Université Catholique de Louvain

Mémoire

Conception d'une microarchitecture SMT avec ordonnancement hardware pour applications temps réel

Auteur:
Martin Donies

Promoteur:
Dr. Jean-Didier Legat

Mémoire remis pour l'obtention
du Master en Science de l'ingénieur en électricité

.

Electricité et télécomunication

Août 2016

Université Catholique de Louvain

Abstract

Ecole Polytechnique de Louvain
Electricité et télécomunication

Master en Science de l'ingénieur en électricité

Conception d'une microarchitecture SMT avec ordonnancement hardware pour applications temps réel

par Martin Donies

Les techniques employées pour améliorer le débit d'instructions des microprocesseurs peuvent avoir un impact sur la prédictibilité des exécutions et l'exploitation du parallélisme peut introduire un surcoût lié à leur gestion. Ce mémoire propose une approche de conception d'une microarchitecture processeur visant à résoudre ces deux problématiques conjointement. Il sera montré expérimentalement qu'il est possible d'obtenir un temps d'exécution et un temps de réponse déterministe sans impacter le taux d'utilisation du processeur. Il sera aussi montré que cette approche permet d'éliminer le coût en performance des changements de contexte et de l'ordonnancement multi-tâches.

Remerciements

Je tiens à remercier mon promoteur, Jean-Didier Legat, pour ses conseils avisés et pour l'autonomie qu'il m'a laissée concernant mon choix de recherche. Je tiens aussi à remercier Juliette Stephany pour avoir accepté de subir un cours accéléré sur les microarchitectures processeur et pour sa relecture ainsi que Florentin Rochet pour ses conseils et sa relecture.

Table des matières

Liste des figures

Liste des tables

Abréviations

ALU	**A**rithmetic **L**ogic **U**nit
ASIC	**A**pplication-**S**pecific **I**ntegrated **C**ircuit
CPU	**C**entral **P**rocessing **U**nit
FPGA	**F**ield-Programmable **G**ate **A**rray
I/O	**I**nput / **O**utput
ID	**I**nstruction **D**ecode
IF	**I**nstruction **F**etch
ILP	**I**nstruction **L**evel **P**arallelism
IPC	**I**nstructions **P**er **C**ycle
MIPS	**M**ega **I**nstructions **P**er **S**econd
MIPS	**M**icroprocessor without **I**nterlocked **P**ipe **S**tages
NoC	**N**etwork **o**n **C**hip
OS	**O**perating **S**ystem
PC	**P**rogramm **C**ounter
SoC	**S**ystem **o**n **C**hip
TLP	**T**hread **L**evel **P**arallelism

Introduction

La démarche à l'origine de ce projet était d'étudier la faisabilité d'intégrer un plus grand niveau d'abstractions aux architectures processeurs tant au niveau du langage machine qu'à celui de la gestion matériel. Dans cette optique, il semblait intéressant d'étudier les noyaux et micro-noyaux ainsi que les architectures et microarchitectures processeur. Le sujet étant vaste, l'étude s'est recentrée sur l'un des aspects les plus critiques en terme de performance : l'ordonnancement de processus et d'instructions, principalement dans le cadre du multithreading et du multiprocessing. Du fait que les applications les plus sensibles aux performances d'ordonnancement sont celles ayant des contraintes temps réel, ce projet s'est naturellement dirigé vers ce type d'application. Les études portant sur l'implémentation de co-design hardware/software de systèmes d'exploitations temps réel sont nombreuses mais ne s'intéressent que rarement au problème d'un point de vue microarchitectural, et c'est précisément ce dont il va être question dans ce projet.

Ce travail se base principalement sur deux ouvrages :

- Operating System - Internals and design principles écrit par William Stallings [2].
- Processor Architecture - From Dataflow to Superscalar and Beyond écrit par Jurij Silc, Borut Robic et Theo Ungerer [3].

Ainsi que sur les travaux de Gaitan Vasile Gheorghita, Gaitan Nicoleta Cristinaet Ungurean Ioan concernant leur microarchitecture MPRA (Multiple Pipeline Register Architecture). [4, 5].

La microarchitecture MPRA offre des performances prometteuses en terme de temps de réponse, de changement de contexte et d'ordonnancement pour les applications temps réel. Ce travail tente de trouver une microarchitecture plus adaptée au multithreading et au multiprocessing du point de vue de la puissance de calcul. De plus, cette recherche tente de respecter la contrainte de temps exacte d'exécution.

Pour y parvenir, ce travail passera en revue la littérature au sujet de l'ordonnancement en général, multithread, et hardware, ainsi que les possibilités d'implémentation au

chapitre 1. Ensuite, une exploration des architectures et microarchitectures processeur sera proposée chapitre 2. Et pour conclure (chapitre 3), une description détaillée de l'implémentation d'un microprocesseur avec ordonnancement et sa validation sur FPGA (Field-Programmable Gate Array) seront présentés, et cette implémentation sera comparée et critiquée.

Chapitre 1

L'ordonnancement de processus

L'ordonnanceur, dans le domaine de l'informatique, est la partie d'un système d'exploitation qui s'occupe de déterminer quels processus et quels threads doivent être exécutés à quel moment. En particulier, dans les systèmes temps réel, il permet d'organiser l'exécution de processus afin de satisfaire des contraintes de temps. Ce chapitre passe en revue l'essentiel du fonctionnement d'un ordonnanceur, les différents choix techniques possibles et leur impact sur les performances d'un système. Il ne traitera pas des différents algorithmes définissant la politique d'ordonnancement, mais uniquement de la manière de classifier les axiomes de base les caractérisant afin d'identifier un socle commun implémentable de façon hardware. Cette section constitue la base théorique sur laquelle s'appuie le développement de l'ordonnanceur hardware développé chapitre 3.

1.1 Objectifs

Selon Linus Torvalds, fondateur du projet Linux, la problématique de la conception d'ordonnanceurs n'est pas complexe tant leur conceptualisation est simple :

"[...] Let's face it - the current scheduler has the same old basic structure that it did almost 10 years ago, and yes, it's not optimal, but there really aren't that many real-world loads where people really care. I'm sorry, but it's true.

And you have to realize that there are not very many things that have aged as well as the scheduler. Which is just another proof that scheduling is easy.

[...]

In comparison to those kinds of issues, I suspect that making the scheduler use per-CPU queues together with some inter-CPU load balancing logic is probably _trivial_. Patches

already exist, and I don't feel that people can screw up the few hundred lines too badly." [6, Linus Torvalds]

Ce dernier reconnaît que l'optimal n'est pas atteint mais trouve qu'il y a de toutes façons peu d'intérêt à y arriver car le problème de répartition des tâches parmi plusieurs processeurs est trivial. Cette vision de l'ordonnanceur Linux est critiquée par l'étude de Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, et Alexandra Fedorova [7]. Ils concluent que la problématique de répartition des cycles CPU entre les threads, bien que considérée comme résolue, ne l'est pas et que la complexité du matériel informatique récent rend la conception d'un ordonnanceur sujette aux bugs.

De plus, Kaustubh R. Joshi [8] et Robert Love [9, p. 67] estiment que la réalisation d'un ordonnanceur performant pour tout type de systèmes est un problème complexe. Cela s'explique par la diversité (à priori infinie) des systèmes possibles et par le fait que, par définition, leur catégorisation est empirique et non exhaustive. L'approche traditionnelle est de spécialiser le système d'exploitation pour une application particulière ou de hiérarchiser les politiques d'ordonnancement et de les faire coexister.

Une autre piste de solution est apportée par le concept de micro-noyaux [10]. L'idée est d'implémenter un ordonnanceur de la façon la plus simple possible tout en laissant la possibilité d'implémentation de fonctionnements plus complexes dans la couche utilisateur.

C'est cette dernière approche qui sera utilisée dans ce mémoire. L'objectif de ce travail est d'envisager une migration d'une partie des ordonnanceurs de la couche noyau vers la couche hardware. La démarche à l'origine du développement des micro-noyaux est semblable. Ce travail ne traite que des ordonnanceurs, mais la même logique peut être appliquée à la totalité des composantes d'un micro-noyau. Sachant que la principale critique émise contre les micro-noyaux se porte sur leur performance concernant la gestion des communications inter-processus [11], leur implémentation hardware pourrait résoudre ce problème. Il est intéressant de noter qu'Intel [12] envisage d'intégrer un micro-noyau à ses processeurs principalement pour l'ordonnancement multiprocesseur. Le travail de Itai Avron et Ran Ginosar [13] conclut aussi qu'un ordonnancement hardware présente un intérêt pour répondre à l'évolution du nombre de processeurs.

1.2 Fonctionnement général

Le concept d'ordonnanceur constitue la base des systèmes d'exploitations multiprogrammation [14, p.261]. Contrairement aux systèmes mono-programmation, un système multiprogrammation nécessite un arbitrage permettant de partager le temps processeur entre

différents processus. Ce rôle est assuré par l'ordonnanceur.

Le principal intérêt de ce type d'OS (Operating System) est de maximiser le taux d'utilisation du CPU (Central Processing Unit) en permettant à ce dernier d'exécuter un processus quand un autre est en attente, par exemple, lors de requêtes I/O (Input / Output). [14, p.261].

Les systèmes informatiques gagnant en complexité (processeurs multi-coeurs, cloud computing, ...), la notion d'ordonnanceur peut être étendue à la gestion de l'allocation des ressources matérielles aux processus et aux threads.

1.2.1 Niveaux d'ordonnancement [2, p.416-421]

Il est proposé dans [2, p.416-421] de subdiviser l'ordonnanceur en quatre sous fonctions indépendantes permettant de répartir des tâches.

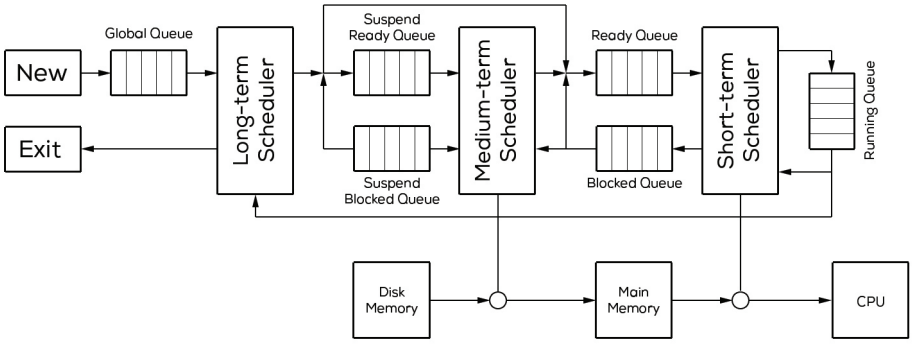

FIGURE 1.1: Fonctionnement général d'un ordonnanceur

Les trois premières concernent la migration des tâches entre les différents niveaux de mémoires jusqu'à leur exécution au sein de l'unité de calcul (Ordonnancement à long, moyen ou court-terme). Tandis que l'ordonnancement I/O se préoccupe uniquement d'ordonnancer les requêtes d'entrée/sortie.

Ordonnancement à long-terme

L'ordonnancement à long terme vise à décider quelle tâche aura accès au système. Il contrôle le degré de multiprogrammation[1] [2, p.417]. Seules la création de nouveaux processus et leur sortie du système sont prises en charge par cette partie de l'ordonnanceur. Son exécution est relativement peu fréquente.

[1]Le nombre de processus pouvant être exécutés simultanément. C'est-à-dire, le nombre de processus présents dans l'ordonnanceur à moyen-terme

Ordonnancement à moyen-terme

Les processus créés par l'ordonnanceur à long-terme sont traités par l'ordonnanceur à moyen-terme. Celui-ci décide quels processus doivent être chargés partiellement ou complètement dans la mémoire vive et être envoyés vers l'ordonnanceur à court-terme [2, p.416]. Sa fréquence d'exécution est intermédiaire entre l'ordonnancement à court et long terme.

Ordonnancement à court terme

L'Ordonnancement à court terme [2] est le contrôle le plus fin de l'exécution des processus. Son exécution est déclenchée par les événements suivants [2, p.420] :

- Interruptions temporelles
- Interruptions I/O
- Appels systèmes
- Signaux

Il permet de contrôler précisément quel processus est exécuté à quel moment. Son rôle principal est d'exécuter des processus en attente quand d'autres sont bloqués pour diverses raisons (attente d'interruption, requêtes I/O, mise en attente, etc.).

L'ordonnanceur I/O

L'ordonnanceur I/O gère les requêtes d'entées / sorties envoyées par les processus et les priorités d'accès aux dispositifs.

1.2.2 Ordonnancement multi-processeur[14]

L'organisation de tâches pour un système multiprocesseur introduit la notion de répartition des ressources. Il existe trois méthodes de répartition pour organiser des tâches parmi plusieurs processeurs : global, partitionné, semi-partitionné.

[2] Parfois appelé Dispatcher dans la littérature

Ordonnancement global

Toutes les tâches sont placées dans une queue unique partagée par l'ensemble des processeurs. A chaque ordonnancement, l'ordonnanceur répartit les tâches parmi les processeurs.

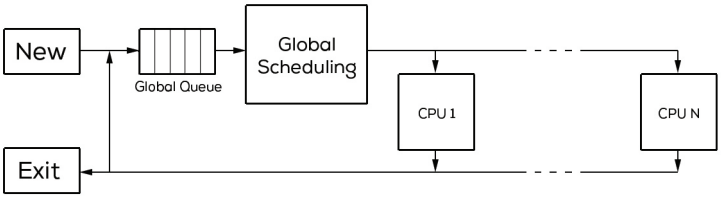

FIGURE 1.2: Fonctionnement d'un ordonnanceur global

Ce type d'ordonnanceur est capable, à chaque ordonnancement, de calculer une répartition proche de l'optimal au coût d'une migration de tâches inter-processeur fréquentes.

Avantages

- Répartition (proche de l') optimale
- Équitable pour chaque processus

Inconvénients

- Mauvaise localité de la mémoire cache
- Mauvaise évolutivité (La queue est partagée pour N processeurs)

Ordonnancement partitionné

Les tâches sont réparties initialement de façon à ce que, pour chaque processeur, un ordonnancement soit possible. De cette façon, les tâches ne migrent pas d'un processeur à l'autre.

Avantages

- Facile à implémenter
- Pas d'arbitrage nécessaire au niveau de la queue.
- Meilleur localité de la cache

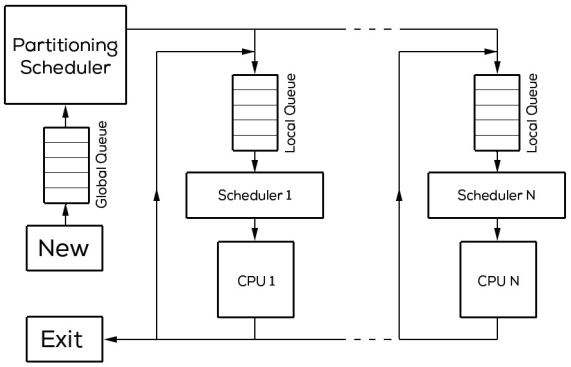

FIGURE 1.3: Fonctionnement d'un Ordonnanceur Partitionné

Inconvénients

- Tâches mal réparties
- Ordonnancement non-équitable
- moins bonne utilisation du CPU

Ordonnancement semi-partitionné

Il s'agit d'un compromis entre les deux premières méthodes. Les tâches sont pré-distribuées entre les processeurs mais peuvent migrer d'un processeur à l'autre.

1.2.3 Ordonnancement multi-threads

Le multi-threading, contrairement au multi-processing vise à augmenter la vitesse d'exécution d'une application donnée soit en exploitant le parallélisme offert par un processeur, soit en exploitant les temps d'attentes pour exécuter d'autres calculs.

L'ordonnancement de tâches multi-threadées, dans le cadre d'un système multi-processeur, ajoute une contrainte supplémentaire qui est la localité de la mémoire. Cette contrainte rend la performance du multi-threading dépendante de l'ordonnanceur et de l'architecture processeur.

Bien qu'un tel système puisse améliorer significativement les performances d'une application, cette avancée est limitée par certains facteurs.

Amdahl remet en cause l'idée selon laquelle le calcul parallèle diminue forcément le temps de calcul. Il montre [15] les performances de 3 machines, comparables en terme de complexité, en fonction du niveau de parallélisation atteignable (voir Fig. 1.4). Une machine

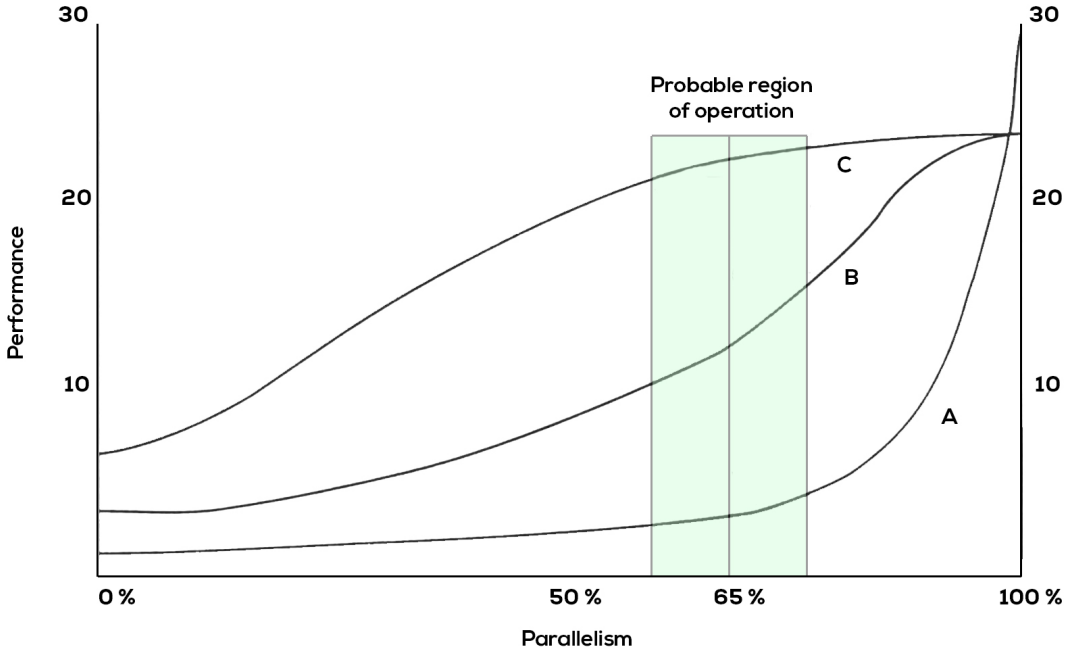

FIGURE 1.4: Parallèle vs Séquentiel - réinfographié sur base de [15]

A avec 32 unités arithmétiques en parallèle, une machine B avec 3 unités pipelinées vectorielles d'une largeur de 8 unités arithmétiques, et une machine C identique à B mais fonctionnant à la fréquence de A. Ainsi, il montre que le temps d'exécution d'un calcul exploitant le parallélisme ne peut être inférieur au temps d'exécution de la plus longue chaîne séquentielle. Ce qui explique le gain en performance des machines B et C.

Amdahl modélise le gain potentiel du parallélisme par l'équation suivante [14, p.167] nommée Loi d'Amdahl :

$$SpeedUp \leq \frac{1}{S + \frac{1-S}{P}}$$

Où S représente la portion de code non parallélisable et P le nombre de processeurs disponible. Cette loi est représentée Fig. 1.5. Cette loi montre que le gain potentiel obtenu en multipliant les processeurs tends vers $1/S$. De ce point de vue, l'intérêt des systèmes multiprocesseurs semble limité excepté pour les tâches complètement parallélisables. Dans ce dernier cas, l'équation tends vers P.

Cette loi est relativisée par Gustafson. Ce dernier affirme dans [16] que, si pour un problème statique, la loi d'Amdahl est correcte, elle ne tient pas compte du fait que la parallélisation permet de traiter des quantités plus importantes de données. Ce dernier modélise le gain potentiel par l'équation [17] :

$$W = \beta T_{par} R + p(1-\beta) T_{par} R$$

$$T_{seq} = W/R$$

$$SpeedUp = \frac{T_{seq}}{T_{par}} = \beta + p(1-\beta)$$

Où W représente le travail total à exécuter, T_{par} le temps total d'exécution parallèle, R le taux d'exécution d'un processeur, p le nombre de processeurs, β la proportion de temps T_{par} qui peut être exécutée dans la partie séquentielle et T_{seq} le temps d'exécution pour un seul processeur. La différence entre les deux modèles est due au fait que Gustafson tient compte du fait que l'augmentation du nombre de processeurs permet de traiter plus de données. Le bénéfice se calcule donc en considérant le temps qu'une machine purement séquentielle prendrait pour traiter les données supplémentaires. Cette manière de montrer le gain obtenu en utilisant le parallélisme permet de se rendre compte de son utilité pour certaines applications. Mais elle ne conteste en rien la limite théorique modélisée par la loi d'Amdahl. Gustafson contredit simplement l'estimation faite par Amdahl en se qui concerne le taux de parallélisation qu'il est possible d'obtenir.

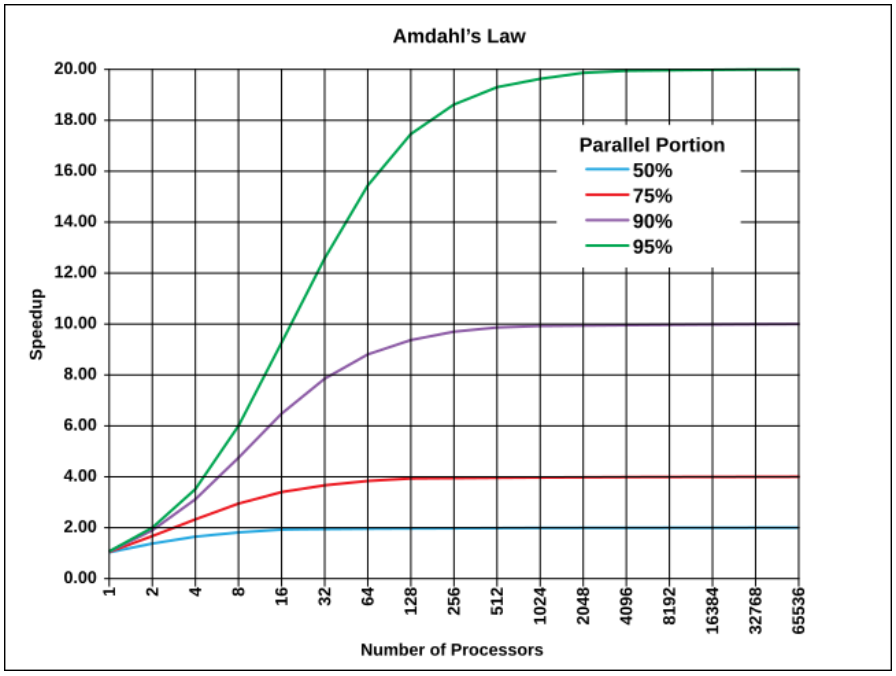

FIGURE 1.5: Loi d'Amdahl [18]

La loi d'Amdahl constitue une limite théorique qui ne tient pas compte des différents overheads provoqués par le système d'exploitation et par le hardware. Il a été montré expérimentalement par Alan H. Karp et Horace P. Flatt [19] qu'il est possible de modéliser ce coût supplémentaire en considérant que la fraction non parallélisable de

l'exécution est dépendante du nombre de processeurs. L'équation du gain potentiel peut donc s'écrire :

$$SpeedUp = \frac{1}{S(P) + \frac{1-S(P)}{P}}$$

Et en tenant compte du fait qu'un système puisse exécuter un nombre de processus supérieur au nombre de processeurs et des overheads qui leur sont liés :

$$SpeedUp = \frac{1}{S(T,P) + \frac{1-S(T,P)}{P}}$$

Où T représente le nombre de processus ou threads.

Un simple overhead tel que S(1) = 0 et S(T) = 0.01 ferait passer le gain maximum théorique de l'infini à 100. Ceci justifie largement l'intérêt d'optimiser ces différents overheads afin de minimiser S(T). Ceci constitue l'une des motivations justifiant le travail décrit chapitre 3.

1.3 Critères de caractérisation des processus

Sur base de [2, 9, 14], cette section vise à énumérer et définir un ensemble de catégories permettant de classer les processus et d'en dégager un ensemble de paramètres pouvant intervenir dans la décision d'un ordonnanceur. Bien que ce travail vise à être le plus complet possible dans le recensement de ces catégories et paramètres, pour les mêmes raisons que pour les critères d'évaluation, celui-ci ne saurait être exhaustif.

Les systèmes d'exploitations décrits dans la littérature citée classent les processus sur base de catégories. Une synthèse de ces catégories pour différents systèmes d'exploitations est proposée Table 1.1.

Sur base de ce tableau, on constate que ces catégories mélangent plusieurs aspects du système :

- **Le niveau d'autorisation du processus** : toutes les catégories de type "Système".
- **Les aspects temporels du processus** : Les catégories "Interactive" et "Real-time".
- **La politique d'ordonnancement utilisée** : Les catégories : "FIFO", "Round-Robin", "Fair share", etc.

	Solaris [14, p.297]	**FreeBSD** [2, p.483]	**Linux Kernel 2.6** [2, p.443, 477-479]	**OS X** [20]
System	System	Bottom-half kernel	System	Kernel mode only
		Top-half kernel		System high priority
Normal	Time sharing	Time sharing	Other	Normal
	Fair share	Idle User		
Real-time	Fixed priority	Real-time	FIFO	Real-time
	Real-time		Round-Robin	
	Interactive			

TABLE 1.1: Classification des processus pour différents systèmes d'exploitation

Le fait de catégoriser les processus réduit l'information disponible sur leurs propriétés. Ce travail propose donc, plutôt que de classer les processus en catégorie, de définir les variables qui les différencie les uns des autres et de considérer la politique d'ordonnancement séparément.

1.3.1 Niveaux d'autorisations

Comme constaté Table 1.1, certaines catégories de processus peuvent opérer à différents niveaux de privilèges. Les catégories de processus s'exécutant dans la couche noyau (Kernel) ont accès au hardware et reçoivent une priorité d'exécution plus importante. Dans le cas des noyaux monolithiques, seul le noyau a accès au hardware. Les micro-noyaux quant à eux, exécutent les drivers dans la couche utilisateur et permettent donc un accès au hardware depuis la couche utilisateur.

Afin de gagner en généralité, il est possible de considérer le niveau d'autorisation d'un processus en faisant abstraction de son niveau de priorité et de la manière dont celui-ci est traité par l'ordonnanceur.

1.3.2 Utilisation des ressources

On rencontre dans la littérature [14, p.113][2, p.419][9, p.43] une proposition de classification basée sur le fait qu'un processus puisse être de type "I/O-Bound" ou "CPU-Bound". C'est-à-dire que celui-ci peut soit être limité par le processeur (peu de requêtes I/O) ou par les requêtes I/O (peu de calcul), soit être équilibré. Le principal intérêt de cette catégorisation est de permettre à un ordonnanceur d'optimiser l'utilisation des ressources.

Le temps de calcul et l'accès aux entrées / sorties peuvent être considérés comme deux ressources particulières d'un processeur. Partant de ce constat, il est possible

d'élargir ce critère à la totalité des ressources partiellement ou totalement indépendantes[3] disponibles. De cette façon, la définition de ce critère de classification n'est plus limitée à une architecture donnée et permet d'envisager, par exemple, des proccessus Memory-Bound, FPU-Bound, GPU-Bound et éventuellement l'appliquer à des ressources n'ayant pas encore été inventées.

1.3.3 Périodicité

Il est possible de différencier les processus en deux types : les processus périodiques et non périodiques. Bien que l'on pourrait considérer une tâche non périodique comme étant une tâche périodique de fréquence nulle, il est intéressant de les classer en deux groupes dans la mesure où l'exécution d'une tâche périodique pourra être anticipée afin d'assurer des contraintes temps réel. Pour une tâche non périodique, l'objectif sera de minimiser son temps d'exécution et donc son temps de réponse. Bien que l'on puisse distinguer les processus de type batch[4] et les processus interactifs [21], il est possible de considérer un processus interactif comme étant un processus batch ayant des contraintes temporelles en terme de temps de réponse et de temps exécution.

1.3.4 Contraintes temporelles

Les contraintes temporelles peuvent être séparées en deux caractéristiques distinctes : le temps de réponse et le temps de calcul. La première caractéristique définit le temps qu'un processus peut attendre avant d'être exécuté tandis que la seconde définit le temps qu'il peut prendre pour s'exécuter. Bien que les processus ayant des contraintes temporelles sont généralement regroupés en deux catégories (processus temps réel et processus à contraintes temps réel strictes), seule la gravité d'un non respect des délais diffère. Si un système est capable de respecter des contraintes temps réel strictes, il sera aussi capable d'exécuter toutes formes de processus temps réel.

1.4 Critères d'évaluation des ordonnanceurs

Cette section est basée sur deux livres de référence traitant des systèmes d'exploitations [2, 14]. Elle vise à énumérer, de façon non exhaustive [5] , un ensemble de critères

[3] La notion d'indépendance est utilisée, dans ce contexte, pour illustrer le fait que deux ressources peuvent être utilisées en parallèle.

[4] Processus apériodique ayant pour rôle d'effectuer une quantité importante de calculs

[5] Étant donné la diversité des façons dont il est possible d'ordonnancer des processus et étant donné que la méthode utilisée dépend des cas d'utilisations d'un système qui, par définition ne peuvent être limités qu'à l'imagination de leur concepteur, la recherche de l'exhaustivité n'apparaît pas pertinente

permettant de juger de la qualité d'un scheduler.

1.4.1 Orienté performance du système

Taux d'utilisation CPU [14, p.265][2, p.421]

Ce critère évalue le nombre de cycles pendant lequel le CPU est utilisé par rapport au nombre de cycles total.

Débit d'exécution [14, p.265][2, p.421]

Ce critère évalue le nombre de tâches exécutées par unité de temps.

Répartition des ressources [2, p.421]

Ce critère évalue la bonne répartition de l'utilisation des différents périphériques et composants d'un processeur.

1.4.2 Orienté performance d'un processus

Temps de réponse [14, p.265][2, p.421]

Il s'agit du temps mesuré entre la soumission d'une tâche et la première réponse calculée par celle-ci.

Turnaround time [14, p.265][2, p.421]

Il s'agit du temps mesuré entre la soumission d'une tâche et son achèvement.

Temps d'attente [14, p.265]

Il s'agit du temps qu'une tâche passe à attendre l'accès à une ressource quand elle est prête à démarrer ou continuer son exécution.

1.4.3 Orienté contraintes

Équité [2, p.421]

Ce critère évalue le fait qu'il n'y ait pas de "starvation[6]" et que les différents processus soient traités de la même façon.

Deadlines [2, p.421]

Dans le cas de processus spécifiant une date limite d'exécution, ce critère évalue le respect de cette contrainte.

Prédictibilité [2, p.421]

Ce critère évalue la mesure dans laquelle un temps d'exécution n'est pas dépendant de la charge totale du système. C'est-à-dire que le temps de réponse et le "turnaround time" restent identiques d'une exécution à une autre.

Respect des priorités [2, p.421]

Ce critère évalue si l'ordonnanceur respecte bien les priorités d'exécution définies par le système.

1.4.4 Critères indépendants des processus

La plupart de ces critères sont fortement dépendants des tâches à exécuter, du système et du processeur. Dans ce travail, la réflexion porte sur la réalisation d'un ordonnanceur hardware et non d'un système. Afin de simplifier ces critères d'évaluation et de les rendre compatibles avec ce qui doit être mesuré, on émet l'hypothèse de processus parfait selon ces propositions :

- Si le processus nécessite le respect de deadline, celui-ci doit disposer d'une borne maximale en terme de nombre d'instructions à exécuter.
- Si le processus nécessite un temps d'exécution déterministe, celui-ci doit lui même avoir un nombre d'instructions à exécuter déterministe.

Ces critères peuvent donc se résumer par :

[6] Terme utilisé en anglais pour décrire le fait qu'un processus n'est jamais exécuté

l'overhead

L'overhead représente, dans le domaines des systèmes d'exploitations et des processeurs, un temps de calcul qui n'est pas directement utilisé par un processus ou un temps de propagation d'un périphérique qui empêche l'exécution de ce dernier. Dans le cadre de l'étude des ordonnanceurs le terme représente le temps utilisé pour calculer l'ordre d'exécution d'une liste de processus[22].

L'overhead a un impact direct sur les critères suivants : taux d'utilisation CPU, débit d'exécution, le temps de réponse et le temps d'attente.

Le jitter

le jitter représente l'incertitude [22] sur le temps de calcul d'un ordonnancement. Ce facteur est critique pour les processus temps réel.

Ce critère a un impact direct sur les critères suivants : la prédictibilité et le respect des deadlines. Une borne supérieure sur le temps d'ordonnancement suffit au respect de deadlines.

Le respect des priorités

Le respect des priorités a déjà été discuté et a une influence sur le respect des deadlines et la prédictibilité.

L'équité

Ce critère a déjà été discuté et a une influence sur le temps d'attente et le "turnaround time".

Le temps de réponse

Le temps de réponse de l'ordonnanceur est principalement lié à son overhead mais le temps de gestion des interruptions et des événements peut aussi interférer avec ce critère.

La répartition des ressources

Déjà décrit parmi les critères généraux, ce critère a aussi un impact sur le taux d'utilisation du CPU.

1.5 Ordonnancement hardware

De façon analogue aux ordonnanceurs software, les ordonnanceurs hardware s'occupent de déterminer quels processus ou quelles instructions doivent être exécutés par un ou plusieurs processeurs. Cette section présente l'intérêt qu'ils ont par rapport à leurs homologues softwares. Leur objectif est de, premièrement, décharger le processeur de l'exécution du software qu'ils remplacent et, deuxièmement, d'assurer des contraintes temps-réel qu'un ordonnanceur software ne serait pas à même de respecter. L'ordonnancement d'instructions sera discuté dans le cadre des microarchitectures processeur Chapitre 2.

La plupart des ordonnanceurs sont implémentés dans la couche software. Le problème d'ordonnancement étant un problème NP-complet[23], seule une heuristique peut être utilisée pour le résoudre en un temps raisonnable. Une heuristique étant par définition imparfaite, le fait de l'implémenter dans la couche software permet aux développeurs d'y apporter des améliorations sans devoir changer le matériel. Ceci est la principale justification expliquant leur popularité. En contrepartie, ceux-ci sont globalement moins performants que leur homologue hardware, ce qui impacte principalement les systèmes temps réels mais aussi les systèmes interactifs ayant des contraintes lourdes en terme de temps de réponse.

1.5.1 Aperçu des travaux étudiés

L'ordonnanceur développé par Jason Agron, David Andrews, Mike Finley, Wesley Peck et Ed Kom [22] est une solution hardware implémentée sur FPGA fonctionnant à travers un Bus.

Celui développé par Pramote Kuacharoen, Mohamed A. Shalan et Vincent J. Mooney [24] est une solution hardware/software implémentée sur FPGA et ASIC (Application-Specific Integrated Circuit). Il fonctionne aussi à travers un Bus, le processeur accède aux fonctionnalités de l'ordonnanceur via des requêtes IO.

Le travail de Nikhil Gupta, Suman K. Mandal, Javier Malave, Ayan Mandal et Rabi N. Mahapatra [25] propose une comparaison entre une implémentation software et hardware d'un même algorithme d'ordonnancement. Leur solution hardware a été implémentée sur ASIC.

Le travail de Vasile Gheorghita Gaitan, Nicoleta Cristina Gaitan, Member, et Ioan Ungurean [4, 5] propose une solution FPGA complètement hardware intégrée à une microarchitecture RISC.

Celui de Itai Avron et Ran Ginosar [13] propose une simulation d'ordonnanceurs hardware selon différentes configurations pour des systèmes de type "many-core".

Cette étude s'est limitée à ces cinq travaux car ils couvrent l'ensemble des sujets nécessaires à la discussion qui suit. Seul le travail de Vasile Gheorghita Gaitan et Al. apporte une solution proche des objectifs choisis pour ce mémoire. Une comparaison entre cette dernière solution, les deux premières et celle développée chapitre 3 est proposée section 3.5.3.

1.5.2 Impact sur les performances

Le temps de calcul d'un ordonnancement en fonction de sa fréquence d'exécution représente l'overhead. Ce même temps de calcul dépensé lors d'un appel extérieur représente son temps de réponse. Minimiser ce temps de calcul d'un ordonnanceur revient donc à minimiser à la fois l'overhead et le temps de réponse.

Un overhead lié au calcul de l'ordonnancement software est inévitable. S'agissant d'une tâche répétitive, une partie du processeur est donc monopolisée pour effectuer ce calcul à intervalle régulier. Bien qu'il soit possible d'implémenter un ordonnanceur ayant une complexité temporelle de O(1)[7][26], le caractère périodique de l'algorithme implique qu'un pourcentage de la puissance de calcul est monopolisé pour cette tâche.

Les sources à l'origine du déclenchement de l'ordonnanceur, comme précisé section 1.2.1, sont déclenchées par : les interruptions I/O; les interruptions du temporelles; les appels systèmes; les signaux. Exceptées les interruptions temporelles qui sont déclenchées à chaque "quantum time"[8], les autres sources sont à priori imprévisibles.

Des mesures de l'overhead en fonction du "quantum time" pour différentes charges en terme de nombre de processus ont été réalisées par [24] et sont représentées Fig. 1.6. L'ordonnanceur mesuré est celui du système d'exploitation Micro-C/OSII et est basé sur une politique d'ordonnancements à priorité. Tout les articles étudiés [4, 13, 22, 24, 25] s'accordent à dire qu'un ordonnanceur hardware permet de réduire significativement l'overhead.

L'article [25] propose une comparaison entre trois implémentations pour un ordonnanceur :

- Ordonnanceur software répliqué sur chaque processeur.
- Ordonnanceur software exécuté sur un processeur dédié.

[7] voir "Big O notation"
[8] Intervalle de temps entre deux déclenchements de l'ordonnanceur

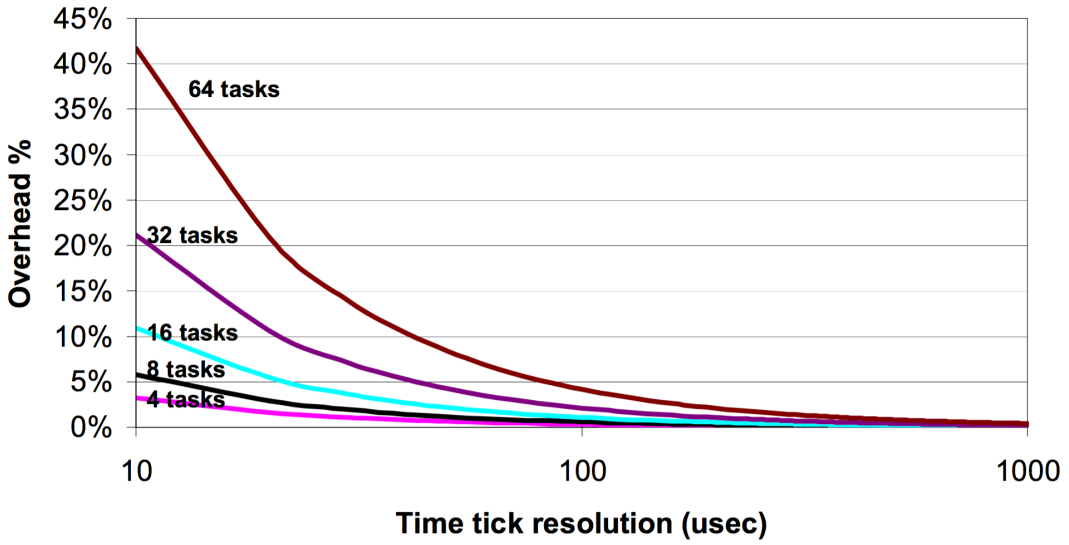

FIGURE 1.6: The scheduler and the time tick processing overheads in MicroC/OS II[24, Figure 8]

- Ordonnanceur hardware.

Il conclut que la version hardware réduit l'overhead de 3 ordres de grandeur (10^3) (voir Fig. 1.7. L'article [4] va plus loin en proposant un ordonnanceur capable d'opérer en moins de 3 cycles processeur. La solution des auteurs sera détaillée chapitre 3. Ces derniers précisent que les solutions classiques d'ordonnancement hardware (via un co-processeur) ne suppriment pas totalement l'overhead du fait que ceux-ci communiquent à travers un bus. Ils proposent donc d'intégrer l'ordonnanceur à la microarchitecture. C'est cette approche qui sera développée chapitre 3.

Rares sont les systèmes ayant besoin d'une résolution d'ordonnancement suffisamment fine pour induire une baisse significative des performances mais améliorer cette résolution offre néanmoins de nouvelles perspectives en terme de calcul temps-réel.

Le "jitter" représente l'incertitude [22] sur le temps de calcul d'un ordonnancement. L'algorithme O(1) permet d'obtenir une borne supérieure en terme de temps d'exécution. Cela permet à l'ordonnanceur d'être à même de respecter une deadline pour un processus temps réel, mais ne permet pas de garantir un temps de réponse constant. L'ordonnancement hardware permet de réduire significativement le "jitter" [4, 22]. La solution proposée chapitre 3 permet de le supprimer complètement.

Le respect des priorités, l'équité et la répartition des ressources sont liés à la politique d'ordonnancement et non à leur implémentation software ou hardware. Ces critères ne seront donc pas discutés dans cette section.

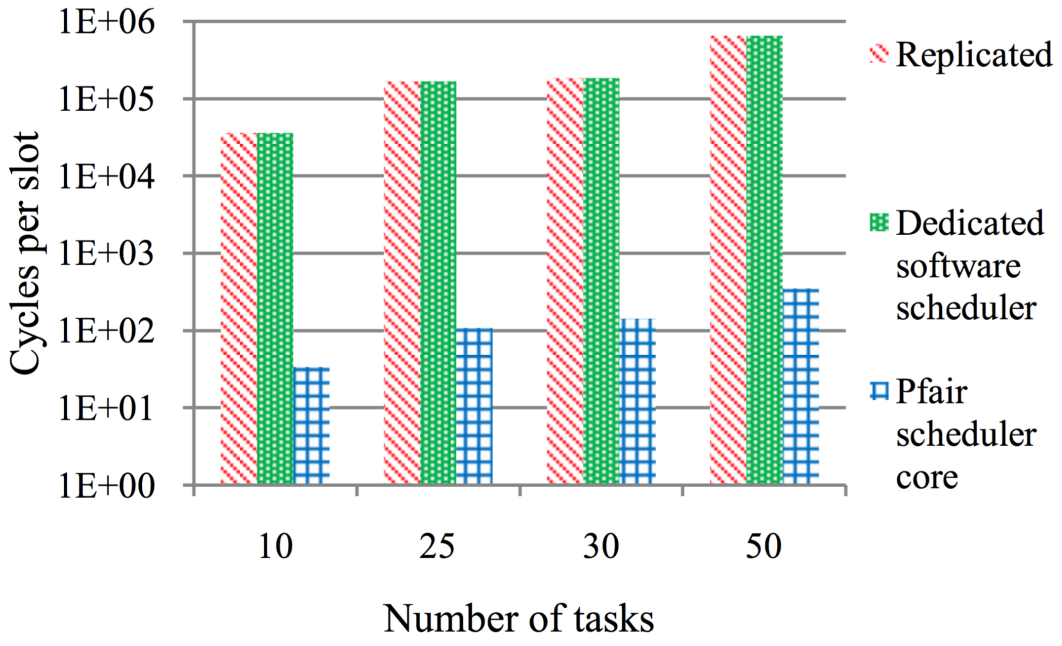

FIGURE 1.7: Scheduling Delay [25, Figure 6]

Seule l'une des solutions hardware permet de calculer un ordonnancement en un temps de l'ordre du cycle processeur[4]. L'autre solution hardware[22] utilise une gestion de l'ordonnancement multi-cycles ayant l'avantage de gérer un plus grand nombre de threads tout en minimisant la quantité de circuit nécessaire. La solution d'un co-design hardware/software offre le même type d'avantage qu'une solution multi-cycles et offre une plus grande flexibilité concernant la politique d'ordonnancement. Ce type d'implémentation souffre des mêmes inconvénients tout en rendant impossible le masquage complet de l'overhead lié au temps de calcul d'ordonnancement.

Il est précisé par Jason Argon et Al. [22] que le temps de calcul d'ordonnancement hardware multi-cycles est plus petit que le temps de changement de contextes et que cela permet de supprimer l'overhead. L'approche de Vasile Gheorghita Gaitan et Al. [4] et celle développée chapitre 3, reposent sur une microarchitecture capable d'effectuer des changements de contextes de l'ordre du cycle processeur. Ce temps est bien entendu trop court pour permettre l'utilisation de solutions multi-cycles.

L'étude de Itai Avron et Ran Ginosar [13] explique que les systèmes "many-core" nécessitent de bonnes performances d'ordonnancement et qu'une implémentation hardware permet d'obtenir de meilleurs performances. Ils ont aussi montré qu'un ordonnanceur local pour aider l'ordonnanceur global améliore significativement les performances.

1.6 Conclusion

Il a été montré section 1.2.3 que les différents overheads d'un système ont un impact sur les performances multithread. Il a ensuite été montré section 1.5 que l'approche hardware du développement d'ordonnanceurs permettait de réduire significativement l'overhead et le jitter liés au calcul d'ordonnancement. Il a aussi été présenté dans cette section qu'une solution intégrée à une microarchitecture pouvait aussi réduire ces facteurs pour le changement de contexte. Enfin, l'utilité des ordonnanceurs [13] et micro-noyaux [12] a été présentée dans le cadre des systèmes "many-core", et il a été mis en évidence que la séparation entre ordonnanceur local et global pouvait améliorer les performances de ce type de système.

Cette voie de solution permet non seulement d'obtenir une résolution d'ordonnancement impossible à atteindre pour des solutions software mais aussi d'améliorer significativement le déterminisme pour les systèmes temps réel. De plus, la réduction considérable, voire l'annulation, des différents overheads laisse entrevoir la possibilité d'approcher, voire d'atteindre, la limite théorique du niveau parallélisme atteignable énoncée par Amdhal.

Sur base de cette analyse théorique, il a été décidé de suivre la démarche proposée par Vasile Gheorghita Gaitan et Al. [4]. C'est-à-dire de travailler sur la question de l'ordonnancement hardware d'un point de vue d'une microarchitecture et non d'un système distinct. Pour y parvenir, le chapitre 2 étudiera la question des microarchitectures dans le cadre de l'ordonnancement d'instructions. Les conclusions de Itai Avron et Ran Ginosar [13] orientent ce travail dans la direction du développement d'un ordonnanceur local sans traiter des aspects plus complexes qu'un ordonnanceur complet devrait respecter. Pour être complet, le système devrait être enrichi d'un ordonnanceur global externe. Plus de détails seront apportés chapitre 3.

Chapitre 2

Architectures, microarchitectures et ordonnancement d'instructions

Ce chapitre constitue la base théorique sur laquelle s'appuie le développement du microprocesseur soft-core décrit Chapitre 3. Seront passés en revue les grandes familles de processeurs ainsi qu'une série d'innovations visant à augmenter le nombre d'instructions exécutées par cycle.

Ce travail étant focalisé sur l'étude de l'ordonnancement de tâches et d'instructions, cette partie se limitera à la description des architectures de type Von Neumann et ne traitera pas des mécanismes d'acheminements des données mais uniquement des instructions (voir section 2.1).

La première section expliquera les deux grandes façons de contrôler un flux d'exécution. La seconde s'occupera de détailler les deux grandes familles d'architecture processeur[1]. La troisième passera en revue un ensemble de techniques microarchitecturales permettant d'augmenter le débit d'instructions. La section 4 présentera différentes façons de combiner ces techniques. Les sections 5 et 6 identifieront les contraintes techniques liées aux microarchitectures et des solutions pour les résoudre. Pour finir, la section 7 analysera l'impact de ces microarchitectures sur des systèmes temps réel strict.

Le rôle central d'un processeur est d'exécuter des instructions. Pour y parvenir celui-ci doit implémenter des mécanismes permettant d'acheminer les instructions et les données depuis la mémoire jusqu'aux unités de calculs tels que l'ALU ou le FPU.

[1] Une architecture processeur représente la façon de coder les instructions processeurs. Une microarchitecture correspond à l'implémentation du microprocesseur chargé de traiter ces instructions.

La performance d'un processeur dépend directement du nombre d'instructions et de données acheminées et traitées par unité de temps, ainsi que du nombre d'instructions nécessaires pour accomplir une tâche.

Voici une mise en équation de la performance d'un ordinateur décrite comme couramment utilisée par [27] :

$$\frac{time}{program} = \frac{time}{cycle} * \frac{cycles}{instruction} * \frac{instructions}{program} \tag{2.1}$$

En y ajoutant la capacité à exécuter plusieurs programmes :

$$\frac{time}{system} = \frac{time}{cycle} * \frac{cycles}{instruction} * \frac{instructions}{program} * \frac{programs}{system} \tag{2.2}$$

Le nombre de cycles par instructions sur une machine capable d'exécuter plusieurs instructions par cycle sera dépendant du niveau de parallélisme atteignable :

$$\frac{instructions}{cycle} = min(\frac{instructions}{cycle}, \frac{ILP * TLP}{cycle}) \tag{2.3}$$

où ILP représente le nombre d'instructions parallélisables en moyenne par flux d'instructions (processus) et TLP le nombre de threads ou processus que la machine peut traiter par cycle.

Vers 2005, AMD et Intel ont arrêté la course vers la montée en fréquence de leur processeur. L'article [28] explique que, comme la montée en fréquence augmente la consommation électrique de façon quadratique, d'autres voies moins consommatrices d'énergies sont à privilégier.

Il apparaît clairement (Fig. 2.1) qu'Intel continue à augmenter son débit d'instructions en multipliant les cœurs de ses processeurs et a délaissé la montée en fréquence et l'ILP (voir section 2.3).

Pour se convaincre de la limite de montée en fréquence et de l'intérêt que peuvent apporter d'autres voies d'améliorations, nous pouvons prendre l'équation de la puissance d'un circuit CMOS [29] :

$$P_{moy} = \alpha * C * V_{dd}^2 * f + (I_{cc} + I_l) * V_{dd} \tag{2.4}$$

Où α représente le nombre moyen de transitions on/off des transistors, C la capacité physique du circuit, I_{cc} le courant de coupe circuit, et I_l le courant de fuite.

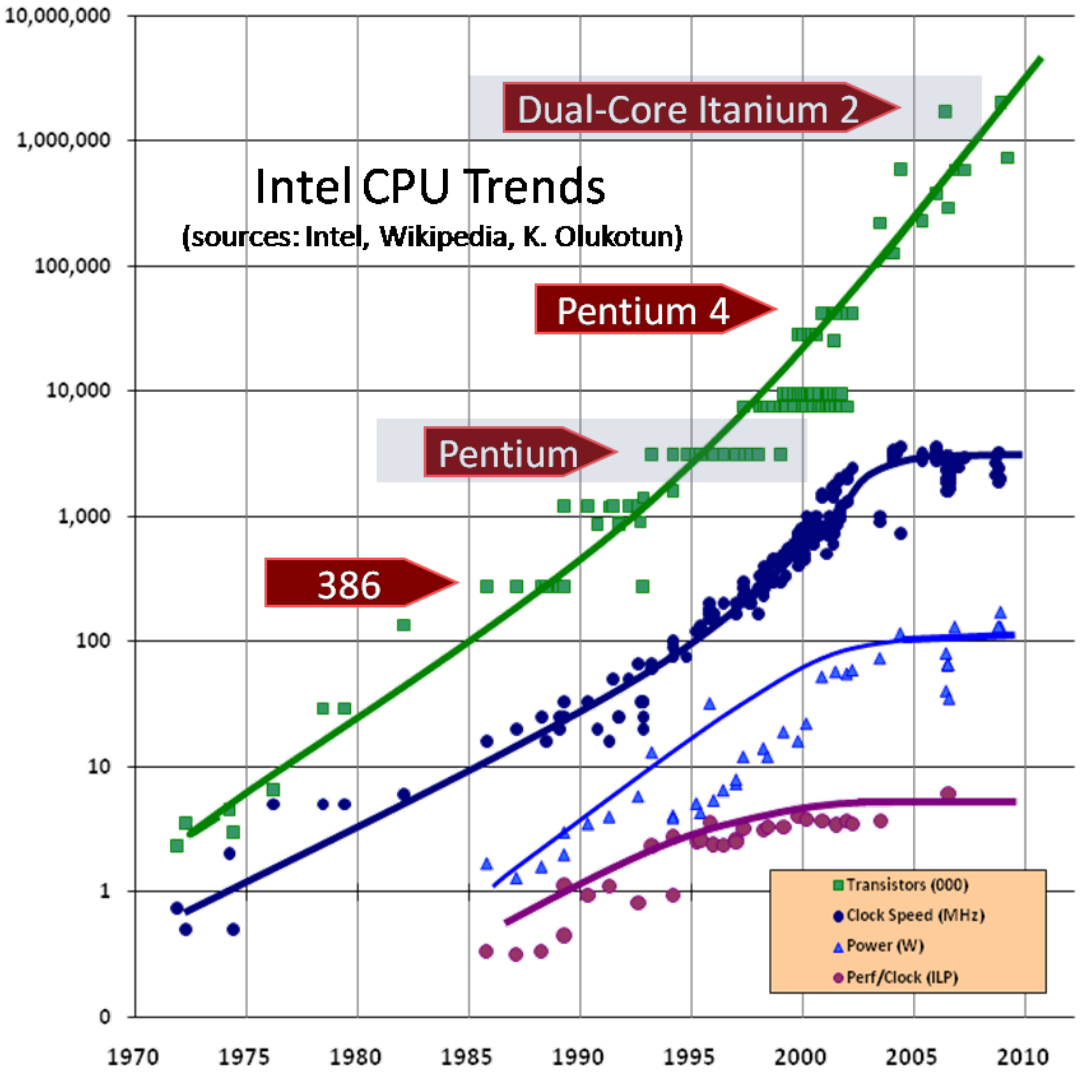

FIGURE 2.1: Évolution des processeurs Intel [1]

Sachant que la fréquence maximum atteignable est dépendante de V_{dd} on peut écrire :

$$P_{moy} = \alpha * C * V_{dd}^2(f) * f + (I_{cc} + I_l) * V_{dd}(f) \tag{2.5}$$

L'énergie dissipée par un travail T exprimé en nombre de cycles nécessaires pour une exécution en tenant compte du nombre de transistors n est donné par :

$$t = T(n)/f \tag{2.6}$$

$$E = P_{moy} * t = \alpha * C(n) * V_{dd}^2(f) * T(n) + (I_{cc}(n) + I_l(n)) * V_{dd}(f) * \frac{T(n)}{f} \tag{2.7}$$

Cette équation montre, entre autres, qu'il est possible d'obtenir une économie d'échelle sur l'énergie dissipée en diminuant la fréquence de fonctionnement du circuit. Cette

économie possible limite donc l'intérêt de la montée en fréquence.

Bien qu'historiquement la consommation statique $(I_{cc}(n) + I_l(n)) * V_{dd}(f) * \frac{T(n)}{f}$ était négligeable, ce n'est plus le cas avec l'évolution de la finesse de gravure [30] et il devient nécessaire de la prendre en compte.

Hormis le compromis à trouver entre $V_{dd}(f)$ et f, il reste à optimiser $\alpha * C(n) * T(n)$ et $(I_{cc}(n) + I_l(n)) * T(n)$ en fonction de la consommation et du nombre de transistors (coût). Le paramètre α peut être optimisé en travaillant sur la gestion électrique du microprocesseur, tout comme $C(n)$ $I_{cc}(n)$ et $I_l(n)$ en travaillant sur la technologie enployée et $T(n)$ sur la microarchitecture.

Bien que ce travail s'intéresse aux microarchitectures d'un point de vue de la puissance de calcul et des fonctionnalités qu'elles peuvent apporter plus qu'à leur consommation et leur coût, ces équations permettent de montrer l'intérêt d'augmenter le débit et l'efficacité des instructions sans passer par une montée en fréquence.

Ce type d'amélioration peut se faire à deux niveaux :

- Le jeu d'instruction : $\frac{instructions}{program}$
- Le débit d'instructions par cycle : $\frac{instructions}{cycle}$

Différentes façons d'augmenter le débit d'instructions sont possibles au niveau de l'architecture d'un CPU. Ce chapitre détaillera une série d'innovations micro-architecturales permettant d'améliorer les performances d'un processeur sans considérer les aspects de montée en fréquence.

2.1 Control-Flow et Dataflow

Définition de Control-Flow [3, p.55]

Le modèle de calcul en flux de contrôle (Control-Flow) spécifie quelle instruction doit être exécutée à quel moment en fonction des instructions précédentes via un compteur de programme (PC) correspondant à l'adresse de l'instruction. Ces instructions sont exécutées même si les opérandes ne sont pas prêtes. Le modèle de flux de contrôle le plus connu est le modèle de Von Neumann.

Définition de Dataflow [3, p.55]

Ce modèle organise une exécution en fonction de la disponibilité des opérandes. Il n'y a donc pas besoin de compteur de programme et ce modèle permet, en théorie [3, Preface XI] d'exploiter la totalité du parallélisme (voir section 2.3) disponible d'un programme.

Bien que l'approche Dataflow soit prometteuse, ce mémoire ne traitera pas de ce type d'architecture et se concentrera uniquement sur des architectures de type "Control-Flow".

2.2 RISC et CISC

Les architectures processeur peuvent être divisées en deux grands groupes basés sur leur jeu d'instructions :

- **Complex Instruction Set Computer - CISC [3, Preface XI]** : Apparus dans les années 70, les processeurs CISC apportent des fonctionnalités provenant des langages de haut niveau. A cette époque, le coût de la mémoire et de l'acheminement des instructions justifie la création d'instructions complexes enregistrées dans une mémoire ROM interne au processeur. Cela permet principalement de diminuer le nombre d'instructions par programme mais aussi d'exécuter des calculs plus complexes par instruction. Ce type d'architecture se distingue par :
  - Instructions complexes.
  - Instructions de tailles variables.
  - De nombreuses formes d'instructions.
  - De nombreuses formes d'adressage.
  - Disparités du nombre de cycle par instruction (CPI) en fonction des instructions.
  - Peu de registres.
  - Le décodage des instructions se réalise à l'aide d'une mémoire ROM contenant les bits de contrôle.
- **Reduced Instruction Set Computer - RISC [3, Preface XI]** : Quelques années plus tard, naît le concept inverse : réduire le jeu d'instructions aux instructions statistiquement les plus utilisées (voir table 2.1). Cela permet de réduire la complexité de la microarchitecture au prix de programmes nécessitant plus d'instructions et donc une plus grande mémoire cache. La quantité de hardware

Instruction	Moyenne d'utilisation
load	22 %
conditional branch	20 %
compare	16 %
store	12 %
add	8 %
and	6 %
sub	5 %
move register-register	4 %
call	1 %
return	1 %
total	**95 %**

TABLE 2.1: Fréquence d'utilisation des instructions [3, Table 1.1]

ainsi économisée peut être utilisée pour des fonctionnalités spécifiques améliorant significativement les performances. Dans le cadre des puces VLSI (Very Large Scale Integration), l'économie d'espace peut être utilisée pour augmenter le nombre de processeurs par puce.

Ce type d'architecture se distingue par [31] :

- Instructions conceptuellement simples et orthogonales.
- Instructions de taille fixe.
- Un ou quelques formats d'instructions.
- Un ou quelques modes d'adressage.
- La plupart des instructions s'exécutent en un cycle.
- Architectures de type load & store.
- Nombre important de registres.
- L'unité de contrôle et le décodeur sont implémentés en dur (hard-wired).

Aujourd'hui, les processeurs CISC [32] sont généralement implémentés avec des unités fonctionnelles simples comme les processeurs RISC. Les instructions complexes sont décodées en une série d'instructions simples appelée "microcode". Selon Rafael Vidal Aroca et Luiz Marcos Garcia Gonçalves [32], le décodage de ces instructions ajoute un "overhead" impliquant une plus grande consommation d'énergie et une micro-architecture plus complexe.

Selon leurs mesures[32], les processeurs ARM[2] obtiennent de meilleures performances par Watt mais n'atteignent pas les performances des processeurs x86[3]. Cette conclusion est contredite par Emily Blem, Jaikrishnan Menon, et Karthikeyan Sankaralingam [33].

[2]Représentant le plus connu de l'architecture RISC
[3]Représentant le plus connu de l'architecture CISC

Ces derniers précisent que, excepté pour le segment des processeurs ultra-basse consommation[4] où le surcoût de la prise en charge du jeu d'instruction CISC est trop important, il n'existe pas de différence significative en terme de performance et de performance par Watt sur des segments de marchés comparables. Ils concluent que le débat RISC/CISC n'est donc, aujourd'hui, plus pertinent.

Comme expliqué dans l'introduction, il est toujours possible d'obtenir une économie d'échelle en réduisant la fréquence de fonctionnement d'un processeur. Il est donc logique qu'un processeur moins performant atteigne de meilleurs résultats par Watt.

2.3 Parallélisme - ILP et TLP

A fréquence constante et en négligeant les accès mémoire, la seule façon d'augmenter le débit d'instructions est d'en exécuter plusieurs simultanément. Ce constat implique qu'il soit possible de paralléliser les tâches que le processeur doit exécuter.

La parallélisation peut se faire à trois échelles différentes :

- **Process Level Parallelism** : Ce niveau de parallélisme se situe à l'échelle du système. Plusieurs tâches indépendantes peuvent s'exécuter sur une même machine. A ce niveau, seules les communications inter-processus peuvent restreindre le niveau de parallélisme atteignable.
- **Thread-level Parallelism [3, p.249]** : Proche du Process-level, il se distingue par le fait que les communications inter-process, dans le cadre des threads se réalisent par un espace mémoire commun. Cela implique plus de contraintes de synchronisation.
- **Instruction-level Parallelism (ILP [34])** La parallélisation se fait au sein d'un processus, on tente de déterminer plusieurs instructions indépendantes afin de les exécuter simultanément.

Une classification des architectures ILP a été proposée par [34] :

- **Architecture séquentielle** : Le parallélisme n'est pas explicite. Le processeur est seul responsable de la parallélisation des instructions. La microarchitecture superscalaire (voir section 2.3.1) en est une implémentation.

[4] basé sur le RISC ATmega324PA opérant entre 1 et 20MHz avec une consommation de 2 à 50mW

- **Architecture à dépendances explicites**[5] : Les dépendances entre les données sont explicites. Le processeur a donc pour mission d'ordonnancer les instructions suivant les dépendances indiquées. Il s'agit principalement des architectures de type Dataflow.
- **Architecture à indépendances explicites**[6] : Le programme précise quelles sont les opérations indépendantes entre elles. L'architecture VLIW (voir section 2.3.2) est une architecture à indépendance.

Afin de simplifier la discussion, nous pouvons considérer le "process-level parallelism" comme un cas particulier du "thread-level parallelism" où les contraintes de synchronisations sont "quasi" absentes. De cette façon, nous pouvons distinguer, comme proposé par [3, p.221 & p.247], deux grandes catégories de niveaux de parallélisme : le "Coarse-Grain Parallelism", traitement grossier du parallélisme (TLP), et le "Fine-Grain Parallelism", traitement fin du parallélisme (ILP). De plus, une architecture exploitant le TLP peut être considérée comme une architecture à indépendances explicites dans la mesure où le programme explicite les flux d'exécutions indépendants tout en gérant lui-même les mécanismes de synchronisation des données.

Les microarchitectures exploitant le TLP permettent d'augmenter le nombre d'instructions exécutées par cycle en exploitant la possibilité d'exécuter des processus partiellement ou totalement indépendants. L'exploitation de l'ILP se réalise au niveau de l'exécution et/ou de la compilation d'un programme. L'exploitation du TLP se réalise au niveau du design de ce programme ou au niveau utilisateur (exécution de programmes indépendants sur une même machine).

La solution la plus évidente pour obtenir un gain de performance en exploitant ce niveau de parallélisme est de multiplier le nombre de processeurs. Par exemple, si deux processus totalement indépendants s'exécutent sur une machine monoprocesseur et exploitent chacun 50% du temps CPU, doubler ce processeur permettra aux deux processus d'en utiliser 100% et donc, les performances seront doublées. La base des microarchitectures multiprocesseur sera expliquée section 2.3.3.

En pratique, les deux processeurs vont devoir se partager certains niveaux de caches, les accès I/O, ainsi que des périphériques. De plus, si les deux processus ne sont pas totalement indépendants, ils nécessiteront des étapes de communication, de synchronisation et/ou une gestion de la localité de la mémoire cache.

Les processeurs multithreads offrent des solutions plus avancées pour résoudre ces contraintes que les solutions multiprocesseurs et seront détaillées section 2.3.4

[5] Traduction libre de "Dependances Architecture"
[6] Traduction libre de "Independances Architecture"

2.3.1 Architecture séquentielle : Superscalaire - ILP [3, p.123-p.168]

Définition

" Une machine superscalaire se distingue par sa capacité à traiter plusieurs instructions par cycle d'horloge depuis un flux d'instruction linéaire conventionnel " [35] [7].

Il est précisé [3, p.129] que cette définition ne permet pas de distinguer une machine superscalaire d'une machine de type VLIW (voir section 2.3.2). La particularité de l'architecture superscalaire décrite par [3, p.129] est de comporter trois sections distinctes :

- Une section "in-order" pour acheminer et décoder plusieurs instructions par cycle en respectant l'ordre séquentiel et pour renommer les registres associés aux instructions. Si la microarchitecture exécute les instructions dans l'ordre, cette section répartit les instructions parmi les files d'attentes vers les unités fonctionnelles.
- Une section "out-of-order" pour exécuter les instructions. Si la microarchitecture exécute les instructions dans le désordre, cette section répartit les instructions parmi les unités fonctionnelles en respectant les dépendances (voir section 2.5).
- Une section "in-order" pour collecter les résultats, les garder en mémoire si des instructions dans le pipeline en dépendent et pour écrire les résultats dans les registres dans l'ordre.

Description fonctionnelle basée sur [3, p.123-p.168]

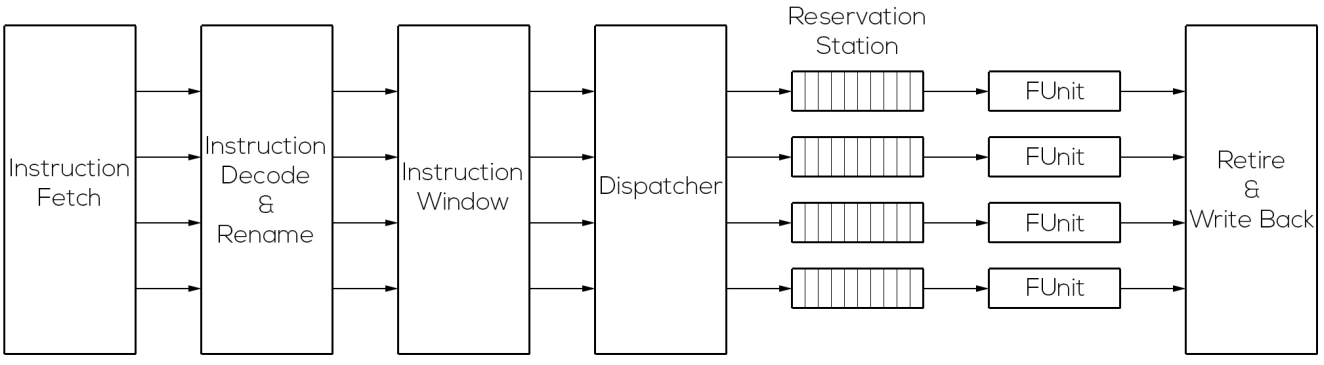

FIGURE 2.2: Pipeline superscalaire basé sur [3, Fig. 4.2 & Fig. 4.18]

[7] Traduction libre

Le premier étage d'un pipeline superscalaire achemine, au minimum et par cycle, un nombre d'instructions équivalent à la bande passante du répartiteur (dispatcher).

Ces instructions sont ensuite décodées. Des registres physiques sont attribués aux registres architecturaux, ceux qui sont référencés dans les instructions. Les instructions ainsi renommées sont envoyées dans "l'instruction window".

A cette étape, les dépendances de contrôle sont résolues par la prédiction de branche (voir section 2.6) et le renommage des registres permet d'éviter les fausses dépendances entre registres.

Le répartiteur [8] achemine les instructions ne présentant pas de dépendances structurelles ou de données vers la station de réservation d'où elles attendent les opérandes manquantes.

Après exécution, les résultats attendent dans l'étage "Retire & Write Back" jusqu'à ce qu'ils puissent être enregistrés. Pour qu'un résultat puisse être enregistré, il faut que :

- l'exécution de l'instruction soit complète.
- l'ensemble des résultats provenant des instructions précédentes soit enregistré ou puisse être enregistré en même temps.
- aucune interruptions n'ait été déclenchée pendant ou avant l'exécution.
- l'instruction ne provienne pas d'une exécution spéculative (voir section 2.6).

Intérêts, contraintes et limites

Le principal intérêt de cette architecture est d'être capable d'exécuter plusieurs instructions par cycles d'un programme séquentiel quelconque. Cette prouesse est réalisée sans l'assistance du compilateur, mais au prix d'un circuit complexe.

Une autre limite de cette microarchitecture, qu'elle partage avec le MIPS (voir section 3.1), est qu'une mauvaise prédiction de branche ou une interruption nécessitent de vider le pipeline et donc d'annuler les opérations en cours. La capacité des microarchitectures superscalaires à gérer des files d'attente et à exécuter plusieurs instructions par cycle aggrave cet impact. Les mécanismes de branch prédiction ont donc un rôle déterminant à jouer pour les performances de ce type de microprocesseur. Cette même limite introduit une incertitude sur le temps d'exécution pouvant impacter les systèmes temps réel.

[8] Généralement appelé Issue dans la littérature anglophone

La microarchitecture proposée chapitre 3 a été conçue dans l'objectif d'exécuter plusieurs instructions par cycles tout en maintenant l'ordre total d'exécution et le temps d'exécution déterministe.

2.3.2 Architecture à indépendances explicites : VLIW - ILP [3, p.203-p.219]

Définition

Un processeur VLIW utilise des instructions longues (128 - 1024 bits / instruction) qui contiennent plusieurs sous instructions indépendantes. Ces instructions sont traitées par le pipeline d'exécution de façon synchrone. Le caractère indépendant des sous instructions constitue la principale différence entre l'architecture VLIW et CISC.

Cette architecture diffère fondamentalement du superscalaire dans la mesure où l'ILP est explicite et que les instructions sont donc exécutées dans l'ordre séquentiel.

Intérêts, contraintes et limites

Le principal intérêt de cette architecture est d'être capable, comme l'architecture superscalaire, d'exécuter plusieurs instructions par cycle. Mais contrairement à l'architecture superscalaire, la gestion de l'ILP est assurée par le compilateur. Cela réduit donc la complexité du processeur tout en augmentant la complexité du compilateur. Le fait que la gestion du parallélisme ne soit pas assurée pendant le temps d'exécution réduit la flexibilité du système face aux événements dynamiques tels que les défauts de cache [3, p.206].

2.3.3 Microarchitecture multiprocesseurs on-chip - TLP

Définition

Microarchitecture comportant plusieurs processeurs sur une même puce et comportant un circuit visant à arbitrer l'accès à la mémoire cache partagée, aux entrées/sorties et aux périphériques.

Il existe de nombreuses façons d'organiser plusieurs processeurs sur une même puce. Historiquement, sur base de [3, p.248-256], trois grandes catégories d'organisation de la cache pour les multiprocesseurs ont été développés :

- **Multiprocesseur symétrique (SMP)** : Ce type d'organisation est basé sur un espace mémoire partagé et commun. Il repose sur un système d'accès uniforme à la mémoire (UMA). Le partage de l'espace mémoire peut se faire à tous les niveaux de cache. Certains niveaux de mémoire cache peuvent donc, ou non, être dédiés aux processeurs.
- **Multiprocesseur à mémoire partagée distribuée (DSM)** : Le système conserve un système d'adressage unique mais les mémoires caches sont réparties parmi les processeurs. Un processeur peut accéder à la mémoire cache d'un autre. Il s'agit d'un système d'accès à la mémoire non uniforme (NUMA).
- **Multiprocesseur sans mémoire partagée** : Chaque processeur possède son propre espace mémoire. Les communications ne se font pas par mémoire partagée mais sont transmises par "message-passing" d'un processeur à l'autre.

Concernant l'organisation des processeurs pour exploiter le TLP, les recherches récentes portent sur les réseaux sur puce (NoC) [36] et leur organisation sous diverses topologies.

- **Homogènes** : répliquant les mêmes unités de calculs.
- **Hétérogènes** : utilisant diverses unités de calcul spécialisées.

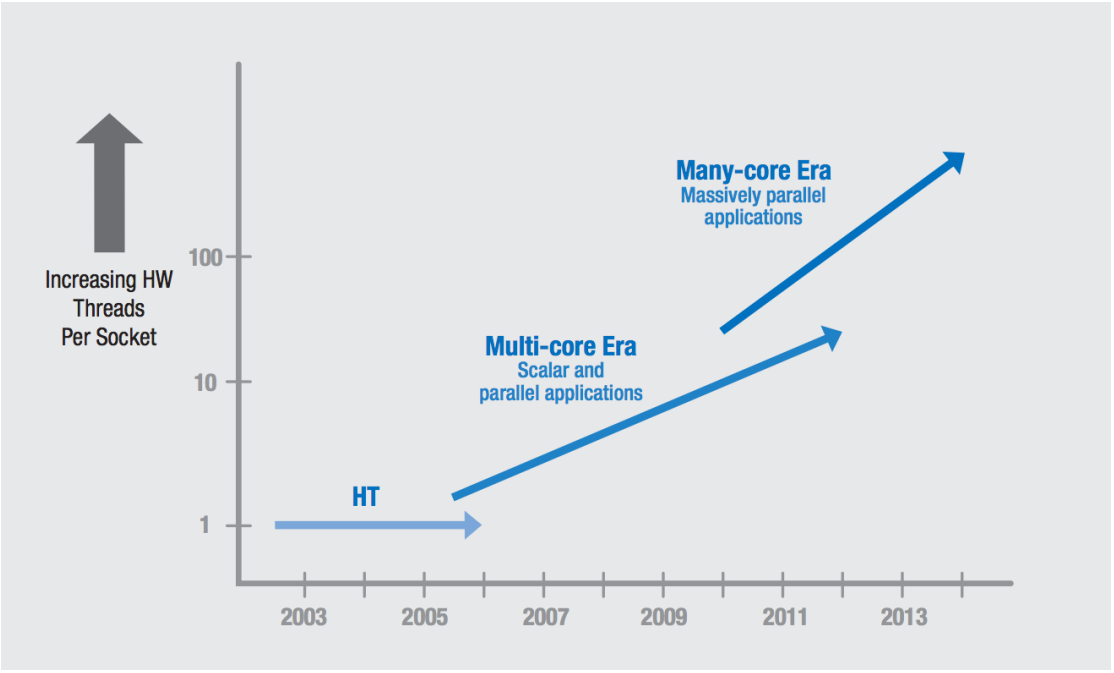

FIGURE 2.3: Driving increasing degrees of parallelism on Intel® processor architectures [12, Fig. 2]

Comme illustré (Fig. 2.1) et confirmé (Fig. 2.3 [12]) depuis le Pentium 4, Intel concentre son effort sur le développement de puces multiprocesseur. Les premières étapes d'Intel en matière d'architecture CMP ont d'abord été dirigées vers une organisation SMP.

Dans sa roadmap présentée en 2015 [12], la compagnie Intel propose de développer des réseaux reconfigurables hétérogènes sur puce et s'intéresse donc au développement des NoC.

Intérêts, contraintes et limites

Les architectures multiprocesseurs permettent d'augmenter le débit d'instruction en exploitant un niveau de parallélisme plus grossier (TLP).

Du point de vue de l'application, plus les threads et processus ont besoin de partager des données, plus les performances sont dépendantes de l'organisation des processeurs, de la mémoire cache et des méthodes de communication inter-processus et inter-processeurs [3, p.250].

2.3.4 Microarchitecture multi-threads - TLP

Définition

Architecture permettant de partager un pipeline et une ou plusieurs unité(s) fonctionnelle(s) entre plusieurs processus et threads ayant leur propre compteur de programme (PC) et leurs propres registres.

Le principal objectif de ce type d'architectures est de masquer au moins en partie les différents temps de latences existant dans une exécution :

- lecture des entrées (input) ou écriture synchrone (output).
- défauts de cache (cache miss).
- dépendances de données : principalement la lecture après écriture (RAW) dans le cadre d'un accès à la cache L1.
- dépendances de contrôle en cas d'absence de prédiction de branche.
- Calculs par des unités fonctionnelles multi-cycle.

Un exemple du coût qu'un défaut de cache peut engendrer a été mesuré sur un multiprocesseur symétrique (l'Alpha Server 4100 SMP) par [37] et repris par [3, p.257]:

- 7 cycles pour un défaut de cache L1.
- 21 cycles pour un défaut de cache L2.
- 80 cycles pour un défaut de cache L3 (accès mémoire principale).
- 125 cycles pour un défaut de cache nécessitant l'accès à la mémoire cache d'un autre processeur.

Ces mesures sont données à titre informatif pour illustrer l'impact que peut avoir un défaut de cache et donc l'intérêt de pouvoir le masquer par l'exécution d'une autre tâche.

Trois approches sont proposée par [3, p.259] :

- **le multithreading simultané (SMT)** : La forme la plus avancée de multithreading parmi ces approches. Elle sera discutée section 2.4.
- **Cycle-by-Cycle interleaving ou barrel-shift pipeline** : Le changement de tâches et donc de contexte se produit à chaque cycle. Le principal intérêt est de simplifier le circuit d'ordonnancement. Un simple compteur suffit pour sélectionner une tâche et il n'est pas nécessaire de décoder l'instruction pour vérifier un changement de contexte. Certaines variantes de cette approche permettent un "burst-mode" rendant l'exécution de plusieurs instructions provenant d'un même contexte possible. Le compilateur doit expliciter les dépendances pour rendre ce mode de fonctionnement possible.
- **Traitement par bloc (Block interleaving technique)** : Le changement de contexte est réalisé quand un événement introduisant une latence ou forçant le changement de contexte est détecté. Ces événements peuvent être de plusieurs natures [3, p.270-271] :
  - **Statique** :
    - **Explicite** : Une instruction provoquant le changement de contexte est ajoutée au set d'instructions.
    - **Implicite** : Certains types d'instruction provoquent un changement de contexte comme une instruction de branche conditionnelle.
  - **Dynamique** : Un événement dynamique tels qu'une interruption ou un défaut de cache provoquent le changement de contexte, induisant potentiellement la non validité des opérations entre l'étage de "fetching" et l'étage provoquant l'événement.

Intérêts, contraintes et limites

Le principal intérêt des architectures multithreadées est d'éviter au processeur de tourner à vide quand certaines instructions ou certains événements provoquent une latence dans l'exécution. Cela permet donc de maximiser le débit d'instruction du système au détriment du débit par thread.

L'approche "Cycle-by-cycle interleaving" contraint les threads à se partager équitablement le pipeline (un étage par thread) et donc diminue la vitesse d'exécution par thread en conséquence. Il faut donc au minimum un nombre de threads équivalent au nombre d'étages du pipeline pour profiter pleinement de cette approche.

L'approche "Traitement par bloc" offre de meilleures performances par thread mais induit potentiellement des overheads liés aux changements de contexte.

Un autre prix à payer est qu'il faut partager la mémoire cache entre les threads.

2.4 Solutions hybrides

Toutes ces possibilités architecturales visant à améliorer l'ILP et le TLP présentent des avantages et inconvénients. Bien qu'il soit intéressant de les comparer en ces termes, ces avancées ne sont que rarement mutuellement exclusives. La plupart de ces possibilités peuvent être combinées. Dans le cas le plus pessimiste, combiner deux solutions permettra d'additionner leur bénéfice. Par exemple, combiner une architecture multi-threadée avec une architecture superscalaire, permettra d'une part de masquer les latences tout en augmentant le débit d'instructions par thread. Ces différentes combinaisons sont illustrées Fig. 2.4.

Certaines combinaisons peuvent même dépasser la somme de leurs parties. C'est par exemple le cas de l'architecture multithread simultané (Simultaneous Multithread - SMT). Cette microarchitecture est détaillée section 2.4.

Cette tendance à combiner différentes solutions microarchitecturales est confirmée par l'évolution des microarchitectures Intel (voir Fig. 2.5).

Simultaneous Multithread - SMT

Cette microarchitecture combine le multithreading avec la technologie superscalaire comme l'exemple proposé en début de section. En plus de profiter des intérêts de ces deux avancées, ce type d'architecture exploite le multi-threading pour combler les limites

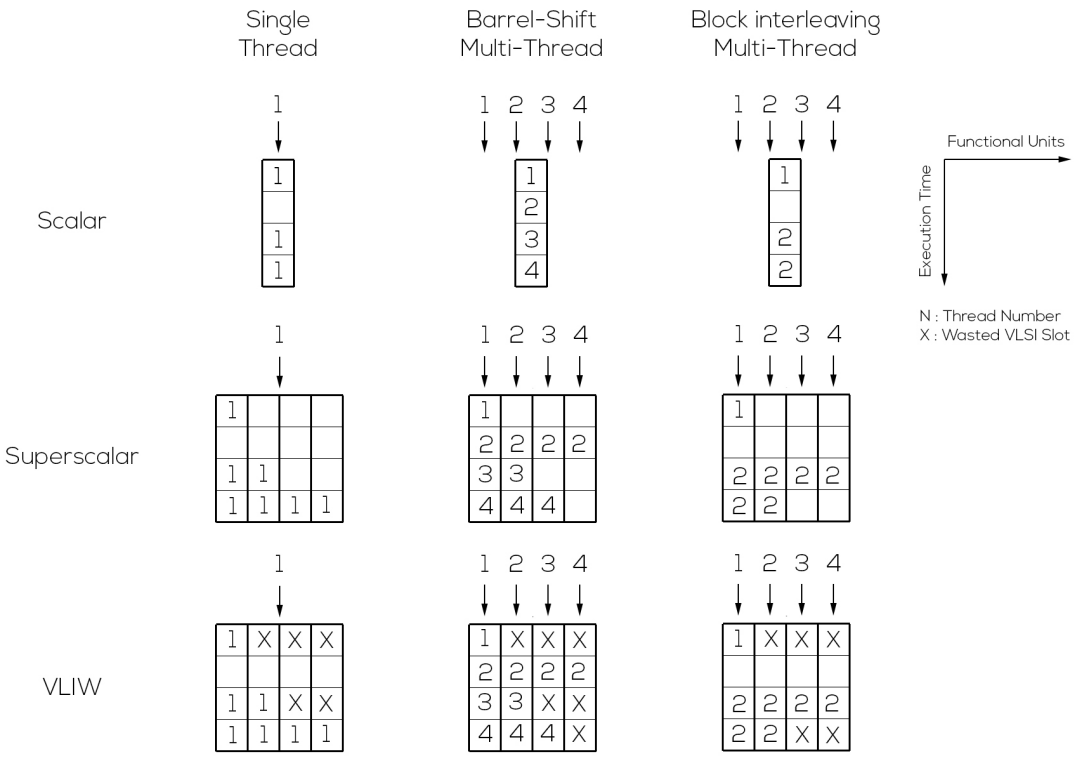

FIGURE 2.4: Différentes solutions hybride ILP/TLP [3, Fig. 6.6 - 6.7]

de l'ILP exploitable par une architecture superscalaire. En plus de tenter de maximiser le nombre d'instructions par fil d'exécution, les unités fonctionnelles non utilisées par ce fil sont remplies par les autres threads (voir fig. 2.6).

2.5 Gestion des contraintes d'exécution (Hazards)

Lors de l'exécution d'instructions dans un processeur, certaines contraintes de dépendances liées à une exécution peuvent, si elles ne sont pas gérées correctement, avoir une incidence sur la conformité de cette exécution. Bien que ces contraintes existent en tant que telle d'un point de vue théorique, certaines micro-architectures, de par leur conception, n'y sont pas sujettes. Ces contraintes existent principalement pour les micro-architectures pipelinées et pour celles exploitant l'instruction level parallelism (voir section 2.3).

Ces dépendances peuvent être de trois types [3, p.22-23] :

- **Contraintes sur les données** : Il s'agit de contraintes de dépendances entre les données lors d'une exécution. Elle peuvent être de trois types :

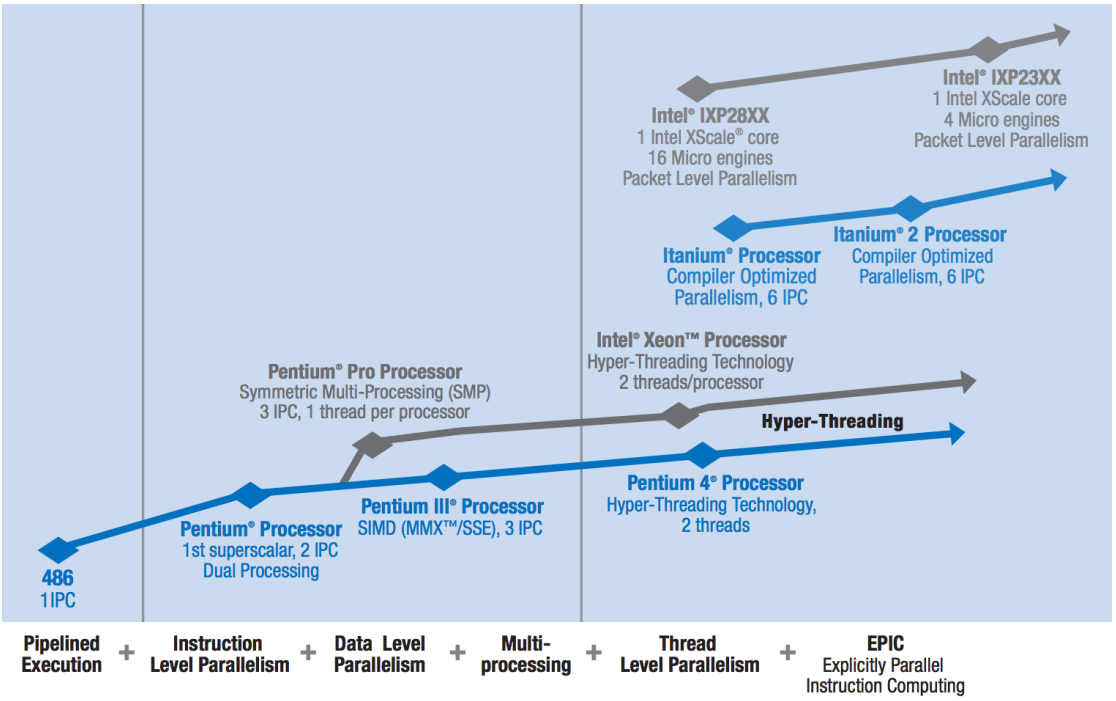

FIGURE 2.5: Driving increasing degrees of parallelism on Intel® processor architectures [12, Fig. 2]

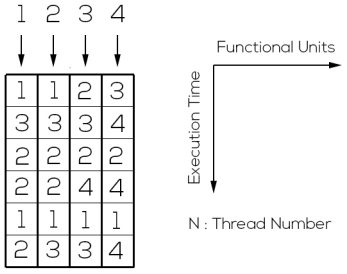

FIGURE 2.6: Simultaneous Multithreading [3, Fig. 6.8]

- **Lecture après écriture (RAW)** : Concerne les micro-architectures pipelinées et / ou gérant l'exécution dans le désordre. Si une instruction nécessite le résultat d'une autre instruction, mais que cette donnée n'est pas encore disponible, cette donnée doit être acheminée ou alors l'exécution doit attendre ce résultat.
- **Écriture après lecture (WAR)** : Concerne les micro-architectures avec exécution dans le désordre et / ou avec un étage d'écriture précédent l'étage de lecture. Se présente si un registre a été accédé par une instruction alors qu'il contenait une valeur obsolète. Dans ce cas il faut annuler l'exécution ou acheminer le résultat.
- **Écriture après écriture (WAW)** : Concerne les micro-architectures avec

multiples écritures par cycle, principalement dans le cas d'architectures superscalaire. Si une instruction veut écrire dans un registre dans lequel a déjà été écrite une valeur, l'écriture par lot de l'étage "Retire & Write Back" d'un processeur superscalaire ou dérivé doit attendre l'écriture réalisée par cette dernière instruction.

- **Contraintes structurelles** : Se présentent si plusieurs instructions doivent accéder à la même ressource, principalement dans le cas d'ILP et de TLP. Ce type de contraintes nécessite un arbitrage d'accès et, dans certains cas, un système d'arrêt du pipeline.

- **Contraintes de contrôle** : Se présentent en cas d'instructions de contrôle du flux d'exécution tels que les jump ou branchement conditionnels. Ces instructions peuvent potentiellement nécessiter l'annulation d'une partie de l'exécution. Les politiques de gestion de ce type de contraintes seront détaillées section 2.6.

2.6 Gestions des branches d'exécution et exécution spéculative

Les contraintes de contrôle peuvent se résoudre de plusieurs façons. Certaines solutions se limitent à éviter une exécution erronée. D'autres essaient de récupérer le temps de calcul perdu. Un troisième type de solutions propose d'anticiper le résultat de branchements afin d'accélérer l'exécution en cas de prévision correcte. Ce dernier type de solution est appelé exécution spéculative.

Voici une liste de solutions possibles :

Solutions Software [3, p.29]

- **Technique de branchements et sauts retardés** : Le compilateur prévoit de placer des instructions n'interférant pas avec l'instruction de contrôle après celle-ci en un nombre équivalent au nombre d'étages du pipeline précédant l'étage assurant le branchement ou le saut. S'il n'y a pas assez d'instructions respectant cette contrainte, des instructions "no-op" sont ajoutées. Cette solution a l'avantage de simplifier les micro-architectures de type scalaire qui ne doivent plus gérer cette contrainte. En revanche, cette solution complexifie les micro-architectures de type superscalaire. Ceci est dû à la gestion d'exécution de plusieurs instructions provenant d'un même flux. Cette solution peut coexister avec d'autres en enrichissant le jeu d'instructions.

- **Prédiction de branche statique** : Le compilateur indique dans l'instruction s'il faut exécuter la branche. De cette façon, le processeur l'exécute spéculativement. En cas d'erreur, les instructions chargées doivent être annulées.

Solutions Hardware[3, p.30, p.133]

- **Blocage de l'exécution (Interlocking)** : En cas de branche conditionnelle ou de saut, le pipeline est bloqué jusqu'à ce que l'adresse du saut ou le résultat de la branche conditionnelle soit connue. Ce blocage provoque un certain nombre de "bulles" dans le pipeline en fonction de la micro-architecture.

- **Branchement jamais pris** : Le processeur exécute spéculativement la suite du programme en prédisant que les branches ne sont jamais prises. Il s'agit de la prédiction la plus simple car la micro-architecture doit uniquement être capable d'annuler l'exécution en cas de mauvaise prédiction. Cette prédiction obtient de mauvaises performances pour les boucles.

- **Branchement toujours pris** : Le processeur exécute spéculativement le branchement en prédisant que les branches sont toujours prises. Cette prédiction obtient de bonnes performances pour les boucles.

- **Branchement de retour en arrière toujours pris** : Quand le branchement pointe vers une instruction précédente, il s'agit généralement d'une boucle et donc on exécute spéculativement cette branche. Sinon, on exécute spéculativement la suite du programme.

- **Prédiction de branche dynamique** : Un circuit s'occupe de prédire si la branche doit être prise en se basant sur l'historique des prédictions. Une grande variété de circuits de ce type ont été développés mais ne seront pas présentés dans ce travail.

- **Micro-architectures sans contraintes** : Certaines micro-architectures ne sont tout simplement pas sujettes aux "hazards". C'est, par exemple, le cas des architectures multi-cycles. Ce dernier type de micro-architecture ne présente pas de bonnes performances. Sa fréquence de fonctionnement est équivalente aux micro-architectures pipelinées sans offrir le même taux d'instructions par cycle.

- **Microarchitectures multithreadées** : Les micro-architectures multi-threadées (voir section 2.3.4) permettent aussi d'éviter ces contraintes tout en maintenant un IPC équivalent aux solutions pipelinées. Elles y parviennent en exécutant d'autres threads aux moments où des contraintes pourraient apparaître.

2.7 Micro-architectures et temps réel strict

Il existe deux types de processus pouvant nécessiter des contraintes temps réel strictes [38] :

- **Processus périodiques** : Processus devant être exécutés à une fréquence donnée et devant respecter un délai de calcul précis.
- **Processus asynchrones ou interactifs** : Processus déclenchés par un événement externe et ayant une contrainte en terme de temps de réponse.

Les contraintes temps réel strictes impliquent que le temps de calcul et le temps de réponse soient bornés de façon précise et toujours respectée. Pour y parvenir, une analyse statique et / ou dynamique est réalisée afin de déterminer le temps de calcul au pire cas.

Certaines techniques micro-architecturales proposées dans ce chapitre complexifient l'analyse du temps d'exécution au pire cas. Les techniques visant à augmenter l'ILP ou à accélérer le temps de calcul via une exécution spéculative peuvent potentiellement induire une incertitude sur le temps de calcul.

Malgré cela, ces techniques peuvent améliorer significativement les performances des processeurs et offrir la possibilité de résoudre de nouvelles classes de problèmes. Il est précisé par Jonathan Barre, Christine Rochange et Pascal Sainrat dans leur travail sur l'architecture SMT[39] que cette évolution en terme de performances se justifie pour les applications temps réel strict principalement dans le domaine automobile et de l'aviation.

La plupart des formes d'exécutions spéculatives induisent une incertitude sur le temps de calcul du fait que le risque de prédiction fausse soit non nul. Il faut alors considérer le pire cas comme étant une prédiction systématiquement fausse.

La seule solution pour réaliser une exécution spéculative n'induisant pas cette incertitude est d'exécuter les deux branches possibles en parallèle. Cela implique que dans tous les cas, une des deux exécutions spéculative soit fausse et donc qu'elle ait gaspillé les ressources du processeur.

Certaines formes d'ILP, en particulier celles utilisant une exécution de type "Out-of-Order" complexifient la prédictibilité des temps de calcul. Une solution est apportée par Jonathan Barre & Al.[39] pour le cas du SMT. La simulation de micro-architecture SMT qu'ils proposent dédie l'accès aux ressources du processeur à un thread et alloue les ressources non utilisées aux threads moins critiques.

Une approche différente est proposée par ce travail et détaillée au chapitre 3.

2.8 Conclusion

Comme vu section 1.2.3, le niveau de parallélisme atteignable pour une exécution donnée n'est pas forcément infini. Seuls certains problèmes dits "Embarrassingly Parallel" [40] ont un temps d'exécution directement inversement proportionnel à la bande passante d'un processeur exploitant l'ILP ou le TLP. Ces problèmes peuvent profiter de machines dédiées au calcul parallèle et surpassent les processeurs "general-purpose" dans ce type de domaine. Pour les problèmes intrinsèquement séquentiels, l'exécution spéculative et les solutions pipelinées semblent particulièrement adaptées. L'amélioration des performances peut se faire en augmentant la fréquence de fonctionnement et en "pipelinant" des unités fonctionnelles plus complexes. Pour les applications de type "general-purpose", un compromis doit être trouvé entre la vitesse d'exécution séquentielle et la capacité des circuits à gérer l'ILP et le TLP.

L'ensemble des technique microarchitecturales détaillées ici présente un intérêt plus ou moins grand suivant le domaine d'application. Encore bien d'autres solutions peuvent être envisagées et leur combinaison peut donner lieu à de nouvelles perspectives comme pour le SMT. Avec l'évolution de la finesse de gravure permettant de concevoir des circuits toujours plus grands, moins chers et plus économes en énergie, l'espace des possibilités de design devient de plus en plus large et son exploration de plus en plus complexe.

Chapitre 3

Implémentation d'une microarchitecture SMT avec ordonnancement hardware

Ce travail tente de répondre, en partie, à deux problématiques des architectures de processeurs : l'ordonnancement des tâches, en particulier dans le cadre de contraintes temps-réel fortes, et la gestion du multiprocessing[1].

Ces deux problématiques sont intimement liées dans la mesure où il incombe à l'ordonnanceur de gérer les contraintes multi-tâches, principalement pour la synchronisation des processus et en particulier des threads.

Comme discuté dans les chapitres précédents, la prise en charge de systèmes multi-tâches, en particulier multi-threads et multi-processus engendre un ensemble de contraintes pouvant avoir un impact sur les performances d'un système :

- Le changement de contexte
- Le calcul de l'ordonnancement
- La synchronisation des processus et des threads

La gestion temps réel stricte implique aussi un ensemble de contraintes :

- Le calcul de deadlines
- Respect des deadlines

[1] Le multi-threading pouvant être considéré comme un cas particulier du multiprocessing

- Temps d'exécution déterministe
- Temps de réponse court

Ces contraintes sont généralement gérées par le software et impliquent donc un surcoût en temps de calcul. Ce surcoût peut avoir un impact sur l'évolutivité du système. Par exemple, plus il y a de threads à exécuter plus les mécanismes de synchronisations auront un impact sur le temps de calcul et plus le multi-threading sera inefficace.

Certaines de ces contraintes ont une implication directe sur la micro architecture : typiquement, un temps d'exécution strictement déterministe implique une exécution dans l'ordre incompatible avec les technologies de type "Out-of-Order Execution" dynamique (non compilé).

Sur base de ces contraintes et des conclusions du chapitre 1, ce travail s'est orienté vers la conception d'un ordonnanceur à court terme hardware local et strictement temps réel ainsi que l'adaptation d'une microarchitecture MIPS vers une microarchitecture multi-threadée à deux accès.

Comme discuté dans le chapitre 1, le calcul des deadlines ou de l'ordonnancement d'un système complet est de la responsabilité d'un ordonnanceur global ne faisant pas l'objet de ce travail. L'ordonnanceur local s'occupe, quant à lui, d'offrir une résolution d'ordonnancement de l'ordre du cycle d'horloge, tout en offrant un ensemble de mécanismes permettant un réglage fin de l'exécution depuis l'extérieur, à savoir, par un ordonnanceur global (qu'il soit hardware ou software).

En travaillant à l'échelle hardware, il est possible d'aller encore plus loin dans la notion de temps réel en ajoutant la notion de temps exact d'exécution. Le fait qu'un programme puisse s'exécuter en un temps parfaitement déterministe en terme de nombre de cycles d'exécution et qu'il soit possible d'anticiper quelle instruction est exécutée à quel moment permet de travailler à des échelles de temps plus petites et avec la précision de l'horloge utilisée par la processeur.

Par exemple, un programme souhaitant effectuer de la "Power Pulse Modulation" pourrait obtenir une résolution d'un ordre de grandeur de l'ordre de la période de l'horloge du processeur.

La microarchitecture qui fait l'objet de ce travail a été développée de façon à permettre la résolution de ces contraintes tout en cachant les différents overheads liés à celles-ci. Elle se base sur la microarchitecture MIPS pour simplifier le design afin de valider le concept.

Dans ce chapitre seront présentés le processeur MIPS et les modifications apportées à cette microarchitecture. Seront ensuite passés en revue le mode opératoire et les outils développés pour contrôler le processeur et l'ordonnanceur. Pour conclure, les résultats et une comparaison avec différents travaux seront détaillés.

3.1 Le MIPS

Cette section détaille le fonctionnement d'un MIPS. Cette micro-architecture est expliquée de façon plus approfondie car elle constitue la base micro-architecturale du processeur développé et détaillé dans ce chapitre.

L'architecture MIPS (Microprocessor without Interlocked Pipe Stage) est de type RISC (voir section 2.2). Elle a été développée à l'Université de Stanford [41].

Elle diffère, selon ses auteurs, par le fait qu'elle minimise la complexité de l'hardware, contrairement à l'architecture RISC de Berkeley[42] qui minimise à la fois la complexité de l'hardware et du set d'instructions. L'architecture exploite le codage des instructions pour permettre la parallélisation de celles-ci. Ses performances reposent d'avantage sur le compilateur.

Micro-architecture générale

Le MIPS est une microarchitecture pipelinée, c'est-à-dire que l'exécution d'une instruction est divisée en plusieurs étages de façon à diminuer le délai des circuits combinatoires situés entre deux registres. Chacun de ces étages peut être occupé par une instruction, ce qui le différencie des micro-architectures de type multi-cycles.

Voici les différents stades de l'exécution d'une instruction à travers le pipeline :

- Acheminement d'une instruction depuis la mémoire.
- Décodage de l'instruction afin de récupérer les données depuis les registres et de savoir comment la traiter.
- Exécution de la fonctionnalité à travers l'unité logique d'arithmétique (ALU).
- Enregistrement ou lecture d'une donnée depuis la mémoire principale via la mémoire cache.
- Enregistrement du résultat ou de la donnée lue depuis la mémoire dans un registre.

Une représentation d'une implémentation[43] de ce circuit est illustré Fig. 3.1.

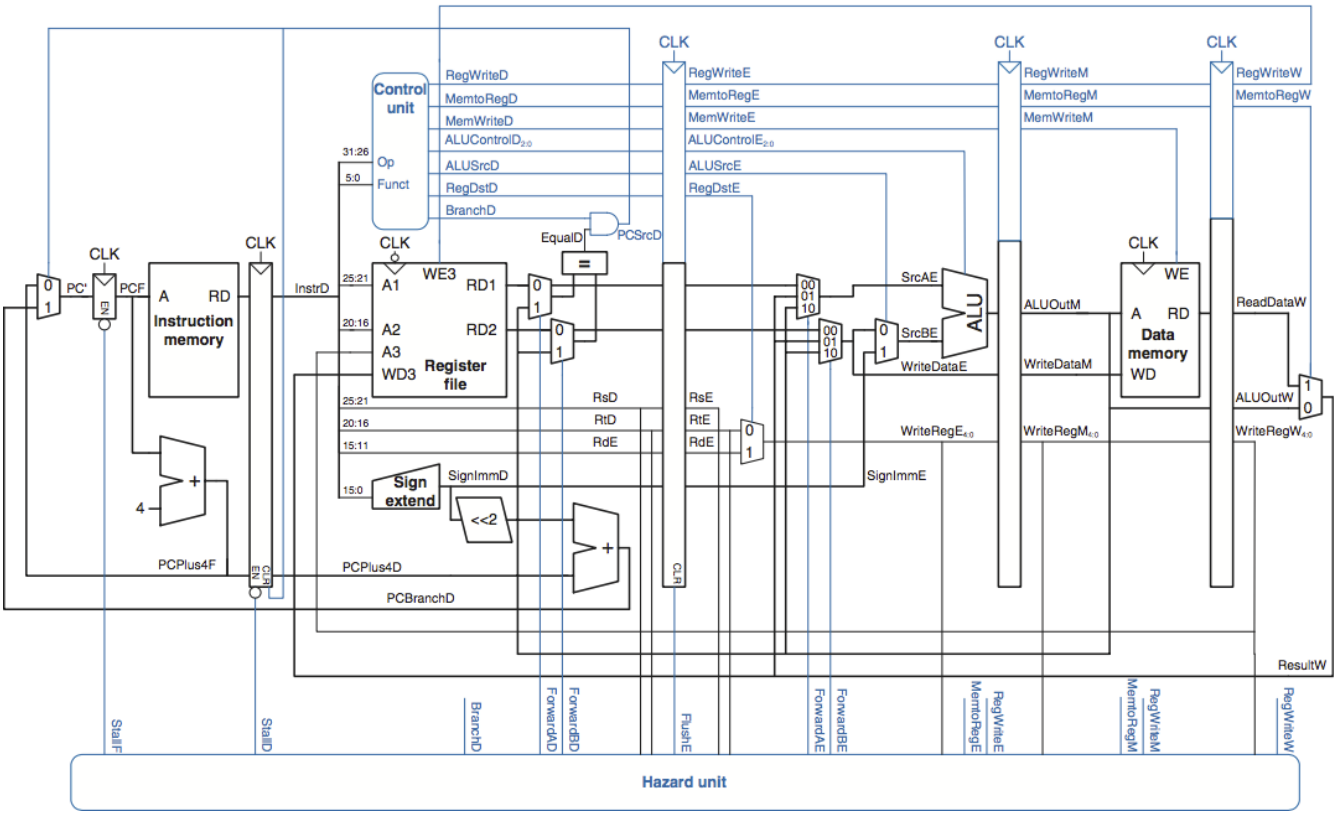

FIGURE 3.1: Pipeline MIPS [43]

Acheminement des instructions (IF)

Les instructions sont stockées dans une mémoire cache d'instructions (L1 I-Cache). Elles sont récupérées par adressage direct. Le MIPS possède un registre appelé Program Counter (PC) retenant l'adresse de la prochaine instruction à acheminer dans le pipeline. Ce registre est incrémenté de 4[2] à chaque cycle.

Le Program Counter peut aussi recevoir une adresse précise depuis l'étage ID si une instruction de type "jump" ou de branchement conditionnel est décodée.

L'acheminement des instructions peut être bloqué par un signal "Stall" indiquant que le pipeline doit être stoppé pour résoudre un problème de dépendances (Hazards).

Décodage de l'instruction (ID)

L'instruction acheminée est ensuite décodée. Les bits de contrôle de l'instruction sont envoyés dans l'unité de contrôle (Control Unit) afin de déterminer la structure de l'instruction et d'en décoder les signaux permettant de contrôler l'exécution. Les opérandes

[2] L'incrément est de 4 car il s'agit d'un mode d'adressage de mots de 8 bits.

sont récupérées depuis le "Register File" grâce aux adresses directement codées dans l'instruction.

S'il s'agit d'une instruction de type "immediate[3]", la valeur immédiate est envoyée dans un circuit appelé "Sign Extender" permettant de convertir un mot signé de 16 bits vers un mot signé de 32 bits.

Si la valeur immédiate correspond à l'adresse d'une instruction pour un jump ou une branche conditionnelle, celle-ci est multipliée par 4 pour adresser la mémoire d'instructions. La vérification des branchements conditionnels se fait à cet étage afin de limiter l'apparition de "bulles" dans le pipeline. De cette façon, si le branchement est pris, seule la dernière instruction chargée n'est pas correcte.

Exécution (EX)

L'étage d'exécution s'occupe de réaliser l'opération sur les opérandes. Cette opération est réalisée par l'ALU (Arithmetic Logical Unit). L'unité de contrôle ayant au préalable décodé la fonction arithmétique à exécuter, les signaux de contrôle sont acheminés vers l'ALU. Ces signaux s'occupent aussi de contrôler les multiplexeurs afin d'acheminer correctement les opérandes. Le résultat est ensuite transmis à l'étage suivant.

Accès Mémoire (MEM)

En cas de lecture ou d'enregistrement (load & Store), le résultat calculé par l'ALU correspond à une adresse calculée. S'il s'agit d'un enregistrement, l'une des deux valeurs retournées par le "Register File" est inscrite dans la mémoire. En cas de lecture, la donnée récupérée depuis la mémoire cache (L1 D-Cache) est envoyée à l'étage suivant.

Cet étage s'occupe aussi de transmettre le résultat de l'ALU vers les étages précédents si une dépendance de donnée de type "Read after Write" est détectée (voir section 2.5).

Retour Résultat (WB)

Le résultat calculé par l'ALU ou le mot lu depuis la mémoire de données est enregistré dans le "Register File".

[3]Les instructions de type "immediate" sont des instructions utilisant un registre pour le premier opérande et un mot directement codé dans l'instruction pour le second opérande

3.2 Microarchitecture MIPS modifiée

Les modifications apportées à la microarchitecture ont été réalisées sur base d'une série de contraintes :

- Minimisation de l'overhead et du jitter liés à l'ordonnancement
- Maintien de la flexibilité d'ordonnancement
- Minimisation de l'overhead lié au changement de contexte.
- Exécution dans l'ordre total.
- Prédictibilité du temps d'exécution.

Ces contraintes ont été choisies du fait que ce travail explore les techniques d'ordonnancement d'instructions et de processus dans le cadre d'applications multithreads orientées temps réel.

Comme montré section 1.2.3, toute forme d'overhead dans le cadre du multithreading et du multiprocessing entraîne une limite à leur bénéfice. Ceci justifie le soucis d'optimisation de l'ordonnancement et du changement de contexte.

Le principal inconvénient lié à l'ordonnancement hardware (voir chapitre 1) est le manque de flexibilité d'ordonnancement. Le fait de travailler sur FPGA (Field-Programmable Gate Array) [22] résout ce problème. Mais une attention, bien que limitée, a néanmoins été apportée à la flexibilité de l'ordonnanceur développé pour que le système reste valide d'une application à une autre.

Le respect de contraintes temps réel strictes justifie de porter une attention particulière à la prédictibilité du temps d'exécution. L'exécution dans l'ordre total n'est pas nécessaire [39] mais apporte un confort d'utilisation non négligeable pour les applications embarquées temps réel.

Étant donné que l'utilisation d'un ordonnanceur hardware a un impact non négligeable sur la taille du circuit, une des lignes directrices de ce projet a été, comme pour le MIPS[41], d'éviter d'implémenter tout circuit pouvant être remplacé par le compilateur.

L'implémentation de cette microarchitecture est basée sur le MIPS par soucis de simplicité. Ce choix sera critiqué dans la conclusion de ce chapitre.

Sur base des contraintes décrites ci-dessus et des solutions détaillées chapitre 2, le choix s'est porté sur une microarchitecture multithread de type "block interleaving" pouvant être qualifiée de "Simultaneous Multithread" dans la mesure où les threads se partagent

les deux voies d'exécutions. Elle diffère néanmoins d'une architecture purement SMT du fait qu'un processus ne puisse occuper qu'un étage du pipeline à la fois et n'est donc pas superscalaire. Cette limite est nécessaire pour garantir un temps d'exécution exact et l'ordre total d'exécution et simplifie considérablement le circuit.

La gestion multithread de type "block interleaving" (voir section 2.3.4) est légèrement différente des solutions habituellement proposées dans la mesure où elle est assurée par l'ordonnanceur qui offre une gestion des priorités et un calibrage fin du partage de temps de calcul.

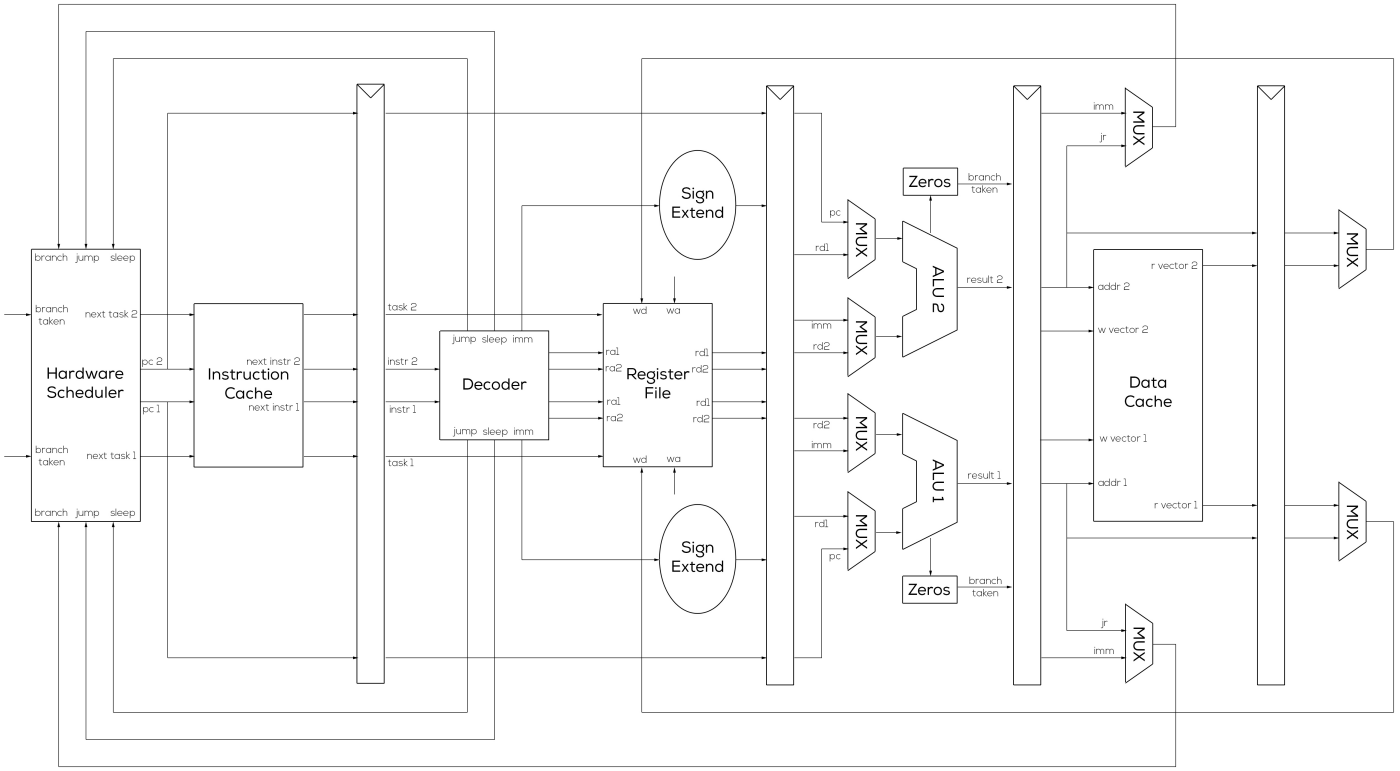

FIGURE 3.2: Microarchitecture MIPS SMT

Cette microarchitecture illustrée Fig. 3.2 est donc fortement inspirée du MIPS mais en diffère par le fait qu'elle comprend :

- Deux "Datapath" étroitement liés.
- Un ordonnanceur multithread gérant jusqu'à 8 processus.
- Un décodeur plus complexe gérant d'une autre façon les "hazards" (voir section 3.2.3).
- Un register file multithread et un accès vers l'extérieur (au-delà de la cache L1).

Son fonctionnement est similaire au MIPS. L'ordonnanceur détermine, à chaque cycle, quels sont les deux processus à exécuter et envoie leur adresse dans le pipeline. Ces adresses sont utilisées pour récupérer les adresses d'instructions et de données propres aux processus (PC - Program Counter). A ce moment, les deux PC, pc1 et pc2 sur le schéma Fig. 3.2 sont utilisés pour "fetcher" deux instructions et sont envoyés vers l'étage suivant. Ensuite, les instructions sont décodées. Une vérification supplémentaire est réalisée afin d'assurer le changement de contexte de façon statique implicite (voir section 2.3.4). Cette vérification peut aboutir sur un changement de contexte forcé au niveau de l'étage de "fetching" en cas de saut, de branche conditionnelle ou d'accès à la mémoire cache L1. Le reste de l'exécution est tout à fait semblable au MIPS excepté que ce processeur traite deux instructions provenant de deux tâches différentes en parallèle.

Le fonctionnement détaillé des différentes parties du circuit sera expliqué dans les sections qui suivent.

3.2.1 Ordonnanceur hardware

L'ordonnanceur réalisé permet de sélectionner 2 processus prêts à être exécutés parmi 8 à chaque cycle. Cette sélection est basée sur une priorité fixe et/ou dynamique ainsi que sur un temps d'attente (sleep). Ce circuit est illustré Fig. 3.3.

Fonctionnement du circuit

Les registres "Task Addr" représentés sur le schéma contiennent l'adresse locale des processus (de 0 à 7). Ils sont initialisés par ordre d'adresse : 0 et 1 sont les tâches les plus prioritaires, 6 et 7 sont les tâches les moins prioritaires. Ce choix d'initialisation repose sur le fait que ces adresses locales sont attribuées aux processus par ordre d'arrivée.

Ces registres sont organisés deux à deux de façon à ce que la distance entre le processus le moins prioritaire et l'exécution soit de 4 étages. A chaque étage, les deux processus sont discriminés par ordre de priorité. De cette façon, il est possible de gérer un changement de contexte pour un seul flux d'exécution.

Un circuit combinatoire non représenté sur le schéma permet de déterminer 2 types de déplacements des processus dans la file d'attente en contrôlant les multiplexeurs :

- **Changement de contexte pour les deux flux d'exécution** : Se produit si le processus ayant la plus petite priorité entre les deux processus en fin de file d'attente est prioritaire par rapport au processus ayant la plus grande priorité

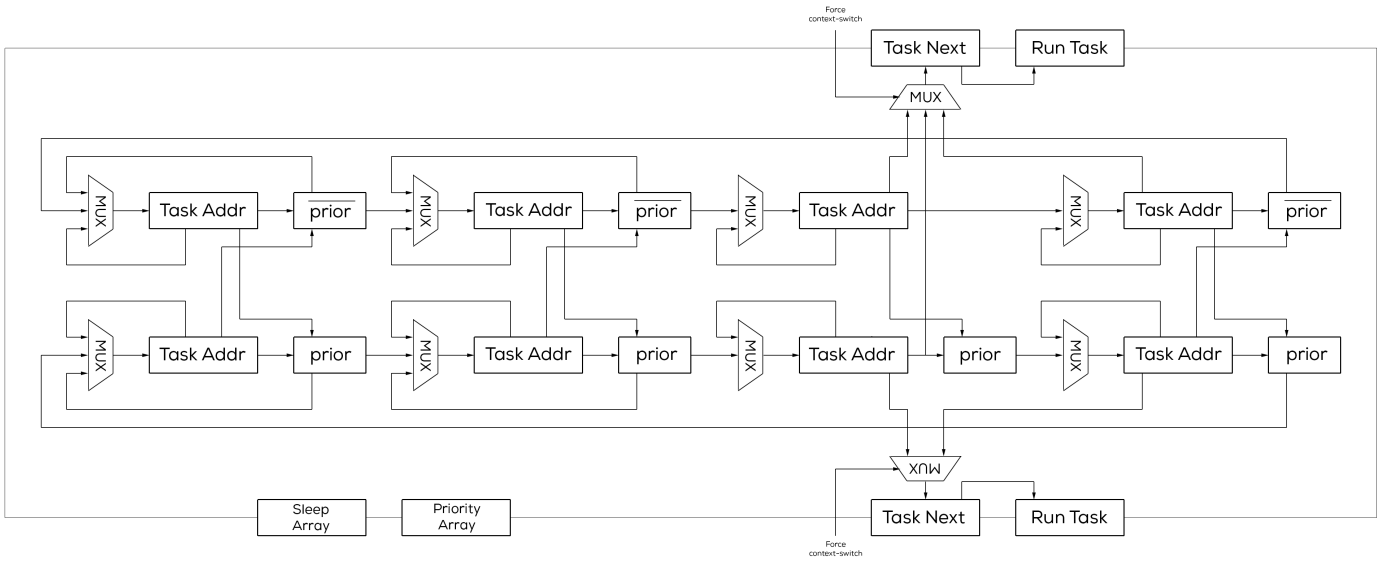

FIGURE 3.3: Ordonnanceur hardware

parmi les deux processus en exécution. Une rotation est donc réalisée : chaque processus passe à l'étage suivant. Les processus en cours d'exécution sont envoyés en début de file d'attente.

- **Changement de contexte pour un flux d'exécution** : Se produit si le processus ayant la plus grande priorité entre les deux processus en fin de file d'attente est prioritaire par rapport aux processus ayant la plus petite priorité parmi les deux processus en exécution. A chaque étage, le processus le plus prioritaire passe à l'étage suivant. Le processus actif sortant est envoyé en début de file d'attente.
- **Pas de changement de contexte** : Les processus actifs continuent leur exécution.

Dans tous les cas (à chaque cycle), une réorganisation interne est réalisée pour les 3 premiers étages afin de garantir qu'une tâche prioritaire progresse d'au moins un étage par cycle. Ces déplacements ne sont pas représentés sur le schéma car leur complexité nuirait à la lisibilité de ce dernier.

Le calcul des priorités ayant un délai non négligeable, celui-ci est réalisé un cycle avant l'ordonnancement. Afin d'éviter l'introduction de bulles dans le pipeline en cas de détection de "hazards", un mécanisme pour forcer le changement de contexte avec une vérification des priorités accélérée (pré-calculée) a été introduit. Du fait que les processus soient pré-triés, cette anticipation n'est pas spéculative.

Contrôle de l'ordonnanceur et des processus

La gestion du déterminisme est réalisée à l'aide de registres contenant un temps d'attente (sleep time) décrémenté à chaque cycle. A chaque étage, un "sleep time" supérieur ou égal à un nombre de cycles équivalent à la position dans la file d'attente limite sa progression. Ce "sleep time" est aussi utilisé pour maintenir un processus à l'arrêt : si tous les bits du registre sont à 0 (running) ou à 1 (wait), il n'est pas décrémenté. Cette solution a été choisie de façon à maintenir les processus bloqués en début de file d'attente.

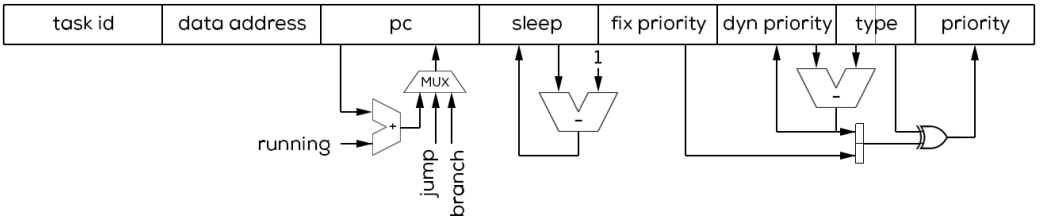

FIGURE 3.4: Structure du descripteur de processus

La gestion des priorités peut se réaliser selon 3 modes :

- **Statique** : La valeur de la priorité ne change pas
- **Dynamique** : Les bits les plus significatifs sont fixes pour permettre plusieurs catégories de processus. Les bits les moins significatifs sont décrémentés quand le processus est en cours d'exécution pour partager le temps d'exécution.
- **Dynamique inversé** : Le fonctionnement est identique au mode dynamique excepté le fait que la priorité est inversée. Ce mode permet d'accélérer les processus dont l'exécution est amorcée.

Ces différents modes ont été implémentés de façon à rendre le système compatible avec des mécanismes tels que le fair-sharing ou le earliest deadline first.

La gestion du program counter se fait depuis le pipeline. Les signaux sont envoyés vers les registres de l'ordonnanceur afin de modifier, le cas échéant, l'adresse de la prochaine instruction d'un processus donné.

Intérêts, contraintes et limites

La solution proposée permet d'obtenir un overhead d'ordonnancement exactement égal à un cycle et sans jitter (excepté celui de l'horloge du processeur). De plus, cet overhead est complètement masqué par le fait qu'il soit pipeliné avec le processeur. Cette performance s'obtient au prix d'un circuit complexe (voir section 3.5) et de mécanismes d'ordonnancements simples. Comme expliqué dans l'introduction de ce chapitre, la responsabilité d'un ordonnancement complexe basé sur des deadlines ou des caractéristiques des processus est reléguée à un ordonnanceur externe (software ou hardware) non développé.

Un autre prix à payer pour obtenir un ordonnancement à chaque cycle sans limiter la montée en fréquence du processeur est qu'un processus prioritaire prend 3 cycles pour parcourir la file d'attente. Ce coût est néanmoins négligeable dans la mesure où les processus replacés dans la file d'attente sont, soit moins prioritaires, soit mis en attente pour 3 cycles suite à une détection de contrainte sur les données ou sur le contrôle.

En revanche, pour la gestion des interruptions, ces 3 cycles d'attente (80 ns à 50MHz) sont nécessaires pour supprimer l'incertitude temporelle sur le temps de réponse. Cette attente ne bloque pas l'ordonnanceur et le processeur continue à exécuter d'autres processus. Pour garantir ce temps de réponse, il est bien entendu nécessaire que la tâche gérant l'interruption ait une priorité suffisante pour progresser à travers les étages de l'ordonnanceur jusqu'à exécution.

La solution trouvée pour accélérer les changements de contextes peut être appliquée au reste du circuit et l'optimiser tant au niveau de la taille du circuit qu'au niveau de la montée en fréquence. Elle permettrait de réduire considérablement le nombre de multiplexeurs tout en conservant la quantité de registres utilisés. Le principe est de réaliser une partie du calcul des priorités un cycle avant la fin du calcul et de faire migrer les valeurs de priorité et de temps d'attente d'un registre à l'autre plutôt que de lire ces données depuis un tableau de registres.

3.2.2 Encodeur d'adresses virtuelles

Ce circuit a été développé afin de gérer la correspondance entre l'adresse physique et l'adresse virtuelle d'un thread ou d'un processus. La microarchitecture développée ne pouvant gérer qu'un nombre limité de processus, il est nécessaire d'établir une relation entre le numéro d'identification (id) du processus et son adresse physique. Le schéma de ce circuit est illustré Fig. 3.5.

Au démarrage du système, le gestionnaire d'adresses libres (Free Addr) s'occupe d'initialiser huit registres avec des valeurs allant de 0 à 7. Ces valeurs correspondent aux adresses

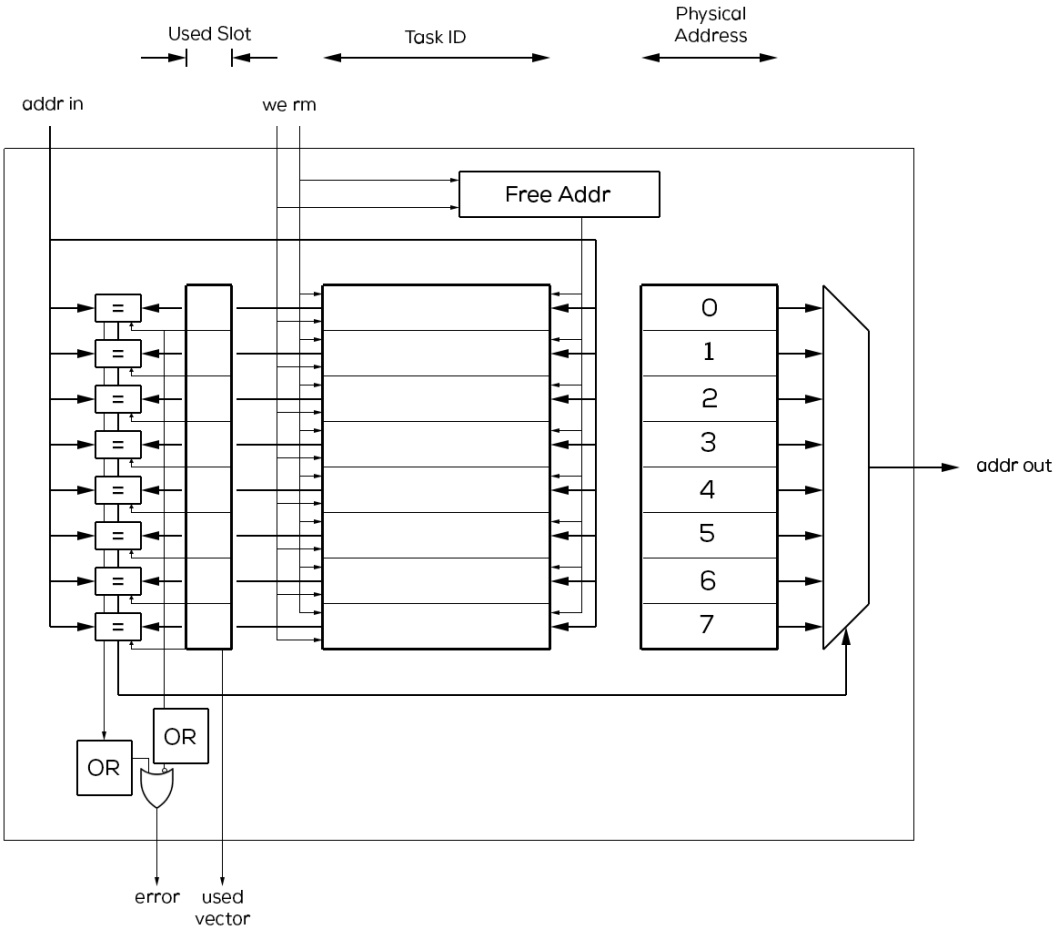

FIGURE 3.5: Encodeurs d'adresses virtuelles

physiques disponibles. L'adresse d'entrée (addr in) sert, à la fois à enregistrer ou supprimer une adresse, mais aussi à lire une adresse physique (addr out). Le signal d'erreur (error) permet de savoir s'il existe un conflit ou si l'adresse virtuelle n'existe pas. Un conflit signifie que plusieurs registres contiennent le même id. La détection de conflit n'est plus nécessaire car le circuit a été conçu pour éviter une telle situation. Cette fonctionnalité a été développée afin de valider le circuit. Le vecteur d'espaces utilisés (used vector) permet de savoir quels sont les emplacements actuellement occupés par un processus.

En mode lecture, l'adresse virtuelle entrée est comparée avec les registres du circuit. Si une adresse correspond, son adresse physique correspondante est propagée vers la sortie (addr out). Si elle ne correspond à aucune adresse, l'adresse physique est définie à 0 et le signal "error" est actif.

Si une adresse virtuelle n'existe pas et que le signal "write enable" (we) est actif, l'adresse est enregistrée en un cycle dans un registre disponible pointé par le gestionnaire d'adresses libres. Ce dernier retire l'adresse physique des adresses disponibles. Le

marqueur “Used Slot” est défini à 1. Si l’adresse virtuelle existe déjà, rien ne se passe et le circuit se comporte en mode lecture.

Si une adresse virtuelle existe et que le signal “remove” (rm) est actif, l’adresse est supprimée en un cycle. Le marqueur “Used Slot” est défini à 0. Le gestionnaire d’adresses libres ajoute l’adresse physique aux adresses disponibles. Si l’adresse virtuelle n’existe pas, rien ne se passe et le circuit se comporte en mode lecture. Le signal “error” est actif car aucune adresse n’est trouvée.

Il faut attendre un cycle pour lire l’adresse physique venant d’être attribuée à une adresse virtuelle. Une attribution d’adresse physique prend donc deux cycles tandis qu’une lecture prend un cycle.

L’implémentation en Verilog permet de paramétrer la largeur des adresses virtuelles et physiques. Cette fonctionnalité limite les optimisations possibles en terme de délai du circuit. Pour améliorer ce circuit il faudrait utiliser une stratégie de développement se concentrant sur les performances et basée sur une étude de l’état de l’art. Cette approche n’a pas été utilisée car les performances actuelles ne sont pas contraignantes pour le fonctionnement de la microarchitecture et que l’objectif premier de ce travail était d’obtenir un résultat fonctionnel.

3.2.3 Décodeur d’instructions

Seule la gestion par le décodeur des ”hazards” et du multithreading diffère du décodeur implémenté sur le MIPS. Comme expliqué section 3.2, le décodeur gère les contraintes de contrôle et de données. En conformité avec le fonctionnement d’une microarchitecture multithread avec traitement par block statique implicite (voir section 2.3.4), les instructions ne respectant pas les contraintes provoquent un changement de contexte. Cette solution a été choisie afin de maximiser l’utilisation du processeur sans recourir à une exécution spéculative ou à l’insertion de bulles dans le pipeline.

Le non recours aux exécutions spéculatives est nécessaire pour assurer un temps d’exécution déterministe. Il en est de même pour l’insertion de bulles car cette technique implique le blocage du pipeline et donc l’impossibilité de garantir un temps d’attente déterministe pour l’ordonnanceur. Le temps de calcul ainsi libéré est utilisé par un thread disponible pour exécution. Le décodeur attribue un temps d’attente au processus sortant égal à la longueur de la file d’attente de l’ordonnanceur. De cette façon, le temps d’attente d’un processus prioritaire sera constant (voir section 3.2.1). Ce temps est de 4 cycles pour le prototype développé.

Ce type de microarchitecture réduit la performance du point de vue d'un processus mais l'augmente du point de vue du système car il permet de combler les temps morts et les accès à la mémoire cache en profitant du TLP.

Le coût en performance du point de vue d'un processus est que les instructions de type load ou de branchement s'exécutent en un nombre de cycles équivalent à la longueur de la file d'attente. Ainsi, sur base de la fréquence d'utilisation des instructions (Table 2.1) et sous l'hypothèse que les instructions non répertoriées n'introduisent pas de "hazards", le coût en performance peut se calculer comme suit :

$$Slow = load + branch + call + return = 44\% \quad et \quad Fast = 100\% - Slow = 56\% \tag{3.1}$$

$$Speed = \frac{Slow + Fast}{4 * Slow + Fast} = 43.1\% \tag{3.2}$$

La performance calculée est normalisée par rapport à un processeur capable d'exécuter toute les instructions répertoriées en 1 cycle sans bloquer le pipeline et sans insérer de bulles dans ce dernier.

A fin de comparaison, sur base des mêmes hypothèses et en considérant l'insertion de bulles après une instruction load comme négligeable, la performance de l'implémentation du MIPS utilisée serait de :

$$Slow = branch + call + return = 22\% \quad et \quad Fast = 100\% - Slow = 78\% \tag{3.3}$$

$$Speed = \frac{Slow + Fast}{4 * Slow + Fast} = 60.24\% \tag{3.4}$$

Sans pouvoir compenser ces temps morts en exécutant d'autres threads.

Bien que cette baisse de performance soit significative, la solution multithreadée concurrente, à savoir : le "cycle-by-cycle interleaving" (voir section 2.3.4), appliquée à une microarchitecture similaire ne pourrait offrir qu'un maximum de 25 % de la capacité de calcul par thread. Une version de cette dernière solution a été développée par Kun Zeng et Fudong Liu [44]. Ces auteurs justifient leur choix par un faible impact sur le hardware (voir section 3.5). Le coût important en hardware lié à la gestion du traitement des processus par bloc avec ordonnancement se justifie par l'apport de fonctionnalités temps réel.

En revanche, en négligeant les accès au-delà de la mémoire cache L1, sachant que les instructions induisent une latence maximale de 4 cycles, 4 processus par voie d'exécution suffisent pour garantir l'absence de temps mort.

Une amélioration simple au niveau du décodeur permettrait d'augmenter les performances par thread : utiliser le changement de contexte statique explicite pour la gestion multithread par bloc (voir section 2.3.4) cumulée à la technique de branchements et sauts retardés (voir section 2.6). Au lieu de détecter les instructions pouvant introduire des latences, la latence peut être codée directement dans l'instruction. Cette solution permettrait de réduire légèrement la taille du circuit. Mais le principal bénéfice est de permettre l'insertion d'instructions là où un changement de contexte serait déclenché.

Avec cette technique, il est non seulement possible de réduire l'impact des changements de branches mais aussi d'éviter de déclencher un changement de contexte avec l'instruction load : soit le compilateur insère une instruction ou un "noop" pour éviter le conflit de type RAW avec insertion de bulle, soit il libère le pipeline en indiquant à la tâche de dormir pour 4 cycles. En appliquant ces techniques il est possible de réévaluer la performance moyenne mono-thread :

$$Slow = branch + call + return = 22\% \quad et \quad Fast = 100\% - Slow = 78\%$$

Afin de simplifier la discussion, l'estimation se base sous l'hypothèse selon laquelle les instructions load nécessitant une insertion de "noop" sont négligeables. La probabilité d'insertion d'au moins une instruction valide après une instruction de branchement est de 60 % [3, p.29], pour 2 instructions la probabilité est de 20 %, pour 3 elle tombe à moins de 10 %. Sur base de ces données il est possible de faire une estimation du nombre moyen de cycles perdus : $0,4 * 3 + 0,4 * 2 + 0,1 * 1 + 0,1 * 0 = 2,1$:

$$Speed = \frac{100\%}{Slow * 3,1 + Fast} = 68.4\%$$

En revanche, cette façon de procéder ne réduit pas le temps d'attente de 4 cycles. Si la solution est appliquée systématiquement, en moyenne, 2,1 cycles seront perdus pour les branches et non récupérables par d'autres processus. Pour éviter ce problème, des modifications devraient être apportées à l'ordonnanceur pour supporter des temps d'attente plus courts que 4 cycles.

3.2.4 Register File

L'apport du multithread sur la microarchitecture MIPS nécessite la gestion de plusieurs banques de registres. Chaque thread doit avoir son propre "register file" afin de permettre un changement de contexte sans overhead. La capacité à exécuter deux instructions simultanément et de façon synchrone nécessite quatre accès en lecture et deux accès en

écriture à ces registres. L'accès synchrone signifie que les différentes voies d'exécution aient un accès dédié aux registres et aux ressources sans utiliser de systèmes de tampon de données tels qu'utilisés dans des microarchitectures de type superscalaire.

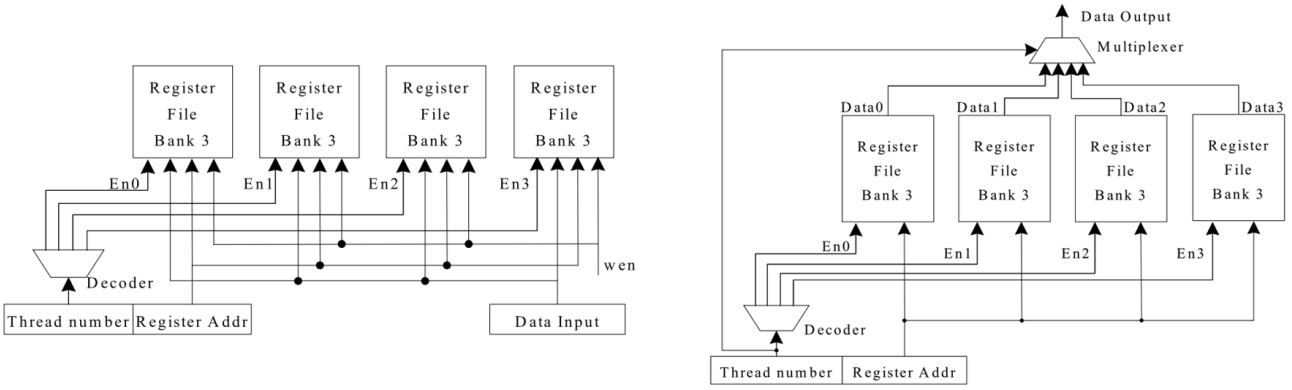

FIGURE 3.6: Logic for the Four-bank Register File [44, Fig 2 & 3]

Une possibilité d'implémentation pour un système multithread est proposée par Kun Zeng et Fudong Liu [44] et illustrée Fig. 3.6. Le principe est, pour l'écriture, de connecter directement le bus de données et l'adresse à tous les register file et de décoder l'adresse du thread pour activer le signal d'enregistrement. Même principe pour la lecture, l'adresse est directement connectée à tous les "register file" et l'adresse du thread est décodée pour activer le signal de lecture. Les différentes sorties sont ensuite multiplexées vers la sortie.

Dans le cadre de ce projet, cette solution nécessiterait quelques modifications. La micro-architecture développée présente deux bus de données et donc deux adresses associées. Il faudrait donc, d'une part, arbitrer l'adresse et l'entrée des données séparément pour chaque banque de registres et, d'autre part, dédoubler le multiplexeur de sortie. Ces modifications impliquent un surcoût en terme de surface mais aussi de montée en fréquence. Le problème de montée en fréquence peut être contourné en pipelinant le circuit d'arbitrage. Cette piste de solution reste néanmoins à explorer car elle permettrait d'envisager la possibilité de changements de contexte asynchrone. Le fait d'utiliser des banques de registres séparées permettrait d'y accéder quand elles ne sont pas utilisées par le pipeline pour y charger / décharger un contexte ou un flux de données depuis la mémoire cache.

La solution choisie a été d'implémenter un register file plus large avec deux accès en écriture et quatre accès en lecture. Ce choix a été posé dans le but de permettre l'exécution simultanée de deux instructions provenant d'un même processus. Cette capacité a été abandonnée au profit d'une solution assurant un temps d'exécution déterministe.

Le circuit résultant de cette implémentation n'a pas été optimisé et a un coût trop important en terme de surface et de montée en fréquence. Ces limites seront détaillées section 3.5.

Face à ce constat, une alternative a été trouvée mais n'a pas été implémentée (voir Fig. 3.7). Cette solution propose un circuit plus simple et mieux adaptée aux FPGA. L'objectif de cette deuxième piste de solution est de minimiser le nombre de multiplexeurs montés en série afin de diminuer les interconnections et réduire le délai de propagation.

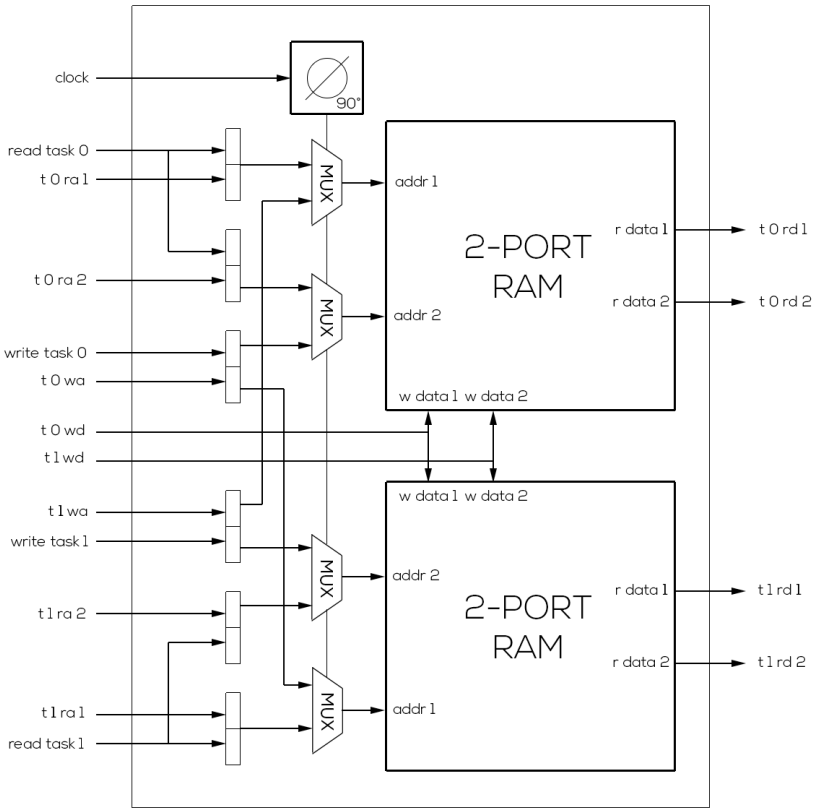

FIGURE 3.7: Register File Optimisé pour montée en fréquence

Cette alternative utilise deux mémoires 2-ports identiques fonctionnant en miroir. La lecture est déclenchée par le flanc montant de l'horloge et l'écriture sur le flanc descendant comme pour le register file du MIPS. Les deux flux de données sont enregistrés identiquement sur les deux blocs mémoire. Les deux fois deux ports de lecture sont donc disponibles pour lire quatre registres simultanément. Seuls quatre multiplexeurs de type 2 vers 1 avec un bus de 8 bits[4] montés en parallèle sont nécessaires. Le surcoût de cette solution est de nécessiter deux fois plus de registres que nécessaire et une PLL (Phase-Locked Loop) de même fréquence que l'horloge principale mais déphasée. Cette

[4] 8 bits pour adresser 8 fois 32 registres

Solution	**Bus Width**	**Nbr RF**	**Nbr Threads**	**32b Words / Threads**	**blocks M10K / RF**	**Total**
Multiple RF	32	8	8	512	2 M10K	16 M10K
Multiple RF	256	8	8	4096	13 M10K	104 M10K
4-port RF	32	1	8	64	4 M10K	4 M10K
4-port RF	256	1	8	512	26 M10K	26 M10K
Opt. 4-port RF	32	1	16	32	4 M10K	4 M10K
Opt. 4-port RF	256	1	128	32	26 M10K	26 M10K

TABLE 3.1: Comparaison des implémentations des multiples banques de registres

solution n'a de sens que pour une implémentation sur FPGA car elle permet d'utiliser des blocs mémoire dédiés[5] plutôt que des blocs logiques. Ces blocs mémoire peuvent atteindre une fréquence de fonctionnement de l'ordre de 315 MHz[6] [45, p.45]. Ces performances dépassent largement l'implémentation actuelle du register file qui limite la fréquence maximale du circuit à 62 MHz.

Cette solution tire profit du fait qu'une mémoire embarquée 2-Ports 32bits utilise au minimum 2 blocs M10K soit 20Kb de mémoire. L'implémentation classique d'un "register file" $32*32$ bits n'en utilise réellement que 1Kb et les 19Kb restants sont non utilisés. Sur base de mesures effectuées sur le logiciel Quartus II, une mémoire 2-port nécessite 1 bloc M10K par 20 bits de largeur de mot. Cela signifie qu'un "register file" de mots de 32bits sera implémenté en utilisant des mots de 40 bits et que 8 bits par mots seront perdus ou utilisés d'une autre façon. En revanche, il est possible d'utiliser le reste de la mémoire comme registres pour 15 autres threads. Différentes configurations possibles sont détaillées Table 3.1. Ces configurations sont propres au FPGA utilisé mais la même logique peut être appliquée à d'autres.

La gestion de deux threads simultanés consomme, sur base de cette implémentation, deux fois plus de mémoire. Ce qui signifie qu'une solution avec deux pipelines isolés pourrait gérer deux fois plus de threads à "register file" équivalent.à

3.2.5 Mémoire D-Cache

Pour supporter l'exécution de deux processus de façon synchrone, la mémoire cache doit pouvoir être accédée en lecture et / ou en écriture simultanément par deux ports distincts. Pour une implémentation sur FPGA, seuls les blocs mémoire 2-ports permettent d'offrir cette possibilité en exploitant la mémoire embarquée. La gestion des conflits d'adresse est gérée par le fait que la mémoire des différents processus est isolée.

[5] Cette solution utilise des blocs M10K sur Cyclone 5
[6] Sur un Cyclone V de classe 6 tel qu'utilisé pour réaliser ce projet

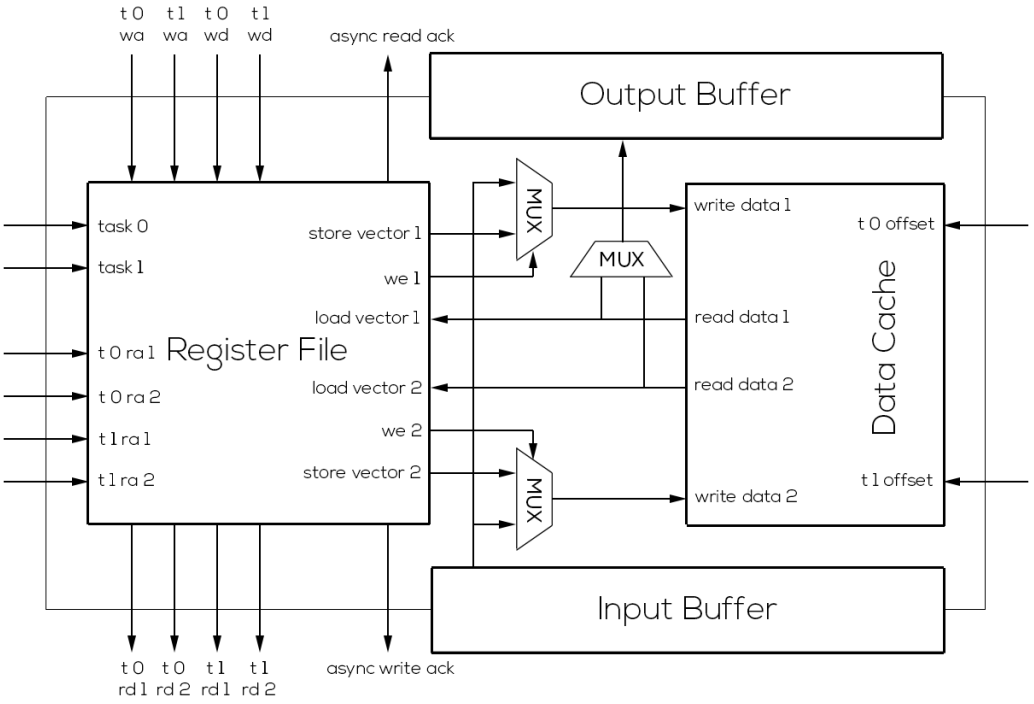

FIGURE 3.8: Register File & Mémoire D-Cache

Afin de garantir le temps d'exécution déterministe, la priorité d'accès à la mémoire cache est donnée aux flux d'exécutions. Pour y parvenir, un circuit (Fig. 3.8) permettant d'accéder à cette mémoire de façon asynchrone a été développé. Ce circuit vérifie si une instruction load ou store n'est pas en cours avant d'enregistrer ou de lire les données depuis la mémoire cache L1. En cas de succès de lecture ou d'écriture, un signal de confirmation est envoyé.

Accéder à la mémoire cache depuis l'extérieur de façon synchrone impliquerait d'utiliser une mémoire ayant au moins trois ports en écriture. La seule façon d'y parvenir en utilisant la mémoire embarquée (limitée à 2 ports) serait de la faire fonctionner à plus haute fréquence que le processeur. Cette solution est tout à fait envisageable du fait que l'alternative optimisée pour la montée en fréquence proposée pour le register file subira cette même contrainte (voir section 3.2.4). La différence étant que le register file effectue une lecture et une écriture par port par cycle tandis que la mémoire cache devrait réaliser une lecture et / ou une écriture deux fois par cycle. Cette solution nécessite donc des tests complémentaires.

La fréquence de fonctionnement de la mémoire embarquée sur FPGA n'est pas proportionnelle à sa taille pour un type de mémoire donné. Cela implique que l'implémentation d'une mémoire cache hiérarchique présente peu d'intérêt. En revanche, l'un des premiers développements futurs devrait être de connecter la mémoire cache au contrôleur mémoire

DDR afin de tester l'efficacité de la microarchitecture à masquer les overheads liés aux défauts de cache.

3.2.6 Étage d'exécution

Seule la gestion des lectures après écriture a été modifiée à cet étage par rapport au MIPS. La gestion du multithreading implique qu'il faille vérifier que les résultats sortant de l'ALU ou de l'étage d'accès à la mémoire et acheminés vers l'entrée de l'ALU proviennent bien du même processus.

Cette vérification est considérablement simplifiée par le fait que l'ordonnanceur garantit qu'un processus en exécution ne change pas de flux d'exécution tant qu'il ne sort pas de ce flux. De cette façon, il n'est pas nécessaire de faire de vérification croisée entre les deux flux.

Ce travail s'est concentré sur l'ordonnancement des processus et des instructions et la gestion du multithreading. Le fait que cette microarchitecture ne gère pas de fonctions avancées tel que le calcul en virgule flottante constitue une limite en terme de performances.

L'ordonnanceur cumulé à la gestion multithread est sensé apporter un gain en performance proportionnel aux performances du microprocesseur. Pour que le coût proportionnel lié à cette modification se justifie, il faut qu'il soit comparable aux bénéfices apportés. Ce coût est donc proportionnellement réduit si ce dispositif est appliqué à un circuit plus complexe et plus performant.

Le caractère synchrone de cette micro-architecture limite les solutions possibles à l'organisation des unités fonctionnelles. Il rend le calcul multi-cycles impossible sans enfreindre le caractère déterministe des exécutions. Cela est dû au fait qu'un calcul multi-cycles implique qu'il soit possible de bloquer le pipeline. Les unités fonctionnelles doivent donc être pipelinées et / ou mises en parallèle. Comme expliqué section 3.2, un pipeline superscalaire ne serait pas non plus compatible avec les contraintes de synchronicité et de déterminisme.

La solution la moins efficace serait de placer plusieurs unités fonctionnelles différentes en parallèle et d'allonger le pipeline en fonction des contraintes de timing de ces unités. Ainsi, suivant l'opérateur, le contrôleur redirigerait l'instruction vers l'unité fonctionnelle correspondante.

Une solution plus performante serait d'utiliser une microarchitecture de type VLIW (voir section 2.3.2). Ce type de microarchitecture permet d'accélérer les programmes

ayant un niveau d'ILP important. Mais la largeur d'instruction constante ne permet pas, comme le permettent les microarchitectures SMT basées sur des pipelines superscalaire, de combler le manque d'ILP d'un processus avec un autre. Un autre problème lié aux microarchitectures de type VLIW est que, pour obtenir de bonnes performances, les unités de calculs ne sont pas homogènes. Cela augmente le risque de non utilisation de certaines ressources en fonction des applications.

Une autre piste de solution proposée par Marcelo Brandalero et Antonio Carlos S. Beck [46], permettrait à la fois d'améliorer les performances parallèles et séquentielles. Ces auteurs proposent d'utiliser un tableau d'ALU permettant de réaliser des calculs plus complexes au sein d'une architecture superscalaire et concluent qu'une amélioration significative des performances est obtenue. Le principal problème de ce type d'unité fonctionnelle est que le résultat est obtenu avec un délai en nombre de cycles égal à la profondeur du tableau d'ALU. Cela implique que les "hazards" de type RAW ne puissent pas être résolus pour deux instructions interdépendantes ayant une distance inférieure à la profondeur du tableau. Ce problème est d'emblée résolu par l'approche multithread. Ce type d'unité fonctionnelle est capable d'effectuer des multiplications et des calculs en virgule flottante tout en permettant l'utilisation des ALU pour effectuer des calculs séquentiels et parallèles plus simples. De plus, l'utilisation de plusieurs étages du pipeline pour exécuter une instruction pourrait aussi être exploitée pour implémenter des instructions de contrôle plus complexes. Cela permettrait d'implémenter, par exemple, des instructions de multiplexage ou d'exécution conditionnelle sans créer de dépendance de contrôle. Il semble donc que cette solution constitue une alternative intéressante à l'approche superscalaire et semble particulièrement adaptée à la microarchitecture développée.

3.2.7 Pipeline à deux voies

Le pipeline développé permet d'exécuter deux threads simultanément tout en partageant la file d'attente, les banques de registres et l'accès à la mémoire cache. Cette approche souffre de deux limite : il est nécessaire de doubler le nombre de blocs mémoires pour l'implémentation choisie (voir section 3.2.4) ; la complexité de l'ordonnanceur hardware est impactée.

Cette solution, de type ordonnancement global (voir section 1.2.2), offre plus de flexibilité d'ordonnancement et la possibilité d'offrir un ordonnancement optimal et équitable pour les threads chargés. Le fait que l'ordonnancement soit hardware et que le processeur soit multithread évite les problèmes de localité de la mémoire cache. En revanche, le problème de l'évolutivité est toujours présent. Si l'implémentation d'un register file à

quatre accès simultanés s'obtient avec un faible surcoût[7], l'implémentation de mémoires plus larges nécessitera un circuit plus complexe pouvant impacter les délais. Même chose pour l'ordonnanceur, multiplier la largeur de la file d'attente nécessite l'ajout de comparateurs additionnels permettant de comparer les priorités de façon croisée. La mauvaise répartition de la mémoire cache est donc corrigée au prix d'une réduction de performance du circuit.

L'alternative serait d'utiliser deux pipelines multithreads tels que développés par Kun Zeng et Fudong Liu [44] et d'utiliser deux ordonnanceurs hardware séparés. Cette alternative utiliserait aussi deux fois plus de blocs mémoire mais pourrait supporter deux fois plus de threads dans le cas d'utilisation de registres 32*32 bits (voir section 3.2.4).

Cette alternative, de type ordonnancement local (voir section 1.2.2), est plus simple à implémenter et nécessite un circuit plus petit et plus rapide mais au prix de processus moins bien répartis et ordonnancés de façon moins équitable. De plus, le risque que le processeur n'ait pas de processus à exécuter est plus important.

Un moyen simple de s'en rendre compte est de considérer l'exécution de trois processus. Pour la première solution, le processus le moins prioritaire pourra s'exécuter quand l'un des deux processus sera en attente. Pour la seconde solution, ce processus ne pourra s'exécuter que si l'unique processus avec qui il partage la file d'attente ne s'exécute pas.

De façon plus générale, considérons la probabilité que le processeur soit actif $P(p_{run})$ en fonction du nombre de threads chargés N et en fonction de la probabilité qu'un processus se mette en attente $P(t_{wait})$. En prenant pour hypothèse que les probabilités $P(t_{wait\_i})$ soient les mêmes pour tous les processus et indépendantes entre elles, on peut modéliser ce problème sur base des propriétés de probabilité.

Sachant que pour le cas des pipelines séparés, les threads sont répartis parmi les files, le nombre de thread N est différent pour les deux cas. Dans ce modèle est considéré que $N_{split1} = 2n + 1$, que $N_{split2} = 2n$ et que $N = N_{split1} + N_{split2}$.

Pour le cas de deux pipelines séparés la probabilité qu'au moins un thread soit actif dans une file d'attente s'écrit :

$$P(p_{runX}) = 1 - P(t_{wait})^{N_{splitX}}$$

[7] 4 blocs M10Ks représentent moins de 1% des blocs disponibles sur le matériel utilisé

Et donc la probabilité qu'au moins un thread parmi les deux files d'attente soit actif $P(p_{a1})$ s'écrit :

$$\begin{aligned} P(p_{a1}) &= 1 - P(\overline{p_{run1}}) * P(\overline{p_{run2}}) \\ &= 1 - P(t_{wait})^{N_{split1}} * P(t_{wait})^{N_{split2}} \\ &= 1 - P(t_{wait})^{t_{split1}+N_{split2}} \\ &= 1 - P(t_{wait})^{N} \end{aligned} \tag{3.5}$$

Et la probabilité qu'au moins deux threads soient actif $P(p_{a2})$ s'écrit :

$$\begin{aligned} P(p_{a2}) &= P(p_{run1}) * P(p_{run2}) \\ &= 1 - P(t_{wait})^{N_{split1}} - P(t_{wait})^{N_{split2}} + P(t_{wait})^{N} \end{aligned} \tag{3.6}$$

Tandis que pour un pipeline à deux voies, les équations deviennent :

$$P(p_{b1}) = 1 - P(t_{wait})^{N} = P_{pa1}$$

$$P(p_{b2}) = 1 - P(t_{wait})^{N} - N * P(t_{wait})^{N-1} * P(\overline{t_{wait}})$$

Les probabilités $P(p_{a2})$ et $P(p_{b2})$ en fonction de N et de $P(t_{wait})$ sont illustrées Fig. 3.9. Leur différence $(P(p_{a2}) - P(p_{b2}))$ et gains relatifs $(P(p_{a2})/P(p_{b2}))$ sont illustrés Fig. 3.10.

Sur base de ces probabilités peut être modélisée l'espérance sur le taux d'utilisation T:

$$E[T_x] = E[P(p_{x2})] + \frac{1}{2}E[P(p_{x1}|\overline{p_{x2}})]$$

Pour le cas de deux pipelines séparés :

$$\begin{aligned} E[T_a] &= E[P(p_{a2})] + \frac{1}{2}E[P(p_{a1}|\overline{p_{a2}})] \\ &= P(p_{run1}) * P(p_{run2}) + \frac{1}{2}(P(p_{run1}) * P(\overline{p_{run2}}) + P(p_{run2}) * P(\overline{p_{run1}})) \\ &= P(p_{run1}) * P(p_{run2}) + \frac{1}{2}(1 - P(p_{run1}) * P(p_{run2}) - P(\overline{p_{run1}}) * P(\overline{p_{run2}})) \\ &= \frac{1}{2}(1 + P(p_{run1}) * P(p_{run2}) - P(\overline{p_{run1}}) * P(\overline{p_{run2}})) \\ &= \frac{1}{2}(1 + (1 - P(t_{wait})^{N_{split1}}) * (1 - P(t_{wait})^{N_{split2}}) - P(t_{wait})^{N}) \\ &= \frac{1}{2}(2 - P(t_{wait})^{N_{split1}} - P(t_{wait})^{N_{split2}}) \end{aligned} \tag{3.7}$$

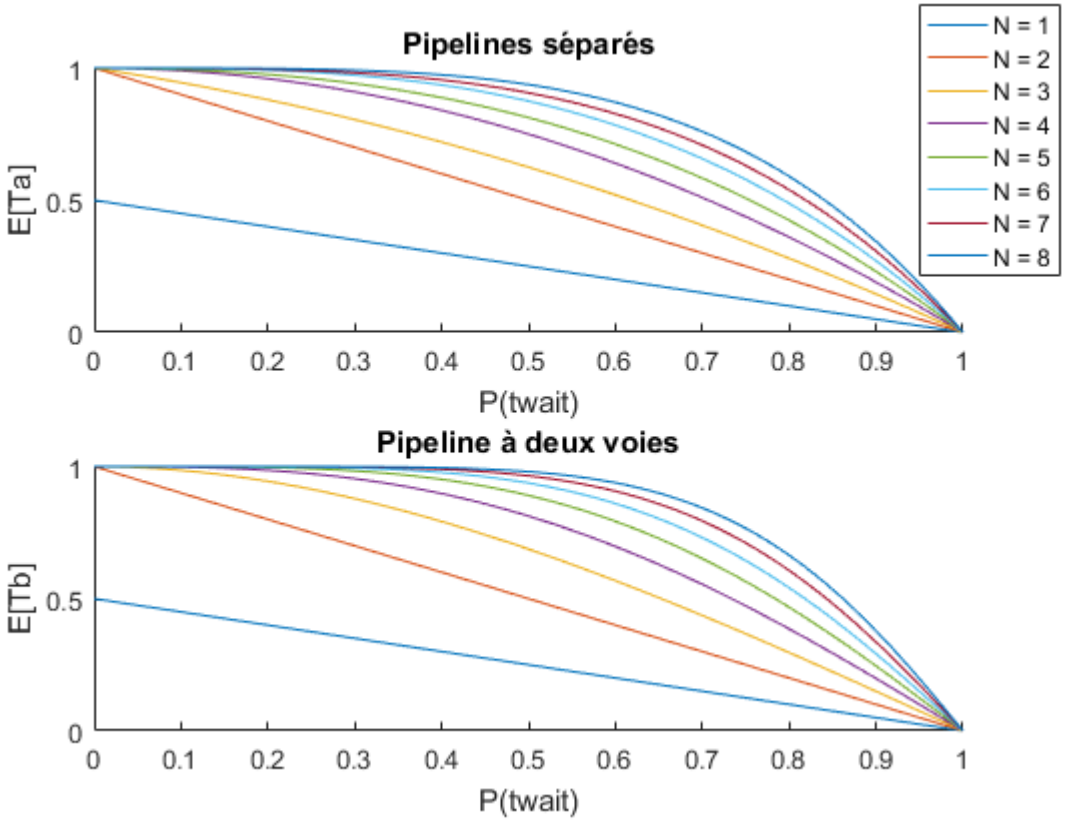

FIGURE 3.9: Comparaison du taux théorique d'utilisation des pipelines

Pour le cas d'un pipeline à deux voies :

$$\begin{aligned} E[T_b] &= E[P(p_{b2})] + \frac{1}{2}E[P(p_{b1}|\overline{p_{b2}})] \\ &= 1 - P(t_{wait})^N - N * P(t_{wait})^{N-1} * P(\overline{t_{wait}}) \\ &\quad + \frac{1}{2}(N * P(t_{wait})^{N-1} * P(\overline{t_{wait}})) \\ &= 1 - P(t_{wait})^N - \frac{N}{2}(P(t_{wait})^{N-1} * P(\overline{t_{wait}})) \end{aligned} \tag{3.8}$$

Le gain est trivialement nul dans le cas où seulement un ou deux threads sont chargés car, dans les deux cas, ces deux threads auront un accès complet aux deux unités de calcul. L'hypothèse d'équiprobabilité émise est en faveur de la solution à deux pipelines séparés car elle implique que la charge soit parfaitement répartie (Load balancing). Malgré cela, sur base de cette estimation, il est possible d'espérer un gain relatif pouvant dépasser les 10 %. Il serait tout de même nécessaire d'évaluer ce bénéfice expérimentalement afin de déterminer si le surcoût que cette solution implique est justifié.

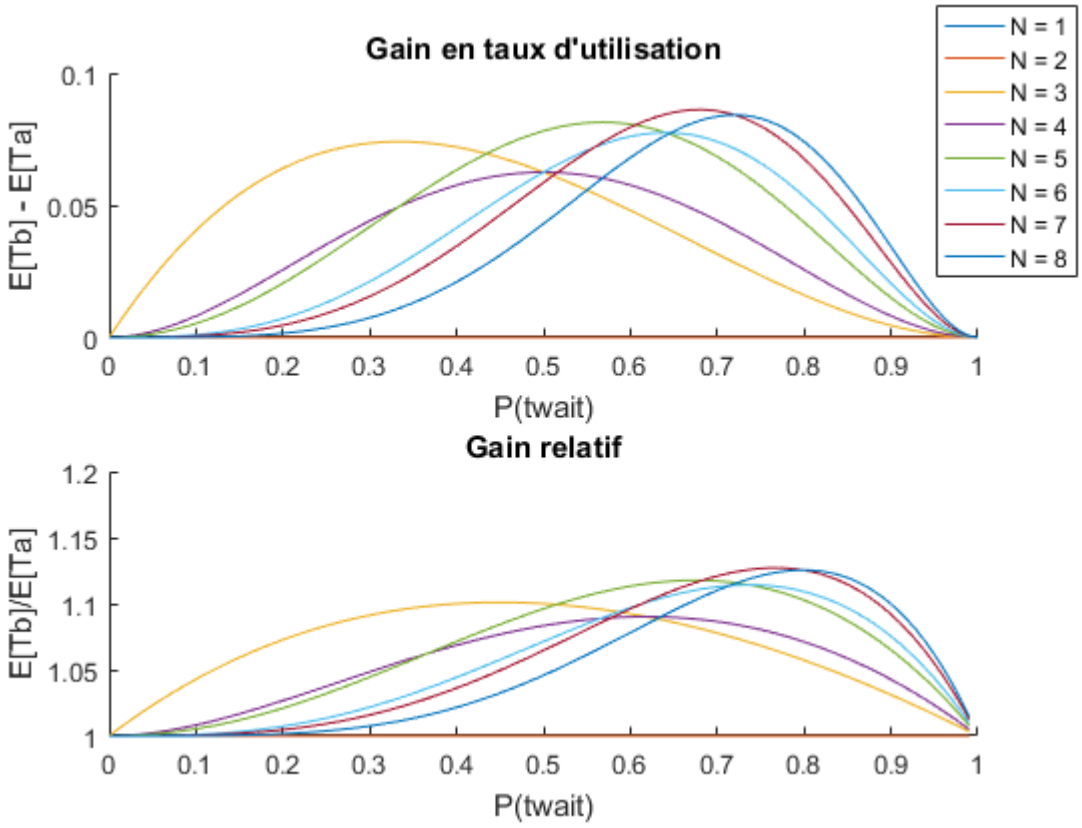

FIGURE 3.10: Comparaison du taux théorique d'utilisation des pipelines

3.3 Mode opératoire

Bien que, comme précisé par [47], les nouvelles méthodologies en terme d'exploration de design se basent sur la conception de simulations tenant compte de la diversités des solutions existantes et leur exploration systématique et automatisée, l'approche utilisée se base sur une analyse théorique suivie d'un développement pratique basé sur celle-ci. Cette méthodologie, bien que moins efficace, a été choisie de façon à pouvoir, dans un délai court, déboucher sur un résultat concret et utilisable tout en offrant un aspect formatif en terme de design hardware.

Le développement de la microarchitecture a été réalisé en Verilog et en SystemVerilog. Le circuit a été simulé sur ModelSim et testé sur un SoC HPS[8]/FPGA Altera Cyclone 5. La plateforme de développement utilisée est un Terasic SoCKit (Fig. 3.11).

Voici les différentes étapes de la conception du système :

- Prise en main de la plateforme de développement
  - Configuration du projet

[8]Hard Processor System - Processeur ARM physique

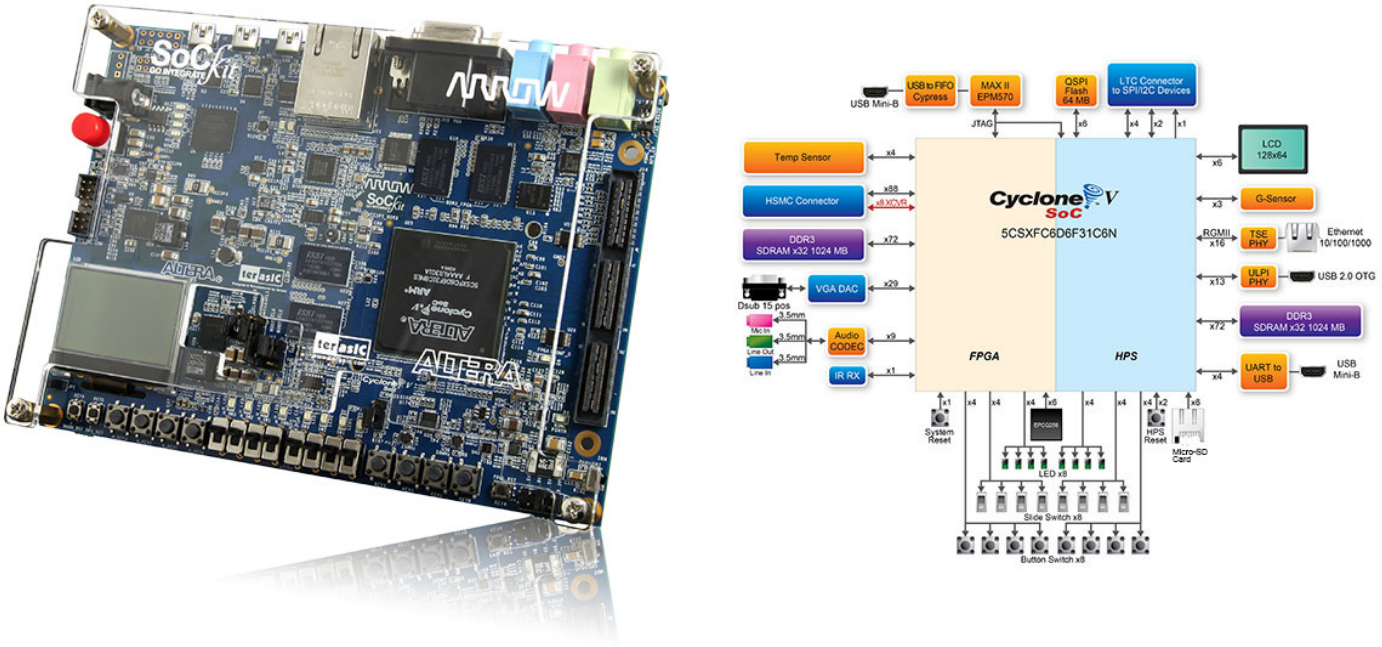

FIGURE 3.11: Carte mère Terasic SoCkit pour FPGA Altera Cyclone 5[48]

  - Configuration du système HPS/FPGA
  - Installation et prise en main de Yocto Linux [9]

- Conception de la microarchitecture
  - Analyse du système global sur base de la littérature.
  - Développement et simulation des différents circuits identifiés par l'analyse.
  - Développement et simulation de la microarchitecture complète.
  - Développement d'un contrôleur de communication entre la microarchitecture et le HPS.
  - Synthèse du circuit pour FPGA et première analyse des résultats.
  - Optimisation et perfectionnement du circuit.
  - Validation du circuit sur FPGA et prise des mesures.
- Conception de la Toolchain
  - Développement d'un serveur web / drivers en C à l'aide de la librairie Mongoose [49].
  - Développement d'un assembleur/configurateur de processus en Javascript.
  - Développement d'une interface de contrôle pour la microarchitecture en Javascript.

Le système complet de l'expérience est illustré Fig. 3.12.

[9] Distribution Linux spécialisée pour les applications embarquées

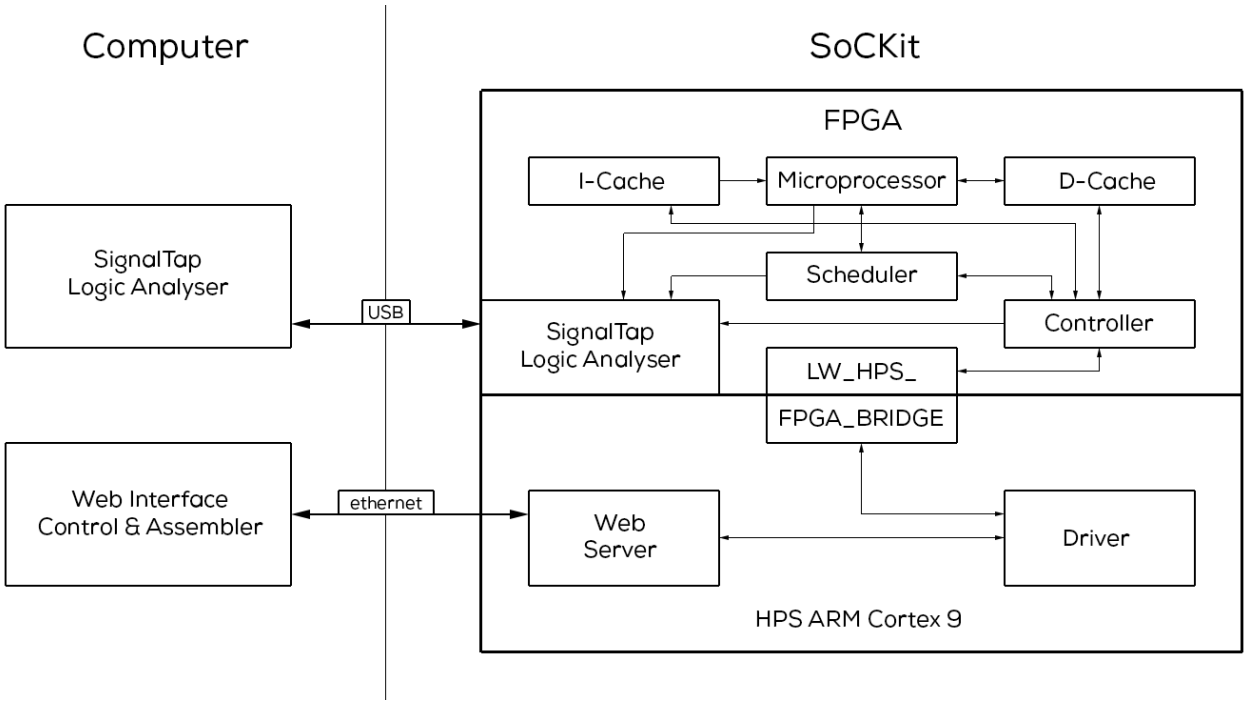

FIGURE 3.12: Système complet

Le logiciel de contrôle communique avec un serveur web hébergé sur le processeur physique du SoC. Le serveur envoie les commandes au contrôleur via un port de communication HPS/FPGA (LW_HPS_FPGA_BRIDGE). Le contrôleur gère un ensemble d'adresses permettant de configurer l'ordonnanceur et de récupérer des données statistiques, de charger des instructions dans la mémoire I-Cache ainsi que de lire et d'écrire des données dans la mémoire D-Cache. L'analyse plus détaillée du circuit est obtenue en utilisant l'analyseur logique SignalTap. Le logiciel de contrôle sera détaillé section 3.4.

3.4 Logiciel de contrôle

L'interface de contrôle intègre deux sous programmes :

- Le gestionnaire des programmes et de leur configuration.
- Le programme de contrôle et le monitorage du processeur.

L'accès aux deux programmes se fait via le menu (voir Fig. 3.13). Le bouton menu permet à tout moment d'accéder aux deux sous-programmes. Le bouton Tasks (Monitoring View) permet d'accéder au programme de contrôle (voir section 3.4.2). Le bouton Monitoring ON/OFF permet de démarrer ou d'arrêter l'enregistrement des données statistiques visibles depuis le programme de contrôle.

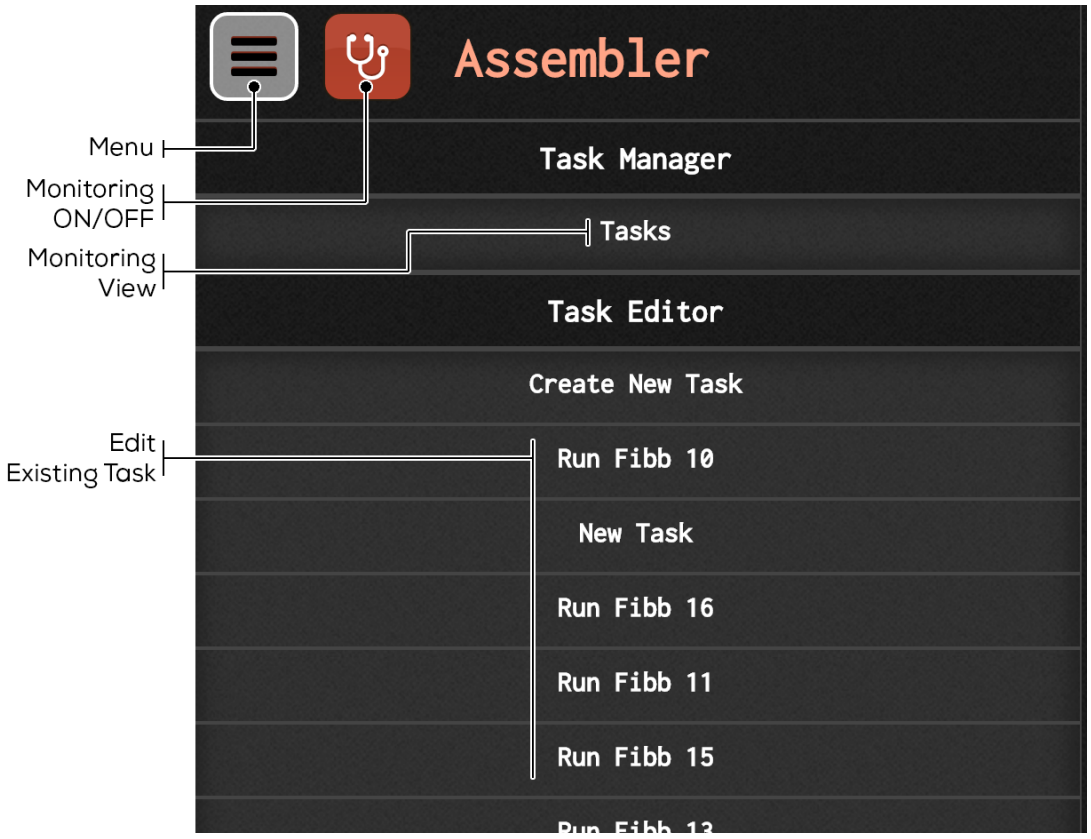

FIGURE 3.13: Menu du logiciel de contrôle

Les boutons en dessous du titre "Task Editor" permettent d'accéder au gestionnaire de programmes (voir section 3.4.1). Le premier bouton (Create New Task) crée un nouveau programme vide tandis que les autres chargent ceux qui ont été enregistrés sur le serveur.

3.4.1 Assembleur et configurateur de programmes

Ce sous-programme comprend un éditeur de code (ASM Code Editor), un éditeur de configuration (Process Configuration Setting) et un éditeur de données (Data Editor). Le bouton "Convert code to bitcode" permet d'assembler le code, d'écrire le segment de configuration et le segment de données. Un bouton sauvegarder permet d'enregistrer le programme sur la carte SD du SoCKit.

La configuration de tâche permet de définir: une priorité d'exécution; la manière dont cette priorité doit être prise en charge; la taille de la mémoire cache dédiée; l'état initial du processus.

L'éditeur de code permet d'écrire un programme en langage d'assemblage MIPS. Celui-ci utilise la librairie Cloud9 [50]. L'assembleur a été entièrement conçu en Javascript afin de simplifier l'ajout de nouvelles instructions.

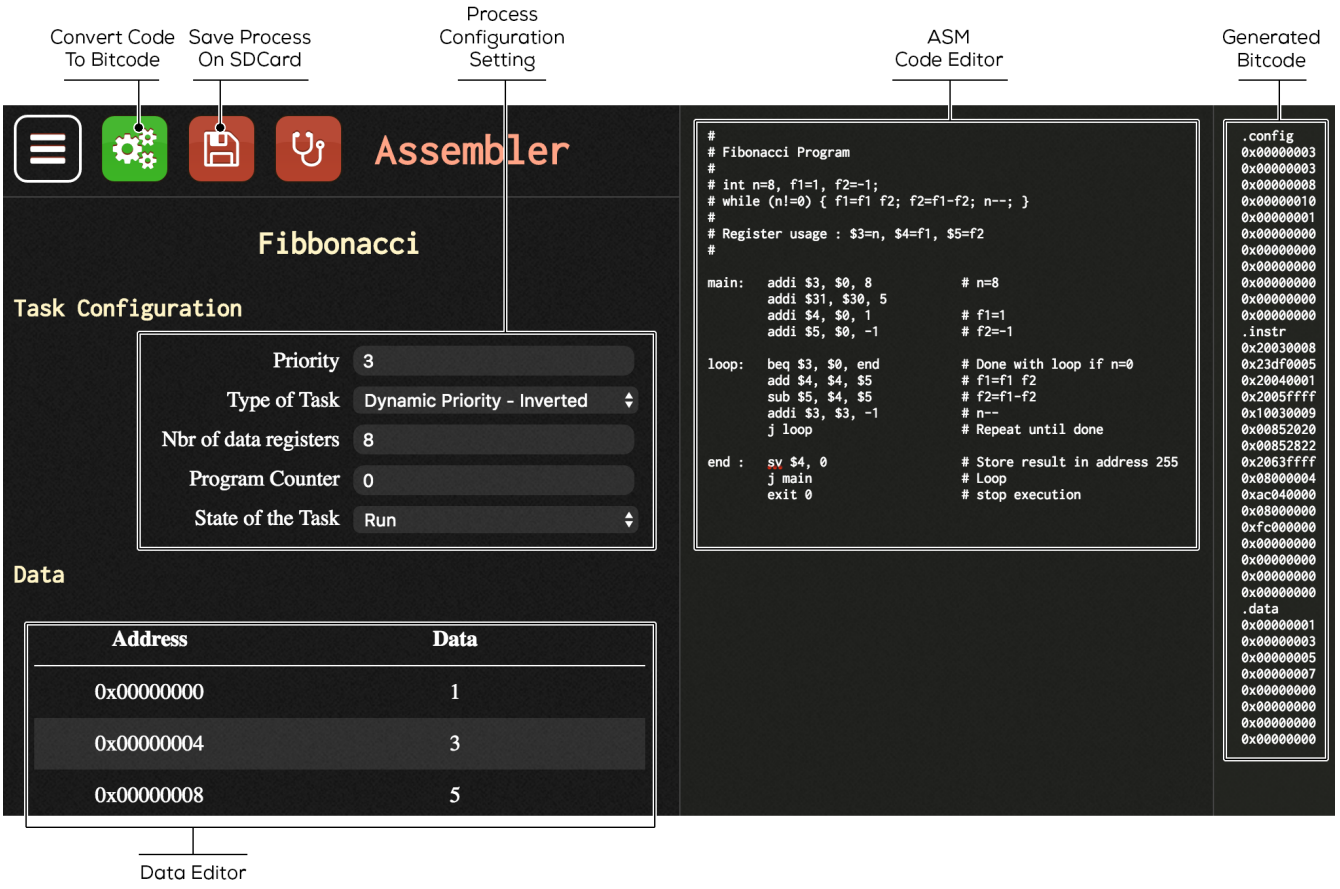

FIGURE 3.14: Editeur de programmes

L'éditeur de données est un tableau éditable permettant d'initialiser la mémoire du processus.

3.4.2 Contrôle du processeur

L'interface de contrôle du processeur (voir Fig. 3.15) permet : d'instancier des programmes; de lancer/arrêter l'exécution de processus; de récupérer les données depuis la mémoire cache; de monitorer les threads hardware.

Le tableau "Available Programs" contient la totalité des programmes sauvegardés. Les boutons "Load Program" permettent de charger le bitcode (voir section 3.4.1) dans le driver. Une fois chargée, l'instance du programme est ajoutée aux processus disponibles (Available Process).

Le tableau "Available Process" contient tous les processus chargés dans le driver. Le bouton "Run Process" permet de charger le processus dans le processeur et de le démarrer.

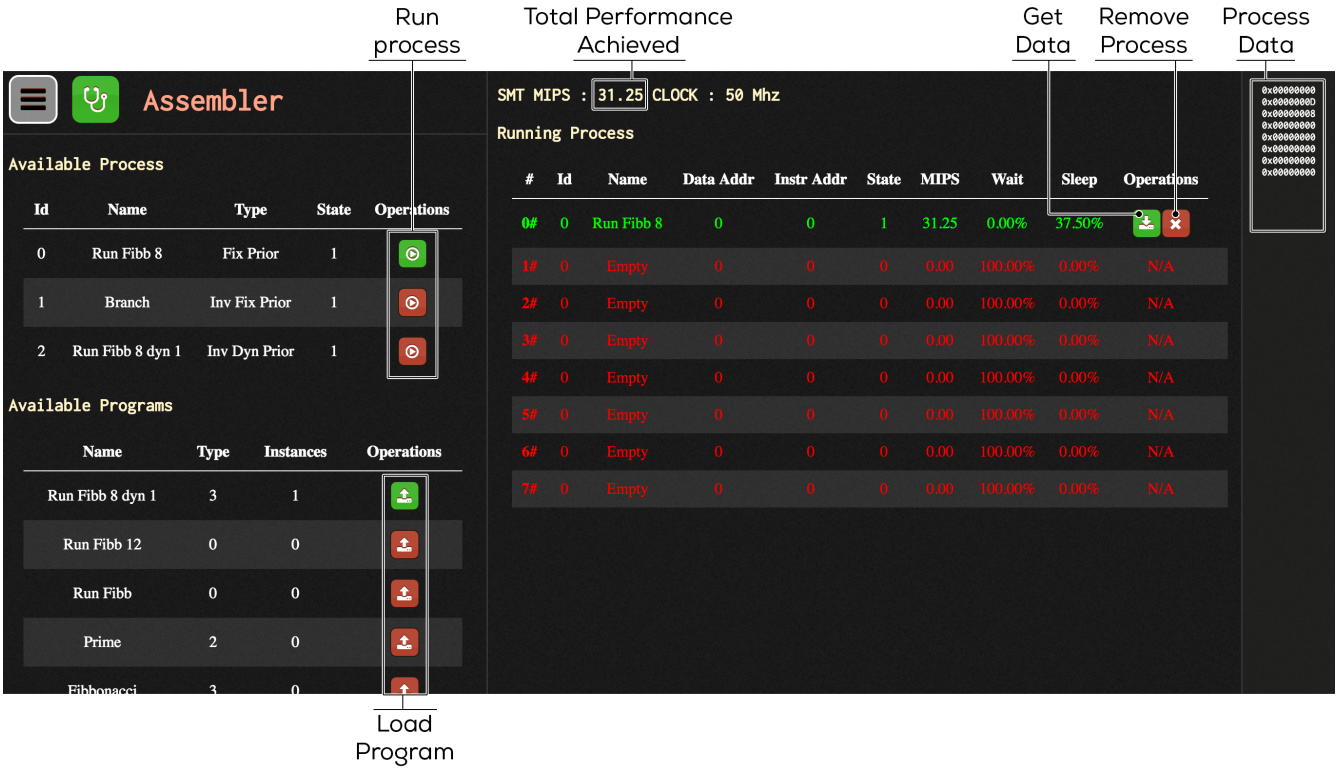

FIGURE 3.15: Interface de contrôle

Le tableau “Running Process” contient les huit threads physiques. La première colonne contient l'adresse physique du processus et la colonne “id” son id ou adresse virtuelle. Les colonnes Data Addr et Instr Addr sont des adresses pointant vers le début des segments mémoires du processus. L'ordonnanceur compte le nombre de cycles qu'un processus exécute et le nombre d'instructions déclenchant un changement de contexte. Le nombre de MIPS (Millions of instructions per second), le temps d'attente (Wait) et le temps d'indisponibilité (Sleep) sont calculés sur base de ces deux informations. Le bouton “Get Data” permet de récupérer la mémoire du processus dans le cadre “Process Data”. Le bouton Remove Process retire le processus du thread physique. La performance du processeur est affichée dans le coin supérieur gauche (Total Performance Achieved) et est comprise entre 0 et 100 MIPS. 100 MIPS correspondant à la performance maximale théorique et mesurée du processeur.

3.5 Résultats

Dans cette section seront présentés une synthèse des résultats et des pistes d'améliorations déjà présentés dans les différentes sections de ce chapitre ainsi qu'une comparaison avec différents travaux.

Les résultats de la synthèse pour FPGA en terme d'utilisation de ressources sont résumés Table 3.2. Les ressources disponibles sur Cyclone 5 et utiles pour ce projet sont les ALM (Adaptative Logic Module illustré Fig. 3.16) et blocs mémoire M10K.

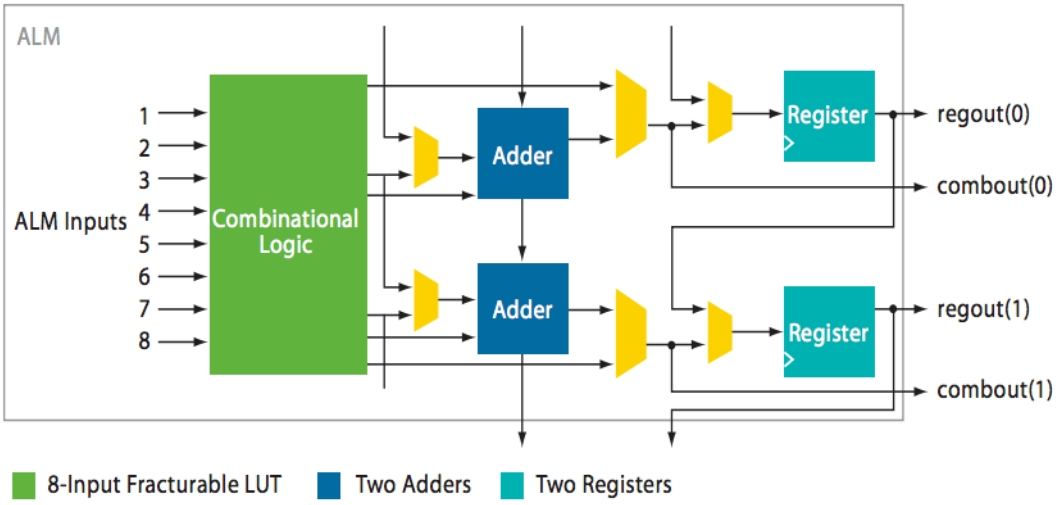

FIGURE 3.16: Adaptive Logic Module (ALM) Block Diagram [51]

Ces blocs logiques peuvent implémenter deux LUT (Look Up Table) 4 ports (ALUT) et deux registres. Le nombre d'ALUT et de registres est reporté à fin de comparaison avec d'autres types de FPGA.

L'utilisation des ressources a été segmentée en trois circuits distincts : Le "Register File", le microprocesseur et l'ordonnanceur (scheduler). Bien que ces circuits ne peuvent être vus comme indépendants, cette séparation permet de simplifier la discussion concernant les résultats obtenus.

3.5.1 Mesures et validation

L'ordonnanceur a été testé selon plusieurs configurations pour valider son fonctionnement. Afin de vérifier si le système est effectivement capable de rester actif sans prédiction de branche, celui-ci a été testé avec huit instances du programme représenté Fig. 3.17.

```
main:   addi $1, $0, 0
dum:    beq $1, $0, dum
```

FIGURE 3.17: Programme ASM de test des branches conditionnelles

Ce programme exécute constamment un saut conditionnel pour lequel la condition a été définie à "vrai". Le résultat mesuré par l'analyseur logique SignalTap illustré Fig. con-firme que chaque instance est mise en attente pour trois cycles (sleep) à chaque exécution

et qu'aucun temps mort (task_running[0] et task_running[1] sont toujours vrais) n'est enregistré.

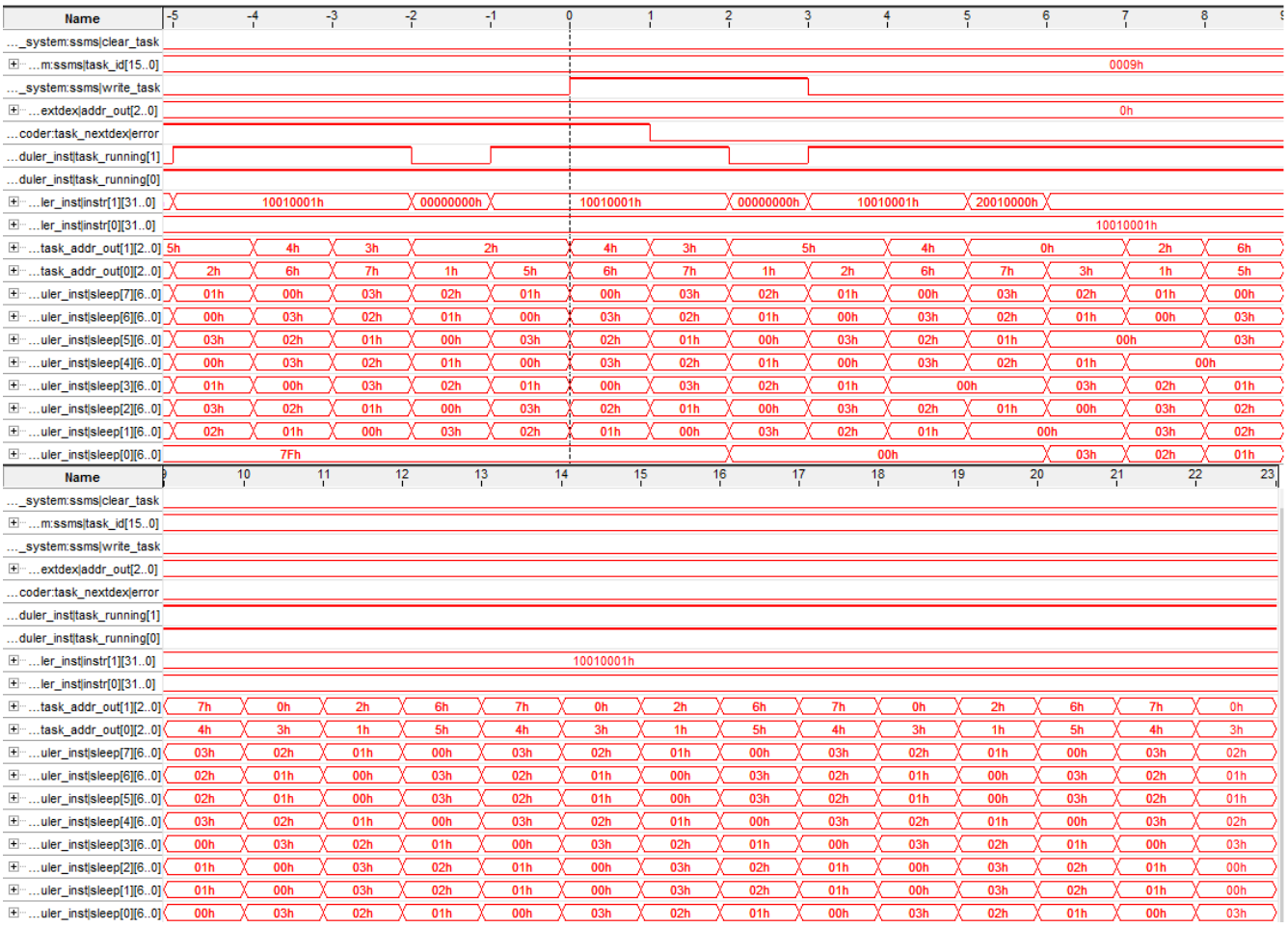

FIGURE 3.18: Validation de l'exploitation des unités fonctionnelles sans exécution spéculative

Avant l'insertion de la huitième instance du programme, l'une des unités fonctionnelles reçoit une instruction "noop" (0x00000000) tous les quatre cycles processeur. Cette instruction est envoyée dans le pipeline car le signal "task_running[1]" indique correctement que la tâche ayant l'adresse physique 2 (task_addr_out[1]) n'est pas encore prête pour exécution. Bien que le signal sleep de ce processus soit à 0, ce signal est pipeliné et indique que le processus sera prêt le cycle suivant. On peut observer que la huitième tâche ayant l'adresse virtuelle 9 (task_id) est en cours d'enregistrement entre le cycle 0 et le cycle 3. Le signal "error" (voir section 3.2.2) indique que l'adresse virtuelle n'est pas attribuée. On peut constater que le signal sleep de l'adresse physique 0 est défini à 0x7F (0b1111111). Cela signifie, comme expliqué section 3.2.1, que le thread physique est soit vide soit arrêté. Au bout de un cycle (cycle 1), le signal "error" passe à 0 car l'adresse a été correctement enregistrée et l'adresse physique 0 lui à été attribuée. Au cycle 3, les métadonnées (sleep et dyn_prior) ont été enregistrées et acheminées dans

l'ordonnanceur. Au cycle 5, la tâche 9 (adresse physique 0) a traversé la file d'attente jusqu'à exécution. Celle-ci exécute l'instruction addi $1, $0, $0 (0x20010000) et ensuite l'instruction beq $1, $0, dum (0x10010001). Pour finir, le système se stabilise et chaque tâche exécute tour à tour cette dernière instruction comme prévu théoriquement.

Le système a ensuite été testé avec quatre programmes identiques de même priorité. Il est possible d'observer (Fig. 3.19) que le système se comporte, configuré de cette façon, comme une solution multithread avec traitement par bloc classique (voir section 2.3.4). C'est-à-dire que les changements de contexte sont appliqués uniquement en cas de détection dynamique de contraintes de contrôle (hazards). Dans cette configuration, le système n'est pas préemptif. Cela signifie que si une des tâches est capable de s'exécuter en boucle sans saut conditionnel, sans accès à la mémoire et sans saut vers une adresse contenue dans un registre, celle-ci monopolisera l'une des unités fonctionnelles disponibles.

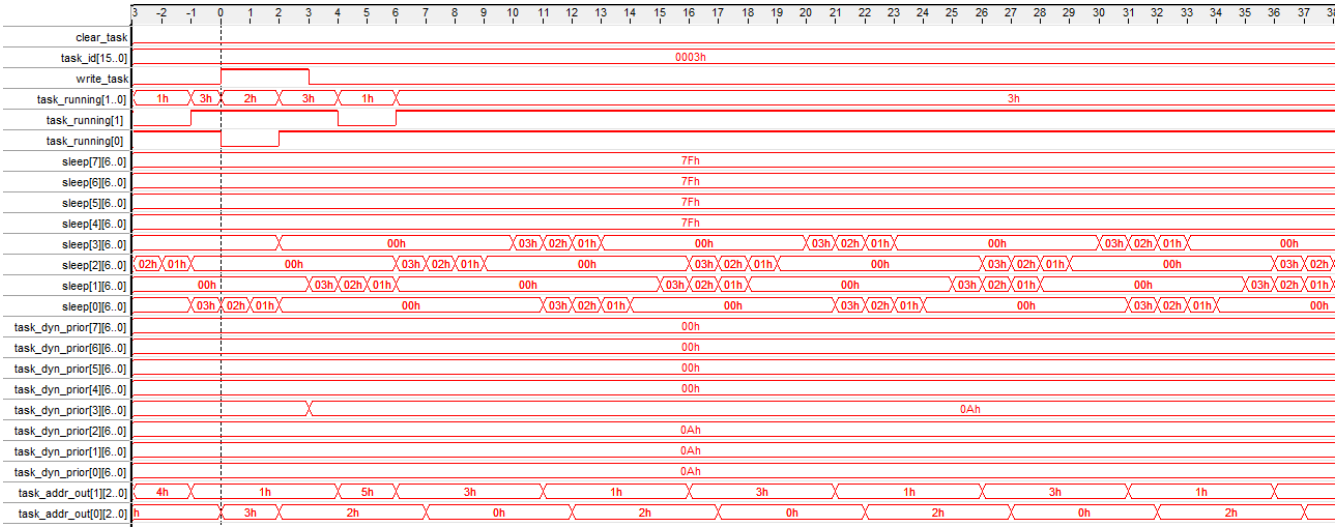

FIGURE 3.19: Test du système avec des processus de même priorité

Pour résoudre ce problème, la gestion de priorités dynamique a été ajoutée. A chaque exécution, la priorité est décrémentée de une unité. De cette façon il est possible de partager les unités fonctionnelles.

Un test (voir Fig. 3.20) a été effectué pour valider le fait que deux tâches prioritaires s'exécutent en un temps déterministe. Pour y parvenir, grâce au fait que cette microarchitecture n'utilise pas de techniques d'exécutions spéculatives, il faut vérifier que ces tâches ne passent pas plus de 3 cycles dans la file d'attente après la détection d'un "hazard". Cette expérience permet aussi de vérifier le temps de réponse d'une interruption.

On voit clairement que les deux tâches prioritaires (6 et 7) restent 3 cycles dans la file d'attente le temps que leur "sleep time" tombe à 0.

FIGURE 3.20: Validation du temps d'attente déterministe

Ce teste à aussi été monitoré depuis le logiciel de contrôle (voir Fig. 3.21) et montre bien que les deux tâches prioritaires s'exécutent au maximum de leur vitesse et que les six autres se partagent le reste du temps de calcul.

FIGURE 3.21: Validation du temps d'attente déterministe (suite)

Circuit	Number	ALM	ALUT	REG	Memory Block
Datapath	1	512	624	691	0
ALU	2	32	64	64	0
Register File	1	20887	32899	11012	0
Register File optimized[10]	1	165	366	177	4 M10k
Instruction cache	1	0	0	0	1 M10k / kB
Data cache	1	0	0	0	1 M10k / kB
Decoder	2	15	25	12	0
Advanced Controller	1	1657	2619	2450	0
Virtual Thread Address	1	144	279	168	0
Scheduler	1	498	567	50	0
Register File		**165** **20887**	**366** **32899**	**177** **11012**	**4 M10k** **0**
Microprocessor		**606**	**802**	**843**	**4 M10k**
Scheduler		**2299**	**3465**	**2668**	**0**
Total		**3070**	**4633**	**3688**	**4 + 2-? M10K**

TABLE 3.2: Résumé des ressources utilisées

3.5.2 Le microprocesseur

Le "Register File" est considéré séparément car les résultats de la synthèse ne sont pas pertinents concernant les performances pouvant être atteintes par ce système. L'implémentation initiale décrit une mémoire ayant 4 ports en lecture et 2 ports en écriture. L'outil de synthèse n'infère[11] pas correctement le circuit. L'implémentation résultante utilise une

[11] L'action d'inférer, pour la synthèse de circuit, signifie que l'outil de synthèse détecte la ressemblance entre la description du circuit décrit dans un langage de description hardware et un circuit réel implémentable

quantité disproportionnée d'ALM et, par conséquent, augmente les délais de propagation. Ce circuit constitue le chemin critique limitant la montée en fréquence à 62MHz. Une proposition d'implémentation optimisée pour FPGA a été proposée section 3.2.4. Une estimation de la quantité d'ALM, d'ALUT et de registres qu'utiliserait la version optimisée du "Register File" est proposée Table 3.2. L'estimation a été réalisée sur base du "Register File" MIPS auquel a été rajoutée une estimation de l'espace occupé par les quatre multiplexeurs d'adresse, les trois multiplexeurs d'accès asynchrones aux données ainsi que du bit de confirmation de lecture/écriture asynchrone. Les multiplexeurs de données sont de type $2 \rightarrow 1$ avec une largeur de bus de 32 bits. Les multiplexeurs d'adresse sont de types $2 \rightarrow 1$ avec une largeur de bus de 9 bits (pour adresser les 512 mots minimaux d'une mémoire M10K).

Afin de simplifier la discussion, en raisonnant au pire cas, considérons la totalité des multiplexeurs comme étant des multiplexeurs 32 bits. Sur base des résultats de synthèse, un multiplexeur 32 bits occupe 9 ALM ou 32 ALUT.

Le bit de confirmation de lecture/écriture asynchrone REG_{async_ack} peut être implémenté avec un simple registre.

Voici donc le calcul utilisé pour l'estimation totale :

$$ALM_{total} \approx 2 * ALM_{rf\_mips} + 7 * ALM_{mux}(32) = 2 * 51 + 7 * 9$$

$$ALUT_{total} \approx 2 * ALUT_{rf\_mips} + 7 * ALUT_{mux}(32) = 2 * 71 + 7 * 32$$

$$REG_{total} \approx 2 * REG_{rf\_mips} + REG_{async\_ack} = 2 * 88 + 1$$

Cette estimation a suivi un raisonnement au pire cas et ne tient pas compte des optimisations que l'outil de synthèse peut apporter en combinant ces multiplexeurs avec d'autres fonctions. Elle permet d'obtenir un ordre de grandeur concernant le nombre d'unités logiques utilisées pour une implémentation cohérente du "Register File". Cette estimation sera utilisée pour comparer ce travail sur base de données cohérentes.

Les ressources utilisées par le microprocesseur sont comptabilisées sans l'ordonnanceur et sans la banque de registres. En comparant ce résultat avec le microprocesseur MIPS à l'origine de ce travail amputé de sa banque de registres, il apparaît que le nouveau circuit occupe moins du double d'ALUT mais plus du double de registres :

$$ALUT_{mips} = 750 - 71 = 679 \qquad ALUT_{arch} = 802$$

$$REG_{mips} = 499 - 88 = 411 \qquad REG_{arch} = 843$$

$$2 * ALUT_{mips} - ALUT_{arch} = 556 \qquad 2 * REG_{mips} - REG_{arch} = -21$$

Les registres supplémentaires sont dûs au fait que le pipeline transporte l'adresse physique des threads. L'économie réalisée sur les ALUT s'explique par le fait que la nouvelle microarchitecture ne nécessite plus de gestions de blocage du pipeline et d'injection de bulles. Elle s'explique aussi par le fait que l'ordonnanceur remplace le contrôleur du MIPS. Si l'on compare la totalité de cette nouvelle microarchitecture avec le MIPS dupliqué (voir Table 3.3), il apparaît que le nouveau circuit utilise 3,08 fois plus d'ALUT et 3,7 fois plus de registres pour un gain en performance dépendant des applications. Sur base de l'estimation (3.4), le gain serait en moyenne de 66% à fréquence équivalente si les huit threads physiques sont utilisés. Ce gain en performance ne tient pas compte des améliorations fonctionnelles : La nouvelle microarchitecture permet des changements de contextes instantanés, calcule un ordonnancement par cycle, monitore le temps d'exécution et le temps d'attente, et permet l'isolation de la mémoire des processus. Ces améliorations sont difficilement quantifiables mais réduisent le nombre d'instructions à exécuter par comparaison à un système multithread software.

Les circuits ajoutés à la microarchitecture de base n'occupent pas un nombre de ressources proportionnel à cette dernière. Ce qui signifie que si le surplus de circuit est important relativement à celle-ci, ce surplus serait relativement moins important par rapport à une microarchitecture plus complexe et plus performante. En revanche cela signifie que l'intérêt de la gestion multithread avec traitement par bloc et ordonnancement hardware n'a de sens que pour une microarchitecture suffisamment élaborée.

Works	Original Work	This Work
ALUT/LUT	750/-	4633/-
REG/FF	499/-	3688/-
MEM Block	2	4
FMax	128-226MHz[1]	62-108 MHz[1]
MIPS/DMIPS	128-226/-	124-216/-
(D)MIPS/MHz	1	2
Hardware Threads	1	8
Context Switch	Software	Each Cycle Scheduled

TABLE 3.3: Microarchitecture - Comparaison des résultats obtenus

Le travail de Kun Zeng et Fudong Liu [44] est similaire mais a suivi une autre approche : utiliser un circuit le plus simple possible pour limiter l'impact en terme de ressource d'utilisation pour implémenter un pipeline multithread. Comme précisé section 3.2.7, leur travail se concentre sur un processeur multithread gérant quatre threads hardware

[1]Analyse temporelle réalisée avec TimeQuest en considérant le modèle Slow et le modèle Fast pour Cyclone 5 de classe 6

[2]Performances mesurée sur un FPGA Xilinx Virtex 5

[3]Performances mesurée sur un FPGA Xilinx Virtex 6

Works	OR1200 [44][52]	[44]	[5][12]
LUT/sLUT	4766/-	5021/-	-/14334
FF/sReg	1848/-	1908/-	-/27374
MEM Block	2	8	1686B ≈2M10K
FMax	60 MHz[2]	-	50MHz
MIPS/DMIPS	-/60	-/-	50/-
(D)MIPS/MHz	1	1	1
Hardware Threads	1	4	8-32
Context Switch	Software Scheduled	Cycle by Cycle Interleaving	MPRA[13]

TABLE 3.4: Microarchitecture - Comparaison des résultats obtenus (suite)

et permettant d'exécuter un thread par cycle. Pour obtenir ce résultat avec une faible quantité de ressources (255 LUT et 60 registres) leur choix s'est porté sur un changement de contexte à chaque cycle (Cycle by Cycle interleaving voir section 2.3.4). Cette solution permet de masquer partiellement les overheads au prix d'une division égale de la puissance de calcul entre les threads hardware. Cela revient virtuellement à obtenir un processeur à N coeurs ayant une vitesse de traitement réduite de 1/N où N correspond au nombre de threads physiques. Les overheads sont réduits, du point de vue d'un processus, à 1/N arrondi à l'unité inférieur.

En comparaison, sous l'hypothèse que les LUT des FPGA Xilinx sont comparables à ceux d'Altera, la solution développée utilise 6,8 fois plus d'LUT et 22,2 fois plus de registres par pipeline mais implémente une solution multithread avec traitement par bloc. Ce type d'implémentation profite des mêmes avantages qu'avec un changement de contexte à chaque cycle mais bénéfice en plus de meilleurs performances du point de vue d'un thread. Un autre avantage est que cette implémentation peut théoriquement masquer complètement des overheads supérieurs à N cycles. A nouveau, il n'est pas pertinent de comparer les deux implémentations en terme de ressources utilisées car ce travail apporte des fonctionnalités supplémentaires dont l'intérêt en terme de performance est difficilement mesurable tant elles dépendent du type d'application. L'approche choisie permet de répondre à des contraintes qu'une implémentation "Cycle by Cycle interleaving" ne saurait assurer. Il serait tout de même intéressant de comparer les gains en performance en implémentant la solution développée sur l'architecture OpenRISC1200 et en utilisant les benchmarks utilisés par Kun Zeng et Fudong Liu. Cette comparaison n'est, en l'état, pas possible car le travail développé n'implémente qu'un "subset" de l'architecture MIPS dans le but de garder la conception simple pour valider l'approche.

3.5.3 L'ordonnanceur hardware

La plupart des ordonnanceurs hardware étudiés dans la littérature sont des co-processeurs permettant d'accélérer un ordonnanceur software. C'est, par exemple, le cas du travail de Jason Agron et Al. [22] et de Pramote Kuacharoen et Al. [24]. Ce type d'approche présente plusieurs intérêts :

- L'ordonnancement est plus flexible car l'algorithme software peut être modifié.
- Les circuits consomment moins de ressources car une partie de l'ordonnancement est assurée par le software.
- La gestion software permet de facilement gérer un nombre non borné de threads.

La plupart des travaux analysés par Vasile Gheorghita Gaitan et Al. [5] confirment que l'approche d'un co-design hardware/software est généralement choisie.

Comme résumé Table 3.5, l'ordonnanceur développé utilise significativement plus de ressources que les deux solutions hardware/software. D'une part, la solution développée ne peut utiliser la mémoire du processeur pour retenir les données liées aux processus et, d'autre part, les comparateurs permettant de calculer les priorités entre les tâches doivent être répliqués pour fonctionner en parallèle et garantir un ordonnancement par cycle. Les solutions utilisant en partie le software utilisent plusieurs cycles pour calculer l'ordonnancement (voir Table 3.6), cela leur permet de réutiliser plusieurs fois le même comparateur. Une technique similaire pourrait être utilisée avec un ordonnanceur purement hardware mais l'objectif choisi est d'obtenir un ordonnancement par cycle pour permettre à celui-ci de fonctionner avec une micro-architecture multithreadée ayant un changement de contexte sans impact sur les performances. D'un point de vue du délai de calcul, l'ordonnanceur développé est indépendant du chemin critique du système mais pourrait le devenir après optimisation du pipeline. Il serait donc intéressant d'étudier le circuit plus en détails afin de voir dans quelle mesure il peut encore être optimisé.

Work	This Work	Kuacharoen [24]	Agron [22]
LUT	3465	421	967
REG/sReg	2668/-	564/-	-/518
MEM	0	0	1

TABLE 3.5: Ordonnanceur Hardware - Comparaison des résultats obtenus

La microarchitecture développée est inspirée du travail de Vasile Gheorghita Gaitan et Al. publié en 2012 [4] et mis à jour en 2015 [5].

Leur approche tire profit d'un ordonnancement hardware intégré au pipeline d'une microarchitecture RISC dans le but d'accélérer les changements de contextes et le temps de réponse. Leur microarchitecture est illustrée Fig. 3.22. Pour y parvenir, ils ont placé des registres multiplexés en parallèle entre les différents étages. De cette façon, la totalité du contexte peut être changée à chaque cycle.

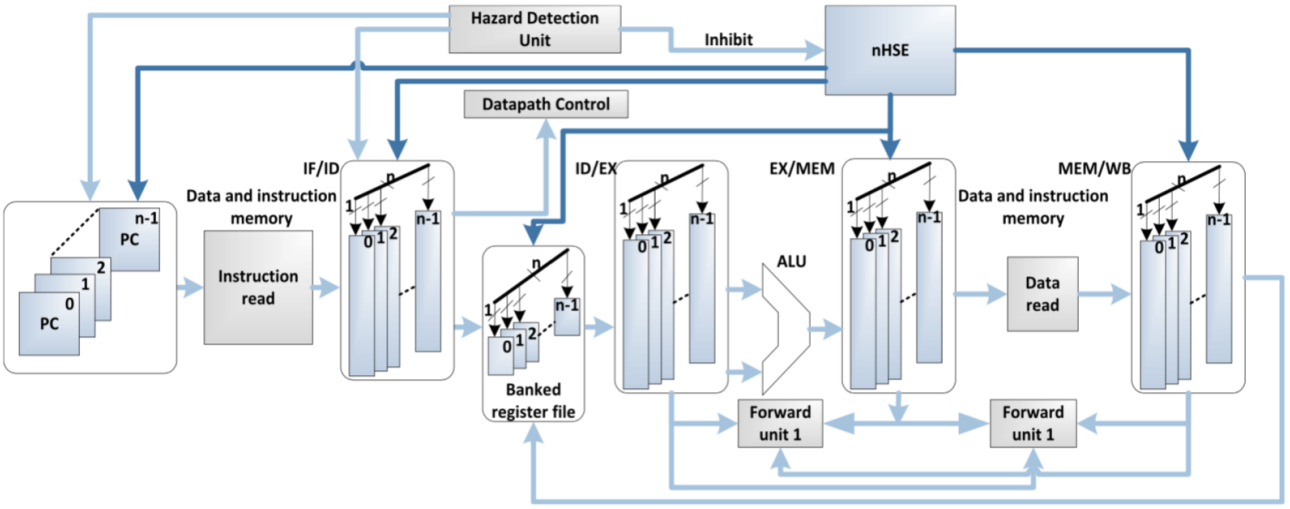

FIGURE 3.22: nMPRA architecture [5]

L'approche choisie diffère de leur travail dans la mesure où l'objectif n'est pas uniquement d'améliorer le changement de contexte et le temps de réponse pour des applications temps réel mais aussi de travailler sur les performances du multithreading et du multiprocessing.

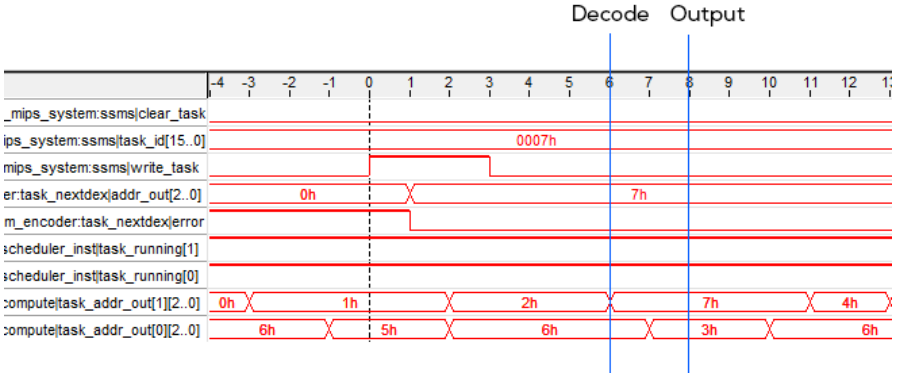

FIGURE 3.23: Temps de réponse

Le compromis choisi a été d'accepter un temps de réponse plus long afin d'éviter d'augmenter les délais entre les étages des pipelines. Le temps de réponse atteint est de 8 cycles d'horloge soit 160ns (voir Fig. 3.23 contre 75ns (voir Fig. 3.24). L'ordonnancement se fait donc à l'entrée du pipeline et non sur sa totalité. L'approche choisie permet néanmoins d'obtenir un changement de contexte à chaque cycle et profite globalement

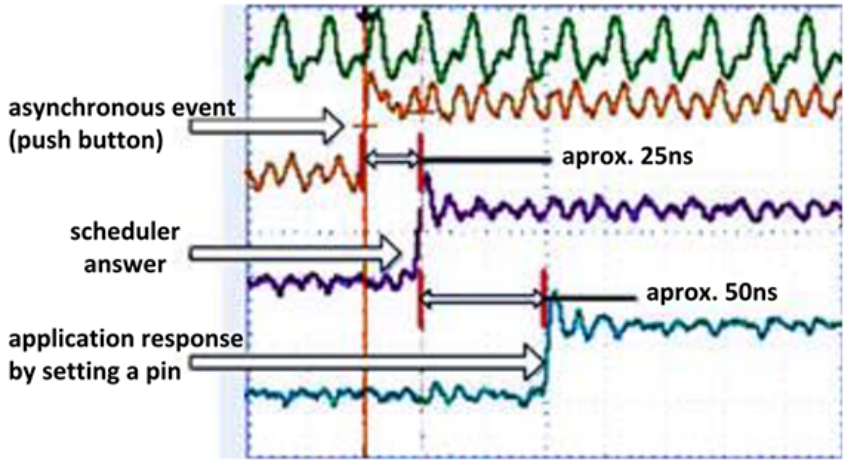

FIGURE 3.24: HSE scheduler and software application response to an external asynchronous event [5]

des mêmes avantages que l'approche MPRA excepté pour le temps de réponse. A ces avantages viennent s'ajouter ceux de l'approche multithread par traitement par bloc discutés section 3.5.2 et du multiprocessing discuté section 3.2.7. Bien que leur circuit utilise significativement[14] plus de ressources, il apporte de nombreuses fonctionnalités absentes dans ce travail. Ces fonctionnalités sont la gestions des événements, de l'envoi de messages interprocessus, de timers et de mutexes pour protéger l'accès aux ressources partagées. Ces fonctionnalités expliquent en grande partie cette différence d'utilisation de ressource. En revanche, une économie de ressources est obtenue en simplifiant le pipeline. Au lieu d'utiliser un registre de contexte par étage par thread hardware, seule l'adresse locale de la tâche est propagée à travers le pipeline.

Leur approche concernant les "Register file" a un impact sur le changement de contexte. Celui-ci varie entre 1 et 3 cycles à cause du routage entre les registres du pipeline et les "Register file". La solution développée dans ce travail ne souffre pas de ce problème pour deux raisons : le nombre de multiplexeurs montés en série est réduit du fait que les registres du pipeline ne sont pas répliqués et un seul "Register File" est utilisé pour sauvegarder tous les contextes. L'utilisation de ce type de pipeline multi-contexte [5] permet d'améliorer la synchronisation et les communications inter-processus. Ces fonctionnalités n'ont pas été implémentées dans la version actuelle de la microarchitecture développée mais celle-ci pourrait en bénéficier.

La Table 3.6 est tirée du travail de Vasile Gheorghita Gaitan et Al. [5] auquel ont été ajoutés les résultats obtenus ainsi que les résultats du travail de Jason Agron et Al. [22]. Sur ce tableau, il apparaît que les solutions hardware couplées au processeur et non

[14] Bien qu'une conversion simple entre les sLUT et les SRegisters de Xilinx et les ALUT et registres d'Altera ne soit pas possible, la différence est suffisamment importante pour considérer qu'elle soit significative.

Features	This Work	nMPRA [5]	hthreads [53]	ARPA-MT [54]	Kuacharoen [24]	Agron [22]
General register replicating	Yes	Yes	No	No	No	No
Pipeline register replicating	No	Yes	No	No	No	No
Task switching speed	20 ns 1 cycle (50MHz)	1-3 cycles	$1.7\mu s$-$3.3\mu s$ 525-990 cycles (300MHz)	$3\mu s$ 72 cycles (24MHz)	125 cycles	-
Response Time	160ns 8 cycles (50MHz)	75ns 3-4 cycles (50MHz)	-	-	-	-
Coprocessors	No	No	Yes	Yes	Yes with external Bus	Yes
Implementation	HW	HW	HW/SW	HW/SW	HW/SW	HW
Real Parallel Execution	Yes	No	Yes	No	No	No
Number Of Threads	8 HW 16 bits address space	8-32 HW	256 SW 256 HW	128 HW	-	256 HW

TABLE 3.6: Ordonnanceur Hardware - Comparaison des résultats obtenus (suite) [5]

intégrées sous forme de coprocesseur permettent d'obtenir de meilleurs performances mais limitent le nombre de threads pris en charge.

Conclusion

Ce travail porte sur la conception d'une microarchitecture avec ordonnancement hardware adaptée aux contraintes temps réel. Pour y parvenir, différents aspects de l'ordonnancement de processus en général ainsi que des solutions hardware ont été étudiés chapitre 1 afin de poser la théorie sur laquelle se base la conception de l'ordonnanceur développé. Ce travail s'est ensuite penché (chapitre 2) sur l'ordonnancement d'instructions à travers l'étude d'architectures et de microarchitectures processeurs de façon à analyser les différentes options disponibles pour l'implémentation d'un microprocesseur adapté aux contraintes posées. Sur base de cette analyse théorique et du choix technique d'intégrer la notion de temps exacte d'exécution (voir introduction chapitre 3), une solution multithread avec traitement par bloc (voir section 2.3.4) intégrant la notion d'exécution simultanée a été sélectionnée (voir section 3.2). À cette solution a été ajouté un ordonnanceur hardware permettant à la fois de contrôler le microprocesseur et de contrôler l'exécution des processus. Cette microarchitecture a été simulée et validée sur FPGA.

Sur base des résultats obtenus (voir section 3.5), il apparaît que ce type de microarchitecture est particulièrement bien adapté aux systèmes temps réel et donc que le multithreading et le multiprocessing sont tout à fait compatibles avec ce type de contraintes. Le renoncement aux exécutions spéculatives a poussé cette recherche à trouver d'autres alternatives permettant de compenser les pertes de performances qui en découlent. Il a été montré (voir section 3.5) que l'approche multithread permet le maintien du taux d'utilisation du processeur sans avoir recours aux prédictions de branches et qu'en négligeant les accès mémoire au-delà de la mémoire cache de premier niveau (L1) il est possible, avec suffisamment de processus, de garantir un taux d'utilisation du processeur de 100 %. Il a été montré [44] que l'approche multithread avec changement de contexte à chaque cycle peut apporter un gain en performance moyen de 16 % allant jusqu'à 33 % grâce au masquage des latences aux accès de la mémoire cache. L'approche par traitement par bloc peut obtenir de meilleurs résultats à nombre de threads équivalent et les performances monothread peuvent atteindre des performances similaires aux approches sans gestion du multithreading[3, p.269]. De plus, l'analyse théorique de l'approche d'un pipeline à deux voies réalisée section 3.2.7 prédit un gain

théorique relatif de l'ordre de 10% dans le cas qui lui est le plus défavorable comparé à une solution à deux pipelines séparés. L'analyse expérimentale n'a pas été réalisée car la microarchitecture développée n'implémente qu'un subset de l'architecture MIPS. Il serait donc intéressant pour un développement futur de confronter cette perspective théorique à l'expérience. Si le multithreading et le multiprocessing peuvent être vus comme des contraintes nécessaires d'un point de vue des performances pour les applications temps réel, ce travail le perçoit comme une opportunité.

D'un point de vue pratique, l'implémentation de ce type de solutions sur un SoC FPGA/HPS (voir section 3.3) pourrait être utilisée comme un co-processeur permettant de seconder le processeur physique pour contrôler les circuits développés sur FPGA de manière plus fine tout en respectant des contraintes temps réel.

Bien que ces résultats soient encourageants, ce travail ne s'est pas concentré sur l'optimisation des différents circuits. Il est donc encore possible d'espérer de meilleurs résultats en terme d'utilisation des ressources FPGA et de montée en fréquence. Il est aussi possible, avec ce type de microarchitecture, d'espérer un gain significatif de performance sur les communications inter-processus [5]. Cette fonctionnalité constitue la principale limite de performance des micro-noyaux [11] et réduit le niveau de TLP atteignable par ceux-ci [19]. Cependant, bien qu'elle ne soit pas traitée dans ce travail, cette fonctionnalité pourrait facilement être intégrée au circuit actuel et par conséquent pourrait supprimer tout overhead lié à sa gestion.

Bien que les performances atteintes par la microarchitecture développée soient limitées en comparaison avec les microprocesseurs SMT basés sur la technologie superscalaire, une piste de recherche a été proposée section 3.2.6 dans le but de compenser cette limite. Cette piste propose d'étudier des unités fonctionnelles basées sur des tableaux d'ALU et ferait passer ce projet dans le rang des architectures CISC. Bien que vivement critiquée [27, 32], ce type d'architecture est encore largement utilisé dans l'industrie et une étude récente [33] tend à montrer que cette approche n'est pas dépassée par les architectures RISC. De plus, la motivation initiale des architectures CISC était d'introduire des fonctionnalités provenant de langages de haut niveau. Cette idée rejoint une des motivations initiale qui a donné naissance à ce travail, c'est-à-dire, l'intégration d'un plus haut niveau d'abstraction du point de vue du langage machine.

Bibliographie

[1] Herb Sutter. The free lunch is over. a fundamental turn toward concurrency in software. http://www.gotw.ca/publications/concurrency-ddj.htm. [Online; accessed 25-July-2016].

[2] W. Stallings. *Operating Systems Internals and design principles*. Pearson Education Limited, England, 7th edition, 2012.

[3] Jurij Silc, Borut Robic, and Theo Ungerer. *Processor architecture: from dataflow to superscalar and beyond*. Springer Science & Business Media, 2012.

[4] E. Dodiu and V.G. Gaitan. Custom designed cpu architecture based on a hardware scheduler and independent pipeline registers; concept and theory of operation. pages 1–5, May 2012. ISSN 2154-0357. doi: 10.1109/EIT.2012.6220705.

[5] Vasile Gheorghita Gaitan, Nicoleta Cristina Gaitan, and Ioan Ungurean. Cpu architecture based on a hardware scheduler and independent pipeline registers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(9):1661–1674, 2015.

[6] Linus Torvalds. Linux kernel mailing list. http://tech-insider.org/linux/research/2001/1215.html, 2001. [Online; accessed 10-April-2016].

[7] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: A decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 1:1–1:16, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4240-7. doi: 10.1145/2901318.2901326. URL http://doi.acm.org/10.1145/2901318.2901326.

[8] Prof. Kaustubh R. Joshi. Process scheduling 2. http://www.cs.columbia.edu/~krj/os/lectures/L12-LinuxSched.pdf, Spring 2013. [Online; accessed 10-April-2016].

[9] Robert Love. *Linux Kernel Development*. Pearson Education Limited, RR Donnelley, Crawfordsville, Indiana, 3rd edition, 2010.

[10] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9): 70–77, 1996.

[11] Brian N Bershad. The increasing irrelevance of ipc performance for micro-kernel-based operating systems. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 205–212. Citeseer, 1992.

[12] Shekhar Borkar, Pradeep Dubey, Kevin Kahn, David Kuck, Hans Mulder, Steve Pawlowski, and Justin Rattner. Platform 2015: Intel processor and platform evolution for the next decade.–edited by rm ramanathan and v. thomas. *Intel Corporation.:(https://www.cs.helsinki.fi/u/kerola/rio/papers/borkar_2015.pdf).*

[13] Itai Avron and Ran Ginosar. Performance of a hardware scheduler for many-core architecture. *International Conference on High Performance Computing and Communications*, (14), March 2012.

[14] A. Silberschatz, P. Baer Galvin, and G. Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., United State of America, 9th edition, December 2012.

[15] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities, reprinted from the afips conference proceedings, vol. 30 (atlantic city, nj, apr. 18–20), afips press, reston, va., 1967, pp. 483–485, when dr. amdahl was at international business machines corporation, sunnyvale, california. *IEEE Solid-State Circuits Society Newsletter*, 12(3):19–20, 2007.

[16] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31 (5):532–533, 1988.

[17] Chopard Bastien. Loi de gustafson. http://cuiwww.unige.ch/~chopard/ATO-II/cours/node15.html. [Online; accessed 27-July-2016].

[18] Amdahl's law. https://fr.wikipedia.org/wiki/Parall%C3%A9lisme_(informatique)#/media/File:AmdahlsLaw.svg. [Online; accessed 26-July-2016].

[19] Alan H Karp and Horace P Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33(5):539–543, 1990.

[20] Apple Inc. Kernel programming guide - mach scheduling and thread interfaces. https://developer.apple.com/library/mac/documentation/Darwin/Conceptual/KernelProgramming/scheduler/scheduler.html, 2016. [Online; accessed 25-April-2016].

[21] Bin Lin and Peter A Dinda. Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 8. IEEE Computer Society, 2005.

[22] Jason Agron, David Andrews, Mike Finley, E Komp, and W Peck. Fpga implementation of a priority scheduler module. 2004.

[23] Jan Karel Lenstra, AHG Rinnooy Kan, and Peter Brucker. Complexity of machine scheduling problems. *Annals of discrete mathematics*, 1:343–362, 1977.

[24] Pramote Kuacharoen, Mohamed Shalan, and Vincent John Mooney. A configurable hardware scheduler for real-time systems. pages 95–101, 2003.

[25] Nikhil Gupta, Suman K Mandal, Javier Malave, Ayan Mandal, and Rabi N Mahapatra. A hardware scheduler for real time multiprocessor system on chip. pages 264–269, 2010.

[26] Jason Nieh, Christopher Vaill, and Hua Zhong. Virtual-time round-robin: An o (1) proportional share scheduler. In *USENIX Annual Technical Conference, General Track*, pages 245–259, 2001.

[27] Greg Novick Crystal Chen and Kirk Shimano. Risc vs. cisc. http://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/. [Online; accessed 25-July-2016].

[28] P. E. Ross. Why cpu frequency stalled. *IEEE Spectrum*, 45(4):72–72, April 2008. ISSN 0018-9235. doi: 10.1109/MSPEC.2008.4476447.

[29] Olivier Sentieys. Réduction de consommation d'énergie en électronique embarquée. *Journée scientifique électronique embarquée du*, 24, 1997.

[30] Alberto Wiltgen, Kim A Escobar, André I Reis, and Renato P Ribas. Power consumption analysis in static cmos gates. In *Integrated Circuits and Systems Design (SBCCI), 2013 26th Symposium on*, pages 1–6. IEEE, 2013.

[31] Tariq Jamil. Risc versus cisc. *IEE Potentials*, 1995.

[32] Rafael Vidal Aroca and Luiz Marcos Garcia Gonçalves. Towards green data centers: A comparison of x86 and arm architectures power efficiency. *Journal of Parallel and Distributed Computing*, 72(12):1770–1780, 2012.

[33] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 1–12. IEEE, 2013.

[34] B Ramakrishna Rau and Joseph A Fisher. Instruction-level parallel processing: history, overview, and perspective. *The journal of Supercomputing*, 7(1-2):9–50, 1993.

[35] Keith Diefendorff and Michael Allen. Organization of the motorola 88110 superscalar risc microprocessor. *IEEE micro*, 12(2):40–63, 1992.

[36] Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. Mapping on multi/many-core systems: survey of current and emerging trends. In *Proceedings of the 50th Annual Design Automation Conference*, page 1. ACM, 2013.

[37] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. *ACM SIGARCH Computer Architecture News*, 26(3):3–14, 1998.

[38] Jia Xu and David Lorge Parnas. On satisfying timing constraints in hard-real-time systems. *IEEE transactions on software engineering*, 19(1):70, 1993.

[39] Jonathan Barre, Christine Rochange, and Pascal Sainrat. Une architecture smt pour le temps-réel strict. 2008.

[40] Robert G. Brown. Engineering a beowulf-style compute cluster. http://www.phy.duke.edu/~rgb/Beowulf/beowulf_book/beowulf_book/node30.html. [Online; accessed 27-July-2016].

[41] John Hennessy, Norman Jouppi, Steven Przybylski, Christopher Rowen, Thomas Gross, Forest Baskett, and John Gill. Mips: A microprocessor architecture. In *ACM SIGMICRO Newsletter*, volume 13, pages 17–22. IEEE Press, 1982.

[42] CH Séquin and D Patterson. Risc i: A reduced instruction set vlsi computer. In *Proc. 8th Itnl. Symp. on Computer Arch., Minneapolis*, 1981.

[43] David Harris and Sarah Harris. *Digital design and computer architecture*. Elsevier, 2012.

[44] Kun Zeng and Fudong Liu. A multithreaded extension to the or1200 processor. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 5, pages 123–127. IEEE, 2010.

[45] Cyclone v device datasheet. https://www.altera.com/en_US/pdfs/literature/hb/cyclone-v/cv_51002.pdf. [Online; accessed 28-July-2016].

[46] Marcelo Brandalero and Antonio Carlos S Beck. Potential analysis of a superscalar core employing a reconfigurable array for improving instruction-level parallelism. *Design Automation for Embedded Systems*, 20(2):155–169, 2016.

[47] Jürgen Teich. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430, 2012.

[48] Terasic official website. http://www.terasic.com.tw/.

[49] Cesanta mongoose official website. https://www.cesanta.com/products.

[50] Cloud9 official website. https://c9.io/.

[51] Altera. White paper fpga architecture, 2006.

[52] Or1200 openrisc processor. http://opencores.org/or1k/OR1200_OpenRISC_Processor. [Online; accessed 10-August-2016].

[53] David Andrews, Wesley Peck, Jason Agron, Keith Preston, Ed Komp, Mike Finley, and Ron Sass. hthreads: a hardware/software co-designed multithreaded rtos kernel. In *2005 IEEE Conference on Emerging Technologies and Factory Automation*, volume 2, pages 8–pp. IEEE, 2005.

[54] Arnaldo SR Oliveira, Luís Almeida, and António de Brito Ferrari. The arpa-mt embedded smt processor and its rtos hardware accelerator. *IEEE Transactions on Industrial Electronics*, 58(3):890–904, 2011.