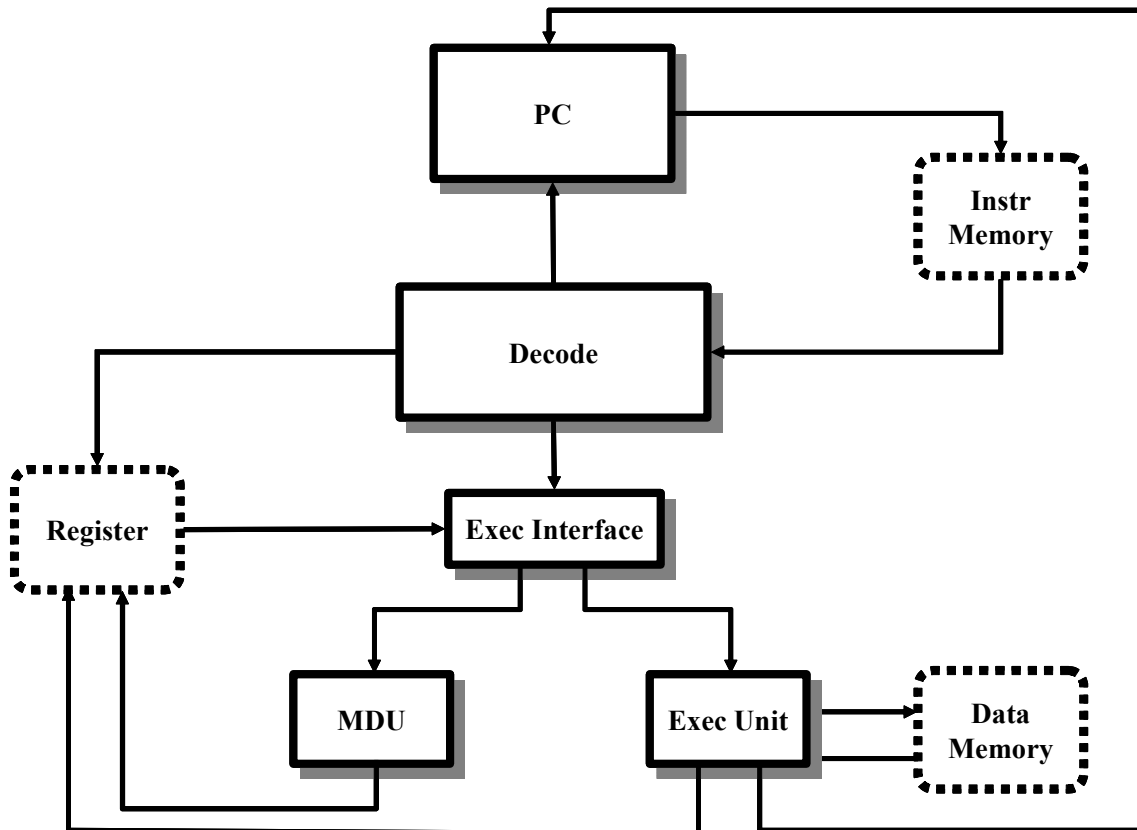


Figure 5.1 Architecture du mini processeur

Les éléments principaux constituant le cœur du processeur sont représentés en carré plein et les éléments mémorisants sont en pointillés. Les éléments principaux du cœur du processeur sont décrits en CHP synthétisable alors que les éléments mémorisants sont spécifiés à l'aide d'un CHP fonctionnel non synthétisable (la synthèse des blocs mémorisants a été réalisée à la main). La suite de ce paragraphe explicite tous ces blocs.

5.1.1 Le compteur de programme (PC)

Le compteur de programme contient un registre PC qui correspond à l'adresse de l'instruction à aller chercher en mémoire. L'adresse contenue dans ce registre peut, par la suite, être incrémentée, ajoutée à un offset ou remplacée par une adresse de branchement. L'opération effectuée dépend de l'instruction exécutée.

Le code CHP permettant la réalisation de ce bloc se présente sous la forme suivante :

```
[ Instr_PC ? G ;
  @[ G0 => Address ! pc + 4
    G1 => Address ! pc + offset
    G2 => Address ! jump_imm
    G3 => Address ! jump_reg
  ]
]
```

La garde G0 permet d'incrémenter l'adresse du PC (l'adresse est incrémentée de 4 octets), G1 additionne le PC à une valeur immédiate correspondant à un offset, G2 permet le saut à une

adresse immédiate et G3 se branche à une adresse contenue dans un registre du banc de registres.

5.1.2 Le décodeur (decode)

Le décodeur est un composant qui reçoit une instruction depuis la mémoire instruction et, comme son nom l'indique, décode l'instruction. Le décodeur permet d'envoyer les commandes aux divers composants du microprocesseur pour dicter leurs comportements.

Le CHP du décodeur s'écrit de la façon suivante :

```
[ Instruction ? G;
@[ G0 => Instr_PC ! G[31...26],
    Cmd_reg ! "111",
    Address_mode ! '0',
    Cmd_exec ! "001"

    G1 => Instr_PC ! G[31...26],
    Cmd_reg ! "111",
    Address_mode ! '0',
    Cmd_mdu ! "010"

    G...
]
]
```

Le code CHP ci-dessus présente deux cas d'émission de commande. Le décodeur final contient autant de cas qu'il existe d'instructions à décoder. La garde G0 envoie une commande au PC (Instr_PC), une commande au banc de registres (cmd_reg) pour dicter le nombre de lecture et/ou écriture (à cette commande sont associées les adresses des registres à lire et/ou écrire), une commande pour l'interface d'exécution (Address_mode) et une commande à l'unité d'exécution pour préciser l'opération à exécuter (cmd_exec). La garde G1 envoie le même genre de commandes à l'exception de la commande cmd_exec qui est remplacée par la commande cmd_mdu car il s'agit d'une instruction qui implique un opérateur de l'unité de multiplication et division.

5.1.3 L'interface d'exécution (Exec Interface)

L'interface d'exécution permet d'aiguiller les opérandes d'une instruction vers les unités de calculs. Les opérandes peuvent provenir du banc de registres ou d'une valeur immédiate contenue dans l'instruction. Les unités de calculs sont l'unité d'exécution (opérations arithmétiques et logiques etc.) et l'unité de multiplication/division (MDU) que nous présentons dans la suite de ce chapitre.

Le CHP de l'interface d'exécution est de la forme suivante :

```
[ Address_mode ? G ;
@[ G0 => Bus_1 ? Bus_1_var,
      Imm ? Imm_var ;
      OP1_exec ! Bus_1_var,
      OP2_exec ! Imm_var

      G1 => Bus_1 ? Bus_1_var,
            Bus_2 ? Bus_2_var;
            OP1_exec ! Bus_1_var
            OP2_exec ! Bus_2_var

      G2 => Bus_1 ? Bus_1_var,
            Bus_2 ? Bus_2_var;
            OP1_mdu ! Bus_1_var
            OP2_mdu ! Bus_2_var

      ]
]
```

Selon la garde, l'interface d'exécution permet d'envoyer les données vers l'unité d'exécution (OP1_exec, OP2_exec) ou vers la MDU (OP1_mdu, OP2_mdu). Ces données peuvent venir du banc de registres (Bus_1, Bus_2) ou d'une valeur immédiate contenue dans l'instruction (Imm).

5.1.4 L'unité d'exécution (Exec Unit)

L'unité d'exécution est un bloc de taille importante qui contient des opérateurs arithmétiques (addition etc.), logiques ("and" logique etc.), de load/store (LW, SW) et de branchement avec ou sans condition (BEQ, JR etc.).

Le code CHP de l'unité d'exécution se présente comme suit:

```
[ cmd_exec ? G;
@[ G0 => OP1_exec ? OP1_var,
      OP2_exec ? OP2_var;
      Result ! (OP1_var and OP2_var)

      G1 => OP1_exec ? OP1_var,
            OP2_exec ? OP2_var;
            Result ! (OP1_var + OP2_var)

      G...
      ]
]
```

Le code CHP illustre une addition et un "and" logique de l'unité d'exécution. Le résultat (Result) est envoyé au banc de registres.

5.1.5 L'unité de multiplication et division (MDU)

L'unité de multiplication et division (MDU) est un bloc complexe qui intègre des opérations de multiplication, de division, de transfert de données et de cryptographie. La MDU que nous avons implémentée ne contient pas d'opération de division.

La forme générale de la spécification CHP de la MDU s'écrit:

```
[ Cmd_mdu ? G;  
@[ G0 => OP1_exec ? LO_register  
  
    G1 => Result ! HI_register  
  
    G2 => OP1_mdu ? OP1_var,  
        OP2_mdu ? OP2_var;  
        Result ! OP1_var * OP2_var  
    G...  
    ]  
]
```

Le code CHP présente une instruction de MTLO (Move To LOw) qui écrit une donnée provenant d'un registre du banc dans le registre LO de l'unité MDU; la garde G1 présente une lecture du registre HI de la MDU; enfin la garde G2 réalise une multiplication.

5.1.6 Le banc de registres

A l'instar du processeur MIPS4ksc, le mini processeur contient un ensemble de 32 registres de 32 bits. Le registre 31 est un registre qui contient l'adresse de retour de saut lorsqu'un branchement avec liaison est effectué. On note que l'adresse du compteur de programme n'est pas contenue dans un registre du banc de registres général mais dans un registre séparé dans le composant PC.

Le banc de registres a été spécifié en CHP fonctionnelle non synthétisable. Celui-ci a été synthétisé à la main pour des raisons de compacité. Le code fonctionnel se présente sous la forme suivante:

```
[ Cmd_reg ? cr;  
    @[ cr[1..0] = "00" => skip  
      cr[1..0] = "01" => bus_1 ! registre[rs]  
      cr[1..0] = "11" => bus_1 ! registre[rs]  
                          bus_2 ! registre[rt]  
    ];  
    @[ cr[2] = '0' => skip  
      cr[2] = '1' => result ? registre[rd]  
    ]  
]
```

La variable registre (type tableau) représente un banc de 32 registres de 32 bits. Les canaux rs, rt et rd sont les adresses de lecture et d'écriture des registres.

Le banc de registres permet de lire un ou deux registres en parallèle et/ou écrire un résultat (issu de l'unité d'exécution ou de la MDU) dans un registre.

5.1.7 Les mémoires

Le mini processeur est implémenté en utilisant une architecture de Harvard où les mémoires instruction et donnée sont séparées.

5.1.7.1 Mémoire instruction (Instr Memory)

La mémoire instruction contient les instructions du programme à exécuter (l'application). La mémoire instruction est seulement lue lors de l'exécution d'un programme. L'adresse de l'instruction provient du compteur de programme, et l'instruction est envoyée vers le décodeur du processeur. La mémoire instruction peut être spécifiée en CHP fonctionnel non synthétisable comme suit:

```
[ Address_instr ? Ad;  
  instruction ! memoire_instr[Ad]  
]
```

Où `memoire_instr` est une variable du type tableau qui contient toutes les instructions à exécuter. A la réception d'une adresse depuis l'unité PC, la mémoire instruction envoie vers le décodeur une donnée de 32 bits correspondant à une instruction située à l'adresse reçue.

5.1.7.2 Mémoire donnée (Data Memory)

La mémoire donnée contient toutes les données qui sont manipulées lors de l'exécution d'un programme. Les données peuvent être lues depuis la mémoire ou écrites en mémoire. La mémoire donnée autorise aussi bien la lecture que l'écriture. Dans le cas d'une lecture, la mémoire reçoit de l'unité d'exécution l'adresse et la commande de lecture, et envoie à cette même unité d'exécution la donnée en mémoire correspondant à l'adresse. Dans le cas d'une écriture, l'unité d'exécution envoie la commande d'écriture, l'adresse, et la donnée à écrire dans la mémoire; celle-ci place cette donnée à l'adresse correspondante. Le CHP fonctionnel non synthétisable illustrant la mémoire donnée s'écrit:

```
[ RW ? rw, Address_instr ? Ad;  
  @[ rw='1' => donnee_r ! memoire_donnee[Ad]  
    rw='0' => donnee_w ? memoire_donnee[Ad]  
  ]  
]
```

La variable `memoire_donnee` est un tableau contenant des données de 32 bits. La mémoire donnée reçoit une commande de lecture/écriture (RW) qui dit si on accède en lecture ou en écriture. S'il s'agit d'une lecture, la donnée située à l'adresse reçue est envoyée sur le canal de lecture (`donnee_r`). Si c'est une écriture, une donnée est écrite (`donnee_w`) à une adresse de la mémoire.

5.2 Jeu d'instructions

Le mini processeur présente un jeu d'instructions réduit (RISC). Ce jeu d'instructions est un sous-ensemble de celui du processeur MIPS4Ksc. Le codage et l'exécution des instructions dans notre mini processeur sont fidèles du codage et de l'exécution des instructions du MIPS 4Ksc original. Le jeu d'instructions du mini processeur est présenté au Tableau 5-1.

Instruction	Description	fonction
ADDIU	Unsigned integer Add immediat	$Rt = Rs +_U \text{Immed}$
ADDU	Unsigned Integer Add	$Rd = Rs +_U Rt$
AND	Logical AND	$Rd = Rs \& Rt$
ANDI	Logical AND Immediate	$Rt = Rs \& (0_{16} \parallel \text{Immed})$
BEQ	Branch On Equal	if $Rs == Rt$ $PC += (\text{int})\text{offset}$
BNE	Branch on Not Equal	if $Rs != Rt$ $PC += (\text{int})\text{offset}$
J	Unconditional Jump	$PC =$
JAL	Jump and Link	$GPR[31] = PC + 1$ $PC = PC + \text{offset}$
JR	Jump Register	$PC = Rs$
LUI	Load Upper Immediate	$Rt = \text{immediate} \ll 16$
LW	Load Word	$Rt = \text{Mem}[Rs + \text{offset}]$
MFHI	Move From HI	$Rd = HI$
MFLO	Move From LO	$Rd = LO$
MTHI	Move To HI	$HI = Rs$
MTLO	Move To LO	$LO = Rs$
MUL	Multiply with register write	$HI \mid LO = \text{Unpredictable}$ $Rd = LO$
MULTU	Unsigned Multiply	$HI \mid LO = (\text{uns})Rs * (\text{uns})Rd$
NOP	No Operation	
NOR	Logical NOR	$Rd = \sim(Rs \mid Rt)$
OR	Logical OR	$Rd = Rs \mid Rt$
ORI	Logical OR Immediate	$Rt = Rs \mid \text{Immed}$
SW	Store Word	$\text{Mem}[Rs + \text{offset}] = Rt$
XOR	Exclusive OR	$Rd = Rs \wedge Rt$
XORI	Exclusive OR Immediate	$Rt = Rs \wedge (\text{uns})\text{Immed}$

Tableau 5-1 Jeu d'instructions du mini processeur

5.3 Etapes d'optimisation du mini processeur

Nous proposons dans ce paragraphe quatre versions du mini processeur notées V0, V1, V2 et V3 correspondant à des étapes successives d'optimisations possibles. On veut ainsi montrer l'évolution de la consommation d'énergie en fonction des optimisations apportées.

5.3.1 Version V0 : Forme primaire du mini processeur

La forme primaire du mini processeur correspond à la spécification CHP initiale telle qu'elle est normalement synthétisée par l'outil TAST [TAST-02]. Les données sont implémentées en dual rail (aussi bien les canaux de données que de contrôles) et les structures de choix sont toutes régulières (aucun déséquilibre).

5.3.2 Version V1 : Mini processeur implémenté en codage 1 parmi 4

La première optimisation importante que nous apportons au mini processeur est le passage d'un codage MR[2] à un codage MR[4] pour le chemin de données. Cependant, le décodeur reçoit une instruction qui est codée en MR[2] pour une question de facilité de décodage des instructions. Les commandes envoyées aux différents blocs par le décodeur sont elles aussi codées en MR[2]. Les données immédiates envoyées au PC et à l'interface d'exécution sont converties en MR[4] à l'aide d'un convertisseur MR[2] vers MR[4] (Figure 5.18).

5.3.3 Version V2 : Optimisation en fonction des probabilités d'exécution

La seconde optimisation que nous proposons résulte des probabilités obtenues par l'estimateur d'activité. Les probabilités qui nous intéressent particulièrement sont celles des gardes de chaque composant. L'application du mini processeur que nous considérons est un filtrage à réponse impulsionnelle finie (RIF). Le programme assembleur du filtre RIF est le suivant :

```
-- Reset Registers
LW R1,$1
LW R2,$2
LW R3,$3
LW R4,$4
LW R5,$5
LW R6,$6
LW R7,$7
LW R8,$8
LW R9,$9
LW R10,$A
LW R11,$B
LW R12,$C
LW R13,$D
LW R14,$E
LW R15,$F
LW R16,$10
LW R17,$11
LW R18,$12
LW R19,$13
LW R20,$14
LW R21,$15
LW R22,$16
LW R23,$17
LW R24,$18
LW R25,$19
LW R26,$1A
LW R27,$1B
LW R28,$1C
LW R29,$1D
LW R30,$1E
LW R31,$1F

-- Load Coeff
LW R1,$COEF4
LW R2,$COEF3
LW R3,$COEF2
LW R4,$COEF1

-- Load inputs
LW R5,$X0
LW R6,$X1
LW R7,$X2
LW R8,$X3

-- Compute
MUL R30,R5,R1
ADD R31,R30,R31
MUL R30,R6,R2
ADD R31,R30,R31
MUL R30,R7,R3
ADD R31,R30,R31
MUL R30,R8,R4
ADD R31,R30,R31

-- Jump address 0
J $0
```

Dans un premier temps, tous les registres du processeur sont mis à zéro en chargeant des valeurs nulles depuis la mémoire donnée (instruction LW).

Ensuite, les coefficients du filtre sont chargés depuis la mémoire donnée (aux adresses COEF1, COEF2, COEF3 et COEF4) dans certains registres du banc de registres (registres R1, R2, R3, R4).

On charge dans d'autres registres (R5, R6, R7, R8) les valeurs des entrées (depuis les adresses X0, X1, X2, X3 de la mémoire donnée).

La dernière étape du filtrage consiste à calculer le produit de convolution entre les valeurs des coefficients et les valeurs des entrées. Ce produit de convolution est réalisé dans la partie "compute" du code assembleur, il est obtenu en faisant des multiplications (instruction MUL) et des accumulations (instruction ADD) entre les coefficients et les entrées.

La dernière instruction du code assembleur est un saut (Jump) permettant de revenir à l'adresse du début du programme pour l'exécuter à nouveau.

Dans la suite de ce paragraphe, nous allons observer les probabilités obtenues par l'estimateur d'activité pour chaque composant du cœur du processeur. Les résultats permettent de déséquilibrer les structures de choix contenues dans les composants afin de privilégier les cas les plus probables.

- Probabilités du composant PC

Le composant PC se présente comme une structure de 5 choix déterministes. L'outil d'estimation de l'activité affecte par défaut des labels aux 5 gardes de la structure de choix (ces labels sont G1, G2, G3, G4, G5). Les résultats des probabilités obtenues sont présentés à la Figure 5.2.

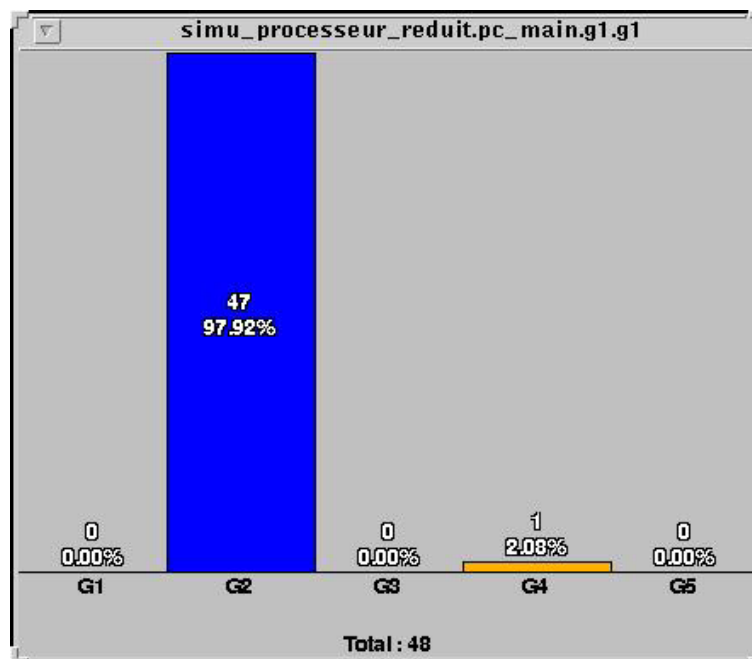


Figure 5.2 Probabilités du composant PC

On note que la garde G2 est la plus exécutée (près de 98% de l'exécution totale du composant). Cette garde exécute l'incrémentation du PC, elle est sollicitée par la plupart des instructions exécutées (LW, MUL et ADD). La garde G4 est exécutée une seule fois, il s'agit de la garde permettant le branchement à l'adresse 0 du programme (le dernier Jump).

Une implémentation judicieuse de ce composant est de privilégier la garde G2. Au lieu d'une implémentation régulière de la structure de choix, il est préférable de déséquilibrer celle-ci comme le montre le code CHP suivant :

```
@[ G2 => I2
  G1 or G3...=> @[G1 => I1
                  G3 => I3
                  ...
                ]
]
```

On note que certaines gardes ne sont jamais utilisées (G1, G3, G5); une optimisation optimale serait de supprimer ces gardes de la structure de choix. Le but de l'application est d'illustrer le déséquilibre des gardes et les bénéfices que cela apporte à la consommation d'énergie tout en conservant la complexité d'un microprocesseur. Nous ne supprimons donc pas ces gardes. Le même raisonnement s'applique aux autres composants du cœur du processeur.

- Probabilités du composant décodeur

Le décodage des instructions se fait dans le décodeur. Les probabilités des gardes exécutées dans le décodeur sont présentées à la Figure 5.3.

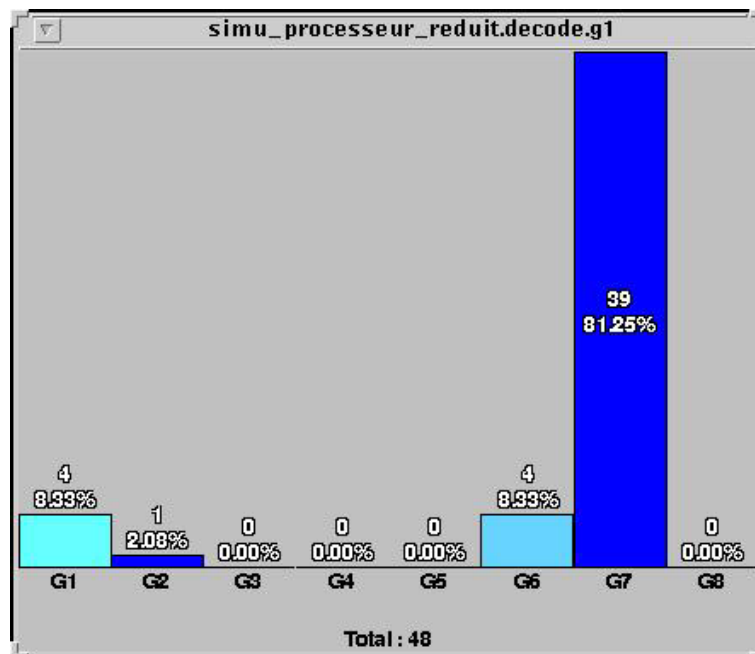


Figure 5.3 Probabilités du décodeur

La garde G1 correspond au décodage de l'instruction d'addition (ADD), la garde G2 au décodage du saut (J), G6 à celui de l'instruction de multiplication (MUL) et enfin la garde G7 correspond au décodage de l'instruction de chargement (LW). Les résultats obtenus sont très cohérents avec les instructions du code assembleur, à savoir qu'il y a 4 ADD (G1), 1 J (G2), 4 MUL (G6) et 39 LW (G7). En connaissance de ces informations, on peut optimiser le

décodeur en privilégiant le décodage de ADD, de MUL et de LW comme dans le programme CHP suivant:

```
@[ G1 => I1
   G6 => I6
   G7 => I7
   G2 or G3 or .. => @[ G2 => I2
                       G3 => I3
                       ...
                       ]
]
```

NB: On pourrait optimiser davantage cette structure de choix en ayant une structure plus hiérarchique:

```
@[ G7 => I7
   G1 or G2 or .. => @[ G1 => I1
                       G6 => I6
                       G2 or G3 or ... @[ G2 => I2
                                           G3 => I3
                                           ...
                                           ]
]
```

Ainsi, on privilégie davantage l'exécution de G7 qui est la garde la plus exécutée (avec 81,25%). Cependant, pour ne pas rendre trop compliquée la structure de choix, nous ne procédons qu'à un déséquilibre d'un seul niveau.

- Probabilités de l'interface d'exécution

L'interface d'exécution reçoit les opérandes et les oriente vers les bonnes unités de calcul. Les probabilités d'exécutions des gardes de l'interface sont présentées à la Figure 5.4.

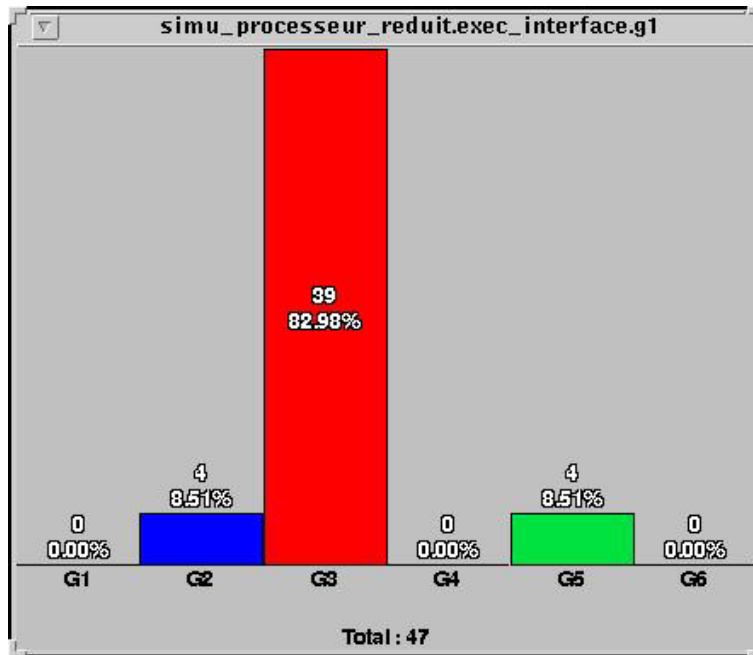


Figure 5.4 Probabilités de l'interface d'exécution

La garde G2 reçoit les opérandes depuis le banc de registres et les envoie à l'unité d'exécution. La garde G3 reçoit un bus depuis le banc de registres. La donnée contenue dans le bus est ajoutée à un offset (correspondant à une valeur immédiate contenue dans l'instruction), le résultat est envoyé à l'unité d'exécution. La garde G5 permet la réception de deux opérandes depuis le banc de registres et l'émission de celles-ci vers la MDU.

Il est intéressant d'implémenter ce bloc en privilégiant les gardes G2, G3 et G5 comme suit:

```

@[ G2 => I2
  G3 => I3
  G5 => I5
  G1 or G4 or ... => @[ G1 => I1
                    G4 => I4
                    ...
                    ]
]
    
```

Ici aussi, on aurait tout intérêt à déséquilibrer la structure de choix comme suit:

```

@[ G3 => I3
  G1 or G2 or ... => @[ G2 => I2
                    G5 => I5
                    G1 or G4 or ... @[ G1 => I1
                                       G4 => I4
                                       .....
                                       ]
]
    
```

- Probabilités de l'unité d'exécution

L'unité d'exécution contient les opérateurs arithmétiques et logiques, ainsi que les chargements et les branchements avec conditions. Il s'agit d'une structure de onze choix au total. Les probabilités d'exécution des commandes gardées de l'unité d'exécution sont ceux de la Figure 5.5.

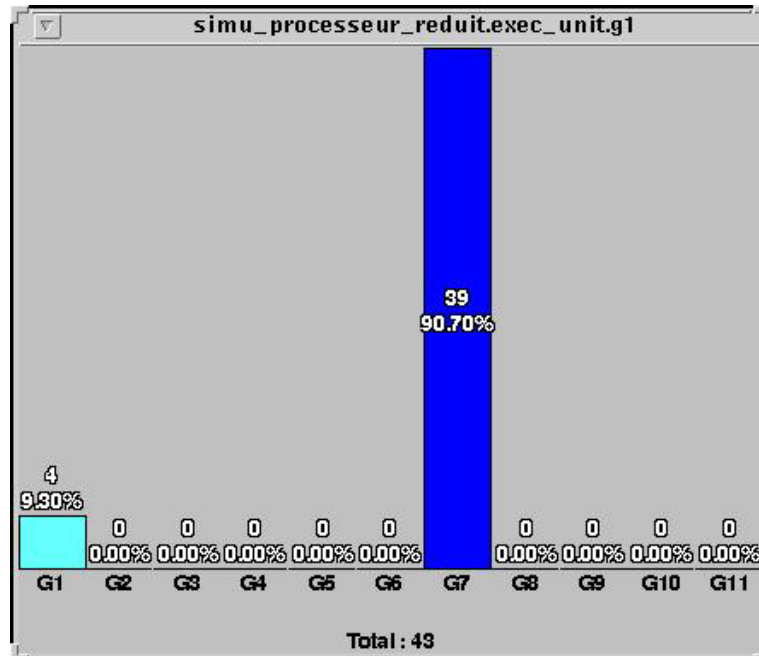


Figure 5.5 Probabilités de l'unité d'exécution

Les gardes qui présentent une activité sont les gardes G1 et G7. La garde G1 correspond à l'exécution de l'opérateur d'addition (ADD) et la garde G7 est l'opération de chargement (LW). On peut déséquilibrer cette structure comme suit:

```
@[ G1 => I1
  G7 => I7
  G2 or G3 or ... => @[ G2 => I2
                      G3 => I3
                      ...
                      ]
]
```

On peut faire la même remarque ici aussi sur une structure plus hiérarchique telle qu'on l'a montrée pour le décodeur et l'interface d'exécution.

Dans certains cas, il est possible d'optimiser davantage la structure. L'unité d'exécution présente une structure de choix imbriquée dans la garde G1. Il est possible de visualiser la sous structure de choix de la garde G1. Il s'agit d'une sous structure de six choix correspondant aux opérateurs ADD, AND, OR, XOR, NOR et JR (Jump Register). Les probabilités obtenues pour cette sous-structure de choix sont les suivantes (Figure 5.6).

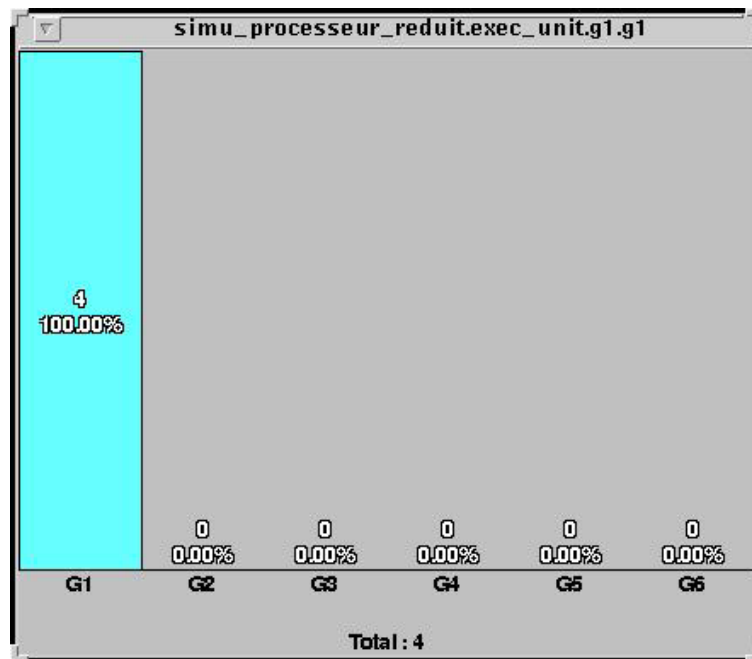


Figure 5.6 Probabilités d'une sous partie de l'unité d'exécution

La garde G1 correspond à l'opérateur ADD, ce dernier est utilisé à chaque fois (100% d'exécution de cette sous-structure). On peut donc déséquilibrer cette sous-structure de la façon suivante:

```

@[ G1 => I1
  G2 or G3 or ... => @[ G2 => I2
                      G3 => I3
                      ...
                      ]
]
    
```

- Probabilités de la MDU

Enfin on souhaite obtenir des informations sur l'exécution de la MDU. Les probabilités de la MDU sont illustrées à la Figure 5.7.

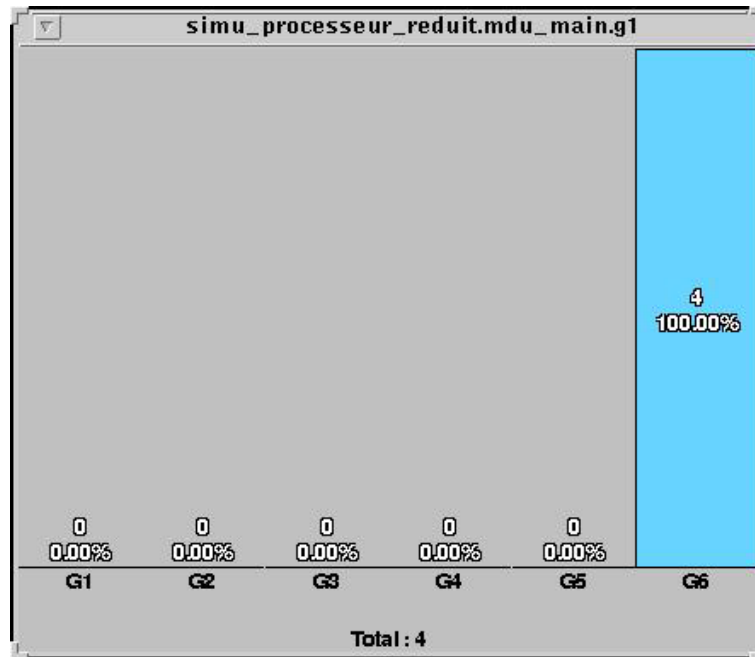


Figure 5.7 Probabilités de la MDU

Chaque garde de la structure de choix de la MDU correspond à un opérateur (de transfert ou de multiplication). L'opérateur qui est toujours utilisé dans la MDU est le multiplieur via l'instruction MUL (garde G6). Cette garde présente donc 100% d'exécution, on déséquilibre cette structure de choix comme fait précédemment:

```
@[ G6 => I6
  G1 or G2 or ... => @[ G1 => I1
                      G2 => I2
                      ...
                      ]
]
```

5.3.4 Version V3 : Optimisation du contrôle

La dernière optimisation que nous proposons est l'optimisation des canaux de contrôle. Les canaux sont découpés pour n'envoyer que les commandes nécessaires et suffisantes aux composants du cœur du processeur. De plus, les canaux sont codés de manière optimale. On devrait observer un bénéfice sur l'envoi de la commande (qui est de taille plus petite car la commande a été découpée et codée de manière optimale) et sur la réception de celle-ci (le codage optimal de la commande permet une évaluation plus simple de la garde).

Nous avons proposé quatre implémentations possibles du processeur correspondant à trois étapes d'optimisation. Il s'agit de trois étapes d'optimisation parmi les nombreuses que nous avons présentées au Chapitre 4. On pourrait optimiser davantage en réalisant une bufferisation adaptée entrée/sortie, une désynchronisation intra canal, un ajout de synchronisation, une optimisation micro architecturale, etc.

5.4 Résultats

Dans ce paragraphe, nous allons présenter l'évolution de la consommation d'énergie du Mini Processeur en fonction des étapes d'optimisation que nous avons présentées précédemment. Nous le faisons tout d'abord pour le cœur du mini processeur global, puis nous détaillerons l'évolution de la consommation d'énergie des composants du cœur du mini processeur.

5.4.1 Consommation de l'ensemble du cœur du mini processeur

Avant de commencer l'analyse de la consommation d'énergie, nous allons donner des informations concernant la taille du cœur de processeur, plus précisément la taille des composants du mini processeur (Figure 5.8). La taille est exprimée en nombre de portes NAND à 2 entrées (unité du rapport fourni par Synopsys), de plus on considère la version V0 qui est la version primaire. Le processeur a été mappé sur une librairie de cellules standard de technologie HCMOS9 0.12 μ m de ST Microelectronics.

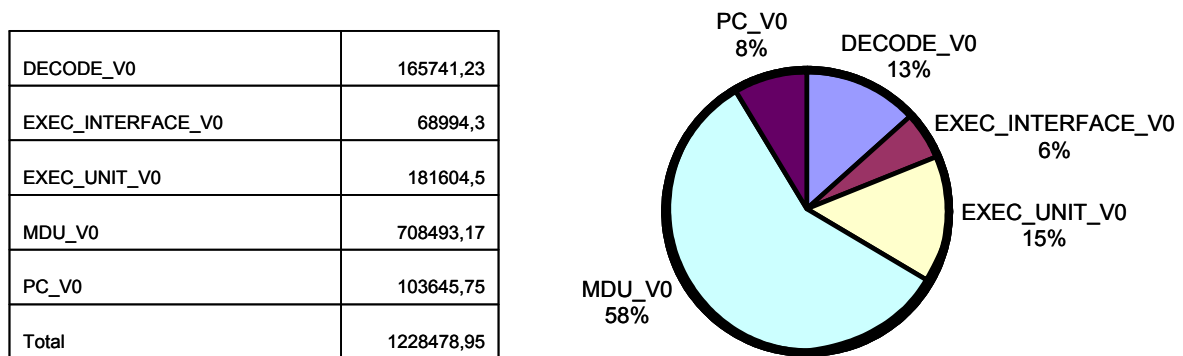


Figure 5.8 Taille des composants du Mini Processeur

Ces chiffres montrent que le processeur est de taille importante. Cette taille importante est liée au fait qu'on ne cherche pas à optimiser la surface. La spécification CHP est la plus simple possible afin de pouvoir valider notre méthodologie d'estimation et les techniques d'optimisation de la consommation d'énergie. On ne fait pas de partage de ressources (il existe dans le processeur 2 multiplieurs et plusieurs additionneurs). De plus, on utilise des cellules synchrones de librairie standard HCMOS9. En utilisant la librairie TAL [MAUR-03] qui est une librairie de cellules asynchrones, on pourrait réduire davantage la consommation d'énergie.

On voit que la MDU à la plus grande taille (58% du cœur du processeur) en raison de la taille du multiplieur. Il y a ensuite l'unité d'exécution (qui intègre les opérateurs arithmétiques et logiques) et le décodeur qui ont des tailles à peu près identiques (15% et 13%). Enfin le PC et l'interface d'exécution qui sont proches en surface (8% et 6%).

En fonction des optimisations réalisées, la taille évolue. La Figure 5.9 décrit l'évolution de la taille du mini processeur pour les versions V0, V1, V2, V3.

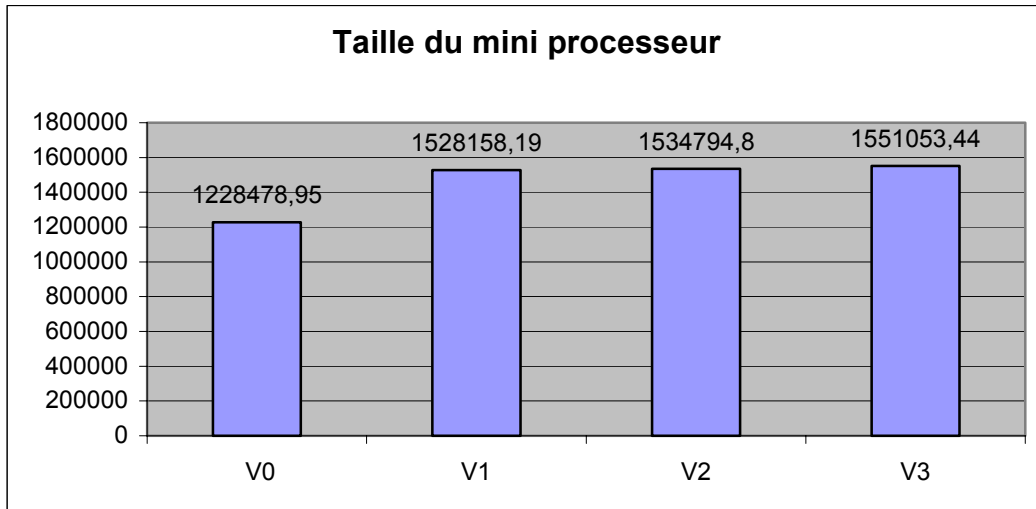


Figure 5.9 Taille du Mini Processeur (V0, V1, V2 et V3)

On remarque que les optimisations proposées pour réduire la consommation d'énergie ont tendance à pénaliser la taille du mini processeur. Le Tableau 5-2 nous donne des informations sur le taux d'augmentation de la surface en fonction des étapes d'optimisations.

V0 vs V1	V1 vs V2	V2 vs V3	V0 vs V3
+24,4 %	+0,4 %	+1,05 %	+26,2%

Tableau 5-2 Evolution de la Taille du Mini Processeur

L'augmentation la plus nette, est le passage de la version V0 à la version V1, soit le passage d'un codage MR[2] à un codage MR[4] où on enregistre une augmentation de 24,4% de surface. Les autres optimisations (V2 et V3) pénalisent très peu la surface. Au final, la version V3 présente une augmentation de surface de près de 26% par rapport à l'implémentation originale V0.

Nous allons maintenant nous pencher sur la consommation d'énergie du mini processeur, et l'évolution de celle-ci en fonction des étapes d'optimisations présentées précédemment.

Comparons la méthodologie d'estimation de la consommation d'énergie proposée au Chapitre 2 et les estimations au niveau porte faites par l'outil commercial Nanosim de Synopsys. Nanosim est reconnu par les concepteurs de circuits intégrés comme un estimateur précis (~10% de précision) et rapide (comparé à une simulation électrique du type AnalogArtist). Les résultats de la simulation à l'aide de Nanosim sont exprimés en picoJoule (pJ) correspondant à l'énergie consommée par les différents composants du cœur du mini processeur lors d'une simulation donnée (notamment l'ensemble du code assembleur proposé au paragraphe 5.3.3 qui correspond à l'application d'un filtre à réponse impulsionnelle finie RIF). Cette estimation est obtenue avant placement routage du circuit. Pour la même simulation, nous appliquons notre technique d'estimation de la consommation d'énergie qui s'exprime en nombre de transitions de porte. Le Tableau 5-3 présente les résultats obtenus pour la version primaire (V0) des cinq composants du cœur du mini processeur.

Composants	Nanosim Energy (pJ)	Méthodologie (nombre de transitions)
DECODE	2856	7,69E+04
EXEC_INTERFACE	1570,92	5,89E+04
EXEC_UNIT	1089,6	4,98E+04
MDU	888,01	4,82E+04
PC	1380	5,95E+04
Total	7784,53	2,93E+05

Tableau 5-3 Consommation obtenue par Nanosim et par la Méthodologie

Nous établissons à partir de cette table, la répartition en pourcentage de la consommation d'énergie obtenue par les deux méthodes (Figure 5.10).

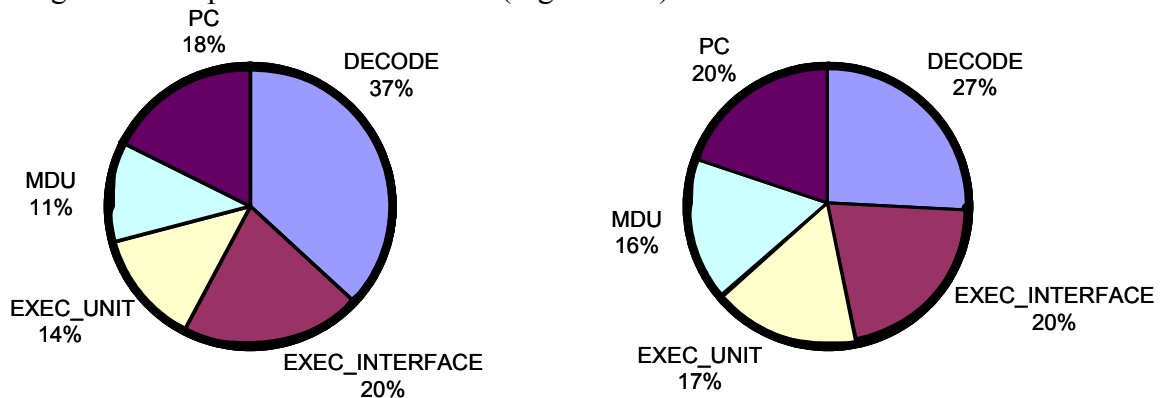


Figure 5.10 Répartition de la consommation obtenue par Nanosim (gauche) et par notre estimateur (droite)

On constate que la répartition de la consommation d'énergie est assez bien respectée par notre méthodologie. Elle est assez proche de la répartition de la consommation d'énergie obtenue par Nanosim. Les différences qu'on observe sont dues principalement aux composants contenant des opérateurs (multiplieur, et additionneur essentiellement). On voit très clairement que la consommation de la MDU qui contient un multiplieur est supérieure à la consommation d'énergie obtenue par Nanosim. Ceci est lié au fait qu'on considère une estimation moyenne, car l'estimation réelle du multiplieur dépend des données qui sont manipulées. Les composants PC et l'unité d'exécution rencontrent aussi une surévaluation de la consommation d'énergie, car ces derniers intègrent des additionneurs où l'estimation que nous établissons est moyenne. De plus, l'estimation des portes avec un nombre d'entrées important est majorée lors de la décomposition de celles-ci en arbre de portes à 2 entrées. Tout cela participe aux différences que nous observons entre l'estimation obtenue par notre méthodologie et par l'outil Nanosim. Il en reste que les proportions sont assez bien respectées.

Nous regardons l'évolution globale de la consommation d'énergie du cœur du mini processeur à partir de notre méthodologie d'estimation de la consommation d'énergie (Figure 5.11).

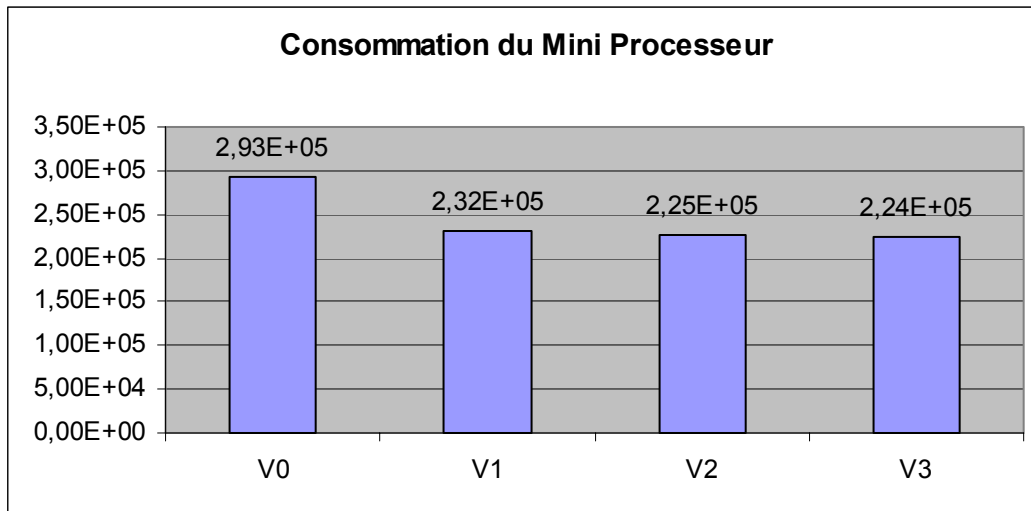


Figure 5.11 Consommation du Mini Processeur

La tendance générale de l'histogramme s'analyse comme une forte diminution de la consommation d'énergie de la version V0 à la version V1 (réduction de 21%), puis de petites diminutions (-2,7% puis -0,7%) comme nous le précise le Tableau 5-4. La consommation du cœur du mini processeur a été réduite de 23,7% après les trois étapes d'optimisation.

V0 vs V1	V1 vs V2	V2 vs V3	V0 vs V3
-21 %	-2,7 %	-0,7 %	-23,7%

Tableau 5-4 Evolution de la consommation du Mini Processeur

Pour avoir un aperçu plus précis de l'évolution de la consommation d'énergie en fonction des étapes d'optimisation, il est intéressant d'observer comment la consommation évolue dans les composants du cœur du mini processeur (sous paragraphes à venir).

Un critère important qu'il est intéressant de remarquer est la vitesse du circuit. Les étapes d'optimisation de la consommation d'énergie peuvent avoir un impact sur la vitesse du circuit. Si cet impact est préjudiciable, à savoir si le temps d'exécution de la version optimisée est plus important que celui de la version non optimisée, il est parfois préférable de ne pas procéder à l'optimisation. En effet, en faisant du DVS [ESSA-02] sur la version non optimisée (pour avoir un temps de simulation équivalent à la version optimisée), on peut obtenir une réduction de la consommation d'énergie qui est plus importante que la réduction de la consommation d'énergie apportée par la version optimisée.

Nous allons essayer de comparer les temps de calcul que nous avons obtenus par Nanosim pour les quatre versions de processeurs (Tableau 5-5).

V0	V1	V2	V3
1,353 μ s	1,324 μ s	1,3182 μ s	1,3182 μ s

Tableau 5-5 Temps de simulation des quatre versions de processeur

Le Tableau 5-6 montre l'évolution du temps d'exécution du programme en pourcentage.

V0 vs V1	V1 vs V2	V2 vs V3	V0 vs V3
-2,15%	-0,44%	0%	-2,59%

Tableau 5-6 Evolution du temps d'exécution en pourcentage

On s'aperçoit que le temps de simulation diminue avec les étapes d'optimisation (sauf pour la version V3 où on observe un temps de simulation à peu près équivalent au temps de simulation de la version V2). Même si le gain en temps n'est pas considérable, il est intéressant de constater qu'il n'y a pas d'augmentation.

La diminution de la version V0 à la version V1 (-2,15%) s'explique par le fait qu'on manipule des données de taille deux fois moindre que la version V0. Grâce au déséquilibre des structures de choix dans la version V2 on observe une diminution du temps d'exécution par rapport à la version V1 (-0,44%). Cela s'explique par le fait que les gardes les plus probables (qui sont les plus exécutées) traversent des portes plus petites. Enfin pour la version V3, on observe un temps d'exécution à peu près équivalent à la version V2 car il s'agit d'une petite optimisation permettant la suppression de petites portes. Au total on note une diminution du temps d'exécution de 2,59%. La diminution n'est certes pas très importante, mais le plus important est que les étapes d'optimisation ne font pas augmenter le temps d'exécution.

5.4.2 PC

Nous analysons l'évolution de la consommation d'énergie du compteur de programme PC en fonction des étapes d'optimisation. La Figure 5.12 montre cette évolution.

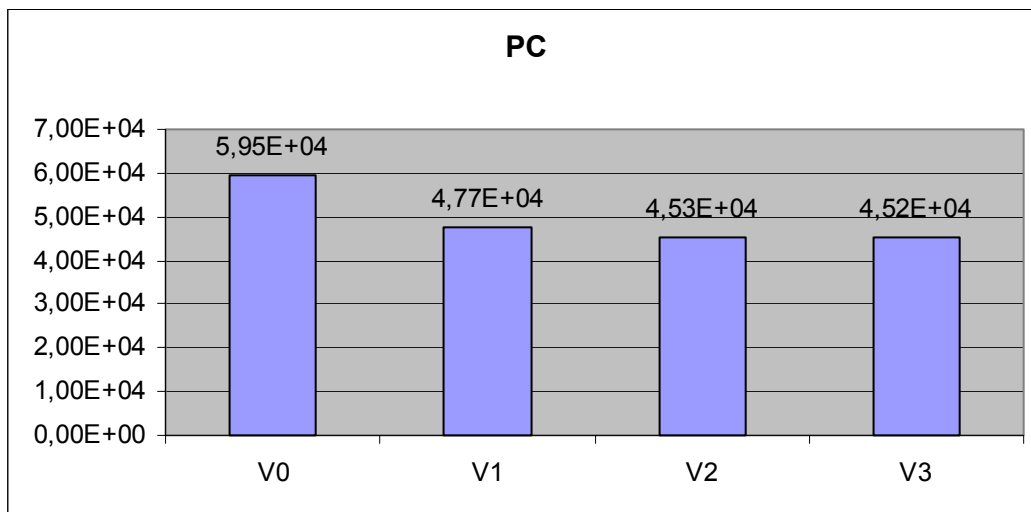


Figure 5.12 Consommation du composant PC

En pourcentage de réduction, cette figure se traduit par le Tableau 5-7.

V0 vs V1	V1 vs V2	V2 vs V3	V0 vs V3
-19,8 %	-5 %	-0,2 %	-24 %

Tableau 5-7 Evolution de la consommation du PC

On constate une forte diminution de la consommation d'énergie lorsqu'on passe d'un codage MR[2] vers un codage MR[4] (de la version V0 à la version V1). Le déséquilibre de la structure de choix principale (de cinq commandes gardées à deux commandes gardées) permet une réduction non négligeable de 5%. Enfin l'optimisation V3 permet une réduction minimale de 0,2% grâce au passage d'une commande codée en MR[2][3] (permettant de coder la structure de cinq choix) à une commande codée en MR[2] (permettant de coder la structure déséquilibrée de deux choix).

Les trois étapes d'optimisation permettent au PC de réduire la consommation d'énergie de 24%.

5.4.3 Décodeur

L'estimation de la consommation d'énergie obtenue à l'aide de notre méthodologie est celle de la Figure 5.13.

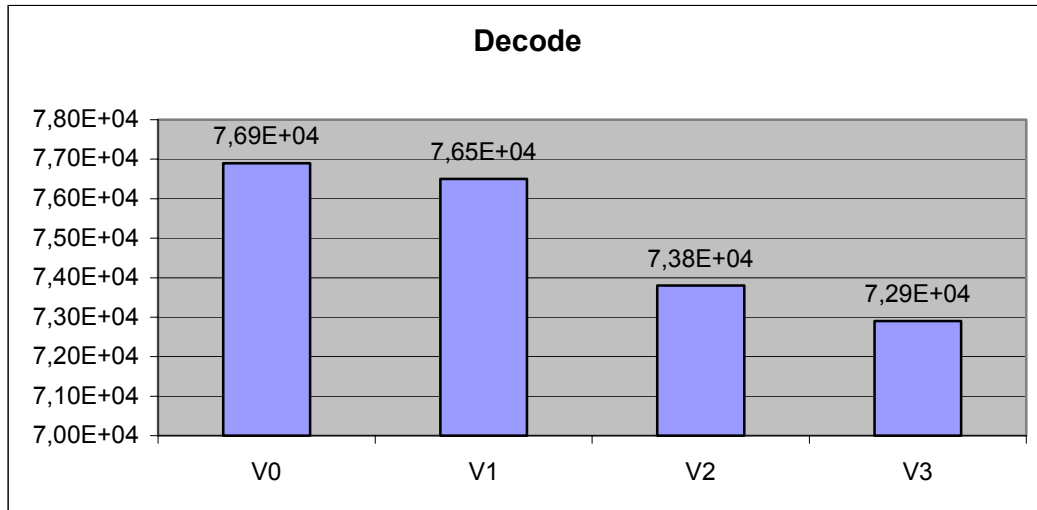


Figure 5.13 Consommation du Décodeur

En terme de pourcentage de réduction de la consommation d'énergie, les résultats sont présentés au Tableau 5-8.

V0 vs V1	V1 vs V2	V2 vs V3	V0 vs V3
-0,5 %	-3,5 %	-1,2 %	-5,2 %

Tableau 5-8 Evolution de la consommation du Décodeur

Contrairement aux autres composants, la version V1 du décodeur ne présente pas de forte réduction de la consommation d'énergie. Cela s'explique par le fait que le décodeur conserve une implémentation MR[2]. On a choisi de garder le codage MR[2] pour des raisons de facilité de décodage des instructions. Le passage en MR[4] rendait le décodage plus délicat. Cependant, les valeurs immédiates que le décodeur envoie au PC et à l'interface d'exécution sont converties en codage MR[4] au moyen d'un convertisseur MR[2]vers MR[4] (Figure 5.18). L'envoi de ces valeurs en codage MR[4] explique la très légère réduction de la consommation d'énergie (-0,5%). La version V2 permet une réduction de 3,5% par rapport à la version V1 grâce au déséquilibre de la structure de choix. Enfin, la version V3 permet une réduction supplémentaire de 1,2%. Le décodeur permet d'envoyer les commandes aux blocs du processeur; en découpant les commandes judicieusement on réduit la consommation d'énergie du décodeur car les commandes envoyées sont de plus petites tailles. On aboutit à une réduction totale du décodeur de 5,2 %.

5.4.4 Interface d'exécution

L'évolution de la consommation d'énergie de l'interface d'exécution est présentée à la Figure 5.14.

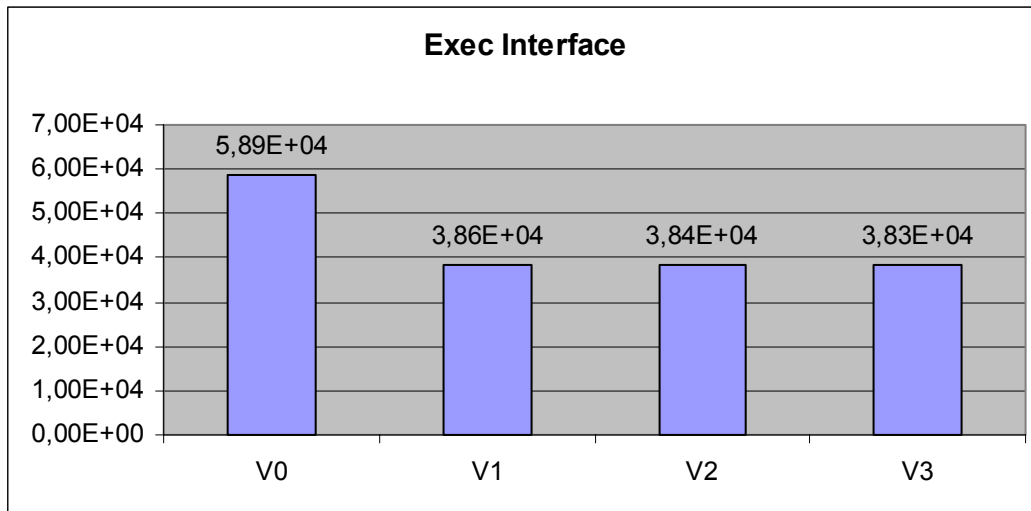


Figure 5.14 Consommation de l'interface d'exécution

L'évolution en terme de pourcentage est présentée au Tableau 5-9.

V0 vs V1	V1 vs V2	V2 vs V3	V0 vs V3
-34,4 %	-0,5 %	-0,3 %	-35 %

Tableau 5-9 Evolution de la consommation de l'Interface d'Exécution

On constate une importante réduction de la consommation d'énergie de la version V0 à la version V1 (-34,4%). Le passage à un codage MR[4] est très intéressant sachant que l'interface d'exécution permet d'aiguiller les données provenant du banc de registres et du décodeur (valeurs immédiates) vers l'unité d'exécution et la MDU. Il s'agit d'un composant qui permet le transit des données du cœur du processeur. Passer à un codage MR[4] permet une réduction considérable de l'énergie dynamique.

Les optimisations V2 et V3 permettent une très faible réduction de la consommation d'énergie. L'interface d'exécution est une petite structure de choix où le déséquilibre avantage très peu la consommation d'énergie (3 gardes sur 5 sont utilisées). De plus l'optimisation du canal de contrôle (version V3) permet le passage d'un contrôle codé en MR[2][3] à un contrôle codé en MR[4]. La réduction est donc très petite (-0,3%).

Les trois étapes d'optimisation permettent une réduction de la consommation d'énergie de 35%.

5.4.5 Unité d'exécution

La réduction de la consommation d'énergie de l'unité d'exécution est décrite à la Figure 5.15.

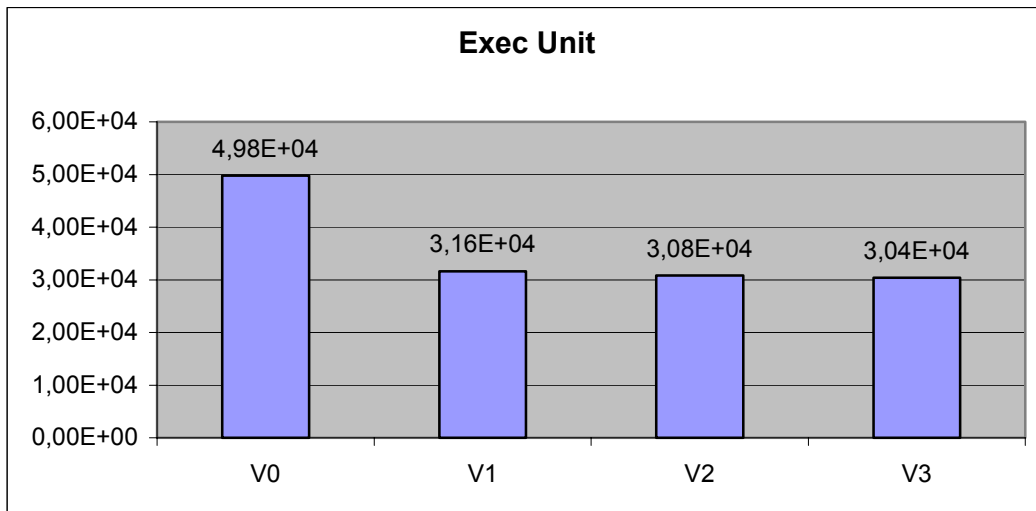


Figure 5.15 Consommation de l'unité d'exécution

Le Tableau 5-10 montre l'évolution en pourcentage de la consommation d'énergie en fonction des trois étapes d'optimisation.

V0 vs V1	V1 vs V2	V2 vs V3	V0 vs V3
-36,5 %	-2,5 %	-1,2 %	-38,9 %

Tableau 5-10 Evolution de la consommation de l'Unité d'Exécution

On note une très nette réduction de la consommation d'énergie en passant d'un codage MR[2] à un codage MR[4], ceci est en parti lié à la réduction de la consommation d'énergie de l'additionneur implémenté en MR[4]. Le déséquilibre de la structure de choix de l'unité d'exécution permettant de privilégier l'exécution de l'addition et du Load réduit la consommation d'énergie de 2,5%. L'optimisation du contrôle de la structure de choix apporte une réduction supplémentaire de 1,2%. On obtient au final un circuit réduit de 38,9% par rapport à la version originale V0.

5.4.6 La MDU

Enfin l'évolution de la consommation d'énergie de la MDU en fonction des étapes d'optimisation est présentée à la Figure 5.16.

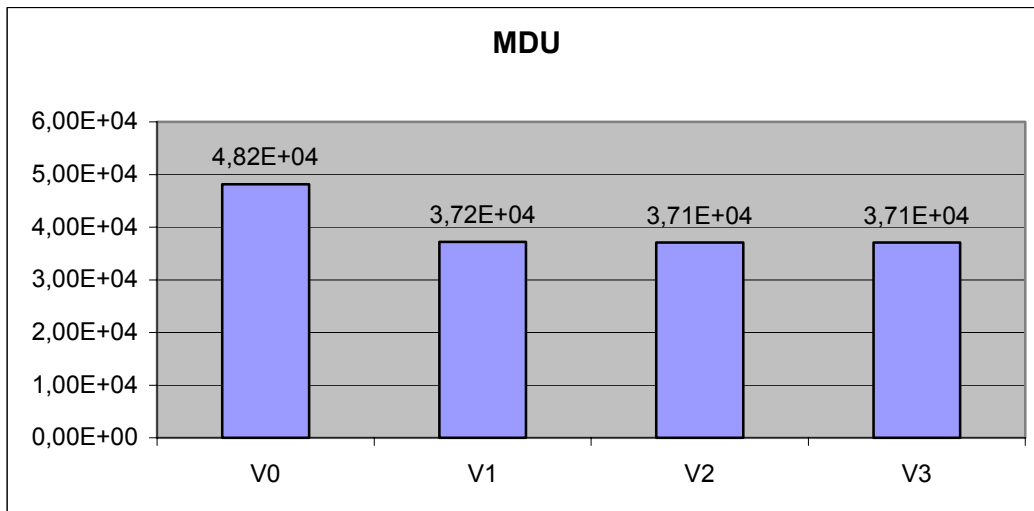


Figure 5.16 Consommation de la MDU

Le Tableau 5-11 représente la réduction de la consommation d'énergie en pourcentage.

V0 vs V1	V1 vs V2	V2 vs V3	V0 vs V3
-22,7 %	-0,27 %	-0,02%	-23 %

Tableau 5-11 Evolution de la consommation de la MDU

Le passage à un codage MR[4] permet ici aussi une réduction importante de la consommation d'énergie (22,7%). Cette réduction s'explique par l'implémentation MR[4] du multiplieur qui est moins coûteuse que l'implémentation MR[2]. Les optimisations V2 et V3 réduisent très peu la consommation d'énergie car la MDU n'est sollicitée que quatre fois pour réaliser quatre multiplications. On aboutit au final à une réduction de la consommation d'énergie de la MDU de 23%.

5.5 Implémentation finale du mini processeur

Le mini processeur est un processeur RISC de 32 bits mappé sur une technologie ST 0.12 μm de STMicroelectronics. Ce paragraphe présente l'implémentation architecturale finale du mini processeur.

5.5.1 Implémentation du cœur

Le cœur du mini processeur a été entièrement décrit à l'aide d'une spécification CHP synthétisable par l'outil de synthèse TAST. Le compteur de programme (PC), l'interface d'exécution (Exec interface), l'unité d'exécution (Exec unit) et l'unité de multiplication et division (MDU) manipulent des données codées en 1 parmi 4. Un buffer WCHB en implémentation 1 parmi 4 est présenté à la Figure 5.17.

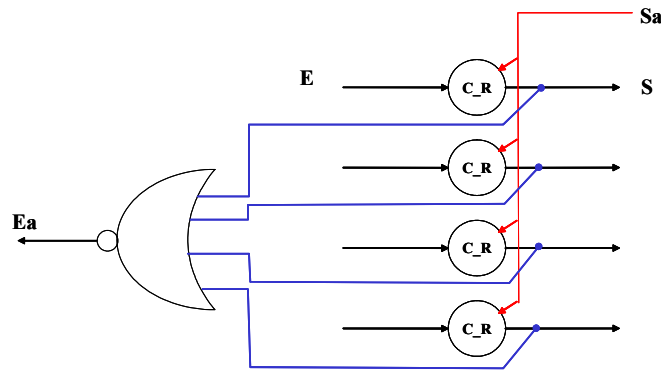


Figure 5.17 Buffer WCHB MR[4]

En revanche, le décodeur reçoit de la mémoire instruction une donnée codée en dual rail pour des raisons de facilités de décodage. Cependant, les commandes envoyées aux divers blocs du circuit par le décodeur sont réalisées à l'aide d'un codage adéquat. Lorsque l'instruction décodée utilise une valeur immédiate ou un offset (pour le load ou store), une conversion dual rail vers 1 parmi 4 est nécessaire. Un convertisseur MR[2] vers MR[4] est réalisé à l'aide du circuit de la Figure 5.18.

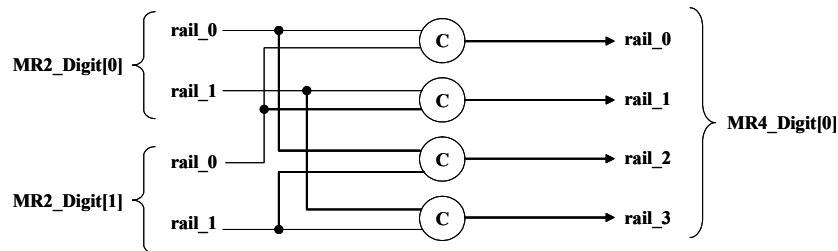


Figure 5.18 Convertisseur MR[2] -> MR[4]

Un convertisseur permettant la manipulation inverse, à savoir une conversion MR[4] vers MR[2] est réalisée à l'aide du circuit de la Figure 5.19.

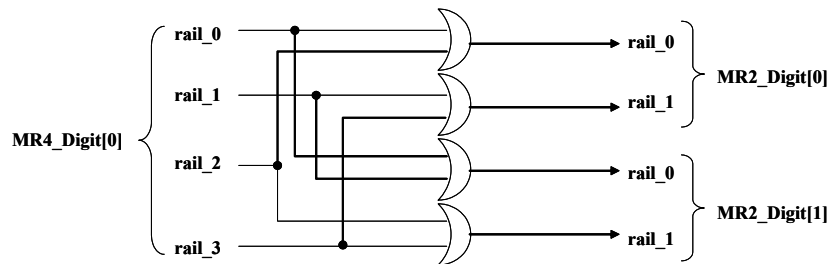


Figure 5.19 Convertisseur MR[4] -> MR[2]

Pour les blocs de calcul, les opérateurs 1 parmi 4 sont générés directement par l'outil de synthèse. Il s'agit d'opérateurs logiques en 1 parmi 4 (and, or, xor etc.) et arithmétiques (additionneur et multiplieur).

5.5.2 Implémentation du banc de registres

Le banc de 32 registres de 32 bits (codé en 1 parmi 4) a été synthétisé à la main. Le bloc essentiel du banc de registres est le point mémoire MR[4]. Il est basé sur le point mémoire décrit en dual rail dans les travaux de thèse de [VIVE-01] et adapté au codage 1 parmi 4 [PALF-03]. L'implémentation en MR[4] est présentée à la Figure 5.20.

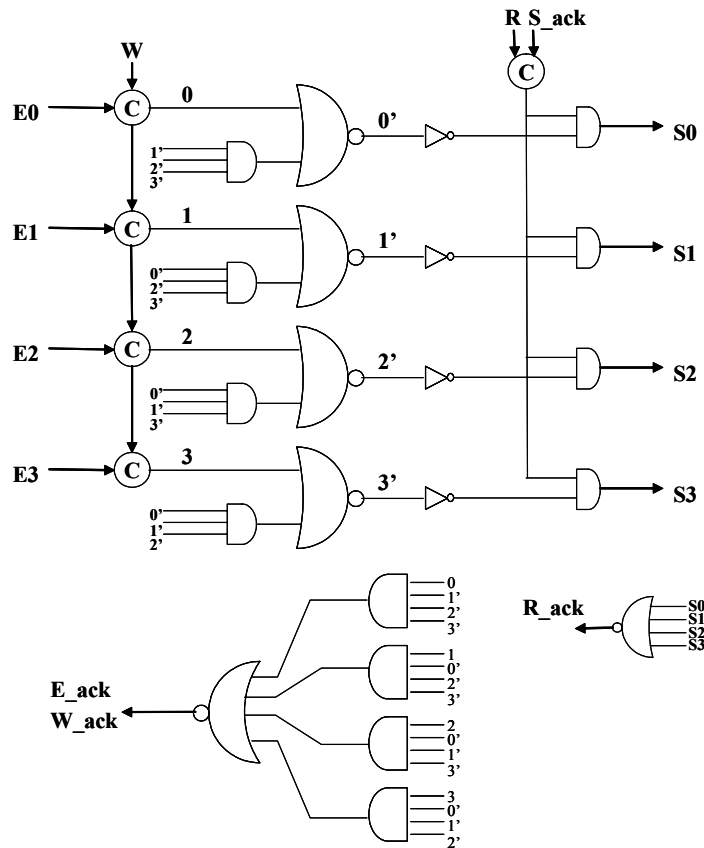


Figure 5.20 Registre MR[4]

La lecture d'un registre se fait à l'aide des portes AND en sortie lorsque la commande autorisant la lecture du registre est valide. Une fois que la donnée du registre est lue, le bloc qui consomme cette donnée retourne un acquittement permettant d'acquitter la commande de lecture.

L'écriture d'un registre se fait à l'aide des portes de muller en entrée. Lorsque la commande d'écriture et la donnée entrante sont valides, le point mémoire est modifié. Les rétroactions qui sont dans le point mémoire permettent de ne mettre à 1 qu'un seul fil. Les autres fils sont mis à 0 (principe du codage 1 parmi 4). Une fois que la donnée est écrite dans le point mémoire et qu'elle est stable, la commande et la donnée entrante sont acquittées.

5.5.3 Implémentation des mémoires

Les mémoires instruction et donnée qui sont utilisées dans le mini processeur sont des mémoires purement synchrones. Elles sont succinctement présentées dans les deux sous paragraphes suivants.

5.5.3.1 Mémoire instruction

La mémoire instruction permet d'aller lire en mémoire les instructions de 32 bits de l'application du mini processeur. On utilise une mémoire synchrone très simple qui reçoit une adresse et qui envoie la donnée correspondante à l'adresse reçue au front montant de l'horloge. Le schéma de la mémoire instruction est représenté à la Figure 5.21.

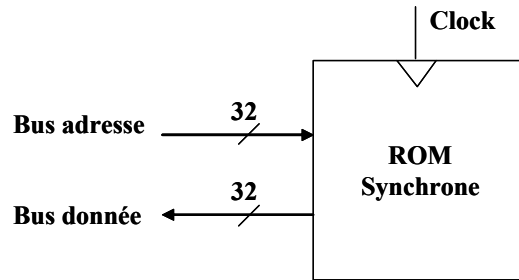


Figure 5.21 Mémoire ROM synchrone

5.5.3.2 Mémoire donnée

La mémoire donnée (mémoire RAM synchrone Figure 5.22) est un peu plus complexe que la mémoire instruction puisqu'elle autorise la lecture et l'écriture. Elle reçoit un signal *rw* pour distinguer la lecture de l'écriture, un bus d'adresse et un bus de donnée (donnée à écrire en mémoire). Elle envoie un bus de donnée (donnée à lire) vers le cœur du processeur. Les bus de donnée et d'adresse sont codés sur 32 bits.

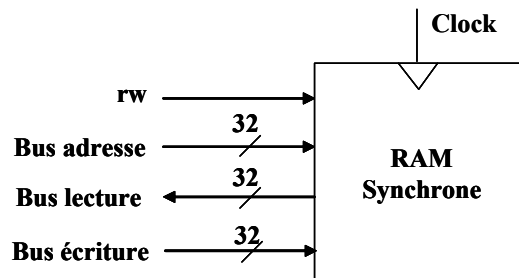


Figure 5.22 Mémoire RAM synchrone

5.5.4 Implémentation des interfaces

La connexion du cœur du mini processeur avec les éléments mémorisants (banc de registres, mémoire instruction et mémoire donnée) nécessite l'utilisation d'interfaces adaptées que nous présentons dans les sous paragraphes suivants.

5.5.4.1 Interface banc de registres

L'interface du banc de registres est un séquenceur, il assure l'accès séquentiel au banc de registres, c'est-à-dire qu'il garantit le non recouvrement des étapes de lecture et d'écriture. Cette interface permet d'envoyer la (ou les) commande(s) de lecture suivit de la commande d'écriture. L'interface est présenté à la Figure 5.23 par un rectangle dans lequel figure le symbole de séquentialité ";".

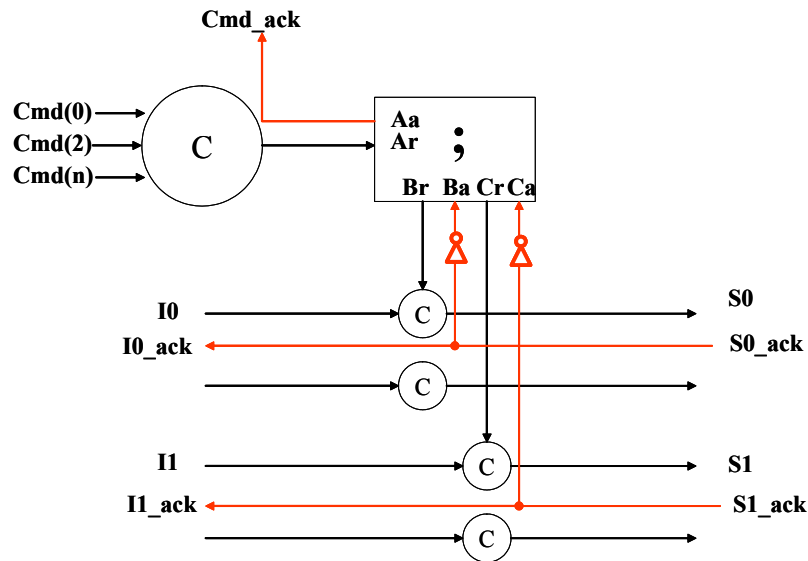


Figure 5.23 Interface registre

On observe dans ce composant 3 ports A, B, C. Chaque port est composé d'une requête et d'un acquittement. A la réception de la requête sur le port A, le handshake sur le port B est réalisé, une fois celui-ci terminé, l'ensemble du handshake sur le port C se fait, puis pour terminer la communication le port A est acquitté. On peut ainsi utiliser les requêtes Br et Cr pour commander respectivement les canaux de lecture (I0) et d'écriture (I1). Les acquittements Ba et Ca sont générés par le banc de registres une fois que les données ont été respectivement lues et écrites. Ce procédé permet d'envoyer séquentiellement les commandes de lecture et d'écriture au banc de registres.

5.5.4.2 Interface mémoire instruction

L'interface mémoire instruction permet de connecter le cœur du mini processeur avec la mémoire ROM synchrone. L'aspect extérieur de l'interface mémoire instruction est représenté à la Figure 5.24.

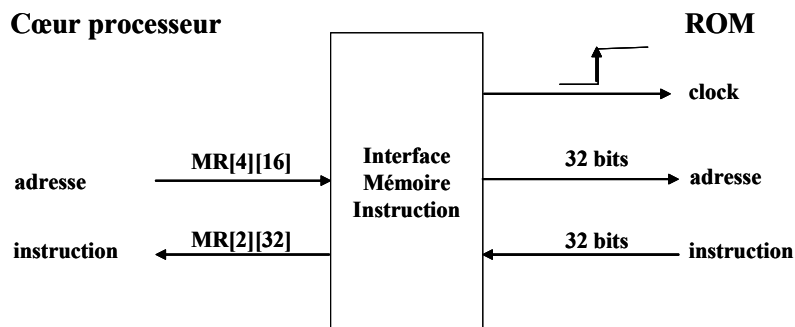


Figure 5.24 Interface mémoire instruction

L'interface mémoire instruction reçoit du cœur du mini processeur un canal d'adresse codé en MR[4][16]. A partir de ce canal, elle génère le signal d'horloge de la mémoire (clock), un signal interne d'autorisation d'envoi de l'instruction au mini processeur (READ) et l'acquiescement du canal d'adresse. La génération de ces signaux est réalisée à l'aide du circuit de la Figure 5.25.

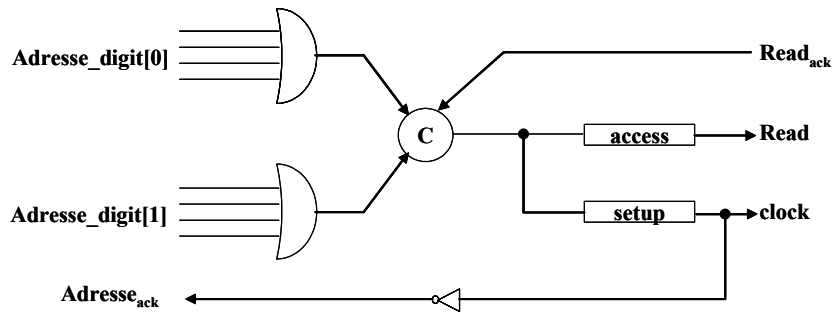


Figure 5.25 Interface mémoire instruction d'entrée

Des délais doivent être introduits pour satisfaire les conditions d'accès à la mémoire (access et setup). Ces délais correspondent aux caractéristiques de la mémoire ROM synchrone utilisée. De plus, des convertisseurs sont nécessaires pour passer d'un canal MR[4][16] vers un bus de 32 bits binaire (pour l'accès mémoire). La conversion d'un digit MR[4] en 2 digits binaires est réalisée par le circuit de la Figure 5.26.

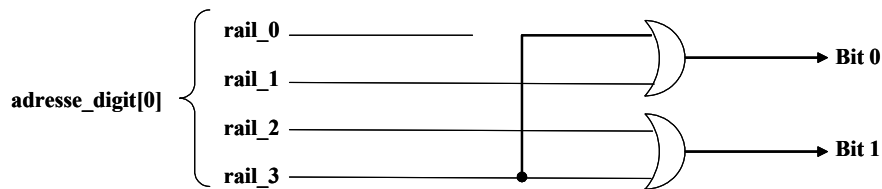


Figure 5.26 Convertisseur MR[4] -> Bit

L'instruction qui est lue en mémoire est codée sous forme d'un signal de 32 bits. Il faut donc implémenter un circuit permettant de convertir ce signal en un canal MR[2][32]. Le circuit de la Figure 5.27 présente un convertisseur d'un bit vers un digit MR[2]. Ce convertisseur utilise le signal intermédiaire READ produit par le circuit de la Figure 5.25.

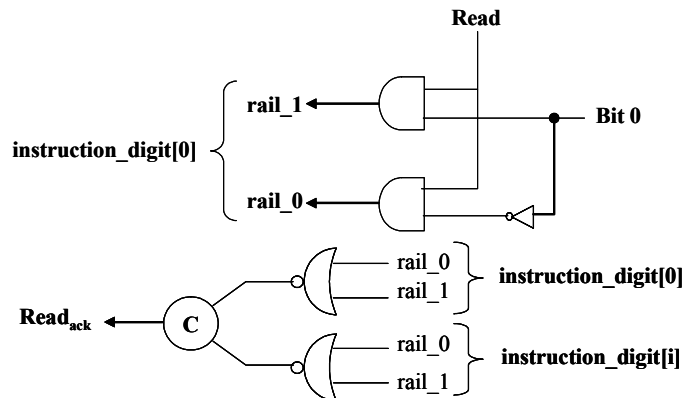


Figure 5.27 Interface mémoire instruction de sortie

5.5.4.3 Interface mémoire donnée

La connexion du cœur du mini processeur avec la mémoire donnée (mémoire RAM synchrone) se fait par l'intermédiaire d'une interface mémoire donnée (Figure 5.28).

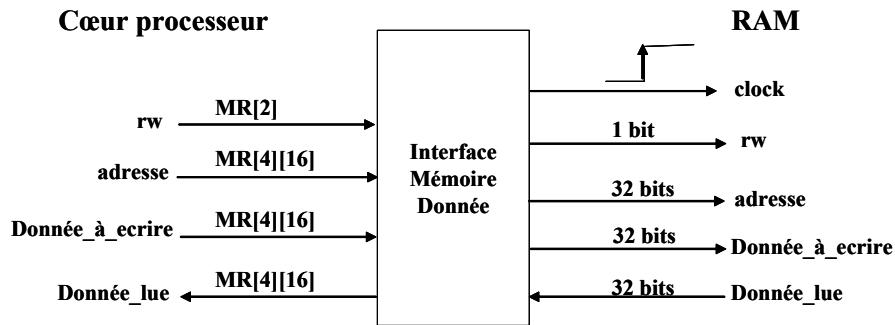


Figure 5.28 Interface mémoire donnée

Cette interface permet dans un premier temps de générer le signal d'horloge, le signal intermédiaire de lecture READ et les acquittements des canaux entrants. Ces signaux sont générés par le circuit de la Figure 5.29 à partir des canaux de lecture/écriture (rw), d'adresse et de donnée entrante (pour l'écriture).

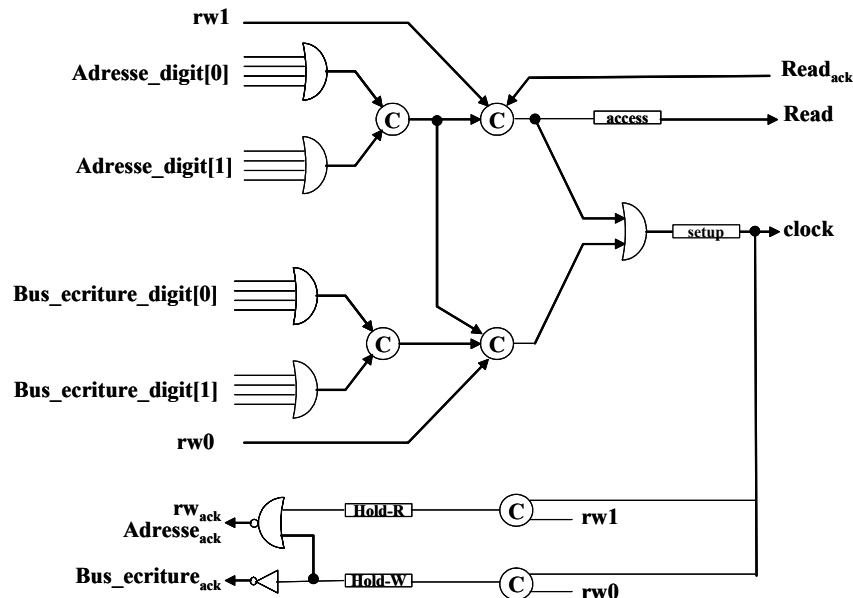


Figure 5.29 Interface mémoire donnée d'entrée

La conversion du canal rw (MR[2]) en un signal d'un bit se réalise très simplement en envoyant le fil 1 du canal rw. La conversion du canal d'adresse et du canal de donnée à écrire (tout deux codés en MR[4][16]) en signaux de 32 bits utilise le circuit de conversion présenté précédemment à la Figure 5.26.

La donnée qui est lue depuis la mémoire RAM doit être convertie en une donnée codée en MR[4][16]. Le circuit réalisant cette conversion est présenté à la Figure 5.30. Ce circuit utilise le signal intermédiaire READ généré par le circuit de la Figure 5.29.

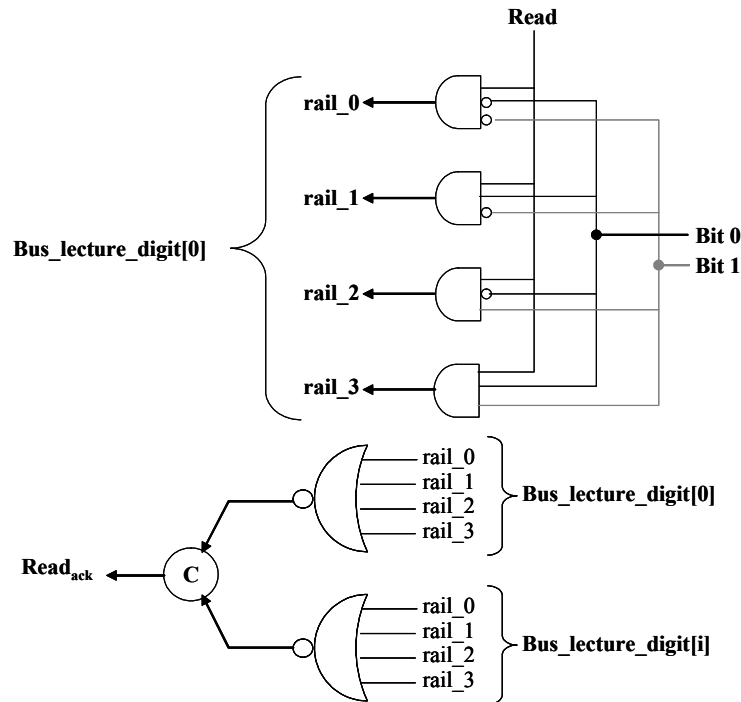


Figure 5.30 Interface mémoire donnée de sortie

5.5.5 Simulation du mini processeur

Les quatre versions du mini processeur après synthèse ont été entièrement simulées à l'aide d'un benchmark. Ce benchmark est un programme écrit en langage assembleur et compilé à l'aide d'un assembleur MIPS. Ce programme permet de tester tous les chemins du mini processeur y compris tous les registres du banc de registres pour tester le non collage à 0 ou à 1 des canaux. Ce test permet par la même occasion de tester tous les acquittements du chemin de donnée.

En ce qui concerne les opérateurs arithmétiques et logiques, on teste les quatre combinaisons d'entrée (\$00,\$00), (\$00,\$FF), (\$FF,\$00) et (\$FF,\$FF).

En testant toutes les instructions manipulant des données permettant d'exciter tous les chemins du mini processeur, on peut dire que le mini processeur ne présente pas d'erreurs d'implémentations.

5.6 Conclusion

Nous avons dans ce chapitre présenté une application complexe permettant d'illustrer notre méthodologie d'estimation de l'activité et de la consommation d'énergie d'une part et d'appliquer quelques techniques d'optimisation de la consommation d'énergie d'autre part. L'application complexe est un mini processeur de la famille MIPS dont l'architecture et le jeu d'instructions ont été introduits. La méthodologie d'estimation de la consommation d'énergie que nous avons présentée au Chapitre 2 a été comparée aux résultats obtenus par un outil d'estimation de la consommation d'énergie commercial (Nanosim). Les résultats montrent que les estimations produites par notre méthodologie sont relativement proches de ce que donne Nanosim. De plus, les optimisations faites sur le processeur qui ont abouties à la réalisation de quatre versions de processeur au total (V0, V1, V2, V3) ont montré une réduction intéressante de la consommation d'énergie. On a par la même occasion montré que la taille du mini processeur était quelque peu pénalisée par les optimisations faites (notamment le passage d'une codage MR[2] à un codage MR[4]), et que la vitesse du circuit était quant à elle

légèrement améliorée. Les quatre versions existent sous forme de netlist de portes VHDL mappées sur la technologie ST 0.12 μ m, elles ont été simulées avec succès.

Conclusion et Perspectives

Conclusion

L'objectif de la thèse est de proposer au concepteur de circuits intégrés numériques des moyens pour concevoir des circuits à faible consommation d'énergie. Ces moyens sont d'une part des outils (estimateur d'activité et de la consommation d'énergie) et d'autre part des techniques d'optimisation (incluant le passage à la logique asynchrone). Ces moyens peuvent être représentés par trois étapes principales :

- ✓ Le passage à la logique asynchrone
- ✓ L'estimation de l'activité et de la consommation d'énergie au moyen d'outils dédiés et développés dans le cadre de cette thèse
- ✓ La proposition de techniques d'optimisation de la consommation d'énergie

L'état de l'art avait pour objectif de montrer l'intérêt que présente la logique asynchrone en terme de modularité, de consommation d'énergie, de vitesse en temps moyen, d'émission électromagnétique, de sécurité et des solutions qu'elle apporte aux problèmes de la logique synchrone. La faible consommation d'énergie est néanmoins la propriété qui nous intéresse précisément pour réaliser un microprocesseur à faible consommation d'énergie. La réalisation d'un certain nombre de circuits asynchrones, notamment des processeurs asynchrones, montre les bénéfices que la logique asynchrone apporte. Les caractéristiques générales de la logique asynchrone et les microprocesseurs de grandes complexités réalisés ont été présentés. Ce chapitre a de plus montré les carences que présente la logique asynchrone en terme d'outils, et plus précisément d'estimateur d'activité et de la consommation d'énergie. Ces outils sont pourtant très importants lors de la conception de circuits intégrés à faible consommation d'énergie puisqu'ils permettent d'identifier les parties du circuit qui présentent la plus forte activité et la plus forte consommation d'énergie. Obtenir ces informations représente une étape importante pour apporter des optimisations judicieuses au circuit.

Nous avons ainsi formalisé une méthodologie d'estimation de la consommation d'énergie des circuits asynchrones en terme de nombre de commutations de porte. Cette méthodologie présente le précieux avantage d'obtenir des estimations très tôt dans le flot de conception, au niveau de la spécification CHP du circuit.

La méthodologie a été entièrement détaillée au Chapitre 2. Elle est réalisée en deux étapes principales qu'on nomme estimation structurelle et estimation dynamique. L'estimation structurelle est réalisée la première, elle permet d'attribuer un coût à chaque processus du circuit. L'estimation structurelle est établie à l'aide des équations de dépendances. On a expliqué qu'il existe une estimation structurelle par processus, elle utilise des opérateurs booléens lorsque le processus présente une structure de choix. L'équation de coût structurelle ainsi obtenue est disposée sur la sous partie du réseau de pétri représentant le processus qu'on estime. La simulation du réseau de pétri permet ensuite de calculer l'estimation dynamique du circuit. L'estimation dynamique correspond à une somme d'estimations structurelles lorsque

les places contenant des estimations structurelles sont atteintes lors de la simulation du réseau de pétri.

La présentation des estimations structurelles et dynamiques a été illustrée par une petite application qu'on a qualifiée d'hétérogène. Ce petit circuit regroupe toutefois la complexité de toutes les architectures de base et permet de bien comprendre comment les différents coûts sont obtenus et ce qu'ils représentent sur le circuit final.

Cette méthodologie d'estimation a conduit à la spécification d'un outil d'estimation de la consommation d'énergie. Cet outil permet dès la simulation du réseau de pétri d'obtenir des informations sur la consommation d'énergie et sur sa répartition en terme de nombre de commutations de porte. La spécification de cet outil a été faite au Chapitre 3. Il présente un petit nombre d'options permettant d'obtenir différents types d'information sur la consommation du circuit. On peut choisir un niveau de granularité allant de la consommation d'énergie d'une commande gardée à la consommation d'énergie de l'ensemble du composant simulé.

Dans ce même chapitre, un estimateur d'activité a été présenté. C'est un estimateur qui permet de fournir des informations sur l'activité et sur sa répartition. Il permet de donner des statistiques sur l'exécution des processus, des instructions, des gardes à un travers un nombre assez important d'options. L'activité du circuit est donnée à partir de la seule simulation du réseau de pétri. Des compteurs sont utilisés pour déterminer le nombre d'exécutions des transitions et des places du réseau de pétri. Ces compteurs sont ensuite exploités pour établir toutes sortes de statistiques sur l'exécution des processus, instructions etc. A travers des options très variées le concepteur peut recueillir un nombre important d'informations. L'estimateur a été spécifié et intégré dans le flot de conception TAST, il est parfaitement utilisable.

Ces deux outils montrent clairement leurs utilités dans la conception de circuits asynchrones à faible consommation d'énergie.

Le Chapitre 4 a proposé un certain nombre de techniques d'optimisation des circuits asynchrones. En partant dans un premier temps de l'estimation structurelle, on a comparé les implémentations possibles d'un certain nombre de circuits. Les estimations obtenues ont permis d'observer le comportement en fonction de paramètres tels que la bufferisation des composants, la synchronisation, la désynchronisation etc. Il s'agit de comparaisons permettant d'identifier l'implémentation la moins consommante.

Puis nous avons utilisé l'estimation d'activité et l'estimation de la consommation d'énergie (l'estimation dynamique) pour voir comment il est possible d'exploiter les statistiques pour déséquilibrer les structures de choix en privilégiant les cas les plus probables. Les informations obtenues sont utiles pour l'outil de synthèse afin qu'il puisse intégrer des options de synthèse à faible consommation d'énergie et au concepteur pour le guider dans ses choix d'implémentation.

Les estimations (d'activité et de la consommation d'énergie) et les optimisations (certaines seulement) ont été appliqués à la réalisation d'un circuit asynchrone complexe (un mini processeur MIPS asynchrone). Le Chapitre 5 a permis de comparer notre méthodologie d'estimation de la consommation d'énergie du circuit à l'estimation obtenue par un outil commercial (Nanosim) du circuit après synthèse. Notre méthodologie montre que nous obtenons des résultats pertinents et qui reflètent les proportions de la consommation d'énergie dans le circuit.

Nous avons appliqué trois étapes d'optimisation et montré les gains apportés grâce à ces optimisations. On obtient au final quatre versions (V0, V1, V2, V3) de processeurs montrant

une réduction progressive de la consommation d'énergie en fonction des étapes d'optimisation.

Les quatre versions existent sous forme de netlist de portes VHDL mappées sur une technologie ST 0.12 μ m, elles ont été simulées avec succès.

La réalisation de ces quatre versions de processeur avait pour but de valider la méthodologie d'estimation et certaines techniques d'optimisation de la consommation d'énergie. Nous obtenons au final un processeur dont la consommation a été réduite de plus de 20%. Il est possible d'améliorer grandement les performances de ce processeur en appliquant d'autres techniques d'optimisation de la consommation d'énergie (adaptation de la bufferisation entrée/sortie, désynchronisation intra canal, etc.). On peut aussi réduire la taille du circuit en réalisant un partage de ressources et en mappant la netlist sur une librairie dédiée à l'asynchrone telle que TAL.

Les objectifs que nous nous étions fixés au début de la thèse ont été atteints et les résultats obtenus par ces travaux permettent d'enrichir le flot de conception TAST avec deux estimateurs et une option de synthèse à faible consommation d'énergie. Les outils qui se greffent petit à petit sur TAST montrent clairement la volonté de proposer au concepteur une diversité d'outils lui permettant de réaliser tout type de circuit performant. C'est un éventail d'outils qui n'existe pas dans tous les flots de conception asynchrone.

Perspectives

- ✓ Réalisation de l'outil d'estimation de la consommation d'énergie

La méthodologie d'estimation de la consommation d'énergie a été entièrement formalisée, le travail qui nécessite d'être fait est l'implémentation de l'outil et l'intégration de celui-ci dans le flot de conception TAST.

- ✓ Synthèse faible consommation d'énergie

Introduire les informations sur la réduction de la consommation d'énergie dans l'outil de synthèse. L'outil tiendra compte de ces informations afin de synthétiser des architectures à faible consommation d'énergie.

- ✓ Réalisation du processeur MIPS4Ksc [MIPS-01]

Le processeur MIPS 4Ksc a été entièrement spécifié en CHP.

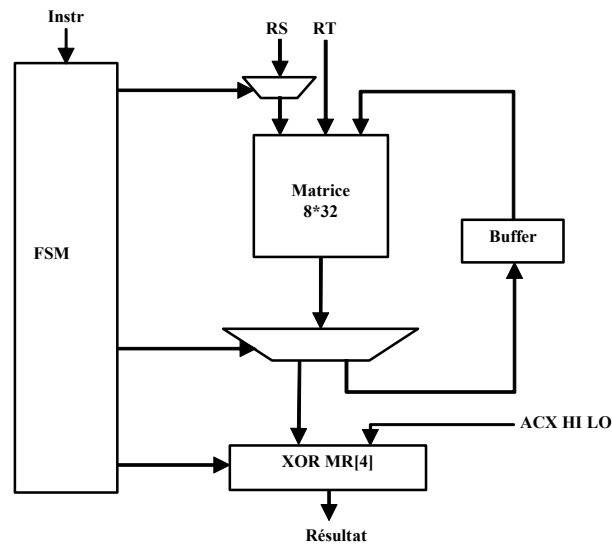
Il faut ajouter les interruptions et la gestion de la MMU.

Il faut de plus faire la synthèse de l'ensemble du processeur qui ne présente pas tout le temps une spécification CHP synthétisable.

Des travaux ont déjà été menés et ont abouti à la réalisation de certaines unités:

- Multiplieur polynomial

Un multiplieur polynomial a été entièrement implémenté [LAWN-03]. L'architecture est très performante, elle est présentée à la figure ci-dessous.



Multiplieur polynomial asynchrone

- Banc de Registres

Un banc de registres dédié à la faible consommation d'énergie a été entièrement réalisé. Ce banc de registres a la particularité d'adapter la taille des opérands à envoyer en fonction de l'instruction. Seuls les digits nécessaires à l'opérateur qui les sollicite sont envoyés [PALF-03].

- Unité d'exécution

L'unité d'exécution du MIPS 4Ksc a été entièrement synthétisée. Elle a été réalisée à l'aide d'une architecture à faible consommation d'énergie [RIOS-04].

Publications et Réalisations Personnelles

✓ **Publications**

K. Slimani, Y. Remond, G. Sicard, M. Renaudin, "TAST Profiler and Low Energy Asynchronous Design Methodology" Proceedings of 14th International Workshop on Power and Timing Modeling, Optimization and Simulation, Sept. 2004, Isle of Santorini, Greece, Pages 268-277.

K. Slimani, J. Fragoso, L. Fesquet, Marc Renaudin, "Low Power Asynchronous Processors", Chapter 22 in "Low-Power Electronics Design" written by Christian Piguet", Published by CRC Press, July 2004, ISBN 0849319412.

K. Slimani, Y. Remond, A. Sirianni, G. Sicard, M. Renaudin, "Estimation et Optimisation de la Consommation d'Énergie des Circuits Asynchrones", 4^{ème} journées francophones d'étude Faible Tension Faible Consommation (FTFC'2003), Mai. 2003, Paris, France.

K. Slimani, G. Sicard, M. Renaudin, "Estimation et Optimisation de la Consommation d'Énergie des Circuits Asynchrones", VI^{èmes} Journées Nationales du Réseau Doctoral Microélectronique (JNRDM 2003), Mai. 2003, Toulouse, France.

F. Bouesse, M. Renaudin, K. Slimani
Rapport de Projet Européen G3card : "Ultra Secure SmartMIPS Design Report", 2002.

✓ **Tutoriaux et démonstrations**

A. Sirianni, Y. Remond, K. Slimani, M. Renaudin, "Estimateur d'Activité et Estimateur de la Consommation d'Énergie des Circuits Asynchrones", Tutorial at the Design And Test in Europe (DATE 2003 conference), Munich, 2003.

M. Renaudin, Y. Remond, K. Slimani, "TAST Profiler", Tutorial at the Design And Test in Europe (DATE 2004 conference), Paris, 2004

M. Renaudin, K. Slimani, J. Fragoso, E. Allier, M. Essalhiène, B. Galilée, Y. Remond, A. Sirianni, J. Quartana, F. Aeschlimann, "La conception de circuits asynchrones et la faible consommation" RTP SoC-AS Faible Conso, ISEP, Paris, France, 20 Novembre 2003.

✓ **Réalisations de circuits**

F. Bouesse, M. Renaudin, K. Slimani
Réalisation d'une unité de Multiplication et Division Asynchrone qui a été intégrée dans un cœur MIPS synchrone. Circuit ASIC en technologie TSMC 0.18 μm . Tape out du circuit juin 2002. Circuit réalisé dans le cadre du Projet G3Card.

Réalisation d'un mini processeur MIPS (4 versions disponibles) en technologie ST 0.12 μm . Disponible en netlists portes VHDL.

Bibliographie

[ABRI-01] A. Abrial, J. Bouvier, M. Renaudin, P. Senn and P. Vivet, "A New Contactless Smart Card IC using On-Chip Antenna and Asynchronous Microcontroller", *Journal of Solid-State Circuits*, Vol. 36, 2001, pp. 1101-1107.

[AKEL-92] V. Akella and G. Gopalakrishnan. SHILPA: A high-level synthesis system for self-timed circuits. In *Proc. of the IEEE/ACM International Conference on Computer Aided Design*, pages 587--591. IEEE Computer Society Press, November 1992.

[APPL-97] S. Appleton, S. Morton, and M. Liebelt. High performance two-phase asynchronous pipelines. *IEICE Transactions on Information and Systems*, E80-D(3):287-295, March 1997.

[BAIN-98] W. J. Bainbridge and S. B. Furber. An asynchronous macrocell interconnect using MARBLE In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 122-132, 1998.

[BAIN-01] W. J. Bainbridge and S. B. Furber. Delay insensitive system-on-chip interconnect using 1-of-4 data encoding. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 118-126. IEEE Computer Society Press, March 2001.

[BARD-02] Doug Edwards and Andrew Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12-18, 2002.

[BEER-95] P. A. Beerel, C.-T. Hsieh, and S. Wadekar. Estimation of energy consumption in speed-independent control circuits. In *International Symposium on Low-Power Design*, pages 39-44, 1995.

[BENI-97] L. Benini, P. Siegel, and G. De Micheli, "Saving power by synthesizing gated clocks for sequential circuits," *IEEE Design Test Comput.*, vol.11, pp. 32-41, 1994. synthesis for a broad range of design applications," in *Proc. ICCAD*, 1997.

[BERK-91] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalijs. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, pages 384-389, 1991.

[BERK-95] Kees van Berkel and Martin Rem. VLSI programming of asynchronous circuits for low power. In Graham Birtwistle and Al Davis, editors, *Asynchronous Circuit Design, Workshops in Computing*, pages 152-210. Springer-Verlag, 1995.

[BOUE-04a] G.F. BOUESSE, G. SICARD, A. BAIXAS, M. RENAUDIN, "Quasi Delay Insensitive Asynchronous Circuits for low EMI", 4th International Workshop on Electromagnetic Compatibility of Integrated Circuits. (EMC COMPO 2004), March 31st, 2004, Angers, France.

[BOUE-04b] F. Bouesse, M. Renaudin, B. Robisson, E Beigne, P.Y. Liardet, S. Prevosto, J. Sonzogni, "DPA on Quasi Delay Insensitive Asynchronous circuits: Concrete Results", To be published in XIX Conference on Design of Circuits and Integrated Systems Bordeaux, France, November 24-26, 2004.

[BRUN-93] E. Brunvand, "The NSR Processor," *Proc. 26th Hawaii Int'l Conf. System Sciences*, Vol. 1, T.N. Mudge, V. Milutinovic, and L. Hunter, eds., IEEE Press, Piscataway, N.J., 1993, pp. 428-435

[CHO-92] Design of a 32-bit fully asynchronous microprocessor (FAM) Cho, K.-R.; Okura, K.; Asada, K.; *Circuits and Systems, 1992.*, Proceedings of the 35th Midwest Symposium on , 9-12Aug. 1992 Page(s): 1500 -1503 vol.2

[CHOI-99] The design of delay insensitive asynchronous 16-bit microprocessor Byung-Soo Choi; Dong-Wook Lee; Dong-Ik Lee; *Design Automation Conference, 1999. Proceedings of the ASP-DAC '99. Asia and South Pacific* , 18-21 Jan. 1999 Page(s): 33 -36 vol.1

[CHRI-98] K. T. Christensen, P. Jensen, P. Korger, and J. Sparsø. The design of an asynchronous Tiny RISC TR4101 microprocessor core. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 108-119, 1998.

[CORT-96] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. Technical report, Universitat Politècnica de Catalunya, 1996

[COVE] <http://www.covermeter.com>

[DAVI-93] A. Davis, B. Coates, K. Stevens, "The Post Office Experience: Designing a Large Asynchronous Chip", in *Proceedings of the 26th Annual Hawaii International Conference on Systems Sciences*, Vol. I, pp. 409-418,1993.

[DEAN-92] M.E. Dean, *STRIP : A Self-Timed RISC Processor*, Tech. Report CSL-TR-92-543, Stanford Univ., Stanford, Calif., July 1992.

[DIAZ-01] Michel Diaz, "Les réseaux de Pétri", HERMES Science Europe Lt, 2001.

[DINH-02a]A.V. Dinh Duc, J.B. Rigaud, A. Rezzag, A. Sirianni, J. Fragoso, L. Fesquet, M. Renaudin, "TAST CAD Tools: Tutorial", tutorial given at the 8th IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems, Apr. 2002, Manchester, UK. TIMA internal report ISRN:TIMA-RR-02/04/01-FR, <http://tima.imag.fr/cis>

[DINH-02b] A.V. Dinh-Duc, L. Fesquet and M. Renaudin, Synthesis of QDI asynchronous circuits from DTL-Style Petri Net, in *Proc. of 11th IEEE/ACM Intl. Workshop on Logic and Synthesis (IWLS)*, pp. 191-196, June 2002.

[EBER-91] J. Ebergen, "A formal approach to designing delay-insensitive circuits, *Distributed Computing*, Vol. 5, N°.3, pp.107-119, July 1991.

[EFTH-03] A. Efthymiou, J.D. Garside, "Adaptive Pipeline Structures for Speculation Control". Proceedings of 9th IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems, May 2003. Vancouver, B.C., Canada. pp. 46-55.

[ENDE-96] Superscalar instruction issue in an asynchronous microprocessor Endecott, P.B.; Computers and Digital Techniques, IEE Proceedings- , Volume: 143 Issue: 5 , Sept. 1996 Page(s): 266 –272

[ENDE-98] P.B. Endecott and S.B. Furber Modelling and Simulation of Asynchronous Systems using the LARD Hardware Description Language Proceedings of the 12th European Simulation Multiconference, Manchester, June 1998, Society for Computer Simulation International, pages 39-43. ISBN 1-56555-148-6.

[ESSA-02] M. Es Salhiene, L. Fesquet , M. Renaudin, "Dynamic Voltage Scheduling for Real Time Asynchronous Systems", 12th International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS), Sept. 2002, Sevilla, Spain.

[FALL-01] Fallah-F; Devadas-S; Keutzer-K. "OCCOM-efficient computation of observability-based code coverage metrics for functional verification". IEEE-Transactions-on-Computer-Aided-Design-of-Integrated-Circuits-and-Systems. Vol.20, no.8; Aug. 2001; p.1003-15.

[FESQ-04] Laurent Fesquet, Mohammed Es Salhiene, Marc Renaudin, "La technologie asynchrone au service de la réduction d'énergie dans les systèmes embarqués", Annales des Télécommunications, Tome 59, n°7-8, juillet-août 2004.

[FRAG-03] João Loenardo Fragoso, Gilles Sicard, and Marc Renaudin. Power/area tradeoffs in 1-of-M parallel-prefix asynchronous adders. In Jorge Juan Chico and Enrico Macii, editors, Power and Timing Modeling, Optimization and Simulation (PATMOS), volume 2799 of Lecture Notes in Computer Science, pages 171-180, September 2003.

[FUHR-99] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, B. Lin, and L. Plana. Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical Report TR CUCS-020-99, Columbia University, NY, July 1999.

[FURB-94] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. AMULET1: A micropipelined ARM. In Proceedings IEEE Computer Conference (COMPCON), pages 476-485, March 1994.

[FURB-96] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and S. Temple. AMULET2e. In C. Muller-Schloer, F. Geerinckx, B. Stanford-Smith, and R. van Riet, editors, Embedded Microprocessor Systems, September 1996. Proceedings of EMSYS'96 - OMI Sixth Annual Conference

[FURB-98] S. B. Furber, J. D. Garside, and S. Temple. Power-saving features in Amulet2e. In Power Driven Microarchitecture Workshop, June 1998.

[FURB-00] S. B. Furber, D. A. Edwards, and J. D. Garside. AMULET3: a 100 MIPS asynchronous embedded processor. In Proc. International Conf. Computer Design (ICCD), September 2000.

[G3CARD-02] Projet Européen G3CARD "Ultra Secure SmartMIPS Design Report". 2002.

Tima : F. Bouesse, M. Renaudin, K. Slimani

Gemplus : V. Corlay, A. Fermy, J. Fournier

[GAGE-98] Hans van Gageldonk, Daniel Baumann, Kees van Berkel, Daniel Gloor, Ad Peeters, and Gerhard Stegmann. An asynchronous low-power 80c51microcontroller. In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 96-107, 1998.

[GARS-00] J. D. Garside, W. J. Bainbridge, A. Bardsley, D. A. Edwards, S. B. Furber, J. Liu, D. W. Lloyd, S. Mohammadi, J. S. Pepper, O. Petlin, S. Temple, and J. V. Woods. AMULET3i --- an asynchronous system-on-chip. In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 162-175. IEEE Computer Society Press, April 2000

[GHOS-92] A. Ghosh, S. Devadas, K. Keutzer and J. White. "Estimation of average switching activity in combinational and sequential circuits.", Proceedings, 29th ACM/IEEE Design Automation Conference, 1992.

[HOAR-78] Hoare C. A. R, Communicating Sequential Processes, Communications of the ACM 21, vol 8, (April 1978), pp.666-677.

[LAWN-03] G. Lawny, " Conception de l'unité de multiplication et de division du processeur MIPS asynchrone", stage de Maitrise d'informatique, Institut National Polytechnique de Grenoble, 2003.

[LEWI-99] M. Lewis, J.D. Garside, L.E.M. Brackenbury, "Reconfigurable Latch Controllers for Low Power Asynchronous Circuits", Proceedings of ASYNC '99, Apr. 1999. pp. 27-35.

[LINE-95] Andrew Matthew Lines, "Pipelined Asynchronous Circuits". Master Thesis, 1995.

[MANO-01] R. Manohar, "Width-Adaptive Data Word Architectures", Proceedings of 19th Conference on Advanced Research in VLSI, 2001. pp. 112-129.

[MART-89] The Design of an Asynchronous Microprocessor. A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, and P.J. Hazewindus. Decennial Caltech Conference on VLSI, ed. C.L. Seitz, 351-373, MIT Press, 1989.

[MART-90a] A. J. Martin, "The Limitations to Delay-Insensitivity in Asynchronous Circuits", in Proceedings of the 1990 MIT Conference on Advanced Research in VLSI, pp. 263-278, 1990.

[MART-90b] Alain J. Martin. Programming in VLSI : from communicating processes to delay insensitive circuits. In C. A. R Hoare, editor, *Developments in concurrency and Communication*, UT Year of Programming Series, pages 1-64. Addison-Wesley, 1990.

[MART-91] Martin, Alain J. (1991) Synthesis of Asynchronous VLSI Circuits. <http://resolver.caltech.edu/CaltechCSTR:1991.cs-tr-93-28>

[MART-97] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nystroem, Paul Penzes, Robert Southworth, and Uri Cummings. The design of an asynchronous MIPS R3000 microprocessor. In Advanced Research in VLSI, pages 164-181, September 1997

[MART-00] Alain J. Martin. An asynchronous approach to energy-efficient computing and communication. In Proc. SSGRR 2000, International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet, August 2000.

[MART-01] A. Martin, M. Nyström and P. Penzes, "ET2: A Metric for Time and Energy Efficiency of Computation". Power-Aware Computing, R. Mellhem and R. Graybill, ed., Kluwer Academic Publishers, 2001.

[MART-03] The Lutonium: A Sub-Nanojoule Asynchronous 8051 Microcontroller. Alain J. Martin, Mika Nyström, Karl Papadantonakis, Paul I. Penzes, Piyush Prakash, Catherine G. Wong, Jonathan Chang, Kevin S. Ko, Benjamin Lee, Elaine Ou, James Pugh, Eino-Ville Talvala, James T. Tong, Ahmet Tura. Accepted for publication, 9th IEEE International Symposium on Asynchronous Systems & Circuits, 2003

[MAUR-03] P. MAURINE, J.B. RIGAUD ; F. BOUESSE ; G. SICARD ; M RENAUDIN. "TAL : Une Bibliothèque de Cellules pour le Design de Circuits Asynchrones QDI", 4ème journées francophones d'étude Faible Tension Faible Consommation (FTFC'2003), May 2003, Paris, France.

[MIPS-01] MIPS32 4K™ Processor Core Family. Software User's Manual. January 3, 2001.

[MODE]

http://www.model.com/products/tours/58/code_coverage_web/code_coverage_web.htm

[MONN-04] Y. Monnet, M. Renaudin, R. Leveugle, "Asynchronous circuits sensitivity to fault injection", 10th IEEE International On-Line Testing Symposium, Madeira Island, Portugal, July 12-14, 2004.

[MOOR-02] Simon Moore, Ross Anderson, Paul Cunningham, Robert Mullins, and George Taylor. Improving smart card security using self-timed circuits. In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 211-218, April 2002.

[MORT-95] ECSTAC: a fast asynchronous microprocessor. Morton, S.V.; Appleton, S.S.; Liebelt, M.J ; Asynchronous Design Methodologies, 1995. Proceedings, Second Working Conference on , 30-31 May 1995 Page(s): 180 –189

[NANY-94] Takashi Nanya, Yoichiro Ueno, Hiroto Kagotani, Masashi Kuwako, and Akihiro Takamura. TITAC: Design of a quasi-delay-insensitive microprocessor. IEEE Design & Test of Computers, 11(2):50-63, 1994.

[NOWI-97] S. M. Nowick and M. Theobald. Synthesis of low-power asynchronous circuits in a specified environment. In International Symposium on Low Power Electronics and Design, pages 92-95, 1997.

[PALF-03] T. Palfer-Sollier, "Conception d'un Banc de Registres Asynchrone", Rapport de Stage 3ieme Année ENST Bretagne, Grenoble, 2003.

[PANY-02] D. Panyasak, G. Sicard, M. Renaudin, "A current shaping methodology for low EMI asynchronous circuits", EMC Compo'02, pp.43-48, Toulouse, France, nov-02.

[PANY-04] D. Panyasak, G. Sicard, M. Renaudin, "A current shaping methodology for lowering EM disturbances in asynchronous circuits", Microelectronics journal 35 June-2004 pp531-540.

[PENZ-02a] Paul I. Péntzes and Alain J. Martin. An energy estimation method for asynchronous circuits with application to an asynchronous microprocessor. In Proc. Design, Automation and Test in Europe (DATE), pages 640-647, March 2002.

[PENZ-02b] Paul Péntzes, Mika Nyström, and Alain J. Martin. Transistor sizing of energy-delay-efficient circuits. In Proc. of the Great Lakes Symposium on VLSI,2002.

[PIGU-98] C. Piguet. Design of low-power libraries. In IEEE International Conference on Electronics, Circuits and Systems, pages 175-180, 1998.

[PLAN-02] L. A. Plana, P. A. Riocreux, W. J. Bainbridge, A. Bardsley, J. D. Garside, and S. Temple. SPA - a synthesisable amulet core for smartcard applications. In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 201-210, April 2002.

[POWE] http://www.synopsys.com/products/etg/powermill_ds.html

[PRAB-94] Prabhakar Kudva and Venkatesh Akella. A technique for estimating power in self-timed asynchronous circuits. In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 166-175, November 1994.

[RENA-98] M. Renaudin, P. Vivet, and F. Robin. ASPRO-216: A standard-cell QDI 16-bit RISC asynchronous microprocessor. In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 22-31, 1998.

[RENA-00] M. Renaudin, "Asynchronous circuits and systems : a promising design alternative", Microelectronic Engineering 54 (2000) 133-149.

[RIGA-02] J.B. Rigaud, "Spécifications de bibliothèques pour la synthèse de circuits asynchrones", *PhD thesis* (in French), Institut National Polytechnique de Grenoble, 2002.

[RIOS-04] D. Rios, "Microprocesseur Asynchrone très faible consommation", Stage de DEA, Institut National Polytechnique de Grenoble, 2004.

[SLIM-03] K. Slimani, Y. Remond, A. Sirianni, G. Sicard, M. Renaudin, "Estimation et Optimisation de la Consommation d'Energie des Circuits Asynchrones", 4ème journées

francophones d'étude Faible Tension Faible Consommation (FTFC'2003), Mai. 2003, Paris, France.

[SLIM-04] K. Slimani, Y. Remond, G. Sicard, M. Renaudin, "TAST Profiler and Low Energy Asynchronous Design Methodology" Proceedings of 14th International Workshop on Power and Timing Modeling, Optimization and Simulation, Sept. 2004, Isle of Santorini, Greece, Pages 268-277.

[SPEC] http://www.cadence.com/datasheets/spectre_cir_sim.html

[SPRO-94] R. F. Sproull and I. E. Sutherland and C. E. Molnar, "The Counterflow Pipeline Processor Architecture", IEEE Design and Test of Computers, pp. 48-59, Vol. 11, No. 3, Fall 1994

[STRO-00] Analysis of power dissipation in double edge-triggered flip-flops Strollo, A.G.M.; Napoli, E.; Cimino, C.; Very Large Scale Integration (VLSI) Systems, IEEE Transactions on , Volume: 8 Issue: 5 , Oct 2000
Page(s): 624 -629

[SUTH-89] Ivan E. Sutherland. Micropipelines. Communications of the ACM, 32(6):720-738, June 1989.

[SUTH-99] Ivan Sutherland, Bob Sproull, David Harris, "Logical Effort, Designing Fast CMOS Circuits", Morgan Kaufmann Publishers, Inc. San Francisco, California, 1999.

[TAKA-97] Akihiro Takamura, Masashi Kuwako, Masashi Imai, Taro Fujii, Motokazu Ozawa, Izumi Fukasaku, Yoichiro Ueno, and Takashi Nanya. TITAC-2: An asynchronous 32-bit microprocessor based on scalable-delay-insensitive model. In Proc. International Conf. Computer Design (ICCD), pages 288-294, October 1997.

[TAST-02] <http://tima.imag.fr/cis/> TAST Project

[TEIF-02] J. Teifel, D. Fang, D. Biermann, C. Kelly, and R. Manohar, "Energy-efficient pipelines", Proceedings of 8th IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems, Apr. 2002. pp. 23-33.

[TIER-96] The Energy and Entropy of VLSI Computations. Jose A. Tierno, Rajit Manohar, and Alain J. Martin. Async96 Second International Symposium on Advanced Research in Asynchronous Circuits and Systems, March 1996.

[TSAN-99] MSL16A: an asynchronous Forth microprocessor Tsang, P.K.; Cheung, C.C.; Leung, K.H.; Lee, T.K.; Leong, P.H.W.; TENCON 99. Proceedings of the IEEE Region 10 Conference, Volume: 2 , 15-17 Sept. 1999 Page(s): 1079 -1082 vol.2

[TSE-97] ASYNMPU: a fully asynchronous CISC microprocessor Tse, J.M.C.; Lun, D.P.K.; Circuits and Systems, 1997. ISCAS'97. Proceedings of 1997 IEEE International Symposium on , Volume: 3 , 9-12 June 1997 Page(s): 1816 -1819 vol.3

[VIVE-01] P. Vivet, "Une Méthodologie de Conception de Circuits Intégrés Quasi-Insensibles aux Délais: Application à l'Etude et à la Réalisation d'un Processeur RISC 16-bit Asynchrone. *PhD thesis* (en francais), Institut National Polytechnique de Grenoble, 2001.

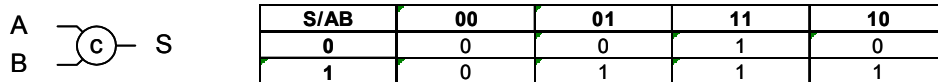
[WILL-94] T.E Williams, "Performance of iterative computation in self timed rings", *Journal of VLSI Signal Processing*, 7:17-31, Feb. 1994.

[YUN-96] K. Y. Yun, P. A. Beerel, and J. Arceo. High-performance asynchronous pipeline circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

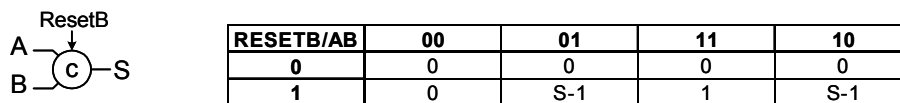
Annexe

A) Portes asynchrones de base

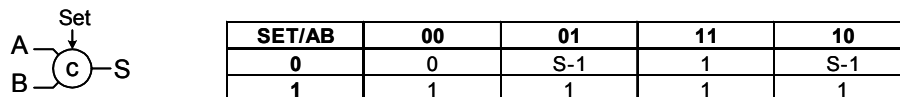
Müller 2 (Muller à 2 entrées)



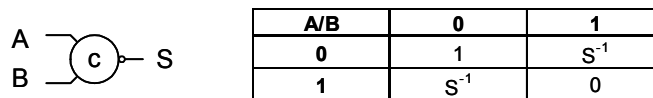
Müller 2 R (Muller à 2 entrées avec reset)



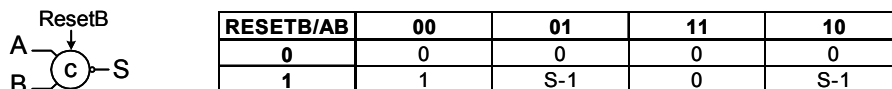
Müller 2 S (Muller à 2 entrées avec set)



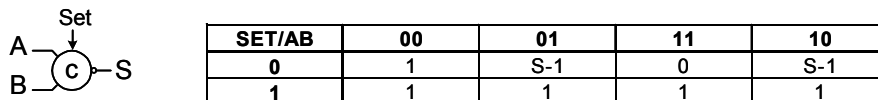
Müller 2 B (Muller à 2 entrées avec sortie complémentée)



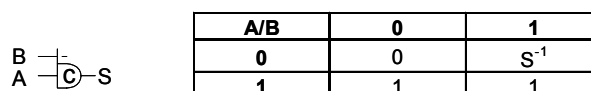
Müller 2 B R (Muller à 2 entrées avec sortie complémentée avec reset)



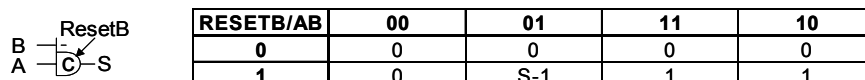
Müller 2 B S (Muller à 2 entrées avec sortie complémentée avec set)



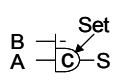
Müller 2 D 1 N (Muller dissymétrique négative)



Müller 2 D 1 N R (Muller dissymétrique négative avec reset)

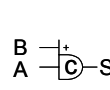


Müller 2 D 1 N S (Muller dissymétrique négative avec set)



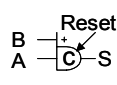
SET/AB	00	01	11	10
0	0	S-1	1	1
1	1	1	1	1

Müller 2 D 1 P (Muller dissymétrique positive)



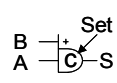
A/B	0	1
0	0	0
1	S ⁻¹	1

Müller 2 D 1 P R (Muller dissymétrique positive avec reset)



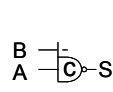
RESETB/AB	00	01	11	10
0	0	0	0	0
1	0	0	1	S-1

Müller 2 D 1 P S (Muller dissymétrique positive avec set)



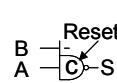
SET/AB	00	01	11	10
0	0	0	1	S-1
1	1	1	1	1

Müller 2 D 1 N B (Muller dissymétrique négative avec sortie complémentée)



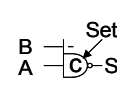
A/B	0	1
0	1	S ⁻¹
1	0	0

Müller 2 D 1 N B R (Muller dissymétrique négative avec sortie complémentée avec reset)



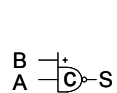
RESETB/AB	00	01	11	10
0	0	0	0	0
1	1	S-1	0	0

Müller 2 D 1 N B S (Muller dissymétrique négative avec sortie complémentée avec set)



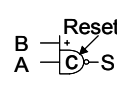
SET/AB	00	01	11	10
0	1	S-1	0	0
1	1	1	1	1

Müller 2 D 1 P B (Muller dissymétrique positive avec sortie complémentée)



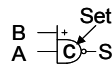
A/B	0	1
0	1	1
1	S ⁻¹	0

Müller 2 D 1 P B R (Muller dissymétrique positive avec sortie complémentée avec reset)



RESETB/AB	00	01	11	10
0	0	0	0	0
1	1	1	0	S-1

Müller 2 D 1 P B S (Muller dissymétrique positive avec sortie complémentée avec set)



SET/AB	00	01	11	10
0	1	1	0	S-1
1	1	1	1	1

Müller 3 D 1 P 1 N (Muller dissymétrique positive négative)



C/AB	00	01	11	10
0	0	0	1	S-1
1	S-1	S-1	1	S-1

Müller 3 D 1 P 1 N R (Muller dissymétrique positive négative avec reset)



RESETBC/AB	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	S-1	S-1	1	S-1
10	0	0	1	S-1

Müller 3 D 1 P 1 N S (Muller dissymétrique positive négative avec set)



SETC/AB	00	01	11	10
00	0	0	1	S-1
01	S-1	S-1	1	S-1
11	1	1	1	1
10	1	1	1	1

Müller 3 D 1 P 1 N B (Muller dissymétrique positive négative avec sortie complémentée)



C/AB	00	01	11	10
0	1	1	0	S-1
1	S-1	S-1	0	S-1

Müller 3 D 1 P 1 N B R (Muller dissymétrique positive négative avec sortie complémentée avec reset)



RESETBC/AB	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	S-1	S-1	0	S-1
10	1	1	0	S-1

Müller 3 D 1 P 1 N B S (Muller dissymétrique positive négative avec sortie complémentée avec set)



SETC/AB	00	01	11	10
00	1	1	0	S-1
01	S-1	S-1	0	S-1
11	1	1	1	1
10	1	1	1	1

B) Syntaxe et Sémantique du CHP

Nous présentons dans cette annexe la syntaxe et la sémantique d'un programme écrit en langage CHP du flot de conception TAST.

1) Introduction

Un programme CHP consiste en une spécification de composants. Un composant communique avec les autres composants via des ports, il est composé de processus parallèles et d'instances de composants.

La syntaxe générale d'un programme CHP est de la forme suivante :

COMPONENT *<component name >*

<port declarations>

<component, channel and constant declarations>

BEGIN

<process declaration or component instance>

END;

2) Types de données

a) Type de données non signé

Les types de base non signés du langage CHP sont:

- $MR[B]$: Type Multi Rail non signé dans une base B (codage 1 parmi B). Ce type représente un nombre entre 0 et B-1
- $MR[B][L]$: Vecteur de L éléments de type Multi Rail non signé dans une base B. Ce type représente un nombre entre 0 et B^L-1
- $MR[B][L][D]$: Tableau de dimension D contenant des éléments du type Vecteur $MR[B][L]$

Ces trois types sont les types non signés de base. Les types que nous présentons maintenant sont des alias de ces types de base :

- DR : Type Dual Rail. Ce type est équivalent au type $MR[2][1][1]$. Il est utilisé pour la représentation du digit binaire.
- BIT : Type binaire. Ce type représente le type binaire comme pour le type Dual Rail.
- BOOLEAN : Type booléen. Il a la même représentation que le type BIT
- SR : Simple Rail. Ce type correspond à un $MR[1][1][1]$ ou $MR[1]$. IL permet la synchronisation entre des éléments communicants. Il permet de déclencher un ensemble d'actions dans un processus ou un composant.

- DR[L] : Vecteur de L éléments du type Dual Rail. Ce type est équivalent à un MR[2][L].
- BIT[L] : Vecteur de bits. Ce type est équivalent à un DR[L].
- BOOLEAN[L]: Vecteur de Booléens. Ce type est équivalent à un MR[2][L][1].
- NATURAL[MAX] : représente un nombre naturel compris entre 0 et la valeur MAX.
- DR[L][D] : Tableau de dimension D contenant des éléments du type vecteur DR[L].
- BIT[L][D] : Tableau de dimension D contenant des éléments du type vecteur BIT[L].
- BOOLEAN[L][D] : Tableau de dimension D contenant des éléments du type vecteur BOOLEAN[L].
- NATURAL[Max][D] : Tableau de dimension D contenant des éléments du type NATURAL[Max].

b) Type de données signés

Les types de base signés du langage CHP sont:

- SMR[B] : Type Multi Rail signé dans la base B. Ce type représente un nombre entre $-(B/2)$ et $(B/2)-1$
- SMR[B][L] : Vecteur d'éléments du type Multi Rail signé dans la base B.
- SMR[B][L][D] : Tableau de dimension D contenant des éléments du type vecteur SMR[B][L].

Ces trois types sont les types signés de base. On présente maintenant des types qui sont des alias de ces types de base :

- SDR : Type Dual Rail signé. Ce type est équivalent à un SMR[2].
- SDR[L] : Vecteur d'éléments du type Dual Rail signé. Ce type est équivalent au type SMR[2][L].
- INTEGER [MAX] : Type entier compris entre $-(MAX+1)$ à MAX. Ce type est équivalent au type binaire SMR[2][L].
- SDR[L][D] : Tableau de dimension D contenant des éléments du type vecteur SDR[L].

- INTEGER[Max][D] : Tableau de dimension D contenant des éléments du type INTEGER[Max].

3) Instructions

a) structures de Contrôle

- Structure de contrôle déterministe

La structure de contrôle déterministe attend jusqu'à ce qu'une garde et une seule garde soit vraie pour exécuter l'instruction correspondante.

```
@[  
guarded commands (BREAK|LOOP)  
guarded commands (BREAK|LOOP)  
{guarded commands (BREAK|LOOP)...}  
];
```

- Structure de contrôle indéterministe

La structure de contrôle indéterministe attend jusqu'à ce qu'au moins une garde soit vraie. Pour cette structure, il est possible que plusieurs gardes soient vraies en même temps. Dans ce cas, la structure fait un tirage aléatoire d'une garde parmi l'ensemble des gardes vraies et exécute l'instruction correspondante.

```
@@[  
guarded commands (BREAK|LOOP)  
guarded commands (BREAK|LOOP)  
{guarded commands (BREAK|LOOP)...}  
];
```

b) Opérations sur les ports et les canaux

- Réception d'une donnée (?)
- Emission d'une donnée (!)
- Sonde (Probe) #

c) Opérateurs

- Affectation (:=)
- Opérateurs de comparaisons (=, /=, <, >, <=, >=)
- Opérateurs Logiques (OR, NOR, AND, NAND, XOR, XNOR, NOT)
- Opérateurs arithmétiques (+, -, *, /, MOD, ABS)
- Opérateurs de décalage (SRL, SRA, SLL, SLA, ROTL, ROTR)
- Composition (;) et (,)
- Divers (skip)

RESUME

L'évolution effrénée de la complexité des appareils portables s'accompagne d'une augmentation de la consommation d'énergie qui rend l'autonomie de ces appareils très limitée. Pour accroître l'autonomie des systèmes embarqués tels que les téléphones ou les ordinateurs portables, de nombreuses recherches sur la conception de circuits intégrés ont été réalisées en vue de réduire la consommation d'énergie. Ces travaux de thèse ont pour but de proposer au concepteur des moyens de concevoir des circuits intégrés numériques à faible consommation d'énergie. Trois étapes importantes vers la réduction de la consommation d'énergie ont été proposées. L'utilisation de la logique asynchrone représente le premier pas vers la réduction de la consommation d'énergie. En effet, de nombreux travaux réalisés ces dernières années ont montré que les circuits asynchrones présentent la propriété intrinsèque de consommer moins d'énergie que les circuits implémentés en logique synchrone. Le second pas important est d'offrir au concepteur des outils lui permettant d'obtenir des informations sur l'activité et la consommation d'énergie du circuit lors de la conception de celui-ci. Nous avons spécifié un estimateur d'activité et un estimateur de la consommation d'énergie qui permettent au concepteur de collecter des informations pertinentes sur la répartition de l'activité et de la consommation d'énergie d'un circuit lors d'une simulation donnée. L'avantage de ces estimateurs est que les informations sont obtenues très tôt dans le flot de conception, au niveau de la spécification CHP d'un circuit. Enfin, des techniques d'optimisation sont proposées pour réduire la consommation d'énergie des circuits. Certaines techniques sont utiles à l'outil de synthèse (pour la synthèse automatique de circuits à faible consommation d'énergie) et d'autres sont utiles au concepteur pour le guider dans ses choix d'implémentation. La méthodologie d'estimation et les techniques d'optimisation de la consommation d'énergie ont été appliquées à la réalisation d'un mini processeur, les résultats ont montré une réduction de la consommation d'énergie de 24%.

**TITLE LOW ENERGY ASYNCHRONOUS DESIGN METHODOLOGY:
APPLICATION TO A MIPS MICROPROCESSOR**

ABSTRACT

The steady evolution of mobile device complexity comes with an increase of the energy consumption which dramatically limits the autonomy of these devices. To increase the autonomy of embedded systems such as portable phones or computers, several researches on integrated circuit design have been performed in order to reduce the energy consumption. This Phd work aims at proposing to the designer the means to design low energy digital integrated circuits. Three important steps have been proposed in this work to reduce the energy consumption of circuits. The first step is to use asynchronous logic to design circuit. Indeed, several works achieved these last years have shown that asynchronous logic can significantly reduce the energy consumption of circuits. It is an intrinsic property of asynchronous logic. The second step is to offer tools that allow the designer to get information on the activity and the energy consumption of a circuit during the design flow. An activity estimator and an energy estimator have been specified that allow the designer to get relevant information on the distribution of the activity and the energy within a circuit during a specific simulation. The inherent advantage of these estimators is to provide information on the activity and the energy very early in the design flow, at the CHP specification. Finally, optimisation techniques have been defined to reduce the energy consumption of circuits. Some techniques can be integrated in the synthesis tool (for automatic low energy-oriented synthesis) and other techniques can be useful to guide the designer in his implementation choices. The estimation methodology and the energy consumption optimisation techniques have been applied to the implementation of a mini processor. The results have shown an energy consumption reduction of 24%.

INTITULE ET ADRESSE DU LABORATOIRE

Laboratoire TIMA, 46 avenue Félix Viallet, 38031 Grenoble Cedex, France

ISBN : 2-84813-040-7 (broché)

ISBNE : 2-84813-041-5 (électronique)