ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

COMME EXIGENCE PARTIELLE
À L'OBTENTION DE LA
MAÎTRISE EN GÉNIE ÉLECTRIQUE
M.Sc.A.

PAR
Louis-Charles TRUDEAU

CONCEPTION D'UNE MÉMOIRE CACHE L1 ASYNCHRONE POUR UN PROCESSEUR ARM

MONTRÉAL, LE 15 DÉCEMBRE 2015

PRÉSENTATION DU JURY

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE:

M. François Gagnon, directeur de mémoire
Département de génie électrique à l'École de technologie supérieure

M. Ghyslain Gagnon, codirecteur
Département de génie électrique à l'École de technologie supérieure

M. Claude Thibeault, président du jury
Département de génie électrique à l'École de technologie supérieure

M. Jean Belzile, membre du jury
Département de génie électrique à l'École de technologie supérieure

M. Thomas Jefferson Awad, examinateur externe
Octasic Semiconductors Inc

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 19 NOVEMBRE 2015

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

Tout d'abord, je tiens à remercier mon directeur François Gagnon et mon codirecteur Ghyslain Gagnon de m'avoir épaulé pendant tout mon parcours académique. Du baccalauréat jusqu'à la fin de ma maîtrise, vous m'avez inculqué une démarche et une rigueur intellectuelle qui m'ont constamment permis de me surpasser dans ce que j'ai entrepris. Merci infiniment pour votre patience et votre écoute, car les innombrables fois où j'ai « squatté » votre bureau m'ont aidé à remettre aujourd'hui un travail dont je suis fier.

J'aimerais ensuite remercier l'équipe incroyablement talentueuse qui travaille chez Octasic ainsi que les nombreux collaborateurs au sein du projet AnARM. Un grand merci à Tom Awad, Pascal Gervais, Jean-Michel Leclerc, Doug Morrissey, Sébastien Renaud, François Dominique Richardson et Claude Thibeault, pour ne nommer que ceux-ci. Je garde d'excellents souvenirs de tous ces *Lunch & Learn* au Frites Alors où, immergés dans les « détails d'implémentation » du ARM asynchrone, nous mangions une bonne poutine graisseuse. J'aimerais tout particulièrement remercier Jean-Michel et Tom pour les nombreuses explications et discussions portant sur la programmation, le design de puces et l'architecture des processeurs. Être entouré d'autant de professionnels hautement qualifiés (lire : gourous) m'a permis de me développer rapidement sur le plan technique et j'en suis réellement reconnaissant.

Finalement, j'aimerais remercier ma famille et mes amis-es de m'avoir insufflé le courage pour commencer cette maîtrise, puis la motivation pour la terminer. Merci à Jean-Marc, Hélène et Véro pour votre immuable soutien et vos encouragements. Merci à toutes et tous mes amis-es qui m'ont permis de décrocher de mon projet afin de profiter des petits bonheurs de la vie : Têtu, Julien, Karine & Alex, Mireille & Pambrun, Marky, Anaïs, Biz, Chuck, Éric, Seb & Francine, Guillaume & Amélie, Val & Alex, Patrice, et j'en oublie certainement ! Un merci tout particulier à Vanessa, mon amour, qui m'encourage et me supporte depuis les tout débuts ; on est passé à travers !

Je vous aime.

CONCEPTION D'UNE MÉMOIRE CACHE L1 ASYNCHRONE POUR UN PROCESSEUR ARM

Louis-Charles TRUDEAU

RÉSUMÉ

Les processeurs conçus par Octasic séparent actuellement le centre de traitement du processeur (CPU) asynchrone des mémoires d'instructions et de données synchrones. Le CPU est scindé en plusieurs unités d'exécutions (XUs) qui sont synchronisées par un système de jetons, qui utilise un protocole de communication basé sur les transitions. Un jeton est associé à chaque ressource du processeur (mémoire d'instructions et de données, fichier de registre, branchement conditionnel, etc.) et ceux-ci sont partagés au sein des XUs. Lorsqu'on accède la ressource, le jeton est utilisé puis relâché en fonction d'un délai proportionnel au temps de propagation de la logique combinatoire de la ressource. Ce type d'architecture asynchrone permet de réduire de moitié l'énergie consommée lors de l'exécution d'une tâche, comparativement à un processeur équivalent synchrone.

Dans le cadre de ce projet, le but est d'améliorer l'accès à la mémoire L1, qui impose présentement une pénalité de synchronisation de 2 cycles d'horloge lors de la lecture d'une instruction ou d'une donnée. Le premier objectif consiste donc à supprimer cette latence due à la synchronisation. D'une autre part, le réseau de distribution de l'horloge synchrone est responsable d'une portion considérable de l'énergie consommée par le processeur. Le deuxième objectif est donc d'améliorer l'efficacité énergétique de la cache en implémentant une architecture asynchrone, tout en réduisant la complexité du pipeline. De plus, un des objectifs complémentaires est d'augmenter le débit du transfert de données entre les niveaux L1 et L2 de la mémoire.

Ce mémoire introduit un type de pipeline asynchrone novateur développé pour une cache d'instruction L1. Ce pipeline asynchrone *auto-cadencé* intègre certaines caractéristiques de l'architecture basée sur le partage par les jetons d'Octasic et des éléments Click. Des simulations et une analyse temporelle post-placement ont été réalisées et basées sur une librairie CMOS 28nm fournie par CMC Microsystems. L'analyse a démontré que l'efficacité énergétique de la cache d'instruction a été améliorée d'au moins 21%, tout en réduisant le temps d'accès moyen à la mémoire de 25%. Les résultats colligés révèlent que la cache conçue est particulièrement efficace lorsqu'on tient compte des procédures de purge du pipeline d'exécution du CPU, en réduisant jusqu'à 68% de la consommation énergétique dynamique. La cache L1 d'instruction asynchrone développée dans le cadre de cette maîtrise a fait l'objet d'une publication à la 21ème conférence internationale IEEE International Symposium on Asynchronous Circuits and Systems (IEEE ASYNC 2015).

Mot-clés : mémoire cache L1, pipeline asynchrone, architecture processeur

DESIGN OF AN ASYNCHRONOUS L1 CACHE FOR AN ARM PROCESSOR

Louis-Charles TRUDEAU

ABSTRACT

As part of a global research program aimed at adapting Octasic's asynchronous architecture to the implementation of general purpose processors, this work targets the design of a level one (L1) cache for ARM-like processors. The overall goal is to obtain processors that are efficient in terms of energy and size while remaining competitive in terms of computing power. Octasic current processors separate the asynchronous CPU core from the synchronous L1 instruction and data caches. The CPU core is segmented in multiple execution units (XUs) that are synchronized by token rings that are based on transition signaling. A token is assigned to each resource (Instruction Fetch, Register Read/Write, Jump/Conditionnal Branch, Data Memory, etc.) and shared among the XUs. Tokens are used and released according to a variable delay that matches the resource's combinational logic. This asynchronous architecture reduces energy consumption by half for the same computing power when compared to its synchronous counterparts.

This work focuses on improving the memory access. Currently, the asynchronous CPU core accesses a synchronous L1 cache. Fetching an instruction or accessing data imposes a 2-cycle synchronization penalty, thus resulting in a performance degradation. The primary objective of this work is to mitigate this latency. Also, the synchronous cache clock network still accounts for a substantial portion of the processor's energy consumption, even though efficient clock-gating strategies are used. The second objective is to increase the cache energy efficiency by removing the substantial clock tree while reducing the cache pipeline complexity. Furthermore, another objective is to increase the data transfer speed between the L1 and L2 memory by eliminating the need to share a global clock network.

This thesis introduces a new self-timed pipeline developed for an L1 instruction cache. The self-timed pipeline integrates features from Octasic's token-based architecture and Click elements. Simulation and post-layout timing were performed using a 28nm bulk process. Analysis have shown that energy efficiency can be improved by over 21% while reducing memory access time by more than 25% in average. The gathered results have also shown that the designed cache is specifically efficient when the processor's execution pipeline flush is taken into account, thus far reducing dynamic power consumption by 68%. The asynchronous L1 cache designed during this Master's degree has been published at the 21st IEEE International Symposium on Asynchronous Circuits and Systems (IEEE ASYNC 2015).

Keywords: L1 cache memory, self-timed pipeline, asynchronous architecture

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

CPU	Unité centrale de traitement « Central Processing Unit »
XU	Unité d'exécution « eXecution Unit »
L1/L2/L3	Niveau 1-2-3 « Level 1-2-3 »
WL	« Word Line »
BL	« Bit Line »
CMOS	« Complementary Metal-Oxide Semiconductor »
VLSI	« Very Large Scale Integration »
DSP	Processeur de signal numérique « Digital Signal Processor »
ARM	« Advanced/Acorn RISC Machine »
ISA	« Instruction Set Architecture »
CISC	« Complex Instruction Set Computer »
RISC	« Reduced Instruction Set Computer »
MPU	« Memory Protection Unit »
MMU	« Memory Management Unit »
CAM	« Content-Adressable Memory »
RAM	« Random Access Memory »
DRAM	« Dynamic Random Access Memory »
SRAM	« Static Random Access Memory »
CAO	Conception Assistée par Ordinateur
CPI	Cycle(s) par instruction
CMPI	Cycle(s) mémoire par instruction
MIPS	Million d'Instructions Par Seconde
MRU	« Most Recently Used »

LRU	« Least Recently Used »
SOI	Silicium sur isolant « Silicon-On-Insulator »
DVFS	Ajustement dynamique de la tension et/ou fréquence. « Dynamic Voltage-Frequency Scaling »
DFF	Bascule D « D-latch Flip-Flop »
RAZ	Remise à zéro
pMOS	« P-channel Metal-Oxide-Semiconductor »
nMOS	« N-channel Metal-Oxide-Semiconductor »
QDI	« Quasi-Delay Insensitive »
ILP	Parallélisme au niveau des instructions « Instruction-Level Parallelism »
PC	Compteur du programme « Program Counter »
IF	Génération d'instruction « Instruction Fetch »
ID	Décodage d'instruction « Instruction Decode »
MEM	Accès mémoire
EX	Exécution
RR	Lecture des registres « Register Read »
WB	Écriture des registes « Write-Back »
BR	Condition de branchement
NoC	Réseau-sur-puce « Network-on-Chip »
PCBP	« Program Counter Branch Predictor »
LFSR	Registre à décalage à rétroaction linéaire « Linear Feedback Shift Register »
STA	Analyse temporelle statique « Static Timing Analysis »
DFT	Conception pour le test « Design For Test »
XOR	OU-Exclusif
XNOR	NON-OU-Exclusif

FIFO	File « First-In First-Out »
WPTR	Pointeur d'écriture
RPTR	Pointeur de lecture
VHDL	« Very High Speed Integrated Circuit Hardware Description Language »
RTL	« Register-Transfer Level »
TCL	« Tool Command Language »
OS	Système d'opération

LISTE DES SYMBOLES ET UNITÉS DE MESURE

ϕ	Phase de l'horloge
μm	micromètre
nm	nanomètre
MHz	MegaHertz
GHz	GigaHertz
Go	Gigaoctet
Mo	Megaoctet
ko	kilooctet
RC	Charge résistive-capacitive
I_{fuite}	Courant de fuite au sein d'un transistor
V_{DD}	Tension d'alimentation du drain d'un transistor
$T_{Horloge}$	Période de l'horloge globale du pipeline
T_{setup}	Temps de stabilisation minimum requis à l'entrée D d'un registre mémoire DFF de l'étage subséquent
T_{hold}	Temps de maintien minimum requis à l'entrée D d'un registre mémoire DFF de l'étage subséquent
T_{ccq}	Délai de contamination minimal pour faire passer une donnée de l'entrée D à la sortie Q d'un registre mémoire DFF suite à un front montant de l'horloge
T_{pcq}	Délai de propagation maximal pour faire passer une donnée de l'entrée D à la sortie Q d'un registre mémoire DFF suite à un front montant de l'horloge
T_{cd}	Délai de contamination minimal pour que la logique combinatoire d'un étage se stabilise
T_{pd}	Délai de propagation maximal pour que la logique combinatoire d'un étage se stabilise

N_{L2}	Délai d'accès à la cache L2 en nombre de cycle
ΔT	Différentiel de température
ΔE	Différentiel d'énergie
μJ	microjoule
ps	picoseconde

INTRODUCTION

Depuis 2013, une équipe de recherche de l'École de technologie supérieure et de l'École Polytechnique de Montréal travaille conjointement avec la compagnie Octasic afin de développer un processeur ARM novateur. La grande particularité de ce projet est que la puce développée intègre une architecture processeur asynchrone, qui a auparavant été développée par Octasic pour ses processeurs de traitement de signal (DSP). Cette architecture éprouvée permet aux différents circuits sur la puce de fonctionner sans être cadencés par une horloge globale, ce qui permet de bénéficier d'une efficacité énergétique accrue. Un autre avantage de l'architecture asynchrone est que les circuits de la puce peuvent fonctionner à leur fréquence naturelle d'opération, sans être contraints par une période d'horloge basée sur le circuit le plus lent.

Toutefois, l'implémentation d'une telle architecture n'est pas sans difficulté et exige certains compromis techniques. En effet, les programmes peuvent être exécutés de manière asynchrone par le processeur, mais une synchronisation doit être faite à un certain moment afin de garantir la cohérence des instructions et des données nécessaires au programme. Ainsi, les accès à la mémoire doivent être synchronisés à un niveau ou à un autre dans le processeur. Préalablement, l'architecture des DSP d'Octasic séparait le coeur du processeur (CPU) asynchrone du premier niveau (L1) de mémoire. Ainsi, une synchronisation entre les deux domaines d'horloge devait s'effectuer à cette interface afin de pouvoir accéder aux instructions et aux données. La problématique est la suivante. Dans un premier temps, cette synchronisation impose une pénalité de 2 cycles d'horloge. L'accès à la mémoire devient alors un des goulots d'étranglement du processeur. Dans un deuxième temps, la mémoire synchrone ne possède pas une architecture efficace d'un point de vue énergétique comparativement au processeur.

Dans le cadre de ce projet, le but est d'améliorer l'accès au premier niveau de mémoire. Ce mémoire présente donc la conception d'une mémoire cache L1 d'instructions asynchrone basée sur un nouveau type de pipeline auto-cadencé (« self-timed »). L'hypothèse, qui a été vérifiée au cours de ce projet, est que l'intégration d'une cache asynchrone permet dans un premier temps de mitiger la pénalité de synchronisation entre le processeur et la mémoire. Dans un second temps, cela permet de réduire le temps d'accès moyen à la mémoire et d'améliorer l'ef-

ficacité énergétique de la cache, tout en réduisant la surface occupée par celle-ci sur la puce. Pour soutenir cette hypothèse, la mémoire cache asynchrone développée doit être basée sur les mêmes spécifications que la cache synchrone actuelle du processeur ARM. De plus, des simulations doivent permettre de comparer objectivement le comportement des caches synchrone et asynchrone pour affirmer qu'il y a bien un gain en performance.

Le chapitre 1 présente une revue de littérature portant sur l'architecture des mémoires caches. Les notions de base des processeurs et des mémoires sont d'abord introduites. Le fonctionnement et l'architecture interne des mémoires caches sont ensuite présentés. Enfin, une revue de littérature exhaustive des optimisations en terme de performance et d'efficacité énergétique est présentée. Ce chapitre présente de nombreuses pistes de solution permettant de comprendre les limitations de la cache actuelle et d'améliorer les performances de celle-ci.

Le chapitre 2 présente une revue de littérature portant sur les pipelines asynchrones. Dans ce chapitre, les concepts fondamentaux des pipelines sont d'abord introduits. L'architecture d'un pipeline synchrone est expliquée et de celle-ci est dérivée l'architecture asynchrone. Les éléments de base constituant un pipeline asynchrone sont ensuite présentés. Finalement, une revue de littérature des principaux pipelines asynchrones est faite. Ce chapitre permet de mettre en contexte le pipeline de la cache synchrone actuelle. À partir de l'architecture synchrone, différentes alternatives sont suggérées afin de modifier cette dernière en un pipeline asynchrone efficace. Les éléments Click, qui sont présentés dans ce chapitre, représentent une partie fondamentale du pipeline asynchrone développé dans le cadre de ce projet.

Le chapitre 3 introduit l'architecture actuelle du processeur ARM ainsi que sa mémoire synchrone. Ce chapitre présente d'abord l'architecture asynchrone des processeurs d'Octasic et décrit comment les différentes tâches du processeur sont partitionnées en fonction des unités d'exécution (XU). Puis, le paradigme de synchronisation et le partage des ressources avec les jetons sont expliqués. La définition de l'architecture et du fonctionnement de la mémoire cache L1 synchrone est ensuite réalisée. En dernier lieu, le pipeline de la cache L1 d'instruction synchrone est minutieusement expliqué. L'exploration de l'architecture asynchrone développée

par Octasic a permis d'obtenir une compréhension approfondie du système de jetons et du concept de partage de ressources. L'intégration de ce modèle de synchronisation représente aussi une partie essentielle du pipeline asynchrone développé dans le cadre de cette maîtrise.

Au chapitre 4, une nouvelle cache asynchrone est proposée pour le processeur ARM. La structure et le fonctionnement du nouveau pipeline asynchrone sont d'abord introduits. Puis, l'intégration et le fonctionnement du pipeline asynchrone au sein de la cache d'instruction sont présentés. Chaque interface entre la cache et son environnement est alors décrite. Ce chapitre englobe l'entièreté de la conception de la nouvelle cache d'instruction et de son pipeline asynchrone spécifiquement développé à cet effet. Une quantité importante de notions élémentaires a été accumulée afin de conceptualiser et réaliser cette cache asynchrone. Une attention particulière a été portée afin de valider l'intégration de cette dernière dans le processeur ARM asynchrone d'Octasic.

Différents bancs de test ont ensuite été développés afin de valider la fonctionnalité et d'analyser la performance de la cache asynchrone proposée. L'analyse des résultats obtenus est réalisée au chapitre 5. Ce chapitre présente dans un premier temps le flot de conception ainsi que l'environnement de simulation utilisé dans le cadre de ce projet. Les bancs de tests ainsi que leurs limitations respectives sont alors exposées. Finalement, les résultats obtenus lors des bancs de tests individuels et comparatifs sont présentés et une analyse globale est réalisée. Cette analyse permet de caractériser les performances de la cache développée en la comparant avec la cache synchrone actuelle. On indique d'abord que la taille physique du pipeline asynchrone a pu être réduite par rapport au pipeline synchrone. Cette étude approfondie révèle ensuite les gains en performance en terme d'efficacité énergétique et en vitesse d'exécution de la cache asynchrone par rapport à son équivalent synchrone. De plus, il a été possible d'augmenter considérablement le débit d'échange de données entre les niveaux L1 et L2 de la mémoire d'instruction, grâce aux interfaces conçues spécifiquement pour la cache asynchrone.

Contributions scientifiques

Les contributions scientifiques de ce travail sont :

- une revue de littérature portant sur les mémoires caches et sur les pipelines asynchrones présentée dans le cadre du cours lectures dirigées (MTR-871) ;
- la proposition et le développement d'un pipeline asynchrone comportant une synchronisation hybride par les éléments Click et le système de jetons ;
- la proposition d'une cache asynchrone à faible latence et à efficacité d'énergie élevée intégrant le nouveau pipeline asynchrone développé. La cache asynchrone conçue dans le cadre de ce travail a fait l'objet d'une publication (Trudeau *et al.*, 2015) et a été présentée à la conférence ASYNC 2015 à Mountain View en Californie le 5 mai 2015. La cache asynchrone développée pour le processeur ARM asynchrone permet dans un premier temps de diminuer la latence moyenne d'accès à la mémoire d'un minimum de 21%, par rapport à l'utilisation de la cache synchrone. Dans un deuxième temps, l'analyse réalisée révèle que la nouvelle cache asynchrone développée consomme de 25 à 68% moins d'énergie dynamique que son équivalent synchrone, pour un mode d'opération équivalent.

CHAPITRE 1

MÉMOIRES CACHES

1.1 Introduction

Un ordinateur dépend de plusieurs types de mémoires différentiables par leur organisation, taille, vitesse, densité, technologie et coût. L'inhérente hiérarchie mémoire d'un ordinateur prend avantage des caractéristiques de chaque type de mémoire afin d'être optimal (Jacob *et al.*, 2010). Cette mémoire hiérarchique est accessible par le ou les processeurs sur la puce (registres mémoires, caches), sur la carte mère (mémoire principale DRAM), sur des périphériques externes (disques durs, CD, DVD), sur le nuage et des périphériques de stockage (bandes magnétiques).

Cette hiérarchie s'est développée avec les années pour pallier à un phénomène aujourd'hui très apparent. La cellule mémoire est devenue plus lente que les cellules logiques utilisées pour traiter l'information qui y est stockée. L'accès mémoire est alors le goulot d'étranglement du processeur (Jacob *et al.*, 2010), (Stallings, 2010), (Hennessy et Patterson, 2011). C'est afin de pallier à cette problématique que les banques de registres et les mémoires caches ont été conçues.

Une mémoire cache [1], qui agit de manière analogue à la mémoire à court terme, permet de faire croire au processeur qu'il peut accéder un information à tous les cycles d'horloge. Avec le temps, différentes stratégies ont été développées pour améliorer les performances des mémoires caches et se rapprocher de la vitesse d'opération du centre de traitement principal, communément appelé CPU. Depuis, tous les processeurs modernes performants implémentent différents niveaux de caches pour optimiser ses opérations.

Dans ce chapitre, les notions de base portant sur les processeurs et la mémoire sont d'abord introduites. Ensuite, l'architecture générale des mémoires caches est présentée. L'organisation

1. Le terme « cache » est employé au féminin afin d'assurer une certaine homogénéité lorsqu'une référence est faite à *une* mémoire cache ou *un* cache, facilitant ainsi la lecture du mémoire.

logique et les mécanismes de gestion interne sont alors expliqués. Finalement, une revue de littérature portant sur les optimisations de performance et d'efficacité énergétique est présentée.

1.2 Concepts fondamentaux

L'accès à la mémoire d'instructions et de données est décrit ainsi que la notion de hiérarchie mémoire. Ce sont les contraintes inhérentes associées à cette hiérarchie qui justifie l'intégration de mémoires intermédiaires, appelées mémoires cache, dans les processeurs modernes. Finalement, le concept de localité de référence est présenté. Ce concept fondamental est à la base du fonctionnement des mémoires caches.

1.2.1 Notions d'architecture des processeurs RISC

La fonction principale d'un processeur est l'exécution d'un programme qui est encodé sous forme d'instructions dans la mémoire (Patterson et Hennessy, 2009). L'architecture d'un processeur est conçue en fonction d'un ensemble d'instructions, dénotée « Instruction Set Architecture » (ISA). Une des ISA les plus répandus est le x86 d'Intel, qui implémente un ensemble d'instructions de type « Complex Instruction Set Computer » (CISC). Cependant, les architectures processeur ARM implémentent plutôt un ensemble d'instructions de type « Reduced Instruction Set Computer » (RISC). Une attention particulière est donc portée pour les architectures RISC dans ce mémoire, puisque l'objectif est de concevoir une mémoire cache pour un processeur ARM.

Les architectures de processeur RISC se distinguent par les caractéristiques suivantes. Tout d'abord, comme son acronyme l'indique, ce type de processeur utilise un ensemble d'instructions simples et limitées. L'ISA implémente un format instructions et des modes d'adressage simples. Ensuite, l'architecture du processeur est basée sur de larges registres mémoires. Cette configuration facilite l'exécution d'opérations simples sur les données à partir d'un registre à un autre. L'exécution des opérations registre à registre ne prend que quelques cycles d'horloge. Contrairement aux architectures CISC, une instruction ne génère pas (ou très rarement) un mi-

croprogramme devant être exécuté sur plusieurs cycles. Finalement, un effort est fait au niveau du compilateur afin d'optimiser l'usage des registres mémoires.

L'exécution au sein d'un processeur RISC peut être résumée en quelques étapes, telles qu'illustrées à la figure 1.1.

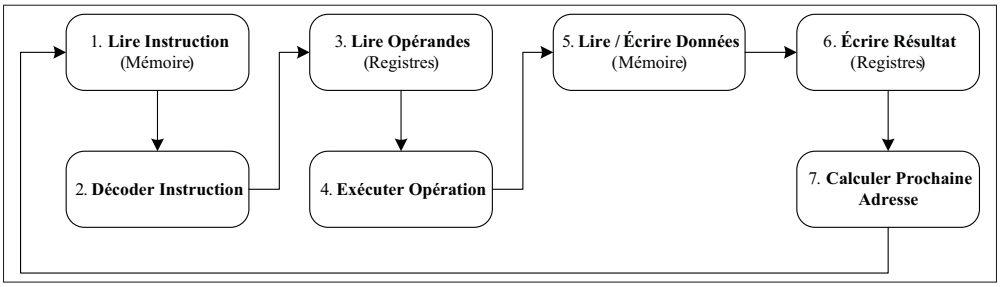


Figure 1.1 Fonctionnement général d'un processeur RISC
Adaptée de Stallings (2010)

La première étape consiste à aller chercher une instruction dans la mémoire à l'adresse indiquée par le programme. Cette étape est communément appelée « Instruction Fetch ». La deuxième étape consiste à décoder cette instruction pour déterminer quelle opération exécuter sur quelles opérandes. La troisième étape consiste à aller lire les opérandes à l'adresse indiquée (ou pouvant être calculée) par l'instruction dans les registres mémoires. La quatrième étape consiste à exécuter l'opération spécifiée par l'instruction sur les opérandes. Lors de la cinquième étape, le processeur accède la mémoire de données pour lire ou écrire à partir d'un registre. Cette étape est généralement issue d'une instruction de chargement en registre (« Load ») ou de stockage en mémoire (« Store »). Il est à noter que cette étape est située après l'étape d'exécution car le calcul de l'adresse en mémoire est fait par ce dernier. La sixième étape consiste en l'écriture du résultat de l'opération fait à l'étape 4 dans un registre mémoire. Cette étape est habituellement dénommée « Write Back ». La septième et dernière étape consiste à calculer la prochaine adresse de l'instruction du programme.

1.2.2 L'architecture RISC des processeurs ARM

Les architectures RISC varient énormément d'un processeur à un autre, ainsi que dans leur façon d'implémenter ces fonctionnalités. ARM, par exemple, est maintenant à sa huitième version de son ensemble d'instructions, dont les trois premières sont aujourd'hui désuètes (ARM, 2010a). Ce type de processeurs RISC domine le marché des systèmes embarqués par son architecture simple et performante (Stallings, 2010). Les versions les plus récentes sont regroupées sous trois types de profils, soit Application (A), Temps Réel (R) et eMbarqué (M).

Le profil A est le plus complet et le plus performant des trois. Il comprend une MMU (« Memory Management Unit ») qui gère les accès dans un contexte multiprocesseur ou avec système d'exploitation. Son architecture inclut plusieurs accélérations et extensions, telles que *TrustZone*, qui permet de protéger les données sensibles. Les jeux d'instructions ARM 32-bit et Thumb 16-bit sont supportés. On retrouve parmi ce type de profil le Cortex-A8 et Cortex-A15, pour ne nommer que ceux-ci. L'architecture du Cortex-A8 est représentée à la figure 1.2.

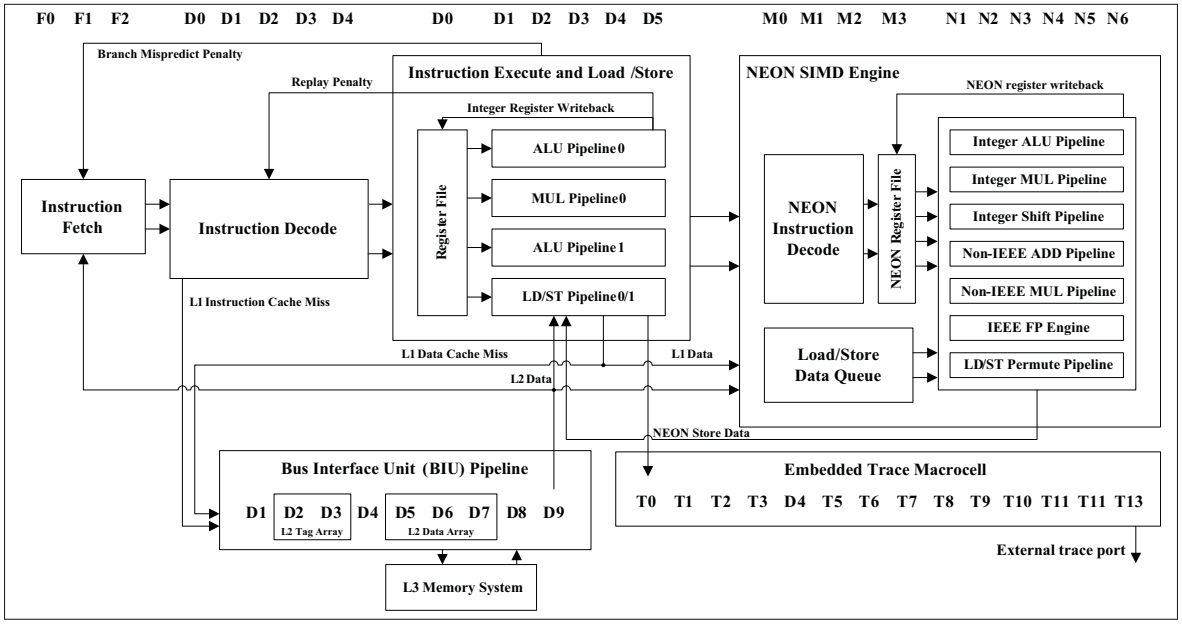


Figure 1.2 Processeur ARM Cortex-A8
Adaptée de ARM (2010b)

Le profil R incorpore plusieurs éléments du profil A, tout en misant sur la prévisibilité d'exécution et la faible latence des systèmes en temps réel. Ainsi, ce type de processeur incorpore une MPU (« Memory Protection Unit) » au lieu d'une MMU, ce qui permet de restreindre l'accès des applications à certaines parties de la mémoire seulement. Ce profil propose un meilleur équilibre entre performance et efficacité énergétique pour les systèmes embarqués. Le Cortex-R4 fait partie de cette famille de processeurs. Les jeux d'instructions ARM et Thumb sont aussi supportés.

Le profil M est orienté microcontrôleur, se basant sur une architecture simple et à faible puissance. Celui-ci implémente une architecture ARM réduite conçue pour le traitement rapide des interruptions. Ceci permet au processeur de garder un comportement déterministe et prévisible, idéal pour les systèmes embarqués. On retrouve sur le marché le Cortex-M3, entre autres. Les jeux d'instructions supportés sont ARM et une partie de Thumb.

1.2.3 Accès aux mémoires d'instructions et de données

Il existe diverses contraintes reliées à l'utilisation de la mémoire pour stocker les instructions et les données du processeur. Richard L. Sites, un des pionniers de l'architecture des processeurs ayant travaillé au développement du DEC Alpha, décrit avec exactitude en 1996 les tendances au niveau de la conception de processeurs.

> Across the industry, today's chips are largely able to execute code faster than we can feed them with instructions and data. There are no longer performance bottlenecks in the floating-point multiplier or in having only a single integer unit. The real design action is in memory subsystems – caches, buses, bandwidth, and latency. (Jacob *et al.*, 2010, p. xxxi)

La figure 1.3 illustre ces contraintes en fonction de la hiérarchie mémoire d'un processeur.

La principale problématique est que la vitesse d'opération d'une mémoire est inversement proportionnelle à sa taille et à sa distance du processeur. Plus une mémoire est proche du processeur, moins elle est dense, mais plus elle est rapide. Le problème est l'inverse pour une mémoire qui est distante du processeur. De plus, le coût par bit de mémoire augmente très rapidement

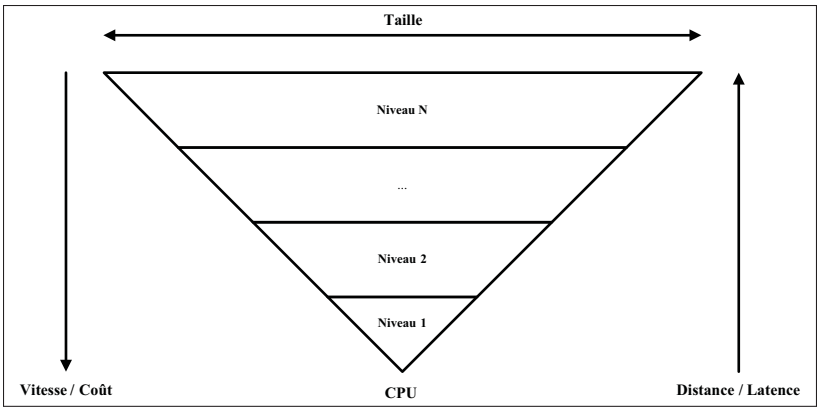


Figure 1.3 Hiérarchie mémoire d'un processeur
Adaptée de Stallings (2010)

plus l'intégration est faite proche du processeur. Un compromis doit donc être fait entre la rapidité de mémoire et la taille de celle-ci pour éviter que la mémoire soit le goulot d'étranglement des processeurs. Pour trouver un équilibre acceptable, plusieurs niveaux intermédiaires de mémoire sont intégrés dans les processeurs modernes. Les niveaux de mémoire inférieurs près du CPU, communément appelés mémoires caches, exploitent le concept de localité de référence pour gérer l'information quelles contiennent. D'ailleurs, plusieurs niveaux de cache peuvent être implémentés, du premier niveau de cache $L1$ jusqu'au dernier niveau L_X.

Il existe deux façons d'entrevoir le stockage des instructions et des données en cache pour un processeur. Tout d'abord, il est possible d'utiliser une cache unifiée, où sont contenues les instructions et les données accédées par le processeur. Ce type de cache s'inspire de l'architecture von Neumann. Ensuite, il est possible d'implémenter deux caches séparées, soit une cache d'instruction et une cache de données. Ce type de cache s'inspire de l'architecture Harvard. Le principal avantage d'utiliser deux caches séparées est la possibilité de paralléliser les accès aux instructions et aux données sans avoir à utiliser port d'accès double (« dual-port »). Finalement, un processeur peut implémenter qu'un seul type de cache, soit celle d'instructions ou de données, ou l'omettre complètement.

1.2.4 Concept de localité de référence

Les mécanismes internes d'une mémoire cache sont basés sur le principe statistique de localité de l'information. La localité d'une information peut être définie principalement en fonction de deux aspects différents, soit de manière temporelle ou spatiale.

D'une part, le principe de localité temporelle repose sur la tendance des programmes à réutiliser les mêmes données plusieurs fois lors de l'exécution. Par exemple, un programme implémentant un filtre numérique fera constamment appel aux mêmes coefficients pour calculer la réponse de ce dernier. L'utilisation de ce principe est limitée en grande partie par la taille de la cache, puisque celle-ci ne représente qu'une infime partie de la mémoire.

D'une autre part, le principe de localité spatiale repose sur le fait que les programmeurs et compilateurs placent l'information sous forme de blocs contigus en mémoire. Ainsi, il est facile d'exploiter ce principe en transférant un bloc d'information plus large que la donnée demandée par le CPU vers la cache. La taille du bloc d'information à transférer est équivalente à la taille de la ligne de cache (« cache line »). La performance de la cache varie en fonction de la taille de la ligne de cache, selon des relations maintes fois analysées et décrites dans la littérature (Jouppi, 1993), (Inoue *et al.*, 2002), (Patterson et Hennessy, 2009). Cela est principalement dû au délai et à la puissance consommée lors du transfert de la mémoire supérieure à la cache.

1.3 Architecture d'une mémoire cache

Comme il l'a été mentionné précédemment (Jacob *et al.*, 2010), (ARM, 2010a), une cache est un bloc mémoire à haute vitesse localisé très proche du processeur. Une cache contient à la fois les données et les adresses permettant de localiser ces dernières, communément appelé étiquette cache, ou « cache tag ». Afin d'accélérer le processus de lecture et d'écriture en mémoire, une cache se base sur le principe statistique fondamental, soit la localité de référence.

Le fonctionnement d'une cache est habituellement transparent aux applications qui sont exécutées par le processeur. Les stratégies et les mécanismes utilisées permettent aux caches d'opé-

rer indépendamment d'un programme et de pouvoir définir quoi retenir dans leur mémoire. Les stratégies utilisées, telles que ceux gouvernant le placement de nouvelles données, l'accès à celles-ci et leur remplacement, déterminent l'efficacité de la cache.

De manière conceptuelle, il est possible de diviser la cache en trois composantes fondamentales. Premièrement, son organisation logique permet de déterminer comment les informations y seront stockées. Deuxièmement, les stratégies de gestion du contenu définissent si les informations doivent être placées ou non dans la cache. Troisièmement, les mécanismes de cohérence de la cache s'assurent que les données et les instructions reçues par le processeur sont à jour et consistantes avec les instances supérieures de mémoire.

L'organisation logique d'une mémoire cache peut être caractérisée selon son type d'adressage, l'unification de ses données, son associativité, sa taille et le type éléments mémoire intégré. Du point de vue du processeur, l'organisation logique d'une mémoire cache est transparente. Peu importe la configuration interne de la cache, l'adresse doit établir une correspondance directe entre la donnée demandée et celle qui existe à un des niveaux de la mémoire.

Cela étant dit, les performances d'une cache sont fonction de son optimisation selon le type de programmes exécutés par ce dernier. L'approche traditionnelle pour caractériser et optimiser une cache se base sur des programmes de référence, dénommés « benchmarks ». En faisant varier les paramètres définissant l'organisation de la cache et en exécutant différents types de programmes, il est possible d'obtenir des statistiques sur la performance de celle-ci. Des simulateurs de cache tel que *Cacti* ont été conçus pour automatiser ce travail d'exploration. Cette approche, davantage empirique que théorique, a été préconisée par Hennessey et Patterson (Hennessy et Patterson, 2011) à la fin des années 80.

1.3.1 Adressage mémoire

Au niveau de la cache, la plage d'adresse peut être représentée de deux manières différentes. D'une part, la plage d'adresse peut être divisée indépendamment entre chaque programme, ce qui correspond à un adressage physique. D'une autre part, la même plage d'adresse peut

virtuellement être utilisée par tous les programmes, ce qui correspond plutôt à un adressage virtuel. La plupart des processeurs à usage général utilisent l'adressage virtuel, pour sa simplicité d'utilisation avec un système d'opération.

L'adresse de l'information demandée par le processeur est donc le point d'accès à la cache. Tout d'abord, l'adresse permet de déterminer si la cache contient l'information demandée. Seulement une partie de l'adresse demandée est comparée avec les adresses des données contenues par la cache pour limiter le coût en énergie et en temps associé à cette recherche. Cette partie de l'adresse est appelée étiquette de cache, ou « cache tag », tel que représenté à la figure 1.4. L'organisation d'une cache est divisée en 2^S sous-ensembles, appelés « cache set », qui comprennent un nombre fixe de lignes de cache, ou « cache line ». Une ligne de cache est un bloc d'information contigu d'une taille définie pouvant être échangé entre deux niveaux de mémoire. L'information demandée par le processeur se situe au $(L-1)^{ème}$ octet de la ligne de cache.

Pour accéder à cette information, il doit y avoir une égalité entre l'étiquette de l'adresse demandée et les étiquettes contenues dans la cache. Lorsque c'est le cas, la requête provoque un « cache hit » et on peut accéder à la donnée à l'emplacement spécifié dans la mémoire cache. Dans le cas contraire, aucune des étiquettes contenues dans la cache ne correspond à l'étiquette de l'adresse demandée. La requête provoque alors un « cache miss ». La cache doit alors envoyer une requête à un niveau supérieur de mémoire pour qu'il lui transfère la donnée. La donnée demandée et son adresse sont alors copiées localement, avant d'être renvoyées au processeur.

1.3.2 Associativité

L'information provenant de la mémoire étant adressée sur les $[31-L]$ bits de l'étiquette peut seulement être placée dans un des 2^S sous-ensembles. Il existe un niveau d'associativité relié à chaque sous-ensemble, ce qui permet de limiter la contention au sein de la cache. L'associativité correspond au nombre de voies disponibles, communément appelées « ways », pouvant sto-

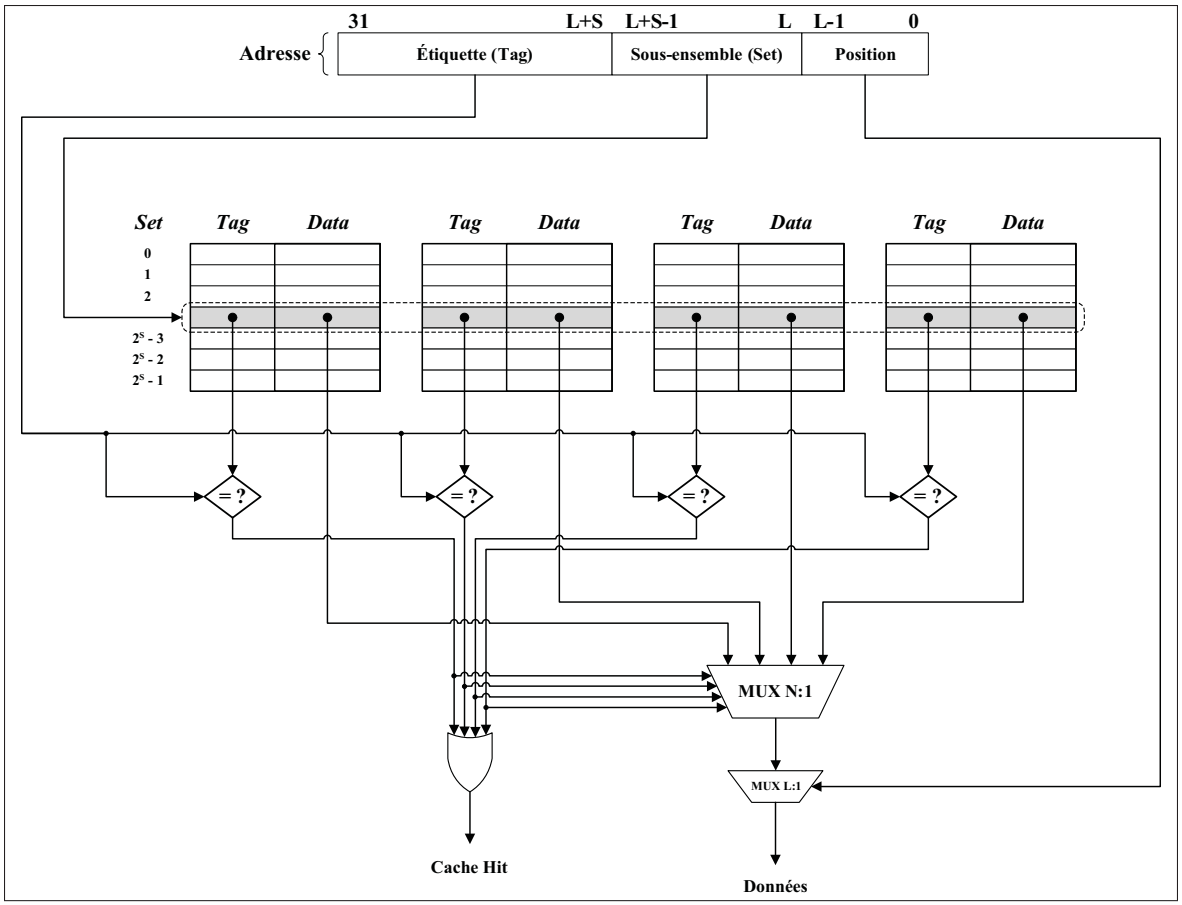


Figure 1.4 Organisation logique d'une mémoire cache et son adressage
Adaptée de Patterson et Hennessy (2009)

cker une même ligne de cache. Ainsi, deux informations distinctes adressées par la même étiquette peuvent être stockées sur deux voies différentes afin d'éviter un conflit d'adressage. Une faible associativité réduit la complexité de la recherche du *cache tag* et la puissance consommée. Cependant, les performances sont réduites lorsque les lignes de cache sont constamment remplacées. Ce problème de contention entraîne un nombre d'accès trop élevé aux niveaux supérieurs de mémoire.

Il existe deux types d'organisation logique implémentant chacun un niveau d'associativité contraire. Le premier type de cache est dénoté « direct-mapped » et n'a qu'une seule voie pour chaque ligne de cache par sous-ensemble. On compare donc qu'une seule fois l'étiquette de la cache pour déterminer si la donnée est présente dans la cache. Ce type de cache a un risque

de contention élevé. Le second type de cache est dénoté « fully-associative » et n'a qu'un seul sous-ensemble, soit la totalité de la taille de la cache. Le nombre de voies correspond au nombre de lignes de cache contenue par celle-ci. Chaque étiquette doit donc être comparée avec l'adresse demandée pour déterminer si la donnée est présente dans la cache. Ce type de cache est souvent basé sur une mémoire adressable par son contenu, ou « content-addressable memory » (CAM). Un compromis entre ces deux organisations logiques est la cache de type « set-associative ». Le nombre de d'emplacements possibles pour une information est limité à N voies. La comparaison de l'étiquette est donc faite N fois en parallèle, ce qui limite la consommation d'énergie et la contention, mais complexifie l'organisation logique. Ce type de cache demeure un bon compromis en terme de vitesse d'accès et de puissance consommée.

1.3.3 Taille de la cache

De manière générale, plus la taille d'une cache augmente, plus la probabilité d'y retrouver une donnée augmente. Ceci réduit le temps d'accès mémoire et améliore les performances, au prix d'une augmentation de la surface de silicium et de la consommation énergétique. Il est donc primordial de faire un choix éclairé en fonction du type de processeur conçu.

Il est possible de définir la taille d'une cache en fonction des trois paramètres, présentés à la figure 1.4. Premièrement, la longueur 2^L de la ligne de cache est exprimée en octets. Deuxièmement, l'associativité de la cache est définie comme étant le nombre N de voies possibles associées aux lignes de caches. Troisièmement, il existe 2^S sous-ensembles disponibles dans la cache pour stocker des lignes de caches.

En utilisant ces définitions, la taille de la cache est décrite comme étant :

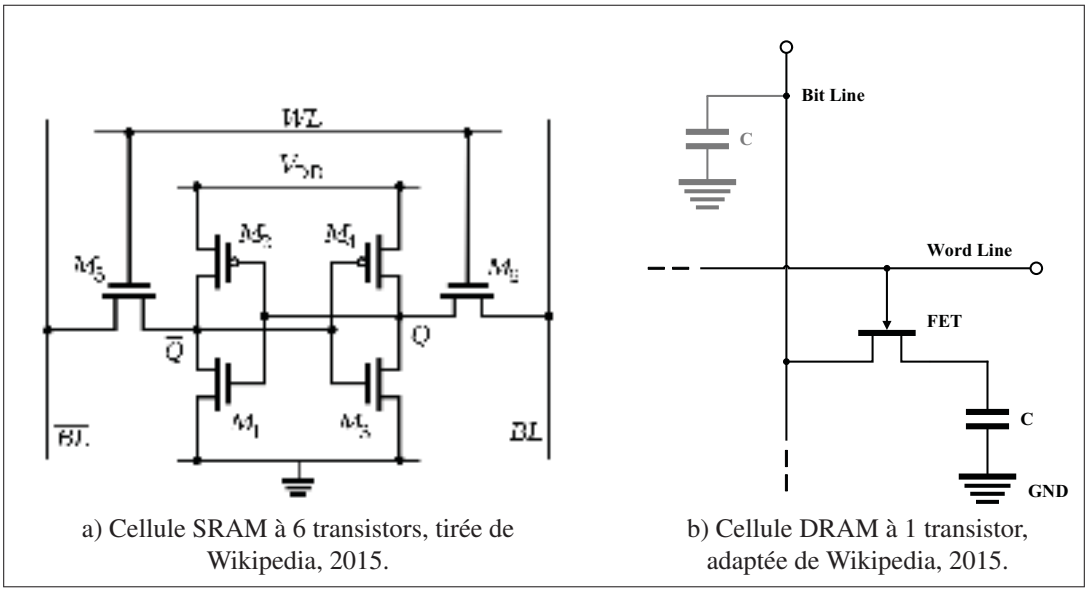
$$\text{Taille}\{\text{Cache}\} = N \times 2^S \times 2^L \text{octets} \tag{1.1}$$

Par exemple, les caches L1 d'instruction et de données du ARM Cortex-A8 peuvent stocker 128 lignes de cache d'une taille de 64 octets sur 4 voies. Ce type de cache possède donc un

espace mémoire total de 32 ko. Pour sa part, la cache L2 du ARM Cortex-A15 stocke 1024 lignes de cache d'une taille de 64 octets sur 16 voies. Cette cache possède alors une mémoire totale de 1 Mo.

1.3.4 Types de cellules mémoires

Le type de mémoire utilisé pour les mémoires caches sur puce est généralement basé sur la « Static Random Access Memory » (SRAM). Cependant, il existe de plus en plus d'architectures mémoires intégrant la « Dynamic Random Access Memory » (DRAM). Ces deux types de cellules mémoires sont illustrées à la figure 1.5.



a) Cellule SRAM à 6 transistors, tirée de Wikipedia, 2015.

b) Cellule DRAM à 1 transistor, adaptée de Wikipedia, 2015.

Figure 1.5 Cellules mémoires a) *Static-RAM* et b) *Dynamic-RAM*

Les cellules SRAM, tel que la version contenant 6 transistors à la figure 1.5a, sont relativement simples à utiliser. Les quatre transistors du centre M_1 à M_4 constitue un élément de mémoire, soit deux inverseurs en chaîne. Lors d'une écriture, la ligne contenant le mot adressée WL est mis à '1', activant la grille des transistors M_5 et M_6. Puis, les lignes du bit en écriture BL et son complément $\overline{BL}$ imposent une nouvelle valeur à la sortie et à l'entrée des inverseurs

assez longtemps pour modifier l'état de cette mémoire. Après une certaine période de temps, la nouvelle valeur est stabilisée et la mémoire peut être lue adéquatement.

Toutefois, il existe un regain d'intérêt envers les cellules DRAM pour diverses raisons, malgré leur complexité et performance moyennes. Tout d'abord, la densité d'intégration des cellules DRAM est supérieure comparée aux cellules SRAM. Comme il est illustré à la figure 1.5b un seul transistor et un condensateur sont nécessaires pour maintenir un bit d'information. Cet aspect devient de plus en plus important avec la miniaturisation constante des puces intégrées. Malgré le fait qu'une cellule DRAM doive utiliser un mécanisme de rafraîchissement pour conserver son information, celle-ci est demeurée aussi fiable que la SRAM Jacob *et al.* (2010). De plus, une cellule DRAM consomme significativement moins de puissance dynamique et statique qu'une cellule SRAM, qui nécessite en moyenne 6 transistors pour garder un bit d'information. Malgré tout, la SRAM demeure le type de cellule mémoire majoritairement intégré dans les logiciels de CAO et privilégiée par l'industrie. Cela est dû en partie car la SRAM est plus robuste que la DRAM et que leur taille peut facilement être adaptée au procédé de fabrication.

1.4 Gestion du contenu et de la cohérence d'une mémoire cache

La gestion de l'information contenue dans la cache est gérée par des stratégies pouvant être regroupées en trois fonctions. Premièrement, la stratégie d'allocation détermine quand et comment on transfert de l'information de la mémoire jusqu'à la cache. Deuxièmement, la stratégie de remplacement détermine où placer l'information dans la cache. Cette stratégie est basée sur le nombre de d'emplacements physiques possibles où l'on peut placer l'information, soit le nombre de voies. Troisièmement, les mécanismes de cohérence assurent que la cache fonctionne de manière rationnelle et que l'information quelle contient reflète celle des niveaux supérieurs de mémoire. Ces stratégies de gestion du contenu et de la cohérence sont essentielles au fonctionnement d'une cache. De plus, l'application de ces stratégies est faite en ligne ; c'est-à-dire qu'aucune action n'est requise par le compilateur ou le programmeur

pour affecter le comportement de la cache. Les mécanismes utilisés réagissent seulement en fonction du programme exécuté.

1.4.1 Stratégie d'allocation

Les politiques d'allocation des données déterminent quel type d'événement permet de transférer une nouvelle information d'une instance mémoire supérieure à la cache. Une stratégie d'allocation en lecture (« read-allocate ») permet d'allouer une nouvelle ligne de cache seulement lors d'un « cache miss » en lecture. Alternativement, la stratégie d'allocation en écriture (« write-allocate » ou « read-write-allocate ») alloue une nouvelle ligne de cache lors d'une opération en lecture ou en écriture. L'avantage d'une technique par rapport à l'autre dépend fortement du programme qui est exécuté par le processeur. Une politique d'allocation en lecture peut permettre d'éviter de transférer de l'information à la cache qui ne sera pas utilisée. Dans le cas contraire où les données seront réutilisées par la suite, la politique d'allocation en écriture permettra d'obtenir un « cache hit » immédiatement.

1.4.2 Stratégie de remplacement

Pour les mémoires caches de type « set-associative » ou « fully-associative », une stratégie de remplacement doit être implémenté. En effet, une des N voies doit être choisie lorsqu'une nouvelle ligne de cache doit être assignée pour une étiquette suite à un « cache miss ». Les stratégies les plus communes sont le remplacement aléatoire et le remplacement « round-robin ». La première stratégie nécessite qu'un générateur de nombre pseudo-aléatoire, tandis que la seconde n'utilise qu'un simple compteur. Encore une fois, les performances varient beaucoup en fonction des données traitées. Par exemple, une stratégie « round-robin » agit de façon plus déterministe tandis que l'approche pseudo-aléatoire maintient une performance moyenne constante. D'autres stratégies plus complexes, comme le « Most Recently Used » (MRU) et « Least Recently Used »(LRU), gardent des statistiques sur l'utilisation des lignes de caches en gardant un décompte de leur accès. Lorsqu'une ligne de cache doit être évincée, celle qui est la moins souvent utilisée est remplacée.

1.4.3 Mécanismes de cohérence

Une cache ne peut être utile que si on peut garantir que l'information qui s'y trouve est valable. Ainsi, des mécanismes doivent être implémentés afin de garantir que l'information est cohérente avec la mémoire principale, avec elle-même et avec les différentes entités qui peuvent y accéder.

En tout temps, la cache doit s'assurer que l'information qui s'y trouve est synchronisée d'une quelconque manière avec la mémoire principale. Une façon triviale de traiter cet aspect est de renvoyer immédiatement les lignes de cache modifiées par une écriture à toutes les instances supérieures de mémoire. Un tel mécanisme est dénommé « write-through » et permet de minimiser le temps où les informations sont désynchronisées. Ce mécanisme de cohérence comporte certains désavantages. En effet, l'envoi constant des lignes de cache modifiées augmente de façon drastique la latence et le trafic aux interfaces mémoires, ce qui limite les performances et l'efficacité énergétique. Une solution à ce problème propose d'utiliser un tampon en écriture, ce qui permet d'opérer en mode « fire and forget » et de pouvoir libérer plus rapidement le processeur. Cet ajout allonge néanmoins le temps où la cache et la mémoire principale sont désynchronisées.

Pour pallier le trafic engendré et la puissance consommée par la synchronisation constante à la mémoire, un mécanisme de type « write-back » peut être utilisé. Cette politique de cohérence permet de repousser la synchronisation de la ligne de cache jusqu'à ce que celle-ci doive être évincée. Ainsi, une ligne de cache peut être modifiée et lue sans avoir à être renvoyée à la mémoire, puisqu'elle reste cohérente avec le processeur. Un bit de contrôle, appelé « dirty bit » garde l'état actuel de la ligne de cache, indiquant si elle est cohérente ou non avec la mémoire principale. Lorsqu'il y a un remplacement de ligne de cache, communément appelé « cache fill », le « dirty bit » est d'abord vérifié. Si le « dirty bit » est activé, la ligne de cache devra d'abord être renvoyée à la mémoire principale avant de pouvoir être évincée pour y placer la nouvelle ligne de cache.

La politique « write-back » permet de réduire considérablement le trafic vers les instances supérieures de la mémoire. Elle regroupe toutes les modifications faites sur la ligne de cache en une seule transaction de synchronisation. Cependant, même si la réduction du trafic coïncide avec une réduction de la latence et de la puissance, ce mécanisme peut engendrer des problèmes de cohérence dans un système multiprocesseurs. La gestion de la cohérence pour un système multiprocesseur n'est cependant pas traitée dans le cadre de ce mémoire.

1.5 Optimisation d'une mémoire cache

La conception d'une cache est en soi un problème d'optimisation. Comme le démontre la section 1.3, il existe de nombreuses manières de concevoir une mémoire cache et de l'interfacer avec un processeur. Une grande partie dépend aussi du contexte dans laquelle la cache sera utilisée ; un contexte de faible puissance ou en temps réel, par exemple. Pour concevoir une cache optimale, il est important de colliger et d'analyser ses métriques adéquatement.

Analyser la qualité d'une mémoire cache n'est pas un exercice trivial, s'approchant davantage de la méta-analyse. Plusieurs métriques propres aux mémoires caches existent et doivent être prises en compte pour boucler l'analyse. Habituellement, on détermine la qualité d'une cache en fonction de certains facteurs de mérite. Les facteurs de mérite d'une cache comprennent habituellement les performances en termes de débit et de temps d'accès, mais aussi en matière de consommation énergétique. De plus, il est primordial d'associer ces facteurs de mérite à un contexte particulier, soit le programme exécuté et l'application du processeur. En effet, l'utilisation de programmes de référence (« benchmarks ») permet de contextualiser les performances d'une cache pour différents scénarios. Finalement, la comparaison de mémoires caches ayant une organisation logique similaire permet de mettre en relief ses avantages et désavantages.

Cette section présente deux axes d'optimisation pour une mémoire cache. En premier lieu, les facteurs de mérite de performances sont expliqués et une revue de littérature sur les techniques d'optimisation existantes est présentée. En second lieu, les facteurs de mérite portant sur l'ef-

ficacité énergétique sont énoncés. Une revue de littérature sur les techniques permettant de réduire la consommation énergétique est ensuite présentée.

1.5.1 Métriques de performance : la latence et le débit

La différence notable entre le temps d'accès à la mémoire et le temps d'exécution d'un processeur est établie depuis un certain temps. En effet, le concept du mur de la mémoire (« memory wall ») a d'abord été mentionné dans (Wulf et McKee, 1995). Les auteurs ont conceptualisé un fait qui était déjà connu et accepté à l'époque, soit que la mémoire deviendrait sous peu le goulot d'étranglement du processeur. Suivant encore (et toujours) la loi de Moore, c'est aujourd'hui chose faite.

La conception d'un processeur moderne performant doit prendre en compte cette problématique. La solution idéale implique le développement d'une cellule mémoire possédant une vitesse de lecture et d'écriture comparable à la logique CMOS. Les solutions actuelles optimisent l'architecture des caches pour pallier au problème, sans pour autant le régler.

La performance d'une cache est principalement évaluée en terme de temps d'accès, le but initial étant de limiter les requêtes à la mémoire principale. Deux métriques sont souvent utilisées pour quantifier les temps d'accès, soit la latence et le débit. La latence exprime le temps requis pour exécuter un type d'opération dans la cache. Plusieurs facteurs peuvent modifier la latence d'une cache, comme le taux de correspondance entre les adresses demandées et les données contenues par la cache. Le taux de « cache miss » ou inversement de « cache hit » affecte le temps d'accès moyen à une cache.

En général, plus le temps d'accès et le ratio de *cache miss* sont petits, plus la cache est performante dans son environnement. Le débit indique plutôt le nombre d'opérations qui peuvent être exécutées sur une période de temps donnée. Cette métrique est habituellement reliée à la structure interne de la cache et plus précisément, l'architecture de son pipeline. Plus le débit d'une cache est élevé, moins le processeur attendra pour recevoir de nouvelles données ou instructions.

Tableau 1.1 Mesures de performance de la mémoire cache

Métrique	**Méthode de calcul**
Cycles par instruction (CPI)	$\frac{\text{Nombre de cycles d'exécution}}{\text{Nombre d'instructions exécutées}}$
Coût de l'accès mémoire	Vrai CPI $-$ CPI avec accès mémoire d'un cycle
Cycles mémoires par instruction (CMPI)	$\frac{\text{Nombre de cycles en mémoire}}{\text{Nombre d'instructions exécutées}}$
Ratio *cache miss*	$\frac{\text{Nombre de } \textit{cache miss}}{\text{Nombre d'accès mémoire}}$
Ratio *cache hit*	$1 - \textit{cache miss}$
Temps d'accès moyen	$(\text{ratio } \textit{cache hit} \times T_{\textit{cache hit}}) + (\text{ratio } \textit{cache miss} \times T_{\textit{cache miss}})$
Million d'instructions par seconde (MIPS)	$\frac{\text{Nombre d'instructions exécutées}}{T_{\text{exécution}} \times 10^6}$

Il existe différentes manières d'exprimer ces métriques, et elles sont présentées dans le tableau 1.1. La métrique de performance préconisée demeure le temps d'exécution total d'une application par un processeur par rapport à un autre (Stallings, 2010). Il est à noter que ce type de métrique fait abstraction de la puissance consommée par le processeur.

1.5.2 Revue des techniques existantes pour améliorer la performance

Une revue de la littérature a été faite pour définir les améliorations architecturales permettant d'augmenter les performances d'une cache. Les techniques dénotées sont réparties en deux catégories. Le premier type de technique réduit le temps nécessaire pour déterminer s'il y a une correspondance dans la cache. Le second type de technique améliore le taux de correspondance en optimisant la localité de référence de la cache. Une arborescence illustrée à la figure 1.6 résume ces différentes techniques. Il est à noter que les techniques recensées ici sont dites classiques puisqu'elles optimisent seulement l'architecture de la cache pour la performance. Les techniques plus récentes se concentrent davantage sur la consommation énergétique des caches. La consommation énergétique d'un processeur est devenue une problématique importante depuis une dizaine d'années (Mittal, 2013).

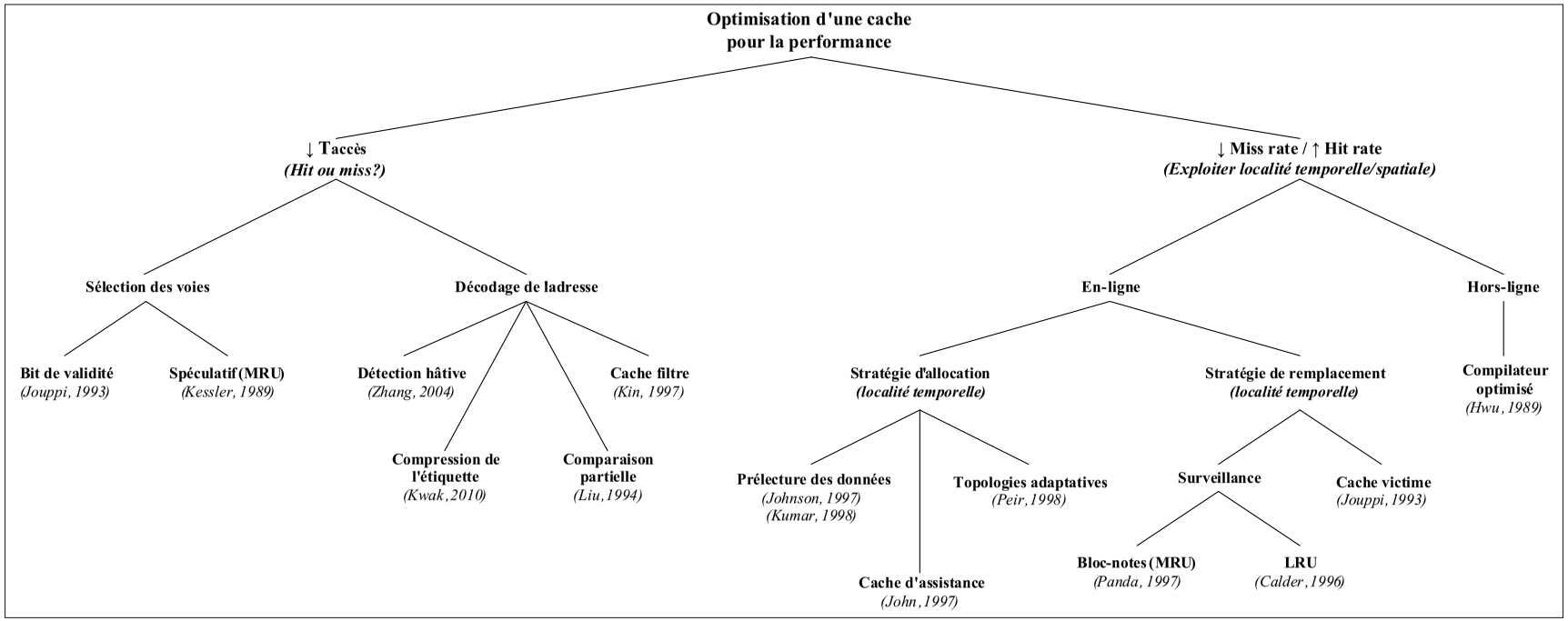


Figure 1.6 Revue des techniques existantes permettant d'améliorer la performance

Le premier type d'optimisation réduit le temps d'accès à la cache en optimisant le traitement de la requête mémoire. Une première méthode consiste à déterminer plus rapidement quelle voie associée à une ligne de cache contient l'information demandée. Dans (Jouppi, 1993), un bit indiquant la validité est combiné à chacune des voies d'une ligne de cache. La valeur du bit est mise à zéro lors de l'initialisation de la cache et est modifiée dès que la voie contient une ligne de cache valide. Cette stratégie permet de déterminer plus rapidement quelle voie contient une information valide et d'éviter de comparer inutilement des étiquettes. Cela dit, cette technique n'est utile qu'à l'initialisation de la cache et devient caduque en régime permanent.

Dans (Kessler *et al.*, 1989), l'implémentation d'une cache « set-associative MRU » est proposée. La comparaison des étiquettes est faite séquentiellement en fonction des voies accédées les plus souvent pour une même ligne de cache. Ce traitement est cependant plus lent qu'une comparaison parallèle des étiquettes, mais devient avantageux en terme d'efficacité énergétique lorsque l'associativité de la cache est élevée.

Une seconde méthode consiste à optimiser le décodage et la comparaison des étiquettes d'adresse. Dans (Kin *et al.*, 1997), une cache-filtre (« filter cache ») est placée à l'entrée de la cache L1 afin d'intercepter les adresses ayant déjà été traitées. Lorsqu'il y une correspondance en lecture, la donnée est immédiatement retournée si c'est une lecture. La cache-filtre agit plutôt comme tampon lors d'un accès en écriture. Afin d'améliorer les performances, la taille de la cache-filtre doit être suffisamment grande. Cependant, plus la taille augmente, plus la latence d'accès et la consommation énergétique augmentent.

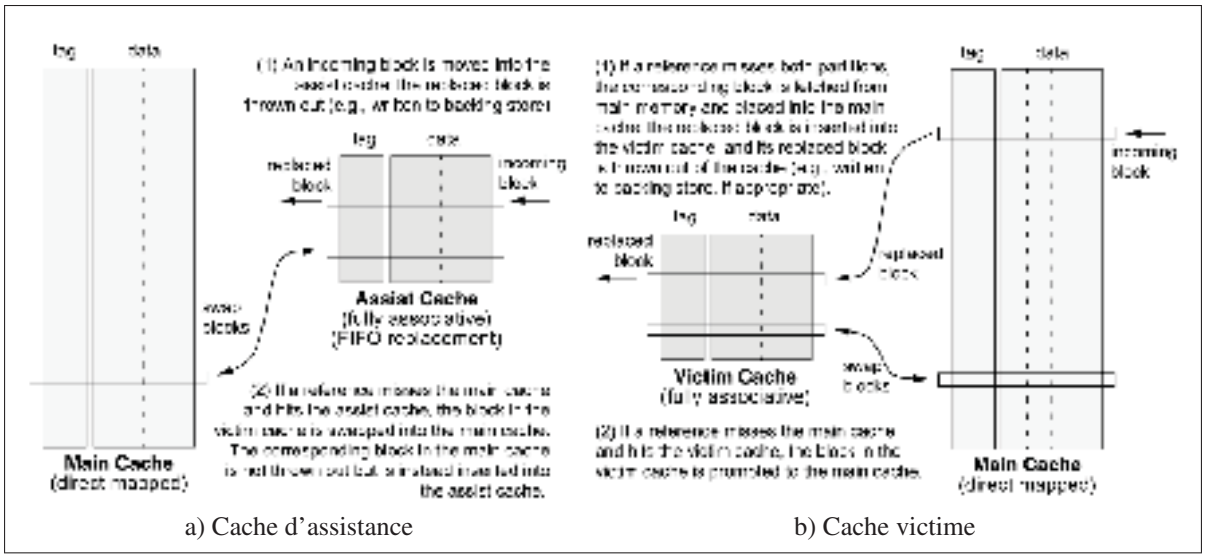
Certaines techniques optimisent le décodage d'adresse afin de détecter rapidement une correspondance. Le circuit de décodage est habituellement un des chemins critiques d'une mémoire cache. Dans (Kwak et Jeon, 2010), (Zhang *et al.*, 2004) et (Liu, 1994) différentes méthodes séquentielles de comparaison des étiquettes sont présentées. Les bits les moins significatifs de l'étiquette sont d'abord comparés, suivis des bits les plus significatifs. Puisque les bits les moins significatifs de l'étiquette changent statistiquement plus souvent, ces derniers permettent de déterminer rapidement s'il y a un « cache miss ».

Le second type d'optimisation réduit le taux de « cache miss ». En exploitant davantage la localité spatiale et temporelle de l'information dans la mémoire, les possibilités de correspondance augmentent. Des techniques hors-ligne existent, comme dans (Hwu et Chang, 1989) où le compilateur est utilisé pour optimiser la répartition des données en mémoire. Cela dit, ce type de méthodologie ne concerne pas l'architecture interne de la cache et ne sera pas traitée dans cette revue.

Les mécanismes en ligne, lors de l'exécution d'un programme, sont d'abord axés sur l'optimisation de la stratégie d'allocation. Les techniques impliquant une opération de prélecture, ou « pre-fetching », permettent d'augmenter le nombre de lignes de cache valides. En effet, les méthodes développées dans (Johnson *et al.*, 1997) et (Kumar et Wilkerson, 1998) augmentent la localité spatiale de l'information présente dans la cache.

Une autre technique intègre une cache d'assistance (« assist cache »), tel qu'illustrée à la figure 1.7a. Dans (John et Subramanian, 1997), la cache d'assistance est placée entre le niveau supérieur de mémoire et la cache principale pour restreindre l'allocation d'une nouvelle ligne de cache. Lorsqu'une information est accédée souvent, elle est promue à la cache principale. Dans le cas inverse, l'information restera dans la cache d'assistance jusqu'à ce qu'elle soit évincée par une nouvelle ligne de cache ayant le même étiquette. Une cache d'assistance limite la contention dans la cache et permet de bénéficier de la localité spatiale des lignes de cache déjà présentes. Cela dit, l'intégration d'une cache d'assistance et la gestion de la cohérence sont complexes à réaliser.

D'autres techniques tentent d'utiliser et de distribuer de manière optimale l'information dans la cache. Dans (Agarwal *et al.*, 1988) et (Peir *et al.*, 1998) différentes topologies d'organisation logique sont intégrées dans la même cache. En fonction du contexte d'opération, l'information contenue peut être accédée à la manière d'une cache « direct-mapped » ou d'une cache « set-associative ». Ces techniques adaptatives sont complexes à intégrer et leur efficacité varie fortement en fonction du programme exécuté.



a) Cache d'assistance b) Cache victime

Figure 1.7 Caches optimisant la localité spatiale en a) et temporelle en b)
Tirée de Jacob *et al.* (2010)

D'autres mécanismes en ligne optimisent le fonctionnement de la cache pour exploiter la localité temporelle de l'information présente en mémoire. Ces mécanismes traitent principalement de la stratégie de remplacement des lignes de cache. Une technique analogue à la cache d'assistance est la cache victime (« victim cache »), telle qu'illustrée à la figure 1.7b. Dans (Jouppi, 1990), la cache victime permet de limiter l'éviction de lignes de cache qui pourraient être réutilisées plus tard dans le programme. Lors du remplacement d'une ligne de cache, la cache victime garde l'information au lieu de l'évincer immédiatement. Si cette information est accédée à nouveau, la ligne de cache en question est à nouveau promue dans la cache. Dans le cas contraire, celle-ci sera définitivement évincée de la cache victime et renvoyée à une instance supérieure de mémoire.

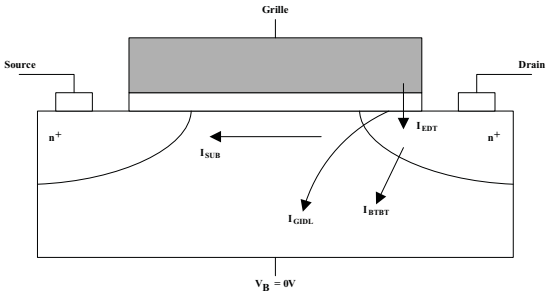
D'autres techniques s'apparentant à la surveillance (« tracing ») permettent aux caches d'adapter leur comportement en fonction de la localité de référence de l'information qui y transige. Dans (Panda *et al.*, 1997) une mémoire de type bloc-notes (« scratchpad ») est couplée au processeur pour y stocker les informations les plus souvent utilisées. Un algorithme « Most Recently Used » (MRU) détermine les transferts de la cache au bloc-notes. Dans (Calder *et al.*,

1996), un algorithme LRU limite la pollution de la cache. L'information la moins utilisée est évincée de la cache de manière adaptative, mais transparente au programme.

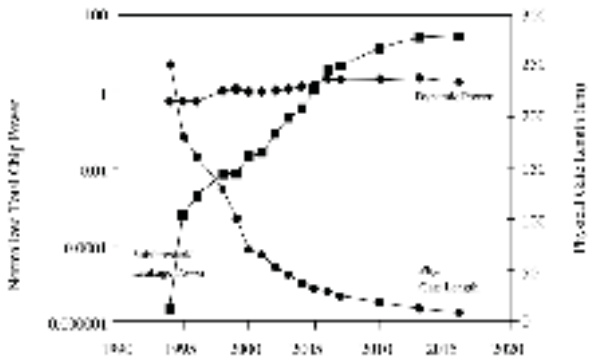
1.5.3 Métriques énergétiques : la consommation dynamique et statique

Depuis les années 2000, il existe une tendance marquée pour le domaine des technologies mobiles. En effet, il n'est plus seulement question de produire le processeur le plus performant, celui-ci doit maintenant être aussi efficace énergétiquement. La consommation énergétique est devenue une problématique importante lors de la conception de circuits électroniques, puisque cette dernière augmente généralement avec chaque nouvelle génération de technologie CMOS. Par ailleurs, les mémoires caches sont devenues un élément essentiel des processeurs modernes. Comme il l'est décrit à la section 1.3, la cache est toujours le seul moyen pour pallier la différence entre la vitesse CPU et la mémoire. Néanmoins, les mémoires caches consomment une partie non négligeable de l'énergie totale d'un processeur. Par exemple, les caches des processeurs Alpha 21264 et StrongARM représentent environ 20% à 30% de l'énergie totale consommée (Mittal, 2013). Dans le cas du processeur ARM d'Octasic, cette proportion atteint environ 50% de l'énergie totale consommée (Laurence, 2013). C'est pourquoi une attention particulière doit être portée à la consommation énergétique d'une mémoire cache lors de sa conception.

La consommation énergétique d'un circuit sur puce, telle qu'une mémoire cache, peut être exprimée en fonction de deux facteurs. Le premier facteur comprend l'énergie qui est consommée dynamiquement en fonction de l'activité réalisée. La consommation dynamique est associée à l'activité de commutation des transistors, où les capacitances parasites sont chargées et déchargées lors de l'exécution d'une tâche. Le second facteur est issu des limitations du procédé de fabrication et est indépendant de la tâche exécutée. La consommation d'énergie statique existe, car on ne peut pas complètement empêcher les électrons de parcourir le canal du transistor, créant ainsi un courant I_{fuite}. La figure 1.8a illustre différents courants de fuite d'un transistor, dont le plus important étant le courant de sous-seuil (« subthreshold »).



a) Courants de fuite d'un transistor



b) Projections émises en 2005 pour les transistors MOS

Figure 1.8 Sources des courants de fuite en a) et l'évolution de la puissance statique par rapport à la puissance dynamique en b)
Tirée de Weste et Harris (2010) et Jacob *et al.* (2010)

Avec la miniaturisation constante des transistors, la puissance statique augmente à un point tel où elle devient plus importante que la puissance dynamique. En réduisant la taille du transistor, on accentue inévitablement les courants de fuite (Weste et Harris, 2010). Les projections actuelles illustrées à la figure 1.8b démontrent une tendance qui n'est pas sur le point de s'estomper pour la technologie MOS (Jacob *et al.*, 2010). Il existe aujourd'hui plusieurs alternatives intéressantes au CMOS, tel que les procédés de fabrication « Silicium-Over-Insulator » (SOI) et les transistors 3D FinFET. Cependant, aucun procédé de fabrication n'est parfait et implique tout de même des considérations énergétiques d'un point de vue dynamique et statique.

Afin d'approximer la consommation énergétique d'une mémoire cache, il est préférable de l'associer à l'énergie consommée durant son opération. La puissance instantanée dissipée par une puce fabriquée selon un processus CMOS provient principalement des courants de commutation et de fuite des transistors (Weste et Harris, 2010).

$$P_{moy}(W) = P_{dynamique} + P_{statique} \equiv C_{tot} \cdot V_{dd}^2 \cdot f + I_{fuite} \cdot V_{dd} \tag{1.2}$$

L'équation 1.2 approxime la puissance moyenne dissipée lors de l'exécution d'un programme. C_{tot} représente la capacitance totale qui est commutée, V_{dd} est la tension d'alimentation, f est la fréquence de commutation et I_{fuite} le courant de fuite total.

Afin d'obtenir une estimation de l'énergie totale consommée lors de l'exécution d'un programme, il faut intégrer cette puissance instantanée sur une période de temps définie. Ainsi, on retrouve l'approximation suivante, où N est le nombre de fois que les capacités parasites des transistors commutent durant le temps d'exécution T :

$$E_{tot}(J) = \int_0^T P_{inst} \equiv P_{moy} \cdot T \equiv C_{tot} \cdot V_{dd}^2 \cdot N + I_{fuite} \cdot V_{dd} \cdot T \tag{1.3}$$

Les métriques représentées au tableau 1.2 permettent d'établir les performances énergétiques d'une mémoire cache. Il est à noter que les facteurs m et n permettent de généraliser la première équation afin d'ajuster le poids de la métrique en fonction de l'énergie/puissance ou du temps. La deuxième métrique correspond à la première lorsque $m = 1$ et $n = 2$.

Tableau 1.2 Mesures d'énergie et de puissance de la mémoire cache

Métrique	**Méthode de calcul**
Produit énergie-délai	$\big($Énergie consommée$\big) \cdot$ (Temps d'exécution de la tâche)
Produit puissance-délai	$($Puissance dissipée$)^m \cdot$ (Temps d'exécution de la tâche$)^n$
MIPS par watt	$\frac{\text{Performance MIPS pour le } \textit{benchmark}}{\text{Puissance moyenne dissipée durant le } \textit{benchmark}}$

Les notions de puissance dissipée et d'énergie consommée sont souvent interchangées dans la littérature. Cela est techniquement incorrect, puisque la première est calculée en watts (W) et la seconde en joules (J). Lorsqu'il est question de temps d'usage et d'autonomie de batteries, l'énergie consommée est une métrique adéquate à utiliser. Cependant, lorsqu'il est nécessaire d'évaluer les effets de la chaleur sur les composantes et comment l'évacuer, il est plutôt question de puissance dissipée.

1.5.4 Revue des techniques existantes pour améliorer l'efficacité énergétique

Un article récent (Mittal, 2013) fait l'anthologie des techniques architecturales développées pour limiter la consommation énergétique des caches. Le premier type de technique réduit la consommation dynamique d'énergie liée à l'activité au sein de la cache. Le second type de technique tente de limiter la puissance statique dissipée par les circuits constituant la cache. Une arborescence telle qu'illustrée à la figure 1.9 résume ces différentes techniques.

Le premier type d'optimisation réduit la puissance dynamique dissipée due à l'activité de commutation au sein de la cache. Les techniques développées sont donc principalement orientées à réduire la quantité de traitement fait par la cache.

Dans (Zhu et Zhang, 2002), la cache progressive (« phased-cache ») présentée implémente un accès séquentiel spéculatif aux étiquettes. La séquence de comparaison des étiquettes est déterminée par un algorithme de prédiction des voies. Cette méthode permet de réduire le nombre de voies accédées et comparées et peut améliorer le temps d'accès à la cache. Néanmoins, un gain n'est quantifiable qu'avec une cache ayant une associativité élevée.

(Kwak et Jeon, 2010), (Park *et al.*, 2012) et (Shafiee *et al.*, 2012) proposent des méthodes pour réduire l'énergie consommée lors de la comparaison des étiquettes. Celles-ci se basent sur la détection rapide du « cache miss » pour les différentes voies comparées. Le décodage et la comparaison des étiquettes sont faits séquentiellement. Globalement, le nombre de voies accédées est moindre et son temps d'accès aussi.

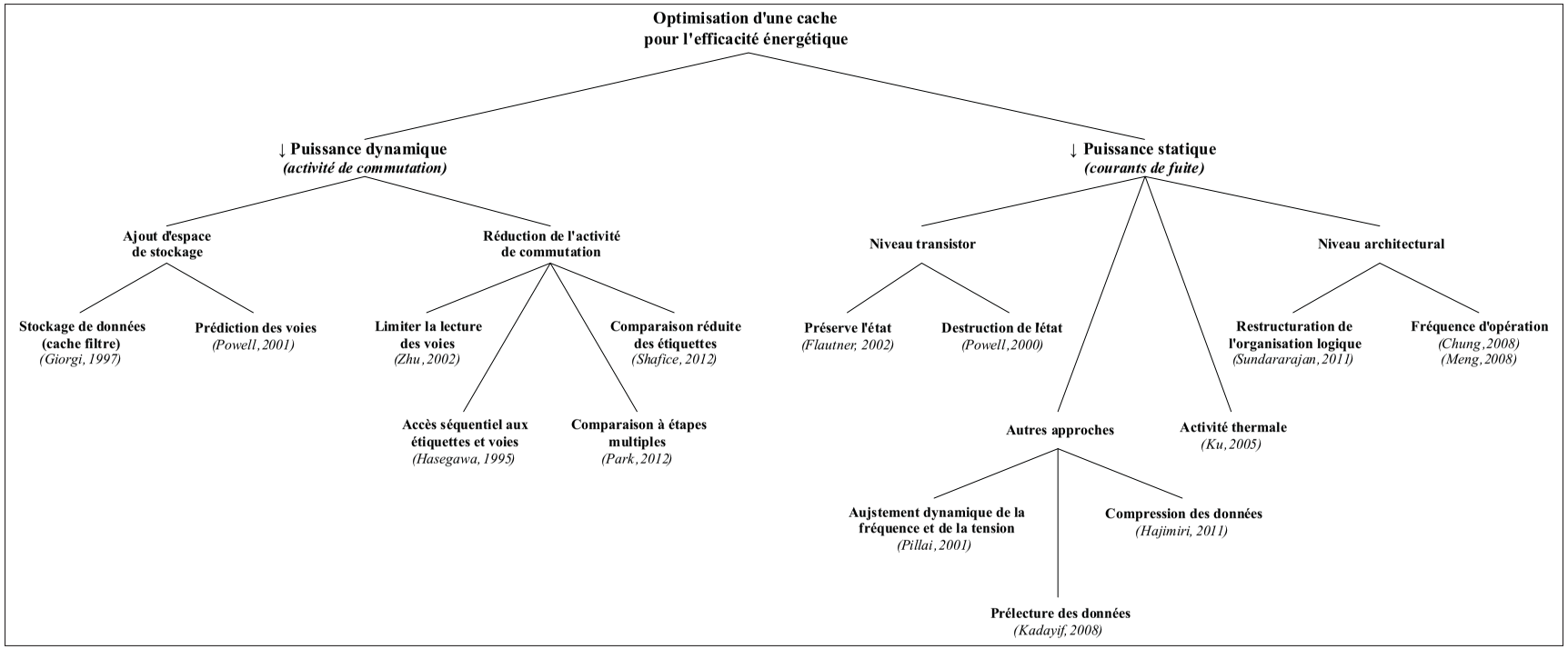


Figure 1.9 Revue des techniques existantes permettant d'améliorer l'efficacité énergétique

D'autres techniques emploient un espace mémoire supplémentaire pour stocker de l'information utile au fonctionnement de la cache. En réduisant le traitement inutile au sein de la cache, on limite la consommation dynamique de celle-ci. Dans (Powell *et al.*, 2001), une mémoire supplémentaire est utilisée pour stocker les prédictions sur la prochaine voie à comparer. Cette technique est similaire à la méthode de (Zhu et Zhang, 2002) et (Zhang *et al.*, 2004) mais s'apparente davantage à un « branch predictor » par la complexité et la taille de la mémoire utilisée.

(Kin *et al.*, 1997) et (Giorgi et Bennati, 2007) utilisent une cache-filtre entre le processeur et la cache. La petite mémoire de la cache-filtre contient les accès les plus fréquents en mémoire, ce qui réduit le traitement effectué par la cache principale.

Il est intéressant de noter qu'une bonne partie des techniques développées pour améliorer la performance peuvent aussi réduire la consommation d'énergie dynamique d'une cache. Le deuxième type d'optimisation vise à réduire la consommation statique d'énergie des caches. Les courants de fuite, tels que décrits dans 1.5.3, sont devenus une problématique non négligeable pour les mémoires cache. Récemment, la miniaturisation et le nombre croissant de transistors sur une même puce n'ont fait qu'amplifier le problème. La réduction de puissance statique est devenue nécessaire pour les caches de niveau supérieur L2 et L3. Ce type de cache atteint maintenant des tailles de quelques dizaines de Mo.

Les méthodes les plus performantes s'appliquent au niveau transistor et limitent directement les courants de fuite au sein des cellules mémoires. Dans (Powell *et al.*, 2000), une technique appelée « Gated-V_{dd} » permet de déconnecter le rail d'alimentation et une partie d'un circuit en insérant un transistor entre ceux-ci. Cette technique est dite destructive puisque toute information placée en mémoire est perdue suite à la perte de l'alimentation.

De manière analogue, une cache somnolente (« drowsy cache ») a été développée et rapportée dans (Flautner *et al.*, 2002). Ce type de cache permet de réduire considérablement les courants de fuite d'un circuit tout en y préservant l'information. La cache somnolente utilise deux rails d'alimentation pour ses circuits, ce qui permet de préserver l'état des éléments mémoires. Lors-

qu'un traitement est nécessaire, le rail d'alimentation à tension élevée est utilisé. Lorsqu'une certaine partie du circuit n'est pas utilisée, le rail d'alimentation à faible tension est connecté. Ces techniques sont très efficaces d'un point de vue énergétique, mais peu performants en terme de vitesse. En effet, les cycles d'activation sur l'alimentation doivent permettre la décharge du réseau capacitif du circuit, ce qui est souvent de l'ordre de milliers de cycles CPU.

Des techniques développées au niveau architectural peuvent limiter la consommation statique des caches. Une réorganisation logique de la cache est combinée de façon adaptative avec des techniques de contrôle d'alimentation dans (Sundararajan *et al.*, 2011). La méthode développée permet d'adapter l'organisation logique de la cache en fonction du programme exécuté à des intervalles définis. Certaines architectures de caches implémentent un ajustement dynamique de la fréquence et de la tension (« dynamic voltage-frequency scaling » ou DVFS) (Pillai et Shin, 2001).

(Chung et Skadron, 2008) et (Meng *et al.*, 2008) utilisent aussi une méthode modifiant la fréquence et la tension d'opération des circuits pour limiter la puissance statique et dynamique de la cache. Cependant, ces techniques sont implémentées avec une logique dynamique et sont donc complexes à intégrer à un flot de conception traditionnel. Dans (Ku *et al.*, 2005), un système de gestion de l'organisation logique en fonction de la température. Exploitant le fait que les courants de fuite augmentent exponentiellement avec la température, cette méthode active les lignes de cache en alternance pour tenter de réduire la température de la puce.

D'autres approches utilisent la compression de données aux niveaux supérieurs de cache pour limiter l'utilisation des cellules mémoires. Cette approche est utilisée dans (Hajimiri *et al.*, 2011) avec des techniques de « Gated-V_{dd} » pour déconnecter l'alimentation des cellules mémoires non utilisées. Dans (Kadayif *et al.*, 2008), les prélectures (« pre-fetching ») en mémoire sont combinées avec la déconnexion des lignes de cache non utilisées par « Gated-V_{dd} » afin de réduire la puissance statique dissipée.

Il est important de noter que les techniques de réduction de puissance statique impliquent habituellement certaines finesses du point de vue de la technologie CMOS utilisée. L'adoption de ces types de technique est donc plus complexe pour les flots de conception standard.

1.6 Conclusion

Les processeurs modernes implémentent aujourd'hui des mémoires cache afin d'augmenter leurs performances. Le goulot d'étranglement créé par les accès à la mémoire a alors été relevé comme étant une problématique importante inhérente aux opérations d'un processeur. La nécessité d'une hiérarchie comprenant des niveaux de mémoire exploitant la localité de référence de l'information a été mentionnée.

Un processeur peut implémenter différentes architectures de mémoires cache pour pallier la problématique des accès mémoire. Leur organisation logique, l'associativité, leur taille ainsi que le type de cellules mémoires utilisées varient en fonction du type de traitement effectué par le processeur. L'optimisation d'une mémoire cache résulte d'une analyse fondée sur des programmes de référence, ou « benchmarks ». Ainsi il est possible d'extraire les facteurs de mérite d'une architecture de mémoire spécifique en simulant des programmes habituellement exécutés par le processeur. Les principaux facteurs de mérite servent dans un premier temps à analyser les performances d'une mémoire cache, soit la latence d'accès et le débit. Dans un deuxième temps, il est possible d'analyser l'efficacité énergétique, soit la consommation d'énergie dynamique et statique. C'est dans l'optique de la conception d'une cache performante et efficace en terme d'énergie que cette revue de littérature a été faite.

Dans ce chapitre, les concepts fondamentaux des processeurs ont d'abord été exposés. Ensuite, la problématique reliée aux accès à la mémoire a mené à la définition des mémoires caches. L'organisation et le fonctionnement logique des caches ont alors été présentés. Les mécanismes de gestion du contenu et de la cohérence ont ensuite été expliqués. Finalement, une revue de la littérature a été présentée sur les techniques permettant d'optimiser les caches pour la performance et pour l'efficacité énergétique.

CHAPITRE 2

PIPELINES ASYNCHRONES

2.1 Introduction

Plusieurs composantes d'un processeur, telles que le module d'exécution et les mémoires caches, implémentent une structure interne basée sur le pipeline. L'architecture pipeline consiste à fractionner les fonctions logiques organisées en de longues séquences en de plus courtes sous-fonctions séparées par des éléments mémoire. Le pipeline permet d'augmenter le débit d'exécution d'une fonction logique en exploitant une forme structurelle de parallélisme (Patterson et Hennessy, 2009).

Traditionnellement, l'exécution de chacune des sous-fonctions du pipeline est cadencée à l'aide d'une seule horloge globale (Patterson et Hennessy, 2009). Ce type d'architecture est dit synchrone. Il existe d'autres techniques pour réaliser un pipeline sans utiliser une horloge globale. Ces derniers exploitent une architecture de type asynchrone. L'architecture des pipelines asynchrone nécessite une logique de contrôle supplémentaire pour garantir la synchronisation des sous-fonctions (Nowick et Singh, 2011).

Dans ce chapitre, les concepts fondamentaux des pipelines sont d'abord introduits. L'architecture de base d'un pipeline synchrone est expliquée et de celle-ci est dérivée l'architecture asynchrone. Les éléments de base constituant un pipeline asynchrone sont ensuite présentés. Finalement, une revue de littérature des pipelines asynchrones existants est exposée.

2.2 Principe d'un pipeline

Une des méthodes classiques pour augmenter les performances d'un processeur consiste à réorganiser sa structure logique interne pour une architecture pipeline. Cette méthode permet d'augmenter le débit d'un processus séquentiel en exploitant une forme structurelle de parallé-

lisme. On réfère au terme pipeline pour désigner ce type de structure, car pour chaque nouvelle donnée traitée à son entrée, une donnée réside à sa sortie (Stallings, 2010).

Le principe d'un pipeline réside dans la décomposition d'une tâche complexe en plusieurs tâches simples séquentielles. La logique combinatoire de la tâche complexe est d'abord scindée en plus petits blocs de logique, créant ainsi des tâches simples. Chacune de ces tâches est ensuite encapsulée dans un étage du pipeline qui contient un registre d'état et la logique combinatoire. Les registres d'état permettent de garder en mémoire le résultat intermédiaire de chaque étage du pipeline. L'architecture d'un pipeline dépend intrinsèquement de la façon dont les étages sont synchronisés entre eux. Pour synchroniser correctement un pipeline, l'exécution de la tâche d'un étage doit être complétée avant de pouvoir transmettre le résultat à un étage subséquent. Cette synchronisation peut être faite globalement (synchrone) ou localement, aux interfaces des étages (asynchrone).

2.3 Architecture des pipelines synchrones

Les pipelines classiques sont basés sur une architecture synchrone. La principale caractéristique d'un pipeline synchrone est l'implémentation d'une horloge globale pour cadencer l'exécution de chaque étage simultanément. La figure 2.1 illustre un pipeline synchrone simple de trois étages utilisant des bascules D (DFF : "D-latch Flip-Flop") comme éléments mémoires.

Pour obtenir la période de l'horloge du pipeline [1], deux contraintes temporelles doivent être satisfaites et vérifiées pour tous les chemins de données entre deux éléments mémoires. La première contrainte détermine le délai de propagation, ou délai logique maximal, entre deux éléments mémoire du pipeline. Dans le cas où les éléments éléments mémoires sont des DFFs, le délai de propagation (Weste et Harris, 2010) est représenté par :

$$T_{pd(n)} \leq T_{Horloge} - (T_{pcq(n)} + T_{setup(n+1)}) \tag{2.1}$$

1. Le décalage de l'horloge (« clock skew ») n'a pas été représenté afin d'alléger les équations temporelles.

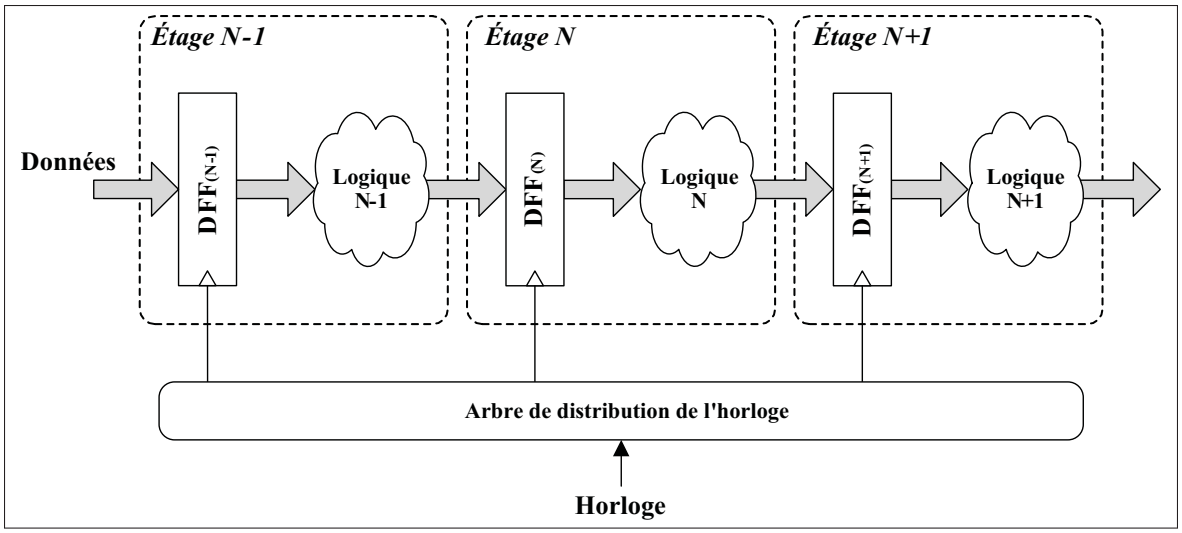


Figure 2.1 Pipeline synchrone

Dans l'équation 2.1 , $T_{pd(n)}$ représente le délai de propagation maximal pour que la logique combinatoire d'un étage n se stabilise. $T_{Horloge}$, représenté comme étant T_c à la figure 2.2, est la période de l'horloge globale du pipeline. $T_{pcq(n)}$ est le délai de propagation maximal pour faire passer une donnée de l'entrée D à la sortie Q d'un registre mémoire DFF à l'étage n suite à un front montant de l'horloge. $T_{setup(n+1)}$ est le temps de stabilisation minimum requis à l'entrée D d'un registre mémoire DFF de l'étage subséquent $n+1$.

Ce temps de stabilisation doit être respecté pour assurer que la nouvelle donnée soit correctement échantillonnée **avant** le prochain front montant de l'horloge. Pour satisfaire la première contrainte temporelle, la période de l'horloge doit être supérieure ou égale au temps de propagation entre deux éléments mémoires du pipeline. Dans le cas où cette contrainte n'est pas respectée, une défaillance de type « setup time » ou délai maximum pourrait se produire. Pour éviter ce type d'erreur, il peut être nécessaire de réduire la taille de la logique combinatoire ou, lorsque ce n'est pas possible, augmenter la période de l'horloge.

La seconde contrainte détermine le délai de contamination, ou délai logique minimal entre deux éléments mémoire du pipeline. Dans le cas où les éléments éléments mémoires sont des

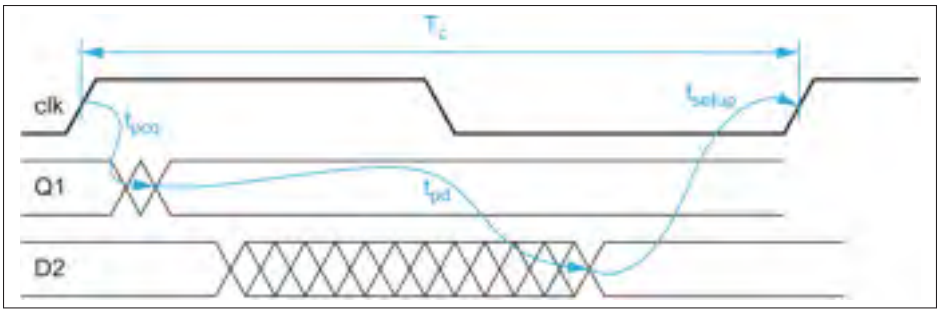


Figure 2.2 Délai de propagation maximal entre deux DFFs
Tirée de Weste et Harris (2010)

DFFs, le délai de contamination (Weste et Harris, 2010) est représenté par :

$$T_{cd(n)} \geq T_{hold(n+1)} - T_{ccq(n)} \tag{2.2}$$

Dans l'équation 2.2, $T_{cd(n)}$ représente le délai de contamination minimal pour que la logique combinatoire d'un étage n se stabilise. $T_{hold(n+1)}$ est le temps de maintien minimum requis à l'entrée D d'un registre mémoire DFF de l'étage subséquent $n+1$. Ce temps de maintien doit être respecté pour éviter de modifier (ou contaminer) la donnée échantillonnée **suite** à un front montant de l'horloge. $T_{ccq(n)}$ est le délai de contamination minimal pour faire passer une donnée de l'entrée D à la sortie Q d'un registre mémoire DFF à l'étage n suite à un front montant de l'horloge.

Pour satisfaire la seconde contrainte temporelle, le temps de maintien doit être inférieur ou égal au temps de contamination entre deux éléments mémoires du pipeline. Dans le cas où cette contrainte n'est pas respectée, une défaillance de type « hold time » ou délai minimum pourrait se produire. Pour éviter ce type d'erreur, il peut être nécessaire de ralentir la logique combinatoire à l'aide de lignes à délai. Il est à noter que cette contrainte est indépendante de la période de l'horloge.

Ces deux contraintes temporelles permettent de réduire les possibilités de métastabilité dans le circuit, conséquence immuable à l'utilisation d'éléments mémoires. Il existe toutefois d'autres contraintes, de nature structurelle, qui complexifie davantage le travail de conception d'un pi-

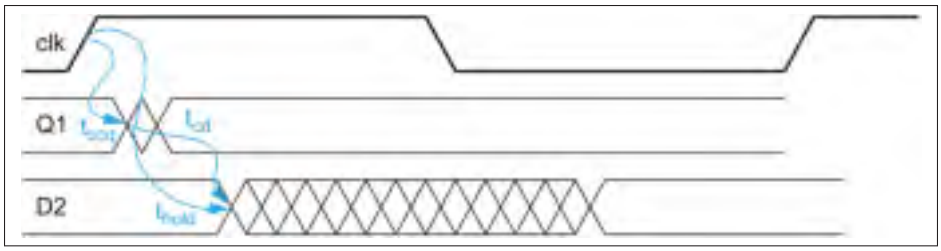


Figure 2.3 Délai de contamination minimal entre deux DFFs
Tirée de Weste et Harris (2010)

peline synchrone. En effet, le pipeline doit implémenter un contrôleur pour gérer les interruptions de traitement au sein du pipeline, communément appelé pipeline « freeze ». Ce mécanisme, appelé « clock-gating » permet de suspendre globalement ou localement la distribution de l'horloge lors de ces événements.

Au fil du temps, l'application de ces deux contraintes temporelles simples sur une horloge globale a pavé la voie à une méthodologie de conception synchrone efficace. Il existe maintenant des outils de développement et d'analyse automatisés, matures, et largement adoptés par l'industrie. Ces logiciels de CAO permettent de réduire considérablement le temps de conception des circuits. Cela étant dit, les circuits basés sur une architecture synchrone exhibent des désavantages notables. Avec la miniaturisation à l'échelle submicronique des transistors et l'augmentation constante de leur nombre sur une même puce, les effets indésirables dus à l'interconnexion deviennent non-négligeables (Beerel *et al.*, 2010).

En effet, les délais attribuables à la longueur des fils ainsi que la diaphonie exacerbée par le nombre d'interconnexions entre les transistors affectent les performances des circuits. Les outils de CAO intègrent maintenant des optimisation post-placement et post-routage pour pallier ce problème grandissant. Cependant, les algorithmes complexes derrière ces optimisations requièrent une très grande quantité de calculs, entraînant souvent des solutions sous-optimales en terme de performance. Un autre effet lié à la miniaturisation des transistors est que ces derniers consomment davantage de puissance statique. Ce phénomène fait qu'il devient de plus en plus difficile de respecter un budget énergétique pour un circuit donné.

2.4 Architecture des pipelines asynchrones

Avec l'augmentation des limitations physiques et de la complexité au sein des circuits synchrones, on assiste à un regain d'intérêt envers les architectures asynchrones. Contrairement aux architectures synchrones, la vitesse d'opération maximale d'un pipeline asynchrone n'est pas dictée par le plus long délai de propagation au sein d'un des étages du pipeline. Les architectures asynchrones n'utilisent pas d'horloge globale. La synchronisation entre les étages de pipeline est plutôt assurée par un protocole de communication. Ce protocole est basé sur une période de négociation, dénotée « handshake », qui détermine lorsqu'un étage du pipeline peut traiter de nouvelles données et les transmettre à un étage subséquent. Les données, pouvant être encodées, sont regroupées avec les signaux de synchronisation dans les canaux de communication unidirectionnels entre les étages du pipeline. Le contrôle de l'horloge est effectué localement et sa distribution généralement point à point.

Il existe plusieurs types d'architecture asynchrone dans la littérature. Ces architectures diffèrent d'une part par l'organisation des canaux de communication, soit l'encodage des données et le protocole de communication implémenté. D'une autre part, les architectures asynchrones sont caractérisées par le type d'éléments mémoires utilisés ainsi que leurs contraintes temporelles. La figure 2.4 illustre un pipeline asynchrone implémentant un encodage de donnée groupé (« bundled data »).

Les circuits basés sur une architecture asynchrone possèdent certains avantages caractéristiques par rapport à leur équivalent synchrone. Bon nombre de réalisations physiques font état de ces bénéfices et ont fait l'objet de publications dans la littérature (Beerel *et al.*, 2010).

Les principaux avantages en terme de performance sont dus à l'absence de réseau de distribution global de l'horloge. Une distribution d'horloge locale permet de limiter le décalage en phase (« clock skew ») de celle-ci. Les contraintes temporelles sont appliquées localement ; individuellement sur chaque étage de pipeline. Les circuits asynchrones bénéficient ainsi d'une période d'horloge basée sur le délai moyen, contrairement au plus grand délai (« worst-case

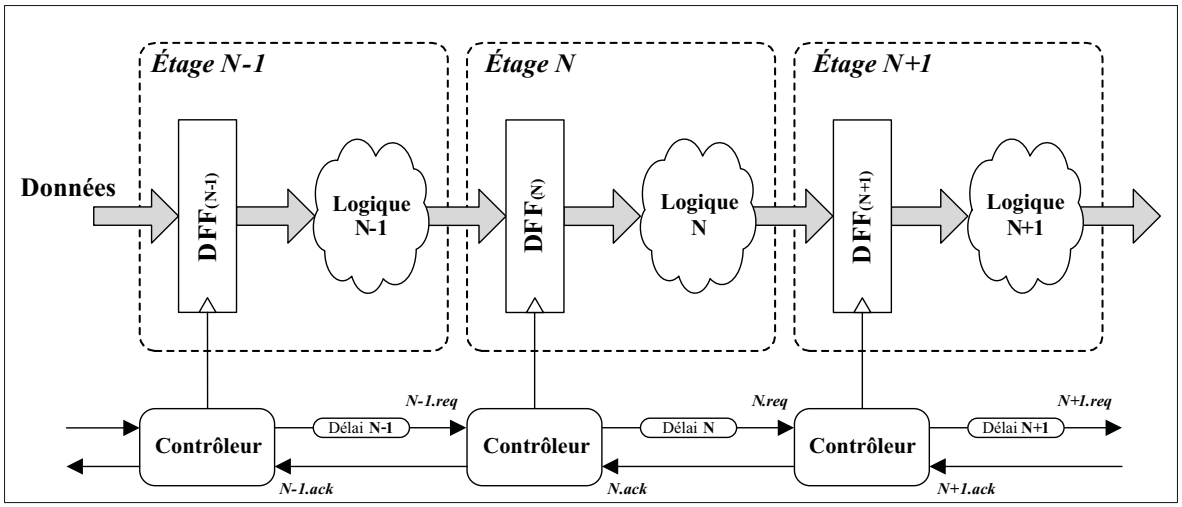


Figure 2.4 Pipeline asynchrone

delay ») dans le cas des circuits synchrones. De plus, les circuits asynchrones s'adaptent généralement mieux aux variations de procédé.

L'implémentation d'une architecture asynchrone comporte aussi des bénéfices en terme d'efficacité énergétique. D'une part, les circuits asynchrones consomment moins d'énergie, car ils n'intègrent pas de vaste réseau de distribution d'horloge. D'une autre part, puisque le traitement des données est basé sur des événements ponctuels et non périodiques, le déplacement de données est limité. Cela est non seulement optimal en terme d'énergie, mais limite le taux d'interférences électromagnétiques par l'activation sporadique de l'horloge (Furber *et al.*, 1999)(Bink et York, 2007). Il est à noter que ce n'est pas tous les types d'implémentation qui possèdent les mêmes avantages. Il existe généralement un compromis entre performance, efficacité énergétique et robustesse.

Cela étant dit, les circuits asynchrones possèdent aussi des désavantages et certaines limitations. En effet, l'existence de boucles combinatoires qui ne sont pas scindées par des registres mémoires rend les tâches de déverminage et de test plus ardues (Kondratyev et Lwin, 2002)(Te Beest *et al.*, 2003). De plus, il n'existe qu'un seul outil de CAO asynchrone commercial, *TiDE* (Solutions), qui utilise son langage de programmation asynchrone propriétaire

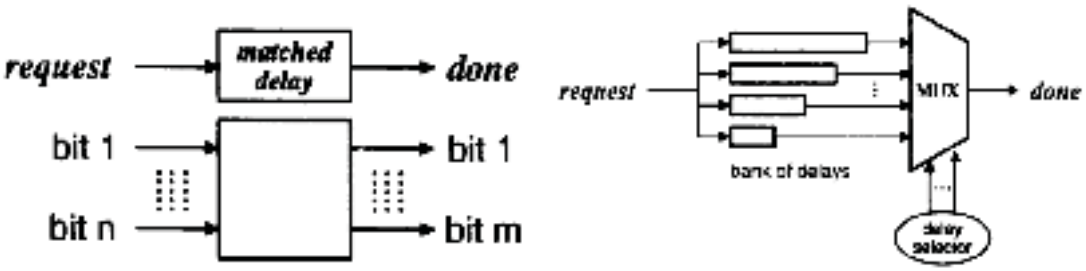
Haste. Il n'existe aucun flot de conception asynchrone standard à ce jour. Pour l'instant, l'intégration des paradigmes asynchrones aux outils de conception synchrone conventionnels est limitée (Ligthart *et al.*, 2000), sinon pratiquement inexistante.

2.4.1 Encodage des données

Dans un système synchrone, on peut supposer que le chemin de donnée est stabilisé lorsque les données sont placées dans un registre suivant le front montant de l'horloge. Ce n'est toutefois pas le cas dans un système asynchrone, c'est pourquoi des techniques d'encodage de données ont été développées afin de garantir la validité de l'information transmise. L'encodage implémente un détecteur de complétion pour déterminer lorsque les données sont valides et prêtes à être mises en mémoire. Deux techniques d'encodage sont présentés dans cette section.

Le premier type d'encodage, appelé données groupées (« bundled data »), ne modifie pas directement le chemin de données. Il propage plutôt un signal de complétion équivalent au délai de propagation maximal du nuage logique $T_{pd(n)}$. Comme il est illustré à la figure 2.5a, une ligne à délai simule $T_{pd(n)}$ et permet au nuage de logique de se stabiliser avant d'envoyer le signal de complétion. Un détecteur de complétion adapté à l'encodage en données groupées est présenté à la figure 2.5b. Un multiplexeur permet de choisir le bon signal de complétion en fonction de l'opération effectuée.

Les principaux avantages de ce type d'encodage sont sa faible complexité et son faible coût. Il est possible de convertir un système synchrone à asynchrone en réutilisant les mêmes blocs de logique et en y adaptant les lignes à délai adéquates. De plus, puisqu'il est possible d'ajuster la ligne à délai en fonction de la logique combinatoire, le chemin de donnée ne doit pas nécessairement être épuré d'aléas statiques. Le désavantage de cette méthode est que la valeur de la ligne à délai est fixée suite à une analyse temporelle STA. Ce type d'encodage est donc indépendant des données.



a) Encodage de type données groupées

b) Détecteur de complétion de type données groupées

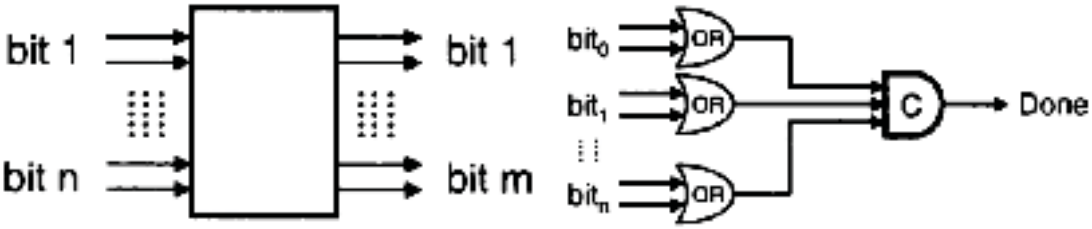
Figure 2.5 Un a) encodeur et un b) détecteur de complétion de type données groupées
Tirée de Singh (2002)

Le second type d'encodage, dénommé 1-de-N (« 1-of-N »), modifie le chemin de donnée à rail unique conventionnel pour un système à N rails. Ce type d'encodage utilise N fils pour transmettre $\log_2 N$ données. Un exemple classique est l'encodage 1-de-2. Chaque bit est encodé sur deux rails afin de fusionner la valeur de la donnée avec sa validité. Ainsi, les valeurs 0 et 1 sont respectivement encodées selon les combinaisons "10" et "01". Cela dit, afin d'assurer la validité d'un signal, une remise à zéro de la combinaison doit préalablement être envoyée, soit "00". Cette séquence est aussi appelée un espaceur. Le tableau 2.1 résume l'encodage 1-de-2.

Tableau 2.1 Encodage des données 1-de-2

Encodage	**Description**
00	Espaceur (RAZ)
01	« 1 » valide
10	« 0 » valide
11	Non valide.

Une autre particularité de l'encodage est qu'il nécessite un détecteur de complétion plus complexe. En effet, tous les signaux du chemin de donnée doivent être valides avant de pouvoir générer le signal de complétion. Un étage de portes OU suivi d'un arbiteur Müller-C (Beerel *et al.*, 2010) sont nécessaires pour déterminer la validité du chemin de donnée. La figure 2.6 illustre respectivement un encodeur et un détecteur de complétion de type 1-de-2.



a) Encodage de données de type 1-de-2 b) Détecteur de complétion de type 1-de-2

Figure 2.6 Un a) encodeur et un b) détecteur de complétion 1-de-2
Tirée de Singh (2002)

Les principaux avantages de ce type de méthode sont sa robustesse et sa capacité à s'ajuster à différentes opérations logiques sans l'utilisation de lignes à délais pré-ajustées. Ceci dit, cette robustesse a un coût élevé en termes de portes logiques et de lignes de routage puisque les circuits doivent être répétés *N* fois.

2.4.2 Protocole de communication

Les protocoles de communication des pipelines asynchrones sont basés sur une période de négociation, dénotée « handshake ». Cette phase de négociation permet de gérer les signaux de complétion entre les étages du pipeline, soit les requêtes (R) et les accusés de réception (A). Typiquement, les protocoles de communication utilisent ces deux signaux de contrôle au lieu de celui de la requête, pour assurer une rétroaction dans la logique de contrôle. La requête est transmise d'un étage du pipeline au suivant, tandis que l'accusé réception est envoyé à l'étage qui le précède. Il existe trois principaux protocoles de communication dans la littérature.

Le premier protocole de communication est basé sur une période de négociation à quatre phases. Ce protocole est aussi dénommé signalisation par niveau logique, puisqu'on se base uniquement sur les niveaux logiques de la requête et de l'accusé réception pour déterminer l'état de l'étage du pipeline. Chaque événement est divisé en quatre étapes distinctes, illustrées à la figure 2.7, soit : *a*) début de l'événement, où R=1 et A=0 ; *b*) fin de l'événement, où R=1 et A=1 ; *c*) préparation pour le prochain événement, où R=0 et A=1 ; *d*) prêt pour le prochain

événement, où R=0 et A=0. Il est à noter que les deux premières étapes de ce protocole sont considérées comme étant actives, puisqu'elles communiquent l'état du traitement des données. Les deux dernières étapes font office de retour à zéro pour rétablir l'état initial du chemin de donnée avant le prochain événement.

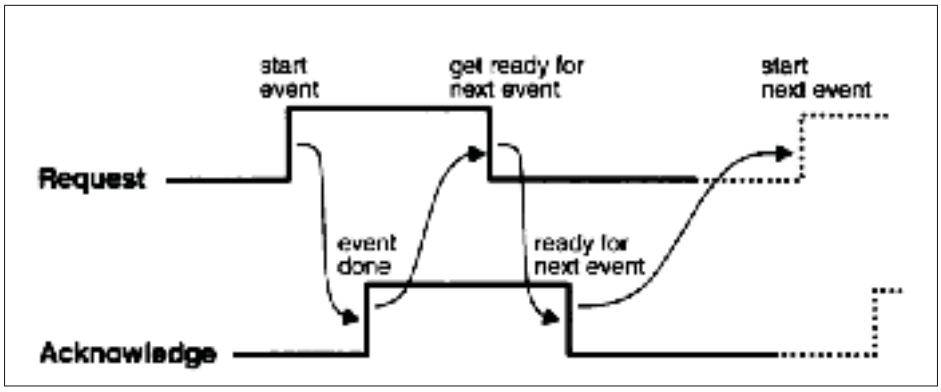


Figure 2.7 Protocole de communication à quatre phases
Tirée de Singh (2002)

Le principal avantage de ce protocole de communication est sa robustesse et sa simplicité, étant donné qu'il est basé sur une combinaison de niveaux logiques. Une ligne à délai génère le signal d'accusé réception et les détecteurs sont composés de portes logiques. Cette simplicité d'implémentation n'est toutefois pas gratuite, puisqu'elle impose une période de remise à zéro nécessaire au bon fonctionnement du pipeline. En effet, pour assurer une bonne synchronisation du pipeline, un étage sur deux doit être remis à zéro pour se préparer au traitement de la prochaine donnée. Les unités de traitement de ces pipelines utilisant ce type de protocole ont donc une capacité maximale de 50%, ce qui est donc un franc désavantage en terme de performance.

Le deuxième type de protocole de communication est basé sur une période de négociation à deux phases, en utilisant que la partie active du protocole précédent. Puisque ce protocole utilise seulement ces deux phases, une signalisation par transition doit être implémentée. Les événements sont divisés en deux étapes distinctes, illustrées à la figure 2.8, soit : *a*) début d'un

événement, lorsque $R \uparrow$ et A=0 ou lorsque $R \downarrow$ et A=1 ; *b*) fin d'un événement, lorsque $A \uparrow$ et R=1 ou lorsque $A \downarrow$ et R=0. Il est à noter qu'il y a absence d'événement lorsque R=A.

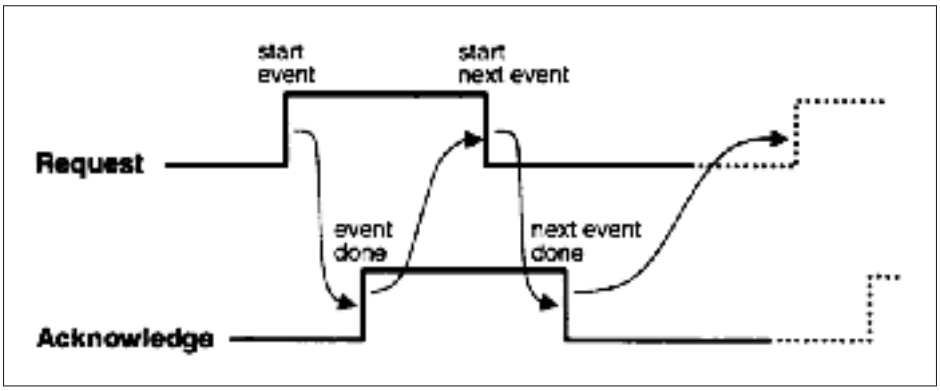


Figure 2.8 Protocole de communication à deux phases
Tirée de Singh (2002)

Le principal avantage de ce protocole de communication est qu'il ne comporte pas de remise à zéro, contrairement au protocole précédent. Il est donc plus efficace en terme d'énergie et en terme de latence. Par contre, cet avantage vient au prix d'un système de détection plus complexe qui doit être muni d'une logique de contrôle sans aléas temporel. Contrairement au protocole précédent, le protocole à deux phases intègre un élément mémoire pour détecter les transitions et être en mesure de garder l'état actuel de l'étage de pipeline. Cela dit, il est possible de concevoir un pipeline ayant une capacité maximale de 100%, moyennant une logique contrôle plus complexe. Ceci est un avantage décisif par rapport au premier protocole.

Le troisième type de protocole de communication transmet les signaux de complétion de la phase de négociation en mode pulsé. Ce protocole de communication en mode pulsé (« pulsed-mode handshake ») propose un compromis entre les protocoles à deux et à quatre phases. En effet, le protocole implémente les deux phases actives de requête et d'accusé réception, mais peut se baser sur les niveaux logiques ou les transitions pour déterminer l'état de la phase de négociation. Ainsi, seulement deux transitions ou niveaux logiques mutuellement exclusifs peuvent modifier la phase de contrôle du pipeline. Les événements sont divisés en deux phases,

telles qu'illustrées à la figure 2.9, soit : *a*) début d'un événement, lorsque R=1 et A=0 ; *b*) fin d'un événement, lorsque R=0 et A=1.

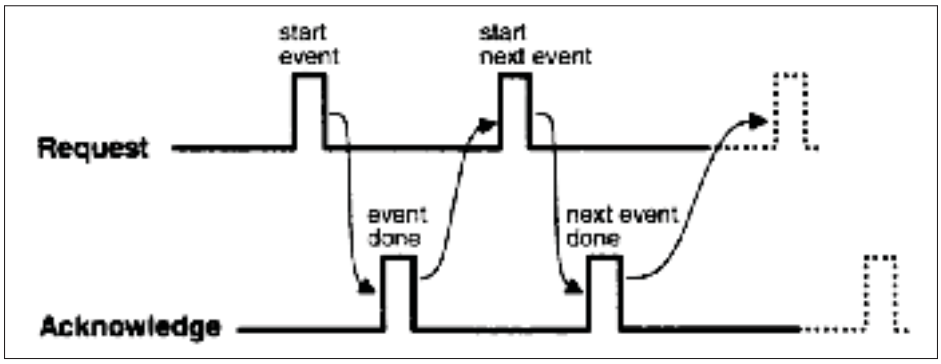


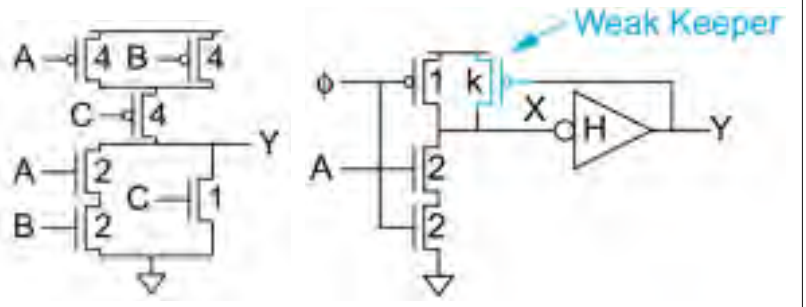
Figure 2.9 Protocole de communication en mode pulsé
Tirée de Singh (2002)

Ce type de protocole doit toutefois satisfaire une contrainte temporelle. Une largeur d'impulsion minimale doit être maintenue pour que le changement de phase soit détecté et que les données soient adéquatement placées dans un registre. De plus, la logique de contrôle doit être exempte d'aléas statiques. Malgré ces contraintes, ce protocole permet de concevoir un pipeline ayant une capacité de 100% tout en gardant une logique de détection simple.

2.4.3 Famille de circuits logiques

La grande majorité des portes logiques des circuits intégrés sont conçues avec des réseaux de transistors pMOS et nMOS complémentaires, respectivement reliés à l'alimentation V_{DD} et la masse. Les librairies standard basées sur la logique CMOS statique sont supportées par tous les outils de CAO conventionnels et sont largement utilisées au sein de l'industrie. Ce type de logique présente certains avantages, tel que son immunité au bruit et aux variations de fabrication, sa faible consommation et pour sa facilité d'intégration (Weste et Harris, 2010). Cependant, lorsque la vitesse d'opération ou la taille de la puce devient une contrainte, il peut être intéressant de recourir à d'autres familles de circuits logiques, lors de la conception d'un pipeline asynchrone par exemple. En effet, les circuits CMOS statiques sont limités par la taille

des transistors pMOS qui doit être au moins deux fois plus grand que celle de son complément nMOS afin d'équilibrer les temps de commutation. La fonction logique AOI21 présentée à la figure 2.10a illustre bien cette contrainte de conception. En général, les fonctions logiques complexes sont difficiles à concevoir en logique CMOS et sont peu performantes, principalement dû au réseau pMOS lent et capacitif.



a) Porte logique CMOS statique

b) Logique domino

Figure 2.10 Une porte AOI21 en a) logique statique CMOS et un inverseur en b) logique domino. Les chiffres représentent le ratio de la largeur vs. la longueur (W/L) du transistor
Tirée de Weste et Harris (2010)

Il existe d'autres familles de circuits logiques qui peuvent pallier au problème de vitesse des circuits CMOS statiques. La logique domino, basé sur les circuits logiques dynamiques, n'utilisent que les transistors nMOS petits et rapides afin de concevoir la fonction logique. La figure 2.10b illustre l'implémentation d'un inverseur en logique domino. Un seul transistor pMOS est placé entre l'alimentation et réseau de logique nMOS et est activé lors de la phase de préchargement. Un transistor nMOS est placé entre la masse et la logique nMOS et active la phase d'évaluation de celle-ci.

Ces circuits logiques utilisent une horloge pour assurer les cycles de préchargement et d'évaluation. Le nœud dynamique, représenté par X à la figure 2.10b, est le point d'interconnexion entre le transistor pMOS et le réseau nMOS. Lors de la phase d'évaluation, la valeur du nœud dynamique est mise à la masse si la condition logique s'avère vraie. Pour contrer les effets

de la fuite de courant sur le noeud dynamique, un inverseur de petite taille (« weak keeper ») est ajouté en rétroaction pour conserver le niveau logique à sa sortie. Afin d'assurer la compatibilité des niveaux logiques et pour pouvoir partager la même horloge, la logique domino insère un inverseur CMOS statique suite au noeud dynamique. Ceci augmente d'une part son immunité au bruit et élimine la rétroconduction.

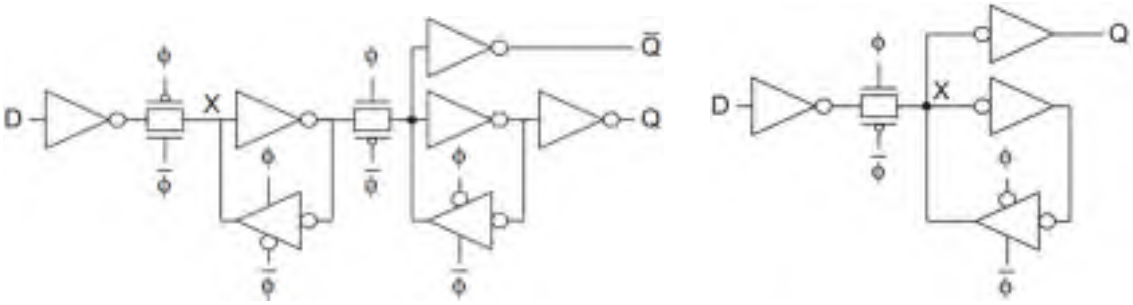
La logique domino bénéficie d'une vitesse d'opération élevée, ce qui est dû à sa faible capacitance d'entrée. De plus, la boucle de rétroaction d'inverseurs à la sortie du noeud dynamique peut être utilisée comme un élément mémoire. Alors qu'aucune puissance statique n'est dissipée en absence de l'horloge (Weste et Harris, 2010), ce type de circuit souffre cependant d'une puissance dynamique très élevée. De plus, une attention considérable doit être portée à l'application de l'horloge et au respect des marges de bruit. Finalement, il existe peu de librairies standard basées sur la logique domino et de support pour les outils de CAO conventionnels.

2.4.4 Types d'éléments mémoire

Les éléments mémoires sont des composantes essentielles des circuits séquentiels. Contrairement aux circuits logiques combinatoires, la sortie de ne dépend pas seulement de l'entrée, mais aussi de l'*état* actuel du circuit. L'état d'un circuit est stocké dans ses registres internes, qui intègrent au moins un type d'élément mémoire. Les machines à états finis et les pipelines sont des exemples classiques de circuits séquentiels. Le choix du bon type d'élément mémoire pour la conception d'un circuit séquentiel n'est pas trivial. Chacun présente un compromis entre vitesse (latence), consommation d'énergie, taille, disponibilité et testabilité.

La bascule D, ou DFF, est un type d'élément mémoire largement utilisé pour les circuits séquentiels. Les contraintes temporelles présentées aux équations 2.1 et 2.2 sont simples et bien intégrées aux logiciels de CAO. Cependant, l'architecture interne de ce type d'élément mémoire impose quelques inconvénients. D'une part, la capacité d'entrée élevée implique un délai d'activation élevé. La configuration en cascade des deux loquets transparents, illustrée à la figure 2.11a, augmente également la latence d'opération des DFFs. D'une autre part, la

capacité élevée lors de l'activation de l'horloge ϕ implique une puissance dynamique importante. De surcroît, les deux boucles d'inverseurs entraînent une puissance statique considérable. Les DFFs représentent une solution sensée pour concevoir un circuit où la taille et la vitesse d'opération ne sont pas critiques.



a) Schéma interne d'une DFF b) Schéma interne d'un loquet

Figure 2.11 Représentation d'une a) DFF et d'un b) loquet en logique statique
Tirée de Weste et Harris (2010)

Les loquets, ou « latch », est un type d'élément mémoire plus compact et rapide que les DFFs. La figure 2.11b illustre un loquet, qui est essentiellement la moitié d'une DFF. Les loquets peuvent opérer en mode transparent ou en mode pulsé. Le mode transparent permet de laisser passer la donnée D jusqu'à la sortie Q lors d'un niveau logique haut ou bas de l'horloge ϕ. Lorsque le niveau logique de l'horloge ϕ est inversé, l'inverseur « tri-state » est activé et capture le niveau logique au noeud X. La porte de transmission demeure *opaque* jusqu'au prochain niveau logique de l'horloge ϕ.

Le mode pulsé utilise la même architecture interne de loquet, mais utilise une impulsion comme signal d'horloge. Les contraintes temporelles des loquets sont plus complexes et une attention particulière doit être portée à période d'activation (transparence). Néanmoins, ce compromis peut être intéressant puisque les loquets sont plus rapides, plus petits et consomment moins d'énergie que les DFFs. Les loquets font partie des librairies de cellules standard et leurs contraintes temporelles sont généralement bien intégrées aux outils de CAO.

Les deux types d'élément mémoire précédents peuvent être implémentés en logique statique, ce qui est idéal pour un flot de conception standard. Il existe toutefois d'autres manières de garder en mémoire l'état d'un circuit séquentiel. Dans le domaine des pipelines asynchrones, les éléments Müller-C sont largement utilisés pour les registres d'état (Nowick et Singh, 2011). Leur implémentation est possible en logique statique ou dynamique, mais elles font rarement partie des librairies de cellules standard. Plus souvent qu'autrement, un flot de conception personnalisé (« custom ») est nécessaire pour utiliser ce type d'élément mémoire. D'autres types d'élément mémoire sont implémentés en logique dynamique, tel que les circuits de logique domino. Dans le cadre de ce projet, seuls les types de mémoires implémentés en logique statique seront implémentés.

2.5 Revue de pipelines asynchrones existants

Il existe deux grands paradigmes de conception de pipelines asynchrones dans la littérature. Le premier modèle base la conception sur certaines contraintes temporelles, tandis que le second tente de faire abstraction de celles-ci. Une grande partie de la littérature asynchrone est consacrée aux pipelines QDI (« Quasi-Delay-Insensitive »). Ce modèle de pipeline élimine plusieurs des hypothèses temporelles aux délais des fils et portes logiques, à l'exception des fourches isochrones (Beerel *et al.*, 2010). Malgré le fait que la méthodologie de conception QDI soit robuste, il est possible d'obtenir de meilleures performances en introduisant certaines contraintes temporelles.

Cette section du mémoire présente plutôt une revue des pipelines asynchrones basés sur des contraintes temporelles. Tout d'abord le pipeline PS0 et le micropipeline sont étudiés brièvement, puisqu'ils sont basés sur des éléments mémoires non-standards. Une première variante du micropipeline utilisant des loquets, le Mousetrap, est ensuite présentée. Finalement, une seconde variante du micropipeline qui utilise est présentée, dénommée éléments Click.

2.5.1 PS0 (William et Horowitz)

Le pipeline PS0 Williams (1990), développé par Ted Williams et Mark Horowitz fait figure de référence dans le domaine des pipelines asynchrones utilisant la logique dynamique. Le terme « PS0 » indique que le pipeline utilise la logique de **P**récharge, utilise une synchronisation **S**imple et n'utilise pas (**0**) d'éléments mémoire.

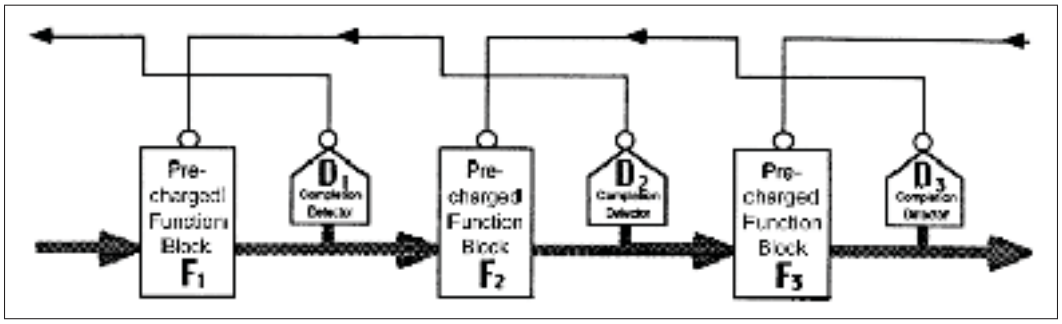


Figure 2.12 Pipeline asynchrone PS0
Tirée de Williams (1990)

Le pipeline PS0, illustré à la figure 2.12, est conçu de la façon suivante. Le chemin de donnée est basé sur des blocs de logique domino à double rail, produisant des sorties F et $\bar{F}$. La phase de négociation est basée sur le protocole de communication à quatre phases. L'encodage et le détecteur de complétion utilisés sont de type 1-de-2. Le signal de complétion contrôle les phases de préchargement et d'évaluation de chaque bloc de logique domino. Les étages du pipeline PS0 alternent donc respectivement entre les états invalide et valide lors son opération. Le principal avantage du pipeline PS0 est sa faible latence due à une logique de contrôle simple, ce qui en fait un candidat idéal pour des « self-timed ring ». Cependant, ce type de pipeline souffre de désavantages notables, soit la capacité de traitement des données limitée à 50% et l'absence d'éléments mémoires.

2.5.2 Micropipeline (Sutherland)

Le micropipeline (Sutherland, 1989) a été proposé par Ivan Sutherland comme étant une alternative aux pipelines asynchrones utilisant des blocs logiques dynamiques. Contrairement au PS0, le chemin de donnée est constitué de blocs de logique CMOS statique à rail unique. Le pipeline implémente un protocole de communication à deux phases, ce qui permet d'éviter les phases de remise à zéro. Un encodage simple en données groupées est combiné à un arbiteur Müller-C, qui fait office de détecteur de complétion. De plus, le micropipeline utilise des registres « Capture & Pass » comme éléments mémoire, ce qui facilite la tâche de conception et de vérification du circuit.

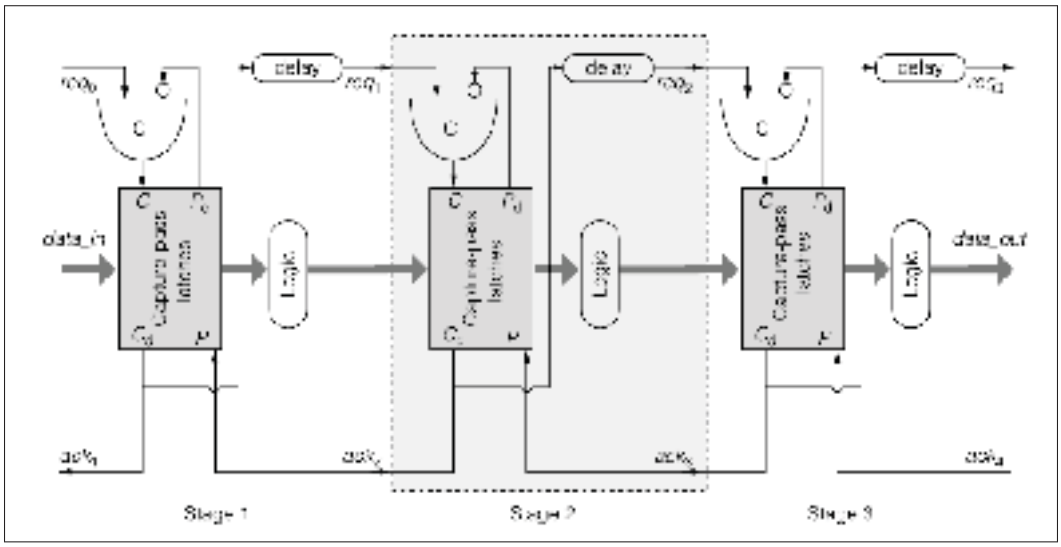


Figure 2.13 Micropipeline
Tirée de Nowick et Singh (2011)

Chaque étage du pipeline communique avec le précédent et le suivant par transition sur les signaux de requête et d'accusé-réception. Chaque registre a deux entrées, *capture* (C) et *pass* (P), ainsi que deux sorties *capture done* (Cd) et *pass done* (Pd), comme l'indique la figure 2.13. Les signaux C et P contrôlent les deux phases du registre, soit : *a*) opaque, ce qui permet de saisir l'information de l'étage précédent lorsqu'il y a une transition sur C ; *b*) transparent s'il y a une transition sur P. Les sorties Cd et Pd indiquent lorsque la phase du registre est

complétée. Cette structure est complexe, ce qui constitue le principal défaut des micropipelines. La boucle de rétroaction qui génère le signal de complétion peut rapidement devenir le goulot d'étranglement d'un étage.

2.5.3 MOUSETRAP (Singh)

Montek Singh reprend le micropipeline et propose une version améliorée : le Mousetrap (Singh et Nowick, 2007). Ce pipeline asynchrone présente une structure simple et épurée, constituée seulement d'éléments logiques standards. Tout d'abord, comme l'illustre la figure 2.14, les registres « Capture & Pass » ont été remplacés par des loquets transparents. Ensuite, le détecteur de complétion a été remplacé par une porte logique XNOR conventionnelle. Finalement, le Mousetrap intègre aussi un protocole de communication à deux phases, ce qui lui permet d'augmenter son débit tout en gardant une capacité de traitement de 100%.

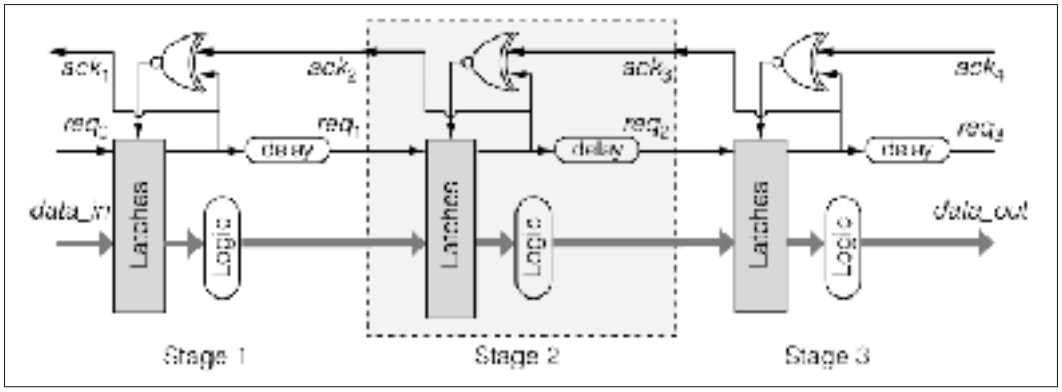


Figure 2.14 Pipeline asynchrone de type Mousetrap
Tirée de Nowick et Singh (2011)

Le protocole de communication demeure simple et résilient à la congestion. Chaque étage fonctionne d'une manière analogue à une trappe à souris, soit :

a. Initialement, le registre 1 est transparent, puisque $ack_1 = ack_2 = 0$;

b. Une requête $req_0 \uparrow$ arrive en phase avec les données qui passent à travers l'étage 1 : req_0 devient alors $ack_1 = 1$. Puisque et $ack_2 \neq ack_1$, le registre 1 devient opaque et met en mémoire les valeurs des données et de la requête à la sortie de celle-ci ;

c. Les données et ack_1 passent respectivement à travers la logique combinatoire et la ligne à délai : ack_1 devient ensuite $req_1 = 1$. Le registre 2 devient à son tour opaque et « trappe » les valeurs des données et de la requête ;

d. Puisque $ack_1 = ack_2 = 1$, le registre 1 redevient transparent. L'étage 1 du pipeline peut alors recevoir une autre requête, soit $req_0 \downarrow$.

2.5.4 Click elements (Peeters)

Plus récemment, Ad Peeters et l'équipe de Handshake Solutions ont repris un concept de pipeline asynchrone provenant d'un ancien brevet d'Intel (Traylor, 1995). Le pipeline asynchrone est similaire au Mousetrap, mais la principale différence réside dans l'utilisation de DFFs pour gérer le protocole de communication. Les contraintes temporelles des DFFs s'appliquent pour le contrôleur ainsi que pour le chemin de données. Une fonction logique simple gère le protocole de communication à deux phases et génère l'horloge qui capture les données à l'entrée de chaque étage du pipeline. La figure 2.15 illustre un pipeline de 3 étages basés sur les éléments Click (Peeters *et al.*, 2010).

Le contrôle du pipeline est basé sur une DFF de type « toggle », ou « toggle-flops ». Cet élément mémoire contient l'état présent, dénoté phase dans la figure 2.15. Deux conditions doivent être respectées pour accéder à un étage de pipeline. Premièrement, il doit y avoir une requête active pour l'étage en question. Cette condition dépend de la phase de l'étage précédent et peut être représentée par l'équation suivante :

$$\text{Requête}_\text{N} = N-1.\text{phase} \oplus N.\text{phase} \tag{2.3}$$

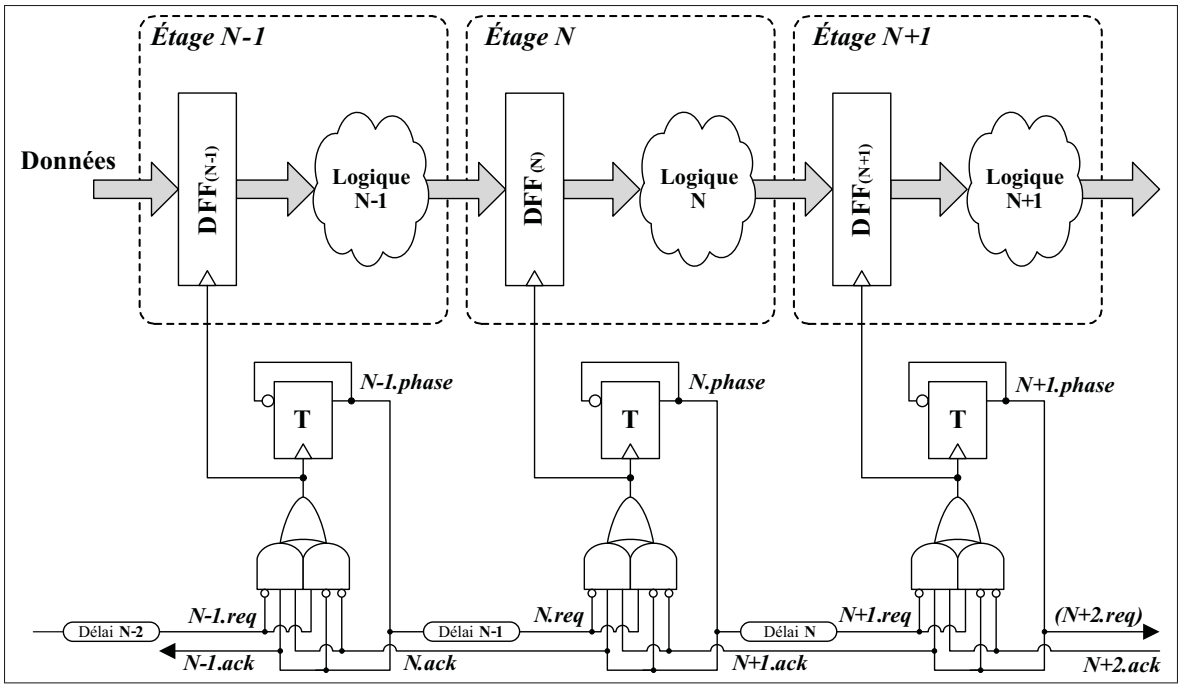


Figure 2.15 Pipeline asynchrone utilisant les éléments Click
Adaptée de Peeters *et al.* (2010)

Deuxièmement, l'étage sollicité doit être disponible et donc prêt à traiter de nouvelles données. Cette condition dépend de la phase de l'étage suivant et peut être représentée par l'équation suivante :

$$\text{Disponibilité}_{\text{N}} = \overline{N.\text{phase} \oplus N+1.\text{phase}} \tag{2.4}$$

Lorsque ces deux conditions sont atteintes, la fonction logique génère un front montant. Ce front montant capture les données à l'entrée de l'étage et inverse la phase de l'étage. La mise à jour de la phase se propage de l'entrée de la bascule T, appelée « toggle-flop », jusqu'à la fonction logique et invalide les conditions d'utilisations. Un front descendant est alors généré et termine l'opération d'activation de l'étage du pipeline. Un délai équivalent au temps de propagation maximal à travers la logique combinatoire doit être inséré au niveau du signal de requête ou de disponibilité. En outre, les éléments Click se prêtent bien à la conception de pipelines asynchrones non linéaires complexes. Différentes variations des éléments Click ont été développées (Peeters *et al.*, 2010) pour gérer la logique de contrôle de ces types de pipeline.

2.6 Conclusion

Plusieurs composantes d'un processeur, telles que les mémoires caches, implémentent une structure interne basée sur le pipeline. Cette méthode permet d'augmenter le débit d'un processus séquentiel en exploitant une forme structurelle de parallélisme. L'architecture d'un pipeline dépend intrinsèquement de la façon dont les étages sont synchronisés entre eux. Cette synchronisation peut être faite globalement (synchrone) ou localement, aux interfaces des étages (asynchrone).

Les pipelines classiques sont basés sur une architecture synchrone et implémentent une horloge globale pour cadencer l'exécution de chaque étage simultanément. Tandis que cette structure bénéficie d'un contrôle global relativement simple et performant. Cependant, certaines difficultés notables surviennent avec la miniaturisation constante de la taille des transistors et la complexité croissante des circuits. Ces problématiques sont principalement reliées aux interconnexions du réseau de distribution du signal d'horloge. Une alternative aux pipelines synchrones est les pipelines asynchrones. Ce type d'architecture bénéficie de l'absence du réseau de distribution global de l'horloge, ce qui permet de relaxer les contraintes temporelles. Ces contraintes sont plutôt appliquées localement à chaque étage de pipeline. Ainsi, les pipelines asynchrones possèdent une période d'horloge basée sur le délai moyen, contrairement au plus grand délai (« worst-case delay ») dans le cas des circuits synchrones. Il existe un intérêt important pour les pipelines asynchrones, étant donné que la mémoire cache du processeur ARM d'Octasic est actuellement basée sur un pipeline synchrone. Un des buts de ce travail est donc d'intégrer un pipeline asynchrone à la l'architecture de la nouvelle cache.

Dans ce chapitre, une vue d'ensemble de la conception et du fonctionnement des pipelines a été présentée. Deux types d'architectures ont été étudiées, soit les architectures synchrones et asynchrones. Leur avantage et désavantage respectifs ont d'abord été mis en évidence. Il a été mentionné que les difficultés grandissantes au niveau de la conception synchrone justifient l'intérêt pour une architecture asynchrone. Les éléments constituant les pipelines asynchrones ont ensuite été exposés. Finalement, quatre variantes notables de pipelines asynchrones ont

été présentées et étudiées. Fort de cette revue de littérature sur les pipelines asynchrones, il est maintenant possible de faire un choix technologique éclairé. La proposition d'un nouveau pipeline asynchrone pour la cache est basée sur une des variantes exposées dans ce chapitre.

CHAPITRE 3

ARCHITECTURE DU PROCESSEUR ARM ASYNCHRONE ET SA MÉMOIRE L1 SYNCHRONE

3.1 Introduction

Il est important de bien définir l'architecture du processeur asynchrone ARM d'Octasic. En effet, les mémoires d'instructions et de données sont considérées comme des ressources qui sont partagées au sein du CPU. Dans un premier temps, il est donc primordial de comprendre le fonctionnement du partage et de l'arbitrage de ces ressources.

Dans un deuxième temps, il est important de définir l'architecture et l'organisation logique de la mémoire cache L1 actuelle. La cache L1 d'instructions asynchrone devra avoir les mêmes caractéristiques que son équivalent synchrone pour pouvoir les comparer adéquatement. En développant d'abord la cache L1 asynchrone en fonction de ces spécifications, il sera alors possible de l'interfacer correctement au CPU. Les performances des deux types de cache pourront ensuite être évaluées et comparées.

Dans ce chapitre, l'architecture asynchrone des processeurs d'Octasic est d'abord introduite. Le partitionnement des tâches du processeur en fonction des unités d'exécution (XU) est alors exposé. Puis, la synchronisation et le partage des ressources sont expliqués ainsi que le concept de jetons. Ensuite, une exploration de l'architecture de la mémoire cache L1 synchrone est réalisée. Finalement, l'organisation et le fonctionnement du pipeline de la cache L1 d'instruction sont représentés et expliqués.

3.2 Méthodologie de développement asynchrone d'Octasic

Cette section du chapitre explore et définit la méthodologie de développement asynchrone d'Octasic. Dans un premier temps, il est primordial de comprendre le fonctionnement du CPU pour savoir comment y interfacer la mémoire cache. Le processeur asynchrone ARM développé

utilise une mémoire cache L1 synchrone, qui fera office de comparaison de base. La mémoire cache L1 asynchrone développée doit être implantée avec le même processeur pour qu'elle puisse ensuite être comparée convenablement.

Dans un deuxième temps, le mécanisme d'arbitrage et de synchronisation développé peut se révéler utile pour le développement de la nouvelle mémoire cache asynchrone. La définition du concept de jeton est d'ailleurs primordiale afin de comprendre l'architecture asynchrone d'Octasic. Les jetons gèrent le protocole de communication ainsi que l'accès aux ressources fondamentales du CPU.

Cette section est divisée de la façon suivante. Tout d'abord, le pipeline classique d'un processeur est exposé et déconstruit. Ce partitionnement du pipeline définit alors les bases de l'architecture asynchrone d'Octasic. Le paradigme entourant les unités d'exécution devient alors l'élément central du processeur asynchrone. Puis, le concept de partage des ressources par l'échange de jeton est présenté. Le réseau d'interconnexion des unités d'exécution et l'ordonnancement des jetons sont alors caractérisés. Finalement, une étude du fonctionnement interne du module de gestion des jetons est réalisée.

3.2.1 Déconstruction du pipeline classique d'un processeur

Pour développer son architecture asynchrone, Octasic s'est d'abord basé sur les principes d'un pipeline synchrone traditionnel pour ensuite le déconstruire (Laurence, 2013). Afin d'obtenir une architecture performante, les processeurs synchrones exploitent le parallélisme au niveau des instructions (ILP) (Stallings, 2010). L'exécution est basée sur le partitionnement en plus petites tâches des opérations sur les données, telles que dictées par une instruction. Dans le cas d'une architecture RISC, l'exécution peut être résumée en sept étapes distinctes, comme il est mentionné à la section 1.2.1. Une instruction est d'abord lue à l'adresse du PC puis est décodée, pour ensuite dicter le fonctionnement de l'unité d'exécution.

La figure 3.1 illustre le pipeline synchrone d'un processeur RISC implémentant une forme d'ILP. Alors que les deux premiers étages de pipeline (IF) et (ID) sont toujours utilisés, l'opé-

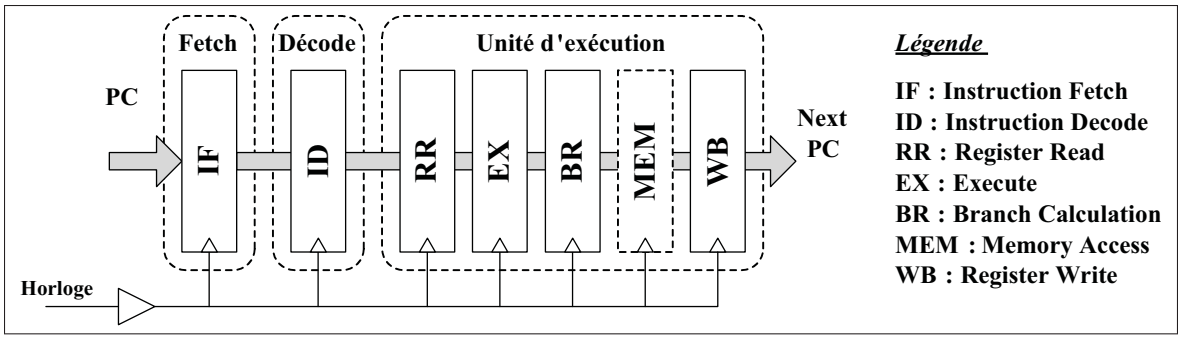


Figure 3.1 Unité d'exécution au sein d'un pipeline synchrone

ration effectuée par les étages de l'unité d'exécution varie en fonction de l'instruction. D'une part, seules les instructions *Load* et *Store* provoquent un accès mémoire en lecture ou en écriture à l'étage (MEM). Cela étant dit, la particularité d'une architecture processeur RISC réside dans son mode d'opération registre à registre. Les données provenant de la mémoire doivent d'abord être placées dans le fichier de registre avant de pouvoir être traitées. Ainsi, les opérations arithmétiques et logiques utilisent l'étage de lecture registre (RR) avant de traiter les opérandes à l'étage (EX). Le résultat est ensuite écrit dans le fichier de registre à l'étage (WB). D'une autre part, les instructions conditionnelles doivent définir si la condition est valide ou non. Le calcul de la prochaine adresse est fait à l'étage (BR) et le résultat de la condition détermine s'il y a branchement ou non pour la prochaine adresse du PC.

Il est possible de constater qu'une unité d'exécution synchrone n'est pas optimale pour deux raisons. Premièrement, les instructions n'accèdent pas nécessairement aux mêmes ressources, telles que les registres et la mémoire de données. Deuxièmement, les opérations effectuées par l'unité d'exécution varient en terme de complexité. Dans le cas d'un pipeline synchrone, le temps d'exécution est fixé en fonction de la plus longue opération, qu'elle requière le calcul d'une condition ou non. Idéalement, une unité d'exécution asynchrone permettrait d'ajuster la cadence des opérations, tel qu'illustré à la figure 3.2.

L'architecture des processeurs asynchrones d'Octasic est basée sur l'unité d'exécution (XU), telle qu'illustrée à la figure 3.3. Chaque unité accède seulement aux ressources dont il a besoin,

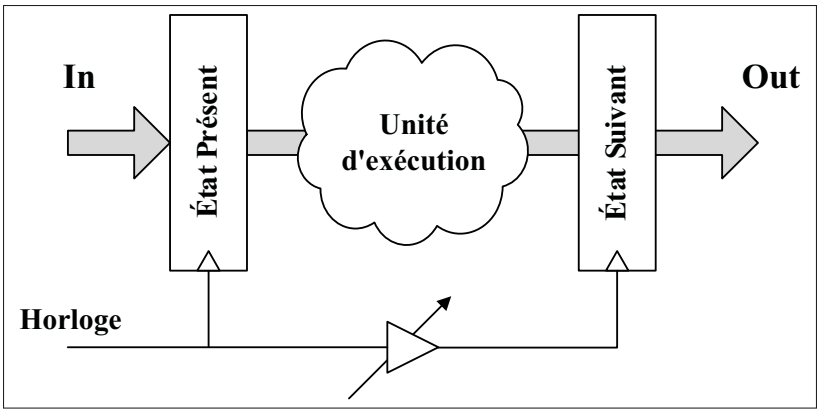


Figure 3.2 Unité d'exécution asynchrone

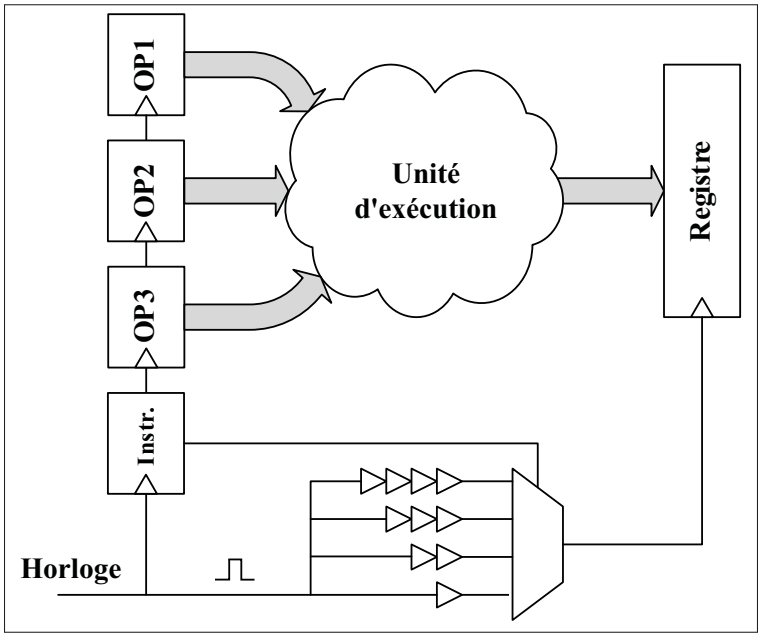


Figure 3.3 Architecture générale des unités d'exécution d'Octasic

et ce, en fonction de l'instruction exécutée. De plus, les XUs se basent sur l'instruction décodée pour déterminer le délai à utiliser lors de l'opération. Les opérandes **OP1**, **OP2** et **OP3** sont mises en mémoire à l'entrée et le résultat de l'opération est ensuite placé dans le fichier de registre à la sortie. Afin d'obtenir des performances équivalentes à un processeur synchrone,

plusieurs XUs sont utilisés en parallèle. Cette disposition est similaire aux architectures des processeurs multicœurs ou des réseaux-sur-puce (NoC).

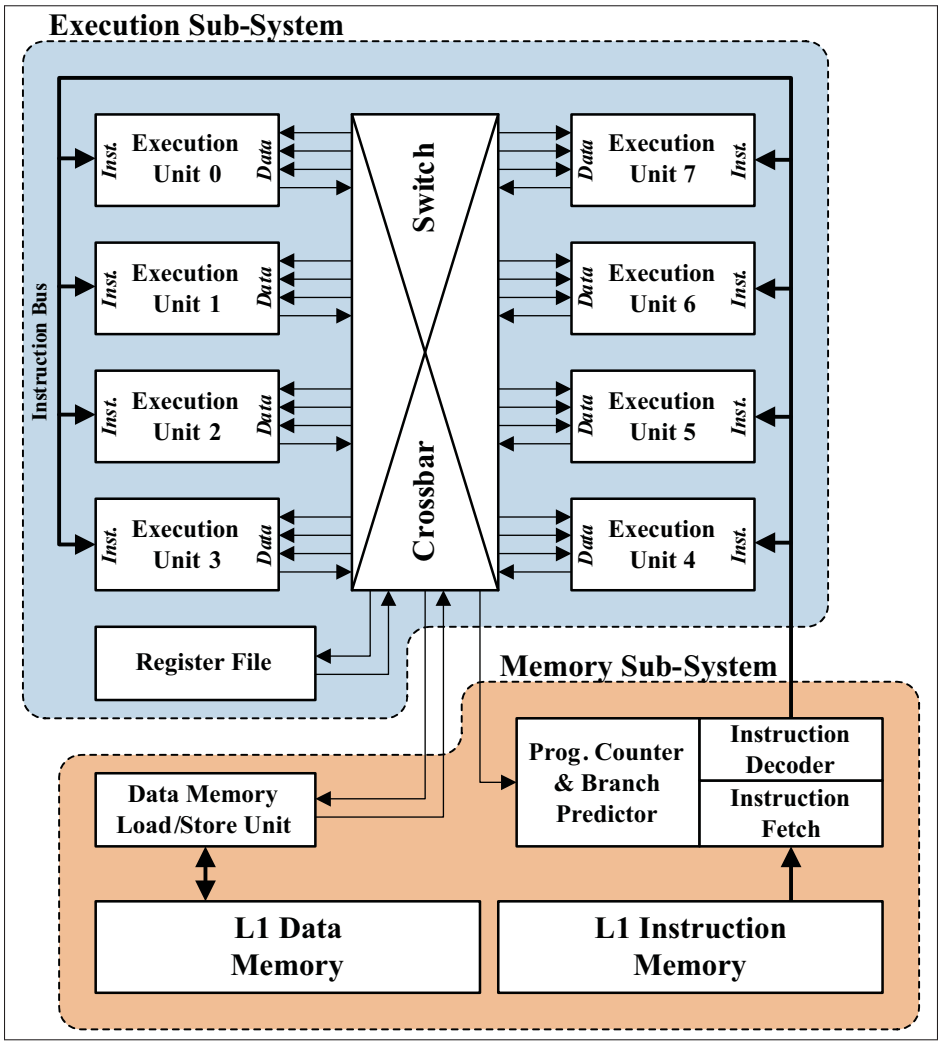


Figure 3.4 Architecture générale du processeur ARM d'Octasic

Le coeur du processeur d'Octasic est assemblé de la manière suivante, tel qu'illustré à la figure 3.4. Tout d'abord, chaque XU est connecté à un réseau d'interconnexion implémenté dans le commutateur (« crossbar ») non bloquant. Ainsi, les opérandes nécessaires à l'exécution proviennent soit du fichier de registre ou du résultat d'un autre XU. Cette méthode permet de gagner du temps en évitant d'écrire dans le fichier de registre avant de pouvoir utiliser une

donnée. Un contrôle est effectué pour déterminer si le XU doit accéder au « crossbar » ou au fichier de registre. Ensuite, chaque XU est relié à la mémoire de donnée, soit la cache de donnée L1, et peut y accéder en lecture ou en écriture. En effet, une architecture RISC nécessite que chaque donnée soit préalablement stockée dans un registre avant de pouvoir être utilisée. Puis, les XUs sont reliés à un bus d'instructions dédié, le CPU étant basé sur une architecture Harvard. Un module implémentant le compteur d'adresse PC et le prédicteur de branchement (« branch predictor », ou PCBP) est couplé à la cache d'instruction L1, qui est ensuite lié au décodeur d'instructions. Chaque requête à la mémoire d'instruction rapporte donc des instructions en fonction de l'adresse prédite par le PCBP. Ces instructions sont ensuite décodées, puis envoyées sur le bus d'instruction à l'unité d'exécution concernée. Il est à noter que les mémoires d'instruction et de données sont implémentées de façon synchrone par rapport aux XUs asynchrones. Ceci implique donc une pénalité de synchronisation lors de chaque accès mémoire.

3.2.2 Utilisation de jetons pour le partage des ressources

L'architecture processeur définie à la figure 3.4 établit les unités d'exécution comme étant le point central de traitement. Les XUs doivent accéder à un certain nombre de ressources pour obtenir les instructions et les données nécessaires au traitement. Le « crossbar », le fichier de registre, le compteur de l'adresse PC, les mémoires d'instruction et la mémoire de données sont définis comme étant les ressources des XUs. Étant donné que les XUs initient des transactions avec les ressources de leur environnement de manière asynchrone, un mécanisme de synchronisation doit être implémenté. Le mécanisme de synchronisation permet d'effectuer l'arbitrage entre chaque requête des XUs aux ressources spécifiques. Cela maintient la séquentialité des opérations et évite les conflits d'accès aux ressources.

Le mécanisme de contrôle des ressources est implémenté et distribué au sein de chaque XU. Des jetons d'utilisation des ressources (« tokens ») font office d'élément synchronisant les accès aux ressources. Lorsqu'un XU possède le jeton d'une ressource, ce XU peut y accéder si la ressource n'est pas occupée. Lorsque la ressource n'est plus utilisée par le XU, le jeton est passé

au XU suivant. Pour implémenter ce protocole de communication, le réseau d'interconnexion entre les XUs prend la forme d'un anneau. Ainsi le XU_n reçoit d'abord le jeton du XU_{n-1} et transmet ensuite le jeton au XU_{n+1}. Le fonctionnement du protocole de communication est illustré à la figure 3.5.

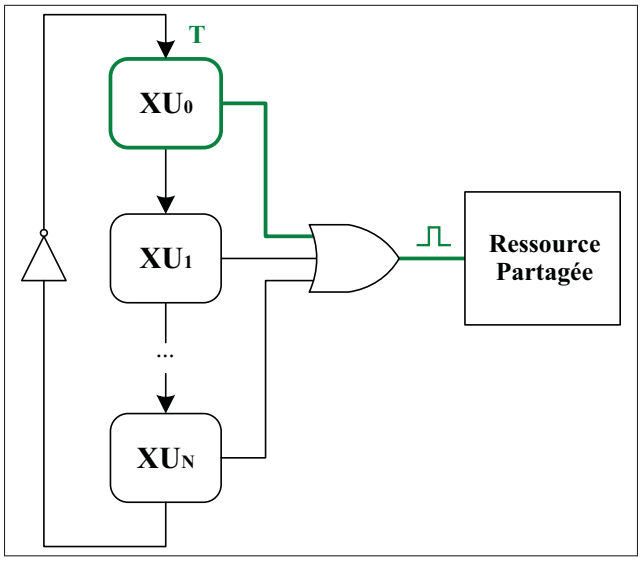


Figure 3.5 Synchronisation de l'accès aux ressources avec les jetons

Les XUs du processeur asynchrone possèdent un minimum de 6 jetons de ressources. Le premier jeton « Instruction Fetch » (IF) permet d'aller chercher une instruction décodée provenant de la cache L1. Le second jeton « Register Read » (RR) lit les opérandes à partir du fichier de registres ou du « crossbar ». Le troisième jeton « Execution » (EX) effectue l'opération provenant de l'instruction au sein de l'unité d'exécution. Il est à noter que ce jeton génère une impulsion qui, en fonction de l'instruction décodée, est retardée proportionnellement au temps d'exécution de la tâche. Le quatrième jeton « Branch Condition » (BR) calcule la prochaine adresse du PC en fonction de la validité de la condition de l'instruction. Ce jeton est seulement nécessaire lors de l'exécution d'une instruction conditionnelle par les XUs. Le cinquième jeton « Data Memory » (MEM) permet d'accéder la mémoire de donnée en lecture ou en écriture. Ce jeton est nécessaire pour effectuer les transferts de données avec la mémoire et le fichier de

registres du CPU. Le sixième jeton « Register Write » (WB) écrit le résultat de l'opération ou de la lecture en mémoire dans le fichier de registres.

Les unités d'exécution opèrent selon une séquence bien précise. Dans le cas du processeur ARM asynchrone, 16 XUs sont placés en série. L'ordonnancement des XUs en fonction des jetons est illustré à la figure 3.6. Lors de l'initialisation, l'unité d'exécution XU_0 possède tous les jetons des ressources. Dès qu'il termine l'utilisation d'un jeton et de sa ressource, il le transmet au prochain XU. Le fonctionnement est similaire à une course à relais, où le bâton est le jeton qui fait office de synchronisation et de signal de départ pour les coureurs. Cette architecture asynchrone permet de fonctionner à la vitesse naturelle des opérations, contrairement aux architectures synchrones où l'opération la plus longue définit la vitesse d'opération globale (Laurence, 2012). De plus, un XU qui ne nécessite pas une ressource peuvent directement passer le jeton au XU suivant. Dans le cas d'un pipeline synchrone, l'horloge de l'étage qui n'est pas nécessité peut être suspendue, mais il faut toutefois attendre le temps d'une période d'horloge.

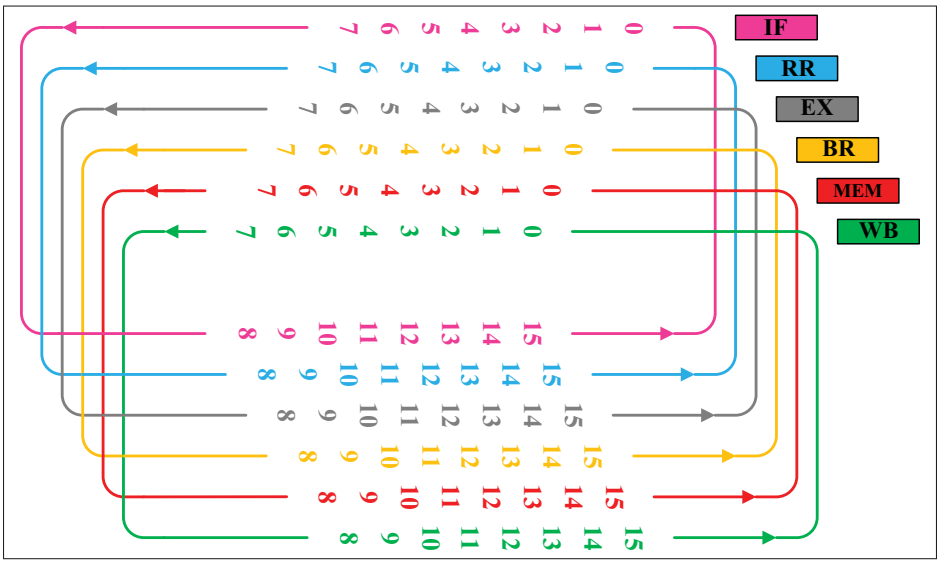


Figure 3.6 Ordonnancement des jetons au sein des XUs
Adaptée de Laurence (2012)

Les jetons permettent aussi de gérer les événements particuliers survenant habituellement dans les pipelines synchrones. Dans un premier lieu, un blocage du pipeline peut survenir lorsqu'une ressource nécessite plus de temps que prévu pour traiter une requête. L'arbitrage fait au sein des modules de contrôle des jetons gère nativement ces événements. En effet, un jeton peut être transmis d'un XU au XU subséquent, toutefois, ce dernier doit attendre que la transaction avec la ressource soit complétée avant d'entamer sa propre opération. Dans un second lieu, une des principales difficultés associées à l'opération d'un pipeline est la dépendance de données et la préséance de celles-ci dans les étages du pipeline. Le système de jetons traite ces événements de la même manière que les blocages. En effet, il est possible d'attendre la complétion du traitement effectué par un XU sur une donnée en question. Durant ce temps d'attente, le XU ne consomme pas d'énergie dynamique. De plus, les données peuvent être accédées rapidement via le réseau d'interconnexion « crossbar » dès qu'elles sont prêtes. Ceci évite au premier XU d'écrire la donnée dans le fichier de registre puis au second XU d'avoir à la relire par la suite. Dans un dernier lieu, lorsque le PCBP effectue une mauvaise prédiction de la prochaine adresse du PC, cela entraîne une purge du pipeline, dénommée « flush ». Afin d'éviter de corrompre la mémoire de données, toutes les unités d'exécutions ainsi que les ressources doivent être réinitialisées. Lors de ce type d'événement, tous les XUs sont réinitialisés et tous les jetons sont assignés au XU_0. Ainsi, le premier XU peut repartir à la bonne adresse PC et exécuter la bonne section du programme.

3.2.3 Fonctionnement interne des modules des jetons

Le module de gestion des jetons est divisé en deux sous-parties. La première sous-partie gère l'arbitrage et le passage du jeton. La seconde sous-partie génère le signal d'activation des données et l'impulsion qui fait office de signal d'horloge dans la ressource. Ces modules de gestion sont distribués localement près des unités d'exécution. Chaque XU du processeur asynchrone possède donc 6 modules de gestion des jetons, tel que mentionné à la section 3.2.2.

La première sous-partie du module de gestion est donc consacrée à l'arbitrage des jetons, illustré à la figure 3.7, et se nomme « token_edge_gen ». Le circuit de ce module implémente trois

fonctions relatives au jeton, soit le posséder et retenir, l'utiliser en le traitant, et le transférer au prochain XU. La possession d'un jeton par un XU est détectée lorsque le niveau logique à la sortie du module est différent que le niveau à l'entrée. La détection et l'arbitrage du jeton sont faits à l'entrée du module aux portes NON-ET et NON-ET-B. Le jeton peut traverser cette fonction logique que lorsque toutes les conditions d'utilisation relatives à la ressource et à l'ordonnancement des jetons sont valides. Une transition s'effectue alors, faisant basculer l'état du loquet de type S-R de '0' à '1' ou l'inverse. Un loquet S-R est implémenté pour garder en mémoire l'état du jeton et sa logique est présentée au tableau 3.1. Ce type de loquet permet d'implémenter un protocole de communication asynchrone à deux phases.

Tableau 3.1 Fonctionnement du loquet S-R

$\overline{\text{Force1}}$ / $\overline{\text{Set}}$	$\overline{\text{Force0}}$ / $\overline{\text{Reset}}$	**Action**
0	0	Impossible (Erreur)
0	1	$Q = 1$
1	0	$Q = 0$
1	1	Aucun Changement

Lorsqu'un des signaux $\overline{Force1}$ ou $\overline{Set}$ est activé, la sortie du loquet $Q = 1$. Lorsqu'un des signaux $\overline{Force0}$ ou $\overline{Reset}$ est activé, la sortie du loquet $Q = 0$. Lorsque tous ces signaux sont à '1', la sortie Q ne change pas. Le loquet devient instable lorsque deux signaux de logique opposés sont mis à '0', ce qui ne devrait jamais se produire. Il est à noter que la logique des loquets S-R est très semblable aux bascules « Toggle » implémentées dans la logique. Lorsqu'un jeton est consommé, la transition traverse le loquet S-R et est transféré à la fois au module de génération d'impulsion et dans une chaîne à délai variable. Si le XU ne requiert pas la ressource associée au jeton, ce dernier est immédiatement transférée au prochain XU en passant par le multiplexeur à la sortie. Dans le cas contraire, le jeton est retenu par la chaîne de délai suffisamment longtemps pour s'assurer que l'accès à la ressource soit complété. Le jeton est alors transféré à la prochaine unité d'exécution.

La seconde sous-partie du module de gestion est consacrée à la génération des signaux nécessaires à l'utilisation de la ressource. Le circuit est représenté à la figure 3.8 et se nomme

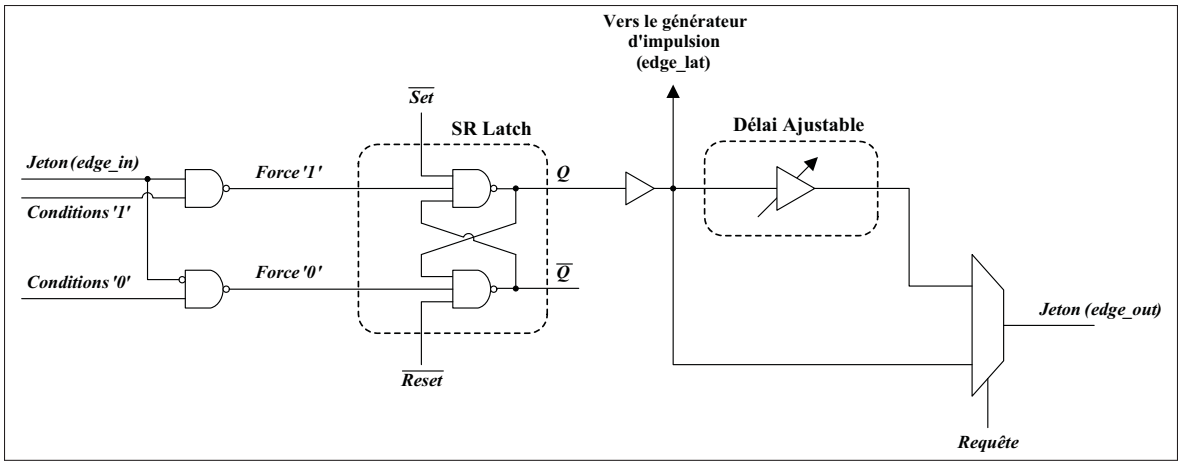


Figure 3.7 Module de gestion des jetons

« token_pulse_gen ». Le circuit implémente deux générateurs de signal de largeur ajustable. Le premier signal généré est le signal d'activation, ce qui est nécessaire pour effectuer le démultiplexage des données jusqu'à la ressource. Le second signal généré est l'impulsion, et celui-ci est utilisé comme signal d'horloge pour les éléments mémoires à l'entrée et à la sortie de la ressource.

La génération de signaux survient suite au passage d'une transition à travers le loquet S-R du module d'arbitrage. Lorsque le front du jeton *edge_in* passe par le loquet, le signal intermédiaire *edge_lat* est envoyé au module de génération de signaux. Pour générer le signal d'activation, ce front est d'abord inversé, puis retardé par deux chaînes de délai. La première chaîne a un délai fixe, tandis que la deuxième sert à ajuster le temps du délai avec une granularité fine. Le front initial *edge_lat* et le front délayé passent ensuite dans une porte logique ET, ce qui génère un signal d'une largeur équivalente au temps combinatoire de la chaîne à délai. Une logique antagonique est ajoutée pour permettre le traitement des fronts de jeton positifs et négatifs. Le signal résultant passe dans une seconde porte logique ET avec le signal de requête. Cette logique empêche la génération de signaux lorsque le jeton n'est pas utilisé par l'unité d'exécution et est directement transmis au prochain XU. Le signal final est issu de la porte logique OU, d'où provient un signal généré à partir d'un front montant ou descendant.

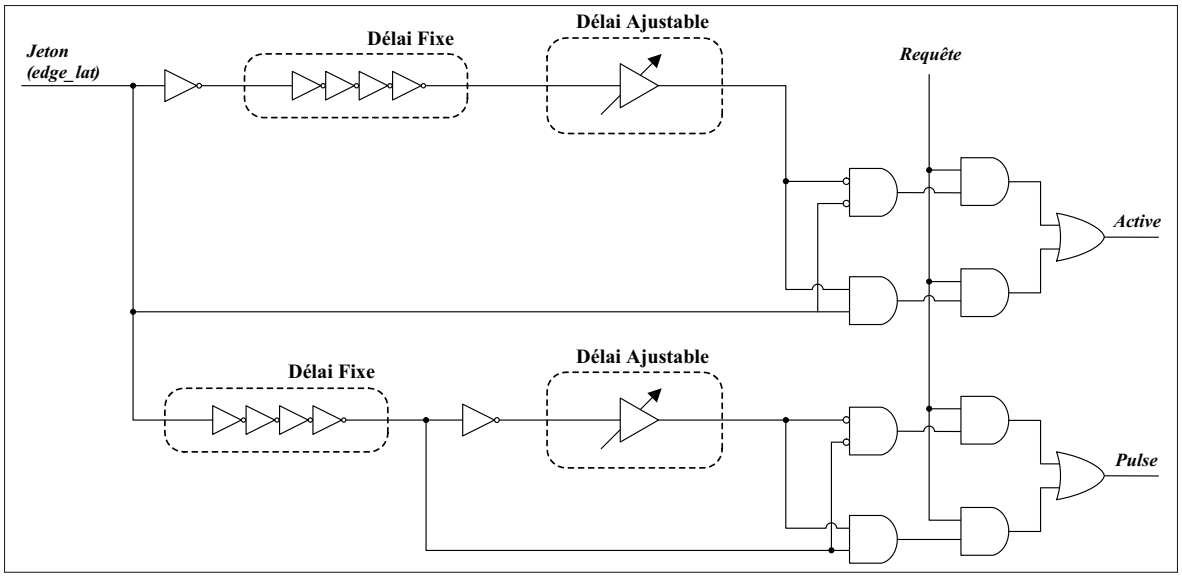


Figure 3.8 Module de génération du signal actif et de l'impulsion

Le circuit résultant est sensiblement le même pour le signal d'activation et de l'impulsion, à l'exception de la position de la chaîne de délai fixe. Pour l'impulsion, le délai fixe est utilisé sur *edge_lat* pour s'assurer que les données soient stabilisées avant de les mettre dans les éléments mémoire. Chaque jeton des ressources doit être ajusté suite à une analyse temporelle statique (STA), et ce pour tous les XUs.

3.3 Définition de la cache synchrone

Cette section du chapitre définit la cache L1 initialement développée pour le processeur asynchrone d'Octasic. La cache L1 est synchrone puisque l'architecture processeur d'Octasic synchronise les accès en mémoire à une horloge. La barrière de synchronisation est donc définie entre les unités d'exécution et la mémoire cache L1 telle qu'illustrée à la figure 3.4.

Cette première version de la cache L1 a été originalement conçue par Pascal Gervais et a ensuite été adaptée par Tom Awad afin d'être compatible avec l'architecture d'un processeur ARM asynchrone. Dans le cadre de cette maîtrise, l'architecture et l'organisation logique de la cache ont par la suite été optimisées, placées et validées en STA pour atteindre des perfor-

mances acceptables. Cette cache synchrone sert donc de base pour comparer les performances et l'efficacité énergétique avec la cache asynchrone.

Cette section est divisée de la façon suivante. Tout d'abord, l'organisation logique de la cache L1 et ses mécanismes de gestion sont décrits. Puis, l'architecture de la mémoire d'instruction et son pipeline sont présentés. Le fonctionnement de la cache ainsi que ses mécanismes de gestion interne sont caractérisés. Finalement, les interfaces de la cache d'instruction avec le processeur et le niveau supérieur de mémoire sont mises en perspective.

3.3.1 Organisation logique

Une série de choix technologique a été prise par l'équipe de conception afin de définir l'architecture mémoire du processeur ARM asynchrone. La mémoire cache a été conçue pour être aux normes de l'architecture ARM v7-A tout en demeurant performante. Son organisation logique reflète donc les balises énoncées par ARM dans le manuel de référence (ARM, 2010a).

La mémoire du processeur ARM développé par Octasic est basée sur une architecture Harvard. Elle intègre donc deux caches séparées et autonomes, soit une pour les instructions et une autre pour les données. Chaque cache a une taille totale de 32ko et son organisation logique est de type « set-associative ». Son contenu est divisé en 256 lignes de cache d'une taille de 32 octets, et ce, sur 4 voies différentes. La figure 3.9 illustre la structure logique de la mémoire cache.

La plage d'adresses disponibles est répartie sur 2^{32} bytes, soit 4Go, et est adressable par octet. Son espace mémoire contient donc 2^{32} octets (8 bits), ou 2^{31} demi-mots (16 bits), ou 2^{30} mots (32 bits), ou encore 2^{29} double-mots (64 bits). Pour supporter les accès non alignés aux données en mémoire et les multiplications ayant des opérandes de 32 bits, chaque requête mémoire accède à 64 bits d'information à la fois. Cette architecture permet à la mémoire d'instruction d'opérer en mode « dual-fetch », où deux instructions sont lues par accès mémoire.

La figure 3.10 illustre comment l'adresse demandée par le CPU est traduite en adresse physique pour la cache. L'adresse de 32 bits est partitionnée en 3 parties, soit l'étiquette, le sous-

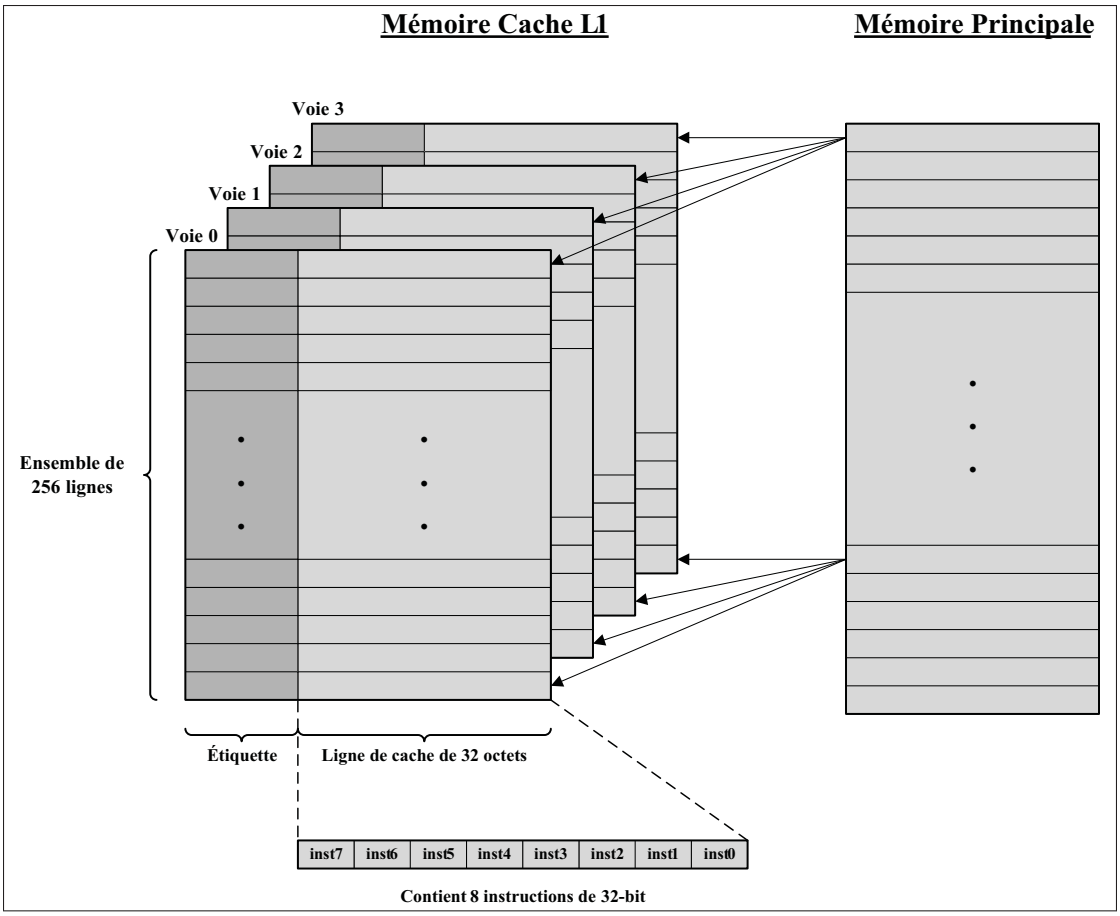


Figure 3.9 Organisation logique de la mémoire cache L1
Adaptée de Stallings (2010)

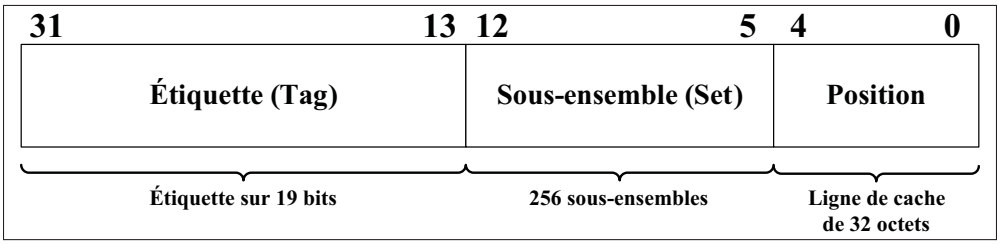


Figure 3.10 Adressage de la mémoire cache

ensemble (« set ») contenant les lignes de cache, et la position de l'information dans la ligne de cache. Les bits 31 à 13 constituent l'étiquette des données mises en cache. Il existe donc 2^{19} lignes de cache possibles pouvant être placé dans chaque sous-ensemble de la cache, pour

chaque voie. Une comparaison est faite entre cette étiquette et les étiquettes des quatre voies correspondant à l'adresse du bon sous-ensemble. Les bits 12 à 5 indiquent lequel des 256 sous-ensembles doit être vérifié pour trouver l'information. Les 4 à 0 déterminent quelle portion de la ligne de cache doit être lue ou écrite, puisqu'elle contient 32 octets d'information.

3.3.2 Gestion du contenu et de la cohérence

La gestion du contenu de la cache est maintenue grâce à un mécanisme d'allocation en lecture et en écriture. La stratégie d'allocation « read-write » signifie qu'une nouvelle ligne de cache est transférée de la cache L2 vers la cache L1 suite à un « cache miss » en lecture ou en écriture. La nouvelle ligne de cache envoyée par la cache L2 est ensuite placée dans une des quatre voies. Une stratégie de remplacement pseudo-aléatoire détermine dans quelle voie placer cette nouvelle ligne de cache. Deux bits d'un simple registre à décalage à rétroaction linéaire (LFSR) permettent d'implémenter cette fonction pseudo-aléatoire. Un bit indiquant la validité des lignes de cache n'a pas été implémenté avec les étiquettes. Une opération d'invalidation des étiquettes doit être effectuée avant de pouvoir utiliser les mémoires caches L1.

Pour maintenir la cohérence entre les informations contenues dans les caches L1 et L2, une stratégie de type *write-back* a seulement été implémentée dans la cache de données. L'état de chacune des 1024 lignes de cache, appelée « dirty bit », est stocké dans une petite mémoire. Un état "0" signifie que la ligne de cache n'a pas été modifiée et que l'information s'y retrouvant est cohérente avec celle contenue dans la cache L2. Un état "1" signifie que la ligne de cache a été modifiée lors une écriture et n'est plus cohérente avec celle contenue dans la cache L2. Lorsqu'une ligne de cache ayant été modifiée (« dirty ») doit être remplacée, elle doit préalablement être renvoyée à la cache L2 avant de pouvoir être évincée.

3.3.3 Architecture de la cache d'instruction L1

Dans le cadre de ce projet, seulement une version asynchrone de la cache d'instruction a été conçue. La structure de la cache d'instruction peut être réutilisée pour la cache de données en y

ajoutant les fonctions d'écriture et le module de gestion de la cohérence (« write-back »). Cela étant dit, seulement l'architecture de la cache L1 d'instruction est présentée.

L'architecture interne de la cache est divisée en trois blocs communs à la mémoire d'instructions et de données. On distingue le stockage des étiquettes, le pipeline et le stockage de l'information. Un bloc supplémentaire effectuant la procédure de *write-back* est ajouté à la cache de donnée.

L'ordinogramme à la figure 3.11 décrit le fonctionnement général de la mémoire cache L1.

Le stockage des étiquettes est implémenté avec trois mémoires RAM précompilées d'une taille de 256×32bits. Chacune des 256 adresses physiques des lignes de cache est référencée par les étiquettes de 19-bit des quatre voies, pour un total de 76 bits. La disposition des étiquettes dans les trois mémoires RAM est illustrée à la figure 3.12. Ainsi, une nouvelle requête à la cache utilise les bits 12 à 5 de cette adresse pour accéder aux quatre étiquettes. Ces étiquettes seront alors comparées avec l'étiquette de l'adresse demandée, soit les bits 31 à 13. Une correspondance indique un « cache hit » et dans le cas échéant, un « cache miss ». Il est à noter que cette configuration mémoire est sous-optimale puisque la troisième RAM n'utilise que 12 des 32 bits disponibles pour stocker la partie de la troisième étiquette.

Le pipeline de la cache est divisé en cinq étages de traitement, tel qu'illustré à la figure 3.13. À l'étage 0 (S0), la requête provenant du CPU est tout d'abord placée en mémoire dans les DFFs. La requête comprend une adresse 32 bits et une partie informative de 41 bits composée des différents signaux de contrôle nécessaires à l'accès en mémoire. L'adresse physique, soit les bits 12 à 5 de l'adresse demandée, est envoyée aux mémoires RAM contenant les étiquettes des quatre voies. À l'étage 1 (S1), la requête a été placée en mémoire dans les DFFs et le résultat de la lecture aux mémoires RAM des étiquettes est disponible. Les quatre étiquettes associées aux lignes de cache retournent donc des mémoires RAM jusqu'au pipeline. L'étiquette de l'adresse demandée, soit les bits 31 à 13, est alors comparée avec les étiquettes des quatre voies. S'il y a une correspondance, il y a « cache hit » et l'information demandée se retrouve à la ligne de cache de la voie en question. Dans le cas contraire, il y a « cache miss ».

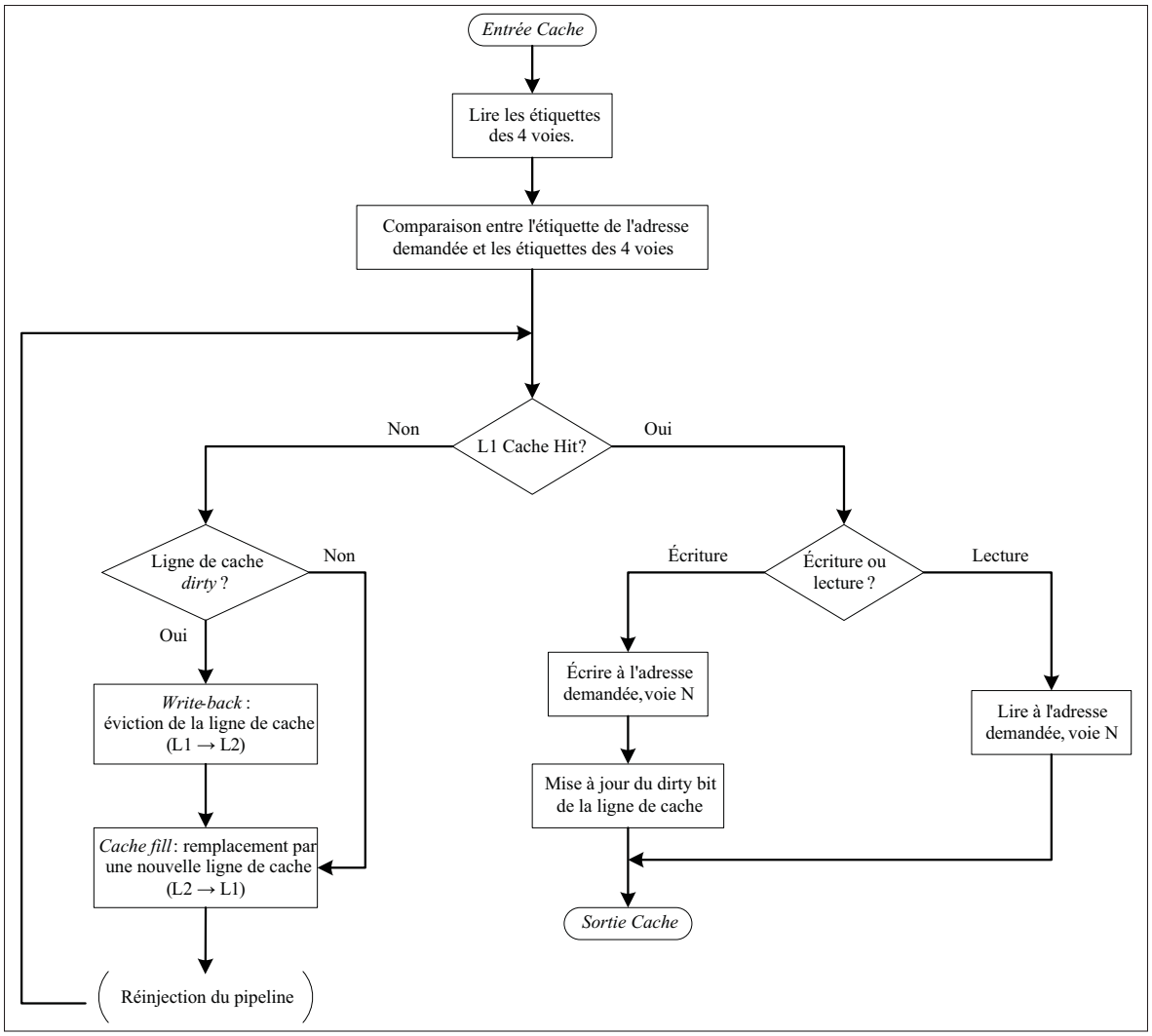


Figure 3.11 Fonctionnement général de la mémoire cache L1

À l'étage 2 (S2), le résultat de la comparaison de l'étage précédent détermine le fonctionnement de la cache. Lorsqu'un « cache miss » est détecté, une procédure de remplissage de la cache (« cache fill ») est entamée.Une requête pour le transfert de la ligne de cache manquante est alors envoyée à la mémoire cache L2. Le pipeline est bloqué durant le traitement de cette requête, jusqu'à ce que le transfert soit complété. Il est à noter que la structure de la cache actuelle implique que les deux accès, aux étages 0 et 1, soient réinjectés suite à une procédure de remplissage. Les limitations de la cache synchrone utilisée implique une pénalité de deux cycles d'opération. Une version améliorée pourrait facilement remédier à cette situation en

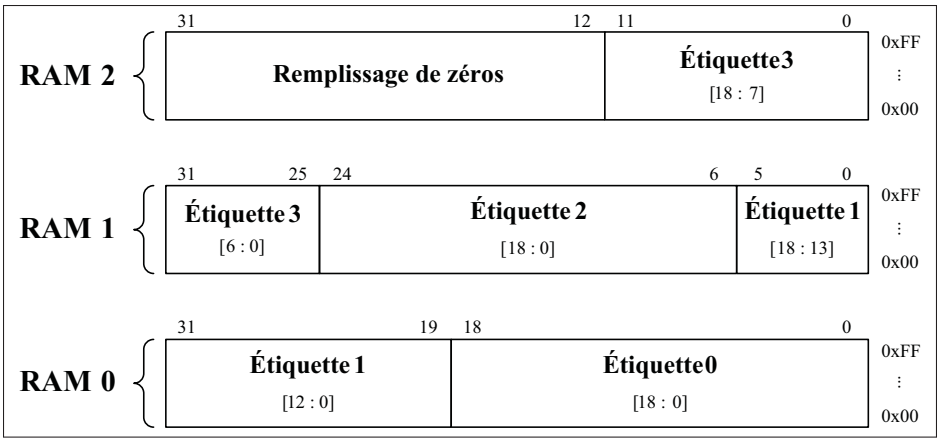


Figure 3.12 Disposition des étiquettes dans les mémoires RAM

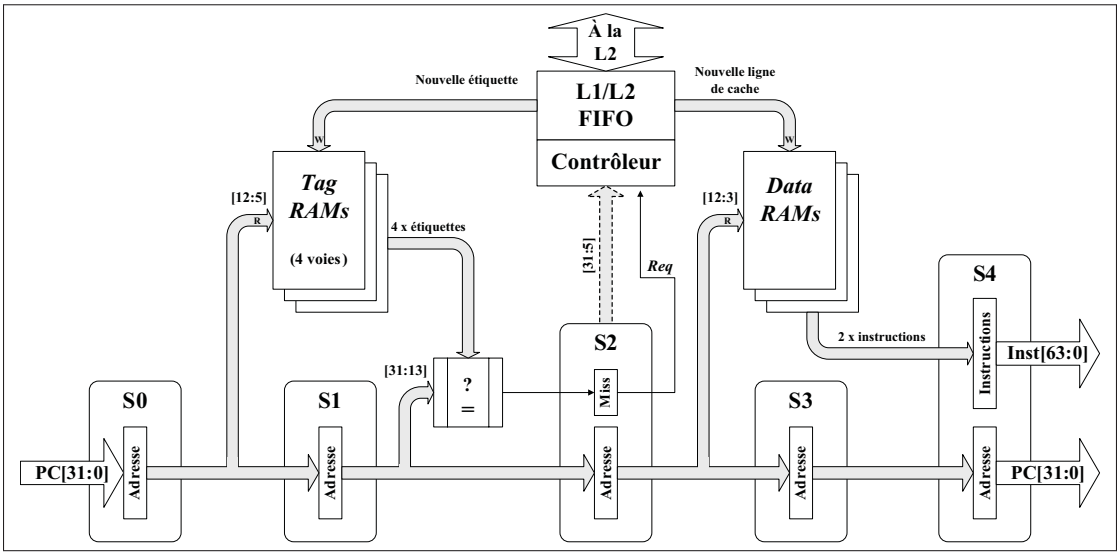


Figure 3.13 Pipeline synchrone de la cache d'instructions

implémentant une stratégie « read-after-miss ». Lorsqu'un « cache hit » est détecté, un accès aux mémoires RAM de données est entamé. La voie contenant la ligne de cache, son adresse physique et la position de l'information contenue permet d'adresser cet ensemble mémoire.

Le fonctionnement de l'étage 3 (S3) dépend aussi du résultat de la comparaison. Dans le cas d'une lecture suivant un « cache hit », le résultat de la lecture aux mémoires RAM de données

est disponible. Les deux instructions 32-bit retournent alors du réseau de mémoires RAM jusqu'au pipeline. Dans le cas d'une lecture suivant un « cache miss », les étages 3 et 4 ne sont pas activés pendant la procédure de remplissage. Des DFFs supplémentaires sont utilisées pour mémoriser le denier accès fait aux mémoires RAM de données. À l'étage 4, l'adresse demandée et les deux instructions de 32-bit sont envoyées à la sortie avec un signal de complétion *cache_rdone*.

Le stockage de l'information est fait dans la mémoire interne de la cache L1. Cette dernière est constituée de 32 mémoires RAM précompilées de $256 \times 32bits$, pour une mémoire totale de 32ko. Chaque voie contient 256 lignes de cache d'une taille de 32 octets et est distribuée verticalement. Puisque la mémoire cache retourne un double-mot de 64 bits, ou 8 octets, chaque voie est partitionnée sur 1024 adresses. Deux blocs de mémoire RAM appelés « rams1024x32 » contiennent chacun 4 mémoires RAM précompilées de 256x32bits. Les deux blocs RAM retournent respectivement le LSB et le MSB du double-mot de 64 bits. L'adressage physique de chaque ligne de cache utilise les bits 12 à 3 et la voie où il y a eu « cache hit », encodé sur deux bits. En somme, la mémoire interne de la cache « set-associative » à 4 voies est l'équivalent d'une mémoire d'une taille de $4096 \times 64bits$, soit 32ko adressés sur 12 bits.

La taille physique verticale de la mémoire interne est d'environ 700 μm. Conséquemment, chaque signal est transmis du pipeline jusqu'aux mémoires RAM grâce à une série d'inverseurs. Ceci minimise l'effort logique sur chaque porte logique tout en optimisant le temps d'accès. De plus, des lignes à délai sont ajoutées aux lignes de transmission pour balancer les temps d'accès entre chaque voie. Les lignes de transmission communes sont « clock-gated » [1] jusqu'aux entrées des mémoires RAM en fonction de la voie et des bits 4 à 3 de l'adresse. Enfin, une structure en arborescence de type « reduced-OR » est placée en sortie. Cette structure permet de réduire la puissance dynamique dissipée sur les lignes de retour en limitant la commutation sur ces dernières. Il est à noter qu'il est primordial de restreindre le nombre de routes physiques entre les mémoires RAM et le pipeline. En effet, puisque les mémoires RAM

1. Le « clock-gating » est implémenté pour masquer la transmission d'un signal en fonction d'une condition particulière.

sont conçues avec les quatre premières couches de métal, le routage est sensiblement diminué au-dessus de celles-ci.

3.3.4 Interfaces de la cache d'instruction L1

La mémoire d'instruction fait partie d'un module plus grand dénommé IFetch, qui signifie « instruction fetch ». Le module IFetch est divisé en quatre sous-parties, soit le générateur d'adresse, la cache d'instruction, le décodeur et le synchroniseur d'impulsion. Chaque sous-partie contient plusieurs sous-modules, qui sont décrits ci-dessous et illustrés à la figure 3.14. Le module IFetch lit en mémoire les prochaines instructions du programme, les décode et les transmet aux unités arithmétiques et logiques (XUs) pour être exécuté. Ce module fonctionne de manière synchrone à l'horloge de la mémoire cache L2.

Puisque les accès à la mémoire d'instruction sont alignés et « dual-fetch », un maximum de deux instructions 32-bit est lu par cycle d'horloge. Les espaces mémoires de 32bits peuvent aussi contenir soit deux instructions Thumb/ThumbEE 16-bit ou une instruction ARM complexe pouvant être divisée en deux instructions simples (ARM, 2010a). Un maximum de quatre instructions peut donc être transmis aux XUs en un coup d'horloge, une fois celles-ci décodées et étendues. Les instructions Jazelle (Java) 8-bit ne sont actuellement pas prises en compte par le processeur.

Le module IFetch, tel qu'illustré à la figure 3.15, s'interface entre les XUs, le coprocesseur CP-14 et la mémoire cache L2. Le coprocesseur CP-14 s'interface au module IFetch afin d'intégrer des fonctions de débogage telles que le mode pas-à-pas (« halt/step ») et l'insertion d'instructions. Le processeur opère alors en mode « Debug ». En mode normal, la requête d'instruction provient du XU pair $2n$ possédant le jeton *imem_pulse*. Seuls les XUs pairs peuvent avoir accès à la mémoire d'instruction puisqu'elle est « dual-fetch ». Le module IFetch retourne les instructions décodées aux XUs concernés et envoie le jeton au XU pair $2(n+1)$ suivant. Il est important de spécifier qu'il n'existe pas de pénalité de synchronisation entre la mémoire d'instruction et les XUs, puisque l'on passe du domaine synchrone à asynchrone.

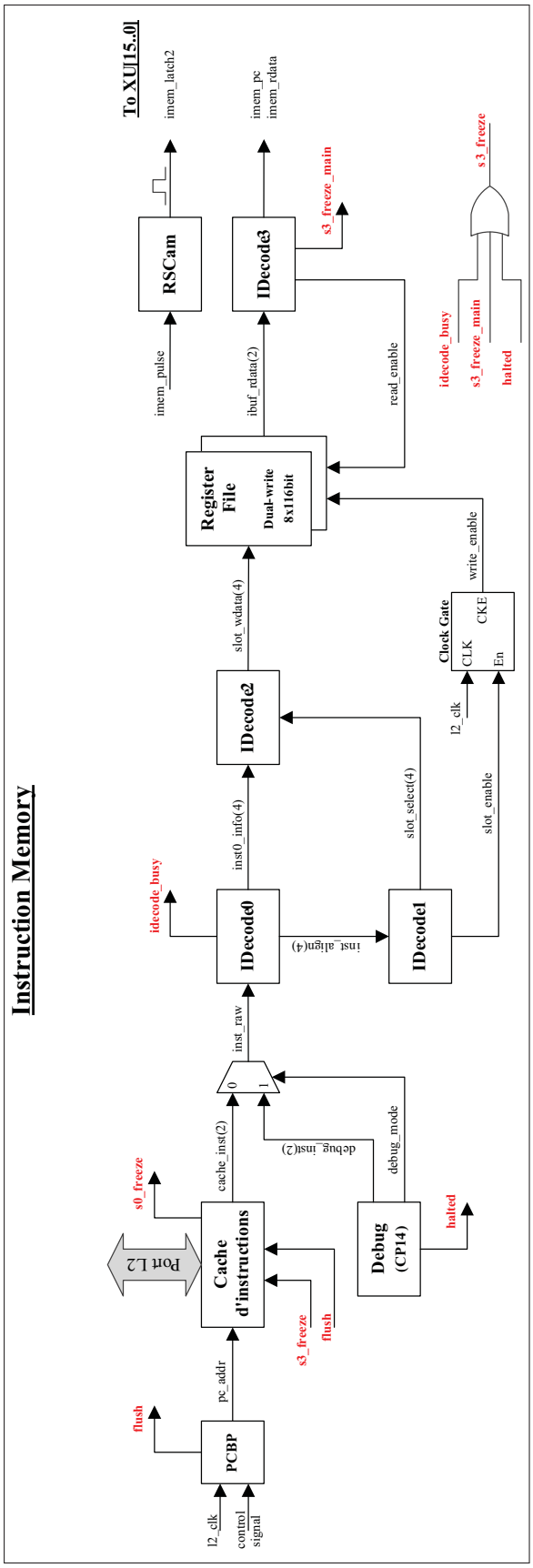


Figure 3.14 Diagramme bloc de la mémoire d'instruction

La génération d'adresse est effectuée au sein du module PCBP, qui signifie « Program Counter Branch Predictor ». En utilisant un historique et des méthodes statistiques, le PCBP prédit la prochaine adresse de l'instruction que le processeur exécute. L'adresse correspond soit à $PC + 8$, étant toujours en mode « dual-fetch », ou $PC + (8 \times index)$ si un branchement est prédit. La condition de branchement est testée ultérieurement par une des XUs. Lorsqu'une prédiction s'avère fausse, un signal annonçant une purge du pipeline, appelée « flush », est envoyé à travers le processeur. Lorsqu'une purge est activée, toutes les instructions en cours d'exécution sont annulées et le PCBP se réinitialise à l'adresse mal prédite. Une attention particulière est portée au circuit global de jeton afin d'éviter de corrompre la mémoire de donnée en exécutant une instruction à une adresse mal prédite. Pour éviter ce problème, le jeton de la mémoire de données *dmem_pulse* est retenu par le XU tant et aussi longtemps que la condition de branchement n'a pas été résolue.

La cache d'instruction, dénommée *cache4w32k_imem*, prend l'adresse prédite par le PCBP et émet deux instructions non décodées. Ce module est synchrone à l'horloge de la cache L2. Certains événements peuvent toutefois bloquer le pipeline interne en activant le signal bloquant *s0_freeze*. Ce dernier peut être activé lors de trois situations différentes. Premièrement, dans le cas où il y a un « cache miss » et par la suite lorsqu'un transfert d'information est effectué entre la cache L1 et la cache L2. Deuxièmement, lorsque le décodeur est occupé à étendre une instruction ou lorsque son tampon mémoire ne peut plus recevoir d'instructions. Troisièmement, lorsque le coprocesseur CP-14 halte le processeur et bloque la mémoire d'instruction. Un autre type d'événement peut bloquer le pipeline momentanément. En effet, le signal *flush* purge les étages du pipeline et retourne celui-ci à un état stable, moyennant une latence de 6 coups d'horloge.

Afin d'assurer une séquentialité des accès à la mémoire d'instructions, le module RSCam a été implémenté. RSCam est un module de synchronisation des impulsions et synchronise les accès en mémoire d'instruction. Chaque accès en mémoire d'instruction détient un temps horodaté en fonction de leur temps d'arrivée. Lorsqu'un XU pair accède la mémoire d'instruction,

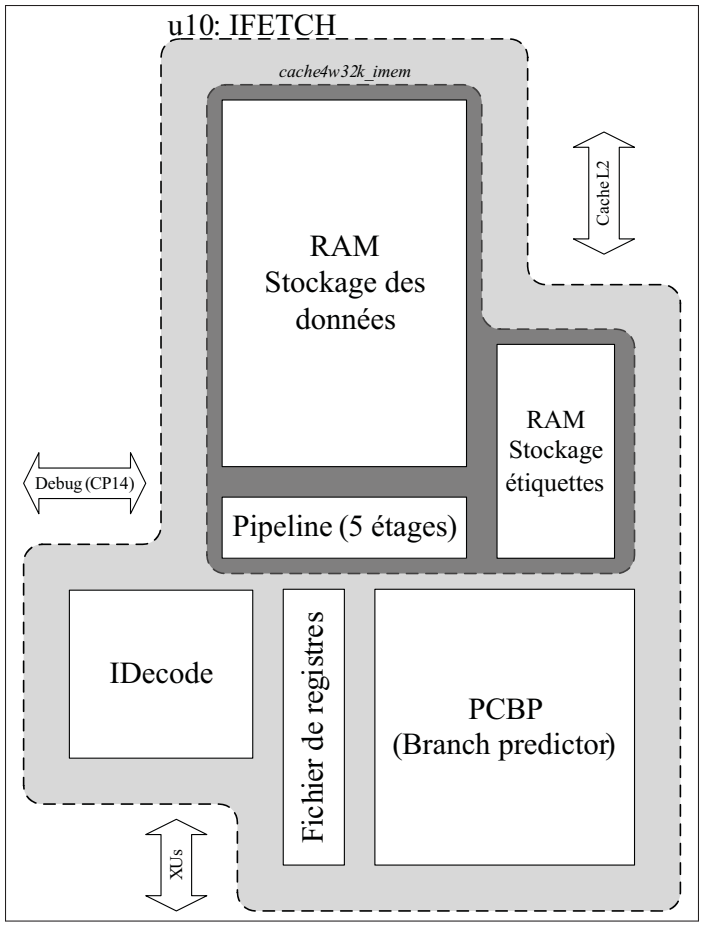


Figure 3.15 Disposition du module IFetch

l'impulsion *imem_pulse* est archivée dans RSCam. Lorsque deux instructions sont prêtes et décodées, ces dernières sont envoyées aux XUs avec le signal de complétion *imem_latch2*.

Une interface existe entre les caches L1 et L2 pour permettre le transfert de l'information des niveaux supérieurs de mémoire à la cache L1. Lorsqu'une instruction ou une donnée n'est pas présente dans la cache L1, un « cache miss » est détecté. La cache L2 est alors interrogée puis retourne la nouvelle ligne de cache contenant cette information. Cette opération de remplacement, nommée « cache line fill » implique que l'on évince une ligne de cache existante. Pour la cache de données, un mécanisme de cohérence de type « write-back » est implémenté et maintien la cohérence entre les deux niveaux de mémoire. Le « dirty bit » de chaque ligne de

cache doit donc être vérifié avant de pouvoir l'évincer et la remplacer. Dans le cas de la cache d'instruction, ce mécanisme n'est pas nécessaire puisque les instructions peuvent seulement être lues.

Puisque le système mémoire L1 du processeur ARM asynchrone est basé sur une architecture Harvard, il existe deux ports d'accès séparés à la cache L2. La cache L2 est unifiée et gère donc ces deux ports en parallèle. Le port d'accès entre la cache L2 et la cache d'instruction est unidirectionnel tandis que celui avec la cache de donnée est bidirectionnel. La largeur du port est de 64bits, en fonction de l'adressage des mémoires RAM de la cache L1.

Lors du remplacement d'une ligne de cache, la cache L2 envoie la nouvelle étiquette avec la ligne de cache contenant l'information manquante à la cache L1. Lors d'un « write-back », la cache L1 envoie d'abord la ligne de cache évincée, puis procède au remplacement de celle-ci. L'échange d'une ligne de cache s'effectue au minimum en cinq cycles d'horloge. Le temps de réponse de la cache L2 est variable, puisqu'elle peut avoir besoin d'accéder à un niveau supérieur de la mémoire. Le délai d'accès à la cache L2, ou N_{L2}, est présentement fixé à un seul cycle. La pénalité de temps engendrée par un « cache miss » sans « write-back » est $8 + N_{L2}$ cycles. La pénalité engendrée par un *cache miss* avec *write-back* est plus coûteuse, soit $12 + (2 \times N_{L2})$ cycles.

Le décodage d'instruction est géré par cinq sous-modules. Le premier sous-module est IDecode0. Ce sous-module décode les deux instructions 32-bit provenant de la cache en vecteur d'information de 116bits pour les XUs. Si une instruction doit être développée pour en générer une seconde, le signal *idecode_busy* est levé, ce qui qui suspend momentanément les accès mémoire. IDecode0 peut transmettre jusqu'à quatre instructions décodées de 116bits à la fois. Ce dernier transmet aussi un alignement préliminaire au module IDecode1.

Le second sous-module est IDecode1. Ce sous-module gère l'alignement des instructions en fonction des XUs paires et impaires. De plus, IDecode1 génère les signaux d'activation en écriture « wena » pour le fichier de registre. Le troisième sous-module est IDecode2. Avant de pouvoir écrire les instructions décodées dans le registre de fichier, celles-ci doivent être

alignées. IDecode2 multiplexe donc le vecteur d'information de 116bits en fonction de l'alignement généré par IDecode1.

Le quatrième sous-module est le fichier de registres. Ce dernier contient 8 entrées pour les XUs pairs et 8 pour les XUs impairs. Jusqu'à quatre entrées peuvent être écrites en même temps dans ce fichier de registres, mais seulement deux peuvent être lues à la fois. Le cinquième et dernier sous-module est IDecode3. Ce sous-module gère l'accès en lecture du registre de fichier grâce au signal d'activation en lecture « rena ». IDecode3 transmet aux XUs concernés la prochaine adresse « imem_pc » et les deux instructions à exécuter « imem_rdata ». IDecode3 génère aussi le signal « s3_freeze_main » lorsque le registre de fichier est presque plein, ce qui suspend les accès en mémoire jusqu'à ce que d'autres instructions soient envoyées.

3.4 Conclusion

Afin de pouvoir correctement interfacer la nouvelle mémoire cache proposée, une étude approfondie du processeur ARM d'Octasic a dû être faite. D'une part, il est primordial de bien définir l'architecture asynchrone pour déterminer comment y interfacer la mémoire cache. En effet, les mémoires d'instructions et de données sont considérées comme des ressources qui sont partagées au sein du CPU. Un système de jetons a été développé par Octasic pour gérer l'arbitrage et le partage des ressources.

D'autre part, l'architecture et l'organisation logique de la mémoire cache L1 synchrone actuelle doivent être clairement définies. En effet, la nouvelle cache asynchrone doit intégrer une structure logique équivalente pour pouvoir la comparer avec cette dernière. De plus, l'analyse faite aura permis de définir les modules nécessaires pour l'interface correctement au CPU. Les performances des caches synchrone et asynchrone pourront ensuite être évaluées et comparées.

Dans ce chapitre, l'architecture asynchrone des processeurs d'Octasic a d'abord été présentée. Le fonctionnement des unités d'exécution et leur interaction avec les ressources ont ainsi été étudiés. Puis, les principes asynchrones qui définissent les jetons comme médium de synchronisation et d'ordonnancement ont ensuite été exposés. Finalement, une revue plus approfondie

de l'organisation logique et de l'architecture de la mémoire cache L1 a été réalisée. Le protocole de communication gouvernant les jetons pourra servir au développement du pipeline asynchrone de la nouvelle cache L1 asynchrone. De plus, la cache d'instruction proposée devra respecter les mêmes spécifications de conception que son équivalent synchrone afin de pouvoir les comparer adéquatement.

CHAPITRE 4

PROPOSITION D'UNE NOUVELLE CACHE ASYNCHRONE POUR LE PROCESSEUR ARM

4.1 Introduction

Les chapitres précédents ont permis de mettre en contexte la conception d'une cache L1 asynchrone performante et efficace en terme d'énergie. En premier lieu, le fonctionnement d'une mémoire cache et son rôle au sein d'un processeur ont été énoncés. Une revue des techniques permettant d'optimiser une mémoire cache pour la performance et pour l'efficacité énergétique a également été faite. Ensuite, dans le but d'intégrer une structure asynchrone à la nouvelle cache L1, les bases de la conception asynchrone ainsi que des modèles de pipelines asynchrones ont été présentées. Enfin, une vue d'ensemble a été réalisée sur la cache d'instruction L1 actuelle et son intégration dans le processeur ARM asynchrone d'Octasic.

Ceci a d'abord permis d'exposer et de comprendre la problématique, pour ensuite établir l'état de l'art. Il a ainsi été possible d'énoncer clairement les requis du projet afin d'orienter le travail de conception. Ce quatrième chapitre élabore donc la conception de la nouvelle cache d'instruction L1 asynchrone proposée pour le processeur ARM.

Dans ce chapitre, la structure et le fonctionnement du nouveau pipeline asynchrone sont d'abord introduits. Puis, l'intégration du pipeline asynchrone au sein de la cache d'instruction et son fonctionnement complet est présentée. Ensuite, les procédures particulières telles que la purge ou la réinitialisation du pipeline sont expliquées. Finalement, chacune des interfaces entre la cache d'instruction et son environnement est présentée.

4.2 Définition du pipeline asynchrone

Cette section permet de définir techniquement comment a été conçu le pipeline asynchrone de la cache L1 proposée. Ce travail est issu d'un processus de compréhension qui a été façonné par

les rencontres avec Tom Awad et Doug Morrissey. À ce jour, le pipeline asynchrone développé est en constante évolution afin de l'améliorer et d'éprouver son fonctionnement.

Avant même d'entamer le travail exploratoire pour la conception du pipeline asynchrone, certaines exigences ont dû être considérées afin de rester fidèle au flot de conception d'Octasic. Premièrement, il est nécessaire d'utiliser des librairies de cellules CMOS standard pour faciliter la migration d'un fabricant de circuits intégrés à un autre. Deuxièmement, il est primordial d'utiliser seulement des DFFs comme élément mémoire, ce qui simplifie le STA et le DFT. Troisièmement, l'utilisation de cellules ayant une tension de seuil élevée doit être priorisée afin de limiter la consommation d'énergie statique du circuit développé.

Ces différentes exigences permettent de définir le type de pipeline asynchrone parmi ceux présentés à la section 2.5 a été utilisé. Tout d'abord, le PS0 est basé sur une logique dynamique, ce qui n'est pas possible avec la première exigence de développement du pipeline asynchrone. De plus, ce type de pipeline doit utiliser des espaceurs d'étage, ce qui résulte en un pipeline peu performant. Ensuite, le micropipeline n'utilise pas de cellules logiques dynamiques, mais implémente plutôt des éléments mémoires différents des DFFs. Ceci entre en contradiction avec la seconde exigence de conception du pipeline. Le Moustrap, quant à lui, est effectivement une solution performante. Cependant, ce type de pipeline implémente des loquets transparents, ce qui ne répond pas non plus à la deuxième exigence énoncée.

Finalement, les éléments Click constituent le type de pipeline asynchrone le plus prometteur pour la nouvelle cache asynchrone. La solution proposée à la figure 4.1 résulte en effet d'une combinaison des éléments Click et du système de jetons d'Octasic. Alors que le protocole de communication à deux phases est basé sur les éléments Click, la synchronisation est assurée par les jetons.

Les caractéristiques du pipeline asynchrone sont les suivantes. Tout d'abord, les éléments Click permettent de stocker facilement l'état de chaque étage du pipeline à l'aide d'une seule DFF. Cela permet d'implémenter un protocole de communication simple et robuste qui utilise le même signal d'horloge que les données qui sont traitées à l'étage. En utilisant des « toggle

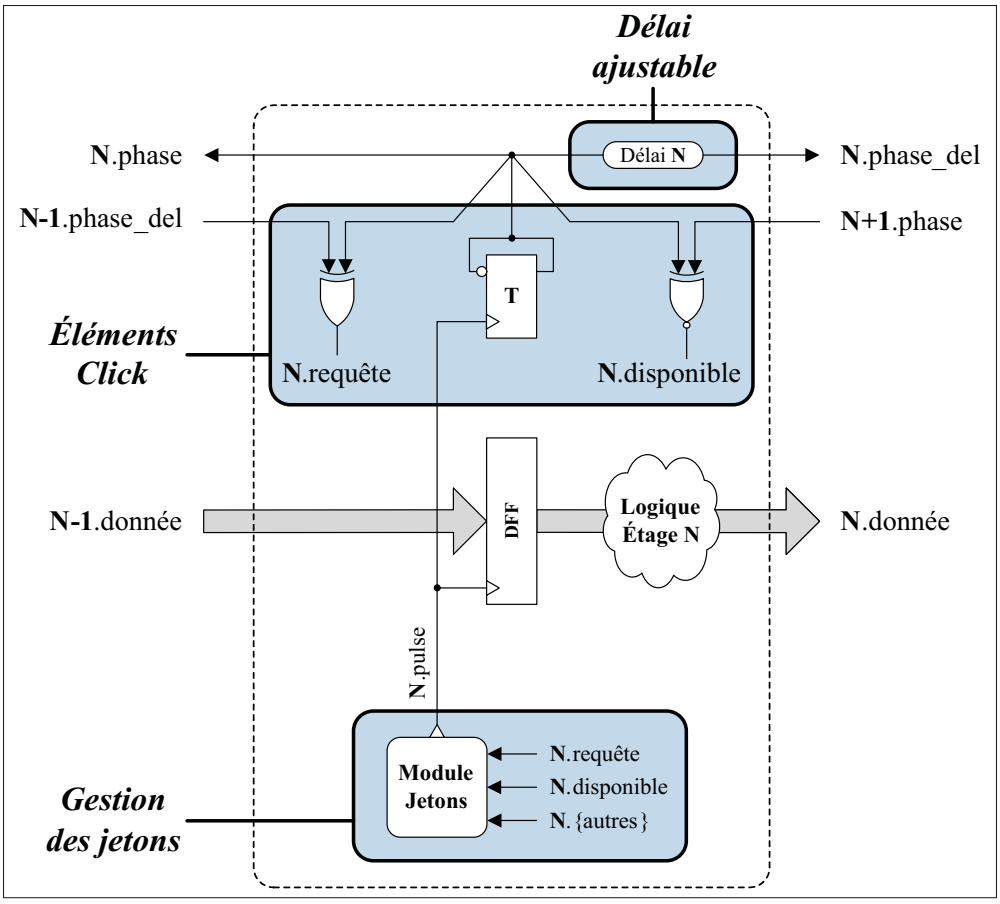


Figure 4.1 Illustration d'un étage du pipeline asynchrone

flops », il est possible d'implémenter un protocole à deux phases basé sur les transitions, appelé « transition-based ». Le protocole à deux phases permet aussi de maintenir la capacité du pipeline à 100%, contrairement au protocole de communication à quatre phases.

Toutefois, la logique des éléments Click générant l'impulsion a été omise pour pouvoir intégrer les modules de jetons. Les jetons gèrent donc l'arbitrage, la synchronisation et la génération du signal d'horloge pour chaque étage de pipeline. Le module de gestion des jetons utilisé par les XUs du processeur a été légèrement modifié et épuré pour être intégré à chaque étage du pipeline.

4.2.1 Gestion du protocole de communication par les éléments Click

Les éléments Click, tels que définis à la section 2.5.4, ont été modifiés pour intégrer un contrôle simple et robuste de la phase de chaque étage du pipeline. La figure 4.2 illustre le contrôle effectué par les éléments Click pour un étage de pipeline. La « toggle-flop » de l'élément Click, qui est en fait constituée d'une DFF et d'un inverseur entre la sortie Q et l'entrée D, conserve la phase (ou l'état) de l'étage du pipeline. Dès qu'un étage du pipeline est utilisé, une impulsion est envoyée à l'élément Click, ce qui met à jour sa phase.

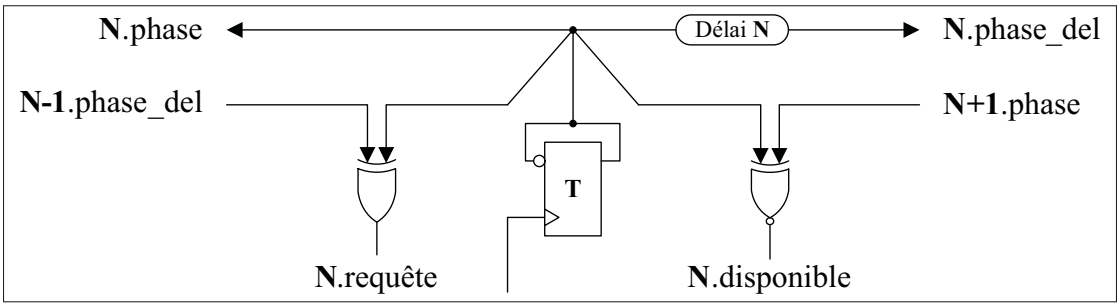


Figure 4.2 Contrôle par les éléments Click pour un étage de pipeline

La fonction logique des éléments Click générant l'impulsion est remplacée par le module de gestion des jetons. Pour remplacer cette fonction logique, deux portes logiques XOR et XNOR ont été utilisées pour effectuer la comparaison des phases de chaque étage du pipeline. En comparant les phases de chaque étage du pipeline, il est possible de dériver les deux états nécessaires au protocole de communication à deux phases, soit le signal *requête* (« request ») et le signal *disponible* (« available »). Ces deux signaux sont ensuite utilisés comme contrôle au module de gestion des jetons.

Le protocole de communication à deux phases implémenté avec les éléments Click est défini de la façon suivante. Premièrement, une requête active existe à un étage de pipeline lorsque la phase délayée de l'étage précédent $N-1.phase\_del$ est différente de la phase de l'étage

présent $N.phase$. Cette condition peut être représentée par l'équation suivante :

$$N.\text{requête} = N-1.\text{phase\_del} \oplus N.\text{phase} \tag{4.1}$$

Deuxièmement, l'étage présent est disponible et prêt à traiter de nouvelles données lorsque la phase de l'étage suivant $N+1.phase$ est la même que la phase de l'étage présent $N.phase$. Cette condition peut être représentée par l'équation suivante :

$$N.\text{disponible} = \overline{N.\text{phase} \oplus N+1.\text{phase}} \tag{4.2}$$

Ces deux signaux sont interprétés comme étant des conditions d'utilisation d'un étage de pipeline.

4.2.2 Arbitrage des étages du pipeline et génération du signal d'horloge par les jetons

Les modules de gestion des jetons ont été intégrés au sein du nouveau pipeline asynchrone afin de profiter de la méthodologie asynchrone développée et éprouvée par Octasic. Ainsi, le contrôleur de chaque étage du pipeline implémente les circuits d'arbitrage et de génération d'impulsion, tel qu'introduit à la section 3.2.2. La logique combinatoire de l'étage du pipeline est donc considérée comme étant la ressource partagée par le jeton. Puisqu'il n'y a qu'un contrôleur par étage de pipeline, il n'y a qu'un seul utilisateur par jeton.

Quelques modifications ont donc été apportées au paradigme de contrôle des jetons afin de l'adapter au pipeline asynchrone. En effet, le circuit de contrôle implémente seulement deux fonctions relatives au jeton, soit de le posséder et le retenir ou l'utiliser et le consommer. Il n'est pas possible de passer le jeton sans utiliser la ressource puisque cela annulerait la requête à l'étage de pipeline. La figure 4.3 illustre le module de gestion des jetons qui est utilisé par un étage du pipeline.

Le circuit de détection et d'arbitrage du jeton « token_edge_gen » de la figure 3.7 a donc été légèrement modifié. Chaque étage de pipeline est synchronisé et séquencé par rapport au précédent et au suivant en fonction de ses conditions d'utilisation. Ces conditions d'utilisation proviennent du protocole de communication à deux phases, soit les signaux *requête* et *disponible*. Les signaux $\overline{\text{Force0}}$ et $\overline{\text{Force1}}$ sont donc basés sur la requête, la disponibilité et d'autres conditions spécifiques à l'étage du pipeline. Les autres conditions d'utilisation tiennent compte du signal de réinitialisation $\overline{\text{Reset}}$ et de purge du pipeline $\overline{\text{Flush}}$. Le multiplexeur placé à la fin du circuit a aussi été retiré puisqu'il n'est pas possible de transférer directement le jeton sans effectuer le traitement de l'étage du pipeline.

Le circuit de génération des signaux d'activation et d'impulsion « token_pulse_gen » de la figure 3.8 a aussi légèrement été modifié. L'étage de logique, où les impulsions et le signal de requête passent par une porte logique ET, a été enlevé. En effet, chaque transition du jeton génère un signal d'horloge à l'étage du pipeline.

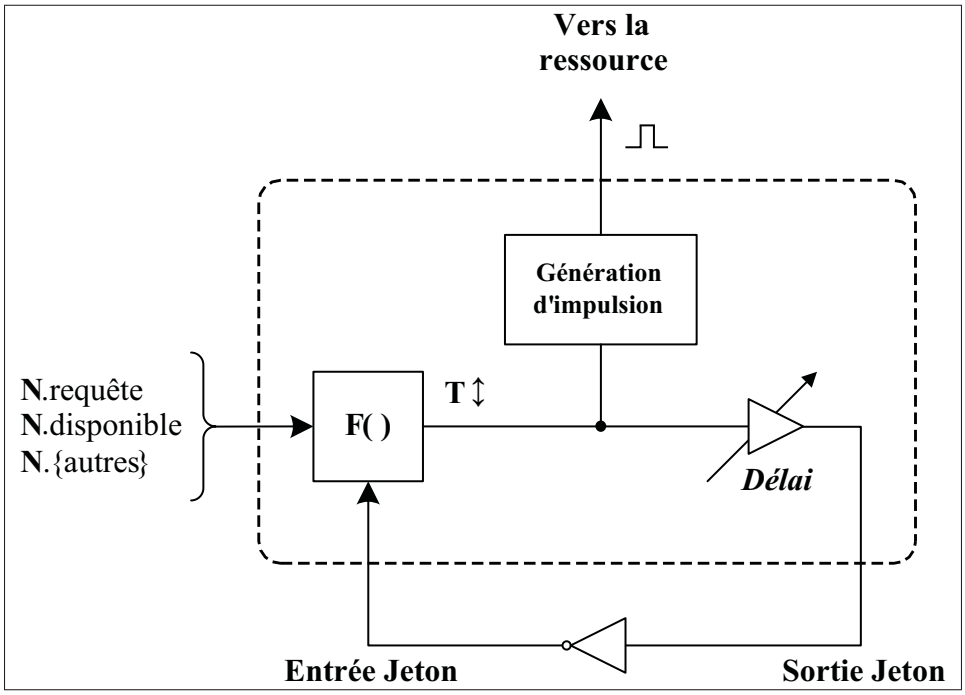


Figure 4.3 Module de gestion des jetons pour un étage du pipeline

Le fonctionnement du nouveau module de gestion des jetons est le suivant, et est illustré à la figure 4.4 pour un étage en particulier du pipeline asynchrone. Lors de l'initialisation, tous les modules possèdent leur jeton respectif. Afin d'activer les modules, les conditions d'utilisation de l'étage de pipeline doivent d'abord être satisfaites (∗*_use_cond*). Il doit y avoir une requête active à l'étage du pipeline (∗*_req*), ce dernier doit être disponible (∗*_avail*) et minimalement ne pas être en processus de réinitialisation ou de purge.

Lorsque ces conditions sont respectées, une transition (∗*_edge_out_px*) s'effectue à la sortie du loquet S-R du module « token_edge_gen », ce qui laisse passer le jeton. La transition du jeton génère alors une impulsion (∗*_pulse*) et un signal d'activation (∗*_active*) à la sortie du module « token_pulse_gen ». L'impulsion, qui sert de signal d'horloge, est transmise à l'élément Click pour changer la phase et aux DFFs de l'entrée de l'étage de pipeline pour mettre en mémoire les données. Finalement, le jeton (∗*_edge_out*) est retardé par la chaîne à délai ajustable avant d'être inversé puis redirigé à l'entrée du module (∗*_edge_in*). Le délai est déterminé en fonction du temps nécessaire pour que l'impulsion soit correctement desservie à toutes les DFFs de l'étage du pipeline.

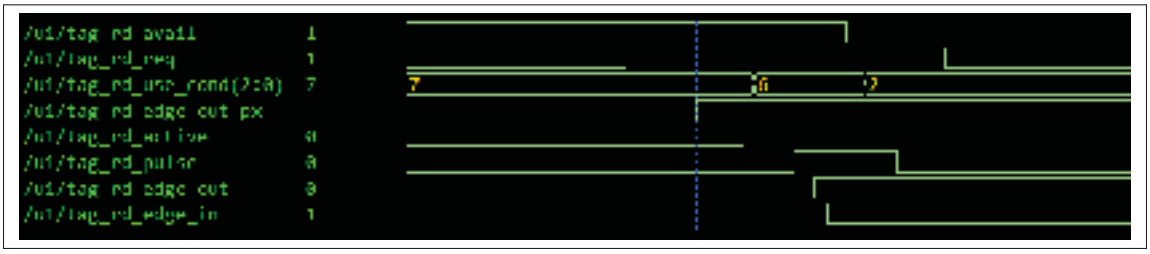


Figure 4.4 Séquence d'utilisation des jetons pour un étage du pipeline

4.2.3 Opération du pipeline asynchrone

Le fonctionnement du pipeline asynchrone est décrit dans cette section. Par souci de clarté, un pipeline scalaire de deux étages est d'abord illustré à la figure 4.5. L'étude d'un pipeline asynchrone linéaire simple permet de bien comprendre son opération.

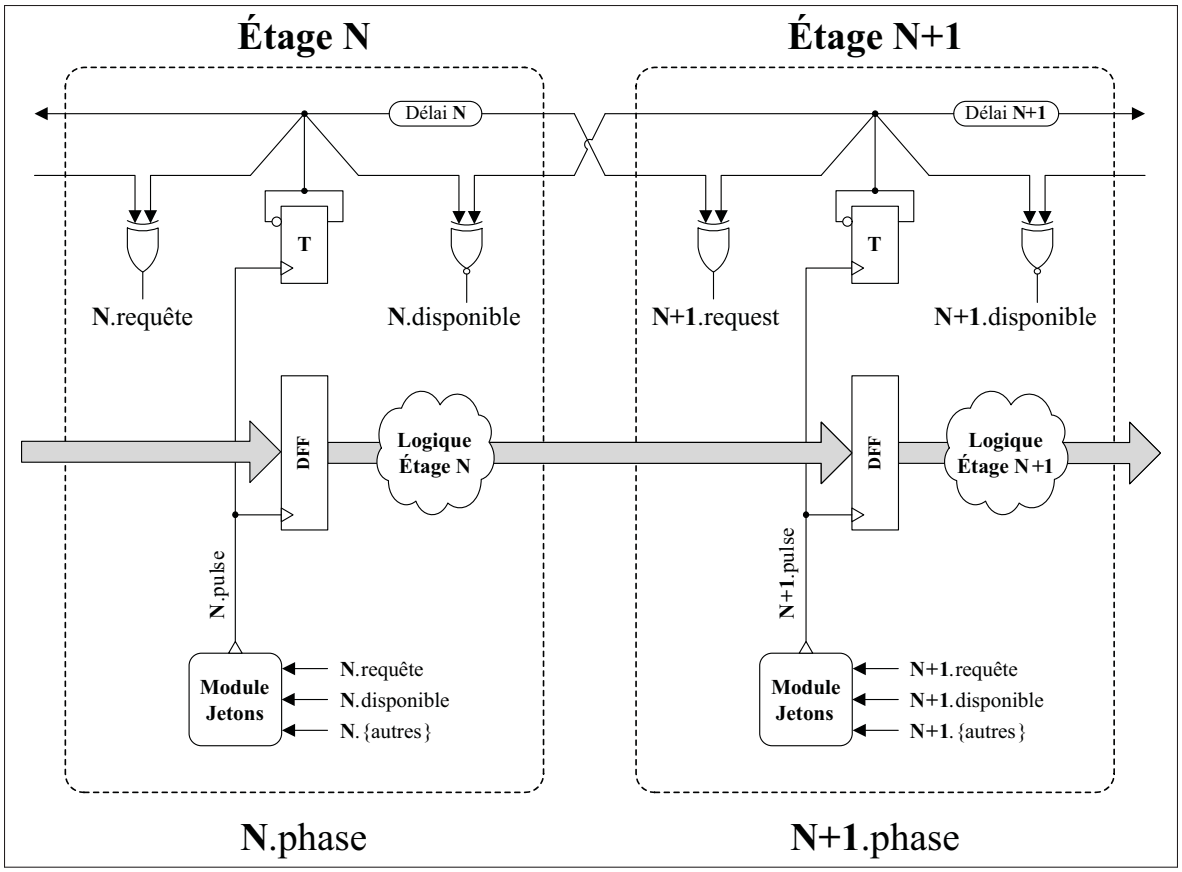


Figure 4.5 Exemple de pipeline asynchrone scalaire à deux étages

L'opération du nouveau pipeline asynchrone peut être résumée en 7 étapes globales. Tout d'abord, lors de l'initialisation ($\overline{\text{Reset}} = 0$) ou suite à une procédure de purge ($\overline{\text{Flush}} = 0$), la sortie des « toggle-flops » des éléments Click est mise à zéro. Toutes les phases des étages sont donc à zéro, soit $N.\text{phase} = 0$ et $N+1.\text{phase} = 0$. Les jetons aussi sont réinitialisés ; toute transition momentanée est annulée et la sortie du loquet S-R est mise à zéro. Ainsi, chaque étage du pipeline est disponible et prêt à traiter des données.

Puis, une nouvelle donnée est disponible à l'entrée de l'étage N, une requête $N.requête$ est donc active pour cet étage. Ainsi, toutes les conditions d'utilisation de l'étage N sont satisfaites. Alors, le loquet S-R du module de gestion des jetons génère une transition. Cette transition génère alors une impulsion et un signal d'activation, qui n'est pas représenté à la figure 4.5. Ces signaux permettent en premier lieu de mettre en mémoire la nouvelle donnée placée à l'entrée

de l'étage N. Dans un second lieu, l'impulsion met à jour la phase de l'étage N, devenant alors N.phase $= 1$. Le changement de phase à l'étage N désactive à la fois la requête $N.requête$ et la disponibilité $N.disponible$ de l'étage.

Par la suite, alors que le traitement des données est effectué à l'étage N du pipeline, le signal de la phase passe par une chaîne d'inverseurs ayant un délai proportionnel au à la logique combinatoire de l'étage. Ainsi, une requête à l'étage $N+1$ devient active dès que le traitement à l'étage N est terminé. À son tour, les conditions d'utilisations de l'étage $N+1$ sont satisfaites, ce qui déclenche le loquet S-R du module de gestion des jetons. Le passage du jeton provoque une transition qui génère le signal d'activation et l'impulsion servant de signal d'horloge de l'étage $N+1$. Ces deux signaux placent en mémoire les données issues de l'étage N précédent, tout en faisant basculer la phase de l'étage $N+1$, devenant alors $N+1$.phase $= 1$.

Immédiatement, la requête et la disponibilité de l'étage $N+1$ sont mises à zéro. En même temps, la condition d'utilisation de l'étage précédent $N.disponible$ redevient active puisque N.phase $= N+1$.phase. L'étage N peut donc traiter une prochaine requête pendant que l'étage $N+1$ est occupé. Le traitement est alors effectué à l'étage $N+1$. La phase délayée $N+1.phase\_del$ génère finalement une requête à la sortie du pipeline, indiquant que la sortie de l'étage contient une nouvelle donnée. Lorsque la donnée à la sortie de l'étage $N+1$ sera lue, la condition d'utilisation $N+1.disponible$ redeviendra active.

Alors que le pipeline asynchrone développé s'intègre facilement pour une configuration scalaire, il est possible d'y adapter des éléments complexes d'un pipeline non linéaire. En effet, les pipelines modernes incorporent des boucles de rétroaction afin de réaliser des tâches complexes. Pour réaliser cette fonction, un étage de pipeline implémente d'abord une fourche (« pipeline fork »), où le chemin de donnée est divisé par un démultiplexeur. Puis, un second étage implémente un joint (« pipeline join »), où le chemin de donnée est réuni par un multiplexeur.

L'utilisation des éléments Click pour la gestion du protocole de communication permet d'intégrer facilement ces deux composants. La figure 4.6 présente la fourche conçue pour le pipeline asynchrone. Lorsque l'étage contenant cette fourche est activé, les données sont stockées et la

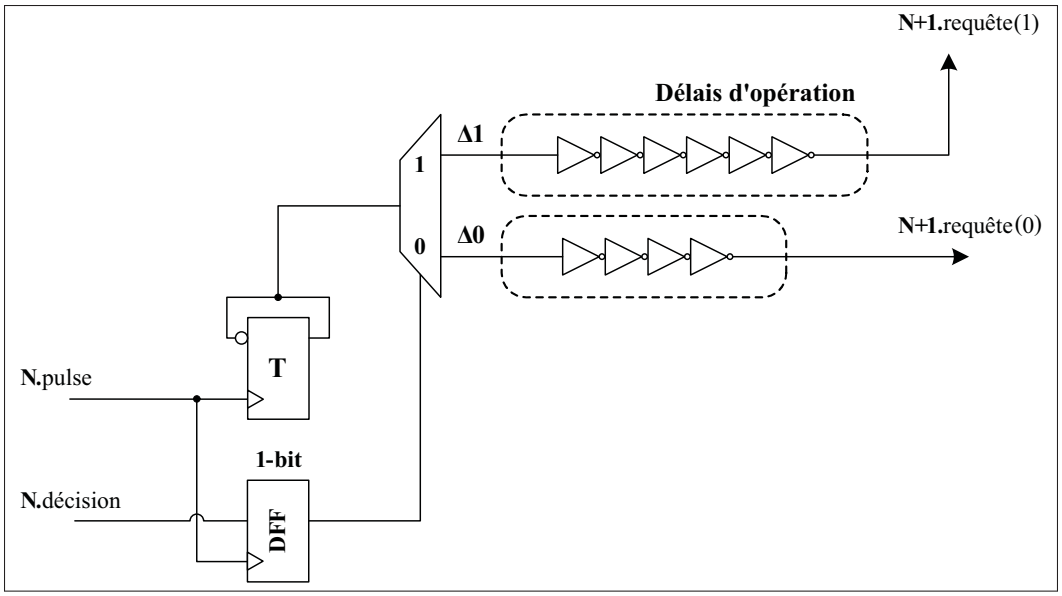


Figure 4.6 Représentation d'un étage intégrant une fourche de pipeline

phase est mise à jour. Une des données provenant de l'étage précédent, appelé *décision*, permet de choisir la voie empruntée par le chemin de donnée. Ainsi, la phase est délayée en fonction du temps combinatoire de cette voie du chemin de donnée. Le démultiplexeur détermine la chaîne de délai à utiliser en conséquence de cette décision pour délayer la phase de l'étage. Ainsi, seulement une requête est générée par le signal de phase, soit la voie empruntée par le chemin de donnée.

La figure 4.7 présente le joint conçu pour le pipeline asynchrone. Lorsque l'étage contenant ce joint est activé, les données sont stockées et la phase est mise à jour. L'état des requêtes de deux étages différents est alors mis en mémoire, ce qui détermine le traitement à effectuer à l'étage en question. Un multiplexeur sélectionne dans un premier temps la provenance des données en fonction d'une logique issue des requêtes. Puis, la phase de l'étage est délayée en fonction du temps combinatoire de la voie empruntée par le chemin de donnée. Ainsi, seulement une requête est envoyée au prochain étage du pipeline, soit la voie empruntée par le chemin de donnée.

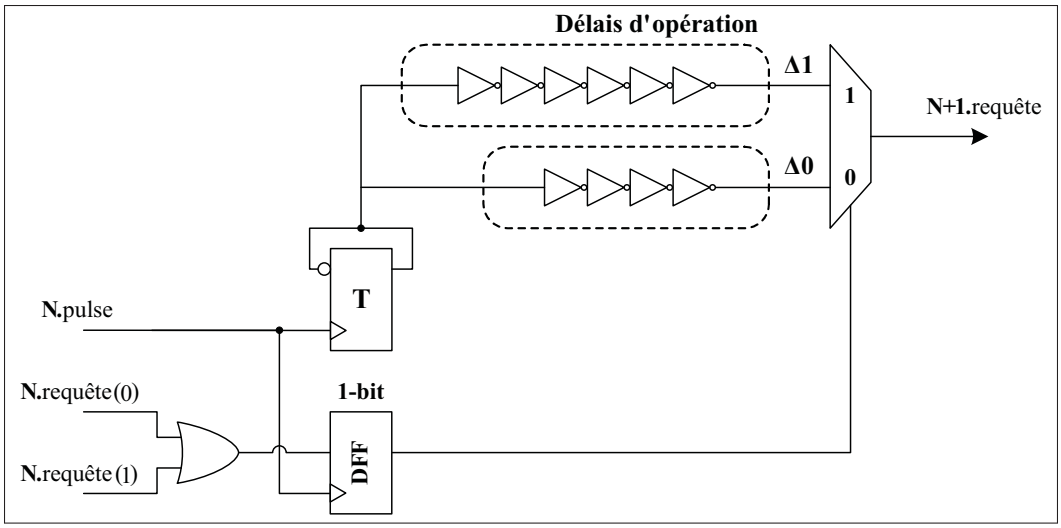


Figure 4.7 Représentation d'un étage intégrant un joint de pipeline

4.3 Définition de la cache L1 asynchrone

L'architecture de la cache L1 asynchrone proposée est présentée dans cette section. Comme il l'a été mentionné dans la section précédente, le but de ce travail est d'effectuer une comparaison entre un mode de fonctionnement synchrone et asynchrone. Ainsi, le but initial était de conserver la même organisation logique de la cache d'instruction afin de bénéficier d'une disposition identique du module, excepté son pipeline.

Les spécifications de la cache L1 d'instruction asynchrone demeurent donc les mêmes que son homologue synchrone, telles que décrites à la section 3.3.3. Le pipeline asynchrone développé gère donc l'architecture d'une cache déphasée « dual-fetch set-associative » à 4 voies d'une taille de 32ko. Chacune des quatre voies peut accéder à 256 lignes de cache d'une largeur de 32 octets. L'adressage est fait de la même façon que pour la cache synchrone et le contrôle logique effectué par la mémoire L2 aussi.

Globalement, la nouvelle cache asynchrone est basée sur quatre étages en « cache hit ». Seulement un étage est bloquant lors d'un « cache miss » et deux autres étages gèrent le remplacement de la ligne de cache. Cette procédure est effectuée de façon concurrente au traitement

normal des autres étages de la cache, ce qui est avantageux en terme de temps d'exécution. La cache L1 d'instruction asynchrone a été testée au sein du processeur ARM asynchrone afin de pouvoir exécuter des programmes de référence tels que Coremark et Dhrystone.

4.3.1 Partitionnement de la cache d'instruction en ressources

La première étape de conception de la nouvelle cache asynchrone consiste à déterminer comment l'information est traitée au sein de son pipeline. Puisque la synchronisation et la séquentialité du pipeline asynchrone sont gérées par les jetons, le contrôleur de chaque étage du pipeline devient l'équivalent d'une unité d'exécution. Cependant, le traitement effectué par chaque étage est *asymétrique*, c'est-à-dire que ces derniers ne peuvent pas utiliser toutes les ressources également. Un étage de pipeline détient donc un seul jeton, ce qui permet d'accéder seulement à la ressource qui lui est assignée. Par conséquent, il est primordial de définir dans un premier temps les ressources qui sont partagées au sein du pipeline, et dans un second temps, assigner des tâches aux différents étages en fonction de ces ressources.

La figure 4.8 présente le pipeline asynchrone développé en fonction des trois ressources partagées et des tâches qui lui sont associées. La première ressource est la mémoire des étiquettes, qui permet de déterminer si une information est présente ou non dans ce niveau de cache. La seconde ressource est la mémoire interne de données, contenant l'information disponible à ce niveau de cache sous forme de ligne de cache. Pour la cache d'instruction, ces deux ressources sont accessibles en lecture lors d'un « cache hit » et en écriture suite à un « cache miss ». La troisième ressource est le contrôleur à l'interface entre la cache L1 et le niveau de mémoire L2. Cette ressource est accédée suite à un « cache miss ».

Cinq tâches primaires sont d'abord assignées aux ressources préalablement définies. La première tâche est la lecture des étiquettes des quatre voies, qui est associée au jeton « Tag Read ». La seconde tâche est l'écriture de la nouvelle étiquette, qui est associée au jeton « Tag Write ». La troisième tâche est la lecture de l'information demandée à partir de la mémoire de données de la cache, qui est associée au jeton « Data Read ».

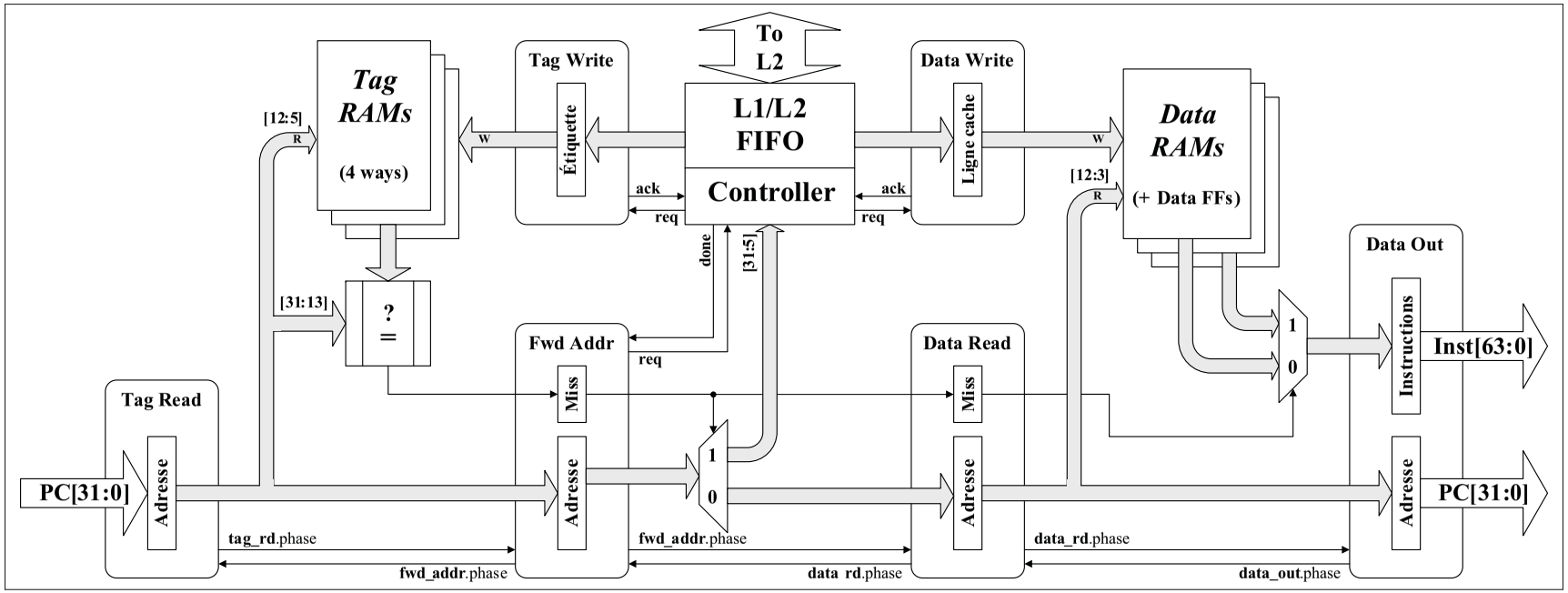


Figure 4.8 Pipeline asynchrone développé pour la cache d'instructions

La quatrième tâche est l'écriture de la nouvelle ligne de cache provenant de la cache L2 et qui contient l'information demandée. Cette tâche est associée au jeton « Data Write ». Une cinquième tâche est assignée au contrôleur de l'interface entre les caches L1 et L2. Cette tâche permet de prendre l'adresse demandée à la cache L1 et la transférer sous forme de requête au niveau de cache L2, suite à un « cache miss ». Le jeton « Forward Address » est assigné à cette tâche.

Deux tâches secondaires sont ajoutées au pipeline asynchrone afin de parachever le fonctionnement de la cache L1. La sixième tâche consiste à tenir en mémoire l'information demandée et l'adresse correspondante à la sortie de la mémoire cache, et ce, jusqu'à ce que le module subséquent ait terminé de les lire. Le jeton « Data Output » est assigné à cette tâche. La septième et dernière tâche permet de purger l'entièreté du pipeline de la cache L1. Le jeton « Flush » est associé à cette tâche. Il est à noter que le module « Flush » effectuant la purge du pipeline n'est pas représenté à la figure 4.8 mais est décrit à la section 4.3.3.

4.3.2 Fonctionnement de la cache d'instruction asynchrone

Le fonctionnement du pipeline asynchrone de la mémoire cache L1 d'instruction est décrit sous forme d'ordinogramme à la figure 4.9. Le premier étage du pipeline implémente le jeton « Tag Read ». L'adresse 32 bits du PC est d'abord mémorisée dans les DFFs de l'étage. Puis l'adresse physique des instructions demandées, soit les bits 12 à 5 du PC, est envoyée à la mémoire contenant les étiquettes des quatre voies de la cache. Les quatre étiquettes de 19 bits sont alors comparées avec l'étiquette de l'adresse du PC, soit les bits 31 à 13. Le résultat de cette comparaison sur 4 bits, qui détermine si une des voies de la cache L1 contient l'information demandée, est alors envoyé au prochain étage du pipeline.

Le deuxième étage du pipeline implémente le jeton « Forward Address ». L'adresse du PC ainsi que le résultat de la comparaison sont ainsi mémorisés. Cet étage intègre une fourche de pipeline telle que représentée à la figure 4.6, où la décision provenant d'un étage antérieur définit l'opération.

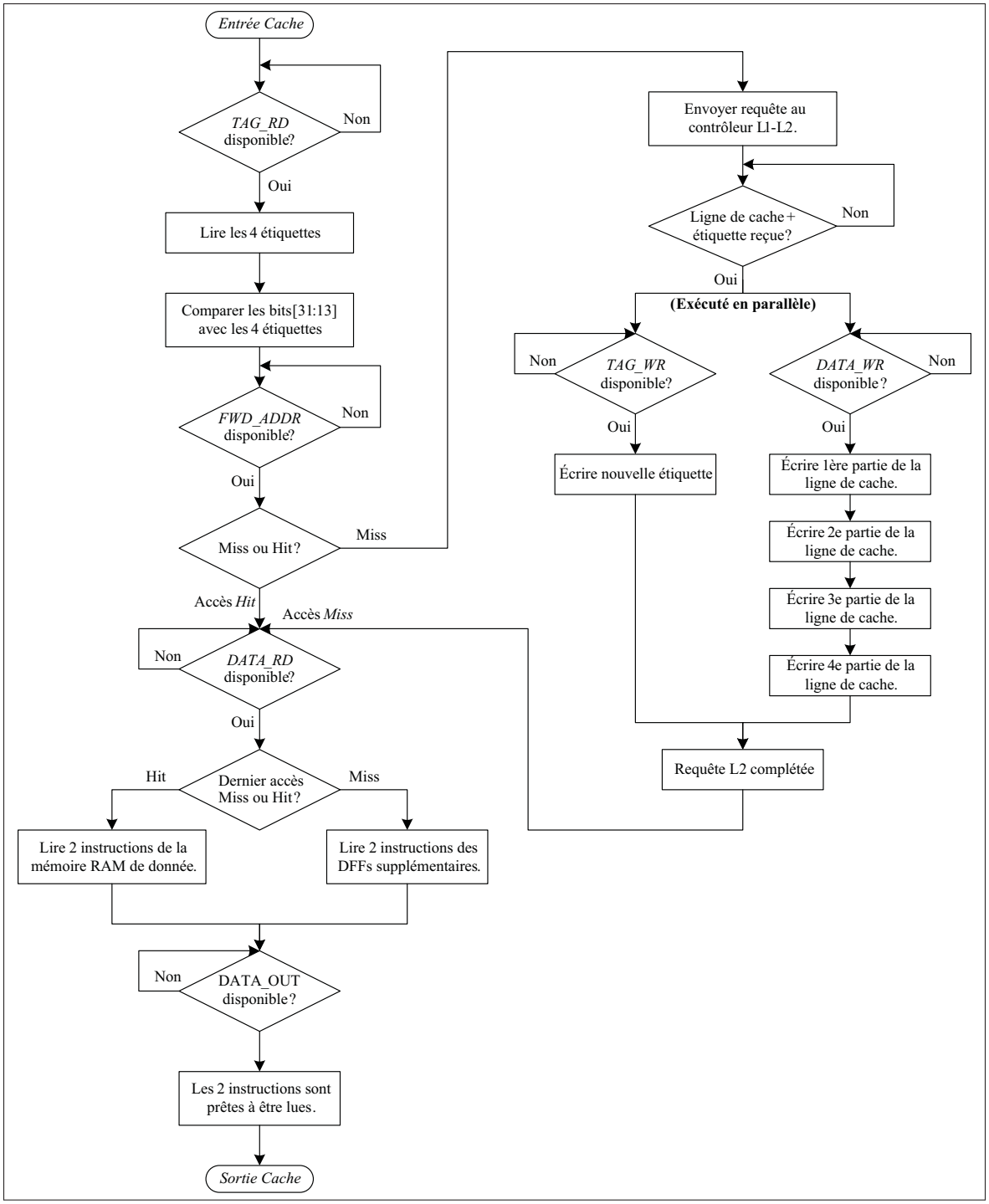


Figure 4.9 Fonctionnement de la cache d'instruction asynchrone

Par conséquent, le résultat de la comparaison détermine si l'adresse du PC est transférée au prochain étage du pipeline ou au contrôleur L1-L2. Dans le cas où il y a un « cache miss », une requête contenant l'adresse du PC demandée et la voie de remplacement est envoyée au contrôleur L1-L2. La voie de remplacement de la nouvelle ligne de cache est définie par un algorithme pseudo-aléatoire, peu complexe et efficace énergétiquement. Cet algorithme est basé sur un circuit de registre à décalage à rétroaction linéaire (ou LFSR) de 32 bits. Dans le cas où il y a un « cache hit », l'adresse du PC et le numéro de la voie contenant l'information demandée sont envoyés au prochain étage du pipeline.

Les troisième et quatrième étages du pipeline implémentent respectivement les jetons « Tag Write » et « Data Write ». Ces deux étages sont intrinsèquement reliés au contrôleur L1-L2 et peuvent opérer de manière séquentielle par rapport aux autres étages de pipeline. Lorsque le contrôleur L1-L2 reçoit une requête, la mémoire cache L2 doit retourner la ligne de cache manquante, la voie de remplacement ainsi que l'adresse correspondante. Ces données sont placées dans un FIFO du contrôleur L1-L2 avant d'être transférées à la cache L1 d'instruction.

La réception des données par le FIFO provoque une requête aux modules des jetons « Tag Write » et « Data Write ». Dès que les ressources sont disponibles, soit les mémoires internes de la cache L1, le transfert des données peut commencer. D'une part, la nouvelle étiquette est écrite à la voie de remplacement dans la mémoire des étiquettes. D'une autre part, la nouvelle ligne de cache est écrite dans la mémoire de données à la voie et à l'adresse physique déterminées. L'opération s'effectue en quatre cycles d'horloge de la L2, en transférant 64 bits à la fois. De plus, l'information de 64 bits initialement demandée par la cache L1 est stockée dans des DFFs supplémentaires afin de pouvoir rapidement être lue par la suite.

Le cinquième étage du pipeline implémente le jeton « Data Read ». L'adresse du PC et la voie où il y a correspondance sont mémorisées. Cet étage intègre un joint de pipeline tel que représenté à la figure 4.7, où le résultat précédent de la comparaison des étiquettes définit l'opération. Le résultat de la comparaison est encodé sous forme de requêtes, d'une valeur '0' pour un « cache hit » et '1' pour un « cache miss ». Lors d'un « cache miss », l'information

stockée dans les DFFs supplémentaires est envoyée au prochain étage du pipeline. Lors d'un « cache hit », une lecture est faite à la mémoire de donnée de la cache à la voie et l'adresse physique déterminée.

Le sixième étage du pipeline implémente le jeton « Data Output ». L'adresse du PC ainsi que les deux instructions sont mémorisées dans des DFFs. Cet étage redevient disponible dès que le module subséquent a correctement lu ces données.

4.3.3 Séquence de purge et de réinitialisation

Il existe deux événements pouvant réinitialiser les bascules DFF du pipeline de la mémoire cache. Tout d'abord, le signal de réinitialisation $\overline{Reset}$ est globalement envoyé à travers le processeur, suite à une réinitialisation matérielle ou logicielle. Lors de ce type d'événement, tous les éléments mémoires de type DFF ainsi que les mémoires RAM sont réinitialisés à '0'.

Toutefois, certaines DFFs nécessitent aussi une réinitialisation lors d'une purge du pipeline par le PCBP. En effet, lorsqu'une mauvaise adresse du PC est demandée à la cache d'instruction, son pipeline doit être purgé afin de repartir à la bonne adresse du PC. La fréquence à laquelle une purge du pipeline se produit est fortement dépendante du type de programme exécuté. Cette procédure impose non seulement une latence avant le retour à un état stable, mais aussi une dépense énergétique supplémentaire.

Le pipeline asynchrone proposé possède un avantage comparativement au pipeline synchrone précédent. En effet, seulement les « toggle flops » des éléments Click et les bascules S-R des jetons doivent être réinitialisées pour effectuer correctement la purge. Ainsi, un total de 6 DFFs et 6 bascules S-R sont réinitialisés lors de la purge.

La principale difficulté d'implémentation de la purge du pipeline consiste à permettre à l'interface des caches L1 et L2 de terminer un transfert avant d'appliquer le signal de réinitialisation. En effet, la version actuelle de la mémoire cache L2 ne permet pas d'interrompre une requête non complétée. Pour pallier cette problématique, un septième module de jeton « Flush » a été

implémenté au sein de la cache asynchrone pour gérer la purge du pipeline. Ce module permet donc d'attendre qu'une requête incomplète à la L2 soit terminée avant d'appliquer le signal de réinitialisation.

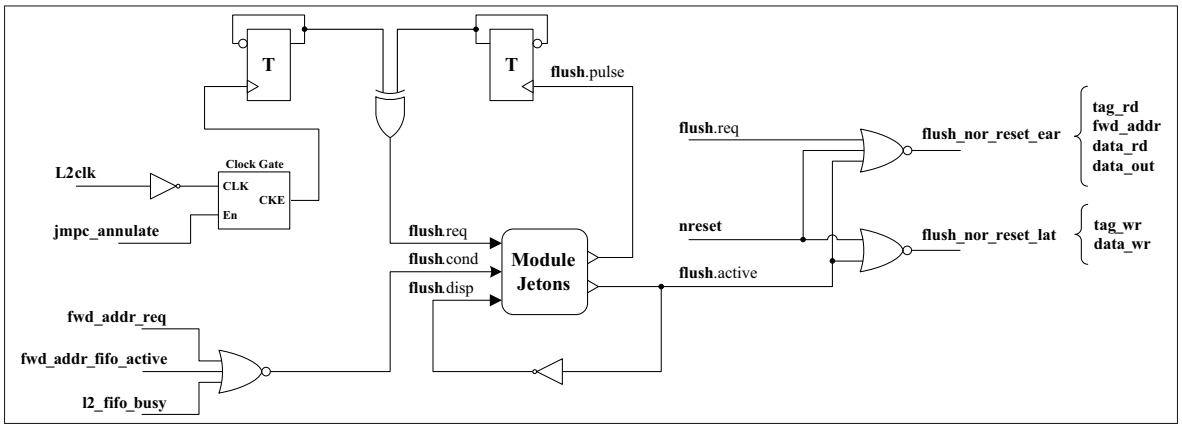


Figure 4.10 Circuit de purge du pipeline de la cache d'instruction asynchrone

Le fonctionnement du circuit effectuant la purge, tel que décrit à la figure 4.10, est le suivant. Lors d'une procédure de purge, le PCBP envoie le signal *jmpc_annulate* à tous les modules du processeur. Ce signal indique que le dernier saut de l'adresse du PC est annulé et que la purge des jetons et des ressources peut commencer. Le signal *jmpc_annulate* est donc maintenu à '1' pendant une période de l'horloge.

Pour gérer la purge du pipeline de la cache, des éléments Click ont été implémentés. La première « toggle-flop » met en mémoire le signal de purge du PCBP et la seconde mémorise le signal d'activation du jeton « Flush ». L'impulsion utilisée pour mettre en mémoire le signal de purge est générée à partir de l'horloge synchrone et du signal *jmpc_annulate*. Lorsque les phases des deux « toggle-flops » sont différentes, la requête *flush_req* est envoyée au module du jeton « Flush ».

Afin d'activer le module du jeton, les conditions d'utilisation suivantes doivent être satisfaites. Tout d'abord, il ne doit pas y avoir une requête active en « cache miss » au module du jeton « Forward Address ». Ensuite, le même module « Forward Address » ne doit pas être activé.

Finalement, il ne doit pas y avoir de transfert actif entre la cache L2 et la cache L1. Lorsque toutes ces conditions sont satisfaites, le module du jeton « Flush » génère un signal d'activation ainsi qu'une impulsion.

Le signal d'activation *flush_active* est utilisé différemment dans deux parties du pipeline. La première partie du pipeline pouvant procéder immédiatement à la purge contient les jetons « Tag Read », « Forward Address », « Data Read » et « Data Output ». Ainsi, les DFFs et les bascules S-R sont réinitialisés lorsque $\overline{Reset}$, *flush_req* ou *flush_active* sont à '1'. Le signal résultant *flush_nor_reset_ear* définit donc le signal de purge prioritaire, sans contraintes.

La seconde partie du pipeline pouvant seulement procéder à la purge lorsque aucun transfert entre la L1 et L2 n'est prévu ou en cours contient les jetons « Tag Write » et « Data Write ». Pour cette partie, les DFFs et les bascules S-R sont réinitialisés seulement lorsque $\overline{Reset}$ ou *flush_active* sont à '1'. Le signal résultant *flush_nor_reset_lat* définit donc le signal de purge tardif, avec contraintes. Par ailleurs, le module du jeton « Flush » n'est évidemment pas réinitialisé lors d'une purge de pipeline. Finalement, l'impulsion *flush_pulse* met en mémoire le signal d'activation du jeton, ce qui indique la complétion de la purge du pipeline.

4.4 Interfaces extérieures

L'intégration de la cache d'instruction L1 dans le processeur ARM asynchrone nécessite la création d'interfaces pour ses entrées et sorties. En effet, ces modules sont nécessaires pour gérer le passage entre les domaines d'horloge et les échanges de données entre les modules. Ainsi, trois interfaces ont été développées, telles qu'illustrées à la figure 4.11.

La première interface est le FIFO d'entrée, qui est définie entre le PCBP et l'entrée du pipeline de la cache L1 asynchrone. Cette interface permet au « branch-predictor » de stocker jusqu'à huit adresses du PC de manière synchrone dans un FIFO. La cache d'instruction peut ainsi accéder aux adresses PC demandées de manière asynchrone, à sa fréquence d'opération native.

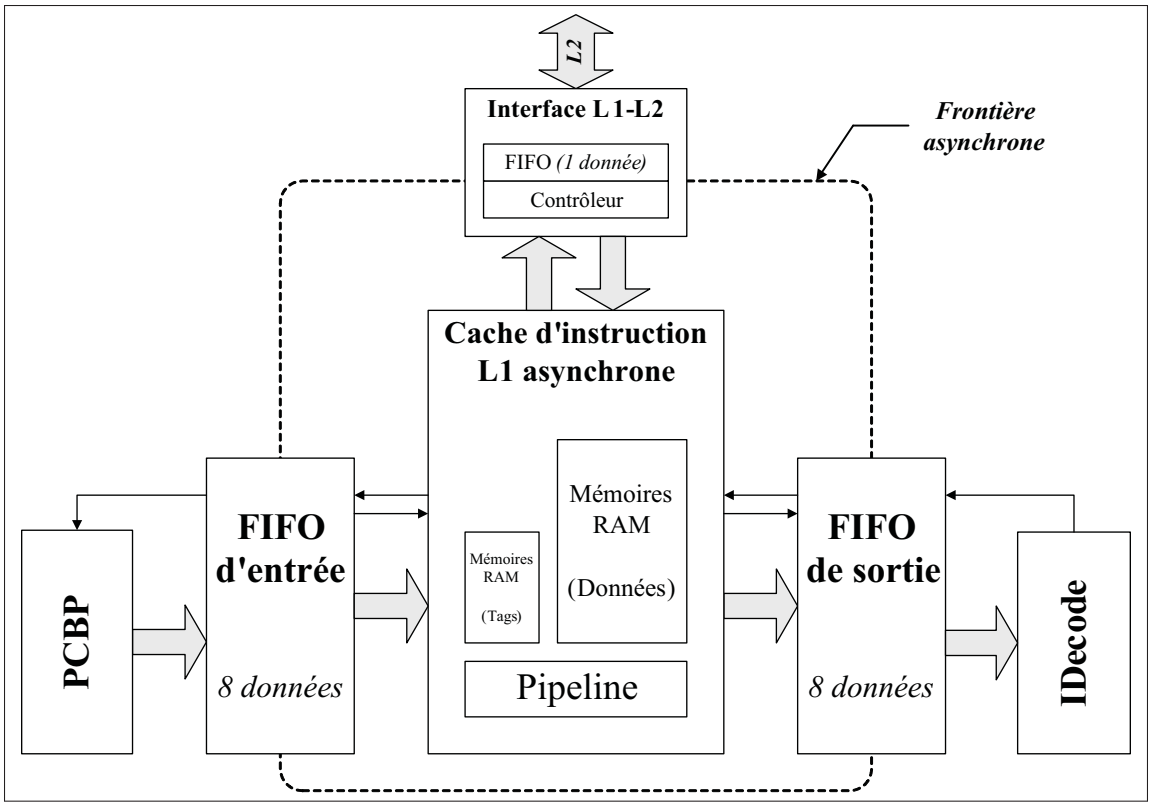


Figure 4.11 Interfaces de la cache d'instruction L1 asynchrone avec le processeur

La seconde interface est définie entre la mémoire cache L1 et la mémoire cache L2. L'interface L1-L2 comprend un contrôleur et une section FIFO, où les données transférées sont stockées momentanément. Cette interface permet à la cache d'instruction d'envoyer une requête à la L2 pour obtenir une ligne de cache manquante, suite à « cache miss ». Le contrôleur de l'interface gère ainsi les transferts de données de la L1 vers la L2, et vice-versa.

La troisième interface est le FIFO de sortie, qui est définie entre la cache d'instruction L1 et IDecode. Cette interface permet à la cache L1 de stocker les instructions demandées avec leur adresse dans un FIFO de manière asynchrone. Dès que IDecode est prêt, il peut lire l'information stockée dans le FIFO et décoder les instructions.

4.4.1 Interface avec le PCBP en entrée

Cette section définit l'interface entre le PCBP et le l'entrée de la cache d'instruction L1, soit le FIFO d'entrée. Ce module permet d'écrire de manière synchrone jusqu'à huit adresses de PC provenant du « branch predictor » dans un FIFO. Chaque adresse peut alors être lue de façon asynchrone par la cache d'instruction L1 lorsque celle-ci est prête.

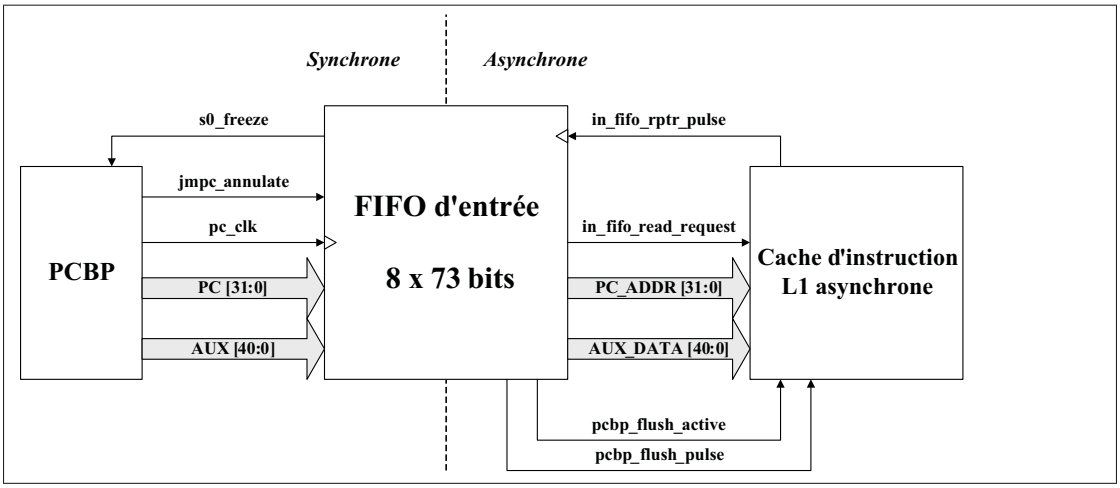


Figure 4.12 Interface entre le PCBP et la cache d'instruction L1 asynchrone

La figure 4.12 illustre l'information qui est transférée de part et d'autre du FIFO d'entrée. De son côté, le PCBP transmet des signaux synchrones à l'horloge globale *L2clk*. À chaque front montant de l'horloge, une nouvelle adresse $PC[31:0]$ et l'information auxiliaire $AUX[40:0]$ sont envoyées au FIFO d'entrée. Le signal d'horloge *pc_clk*, provenant directement du pipeline de sortie du PCBP, fait office de signal d'horloge et met en mémoire les données envoyées. Lorsque le FIFO d'entrée est plein, le signal *s0_freeze* permet de suspendre l'envoi de nouvelles adresses par le PCBP. Finalement, lorsque le PCBP effectue une mauvaise prédiction, le signal *jmpc_annulate* est envoyé pendant une période d'horloge et effectue la purge du FIFO d'entrée.

Du côté de la cache d'instructions L1, l'information est reçue et envoyée de manière asynchrone. Lorsque le FIFO contient au moins une entrée valide, celle-ci est multiplexée à sa

sortie et le signal de requête *in_fifo_read_request* est envoyé à la cache d'instruction. Lorsque le premier étage de la cache est prêt à lire l'information, l'impulsion *in_fifo_rptr_pulse* est envoyée au FIFO d'entrée. Cette impulsion indique que la donnée a bien été lue. Le sélecteur du multiplexeur de sortie pointe alors sur la prochaine entrée du FIFO. De plus, lorsque le PCBP effectue une purge, le signal *jmpc_annulate* est d'abord modifié au sein du FIFO d'entrée. Puis, l'impulsion *pcbp_flush_pulse* et le signal d'activation *pcbp_flush_active* à la cache d'instruction.

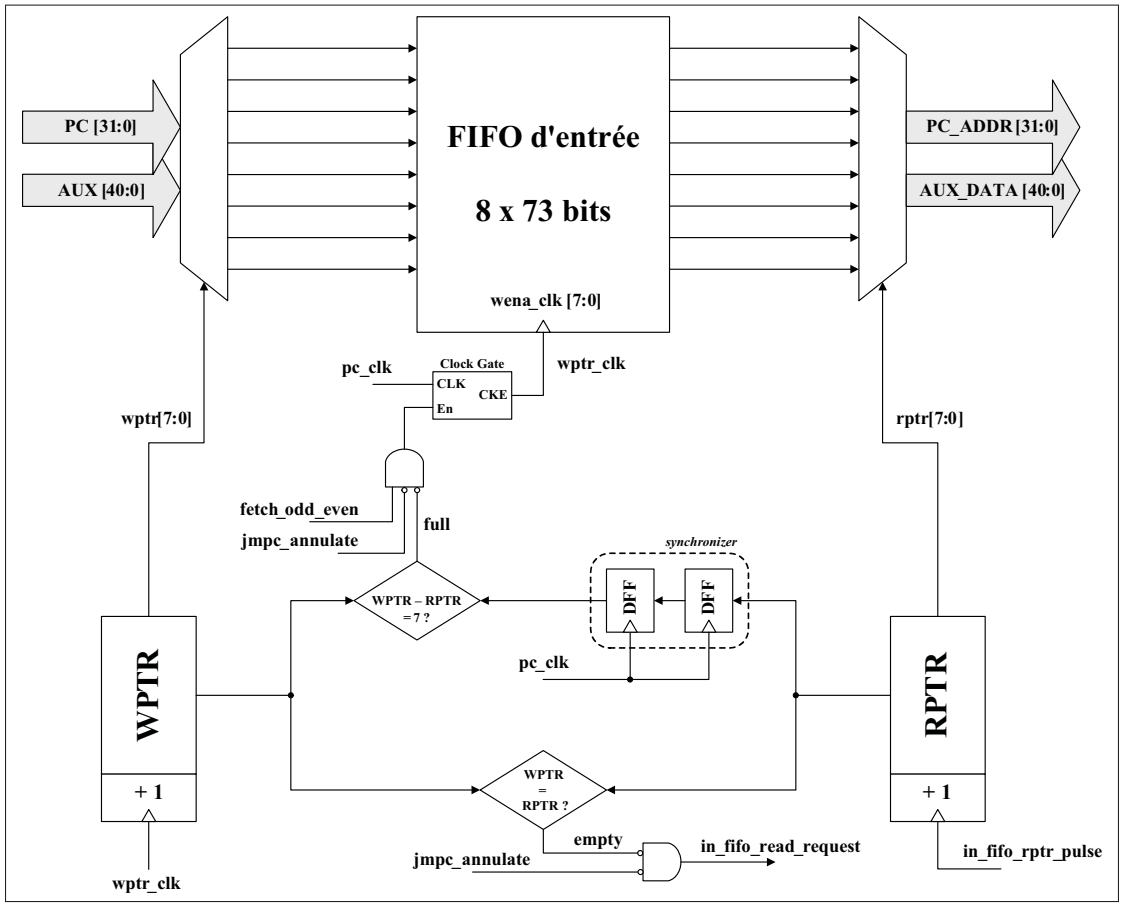


Figure 4.13 Fonctionnalité du FIFO d'entrée de 8x73 bits

La structure interne du FIFO d'entrée 8x73bits est présentée à la figure 4.13. Le fonctionnement du FIFO est basé sur deux pointeurs de 8 bits implémentés en logique « one-hot » [1] grâce à un registre à décalage circulaire. Le pointeur en écriture $WPTR[7:0]$ contrôle le démultiplexeur d'entrée du FIFO et détermine l'espace mémoire où s'effectue l'écriture synchrone. Le pointeur en lecture $RPTR[7:0]$ contrôle le multiplexeur de sortie du FIFO et détermine l'espace mémoire où s'effectue la lecture asynchrone.

La comparaison des deux pointeurs permet d'indiquer lorsque le FIFO est vide ou plein. Le FIFO est vide lorsque la valeur des pointeurs d'écriture et de lecture est égale. Ainsi, lorsque le FIFO n'est pas vide et que le PCBP n'est pas en procédure de purge, le signal *in_fifo_read_request* indique à la cache d'instruction qu'il peut lire le FIFO d'entrée. Le FIFO est plein lorsque la différence entre la valeur des pointeurs d'écriture et de lecture équivaut à 7. Pour effectuer cette comparaison, le pointeur de lecture $RPTR[7:0]$ doit d'abord être synchronisé à l'horloge globale. Par conséquent, une écriture est effectuée lorsque le FIFO n'est pas plein, que le PCBP n'est pas en purge et lorsque le signal *fetch_odd_even* est valide. Ce signal provient des bits 1 et 0 des données auxiliaires et détermine si on doit lire une instruction impaire, paire, ou les deux.

4.4.2 Interface avec IDecode en sortie

Cette section définit l'interface entre la cache d'instruction L1 et IDecode, soit le FIFO de sortie. Ce module permet d'écrire de manière asynchrone jusqu'à huit vecteurs d'information, soit deux instructions, son adresse PC correspondante et l'information auxiliaire, dans un FIFO. Cette information peut alors être lue de façon synchrone ou asynchrone par le décodeur d'instruction lorsque celui-ci est prêt.

La figure 4.14 illustre l'information qui est transférée de part et d'autre du FIFO de sortie. De son côté, la cache d'instruction transmet des signaux de manière asynchrone au déco-

1. Une logique « one-hot » permet de représenter n états en utilisant une représentation binaire sur n bits n'ayant qu'un seul bit à 1 à la fois. Par exemple, les trois états A, B, et C peuvent être représentés avec un encodage one-hot sur 3 bits, soit respectivement 001, 010, et 100.

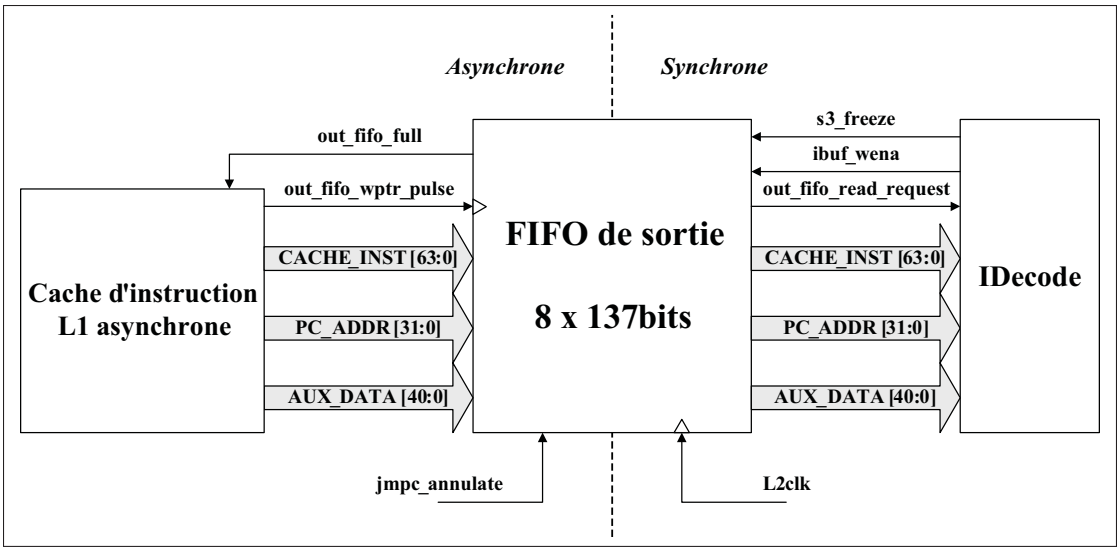


Figure 4.14 Interface entre la cache d'instruction L1 asynchrone et IDecode

deur d'instruction. Dès que le dernier étage de pipeline « Data Output » est prêt, deux instructions $CACHE\_INST[63:0]$, leur adresse $PC\_ADDR[31:0]$ et l'information auxiliaire $AUX\_DATA[40:0]$ sont envoyées au FIFO de sortie. L'impulsion *out_fifo_wptr_pulse* met en mémoire les données envoyées. Lorsque le FIFO de sortie est plein, le signal *out_fifo_full* permet de suspendre l'envoi de nouvelles instructions par la cache L1. Finalement, lorsque le PCBP effectue une mauvaise prédiction, le signal *jmpc_annulate* est aussi envoyé pendant une période d'horloge et effectue la purge du FIFO de sortie.

Du côté du décodeur d'instruction, l'information est reçue et envoyée de manière synchrone. En effet, dans cette version du processeur ARM asynchrone d'Octasic, le décodage d'instruction a été simplifié et implémenté de manière synchrone. Ce n'est plus le cas pour la version plus récente du processeur, où IDecode est complètement asynchrone. Cela étant dit, lorsque le FIFO contient au moins une entrée valide, celle-ci est multiplexée à sa sortie et le signal de requête *out_fifo_read_request* est envoyé à IDecode. Lorsque IDecode est prêt à lire de nouvelles instructions, le signal *s3_freeze* est mis à '0' et *ibuf_wena* est valide. Le signal *ibuf_wena* indique que le registre interne de IDecode a assez d'espace pour décoder de nouvelles instructions. Le signal d'horloge est alors utilisé pour mettre à jour le sélecteur du multiplexeur

de sortie. Celui-ci pointe alors sur le prochain espace mémoire du FIFO contenant les deux instructions.

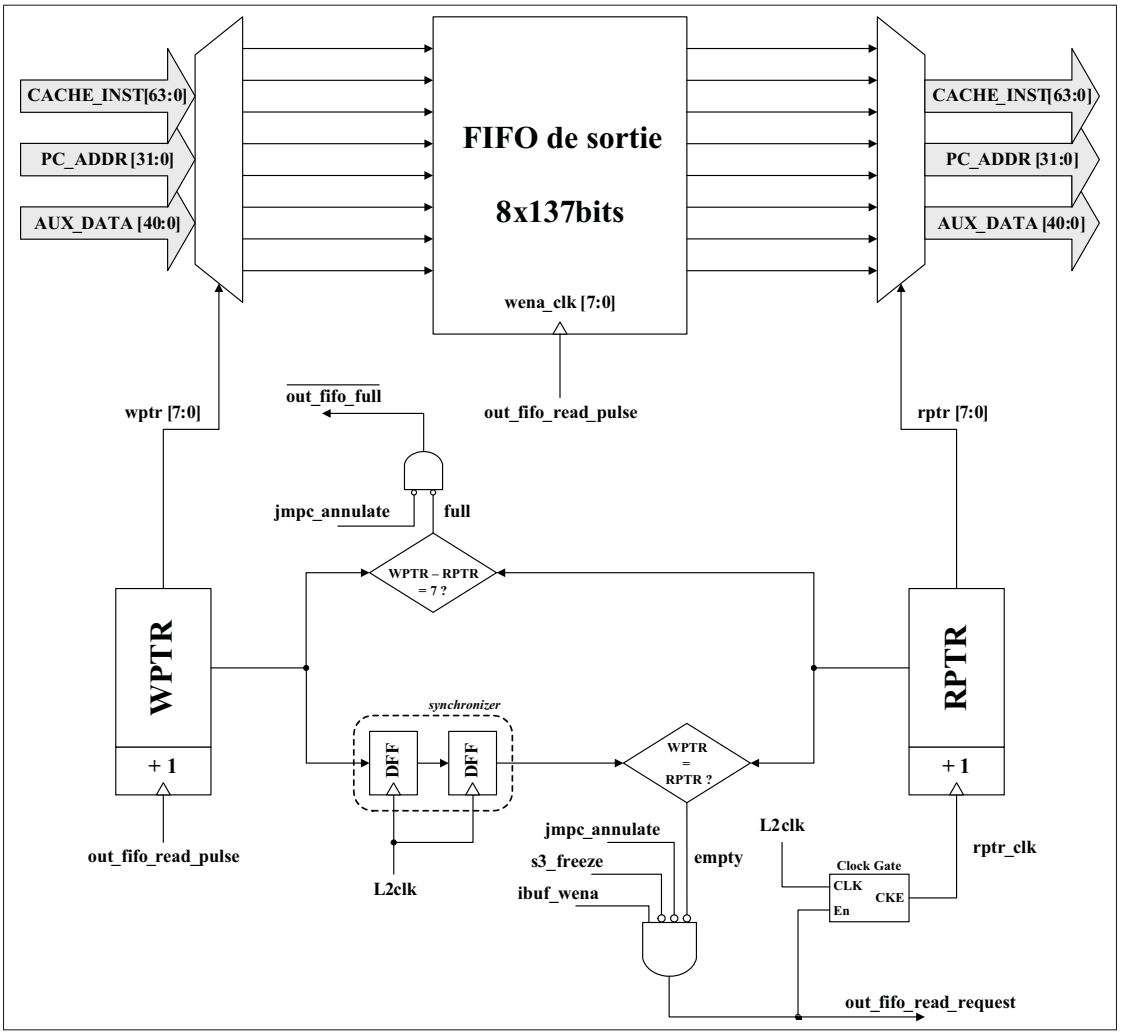


Figure 4.15 Fonctionnalité du FIFO de sortie 8x137bits

La structure interne du FIFO de sortie 8x137bits est présentée à la figure 4.15. Le fonctionnement du FIFO est basé sur le même système de pointeurs d'écriture et de lecture que le FIFO d'entrée. Cela étant dit, la logique de comparaison est légèrement différente étant donné que l'on synchronise plutôt le pointeur d'écriture à l'horloge globale. Le pointeur en écriture $WPTR[7:0]$ contrôle le démultiplexeur d'entrée du FIFO et détermine l'espace mémoire où

s'effectue l'écriture asynchrone. Le pointeur en lecture $RPTR[7:0]$ contrôle le multiplexeur de sortie du FIFO et détermine l'espace mémoire où s'effectue la lecture synchrone.

Le FIFO est vide lorsque les valeurs des pointeurs d'écriture et de lecture sont égales. Pour effectuer cette comparaison, le pointeur d'écriture $WPTR[7:0]$ doit d'abord être synchronisé à l'horloge globale. Ainsi, lorsque le FIFO n'est pas plein, que la procédure de purge n'est pas active et que IDecode est prêt à recevoir de nouvelles instructions, le signal *out_fifo_read_request* indique au décodeur d'instruction qu'il peut lire le FIFO de sortie. Le FIFO est plein lorsque la différence entre les valeurs des pointeurs d'écriture et de lecture est égale à 7. Par conséquent, une écriture peut être effectuée lorsque le FIFO n'est pas plein et qu'il n'y a pas de purge active. Le signal $\overline{out\_fifo\_full}$ est alors mis à '1' et l'étage « Data Output » du pipeline peut envoyer l'information au FIFO de sortie avec l'impulsion *out_fifo_wptr_pulse*.

4.4.3 Interface avec le niveau de mémoire L2

Cette section définit l'interface entre la cache d'instruction L1 et la mémoire cache L2, soit l'interface L1-L2. Ce module permet à la cache d'instruction d'envoyer une requête à la L2 pour obtenir une ligne de cache manquante, suite à « cache miss ». Le contrôleur de l'interface gère aussi les transferts de données de la L1 vers la L2 et l'inverse en incorporant des éléments Click. Cela permet à la cache d'instruction L1 d'écrire et lire des données de manière asynchrone et à la cache L2 de lire et écrire des données de façon synchrone. De plus, l'interface L1-L2 continue à opérer même lors d'une procédure de purge par le PCBP. La cache d'instruction L1 doit donc conclure la transaction avec la cache L2 avant de pouvoir complètement purger son pipeline.

La figure 4.16 illustre l'information qui est transférée de part et d'autre de l'interface L1-L2. De son côté, la cache d'instruction transmet des signaux de manière asynchrone à la cache L2. Lorsque l'étage « Forward Address » détecte un « cache miss », l'adresse de la ligne de cache manquante, soit les bits 31 à 15 de l'adresse du PC, sont envoyé avec l'impulsion *l2_fifo_req_pulse*. De plus, la voie de remplacement pour la nouvelle ligne de cache, générée de manière

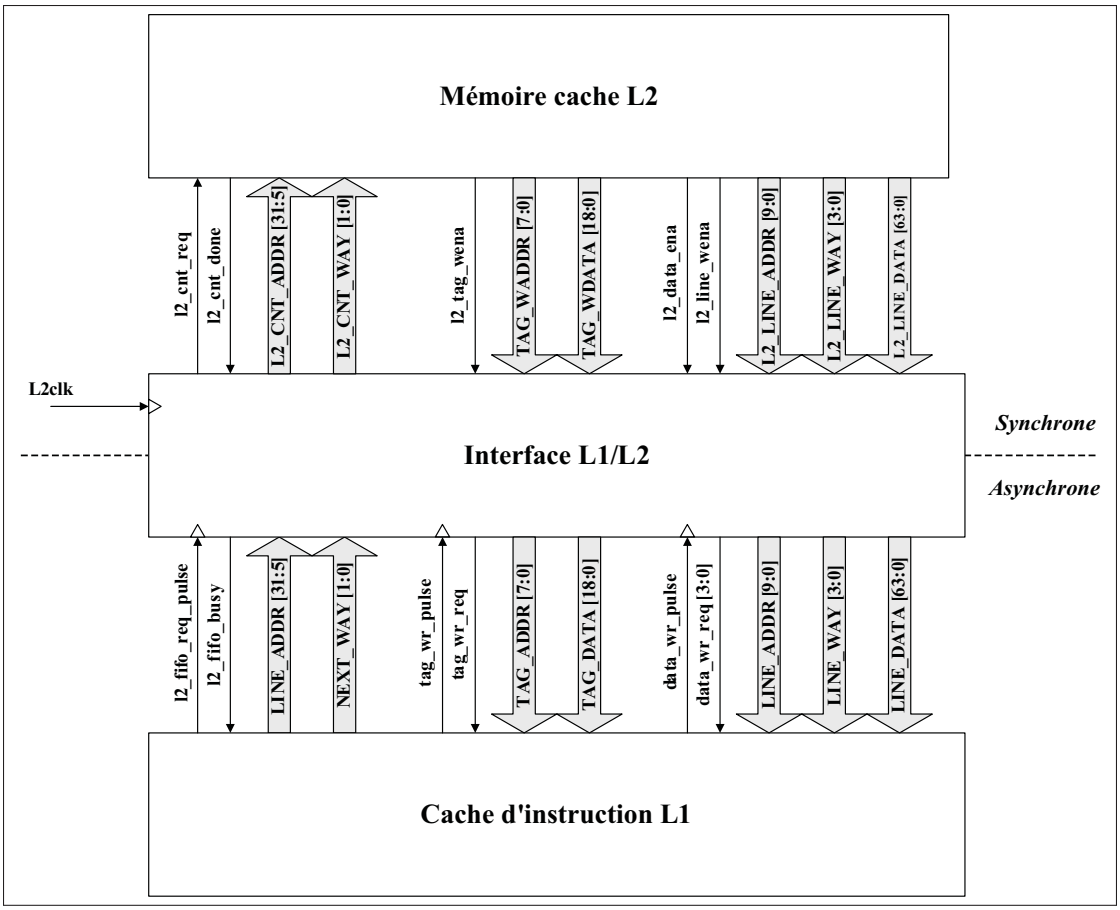


Figure 4.16 Interface entre la cache d'instruction L1 asynchrone et la mémoire cache L2 synchrone

pseudo-aléatoire, est aussi envoyée à l'interface L1-L2. Dès que la ligne de cache provenant de la cache L2 est prête à être lue, les requêtes synchrones *tag_wr_req* et $data\_wr\_req[3:0]$ sont envoyés à la cache d'instruction L1. Les modules « Tag Write » et « Data Write » peuvent alors respectivement écrire la nouvelle étiquette et la ligne de cache à la voie préalablement définie dans la cache L1. Lors de ces opérations, les impulsions *tag_wr_pulse* et *data_wr_pulse* sont envoyés à l'interface L1-L2 pour compléter la transaction. Finalement, lorsqu'une transaction est conclue, le signal *l2_cnt_done* est mis à '1' pour une période d'horloge.

Du côté de la cache L2, l'information est reçue et envoyée de manière synchrone. Dans cette version du processeur ARM asynchrone d'Octasic, la gestion de la cache L2 est faite de façon

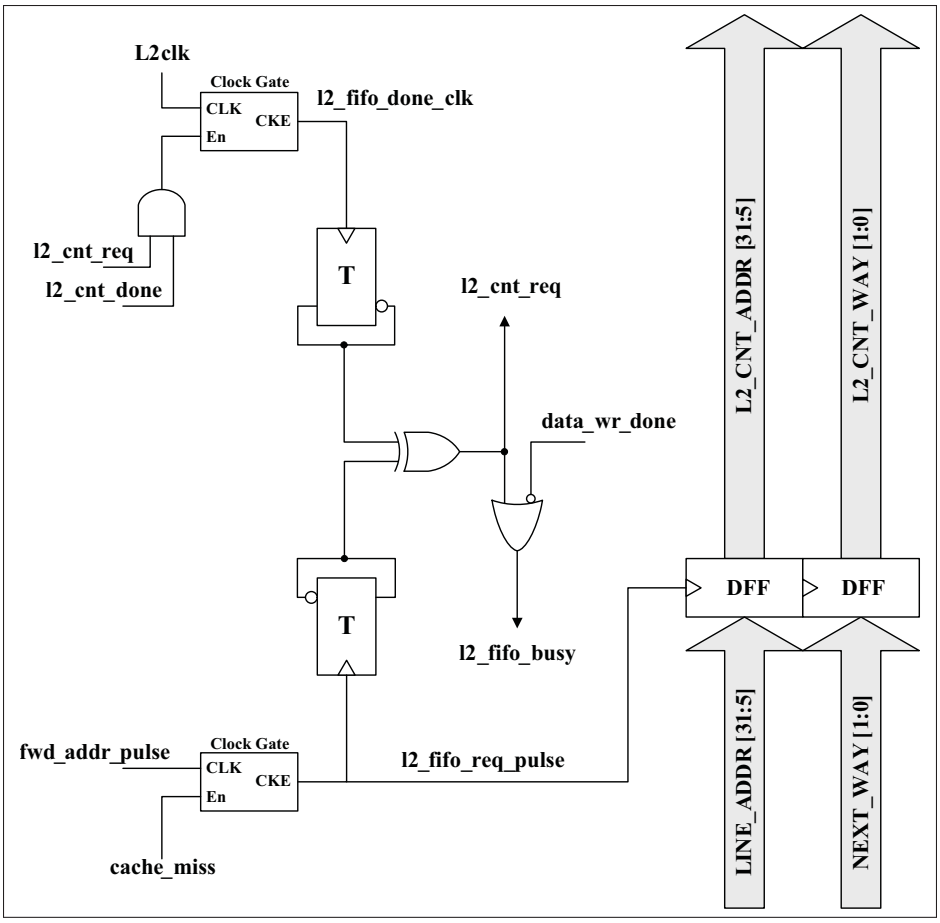


Figure 4.17 Fonctionnalité du contrôleur L1 vers L2

logicielle. Une implémentation physique est maintenant implémentée dans la version plus récente du processeur. Cela étant dit, la cache L2 reçoit l'adresse de la ligne de cache et la voie de remplacement de l'interface L1-L2 lorsque le signal de requête *l2_cnt_req* est mis à '1'. Lorsque la cache L2 est prête, elle écrit la nouvelle étiquette, l'adresse de l'étiquette, la ligne de cache et son adresse à la voie de remplacement spécifiée. Lors d'un transfert de données de la cache L2 vers l'interface, le signal d'activation global *l2_data_ena* est mis à '1'. De plus, les signaux *l2_tag_wena* et *l2_data_wena* permettent respectivement d'activer l'écriture de la nouvelle étiquette et la ligne de cache.

La structure du contrôleur est partitionnée en deux sous-parties. La première partie de l'interface qui gère les requêtes de la L1 vers la L2 est représentée à la figure 4.17. Le protocole

de communication est basé sur les éléments Click, l'état de l'interface est donc géré par deux « toggle-flops ». Lorsque l'étage « Forward Address » détecte un « cache miss », la phase de la partie L1 de l'interface est mise à jour par l'impulsion *l2_fifo_req_pulse*. Ce signal met alors en mémoire l'adresse de la ligne de cache manquante $line\_addr[31:5]$ et la voie de remplacement $line\_way[1:0]$. Lorsque les deux phases des « toggle-flops » sont différentes, le signal *l2_cnt_req* indique qu'il y a une requête active entre le L1 et la L2. Lorsque la cache L2 termine l'envoi de la nouvelle étiquette et de la ligne de cache à l'interface, le signal *l2_cnt_req* est mis à '1' pendant une période d'horloge. Cela termine donc la requête la cache L2.

Cependant, tant que le transfert entre l'interface L1-L2 et la cache d'instruction n'est pas complété, le signal *l2_fifo_busy* indique que l'interface L1-L2 est occupée. Ce signal bloque le premier étage de pipeline, ce qui prévient un « cache miss » erroné et préserve la séquentialité des transactions.

La deuxième partie de l'interface qui gère les transferts de données de la L2 vers la L1 est représentée à la figure 4.18 Le protocole de communication, basé partiellement sur les éléments Click, gère le transfert de la nouvelle étiquette. Le transfert de la ligne de cache est assuré par un système de comparaison de pointeurs en écriture et en lecture, similaire à celui implémenté dans les FIFO d'entrée et de sortie.

Lorsqu'une nouvelle ligne de cache est prête à être écrite dans l'interface L1-L2, le signal d'activation *l2_tag_wena* est mis à '1'. De plus, les signaux *l2_cnt_req* et *l2_data_ena* sont valides, indiquant qu'une requête et un transfert de données sont actifs. Ainsi, la nouvelle étiquette $l2\_tag\_wdata[18:0]$ et son adresse physique $l2\_tag\_waddr[7:0]$ sont mises en mémoire de façon synchrone à l'horloge globale *L2clk*. Cette opération met aussi à jour la phase du contrôleur L1-L2, ce qui active le signal de requête *tag_wr_req*. La cache d'instruction peut alors écrire la nouvelle étiquette $tag\_data[18:0]$ à l'adresse physique $tag\_addr[7:0]$ et à la bonne voie dans la mémoire RAM contenant les étiquettes. Lors de ce transfert, l'impulsion *tag_wr_pulse* met à jour la phase du contrôleur, ce qui désactive la requête en écriture de l'étiquette.

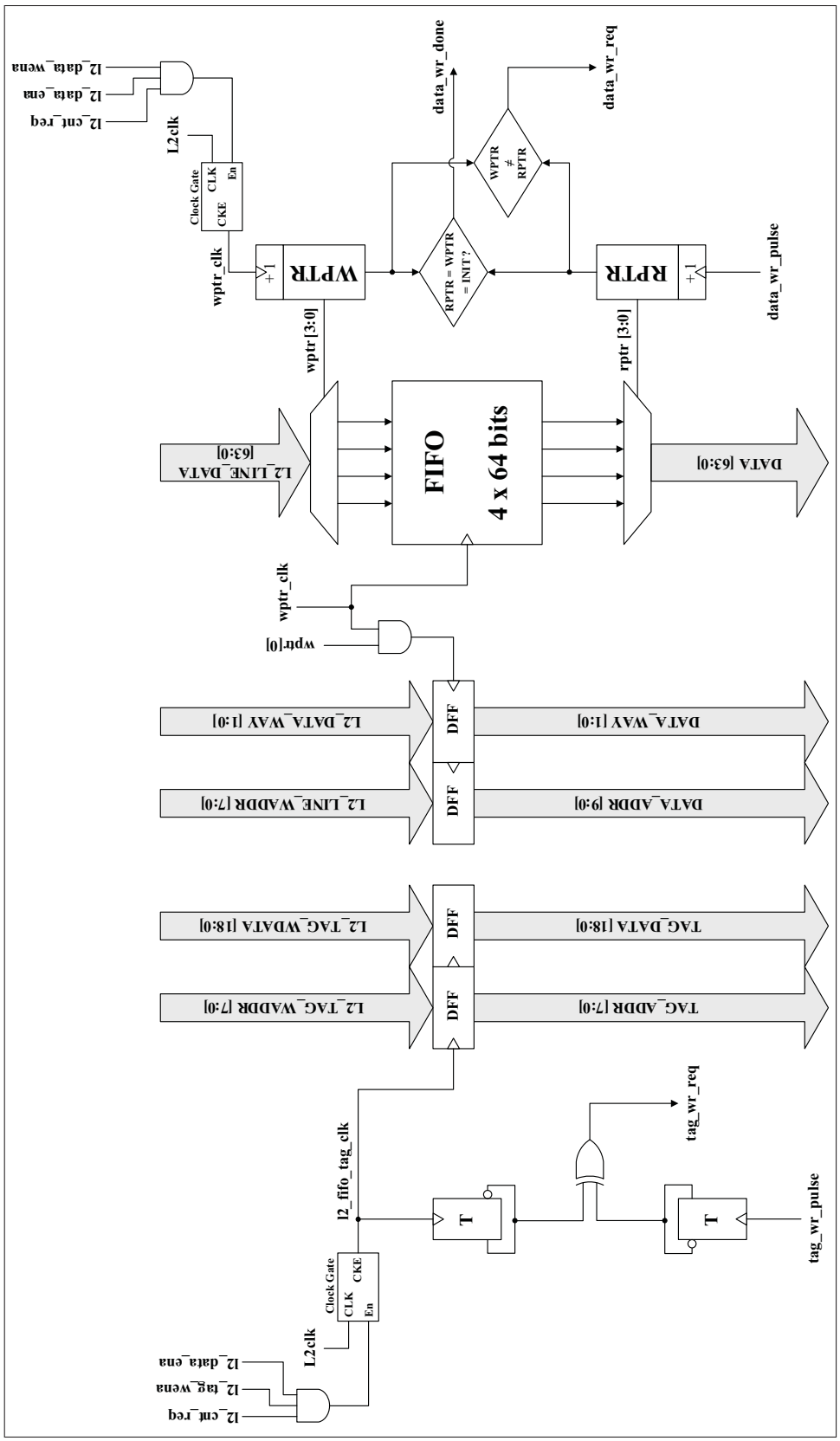


Figure 4.18 Fonctionnalité du du contrôleur L2 vers L1

Lorsque la nouvelle ligne de cache est prête à être écrite dans l'interface L1-L2, le signal d'activation *l2_data_wena* est mis à '1'. Les mêmes signaux *l2_cnt_req* et *l2_data_ena* sont valides, indiquant qu'une requête et un transfert de données sont actifs. Contrairement aux autres données placées en mémoire à l'interface L1-L2, le transfert de la ligne de cache est effectué en quatre cycles d'horloge consécutifs. Ceci permet d'avoir la même largeur que le bus d'accès aux mémoires RAM de données du pipeline de la cache d'instructions. Chaque partie de la ligne de cache $L2\_LINE\_DATA[63:0]$ est donc écrite dans le FIFO de l'interface L1-L2 à l'espace mémoire indiqué par le pointeur en écriture $WPTR[3:0]$. Lorsque le pointeur en écriture est équivalent à 0001, soit lors de la première écriture, l'adresse physique $L2\_LINE\_WADDR[9:0]$ et la voie correspondante $L2\_LINE\_WAY[1:0]$ sont mis en mémoire. Lorsque les pointeurs $WPTR[3:0]$ et $RPTR[3:0]$ diffèrent, le signal de requête *data_wr_req* est mis à '1'. L'étage « Data Write » de la cache d'instruction peut alors écrire chaque partie de la ligne de cache, soit les données $DATA[63:0]$ qui correspondent à l'espace mémoire du FIFO indiqué par le pointeur en lecture $RPTR[3:0]$. Suite à chaque lecture, l'impulsion *data_wr_pulse* incrémente $RPTR[3:0]$. Le signal de complétion *data_wr_done* est mis à '1' lorsque les deux pointeurs reviennent à leur valeur initiale, soit 0001. Le signal *l2_fifo_busy* est alors mis à '0', indiquant que l'interface L1-L2 a terminé le traitement de la requête.

4.5 Conclusion

Dans ce chapitre, une nouvelle cache L1 asynchrone a été proposée pour le processeur ARM asynchrone d'Octasic. Fort du travail d'exploration réalisé aux chapitres précédents, le processus de conception ayant comme objectif la performance et l'efficacité énergétique a été exposé. Ce travail a permis dans un premier temps à exposer et comprendre la problématique, pour ensuite établir l'état de l'art.

La cache d'instruction L1 asynchrone proposée est basée sur un nouveau type de pipeline asynchrone. Ce pipeline intègre des caractéristiques des éléments Click et du système de jetons développé par Octasic. Ce nouveau pipeline asynchrone répond ainsi aux exigences de conception afin de rester fidèle au flot d'Octasic. Tout d'abord, trois ressources partagées à

l'interne ont été identifiées, soit les mémoires des étiquettes et de données, ainsi que l'interface avec la mémoire cache L2. Puis, le fonctionnement de la mémoire cache a été partitionnée en six tâches qui interagissent sur les trois ressources. Les étages du pipeline sont donc basés sur ces six tâches et intègrent un protocole de communication basé sur les éléments Click. Par ailleurs, trois interfaces ont été conçues pour gérer globalement l'entrée, la sortie et l'accès à la mémoire cache L2 de la cache d'instruction L1 asynchrone.

Dans ce chapitre, il a d'abord été question de l'architecture et du fonctionnement du nouveau pipeline asynchrone développé. Puis, la cache d'instruction asynchrone L1 ainsi que son fonctionnement ont été présentés. D'ailleurs, les particularités de l'intégration de la nouvelle cache au sein du processeur ARM ont été évoquées. Finalement, les interfaces à l'entrée et à la sortie de la cache d'instruction ont été présentées, ainsi que l'interface de couplage avec la mémoire cache L2. Bénéficiant d'une cache d'instruction L1 fonctionnelle, il est maintenant possible d'obtenir différents résultats de performance. Le prochain chapitre fait donc état, entre autres, des performances de la nouvelle cache proposée en termes d'énergie et de vitesse d'accès

CHAPITRE 5

ANALYSE DE PERFORMANCES DE LA CACHE ET DU PIPELINE

5.1 Introduction

Ce chapitre présente l'analyse qui a été faite pendant et après la conception de la mémoire cache d'instruction L1 asynchrone. Toutefois, l'angle de cette analyse ne permet pas de comparer directement la mémoire cache asynchrone conçue avec l'état de l'art. En effet, plusieurs facteurs extérieurs peuvent influencer les performances de la mémoire cache conçue, tels que le type de processeur réalisant les accès mémoire, le type de hiérarchie mémoire ou le type de technologie CMOS utilisée pour la conception. Il est généralement difficile de comparer les caches entre elles, car celles-ci sont intrinsèquement reliées à la fonctionnalité globale du processeur. Qui plus est, il n'est pas possible d'analyser ou de comparer pleinement les performances de la nouvelle mémoire cache au sein du processeur. Les métriques de performance établies sont fonction des modules adjacents à la cache, soit le PCBP, IDecode et la mémoire cache L2. Ultimement, la vitesse d'accès à la cache d'instructions dépend fortement de la performance du CPU qui lui est couplé.

> [T]he bulk of lost performance is not due to the number of CPU pipeline stages or functional units or choice of branch prediction algorithm or even CPU clock speed ; the bulk of lost performance is due to poor configuration of system-level parameters such as bus widths, granularity of access, scheduling policies, queue organizations, etc. Today's computer system performance is dominated by the manner in which data is moved between subsystems, i.e., the scheduling of transactions, and so it is not surprising that seemingly insignificant details can cause such a headache, as scheduling is known to be highly sensitive to such details. Consequently, one can no longer attempt system level optimization by designing/optimizing each of the parts in isolation (which, unfortunately, is often the approach taken in modern computer design). In subsystem design, nothing can be considered "outside the scope" and thus ignored. (Jacob *et al.*, 2010, p. 15)

L'étude des performances d'une mémoire cache fait habituellement partie d'une analyse plus globale du processeur en fonction de programmes de référence (Patterson et Hennessy, 2009),

tel que mentionné à la section 1.5. La métrique de performance la plus fiable demeure la différence de temps entre l'exécution d'un ensemble de programmes d'un processeur à un autre. Idéalement, l'analyse porterait sur la comparaison entre deux processeurs identiques ayant comme seule différence le type de mémoire cache d'instruction et/ou de données. Cependant, il n'est pas possible de simuler la nouvelle mémoire cache proposée en produisant deux processeurs dans le cadre de ce projet de maîtrise.

En tenant compte de ces contraintes, il faut être en mesure de comparer différentes architectures de mémoires cache dans un environnement *agnostique* au processeur. Dans le cadre de ce travail, il est donc plus avantageux d'intégrer la cache développée à un environnement virtuel non contraignant, c'est à dire en simulant les accès mémoire de manière logicielle avec un banc de test. Ensuite, en fixant les mêmes conditions d'opération entre la première cache synchrone développée et la cache asynchrone proposée, il est possible de les comparer adéquatement au sein du même processeur. En outre, la comparaison des deux architectures spécifiques de mémoires cache est plus intéressante pour le partenaire industriel.

Ce chapitre définit donc dans un premier temps comment a été validé et testé la nouvelle cache d'instruction asynchrone proposée. Cette section décrit l'environnement de simulation utilisé dans le cadre de ce projet, les différents bancs de test et leurs limitations respectives en terme d'analyse. Dans un second temps, les résultats obtenus sont présentés. Cette section expose les dimensions physiques de la cache d'instruction conçue, la vitesse d'opération de chaque étage du pipeline ainsi que le débit résultant aux interfaces. Dans un troisième temps, les résultats obtenus pour la cache d'instruction asynchrone sont comparés avec ceux de la cache synchrone. Dans un quatrième temps, une analyse globale est faite en fonction des résultats obtenus dans le cadre de ce projet.

5.2 Validation de la fonctionnalité de la cache asynchrone

Cette section introduit les étapes nécessaires à la validation de la fonctionnalité de la cache asynchrone. Avant de pouvoir évaluer les performances de la cache conçue, il est nécessaire de tester la fonctionnalité et ce, à différentes phases du flot de conception.

Il est d'abord nécessaire de décrire le flot de conception et de simulation utilisé par Octasic. Chacune des étapes du flot comprend une partie inhérente à la validation du circuit conçu, avant même d'entamer la simulation. Puis, des bancs de tests sont développés afin d'effectuer une validation fonctionnelle du circuit. L'environnement du banc de test, intégrant une section partielle ou complète du circuit, permet d'appliquer un ou plusieurs tests dirigés. Il existe plusieurs niveaux de banc de tests, allant de la simple vérification d'un étage de pipeline à l'intégration complète dans le processeur.

Globalement, l'ensemble de ces simulations possède un certain nombre de limitations ; il n'est pas possible de tester tous les cas possibles d'opération. Dans le cadre de ce projet, une expérimentation physique n'a pu être réalisée afin de tester la cache asynchrone dans un environnement réel. Même si une puce intégrant la cache asynchrone proposée avait été fabriquée, des limitations concernant la fiabilité auraient tout de même été applicables. Il est donc important de définir clairement le contexte et les limitations du travail réalisé afin de rapporter des résultats sensés et significatifs.

5.2.1 Environnement de conception et de simulation

Le flot de conception ASIC utilisé dans le cadre de ce projet n'est pas spécifique à Octasic, mais comporte quelques particularités. Étant donné que les circuits intégrés conçus nécessitent une analyse propre au monde asynchrone, leurs outils de CAO sont principalement développés à l'interne. Afin de mettre en contexte de travail de conception, de simulation et d'analyse, les étapes du flot de conception ainsi que les outils de CAO utilisés à l'interne ont été résumés.

La première étape consiste à se baser sur une spécification et des exigences précises afin de programmer dans un langage de description matérielle (HDL) ladite fonctionnalité. Dans le cas de ce projet, les fonctions de la mémoire cache asynchrone ont été programmées en VHDL structurel, mais auraient aussi pu être programmées de manière comportementale.

La seconde étape consiste à compiler et synthétiser le code VHDL avec le logiciel de compilation et de synthèse développé par Octasic : OctGTech (Octasic General Technology). Ce logiciel extrait le fonctionnement logique du VHDL et le synthétise en circuit constitué entièrement de portes logiques optimisées en logique négative. Dans le cadre de ce projet, la conception RTL de la cache a entièrement effectué en entrée schématique au niveau *portes logiques* (« gate-level »). Cela implique que le logiciel OctGTech n'a pas synthétisé la logique se trouvant dans les fichiers VHDL, puisque la logique était déjà décrite en portes logiques génériques. Ensuite, OctGTech traduit les portes logiques génériques avec celles de la technologie CMOS disponible, qui est la librairie CMOS 28nm bulk de ST Microelectronics dans ce cas-ci. L'information contenue dans la librairie CMOS détermine le comportement logique, les capacitances d'entrée, la taille et la disposition physique des cellules logiques utilisées. Suite à cette étape appelée « mapping », un fichier Verilog sans hiérarchie (« flattened ») comprenant l'entièreté du circuit conçu est créé.

La troisième étape consiste à simuler le circuit compilé et synthétisé. Pour pallier le manque de fonctionnalité et d'analyse asynchrone des outils de CAO conventionnels, le logiciel de simulation OctFire a été développé par Octasic. Les bancs de test sont programmés en C, contrairement aux langages TCL ou Python habituellement utilisés. Le simulateur importe d'abord le fichier Verilog sans hiérarchie de l'entité supérieure et les librairies de la technologie CMOS. Puis, des signaux sont appliqués aux ports de l'entité supérieure pour une période de temps déterminée, simulant ainsi le comportement du circuit. Des fichiers contenant les formes d'ondes résultantes et de l'information pertinente (« logs ») sont alors créés. Le simulateur importe les librairies de la technologie CMOS afin d'effectuer des simulations fonctionnelles en fonction des caractéristiques temporelles pré-placement (« pre-layout timing »). Ce dernier peut aussi

extraire de l'information utile telle que les accès mémoire et une estimation de l'énergie dynamique consommée sans placement (sans les charges RC des lignes de transmission).

Lorsque les premières simulations fonctionnelles sont concluantes, la quatrième étape consiste à effectuer un placement préliminaire avec le logiciel HLGen, aussi développé par Octasic. Ce logiciel crée un fichier contenant les positions relatives de tous les composants (cellules logiques) instanciés dans le fichier Verilog. Il est ensuite possible de visualiser le placement généré avec le logiciel LT4B d'Octasic, et de vérifier qu'il n'y a aucun chevauchement entre les cellules du circuit.

Une fois le fichier de placement créé, il est possible d'effectuer une analyse temporelle post-layout. Cette cinquième étape consiste à effectuer une analyse temporelle statique (STA) avec le logiciel Octimize, évidemment développé par Octasic. Le logiciel génère d'abord une estimation Manhattan [1] des lignes de routage entre chaque cellule logique. Cette longueur permet alors d'estimer la charge RC de chaque interconnexion, qui est ensuite ajoutée aux capacitances d'entrée des cellules. En utilisant cette estimation avec d'autres fichiers de configuration, le logiciel Octimize peut analyser des chemins de données en fonction d'un signal d'horloge. Différents rapports sont alors générés, tels que les analyses des temps de « setup » et de « hold », qui permettent de valider si les contraintes temporelles sont respectées entre chaque élément mémoire.

Le flot de conception ASIC est un processus itératif. Lorsque les contraintes appliquées aux logiciels de CAO ne peuvent être atteintes, un processus d'optimisation est enclenché. Ce processus peut être appliqué à toutes les étapes du flot de conception, ce qui implique que la conception ASIC est généralement un travail de longue haleine.

1. La distance Manhattan, ou norme L_1 est la plus courte distance en X et Y entre deux points sur un plan. Étant donné que le routage en VLSI est seulement effectué avec des couches de métal horizontales et verticales, cette estimation est beaucoup plus fiable que la distance euclidienne, qui correspond plutôt à $\sqrt{X^2+Y^2}$.

5.2.2 Caractéristiques des bancs de test

Trois principaux bancs de test ont été développés pour simuler et valider le fonctionnement de la cache d'instruction asynchrone proposée. Le premier est un banc de test individuel incluant seulement la cache d'instruction, avec ou sans interfaces. Le second banc de test intègre le tout dans le processeur ARM asynchrone développé par Octasic. Le troisième et dernier banc de test a été conçu spécialement afin de simuler et comparer les caches synchrones et asynchrones. Ces trois bancs de test sont décrits en détail ci-dessous.

Le premier banc de test permet de valider le fonctionnement de la mémoire cache d'instruction L1 asynchrone indépendamment de son environnement. Dans un premier temps, ce banc ne comprenait aucune interface en entrée ou en sortie. Son développement a été réalisé de façon incrémentale avec le reste de la cache d'instruction asynchrone lors de la phase de conception. Dans un second temps, les interfaces avec le PCBP, IDecode et la mémoire cache L1/L2 ont été intégrés et validés.

Afin de tester le fonctionnement de la cache d'instruction, les comportements du PCBP et de IDecode ont été émulés. Pour ce faire, un espace mémoire de la taille de la cache L1 (32ko) est alloué et contient des instructions ARM v7 aléatoires. Le banc de test génère des adresses PC selon deux modes, soit de manière contiguë ou aléatoire. Dans un premier temps, il est possible de tester la fonctionnalité de pression d'anticipation (« forward pressure ») en envoyant rapidement de nouvelles adresses PC et en arrêtant momentanément le PCBP. Dans un second temps, il est possible de tester la pression de réaction (« back pressure ») de la part du IDecode et de l'interface L1-L2. Après avoir complété la simulation, le banc de test enregistre un fichier « log » comprenant les adresses PC , les instructions résultantes ainsi qu'un échantillon de temps. La validation des simulations est faite en comparant les instructions émulées par les adresses envoyées avec celles retournées par la cache.

Le second banc de test permet de valider le fonctionnement de la cache asynchrone au sein du processeur ARM. Tout d'abord, ce banc de test est important puisqu'il permet dans un premier temps de valider que les interfaces fonctionnent de manière adéquate. Ensuite, il est

aussi possible d'utiliser cette plateforme pour valider que la purge (*flush*) du pipeline s'effectue correctement par rapport au reste du processeur. Finalement, les programmes émulés par le processeur ARM permettent de vérifier la cache L1 retourne toutes les bonnes instructions en fonction de l'adresse PC.

Toutefois, il n'est pas possible de comparer la performance des deux caches L1 au sein de ce banc de test. En effet, le PCBP et le décodeur d'instruction implémentés sont limités en terme de fréquence d'opération et sont synchrones à l'horloge globale. En augmentant la période d'horloge de « L2clk », différents modules tels que le PCBP et IDecode arrêtent tout simplement de fonctionner. Un troisième banc de test a donc été développé pour pallier cette problématique.

Le troisième banc de test consiste à adapter le premier pour qu'il soit possible de tester les caches synchrones et asynchrones en parallèle. Chaque mémoire cache possède ses propres ports d'entrée et de sortie, simule une séquence d'adresse PC identique et accède à la même mémoire L2. Les PCBP et IDecode couplés à chaque cache fonctionnent de manière indépendante et ne sont pas limités en termes de fréquence d'opération. Cela implique donc que le fonctionnement des modules émulant le PCBP et le IDecode sont entièrement gérés par les signaux de requête et de complétion de leur protocole de communication. Le but de ce test est de découpler la cache des modules environnant du ARM, présentement non-optimaux, afin de simuler ces derniers de manière logicielle. Ainsi, il sera possible de déterminer les performances maximales théoriques des deux caches, tout en simulant un contexte réel d'utilisation.

Grâce à ce troisième banc de test, il est possible d'extraire à la fois le temps d'exécution et la consommation d'énergie pour une routine d'adresse PC prédéfinie. De plus, une variable a été ajoutée à ce banc de test, soit la proportion des adresses PC qui résulte d'une mauvaise prédiction du PCBP. Il est donc possible de faire varier le nombre d'accès mémoire en *flush* et de simuler son effet sur la temps d'exécution et la consommation énergétique.

5.2.3 Limitations

Les limitations sont principalement en terme de fonctionnalité et de validation. Une des principales limitations en terme de fonctionnalité est que la cache asynchrone ne fait pas partie d'un processeur ARM v7 complet. En effet, elle a été intégrée au sein d'une version sommaire, voire préliminaire, de ce dernier. Afin de répondre aux critères de la spécification ARM, une unité de gestion mémoire (MMU) doit être implémentée entre le PCBP et la cache d'instruction. Ce module permet de traduire les adresses virtuelles utilisées par les programmes (ultimement exécutés par le système d'opération (OS)) en adresses physiques pour les mémoires d'instructions et de données. La cache d'instruction implémentée dans le cadre de ce projet n'utilise qu'un adressage physique simple.

Une autre limitation est la mémoire cache L2 utilisée aux fins de simulation est implémentée de manière strictement logicielle. En effet, lorsqu'une requête est envoyée à la cache L2 suite à un « miss » à la cache L1, la nouvelle ligne de cache et son étiquette sont retournées systématiquement après un cycle d'horloge. Dans une implémentation réelle, il faudrait d'abord déterminer la donnée demandée est contenue dans la mémoire cache L2 avant de procéder à son envoi vers la L1. Certains des résultats obtenus, tels que le débit et l'énergie consommée, pourraient varier considérablement en fonction de ce type de scénario.

Une autre limitation importante existe au niveau de la validation du placement de la cache et de ses mémoires RAM. Le placement du pipeline, particulièrement la section des mémoires RAM de données, n'a pas été validé suite au routage. Seule une analyse *post-placement* a été réalisée, à défaut de pouvoir effectuer une analyse « post-routing » complète. En effet, l'analyse de la cache asynchrone est basée sur un placement manuel préliminaire et établit l'analyse temporelle sur une estimation du routage avec la distance Manhattan. Cette analyse permet dans un premier temps d'établir une approximation de l'énergie consommée par la cache d'instruction et dans un second temps une estimation de la congestion attribuable au routage. L'analyse de la congestion est plutôt basée sur la distance minimale euclidienne et agit à titre indicatif seulement. Ainsi, il est possible de déterminer la congestion en superposant

d'abord toutes les droites euclidiennes, puis en repérant les zones denses où plusieurs droites s'intersectent.

Malgré ces analyses préliminaires, il n'est toutefois pas possible de valider la faisabilité de cette cache asynchrone sans effectuer le routage physique avec un logiciel de CAO. Finalement, l'absence de routage final implique également qu'aucune implémentation physique n'a été fait sur une même puce avec le processeur ARM asynchrone. Un travail d'une telle ampleur n'est pas réalisable dans le cadre de ce projet de maîtrise.

5.3 Résultats des bancs de tests individuels

Cette section du chapitre présente les résultats obtenus lors de l'intégration de la cache asynchrone dans les deux bancs de tests individuels, tels que décrits à la section 5.2.2. Le premier banc de test intègre valide le fonctionnement de la cache hors du processeur en simulant les échanges d'information à ses interfaces. Cette étape permet d'abord de vérifier que la cache répond aux spécifications et permet ensuite d'optimiser son architecture avant de l'intégrer dans le processeur ARM d'Octasic. Le second banc de test permet d'émuler le fonctionnement réel processeur ARM en y intégrant la nouvelle cache asynchrone. Cette étape permet d'effectuer une seconde validation de la conception. De plus, l'intégration dans le processeur permet d'ajuster certains éléments qui n'étaient pas triviaux lors des tests individuels, soit l'implémentation de la séquence de purge et de la pression de réaction imposée par IDecode et le PCBP.

Les résultats obtenus lors des simulations sont présentés dans cette section de la manière suivante. Premièrement, le placement préliminaire effectué afin d'estimer les délais dus au routage est analysé. Deuxièmement, les principales contraintes temporelles de la STA sont énoncées puis l'évaluation de la vitesse de traitement de chaque étage du pipeline est faite. Troisièmement, le débit maximum aux interfaces est déterminé et analysé. Ces résultats seront globalement comparés avec ceux de la cache synchrone à la prochaine section du chapitre.

5.3.1 Placement préliminaire

La figure 5.1 présente une vue globale du placement de la cache d'instruction L1 asynchrone. À partir de cette vue d'ensemble, il est possible de discerner les trois principales parties de la cache.

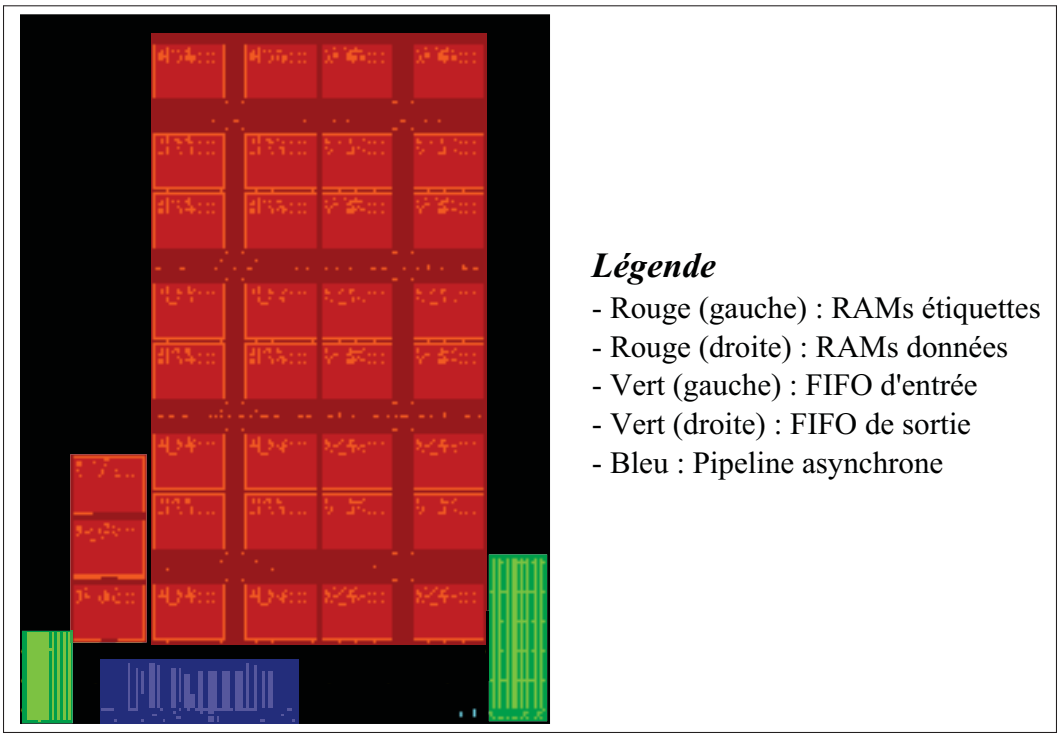


Figure 5.1 Disposition physique de la cache d'instruction L1 asynchrone

Tout d'abord, les modules en vert sont les FIFO d'entrée et de sortie, respectivement à gauche et à droite. Le FIFO d'entrée est sensiblement plus petit, puisqu'il ne garde en mémoire que l'adresse du PC et les données auxiliaires, pour un total de 8 x 73bits. Quant au FIFO de sortie, il garde aussi en mémoire les deux instructions de 32bits , pour un total de 8 x 137bits. La disposition des FIFOs a été faite en fonction des 8 tranches (« slices ») verticales de données de 73bits et 137bits. Le FIFO d'entrée a une aire de 4900 μm^2, alors que le FIFO de sortie occupe environ deux fois plus d'espace, avec une aire de 10000 μm^2.

Ensuite, il est possible de distinguer en rouge les mémoires RAM précompilées, qui ont une taille individuelle de 256 x 32bits. Les trois mémoires RAM à gauche contiennent les étiquettes, les trente-deux mémoires RAM à droite contiennent les données de la cache d'instruction. L'espacement entre les blocs RAM est nécessaire pour effectuer le routage des signaux d'accès aux mémoires, incluant les répéteurs et le « clock-gating ». Il est important de mentionner que les mémoires RAM utilisent les quatre premières couches de métal sur un total de sept (Microelectronics), ce qui limite le routage au-dessus de ces dernières. Les mémoires RAM des étiquettes ont une aire de 18000 μm^2, tandis que les mémoires RAM de données occupent plus de dix fois d'espace, soit 225000 μm^2. Finalement, le pipeline de la cache d'instruction est représenté en bleu. Cette section intègre les 6 étages de la cache, ainsi que l'interface L1-L2 et le module de gestion de la purge (*flush*).

La figure 5.2 présente une vue en détail du placement du pipeline asynchrone. La section horizontale supérieure contient le chemin de données tandis que la partie inférieure comprend le contrôle du pipeline, soit les modules de gestion des jetons.

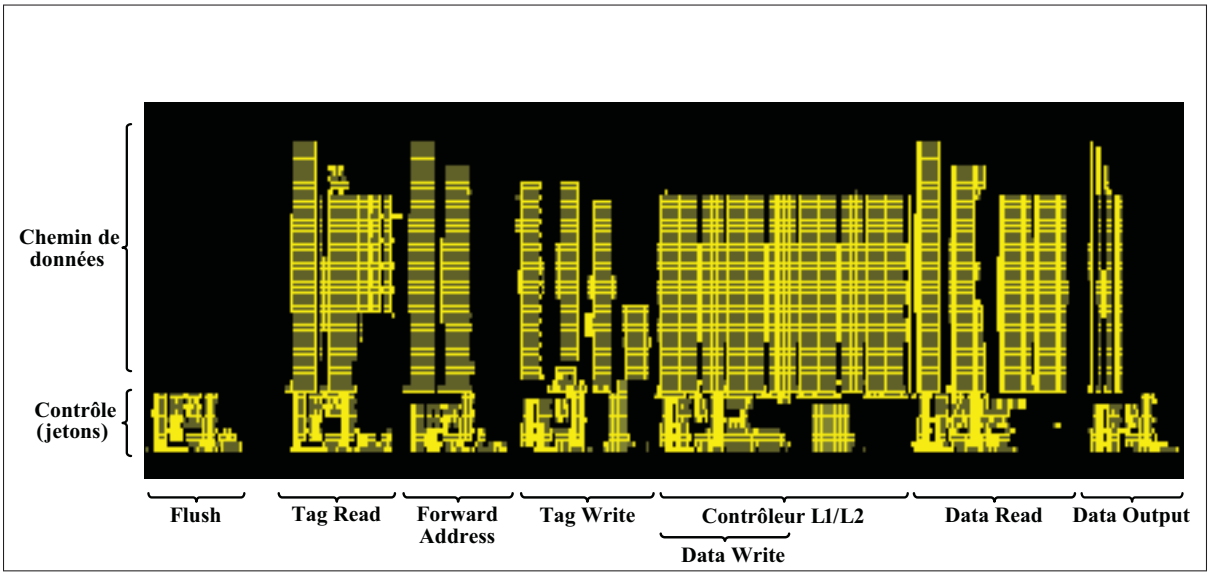


Figure 5.2 Disposition physique du pipeline de la cache d'instruction L1 asynchrone

Le pipeline asynchrone est placé de façon à respecter l'ordre séquentiel des accès à la mémoire cache d'instruction. Tout d'abord, le bloc placé à l'extrémité gauche est le module de gestion de la purge du pipeline. Ce module reçoit les signaux *flush* générés au sein du FIFO d'entrée et propage ensuite la séquence de réinitialisation aux autres étages du pipeline. Ensuite, suivent les étages « Tag Read » et « Forward Address ». L'étage « Tag Read » contient les DFFs de l'adresse PC et des données auxiliaires ainsi que les lignes de transmission pour l'accès aux mémoires RAM des étiquettes. Cet étage contient aussi les quatre modules de comparaison de l'étiquette, qui génèrent le signal indiquant une correspondance « cache hit ». L'étage « Forward Address » contient les DFFs de l'adresse PC et des données auxiliaires, mais implémente aussi le multiplexeur responsable de la fourche et le générateur de voie pseudo-aléatoire.

Puis, les étages « Tag Write » et « Data Write » sont placés, suivis de l'interface L1-L2. L'étage « Tag Write » garde en mémoire la nouvelle étiquette et son adresse physique et gère l'écriture aux mémoires RAM des étiquettes. L'étage « Data Write » gère l'écriture séquentielle de la nouvelle ligne de cache dans les mémoires RAM de données à partir du FIFO de l'interface L1-L2. L'interface L1-L2 contient les démultiplexeurs en entrée, les DFFs contenant la ligne de cache de 256bits et les multiplexeurs de sortie.

Finalement, l'étage « Data Read » et « Data Output » sont placés à la suite de l'interface L1-L2, à la sortie du pipeline asynchrone. L'étage « Data Read » contient les DFFs de l'adresse PC, des données auxiliaires et des deux instructions de 32bits. Cet étage implémente un joint pour déterminer la source des deux instructions, qui proviennent soit du FIFO de l'interface L1-L2 ou des mémoires RAM de données de la cache. L'étage « Data Output » n'implémente que des lignes de transmission pour écrire dans la FIFO de sortie, lorsque ce dernier est disponible.

Le tableau 5.1 présente les dimensions de chaque module ou étage du pipeline et leur proportion en fonction de la taille totale du pipeline. Les valeurs représentées dans le tableau ont été calculées en fonction de l'aire effective de chaque circuit et ne tiennent pas compte de la surface de routage estimée.

Tableau 5.1 Taille physique des modules du pipeline

Module / Étage	**Aire** (μm^2)	**Proportion** (%)
Tag Read	1555	15,7
Forward Address	1650	16,67
Data Read	1950	19,7
Data Output	1050	10,6
Tag Write	330	3.33
Data Write	385	3,88
Flush	250	2,53
Interface L1-L2	2730	27,58
Total	9900	100

Il est à noter qu'une brève analyse de congestion a été faite suite au placement de la cache d'instruction asynchrone. Les sections les plus congestionnées sont l'interface L1-L2 et l'étage « Tag Read ». L'interface est la plus congestionnée puisqu'elle est relativement dense et intègre un grand nombre de signaux avec les démultiplexeurs et multiplexeurs. L'étage « Tag Read » collige les accès en lecture et en écriture aux mémoires RAM des étiquettes, en plus de procéder à la comparaison de ces derniers. Somme toute, l'outil de vérification n'a pas signalé d'endroit où la congestion était critique, mais ces résultats demeurent néanmoins préliminaires.

5.3.2 Vitesse d'opération par étage du pipeline

L'évaluation de la vitesse d'opération maximale de chaque du pipeline se fait en deux étapes itératives. La première étape consiste à valider que chaque étage du pipeline fonctionne correctement et respecte les contraintes temporelles imposées par l'architecture asynchrone du pipeline. Un premier travail doit être fait soit au niveau du placement, de l'ajustement de « l'effort logique » au sein du circuit ou de l'optimisation de la logique combinatoire. Plusieurs itérations peuvent être nécessaires afin d'atteindre les contraintes temporelles.

Une fois que les contraintes minimales sont satisfaites, la deuxième étape consiste à optimiser davantage le circuit conçu. Toutefois, un équilibre entre fréquence d'opération et efficacité énergétique doit être respecté lors de ce processus. Il est donc souhaitable d'éviter le plus possible l'utilisation de portes logiques ayant une tension de seuil bas (« low-threshold ») et

les portes logiques ayant une puissance de transmission (« driving strength ») élevée. Cette deuxième étape est aussi itérative, puisqu'il faut continuer à valider les modifications apportées au circuit grâce à l'analyse STA. Finalement, il est nécessaire de valider le fonctionnement du circuit en simulation avec les nouvelles optimisations.

La première contrainte temporelle inhérente à la première étape de l'analyse temporelle STA est la suivante. L'équation 5.1 permet de valider le fonctionnement interne de chaque étage du pipeline, plus particulièrement du module de jeton et de la génération de l'horloge. En effet, un des chemins critiques reliés au module de jeton est la génération du signal de disponibilité de chaque étage.

L'opération normale d'un étage du pipeline consiste à attendre que toutes les conditions d'utilisation du module de jetons soient valides afin de pouvoir accéder à cet étage. Lorsque le module de gestion détient son jeton, détecte une requête active à cet étage et que ce dernier est disponible, le module génère une horloge et un signal d'activation. Le signal de disponibilité représentée à l'équation 4.2, indiquant que l'étage est présentement utilisé, doit revenir assez rapidement au module de jetons pour éviter de générer un second signal d'horloge. Cependant, le module de gestion des jetons intègre une chaîne de délai ajustable sur le chemin de retour du jeton, tel qu'illustré à la figure 3.7. Ce délai permet d'imposer une latence entre l'utilisation de l'étage du pipeline et la prochaine évaluation des conditions du jeton.

Afin d'opérer correctement, chaque étage du pipeline doit d'abord valider que l'équation 5.1 est respectée lors de l'analyse (STA).

$$T_{disponibilité} < T_{délai\_jeton} \tag{5.1}$$

La seconde contrainte temporelle de l'analyse temporelle STA est la suivante. L'équation 5.2 permet de valider la synchronisation entre un étage de pipeline et le subséquent, dont la gestion est assurée par les éléments Click. Ce chemin critique est fondamental et gouverne tous les circuits séquentiels, qu'ils soient synchrones ou asynchrones. Cette contrainte sert à valider

que la logique combinatoire est stabilisée et que le temps de configuration (« setup time ») des DFFs est respecté.

L'opération normale d'un étage du pipeline consiste à attendre que toutes les conditions d'utilisation du module de jetons soient valides afin de pouvoir accéder à cet étage. Lorsque toutes les conditions d'utilisation du module de gestion du jeton sont valides, ce dernier génère une horloge et un signal d'activation. Les données à l'entrée de l'étage sont mises en mémoire, puis passent par le nuage combinatoire pour atteindre le prochain étage du pipeline. Simultanément, le signal de requête pour le prochain étage est généré par la mise à jour de la « toggle-flop » de l'étage.

Théoriquement, il faut s'assurer que le signal de requête à l'étage subséquent arrive un peu après les données, afin de satisfaire l'équation 2.1. Toutefois, il est possible de modifier la contrainte temporelle afin de prendre en compte le temps d'évaluation du module de gestion des jetons. L'ajout de ce décalage (« skew ») permet d'optimiser les temps de synchronisation entre les étages, réduisant davantage la latence globale du pipeline.

Ainsi, chaque étage du pipeline doit respecter l'équation 5.1 suite à l'analyse (STA) pour opérer correctement.

$$T_{prochaine\_requête} > (T_{combinatoire} + T_{setup} - T_{évaluation\_jeton}) \tag{5.2}$$

L'analyse STA est effectuée de la manière suivante. Une liste de points d'intérêts (« end-points ») qui devront être recensés par l'outil de STA est d'abord définie. L'événement déclencheur, qui détermine le temps initial de l'analyse, est la génération d'une transition du jeton. Cet événement simule l'utilisation de l'étage de pipeline suite à l'activation du module de gestion de son jeton.

Pour la première contrainte, les points d'intérêt sont les signaux de transition du jeton et de disponibilité de l'étage. Pour la deuxième contrainte, les points d'intérêt sont déterminés par

l'étage suivante, soit les données à l'entrée des DFFs et le signal de requête respectif. Pour chaque étage de pipeline, il est ensuite possible de vérifier que les équations 5.1 et 5.2 sont respectées.

Le tableau 5.2 collige les vitesses maximales d'opération pour chaque étage du pipeline suite à l'analyse STA.

Tableau 5.2 Vitesse d'opération des modules du pipeline

Module / Étage	$\mathbf{T_{COMB}}$ (ps)	**Chemin critique**
Tag Read	1700	Condition cache *miss* ou *hit*.
Forward Address	650	Vers contrôleur L1-L2 : calcul de la prochaine voie .
	300	Vers Data Read : génération de la requête
Data Read	450	Du contrôleur L1-L2 : lignes de transmission.
	2100	Du réseau de RAMs : aller-retour pipeline vers RAMs.
Data Output	800	Génération du signal de disponibilité (FIFO plein).
Tag Write	910	Écriture dans les RAMs (pire cas).
Data Write	1370	Écriture dans le réseau de RAMs (pire cas).
Flush	1100	Lignes de transmission jusqu'à la dernière RAM.

Le module du pipeline qui gère la procédure de *flush* ne restreint pas directement la vitesse d'opération du pipeline et n'a donc pas été ajouté au tableau 5.2. Ce module influence plutôt la fréquence d'horloge globale utilisée par les modules synchrones. Cela est principalement dû au fait que le signal de réinitialisation est synchrone à l'horloge globale et est appliqué pendant un cycle d'horloge. La contrainte temporelle consiste à assurer que le signal *flush* peut être propagé à travers la cache à l'intérieur d'une période de l'horloge. Le réseau de distribution du signal *flush* est semblable à celui de l'horloge globale, mais n'utilise que des portes logiques ayant une puissance de transmission faible afin de limiter la consommation énergétique. Néanmoins, les résultats de l'analyse STA démontrent qu'il est possible d'appliquer à tous les modules dépendant dans une période de 1100ps. Une horloge globale ayant une fréquence nominale avoisinant 900MHz pourrait donc être utilisée.

Finalement, il est important de mentionner que les résultats de la table 5.2 sont issus d'une analyse STA configurés avec un déclassement (« derating ») de 15%. De plus, les librairies de cellules standard pour la STA reproduisent un cas limite d'opération des transistors, soit une température de 125 °C et une sous-tension de 0.9V.

5.3.3 Vitesse d'opération aux interfaces de la cache asynchrone

Les interfaces de la cache d'instruction permettent présentement de synchroniser les signaux entre les domaines synchrones et asynchrones. L'analyse STA faite est donc basée sur l'optimisation du FIFO d'entrée, du FIFO de sortie et le contrôleur L1-L2 en fonction de l'horloge globale « L2clk ». Les contraintes temporelles applicables à ces modules sont donc les mêmes que celles évoquées aux équations 2.1 et 2.2.

Tableau 5.3 Vitesse d'opération des modules du pipeline

Interface	**T_{COMB}** (ps)	**Chemin critique**
FIFO d'entrée	700	Comparaison des pointeurs (FIFO plein).
FIFO de sortie	850	Comparaison des pointeurs (FIFO vide).
Interface L1-L2	800	Distribution et multiplexage des signaux d'activation.

Le temps de traitement maximal ainsi que la description des chemins critiques pour les trois interfaces sont relevés dans le tableau 5.3. Il est à noter que l'analyse STA n'inclut pas les délais engendrés par le routage des lignes de transmission avec les interfaces de la cache d'instruction. Une analyse globalement au sein du processeur ARM et l'intégration d'un modèle physique de la cache L2 permettrait de mieux estimer ces délais.

5.4 Comparaison avec la cache d'instruction synchrone

Cette section présente les résultats obtenus lors du troisième banc de test intégrant les caches synchrone et asynchrone en parallèle, tel que décrit à la section 5.2.2. Ce banc de test couple chaque cache d'instruction à un port dédié et simule une séquence d'adresse PC aléatoire identique. Cela permet d'émuler le fonctionnement de la cache d'instruction sans être contraints

à la fréquence d'opération des modules extérieurs, puisque leur fonctionnement est généré par le simulateur logiciel. Chacune des caches d'instruction fonctionne donc à leur fréquence d'opération maximale . De plus, une seconde variable a été ajoutée à ce banc de test, soit la proportion de requêtes mémoire résultant d'une mauvaise prédiction du PCBP. Cette nouvelle variable permet d'adresser l'effet des purges du pipeline sur les performances globales des caches d'instruction.

Les résultats obtenus lors des simulations sont présentés de la façon suivante. Tout d'abord, une comparaison entre les tailles physiques des deux caches d'instruction est faite, suite à leur placement préliminaire respectif. Ensuite, les performances des caches synchrone et asynchrone sont analysées et comparées en terme de débit d'opération. Finalement, une comparaison est faite entre les deux caches du point de vue de l'efficacité énergétique. Une analyse globale portant sur tous les résultats obtenus lors des simulations des bancs de test sera fait à la prochaine section du chapitre.

5.4.1 Taille physique

Les résultats obtenus concernant la taille physique du pipeline des caches synchrones et asynchrones sont colligés dans le tableau 5.4. L'aire totale du pipeline ainsi que la proportion occupée par ce dernier sans l'interface L1-L2 et la logique de contrôle ont été extraites suite au placement grâce au logiciel LT4B d'Octasic. Il est à noter que seulement la taille du pipeline est comparée entre les deux caches d'instruction, étant donné que le placement des mémoires RAM des étiquettes et des données est identique.

Tableau 5.4 Comparaison de la taille du pipeline des caches synchrone et asynchrone

Pipeline	**Synchrone**		**Asynchrone**	
	Aire (μm^2)	(%)	Aire (μm^2)	(%)
Total	11185	100	9900	100
(Sans Interface L1-L2)	8455	75,6	7170	72,4
(Contrôle)	375	3,35	915	9,24

De plus, il a été possible d'obtenir un décompte total du nombre d'éléments mémoires implémenté pour chaque pipeline. Sans l'interface L1-L2, le pipeline asynchrone compte un total de 573 DFFs et 50 loquets transparents, tandis que le pipeline synchrone compte 687 DFFs et 53 loquets transparents. L'architecture de la cache asynchrone proposée intègre environ 16% moins de cellules mémoires que sa version synchrone.

Les résultats obtenus suite au placement des caches permettent d'estimer grossièrement leur comportement énergétique dynamique et statique. La taille du pipeline et le nombre d'éléments mémoire permettront ensuite de valider les chiffres obtenus lors de l'analyse énergétique.

5.4.2 Performance en terme de vitesse d'exécution

En utilisant le banc de test comparatif, il a été possible d'extraire le temps d'exécution nécessaire pour lire une série d'instructions dans les caches asynchrone et synchrone. Tel qu'indiqué à la section 5.2.2, le banc de test génère de façon pseudo-aléatoire une série d'adresses PC puis détermine si cette adresse résulte d'une mauvaise prédiction du PCBP. En utilisant un générateur pseudo-aléatoire de type Bernouilli, la proportion des adresses PC en *flush* peut être ajustée précisément. L'utilisation du même état initial des générateurs pseudo-aléatoire, il est possible d'obtenir exactement les mêmes séquences de lecture pour les deux caches d'instruction.

Le premier graphique 5.3 illustre le temps d'exécution moyen pour effectuer N accès à la mémoire, variant selon $\mathrm{N} = \{10\mathrm{k}, 50\mathrm{k}, 100\mathrm{k}, 500\mathrm{k}, 1\mathrm{M}\}$. Chaque accès en mémoire lit deux instructions de 32 bits, puisque les caches d'instructions ont une architecture « dual-fetch ». Afin de préserver une certaine lisibilité, ce graphique n'illustre que les valeurs moyennes obtenues pour une proportion d'accès P nécessitant une purge, variant selon $N = \{0\%, 10\%, 20\%\}$. Sur le graphique 5.3, les valeurs obtenues pour la cache asynchrone sont représentées par les courbes en pointillé avec les cercles. Les valeurs obtenues pour la cache synchrone sont plutôt représentées par les courbes pleines avec les carrés.

Le second graphique 5.4 compare les temps d'accès moyens des deux caches d'instruction pour un point d'opération en particulier. Pour ce faire, le nombre d'accès à la mémoire est fixé

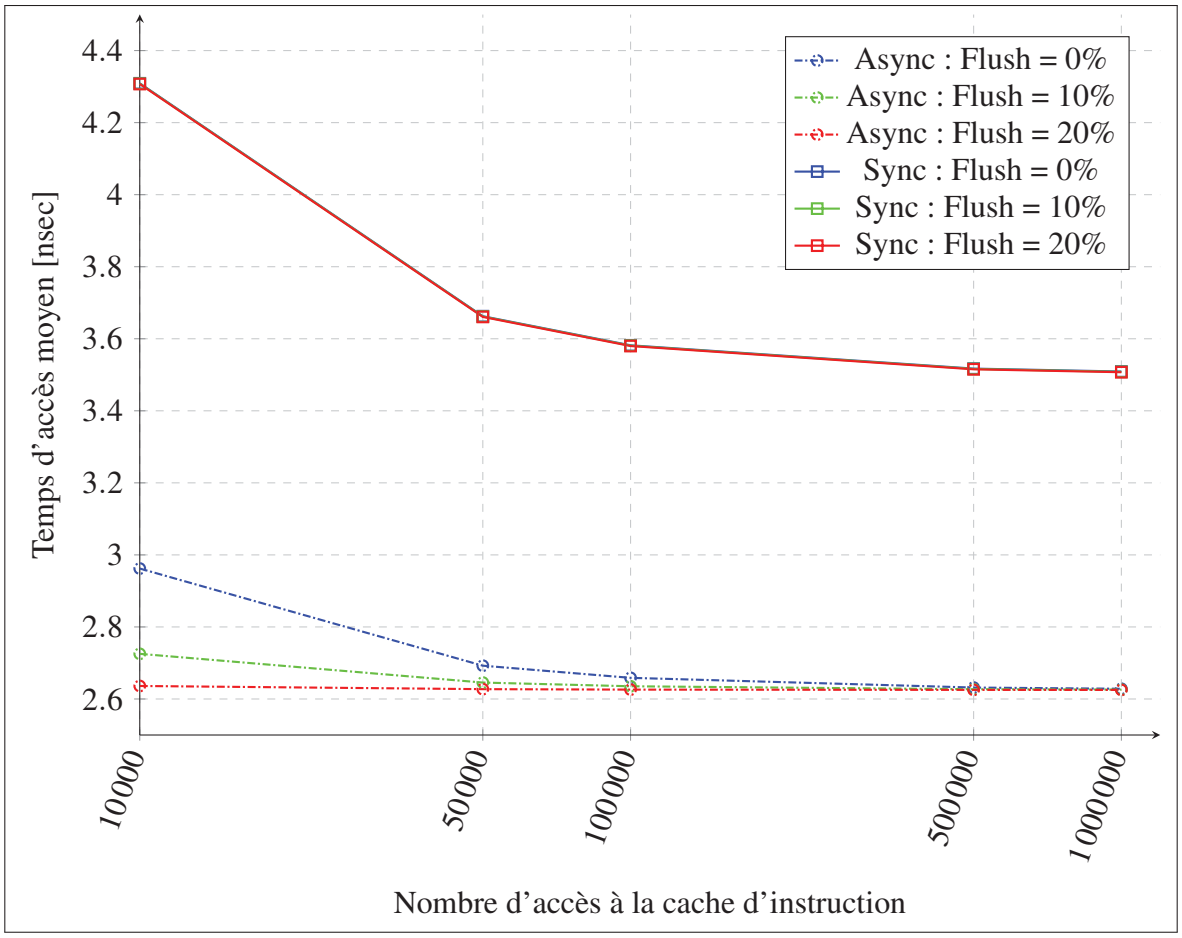


Figure 5.3 Comparaison du temps d'accès moyen à la mémoire en fonction du nombre d'accès et de la proportion en *flush*

à une valeur démontrant l'état transitoire ou permanent de la cache, soit respectivement 10k et 1M d'accès. Un petit nombre d'accès mémoire implique que la plupart de ceux-ci provoquent un « cache miss » et qu'une requête à la cache L2 doit être faite. En contrepartie, un grand nombre d'accès mémoire illustre plutôt le comportement de la cache en régime permanent, lorsque la majeure partie des lignes de cache contenant les instructions sont stockées dans leur mémoire interne. Pour chaque point d'opération, il est possible d'observer le temps d'accès moyen à la mémoire en fonction du nombre d'accès issus d'une mauvaise prédiction du PCBP. Les deux courbes en bleu illustrent les performances des caches en régime transitoire, alors que les courbes en rouge illustrent leurs performances en régime permanent. Les valeurs obtenues

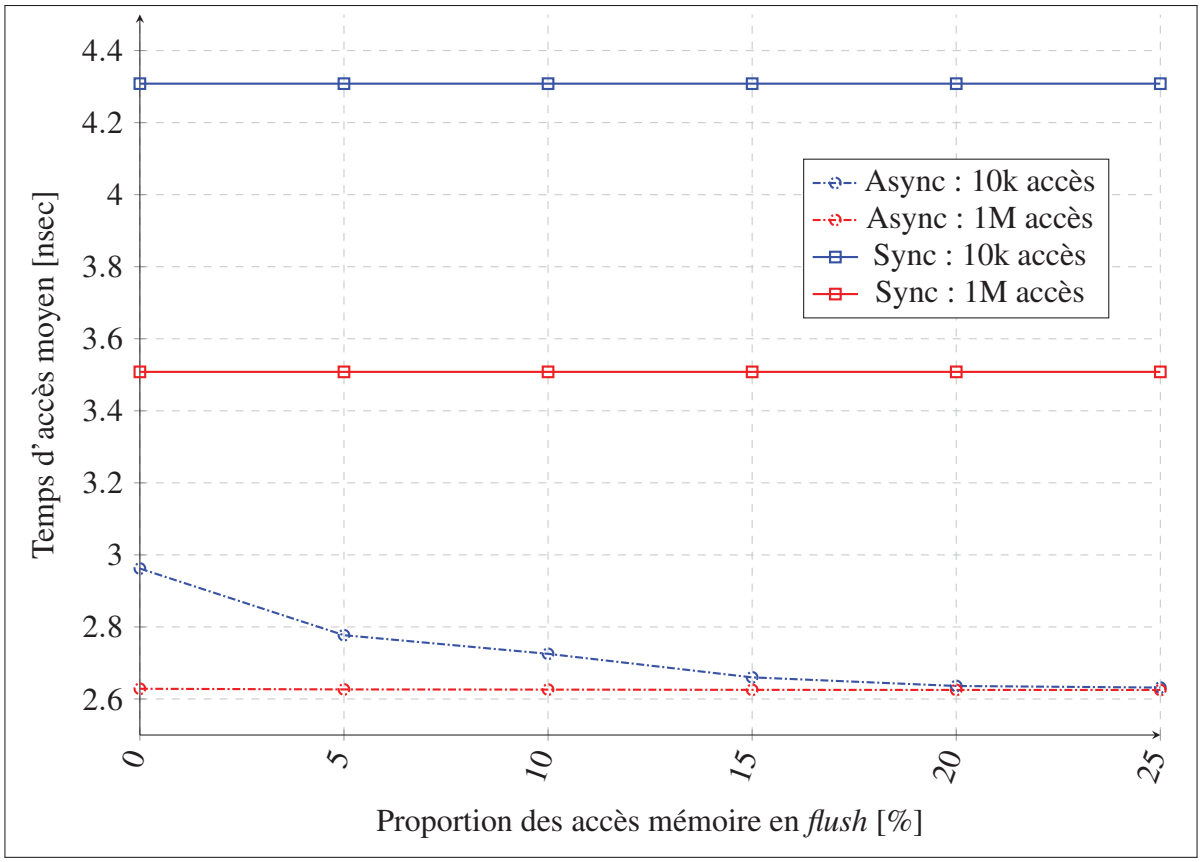


Figure 5.4 Comparaison du temps d'accès moyen pour 10k et 1M d'accès mémoire en fonction de la proportion d'accès en *flush*

pour la cache asynchrone sont représentées par les courbes en pointillés avec les cercles, tandis que la cache synchrone est représentée par les courbes pleines avec les carrés.

Le tableau récapitulatif 5.5 regroupe tous les temps moyens d'accès obtenus lors du banc de test comparatif. Ce tableau permet d'évaluer de quantifier le gain en performance en comparant l'utilisation d'une cache d'instruction asynchrone et synchrone. La valeur ΔT illustre le pourcentage de temps moyen sauvé lors des accès mémoire.

5.4.3 Performance en terme d'efficacité énergétique

Un des buts inhérents de ce projet est de concevoir une architecture de mémoire cache efficace d'un point de vue énergétique. Le banc de test comparatif est particulièrement utile puisqu'il

Tableau 5.5 Comparaison du temps d'accès moyen à la mémoire en fonction de la proportion d'accès en *flush*

Nb instructions	Flush (%)	Temps d'accès moyen (nsec)		ΔT (%)
		Synchrone	Asynchrone	
10000		4,31	2,96	31,2
50000		3,66	2,69	26,5
100000	0	3,58	2,66	25,8
500000		3,52	2,63	25,2
1000000		3,51	2,63	25,1
10000		4,31	2,78	35,5
50000		3,66	2,66	27,5
100000	5	3,58	2,64	26,3
500000		3,52	2,63	25,3
1000000		3,51	2,63	25,1
10000		4,31	2,73	36,7
50000		3,66	2,65	27,7
100000	10	3,58	2,64	26,4
500000		3,52	2,63	25,3
1000000		3,51	2,63	25,1
10000		4,31	2,66	38,3
50000		3,66	2,63	28,1
100000	15	3,58	2,63	26,6
500000		3,52	2,63	25,3
1000000		3,51	2,63	25,2
10000		4,31	2,64	38,8
50000		3,66	2,63	28,2
100000	20	3,58	2,63	26,7
500000		3,52	2,63	25,3
1000000		3,51	2,63	25,2
10000		4,31	2,63	38,9
50000		3,66	2,63	28,3
100000	25	3,58	2,63	26,7
500000		3,51	2,63	25,2
1000000		3,51	2,63	25,2

permet d'estimer l'énergie consommée lors d'une séquence de lecture d'instructions dans les caches asynchrone et synchrone. Ce banc de test est identique à celui utilisé pour déterminer le temps d'accès mémoire moyen. Il est donc possible de faire varier la proportion des accès

mémoire résultant d'une mauvaise prédiction du PCBP, afin d'étudier les répercussions sur l'efficacité énergétique des caches.

Le premier graphique 5.5 illustre l'énergie consommée en μJ pour effectuer N accès à la mémoire, variant selon $\mathrm{N} = \{10\mathrm{k}, 50\mathrm{k}, 100\mathrm{k}, 500\mathrm{k}, 1\mathrm{M}\}$, et ainsi lire deux instructions. Encore une fois, ce graphique n'illustre que les valeurs obtenues pour une proportion d'accès P nécessitant une purge, variant selon $N = \{0\%, 10\%, 20\%\}$, afin de préserver une certaine lisibilité. Sur le graphique, les valeurs obtenues pour la cache asynchrone sont représentées par les courbes en pointillé avec les cercles. Les valeurs obtenues pour la cache synchrone sont plutôt représentées par les courbes pleines avec les carrés.

Les deux graphiques 5.6 et 5.7 comparent l'énergie consommée des deux caches d'instruction pour un point d'opération en particulier. Encore une fois, le nombre d'accès à la mémoire est fixé à une valeur démontrant l'état transitoire ou permanent de la cache, soit respectivement 10k et 1M d'accès. Alors que le petit nombre d'accès mémoire reproduit le fonctionnement de cache en régime transitoire, un grand nombre d'accès mémoire illustre plutôt le comportement de la cache en régime permanent.

Pour chaque point d'opération, il est possible d'observer l'énergie consommée par les caches synchrone et asynchrone en fonction du nombre d'accès mémoire issus d'une mauvaise prédiction du PCBP. Les deux courbes en bleu illustrent les performances des caches en régime transitoire, alors que les courbes en rouge illustrent leurs performances en régime permanent. Les valeurs obtenues pour la cache asynchrone sont représentées par les courbes en pointillés avec les cercles, tandis que la cache synchrone est représentée par les courbes pleines avec les carrés.

Le graphique représenté à la figure 5.6 compare l'efficacité énergétique des caches pour un point d'opération en régime transitoire, soit dix mille accès mémoires.

Le graphique de la figure 5.7 compare l'efficacité énergétique des caches pour un point d'opération en régime permanent, soit un million d'accès mémoires.

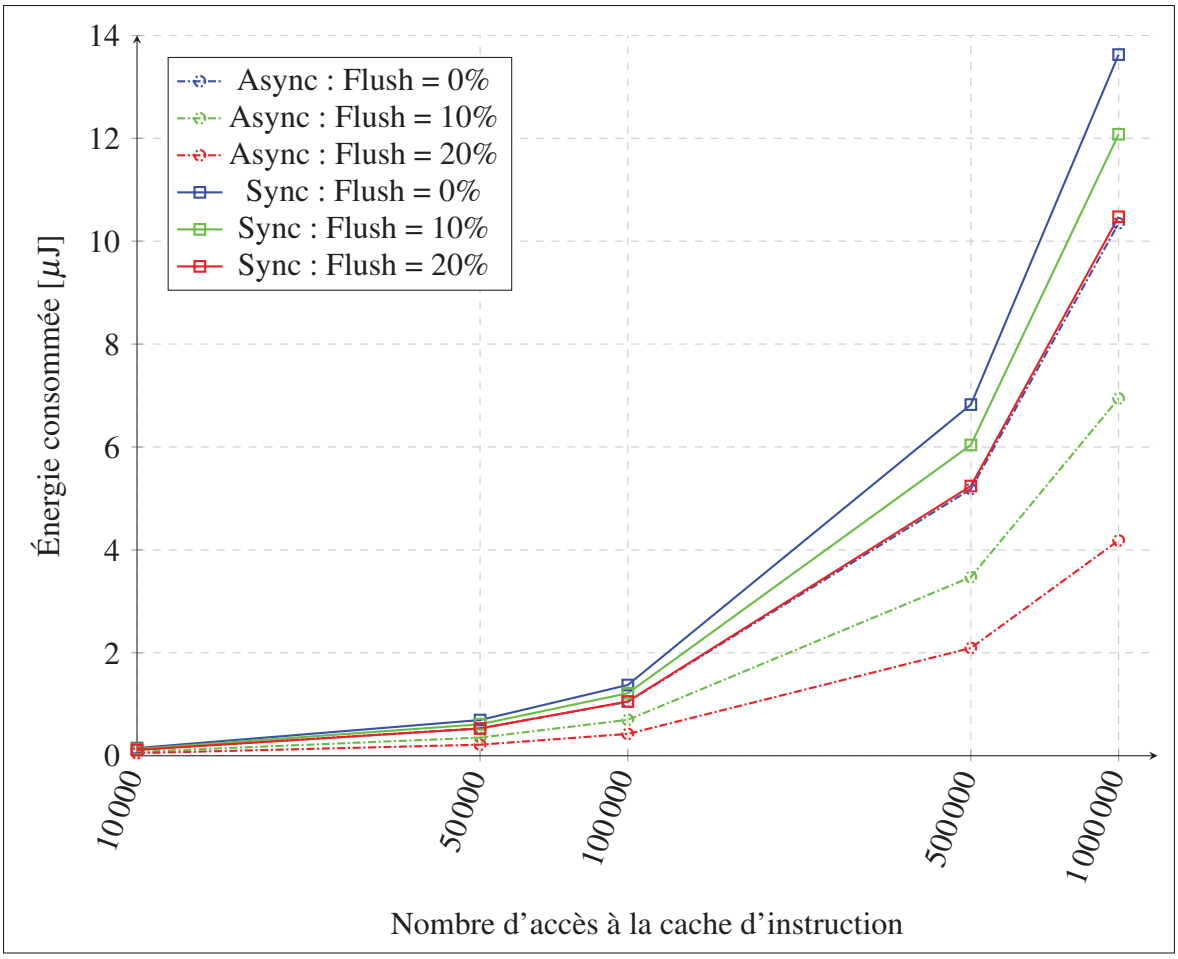


Figure 5.5 Comparaison de l'énergie consommée en fonction du nombre d'accès mémoire et de la proportion en *flush*

Le tableau récapitulatif 5.5 regroupe les différentes quantités d'énergie consommée relevées lors du banc de test comparatif. Ce tableau permet d'évaluer de quantifier le gain en efficacité énergétique en comparant l'utilisation d'une cache d'instruction asynchrone ou synchrone. La valeur ΔE illustre le pourcentage d'énergie épargnée lors des accès à la mémoire.

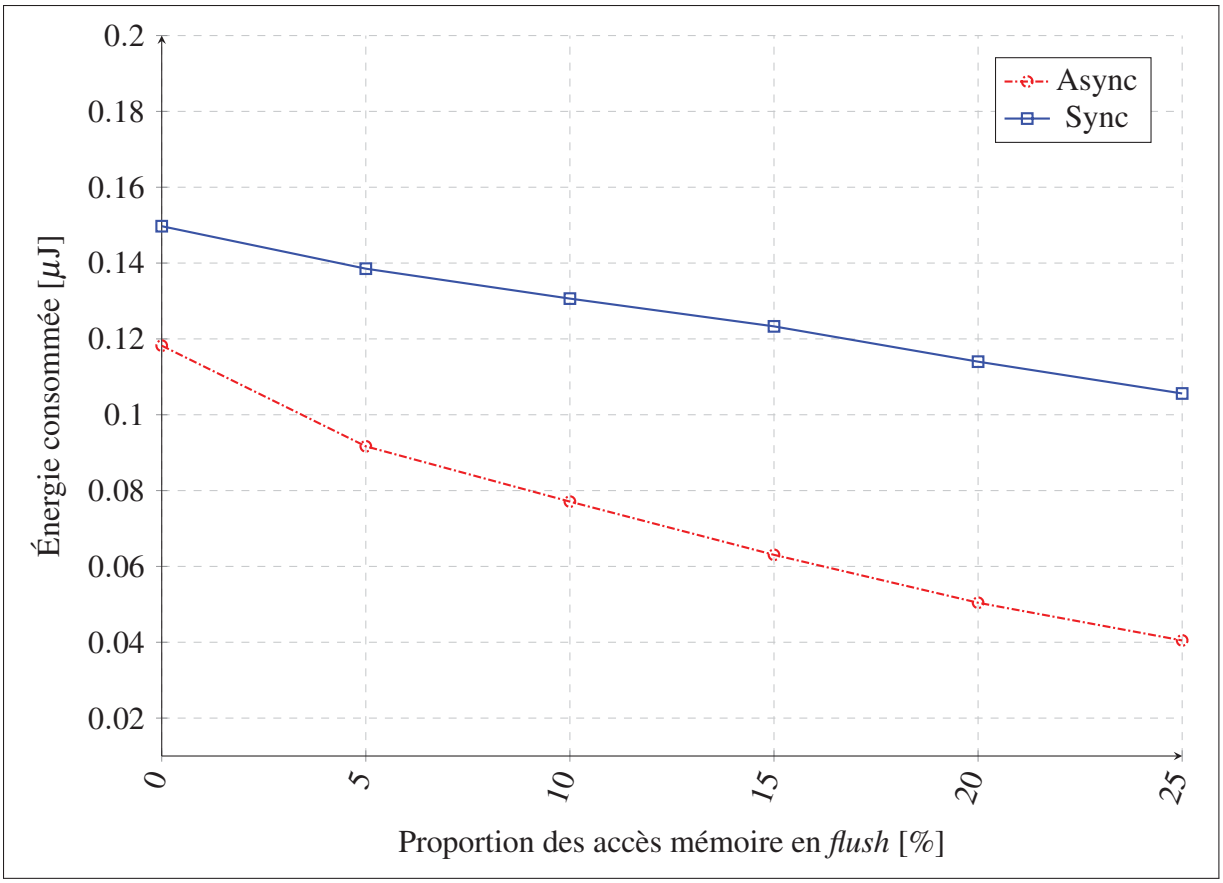


Figure 5.6 Comparaison de l'énergie consommée pour 10k d'accès mémoire en fonction de la proportion d'accès en *flush*

5.5 Analyse des résultats

Cette section présente l'analyse des résultats obtenus pour les bancs de test individuel et comparatif des sections 5.3 et 5.4. Dans un premier temps, une analyse individuelle est réalisée et se concentre sur le placement préliminaire de la cache asynchrone, la vitesse d'opération des différents étages du pipeline et sur le débit résultant aux interfaces de la cache. Dans un deuxième temps, une analyse comparative met en perspective les différences entre les caches synchrone et asynchrone en terme d'aire occupée sur la puce, de temps d'accès mémoire moyen et d'efficacité énergétique.

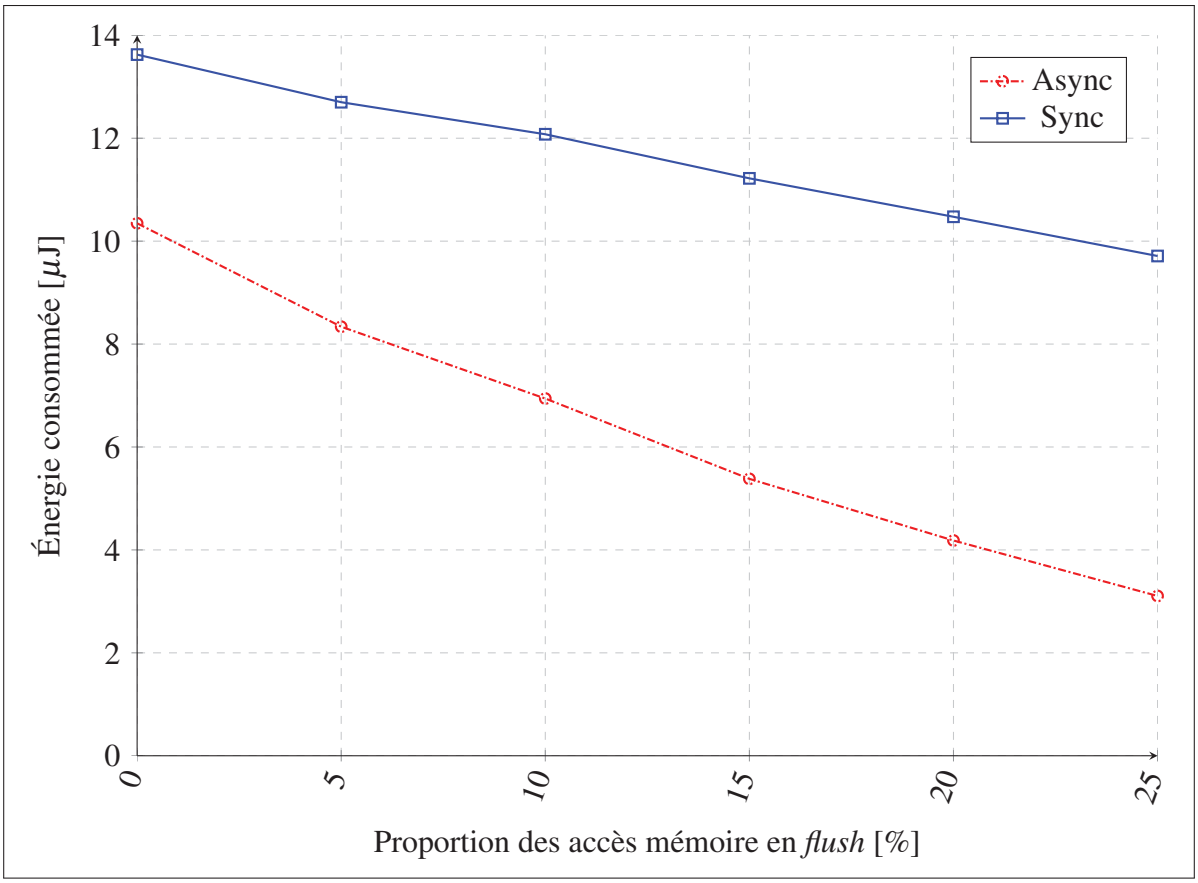


Figure 5.7 Comparaison de l'énergie consommée pour 1M d'accès mémoire en fonction de la proportion d'accès en *flush*

5.5.1 Analyse individuelle

Premièrement, le banc de test individuel a permis d'identifier certaines particularités concernant le placement de la cache L1 d'instruction asynchrone. Tout d'abord, il est important de mentionner est que la grande majorité de l'aire de la cache est occupée par les mémoires RAM des étiquettes et de données. En effet, ces mémoires RAM représentent 90% de l'aire totale de la cache d'instruction. Il est donc raisonnable d'affirmer que la majeure partie de l'énergie statique totale consommée par la cache est due à ce réseau de mémoires RAM. Cela étant dit, puisque les caches synchrone et asynchrone utilisent les mêmes mémoires RAM pour stocker les étiquettes et les données, une attention particulière a plutôt été portée au placement du pipe-

Tableau 5.6 Comparaison de l'énergie consommée en fonction de la proportion d'accès en *flush*

Nb instructions	Flush (%)	Énergie consommée (μJ)		ΔE (%)
		Synchrone	Asynchrone	
10000		0,150	0,118	21,0
50000		0,694	0,532	23,4
100000	0	1,375	1,048	23,8
500000		6,824	5,182	24,1
1000000		13,625	10,347	24,1
10000		0,139	0,092	33,8
50000		0,646	0,425	34,2
100000	5	1,279	0,840	34,3
500000		6,354	4,171	34,3
1000000		12,700	8,338	34,3
10000		0,131	0,077	41,0
50000		0,612	0,353	42,3
100000	10	1,215	0,698	42,5
500000		6,040	3,472	42,5
1000000		12,077	6,941	42,5
10000		0,123	0,063	48,9
50000		0,569	0,274	51,9
100000	15	1,130	0,542	52,0
500000		5,614	2,692	52,0
1000000		11,220	5,381	52,0
10000		0,114	0,050	55,8
50000		0,530	0,215	59,5
100000	20	1,055	0,425	59,7
500000		5,241	2,094	60,0
1000000		10,474	4,183	60,1
10000		0,106	0,040	61,7
50000		0,491	0,162	67,0
100000	25	0,978	0,318	67,5
500000		4,864	1,561	67,9
1000000		9,712	3,105	68,0

line. La comparaison au niveau du placement est donc seulement faite pour la taille du pipeline des caches synchrone et asynchrone.

Ensuite, on remarque que les FIFO d'entrée et de sortie représentent communément environ une fois et demie la taille du pipeline asynchrone. De plus, le FIFO de sortie a été placé relati-

vement loin du dernier étage du pipeline, ce qui implique que des répéteurs (portes logiques inverseurs) ont été utilisés pour transmettre les signaux sur les quelques 220000 μm qui séparent ces deux modules. Cela a néanmoins peu d'importance, car l'analyse comparative ne comptabilise pas la taille occupée ni l'énergie consommée par ces deux FIFO. En effet, ces FIFO sont seulement intégrés à la cache asynchrone pour s'interfacer facilement avec les modules synchrones extérieurs, soit le PCBP et IDecode. Dans l'éventualité où le processeur asynchrone intègrerait une mémoire d'instruction asynchrone, ces deux derniers modules seraient aussi modifiés pour opérer de façon asynchrone, afin d'éviter une resynchronisation de l'horloge.

Finalement, il faut mentionner que ce placement demeure strictement préliminaire. Comme il l'a été mentionné précédemment, le placement décrit ici ne peut être validé que suite au routage effectué par un locigiel de CAO. Les analyses de congestion et les délais de routage sont seulement estimés suite à un premier placement préliminaire complet. Il est donc important de rappeler cette limitation, puisque celle-ci affecte les résultats de performance obtenus par la suite.

Deuxièmement, le banc de test individuel a permis d'identifier certaines contraintes et particularités imposées par l'architecture du pipeline de la cache L1 d'instruction asynchrone. Tout d'abord, les données du tableau 5.2 indiquent quels étages du pipeline possèdent une latence élevée par rapport aux autres. Une différence importante entre ces latences introduit de la pression de réaction (« feedback pressure ») et peut créer un déséquilibre entre l'opération des étages. Lorsque la cache est mode « hit », les étages *Forward Address* et *Data Output* sont dépendants des étages *Tag Read* et *Data Read*. Ces derniers ont une latence deux fois plus grande que les étages les plus rapides. Dans le but de rééquilibrer le pipeline, il pourrait être intéressant de scinder les étages plus lents en deux, afin d'augmenter le débit de la cache en sortie.

Par la suite, l'analyse STA révèle que les étages *Tag Read* et *Data Read* possèdent la vitesse d'opération la plus lente. D'une part, cela est dû au fait qu'une lecture implique un aller-retour du pipeline au réseau de mémoires RAMs. Ces lignes de transmission peuvent induire jusqu'à

1200ps de délai pour l'étage *Data Read* et 800ps pour l'étage *Tag Read*. Quant aux mémoires RAM précompilées utilisées, elles imposent une latence approximative de 400ps en lecture, ce qui n'est pas négligeable. D'une autre part, une proportion considérable de la logique de l'étage *Tag Read* est dédiée à la comparaison en parallèle des étiquettes de 18 bits. Cette logique combinatoire impose un délai moyen de 650ps, ce qui s'ajoute au délai des lignes de transmission.

Enfin, il est important de mentionner que le débit moyen de la cache est strictement trop lent. Entre chaque écriture au FIFO de sortie, il s'écoule un peu plus de 2800 ps, ce qui est principalement dû au déséquilibre entre les latences des étages. En effet, la valeur du débit résulte de la latence de l'étage *Data Read* ajoutée à la latence engendrée par le contrôle du pipeline. Alors que le débit global obtenu pour la cache asynchrone proposée est présentement inadéquat pour le processeur d'Octasic, il serait toutefois possible de corriger la situation. Dans un premier temps, il serait possible d'atteindre une vitesse avoisinant 650MHz en *pipelinant* davantage les étages lents et ainsi balancer les latences inter-étages. Dans un deuxième temps, il serait possible d'atteindre des fréquences d'opération encore plus élevées en intégrant une architecture de type « super-pipeline ». Un « super-pipeline » implique que chaque étage soit pipeliné de deux à quatre fois à l'interne pour permettre une opération concurrente des étages du pipeline (Stallings, 2010). Cela permet effectivement d'augmenter le débit global d'opération, moyennant une consommation énergétique accrue non négligeable dû à l'intégration de deux à quatre fois plus de DFFs .

Troisièmement, le banc de test individuel révèle que d'un point de vue synchrone, les débits obtenus aux interfaces de la cache sont relativement élevés. En effet, la logique combinatoire est limitée au sein de ces modules, ces derniers étant principalement constitués des démultiplexeurs à l'entrée d'une tranche de DFFs, puis suivis de multiplexeurs en sortie. Les résultats obtenus indiquent qu'il serait possible d'utiliser la cache d'instruction L1 asynchrone proposée dans un contexte où la fréquence de l'horloge globale « L2clk » varie entre 1.15GHz et 1.4GHz.

Toutefois, les signaux sensibles tels que l'horloge et la remise à zéro doivent être distribués de façon équilibrée jusqu'à 73 et 137 DFFs respectivement au niveau des FIFOs. Ce réseau de distribution implique un temps de propagation non négligeable, équivalent à 30% de la valeur de T_{COMB}. Cela étant dit, le chemin critique est plutôt imposé par la comparaison des pointeurs de lecture et d'écriture du FIFO. Du point de vue de l'interface extérieure, la comparaison doit être fait de manière synchrone. Ainsi, tel qu'il est illustré aux figures 4.13 et 4.15, le pointeur asynchrone doit d'abord passer par un circuit éliminant la métastabilité avant de pouvoir être comparé. Cette logique, qui comprend 5 étages de portes logiques, détermine le chemin critique des modules de FIFO d'entrée et de sortie.

L'interface L1-L2 ne comprend pas de comparaison de pointeur puisque la cache L2 envoie la nouvelle ligne de cache et l'étiquette de façon « fire-and-forget », sans effectuer de validation. Voilà pourquoi le chemin critique à l'interface L1-L2 est plutôt déterminé par le réseau de distribution et par la logique d'activation (masque d'horloge). Plus spécifiquement, le chemin critique est déterminé par l'écriture d'un quart (64 bits) de la nouvelle ligne de cache dans le FIFO de l'interface L1-L2.

5.5.2 Analyse comparative

Premièrement, avant même d'utiliser le banc de test comparatif, il est possible de comparer la taille des caches afin d'estimer s'il y aura un gain en terme d'efficacité énergétique entre la cache asynchrone et synchrone. En effet, il existe une corrélation directe entre la surface occupée sur la puce et la quantité d'énergie statique consommée par le circuit. Dans le cas de la cache asynchrone proposée, on dénote que la taille du pipeline est 15% plus petite sans l'interface L1-L2 et 11.5% plus petite en incluant cette dernière. Il est aussi intéressant de mentionner que la surface occupée par les circuits de contrôle asynchrone, soit les modules des jetons, est trois plus grande que le contrôle synchrone. Cela est dû au fait que le contrôle asynchrone d'un étage de pipeline, qui implique la synchronisation et la génération d'une horloge locale, est plus complexe que le contrôle d'une horloge synchrone globale. En comparant

les tailles des deux caches, il serait raisonnable d'estimer que la cache asynchrone consomme moins d'énergie statique que son équivalent synchrone.

Dans un deuxième temps, le décompte du nombre d'éléments mémoire permet d'estimer les gains d'efficacité en terme d'énergie dynamique consommée. En effet, une bonne proportion de la puissance dynamique est résultante de l'activité de commutation aux entrées de ces cellules mémoires. Pour la cache asynchrone, on dénote 16% moins d'éléments mémoire par rapport au nombre présent dans la cache synchrone. Malgré qu'un décompte complet du nombre de portes logiques implémentées pour chaque pipeline n'a pas été fait, on peut déduire que ce nombre est similaire étant donné que la surface occupée par la cache asynchrone est moindre que la cache synchrone. Ainsi, étant donné que les deux caches d'instruction utilisent la même configuration de mémoires RAM, on peut logiquement prédire que la cache d'instruction asynchrone consommera moins d'énergie dynamique que son équivalent synchrone.

Deuxièmement, le banc de test comparatif permet de déterminer s'il y a un gain en performance en terme de temps d'accès moyen à la mémoire entre la cache asynchrone et synchrone. Tout d'abord, les données représentées à la figure 5.3 démontrent que les temps moyens d'accès à la mémoire des caches d'instruction suivent une courbe décroissante asymptotique similaire. Tel que projeté initialement, plus le nombre d'accès à la mémoire augmente, plus le temps d'accès moyen est réduit. Cela est principalement dû au fait que les accès en « cache hit » sont rapides et que cette proportion augmente plus le nombre d'accès mémoire est élevé. Le graphique de la figure 5.3 illustre aussi que peut importe la proportion d'accès en purge, la cache asynchrone présente une réduction (ΔT) du temps d'accès d'environ 1 ns. Un autre fait intéressant est que la cache asynchrone garde un temps d'accès mémoire minimal lorsque la proportion d'accès en purge est de 20%. D'ailleurs, le temps moyen d'accès reste relativement constant, et donc à un minimum, peu importe le nombre d'accès à la mémoire.

Ensuite, le graphique de la figure 5.4 permet d'observer le comportement des deux types de cache d'instruction pour deux points d'opération. Ce graphique permet donc d'examiner l'impact de la proportion des accès à la mémoire en purge, résultant d'une mauvaise prédiction

du PCBP, pour un mode d'opération majoritairement en « cache miss » ou en « cache hit ». Ainsi, on dénote que le temps d'accès de la cache synchrone ne diminue pas en fonction de la proportion des accès en *flush*. Pour la cache asynchrone, le temps moyen d'accès varie en fonction de cette proportion lorsque le mode d'opération est principalement en « cache miss », mais peu ou pas lorsque le mode est majoritairement en « cache hit ». Ces résultats démontrent donc l'avantage que possède la cache asynchrone lorsque les proportions d'accès en *flush* sont élevées.

Finalement, les données colligées au tableau 5.5 révèlent que la cache d'instruction asynchrone possède un temps d'accès moyen à la mémoire au moins 25% plus rapide que celui de la cache synchrone. Cette différence est encore plus notable lorsque la cache opère principalement en mode transitoire, soit en « cache miss ». En effet, l'écart varie de plus de 30% et s'accentue jusqu'à pratiquement 40% plus la proportion d'accès en *flush* augmente. Globalement, l'utilisation de la cache asynchrone proposée est avantageuse en terme de réduction du temps d'accès.

Troisièmement, le banc de test comparatif permet de déterminer s'il existe un gain en terme d'efficacité énergétique entre la cache asynchrone et synchrone. Tout d'abord, les données de la figure 5.5 illustrent que la consommation d'énergie dynamique des caches d'instruction suit une courbe quadratique similaire. Alors qu'il est difficile d'évaluer sur ce graphique le gain d'efficacité énergétique pour un nombre réduit d'accès mémoire à la cache asynchrone, les données suggèrent que la réduction d'énergie s'accentue lorsque le nombre d'accès mémoire augmente. Le graphique de la figure 5.3 démontre aussi que plus la proportion des accès à la mémoire en *flush* est grande, plus la consommation d'énergie des caches d'instruction baisse. Puisque les accès en purge n'accèdent pas aux mémoires RAM des étiquettes ou de données, les caches consomment donc moins d'énergie. Cela étant dit, ce phénomène est davantage prononcé pour la cache asynchrone, ce qui indique que la gestion des purges par les jetons est plus efficace d'un point de vue énergétique. En effet, un *flush* ne réinitialise que 8 DFFs au sein de la cache asynchrone lors d'une purge, contrairement à tous les étages du pipeline pour la cache synchrone.

Ensuite, les figures 5.6 et 5.7 permettent d'observer la réduction de la consommation énergétique des deux types de caches d'instruction pour deux points d'opération. Il est donc possible d'examiner l'impact de la proportion des accès à la mémoire en purge pour un mode d'opération majoritairement en « cache miss » ou en « cache hit ». Ces graphiques démontrent que la consommation d'énergie diminue en fonction d'une augmentation de la proportion des accès en *flush*. Toutefois, cette réduction énergétique est accentuée pour la cache asynchrone, puisqu'on dénote que la courbe diminue plus abruptement. Ces résultats démontrent une fois de plus l'avantage que possède la cache asynchrone lorsque les proportions d'accès en *flush* sont élevées.

Finalement, les données colligées au tableau 5.6 révèlent que la cache d'instruction asynchrone permet de réduire la consommation énergétique d'au moins 20% par rapport à la cache synchrone. On remarque que cette réduction énergétique augmente de 3 à 8% lorsque la cache asynchrone opère en régime permanent, soit majoritairement en « cache hit ». Cependant, cet écart se creuse plus la proportion d'accès en *flush* augmente, réduisant d'environ 20% à plus de 60% la consommation d'énergie. Globalement, l'utilisation de la cache asynchrone proposée est très avantageuse en terme d'efficacité énergétique.

5.6 Conclusion

Dans ce chapitre, une analyse approfondie de la nouvelle cache d'instruction L1 asynchrone a été faite. Le flot de conception d'Octasic utilisé dans le cadre de cette maîtrise a d'abord été défini, ainsi que l'environnement de simulation. Les différents bancs de test, indépendant ou comparatif, ont été décrits et les limitations inhérentes aux résultats obtenus ont été clairement exposées. L'analyse des résultats obtenus suite au banc de test indépendant a permis de valider les spécifications ainsi que le fonctionnement et les limitations internes de la cache asynchrone, suite aux multiples itérations de STA. Puis, l'analyse des résultats suite au banc de test comparatif avec la cache synchrone a permis de mettre en contexte les performances globales de la cache asynchrone en terme de temps d'accès mémoire moyen et d'efficacité énergétique.

Comme il a été mentionné au début du chapitre, afin de valider et comparer adéquatement les performances de la nouvelle architecture de cache, il aurait été nécessaire de l'intégrer dans un processeur complet, ou encore mieux, de faire fabriquer le processeur en question. Néanmoins, ce type d'analyse expérimentale est hors de portée dans le cadre de ce projet. Cela étant dit, plusieurs éléments d'analyse pertinents ont été soulevés suite à la simulation des bancs de test indépendant et comparatif.

Tout d'abord, on retient que seul le placement du pipeline diffère entre les deux caches d'instruction. L'intégration et le placement des mémoires RAM demeurent identiques et l'impact de l'intégration des FIFOs n'a pas été comptabilisé. La comparaison du placement préliminaire indique que le pipeline de la cache asynchrone occupe environ 15% moins d'espace sur la puce et intègre 16% moins de DFFs que son équivalent synchrone.

Ensuite, la vitesse d'opération n'est balancée entre les étages du pipeline asynchrone, ce qui affecte et réduit considérablement le débit à la sortie de la cache. Présentement, la cache asynchrone peut distribuer deux instructions à une fréquence d'environ 350MHz. En rééquilibrant et en *pipelinant* davantage les étages les plus lents, il serait possible d'augmenter facilement la fréquence à plus de 650MHz. En intégrant le « superpipeline », il serait possible d'augmenter cette fréquence à environ 1GHz. Finalement, la vitesse d'opération des interfaces développées est relativement élevée d'un point de vue synchrone. Les FIFO d'entrée et de sortie ainsi que l'interface L1-L2 peuvent transférer des données à une fréquence variant entre 1.15GHz et 1.4GHz.

Finalement, le banc de test comparatif a permis de quantifier le gain en performance en terme de temps d'accès moyen à la mémoire et d'efficacité énergétique. Les résultats obtenus révèlent que le temps d'accès moyen à la mémoire est réduit d'un minimum de 25% et ce, peu importe la proportion des accès en purge ou son mode d'opération (majoritairement en « cache miss » ou en « cache hit »). Les résultats portant sur l'efficacité énergétique suggèrent que l'utilisation de la cache asynchrone peut réduire la consommation énergétique d'au moins 21%. De plus, ce taux s'accentue jusqu'à lorsque la cache asynchrone opère en régime permanent et encore plus

lorsque la proportion des accès en *flush* augmente. L'intégration du nouveau module de gestion des purges du pipeline asynchrone améliore globalement les performances de la nouvelle cache asynchrone proposée.

En résumé, plusieurs simulations ont permis de soutenir l'hypothèse qu'une l'intégration d'une cache L1 asynchrone peut améliorer les performances du processeur ARM. En effet, remplacer la cache synchrone actuelle par son équivalent asynchrone, suite au développement d'une nouvelle architecture de pipeline asynchrone, permettrait de réduire la surface de la puce utilisée par la cache et la latence d'accès à la cache (réduire le temps moyen d'accès à la mémoire) et d'améliorer son efficacité énergétique. Par rapport à la cache synchrone, les simulations faites lors de ce projet démontrent que la cache asynchrone proposée bénéficie d'une surface de pipeline réduite de 15%. De plus, les résultats suggèrent que la cache asynchrone permet d'obtenir une réduction moyenne du temps d'accès à la mémoire d'au moins 25% et d'une augmentation de l'efficacité énergétique d'au moins 21% par rapport à la cache synchrone précédente.

CONCLUSION

Les caches sont des blocs de mémoire rapides situés près du processeur. Ces mémoires caches sont aujourd'hui implémentées dans pratiquement tous les processeurs modernes performants et permettent d'accélérer l'accès aux instructions et aux données nécessaires à l'exécution d'un programme. L'optimisation d'une mémoire cache dépend de plusieurs facteurs, tels que son organisation logique et ses mécanismes de gestion interne. Cependant, l'architecture d'une cache dépend aussi de la structure de son pipeline est implémenté.

L'architecture d'un pipeline dépend intrinsèquement de la façon dont les étages sont synchronisés entre eux. Le principe d'un pipeline réside dans la décomposition d'une tâche complexe en plusieurs tâches simples séquentielles, pour réorganiser sa structure interne en plusieurs étages d'un pipeline. Cette méthode permet d'augmenter le débit d'un processus séquentiel en exploitant une forme structurelle de parallélisme. L'architecture d'un pipeline dépend intrinsèquement de la façon dont les étages du pipeline sont synchronisés entre eux. Alors que les pipelines synchrones sont généralement la norme, il existe d'autres types de pipelines dits asynchrones possédant de nombreux avantages.

Dans le cadre de ce projet, le processeur ARM conçu par Octasic sépare le coeur du processeur asynchrone de la mémoire cache synchrone. Il existe donc une barrière de synchronisation entre le CPU et la mémoire cache L1. Une des principales problématiques de ce travail réside en l'accès à la mémoire cache L1, qui est présentement un des goulots d'étranglement du processeur. D'une part, le but est d'éliminer la pénalité de synchronisation de 2 cycles inhérente entre le processeur et la mémoire. D'une autre part, le but est de réduire le temps d'accès moyen à la mémoire, réduire la consommation d'énergie et la surface occupée par la cache sur la puce. En intégrant une cache asynchrone basée sur un pipeline auto-cadencé, il est révélé possible d'atteindre ces objectifs.

La conception d'une mémoire cache d'instruction L1 asynchrone a été réalisée dans le cadre de ce projet de maîtrise. La structure du pipeline auto-cadencé asynchrone présenté dans la première section du chapitre 4 est dérivée des éléments Click et des modules de jetons développés

par Octasic, provenant respectivement de la revue de littérature au chapitre 2 et méthodologie asynchrone d'Octasic énoncée au chapitre 3. L'architecture et le fonctionnement de la cache d'instruction sont présentés au chapitre 4 et sont basés sur la revue de littérature faite à ce sujet au chapitre 1. Les balises de conception sont dérivées des spécifications et des contraintes imposées par le processeur ARM asynchrone et par la cache d'instruction synchrone actuelle, énumérées au chapitre 3. Des interfaces entre la cache d'instruction et les modules environnants ont été développées afin de permettre son intégration complète dans le processeur développé par Octasic.

La cache asynchrone proposée a été testée en simulation post-placement et les résultats préliminaires de cette analyse sont présentés au chapitre 5. Des bancs de test ont été développés afin de valider la fonctionnalité et d'analyser la performance de la cache asynchrone proposée. Les résultats obtenus suite aux bancs de tests comparatifs confirment que la cache d'instruction asynchrone proposée permet de réduire le temps moyen d'accès à la mémoire et la consommation d'énergie. Cependant, une analyse STA complète post-routage aurait été nécessaire afin de valider formellement la conception. De plus, la réalisation physique de cette cache asynchrone dans le processeur ARM d'Octasic n'a pas été possible dans le cadre de ce projet de maîtrise. Néanmoins, les résultats suggèrent que l'implémentation d'une cache asynchrone serait avantageuse dans le contexte d'un processeur asynchrone.

Limitations

Les limitations de la cache asynchrone conçue ont été identifiées suite aux simulations des bancs de test du chapitre 5 et de l'analyse des résultats obtenus. Lors d'une prochaine itération, certaines de ces limitations pourraient être adressées et corrigées.

- la cache d'instruction développée dans le cadre de ce projet ne comprend pas de module MPU ou MMU permettant de traduire l'adressage virtuel d'un système d'exploitation vers l'adresse physique de la mémoire. Ce type de module est inhérent à la réalisation d'un processeur ARM implémentant les spécifications v7-A ;

- une analyse STA post-routage de la cache asynchrone n'a pas été effectuée. Idéalement, un routage physique devrait être effectué par un logiciel de CAO afin de valider le placement et ensuite simuler le circuit avec les délais obtenus suite au routage ;
- le circuit n'a pas été fabriqué et testé dans un processeur ARM réel. Réaliser un circuit physique permettrait de quantifier plus facilement les améliorations liées à l'intégration de la cache asynchrone proposée.

Contributions scientifiques

Les contributions scientifiques de ce travail sont :

- une revue de littérature portant sur les mémoires caches et sur les pipelines asynchrones présentée dans le cadre du cours lectures dirigées (MTR-871) ;
- la proposition et développement d'un pipeline asynchrone comportant une synchronisation hybride par les éléments Click et le système de jetons ;
- laproposition d'une cache asynchrone à faible latence et à efficacité d'énergie élevée intégrant le nouveau pipeline asynchrone développé. La cache asynchrone conçue dans le cadre de ce travail a fait l'objet d'une publication (Trudeau *et al.*, 2015) et a été présentée à la conférence ASYNC 2015 à Mountain View en Californie le 5 mai 2015 ;
- la cache asynchrone développée pour le processeur ARM asynchrone permet potentiellement de diminuer la latence moyenne d'accès à la mémoire d'un minimum de 21%, par rapport à l'utilisation de la cache synchrone. Dans un deuxième temps, l'analyse réalisée révèle que la nouvelle cache asynchrone développée consomme de 25 à 68% moins d'énergie dynamique que son équivalent synchrone, pour un mode d'opération équivalent.

Travaux futurs

En conclusion, différents axes de recherche pourraient être empruntés afin d'améliorer la cache asynchrone développée lors de ce projet :

- optimiser davantage la cache asynchrone réalisée, potentiellement suite à une revue avec l'équipe de conception d'Octasic.
- simuler et évaluer la performance des caches au sein d'un processeur ARM asynchrone complet afin de simuler des programmes de référence « benchmarks ».
- un des travaux importants à réaliser est dans un premier temps d'effectuer un routage physique de la cache afin de simuler et valider la cache asynchrone conçue. Dans un deuxième temps, il serait intéressant de fabriquer un processeur ARM qui intégrerait cette nouvelle cache asynchrone afin d'évaluer réellement ses performances ;
- modifier le pipeline asynchrone afin d'équilibrer les étages et implémenter un accès non bloquant à la cache. Ces nouvelles fonctionnalités permettraient d'améliorer considérablement le débit de la cache proposée ;
- intégrer des accès multiples aux ressources gérées par les jetons grâce à la synchronisation des éléments Click. Cela permettrait d'implémenter une forme de *super-pipeline* et d'augmenter considérablement le débit de la cache. Cependant, un équilibre devrait être déterminé entre débit et consommation énergétique ;
- développer un pipeline asynchrone seulement avec les éléments Click, sans utiliser les modules de gestion des jetons pour générer les signaux d'horloge. Il serait peut-être possible d'augmenter la fréquence d'opération et l'efficacité énergétique en se basant uniquement sur le protocole de communication des éléments Click ;
- intégrer la nouvelle architecture asynchrone pour la mémoire cache de données. En évitant la pénalité de synchronisation, beaucoup plus critique lors des accès à la mémoire de données, il serait possible d'améliorer globalement les performances du processeur.

BIBLIOGRAPHIE

Agarwal, Anant, John Hennessy, et Mark Horowitz. 1988. « Cache performance of operating system and multiprogramming workloads ». *ACM Transactions on Computer Systems (TOCS)*, vol. 6, n° 4, p. 393–431.

ARM. 2010a. « ARM Architecture Reference Manual - ARM v7-A and ARM v7-R edition Errata markup 8.0 », vol. 3. p. 2004–2010.

ARM. 2010b. « A8 Technical Reference Manual ». *Rev. r3p2*.

Beerel, Peter A, Recep O Ozdag, et Marcos Ferretti, 2010. *A Designer' s Guide to Asynchronous VLSI*. Cambridge, United-Kingdom : Cambridge University Press, 359 p.

Bink, Arjan et Richard York. 2007. « ARM996HS : The First Licensable Clockless 32-bit Processor Core ». p. 58–68.

Calder, Brad, Dirk Grunwald, et Joel Emer. 1996. « Predictive sequential associative cache ». In *High-Performance Computer Architecture, 1996. Proceedings., Second International Symposium on*. p. 244–253. IEEE.

Chung, Sung Woo et Kevin Skadron. 2008. « On-demand solution to minimize I-cache leakage energy with maintaining performance ». *Computers, IEEE Transactions on*, vol. 57, n° 1, p. 7–24.

Flautner, Krisztián, Nam Sung Kim, Steve Martin, David Blaauw, et Trevor Mudge. 2002. « Drowsy caches : simple techniques for reducing leakage power ». In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. p. 148–157. IEEE.

Furber, Stephen B, James D Garside, Peter Riocreux, Steven Temple, Paul Day, Jianwei Liu, et Nigel C Paver. 1999. « AMULET2e : An asynchronous embedded controller ». *Proceedings of the IEEE*, vol. 87, n° 2, p. 243–256.

Giorgi, Roberto et Paolo Bennati. 2007. « Reducing leakage in power-saving capable caches for embedded systems by using a filter cache ». In *Proceedings of the 2007 workshop on Memory performance : Dealing with Applications, systems and architecture*. p. 97–104. ACM.

Hajimiri, Hadi, Kamran Rahmani, et Prabhat Mishra. 2011. « Synergistic integration of dynamic cache reconfiguration and code compression in embedded systems ». In *Green Computing Conference and Workshops (IGCC), 2011 International*. p. 1–8. IEEE.

Hennessy, John L. et David A. Patterson, 2011. *Computer Architecture, Fifth Edition : A Quantitative Approach*. éd. 5th. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc.

Hwu, W-m W et Pohua P Chang. 1989. « Achieving high instruction cache performance with an optimizing compiler ». In *ACM SIGARCH Computer Architecture News*. p. 242–251. ACM.

Inoue, Koji, Vasily G. Moshnyaga, et Kazuaki Murakami. 2002. « Trends in high-performance, low-power cache memory architectures ». *IEICE Transactions on Electronics*, vol. 85, n° 2, p. 304–314.

Jacob, Bruce, S Ng, et D Wang, 2010. *Memory systems : cache, DRAM, disk*.

John, Lizy Kurian et Akila Subramanian. 1997. « Design and performance evaluation of a cache assist to implement selective caching ». In *Computer Design : VLSI in Computers and Processors, 1997. ICCD'97. Proceedings., 1997 IEEE International Conference on*. p. 510–518. IEEE.

Johnson, Teresa L, Matthew C Merten, et Wen-Mei W Hwu. 1997. « Run-time spatial locality detection and optimization ». In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. p. 57–64. IEEE Computer Society.

Jouppi, Norman P. 1990. « Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers ». In *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*. p. 364–373. IEEE.

Jouppi, Norman P. 1993. « Cache write policies and performance », vol. 21, n° 2.

Kadayif, Ismail, Ayhan Zorlubas, Selcuk Koyuncu, Olcay Kabal, Davut Akcicek, Yucel Sahin, et Mahmut Kandemir. 2008. « Capturing and optimizing the interactions between prefetching and cache line turnoff ». *Microprocessors and Microsystems*, vol. 32, n° 7, p. 394–404.

Kessler, Richard E, Richard Jooss, Alvin Lebeck, et Mark D Hill. 1989. « Inexpensive implementations of set-associativity », vol. 17, n° 3.

Kin, Johnson, Munish Gupta, et William H Mangione-Smith. 1997. « The filter cache : an energy efficient memory structure ». In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. p. 184–193. IEEE Computer Society.

Kondratyev, Alex et Kelvin Lwin. 2002. « Design of asynchronous circuits by synchronous CAD tools ». In *Design Automation Conference, 2002. Proceedings. 39th*. p. 411–414. IEEE.

Ku, Ja Chun, Serkan Ozdemir, Gokhan Memik, et Yehea Ismail. 2005. « Thermal management of on-chip caches through power density minimization ». In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. p. 283–293. IEEE Computer Society.

Kumar, Sanjeev et Christopher Wilkerson. 1998. « Exploiting spatial locality in data caches using spatial footprints ». In *ACM SIGARCH Computer Architecture News*. p. 357–368. IEEE Computer Society.

Kwak, Jong Wook et Young Tae Jeon. Septembre 2010. « Compressed tag architecture for low-power embedded cache systems ». *Journal of Systems Architecture*, vol. 56, n° 9, p. 419–428.

Laurence, Michel. Mai 2012. « Introduction to Octasic Asynchronous Processor Technology ». *2012 IEEE 18th International Symposium on Asynchronous Circuits and Systems*, p. 113–117.

Laurence, Michel. 2013. « Low-Power High-Performance Asynchronous General Purpose ARMv7 Processor For Multi-Core Applications ». *13th International Forum on Embedded MPSoC and Multicore, Otsu, Japan*, p. 304–314.

Ligthart, Michiel, Karl Fant, Ross Smith, Alexander Taubin, et Alex Kondratyev. 2000. « Asynchronous design using commercial HDL synthesis tools ». In *Advanced Research in Asynchronous Circuits and Systems, 2000.(ASYNC 2000) Proceedings. Sixth International Symposium on*. p. 114–125. IEEE.

Liu, Lishing. 1994. « Cache designs with partial address matching ». In *Proceedings of the 27th annual international symposium on Microarchitecture*. p. 128–136. ACM.

Meng, Ke, Russ Joseph, Robert P Dick, et Li Shang. 2008. « Multi-optimization power management for chip multiprocessors ». In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. p. 177–186. ACM.

Microelectronics, ST. « A8 Technical Reference Manual ». *User Manual*.

Mittal, Sparsh. Novembre 2013. « A survey of architectural techniques for improving cache power efficiency ». *Sustainable Computing : Informatics and Systems*, p. 1–11.

Nowick, Steven M. et Montek Singh. Septembre 2011. « High-Performance Asynchronous Pipelines : An Overview ». *IEEE Design & Test of Computers*, vol. 28, n° 5, p. 8–22.

Panda, Preeti Ranjan, Nikil D Dutt, et Alexandru Nicolau. 1997. « Efficient utilization of scratch-pad memory in embedded processor applications ». In *Proceedings of the 1997 European conference on Design and Test*. p. 7. IEEE Computer Society.

Park, Hyunsun, Sungjoo Yoo, et Sunggu Lee. 2012. « A multistep tag comparison method for a low-power L2 cache ». *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 31, n° 4, p. 559–572.

Patterson, David A et John L Hennessy, 2009. *Computer Organization and Design : the Hardware/Software Interface*. éd. 4th. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc.

Peeters, Ad, Frank Te Beest, Mark De Wit, et Willem Mallon. 2010. « Click Elements : An Implementation Style for Data-Driven Compilation ». *2010 IEEE Symposium on Asynchronous Circuits and Systems*, p. 3–14.

Peir, Jih-Kwon, Yongjoon Lee, et Windsor W Hsu. 1998. « Capturing dynamic memory reference behavior with adaptive cache topology ». In *ACM SIGPLAN Notices*. p. 240–250. ACM.

Pillai, Padmanabhan et Kang G Shin. 2001. « Real-time dynamic voltage scaling for low-power embedded operating systems ». In *ACM SIGOPS Operating Systems Review*. p. 89–102. ACM.

Powell, Michael, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, et TN Vijaykumar. 2000. « Gated-V dd : a circuit technique to reduce leakage in deep-submicron cache memories ». In *Proceedings of the 2000 international symposium on Low power electronics and design*. p. 90–95. ACM.

Powell, Michael D, Amit Agarwal, TN Vijaykumar, Babak Falsafi, et Kaushik Roy. 2001. « Reducing set-associative cache energy via way-prediction and selective direct-mapping ». In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. p. 54–65. IEEE Computer Society.

Shafiee, Ali, Narges Shahidi, et Amirali Baniasadi. 2012. « Using partial tag comparison in low-power snoop-based chip multiprocessors ». In *Computer Architecture*. p. 211–221. Springer.

Singh, M. et S.M. Nowick. Juin 2007. « MOUSETRAP : High-Speed Transition-Signaling Asynchronous Pipelines ». *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, n° 6, p. 684–698.

Singh, Montek. 2002. « The design of high-throughput asynchronous pipelines ». PhD thesis, Columbia University, 265 p.

Solutions, Handshake. « Asynchronous circuit technology is on the market Overview ».

Stallings, William, 2010. *Computer Organization and Architecture : Designing For Performance*. éd. 8th. Upper Saddle River, NJ, USA : Prentice Hall.

Sundararajan, Karthik T., Timothy M. Jones, et Nigel Topham. Juillet 2011. « Smart cache : A self adaptive cache architecture for energy efficiency ». *2011 International Conference on Embedded Computer Systems : Architectures, Modeling and Simulation*, p. 41–50.

Sutherland, I. E. Juin 1989. « Micropipelines ». *Communications of the ACM*, vol. 32, n° 6, p. 720–738.

Te Beest, Frank, Ad Peeters, Kees Van Berkel, et Hans Kerkhoff. 2003. « Synchronous full-scan for asynchronous handshake circuits ». *Journal of Electronic Testing*, vol. 19, n° 4, p. 397–406.

Traylor, R.L. Janvier 31 1995. « Self-timed data pipeline apparatus using asynchronous stages having toggle flip-flops ». <https://www.google.com/patents/US5386585>. US Patent 5,386,585.

Trudeau, LC, G Gagnon, F Gagnon, C Thibeault, T Awad, et D Morrissey. 2015. « A Low-Latency, Energy-Efficient L1 Cache Based on a Self-Timed Pipeline ». In *Asynchronous Circuits and Systems (ASYNC), 2015 21st IEEE International Symposium on*. p. 17–18. IEEE.

Weste, Neil et David Harris, 2010. *CMOS VLSI design : a circuits and systems perspective*.

Williams, Ted. 1990. « Latency and Throughput Tradeoffs in Self-Timed Speed-Independent Pipelines and Rings Independent Pipelines and Rings », .

Wulf, WA et SA McKee. 1995. « Hitting the memory wall : implications of the obvious ». *ACM SIGARCH computer architecture news*, p. 20–24.

Zhang, Chuanjun, Frank Vahid, Jun Yang, et Walid Najjar. 2004. « A way-halting cache for low-energy high-performance systems ». *Proceedings of the 2004 international symposium on Low power electronics and design - ISLPED '04*, p. 126.

Zhu, Zhichun et Xiaodong Zhang. 2002. « Access-mode predictions for low-power cache design ». *IEEE micro*, vol. 22, n° 2, p. 58–71.