Titre: Title:	Modélisation et conception d'une architecture à flots d'exécution multiples selon une approche de réutilisation
Auteur: Author:	Mortimer Hubin
Date:	2006
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Hubin, M. (2006). Modélisation et conception d'une architecture à flots d'exécution multiples selon une approche de réutilisation [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7894/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/7894/
Directeurs de recherche: Advisors:	Guy Bois, & Robert Roy
Programme: Program:	Non spécifié

UNIVERSITÉ DE MONTRÉAL

MODÉLISATION ET CONCEPTION D'UNE ARCHITECTURE À FLOTS D'EXÉCUTION MULTIPLES SELON UNE APPROCHE DE RÉUTILISATION.

MORTIMER HUBIN
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)

AOÛT 2006

Your file Votre référence
ISBN: 978-0-494-19309-9
Our file Notre référence
ISBN: 978-0-494-19309-9

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

MODÉLISATION ET CONCEPTION D'UNE ARCHITECTURE À FLOTS D'EXÉCUTION MULTIPLES SELON UNE APPROCHE DE RÉUTILISATION.

présenté par : HUBIN Mortimer

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de :

Mme. NICOLESCU Gabriella, Doct., présidente

M. BOIS Guy, Ph.D., membre et directeur de recherche

M. ROY Robert, Ph.D., membre et codirecteur de recherche

M. SAVARIA Yvon, Ph.D., membre

Remerciements

Je tiens d'abord à remercier mon directeur de recherche Guy Bois pour sa patience, sa confiance et son implication auprès de ses étudiants. J'aimerais surtout le remercier pour son soutient autant technique que financier, sans quoi je n'aurais pu participer a une telle aventure.

Je désire aussi souligner l'implication de tous les étudiants qui ont participé de près ou de loin à la réalisation de ce projet. Je pense principalement a mes collègues, David Quinn, Simon Provost, Francis St-Pierre, Patrick Samson ainsi que Jean-Francois Thibeault faisant tous partie du CIRCUS et certain membres du GRM qui ont contribué à mon bonheur durant mes deux années de maîtrise, rendant l'ambiance d'étude stimulante, plaisante et toujours relevée de défi.

Finalement, j'aimerais remercier ma femme Brigitte qui m'a soutenu, supporté et surtout poussé à terminer la longue rédaction de ce mémoire qu'elle a brillamment lu, relu et corrigé.

Résumé

Poussé par la loi de Moore, le degré de miniaturisation et d'intégration permet désormais de créer des systèmes embarqués sur une seule puce toujours plus performante et offrant un plus vaste éventail de fonctionnalités. Ainsi, lémergence des systèmes sur puce améliore constamment les performances de nombreux systèmes possédant de la logique dédiée. De ce fait, plusieurs domaines se sont tournés vers cette technologie qui permet sans cesse de repousser les limites.

La demande croissante des applications multimédia et de télécommunication nécessite la puissance des fonctionnalités dédiées, le parallélisme et la performance qu'offre de telles puces. Cependant, le prix à payer est fort dispendieux. Une alternative plus qu'intéressante consiste à utiliser des puces reprogrammables pour atteindre le même but. Évidemment, comme ces possède de la logique reprogrammable, elles ne peuvent atteindre les mêmes performances même si leur conception est basée sur une même technologie. Par contre, le degré de parallélisme et les fonctionnalités dédiées se retrouvent autant sur ces puces que sur des systèmes embarqués spécialisés.

Malheureusement, malgré ces performances hors de l'ordinaire, les applications orientées données rencontrent leurs goulots d'étranglement au niveau des latences de communication. Plusieurs techniques ont été proposées dans divers systèmes commerciaux pour alléger ce problème. La prélecture de données, la cohérence de cache ainsi que le multi-processus matériel en font partie. Ainsi, ce projet présente une approche différente basée sur le principe du multi-processus matériel permettant de cacher les latences de communication. L'architecture proposée contient plusieurs processeurs et vise à utiliser les modules disponibles dans les bibliothèques d'une puce reprogrammable Virtex II pro.

Le concept consiste à partager le temps d'exécution de quatre microprocesseurs pré synthétisé. Chacun de ces noyaux fonctionne à une fréquence réduite pour obtenir une

part égale dans un domaine de fréquence plus élevé. Cette organisation émule quatre contextes matériels de flots d'exécution générant un genre de parallélisme à grains fins. Ce parallélisme réduit considérablement des périodes vides, imposées par des latences de communication, à l'intérieur d'un microprocesseur.

Ensuite, l'architecture est analysée à l'aide de trois applications de test. La première permet d'évaluer l'impact sur les accès mémoire. La seconde, quant à elle, calcule une somme de contrôle sur plusieurs morceaux de données. Cette application permet d'observer les gains obtenus avec une fonctionnalité typique de traitement de paquets. Les résultats démontrent clairement les capacités de l'architecture à cacher la latence de communication.

Abstract

Modern chip design techniques constantly improve system performances. This progress continuously encounters growing communication bottlenecks. Several techniques have been proposed in commercial systems to alleviate this issue, such as data prefetching, cache coherency and hardware multithreading. However, they are relatively recent techniques in the field of system-on-chip (SoC). This paper presents a multiprocessor approach to latency-hiding based on hardware multithreading. The proposed multiprocessor architecture is targeted for a reconfigurable SoC based on the Virtex II Pro FPGA library. It essentially consists in interleaving the execution of four soft-core microprocessors each running at a reduced frequency to obtain an equal share of a faster communication channel. This organization emulates four hardware contexts implementing a fine-grain thread parallelism mode. This parallelism greatly reduces idle times inside the microprocessor imposed by long-latency operations. The design is evaluated with two test applications. The first one evaluates the pure memory-hiding abilities of the hardware multithreading platform while the second one implements a packet-processing application. The results clearly demonstrate the latency-hiding ability of hardware multithreaded processors.

Table des matières

Liste des figures

Liste des tableaux

Liste des acronymes

ASIC	Application Specific Integrated Circuit
CLB	Configurables Logic Blocs
DCM	Digital Clock Managment
DLL	Digital Lock Loops
DSP	Digital Signal Processor
E/S	Entrées / Sorties
FIFO	First-in First-out
FIR	Finite Impulse Response
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
FSL	Fast Simplex Link
IP	Intellectual Property
IP	Iternet Protocol
LC	Logic Cells
LE	Logic Element
LUT	Look-up Table
NoC	Network-on-a-Chip
NCO	Numerically Controlled Oscillator
PLL	Phase-Locked Loops
PLD	Programmable Logic Device
RAM	Read Access Memory
RISC	Reduce Instruction Set Computer
SOC	System On Chip
SOCP	System On Programmable Chip
V2P	Virtex 2 Pro
V4	Virtex 4

Liste des annexes

Introduction

Depuis les dernières décennies, la pénétration des systèmes embarqués sur le marché des semi-conducteurs continue de croître à un rythme incroyable. Effectivement, ces systèmes de plus en plus complexes que l'on retrouve désormais souvent sur une seule puce sont maintenant exploités dans plusieurs domaines importants, dont les télécommunications, le multimédia, l'automobile et bien d'autres. Pourtant, bien que très performants, ces systèmes sur puce (SOC) sont très coûteux, autant au niveau du développement qu'a celui de la fabrication. Pour pallier à ce problème, une grande proportion, voire près de 90%, des nouvelles puces contiennent au moins un processeur, qu'il soit d'usage général ou pour le traitement de signaux digitaux [JERR04]. Bien que le temps de conception soit sensiblement réduit par la réutilisation des modules de propriétés intellectuelles (IP) pré-développées et de la partie logicielle exécutée sur le processeur, l'intégration et la vérification du système en entier représentent toujours une large portion du temps de conception. En plus, le coût de fabrication de ces puces reste énorme. Ce désavantage majeur de ce type de circuits dédiés fait partie des raisons de la croissance des FPGA depuis leur introduction en 1985 [JAWC94]. Effectivement, leur quantité de logique programmable combinée à une facilité de développement équivalente amène une mise en marché plus rapide, et ce tout en gardant une performance plus qu'acceptable.

De plus, pour améliorer les performances de ces puces, les FPGA ne contiennent plus que de la logique programmable depuis déjà un certain temps. En effet, la majorité de ces puces reprogrammables possède maintenant plusieurs modules intéressants pour l'élaboration d'un système sur puce complet. Ainsi, on retrouve très souvent, sur ce type de puce, de la mémoire, des modules de communication et même des processeurs embarqués sur la puce.

Le FPGA devient donc, pour plusieurs, plus intéressant qu'un simple outil de prototypage. Ainsi on le retrouve de plus en plus, dans les systèmes orientés données, tel que dans les domaines du multimédia et de la réseautique.

Problématique

L'émergence d'Internet et le transfert grandissant des données multimédias ont amené des défis de taille dans l'industrie des télécommunications et du multimédia. Ainsi, il n'est pas rare d'avoir à décompresser de la vidéo et de l'audio en temps réel afin de traiter leurs données. Effectivement, les applications orientées données souffrent des longues latences de communication ainsi que des collisions au niveau du réseau. Or, les processeurs disponibles dans les FPGA n'exploitent pas le parallélisme nécessaire permettant de réduire de telles latences et ainsi d'atteindre des performances qui pourraient être acceptables dans ces domaines [EEL98].

De nombreuses recherches on tenté de découvrir plusieurs mécanismes permettant de réduire le délai d'attente dû aux latences des communications. Parmi ceux-ci, on retrouve le multi-processus matériel (HMT) qui, quant à lui, ne propose pas d'éliminer la latence de communication, mais plutôt de la cacher [EEL98]. Effectivement, ce concept propose de donner le temps d'attente d'un processeur à un autre flot d'exécution pour débuter ou compléter d'autres instructions. Ainsi, les trous créés par les latences de communication demeurent mais elles sont recyclées pour exécuter d'autres tâches.

Pour obtenir ce comportement, il faut multiplier les registres matériels et modifier le pipeline du processeur ciblé. Cependant, il serait intéressant d'utiliser plusieurs processeurs de base disponibles dans les bibliothèques des FPGA pour créer une architecture simple permettant d'obtenir un tel comportement.

Objectifs

Le principal objectif de ce mémoire consiste à démontrer qu'il est possible, à l'aide de composants existants et disponibles sur le marché, de concevoir un système pouvant être déployé sur un FPGA et possédant un amalgame de processeurs de base supportant une version de multi-processus matériel. Il s'agira ensuite d'analyser les gains possibles avec ce système, autant au niveau de la performance qu'au niveau de l'effort à déployer pour en tirer profit. Ainsi, si très peu de modifications aux modules disponibles sur le marché sont nécessaires afin d'obtenir une architecture performante, une solution de ce genre devient beaucoup plus intéressante que la création d'un système embarqués sur puce qui entraînerait une perte de temps de développement ainsi qu'un coût de fabrication beaucoup plus élevé.

Le second objectif consiste à explorer les diverses facultés et le potentiel des outils fournis par Xilinx pour le développement.

Méthodologie

Afin d'atteindre ces objectifs, plusieurs tâches devront être réalisées. D'abord, il semble impératif de commencer par l'exploration des différentes possibilités de l'outil EDK de Xilinx. Ensuite, il sera important de concevoir et de tester les modules utiles qui inexistent sur le marché et dont nous avons besoin pour la suite du projet. Ces modules nous permettront de réaliser une version hybride d'une architecture multi-processus matérielle. Une fois cette architecture conçue, l'effort sera déployé sur l'adaptation de plusieurs applications tests. Ces applications nous permettront de recueillir les résultats et de les comparer à des résultats compilés sur une architecture de référence.

Contribution

Bien que beaucoup de tâches et projets connexes aient dû être réalisés pour l'obtention de ces objectifs, ma principale contribution constitue un modèle simulable d'une architecture supportant le multi-processus matériel. Ce modèle a permis de recueillir les

données nécessaires pour démontrer qu'il est possible de créer ce type d'architecture à l'aide de modules existants. Ma seconde contribution provient directement de l'exploration des possibilités des outils de Xilinx et de la carte de développement. En effet, parallèlement à ce projet, j'ai créé une architecture, un module configurable, ainsi qu'une application actuellement utilisée dans le cadre du cours « Systèmes temps réel ». Cette réalisation a engendré la publication d'un article expliquant la nature des travaux préparatifs effectués [THDS05]. Finalement, l'élaboration d'un modèle d'enveloppe en couche permettant de garder une partie générique dans l'interface lors d'un changement de système constitue ma dernière contribution.

Distribution des chapitres

Ce mémoire se divise en quatre chapitres. Le premier chapitre survole et décrit certains des différents FPGA, certains processeurs logiciels ainsi que leurs modules de communication associés. Ce chapitre détaille aussi le concept de multi processus matériel. Le second chapitre quant à lui définit les choix architecturaux adoptés pour le projet, et explique plus en détails le comportement de chacun des composants choisis. L'avant dernier chapitre traite du design de l'architecture, des modifications apportées aux modules existants ainsi que de la présentation des modules créés spécialement pour le système. Finalement, le dernier chapitre, le quatrième, présente les applications utilisées pour tester l'architecture ainsi que les résultats et l'analyse découlant de ces derniers.

CHAPITRE 1

Revue de littérature

Avec la croissance incroyable du marché des semi-conducteurs et des autres technologies, de nombreuses possibilités de produits s'offrent aux architectes pour la confection d'un système sur puce. Ainsi il devient difficile de choisir chacun des composants nécessaires au bon fonctionnement de l'architecture voulue.

En effet, la conception d'une architecture embarquée sur puce avec des technologies récentes nécessite généralement une connaissance importante des différents composants nécessaires à son élaboration. Ainsi les architectes doivent connaître les types de processeurs, de mémoire, ainsi que différents périphériques qui pourrait devenir, une fois réuni, un système fiable, peu coûteux, performant et rapide à mettre en marché. Le choix des technologies de chacun des éléments est crucial pour atteindre les objectifs fixés, tant au niveau de la puissance, de la performance, de l'espace occupé par le système sur puce, ou encore du temps de mise en marché.

On retrouve actuellement en abondance des modules matériels qui permettent de créer des systèmes sans devoir réinventer la roue. Ces modules, offerts ou vendus, sont appelés « IP », acronyme tiré du terme anglais « Intellectual Property ». La majorité de ces IP sont testés et optimisés pour certaines plateformes, certaines puces reprogrammable nommés FPGA (« Field Programmable Gate Array ») ou encore certaines architectures. Parmi ces IP, on retrouve entre autres des processeurs logiciels pour systèmes embarqués, des modèles de bus, arbitres et ponts, ainsi que différents autres périphériques nécessaires à la conception d'un système sur puce. Ces éléments sont aussi généralement interfacés avec les bus auxquels ils devraient être connectés. L'utilisation de ces modules n'est donc pas évidente sur toutes les architectures : un certain travail de compatibilité peut être nécessaire au bon fonctionnement du système. Cette diversité de solutions pour la création d'un système amène un architecte à devoir

faire plusieurs choix importants, que ce soit le choix de la bibliothèque d'IP ou celui du bon FPGA. Ces décisions deviennent cruciales pour les performances du nouveau système, et c'est pourquoi il apparaît essentiel de connaître le matériel avant de se lancer dans la grande aventure. En ce sens, le présent chapitre vise à débattre de certains choix possibles pour créer un système sur puce programmable (SOPC). Plusieurs processeurs logiciels, certaines structures d'interconnexion disponibles ainsi que quelques familles de FPGA qui se trouvent actuellement sur le marché, seront donc abordés.

1.1. FPGA

Gordon Moore, co-fondateur d'Intel, prédit en 1965 que la densité des transistors intégrés sur puces allait doubler chaque 18 mois. Cette prédiction, maintenant connue comme la Loi de Moore, s'est avérée assez juste jusqu'à ce jour.

Cette croissance incroyable de la densité des transistors a permis aux ingénieurs et architectes de créer des systèmes sur puce toujours plus denses, plus complexes et plus rapides qu'auparavant. Au fil des ans, le coût d'une puce de complexité donnée a diminué grandement. Malheureusement, cette diminution n'empêche pas que leur coût de production demeure toujours très élevé, et généralement, la complexité augmente les coûts de développement [ALTE04]. Ainsi, la création de circuits intégrés à application spécifique (ASIC) étant très dispendieuse, elle n'est vraiment pas une solution viable pour une production moyenne. Par ailleurs, la capacité de programmation des FPGA continue à croître, alors que leur coût d'exploitation s'avère inférieur à celui d'un ASIC [ALTE04]. Ces deux facteurs non-négligeables ont permis aux FPGA de passer d'outil de prototypage à un médium de production de SOPC. Cette solution viable et moins coûteuse devient donc de plus en plus utilisée sur le marché des architectures embarquées.

L'évolution des FPGA ne s'arrête pas à la simple croissance de cellules logiques à l'intérieur d'une puce. Bien que très convoité et toujours grandissant, le pourcentage de

cette capacité de programmation logique tend à diminuer avec les dernières familles de FPGA, remplaçant une partie de superficie gagnée de logique par plusieurs éléments intégrés très intéressants pour nombre d'architectures sur puces. La figure 1.1 illustre la diminution de la proportion des ressources disponibles sous forme de cellules logiques et l'ajout de modules au sein des FPGA. En effet, plusieurs sociétés compétitionnent quant à la valeur ajoutée de leurs FPGA. Outre la quantité de mémoire et les entrées/sorties rapides, certains de ces PLD (« Programmable Logic Device »)) offrent des options plus surprenantes mais très intéressantes, illustrées par la portion IP dans la figure 1.1. On y retrouve notamment l'intégration d'un processeur embarqué, des processeurs de signal numérique (DSP), des modules de gestion digitale d'horloge et bien d'autres. Bien que la proportion des ressources consacrées à la logique diminue, il n'en demeure pas moins que la densité de cellules programmables dans un FPGA continuera de croître pour les futures générations de FPGA.

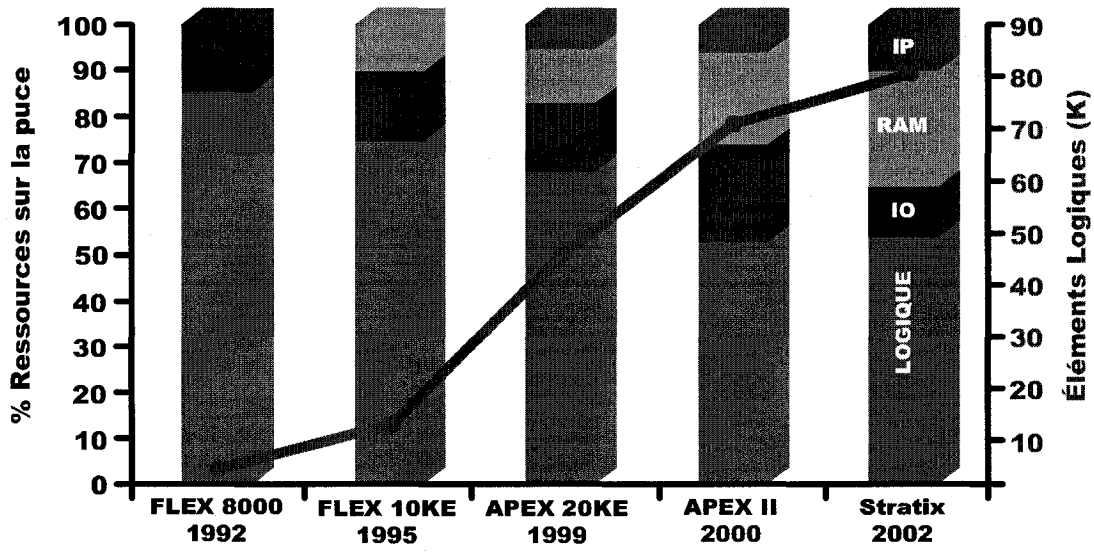

Figure 1.1 : Évolution du contenu des FPGA de Altera [ALTE04]

Parmi les éléments supplémentaires, on peut remarquer l'arrivée des processeurs directement intégrés dans la puce. Ce nouvel élément n'est pas une évolution mais une branche supplémentaire dans la généalogie des FPGA. En effet, les FPGA avec et sans processeur embarqué continuent d'évoluer chacun de leur coté. Cette distinction permet donc désormais de classer les FPGA en deux groupes : les puces orientées logique par opposition à celles orientées système.

1.1.1. FPGA orienté logique

Les FPGA orientés logique représentent généralement les éléments programmables les plus denses. En réalité, ce sont les puces qui, généralement, contiennent le plus grand pourcentage de cellules programmables. Ces FPGA représentent la continuité logique des tout premiers PLD. Ce groupe de puces est spécialement utilisé pour créer un design personnalisé, un module d'accélération, un petit système, une fonction DSP, ou encore un système complet avec un processeur logiciel. La majorité des PLD font partie de ce groupe. Parmi tous les FPGA disponibles sur le marché, la génération des « Cyclone 2 » d'Altera, considérée, selon la société, comme « les FPGA les moins chers de tous les temps », est choisie pour représenter ce groupe dans le présent chapitre. Altera propose également ses fameux « Stratix 2 » qu'elle vante comme étant les plus gros et les plus performants. Du coté de Xilinx, les « Spartan » et les « Virtex » figurent dans cette catégorie de PLD. Il est possible d'obtenir plus de renseignements sur les pages web respectives de ces sociétés [ALTE04b] et [XILI04].

Il est actuellement difficile de comparer des familles de FPGA car chacune des sociétés conceptrices utilise des métriques qui les avantagent. En effet, Altera quantifie la densité de ses produits en terme d'éléments logiques (LE), alors que Xilinx, de son coté, utilise les cellules logiques (LC) ou encore les portes logiques équivalentes. Comme les deux sociétés utilisent des tables de conversion (LUT) à quatre entrées, il semble plus sage d'utiliser cette métrique pour faire des comparaisons. Par exemple, une LE d'Altera correspond à une LUT de quatre entrées avec quelques modifications [ALTE03].

1.1.1.1. Cyclone 2

« Le FPGA le moins cher de tous les temps », c'est ce que scande Altera à propos de la génération « Cyclone™ II » qui vient surpasser la famille précédente, « Cyclone », en tous points. Cette famille orientée logique offre, selon Altera, le meilleur prix par élément logique. En effet, ce FPGA, produit sur des gaufres de 300mm et issu d'une technologie de 90 nm, offre, en plus d'un faible coût d'exploitation, une plus grande densité et plusieurs options exceptionnelles. Parmi celles-ci figurent l'intégration de multiplieurs et de mémoires embarquées, l'ajout d'entrées/sorties à très hauts débits et l'utilisation possible de boucles à verrouillage de phase (PLL) pour la gestion efficace des horloges internes au FPGA. Cette solution permet donc la création d'un système à des prix défiant celui des ASIC, et ce, avec moins de risques et un temps de mise en marché beaucoup plus rapide. Cette nouvelle génération est principalement utilisée dans l'industrie de l'automation, en communication sans-fil, dans le domaine médical et bien d'autres [ALTE04c].

La mémoire embarquée sur ces puces est instanciée de façon matérielle par blocs de 4Kb, et la totalité de sa capacité peut atteindre jusqu'à 1.1 Mb sur les plus gros FPGA de cette famille. Cette mémoire satisfait la demande nécessaire pour la création de systèmes nécessitant de la mémoire embarquée, du tamponnage de données, de la translation d'horloge et d'architecture basée sur des queues de type premier entré, premier sorti (FIFO). Cette mémoire ambivalente peut être configurée comme une mémoire vive (RAM) à double port, une mémoire vive à simple port, une mémoire morte (ROM) ou encore une structure de données de type FIFO.

Les multiplieurs, quant à eux, se retrouvent au nombre de 180 et comportent 18 bits de large ; ils sont capables d'opérer à une fréquence d'horloge d'environ 250MHz. Cet opérateur embarqué permet de modéliser plusieurs fonctions DSP communes, telles que des filtres à réponse instantanée (FIR), des fonctions de transformation de Fourier rapide

(FFT), des encodeurs/décodeurs et des oscillateurs contrôlés de façon numérique (NCO), tout en augmentant les performances du système et en allégeant le travail du reste de la logique programmée.

Le « Cyclone 2 » contient jusqu'à 68416 éléments logiques (l'équivalent d'une LUT à quatre entrées) et offre quatre PLL pour la gestion d'horloge. Ses entrées/sorties à haut débit permettent un transfert en amont de 622 Mbps, ainsi qu'un transfert en aval pouvant atteindre les 805 Mbps. Les autres caractéristiques sont présentées sur la figure suivante, selon les modèles disponibles actuellement.

Tableau 1.1: Vue d'ensemble de la famille Cyclone II

Fonction	EP2C5	EP2C8	EP2C20	EP2C35	EP2C50	EP2C70
Éléments logiques	4 608	8 256	18 752	33 216	50 528	68 416
Blocs RAM M4K	26	36	52	105	129	250
Total bits RAM	119 808	165 888	239 616	483 840	594 432	1 152 000
Mult 18x18 Embarqués	13	18	26	35	86	150
PLL	2	2	4	4	4	4
Broches utilisateur d'E/S	142	182	315	475	450	622

De plus, sur ce type de FPGA, il est possible, entre autres, d'utiliser les cellules logiques encore disponibles pour instancier le processeur logiciel « Nios 2 », conçu par Altera, qui permettra de créer un système sur puce complet. La section 1.2 introduit les processeurs logiciels.

1.1.2. FPGA orienté système

Le groupe orienté système est généralement utilisé dans le but ultime de faire un système complet possédant un ou plusieurs processeurs performants, des fonctions DSP, un système d'interconnexion, une bonne quantité de mémoire et un grand nombre de périphériques propres à un système sur puce (SOC). Ce type de FPGA contient d'habitude un processeur intégré directement dans sa logique. Par exemple, les modèles de la famille « Excalibur », développée par Altera, possèdent, en plus de la mémoire, des cellules programmables, des entrées/sorties rapides et certains « IP », ainsi qu'un ARM 9 directement intégré dans le FPGA [ALTE99]. De plus, pour un maximum d'opérabilité, la majorité des modules pouvant être ajoutés à cette puce sont compatibles avec le bus AMBA, ce qui permet une communication facile avec le processeur embarqué. Les détails de cette famille d'Altera ne sont pas présentés dans cette section. Pour plus d'informations, consutler [ALTE99].

1.1.2.1. Virtex 2 Pro

Pour sa part, Xilinx offre le « Vitrtex 2 Pro », la seule famille de cette société qui figure entièrement parmi les FPGA orientés système. La partie FX des tout nouveaux « Virtex 4 », décrits dans la section suivante, représentent également les FPGA orientés systèmes. Le « Virtex 2 Pro » est sans équivoque l'évolution naturelle du populaire « Virtex 2 ». Ce circuit intégré est conçu selon un procédé technologique à 0.13 micron. Xilinx a utilisé une combinaison de 9 couches et ces puces ont été produites sur des gaufres de 300 mm. Pour terminer la description technique, le boîtier est capable de détenir jusqu'à 1200 broches [XILI04]. Le « Virtex 2 Pro » est basé sur le même arrangement physique que le « Virtex 2 ». Cet arrangement consiste en une matrice de 2 dimensions qui contient des blocs logiques configurables couramment appelés CLB. Chacun de ces blocs contient quatre tranches[1], tel qu'illustré à la figure suivante, qui contiennent à leur

[1] Terme généralement utilisé en anglais : Slice

tour deux LUT (« Look-up Tables »), deux registres flip-flop, ainsi que deux entrées dédiées pour la retenue de somme. Avec cette configuration, chaque LUT peut être utilisé comme n'importe quelle fonction, registrée ou non, de 4 bits d'entrée, dont un registre à décalage programmable de 16 bits ou encore une mémoire RAM de 16 bits. Chaque CLB contient des composants intégrés pour interconnecter tous les éléments qui y sont inclus.

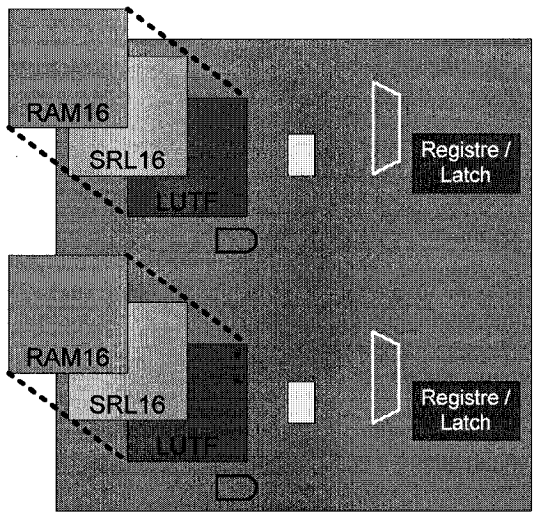

Figure 1.2 : Arrangement d'une tranche d'un Virtex 2 et Virtex 2 Pro [XILI04]

On retrouve, en périphérie de cette grande matrice de blocs logiques programmables, un amalgame d'entrées/sorties qui peuvent supporter environ 25 différentes normes de communication. La vitesse maximale que peuvent atteindre ces blocs est de 840 Mbps.

L'architecture de cette famille contient aussi un large éventail d'autres composantes intéressantes. Parmi celles-ci, on retrouve une grande quantité de blocs de mémoire RAM de 18Kbits. Ces blocs de mémoire vive (BRAM) peuvent être configurés de plusieurs façons, notamment en mémoire à double port de différentes grandeurs et profondeurs. Outre la mémoire intégrée au FPGA, on observe aussi une variété de fonctions DSP, dont une quantité plus qu'intéressante de multiplieurs dédiés d'une largeur de 18 bits. Finalement, cette puce inclut également plusieurs modules de gestion

numérique d'horloge (DCM) permettant de gérer efficacement un système à plusieurs domaines de temps. [XILI05]

Enfin, un processeur « Power PC » 405 d'IBM est directement intégré dans le FPGA, ce qu'on appelle généralement un « Hard Core ». Ce dernier module est en fait un processeur 32-bits, basé sur une architecture de style « Harvard[2] » et pouvant fonctionner théoriquement à 300 MHz et 420 Dhrystone MIPS. Certains FPGA de la famille ont un seul PPC, mais il en existe aussi à deux. De plus, il y a une série d'entrées/sorties supportant de très hauts débits, nommé communément « RocketIO », communiquant jusqu'à 3.125Gbps. De ceux-ci, on en retrouve de 4 à 24, selon le « Virtex 2 Pro » choisi. [XILI05]

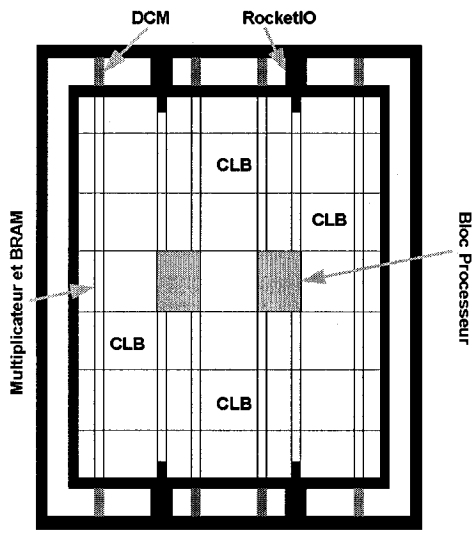

Figure 1.3 : Vue d'ensemble de l'architecture d'un Virtex 2 Pro [XILI04]

Pour communiquer avec le PPC405, IBM procure sa suite « CoreConnect » décrite plus loin dans ce mémoire. Afin de rendre la tâche de conception de système plus facile aux ingénieurs, Xilinx, en conjonction avec IBM, offre un amalgame de noyaux déjà

---

[2] Architecture Harvard : processeur possédant un bus de données et un bus d'instructions séparés.

interfacés avec les protocoles de communication de « CoreConnect ». Il devient donc aussi facile que quelques clics de créer un système contenant un processeur, de la mémoire et une interface d'entrées/sorties, quasi complet et fonctionnel. Évidemment, ces quelques manipulations nécessitent la connaissance préalable des outils de design offerts par ces sociétés. Les outils proposés par Xilinx pour le design de leurs FPGA et CPLD sont décrits dans le chapitre 2 de ce mémoire.

1.1.3. Ce que le futur nous réserve

Il est difficile de concevoir que les « Virtex 2 Pro » puissent finir par être surclassés. Effectivement, avec tous les ajouts récents sur les FPGA, ce que nous réserve le domaine des FPGA dans un futur très proche devient difficilement imaginable.

Du coté de Xilinx, on continue de sortir les dernières sous-familles des « Virtex 2 Pro ». Parmi toutes les puces de Xilinx, ces dernières contiennent plus de logiques programmables, de mémoires, de fonctions DSP (digital signal processing), etc. De plus, une toute nouvelle famille a fait son apparition dernièrement, il s'agit des «Virtex 5 » qui, basé sur une technologie de 65 nm, contient une toute nouvelle architecture constitué de LUT à 6 entrées. Cette technologie, conjuguée a sa nouvelle structure de base et aux nombreux blocs embarqués et fonction DSP ajoutée à la puce, lui permet d'effectuer un éventail de fonctionnalité plus rapidement et avec une moins grande dissipation de puissance [XILI06] par rapport aux autres familles de FPGA que Xilinx possède. Cependant, cette section détaillera plutôt une autre famille tel le « Virtex 4 » [XILI04]. Il s'agit d'une puce permettant trois types d'exploitation. En effet, cette famille comporte trois types de puces permettant de réaliser des systèmes complètement différents. À la fois orientée logique, et orientée système, cette famille de FPGA est sûrement la plus versatile de notre époque. Son coté design composé des puces V4 LX, possédant un très grand rapport de logique programmable versus le nombre de cellules totales, et des V4 SX spécialisées pour les design nécessitant beaucoup de fonctions DSP. D'autre part, elle est également orientée système grâce aux V4 FX qui contiennent un processeur

embarqué et de multiples autres composants à cet effet. La figure suivante illustre très bien la flexibilité de la famille « Virtex 4 ».

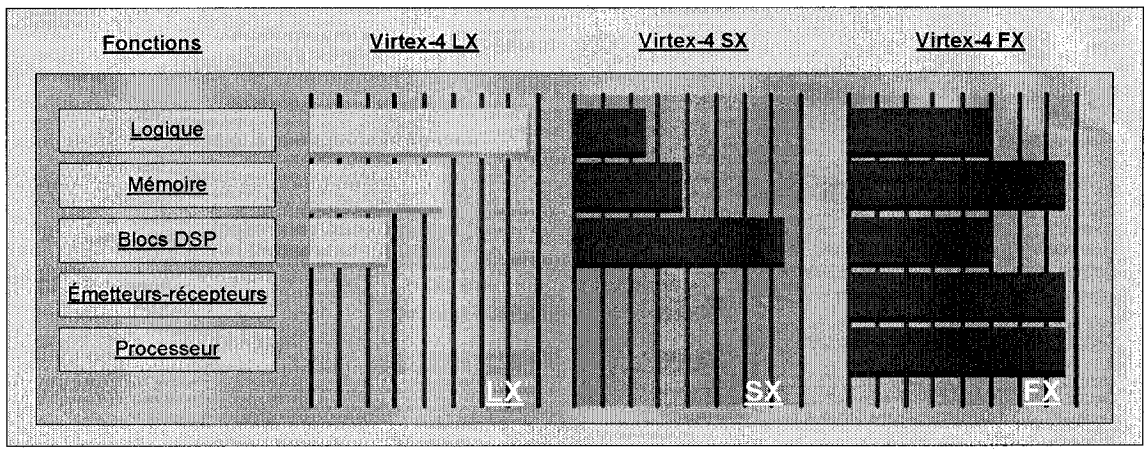

Figure 1.4 : Caractéristiques de la famille Virtex 4 [XILI04]

Altera, pour sa part, semble prendre un tournant différent. Tout en continuant de déployer tous les efforts requis pour continuer à développer ses familles actuelles de FPGA et procurer une impressionnante bibliothèque de IP intégrable a leur puce dont le processeur logiciel Nios II, une branche de cette société se tourne désormais vers les ASIC structurés dont leur premier essai se nomme le « HardCopy ». En résumé, les ASIC structurés consistent en des ASIC configurables où la puce n'est pas entièrement complétée. Il ne reste alors qu'à appliquer quelques couches de métal pour que la puce soit terminée [BEHR03]. En réalité, ces couches de métal permettent d'intégrer le comportement désiré dans la puce. Ainsi, le « HardCopy » permet de joindre la flexibilité à la performance. Effectivement, Altera propose de conjuguer le flot de conception rapide et efficace des FPGA et la performance des ASIC. Le « HardCopy » est basé sur la technologie « Stratix » (une des familles classiques de Altera), ce qui permet à l'entreprise d'utiliser ses outils de conception. Une fois le système entièrement réalisé avec un FPGA « Stratix », Altera s'engage à migrer le « netlist » du système vers un « HardCopy ». De plus, ce composant possède exactement la même empreinte de puce que la famille « Stratix ». Il devient donc facile de les interchanger sur une carte.

Ces quelques étapes de plus procurent un nouveau flot permettant de garder un temps de conception rapide, d'augmenter les performances de 50% et de diminuer la consommation de puissance d'un facteur 40% par rapport à un FPGA Stratix équivalent [ALTE03a].

Les familles d'Altera sont autant diversifiées que celles de Xilinx. Par contre, Altera semble avoir abandonné le concept de processeur intégré au FPGA, comme le PPC dans le V2P, et semble mettre plus d'emphase sur leur concept de ASIC structuré et sur l'évolution de leur bibliothèque de modules logiciels (IP) [ALTE04].

Pour sa part, Actel se lance aussi dans la guerre des modules logiciels, parmi leur bibliothèque se retrouve une version du ARM7 qui, selon cette société, semble être le processeur embarqué le plus utilisé au monde. Évidement, leur bibliothèque procure aussi les modules nécessaires à la communication du ARM7, soit la suite de bus AMBA. Cette bibliothèque conjuguée a une impressionnante liste de FPGA leur donne assurément une place importante de le marché de la puce programmable [ACTE05].

1.2. Processeurs présynthétisés

En plus d'offrir une gamme complète de FPGA et CPLD, Altera et Xilinx offrent aussi des processeurs pré-synthétisés qu'on peut programmer dans leurs FPGA respectifs. Pour Xilinx, il s'agit du « Microblaze », alors qu'Altera propose le « Nios » et le « Nios II ». Nous détaillerons principalement les processeurs « Microblaze » ainsi que le « Nios II » dans les sous-sections suivantes.

1.2.1. Nios II

Le fameux « Nios II » se présente en fait comme une famille de processeurs pré-synthétisés. En effet, un peu de la même façon que Xilinx avec son « Virtex 4 », Altera cherche à élargir le champ d'application de son processeur. Fort de son architecture

RISC, ce noyau de 32 bits se divise en trois processeurs distincts : le « Nios II/e », le « Nios II/s » ainsi que le « Nios II/f » [ALTE04b]

Malgré moult différences, ces trois processeurs distincts possèdent une architecture semblable. En effet, leurs nombreux points communs leur permettent de garder une compatibilité logicielle. La figure suivante illustre le schéma bloc global des « Nios II ».

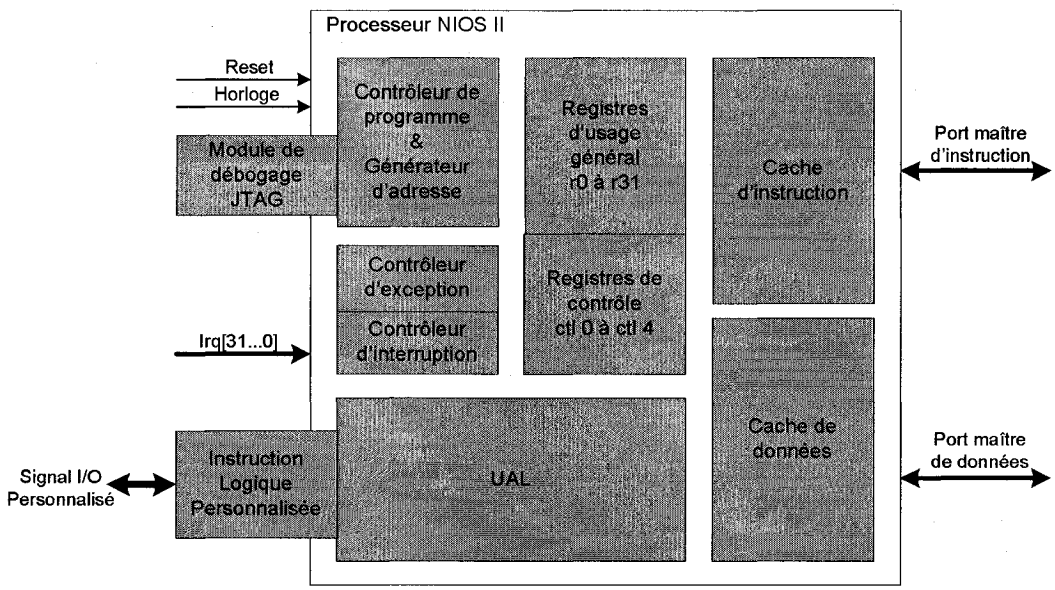

Figure 1.5 : Schéma bloc du Nios II [ALTE04a]

Ces processeurs utilisent un fichier de registres contenant 32 registres d'usage général ainsi que cinq registres de contrôle. Comme la majorité des RISC, chacun de ces registres possède une capacité de 32 bits.

L'unité arithmétique et logique possède un jeu d'instruction riche permettant d'effectuer de nombreuses opérations différentes. De plus, certaines instructions demeurent dépendantes du matériel : elles restent donc non implémentées. Effectivement, dépendamment du noyau choisi, les instructions de multiplication et de division sont simulées de façon logicielle. Afin que cette simulation reste transparente à l'utilisateur, lorsqu'une instruction de ce type est appelée, une exception se lève et la routine de

simulation de l'instruction s'exécute. De plus, chacun de ces processeurs peut utiliser des instructions spécialisées préalablement programmées par l'utilisateur. Effectivement, les outils de développement permettent de créer des instructions matérielles qui seront insérées en parallèle à ALU. Suite à l'utilisation de l'instruction spécialisée, le flot de données sera dévié vers le module matériel. Ainsi, le calcul résultant de l'instruction aura été exécutée par un module généré par le programmeur.

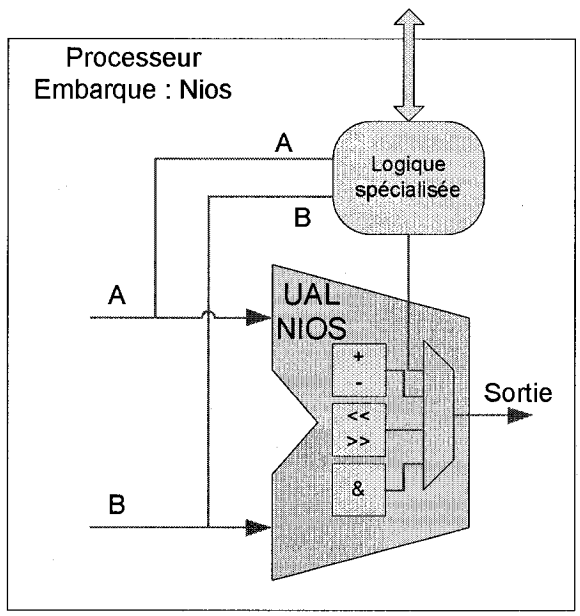

Figure 1.6 : Flot des instructions spécialisées du Nios II [ALTE04]

Toutes les exceptions et les interruptions matérielles restent liées à une seule adresse mémoire. Cette adresse contient généralement une méthode d'aiguillage des évènements. Ainsi, toutes les exceptions et les interruptions finissent par atteindre leur routine prédéterminée.

Comme le schéma bloc de la Figure 1.7 l'illustre, le « Nios II » possède un contrôleur d'interruption intégré pouvant supporter jusqu'à 32 niveaux d'interruptions. De plus, cette architecture supporte les interruptions imbriquées.

Tous les « Nios II » possèdent une architecture « Harvard » : leurs bus d'instruction et de données se trouvent donc séparés. Ces deux bus sont implémentés selon le protocole de bus « Avalon » développé par Altera. Ce protocole sera survolé dans les sections suivantes. Par ailleurs, les processeurs « Nios II » sont « little endian » et permettent des accès par mot ou par demi-mot.

De plus, des caches d'instructions et de données sont disponibles pour certains des trois processeurs. Ces caches se gèrent de façon logicielle, directement à l'aide d'instructions prévues à cet effet. Plusieurs paramètres dont la grandeur des caches restent pourtant configurables par l'utilisateur.

Finalement, les « Nios II » contiennent un module JTAG complet pour le débogage. Ce module permet de :

- transférer du code,
- démarrer et arrêter le code en exécution,
- ajouter des points d'arrêt,
- analyser les registres et la mémoire,
- obtenir des données de traçage en temps réel.

Certains noyaux ne possèdent qu'un sous-ensemble de ces fonctionnalités. Les caractéristiques de chacun des trois processeurs sont présentées brièvement dans la prochaine section, afin de mieux comprendre leurs différences

1.2.1.1. "Nios II/e"

Le premier membre de la famille le «Nios II/e », aussi appelé le « Nios 2 Economy », vise principalement à utiliser le moins de ressources possible pour atteindre le noyau le plus petit possible, tout en gardant sa compatibilité avec les instructions standard des « Nios II ». De ce fait, ce noyau se montre beaucoup moins performant que les autres membres de sa famille, mais il est environ deux fois plus petit que le « Nios II/s ». En

effet, ce noyau peut atteindre 200MHz mais ne peut pas dépasser 31 Dhrystone MIPS. [ALTE04b]

Pour épargner de l'espace et des ressources, tout doit être sacrifié. C'est pourquoi ce noyau ne contient qu'un pipeline d'un seul étage. Il en découle qu'aucune dépendance de données entre les étages n'est possible, et on peut ainsi sauver l'espace dédié pour résoudre les aléas de données.

L'espace adressable par ce processeur est de 2Go. Ce processeur économique ne supporte en aucun cas les caches de données, ni celles d'instructions.

La performance maximale de ce petit processeur est de 6 cycles par instruction, et ce, seulement si le bus d'instruction est assez performant pour récupérer l'instruction courante en un seul cycle.

Pour atteindre ces objectifs, les unités de multiplication, de division et les registres à barillet ne sont pas disponibles dans ce noyau. Par le fait même, les instructions non-implémentées sont automatiquement résolues de façon logicielle, suite à une exception émise par le processeur.

Finalement, l'interface de débogage logiciel via JTAG reste disponible, mais avec des fonctionnalités réduites. Dans cette version minimaliste, le module JTAG ne supporte pas les points d'arrêt matériels, ni le traçage de signaux en temps réel.

1.2.1.2. "Nios II/s"

Le second noyau, appelé « Nios II Standard », a pour objectif de rester petit et performant. Tout comme pour son homologue économique, toute réduction de la taille du processeur se fait au détriment de ses performances. Ce noyau « Standard » utilise approximativement 20% moins de logique programmable que le « Nios /f », mais semble aussi 40% moins performant que celui-ci.

Pour atteindre ces objectifs, seuls les modules avec les plus grands rapport utilisation de ressources versus impact sur la performance on été bannis. Ce type de processeur est parfait pour tout système nécessitant un petit processeur avec des performances moyennes. Malgré tout, ce noyau peut atteindre 127 Dhrystone MIPS à 165 MHz.

Le pipeline de ce noyau ressemble étrangement à celui du classique DLX décrit dans [HEPA90]. Effectivement, ce pipeline contient 5 étages définis comme suit : « Fetch », « Decode », « Execute », « Memory », « Writeback ». Une politique de prédiction de branchement statique est utilisée pour réduire les pénalités dues à une erreur de branchement. Pour plus d'informations sur les politiques utilisées pour le pipeline des « Nios II », le lecteur peut se référer à [ALTE04b].

Comme le « Nios II Economy », l'espace adressable de ce processeur est de 2Go. Par contre, le « NIOS II/s » supporte les caches d'instructions. Ses performances lui permettent également d'exécuter une instruction à chaque 1 ou 2 cycles.

De plus, si le FPGA cible le permet, le processeur supporte les instructions matérielles de multiplication. Par contre, les opérations de division sont toujours simulées en logiciel.

Finalement, le module de débogage via JTAG offre toutes les fonctionnalités dont le débogage logiciel, ainsi que le traçage des signaux matériels en temps réel.

1.2.1.3. "Nios II/f"

Le dernier membre de cette famille de processeur pré-synthétisé demeure le plus performant d'où son nom : « Nios II Fast ». Les objectifs de ce noyau consistent à maximiser le rapport instructions par cycle tout en optimisant sa fréquence maximale d'exécution. Ainsi, ce noyau s'avère parfait pour les systèmes devant traiter une grande quantité de données et dont les performances deviennent essentielles. Il reste à noter que ce processeur peut atteindre 218 Dhrystone MIPS à 185 MHz.

Si le FPGA cible contient les modules DSP nécessaires, toutes les instructions non-implémentées (multiplication, division, rotation et décalage de bits) s'exécutent directement en matériel.

Contrairement au « Nios II/s », le modèle rapide possède un pipeline de 6 étages : « Fetch », « Decode », Execute », « Memory », « Align », « Writeback ». De plus, ce noyau utilise une technique de prédiction de branchement dynamique pour restreindre les pénalités de branchements. Ainsi ce processeur peut exécuter une instruction à chaque cycle ou plus.

Comme les deux autres processeurs de sa famille, l'espace adressable de ce processeur est de 2Go. Pour sa part, le « NIOS II/f » supporte autant les caches d'instructions que les caches de données. La grandeur de chacune de ces caches est configurable par les utilisateurs.

Comme le « Nios II Standard », toutes les fonctionnalités du module de débogage intégré dans le noyau sont disponibles dans la version rapide. Ces fonctionnalités comprennent le débogage logiciel, les points d'arrêt matériel ainsi que le traçage en temps réel.

1.2.2. MicroBlaze

Le « Microblaze » ressemble beaucoup au « Nios II » : c'est un processeur pré-synthétisé performant, configurable à sa manière et offrant diverses possibilités.

Également un RISC à 32-bits, ce processeur est optimisé spécialement pour son intégration dans des FPGA de Xilinx. De ce fait, il peut atteindre une fréquence théorique de 150MHz sur un « Virtex 2 Pro », et de 185 MHz sur un « Virtex 4 ». Comme ce processeur est paramétrable, ses performances peuvent différer dépendemment de la configuration du processeur à générer. Le tableau suivant expose des données obtenues

par Xilinx avec un Microblaze optimisé pour la performance d'une application de recherche de patron. Le processeur utilisé contient un multiplieur intégré, un décaleur à barillet ainsi qu'une instruction spécialisée interfacée par FSL.

Tableau 1.2: Performance du Microblaze [XILI04]

FPGA	Grandeur	Fréquence d'horloge	Dhrystone 2.1	Performance
Virtex-4 (4VLX25-12)	1,269 LUTs	180 MHz	166 DMIPS	0.92 DMIPS/MHz
Virtex-II Pro (2VP20-7)	1,225 LUTs	150 MHz	138 DMIPS	0.92 DMIPS/MHz
Spartan-3 (3S1500-5)	1,318 LUTs	100 MHz	92 DMIPS	0.92 DMIPS/MHz

L'illustration suivante représente un diagramme bloc de ce processeur avec plusieurs des périphériques disponibles dans la bibliothèque d'IP offerte par Xilinx. Ces périphériques sont connectés directement sur le bus OPB, mais toutes ces connexions se font très facilement grâce aux outils fournis. À partir du diagramme, on observe que le processeur suit une architecture « Harvard » avec ses bus de données et ses bus d'instructions séparés. On remarque également les modules optionnels, tels que l'unité de division matérielle, le barillet à décalage, les caches de données et d'instructions disponibles, ainsi que le bus FSL, qui donnent une tout autre dimension à ce processeur.

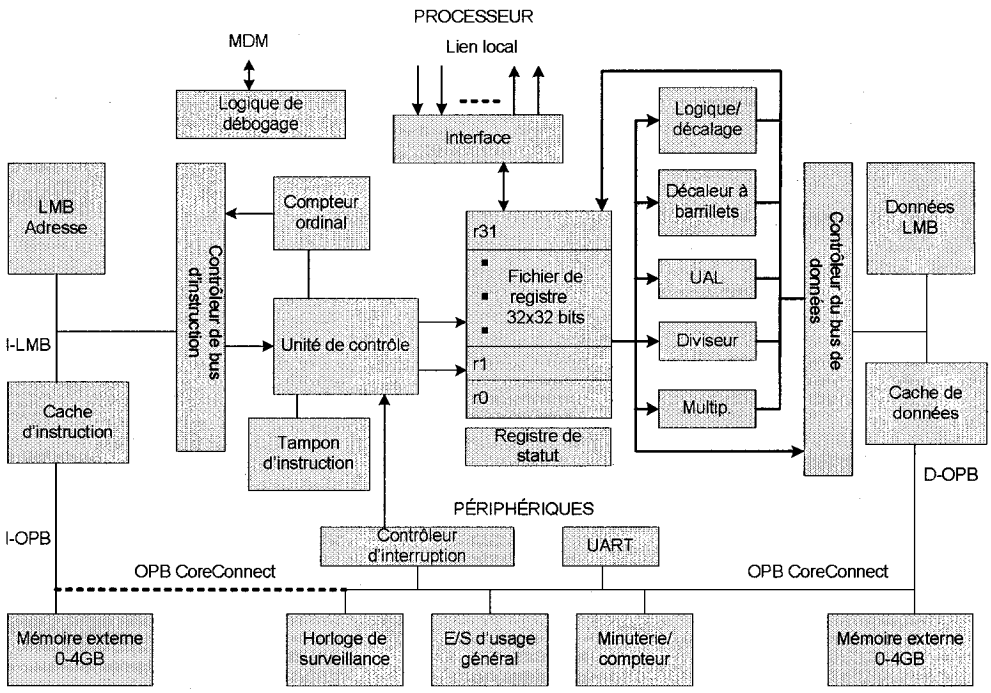

Figure 1.7 : Schéma bloc du Microblaze et de certains périphériques [XILI04c]

Ce processeur possède 32 registres d'usage général, ainsi que deux registres à usage spécialisé de 32 bits de large. Les instructions sont codées également sur 32 bits, et naturellement, ses bus de données et d'instructions sont de cette largeur.

Comme la majorité des RISC, le « Microblaze » est pipeliné. Son pipeline est composé de trois étages (« Fetch, decode, execute »), et deux techniques de réduction de pénalités de branchement sont implémentées, afin d'augmenter les performances d'exécution.

De plus, ce processeur « big endian » possède un jeu d'instructions très complet, permettant même d'utiliser les multiplieurs et le diviseur intégrés dans le FPGA. Ces instructions sont divisées en deux types, soient A et B. Le type A permet d'effectuer des opérations sur deux registres sources, alors que le type B utilise une valeur immédiate et un registre source.

Il est aussi intéressant de remarquer, sur la figure précédente, que le « Microblaze » peut communiquer sur deux types de bus : le LMB, pour « Local Memory Bus », ainsi que l'OPB, pour « On-Chip Peripheral Bus », ce dernier faisant partie de la suite « CoreConnect » de IBM présentée dans la section suivante. Comme son nom l'indique, le «LMB » permet d'interfacer de la BRAM localement au processeur : la latence de communication devient ainsi très faible. Comme il n'y a qu'une faible quantité de BRAM sur des FPGA, ce type de connexion est parfait pour créer un niveau de cache dans un système. On pourrait imaginer un second niveau de cache directement sur l'OPB, qui pourrait être partagée, ainsi qu'une mémoire principale, à l'extérieur du FPGA, à laquelle on accède par un contrôleur branché également sur l'OPB. Il existe six différentes configurations de mémoire possibles avec le « Microblaze ». Il est à noter que chaque configuration doit posséder minimalement une connexion pour les instructions et au moins une pour les données.

Le « Microblaze » possède aussi des contrôleurs de cache pour ses instructions et ses données. Plusieurs paramètres du « Microblaze » sont dédiés à la configuration des caches. Par exemple, la taille de ces caches peut être configurée par l'utilisateur. De plus, ce type de mémoire peut être utilisé soit via l'OPB ou encore via l'interface « CacheLink »[3]. Cette nouvelle interface se connecte par le biais du FSL pour effectuer les opérations de cache.

Comme le montre la figure suivante, le bus « Fast Simplex Link » est en fait une interface permettant de connecter n'importe quel périphérique directement et localement au Microblaze. La communication avec les co-processeurs via FSL se fait facilement grâce à un API écrit en C. Chaque processeur possède 8 entrées et 8 sorties FSL de 32 bits chacune : il est donc possible de créer un coprocesseur FSL pouvant transférer des paquets allant jusqu'à 32 octets. L'avantage de ce type d'ajout, comparativement à un

[3] Le « CacheLink » est seulement disponible dans les versions supérieures à la 3.00a du « Microblaze »

processeur possédant un jeu d'instructions configurable, consiste dans le fait que le co-processeur ne bloque pas l'exécution du « Microblaze » pendant ces opérations. Les performances de l'interface FSL peuvent atteindre jusqu'à 300 MB/sec. Ce type d'interconnexion se montre idéal pour communiquer d'un « Microblaze » à un autre « Microblaze ».

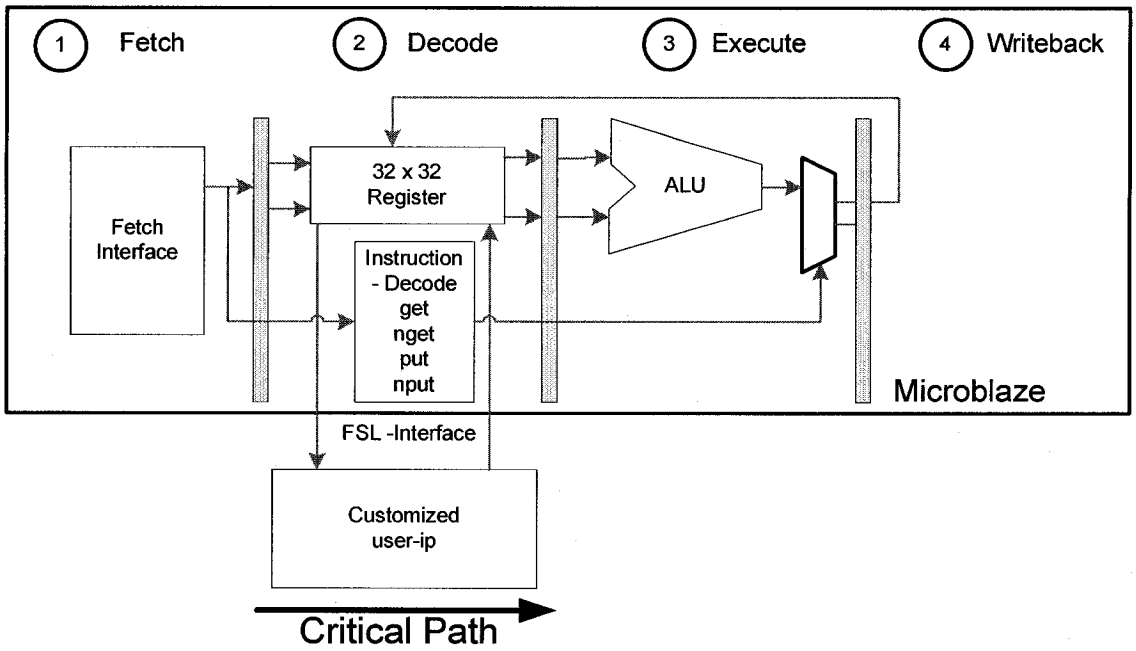

Figure 1.8 : Interface FSL dans le pipeline du Microblaze [XILI04c]

Contrairement au « Nios II », aucun module de débogage n'est directement intégré dans le « Microblaze ». Cependant, une interface JTAG est disponible pour connecter certains modules permettant de parcourir le code en exécutions pas par pas, d'insérer des points d'arrêt, et d'examiner des variables, des registres et des plages mémoires. Un second aspect du débogage est également disponible grâce à cette interface. En effet, l'outil « Chipscope Pro » permet de tracer des diagrammes temporels des signaux internes au FPGA pendant l'exécution du projet. Pour ce faire, quelques modules propres à l'outil doivent être instanciés dans le système.

1.2.3 Comparaison des processeurs présynthétisées

Le tableau suivant compare certaines caractéristiques discutées dans les sections précédentes sur les trois « Nios II » et le « Microblaze.

Tableau 1.3: Comparaison des processeurs logiciels

	NIOS II			Microblaze
	Economy	Standard	Fast	
Max DMIPS	31[4]	127[4]	218[4]	125[5]
F_{MAX}	200 MHz[4]	265 MHz[4]	185 MHz[4]	150 MHz[5]
Surface	< 700 LEs[4]	< 1400 LEs[4]	< 1800 LEs[4]	< 900 LCs[5]
Pipeline	1 étage	5 étages	6 étages	3 étages
ICache	Non	Oui 512o à 64Ko	Oui 512o à 64Ko	Oui 2Ko à 64Ko
DCache	Non	Non	Oui 512o à 64Ko	Oui 2Ko à 64Ko
Multiplication et division	Opérations logicielles	Mult. matérielles[6]	Mult. et div. matérielles[6]	Mult. et div. matérielles[6]
Débogage	Logiciel seulement	Logiciel et matériel	Logiciel et matériel	Logiciel et matériel via modules séparés
Interruptions	Contrôleur d'interruption intégré.	Contrôleur d'interruption intégré.	Contrôleur d'interruption intégré.	Contrôleur d'interruption séparé.
Options	Instructions spécialisées	Instructions spécialisées	Instructions spécialisées	FSL

[4] Calculé avec les Stratix les plus rapides [ALTE04b]

[5] Calculé avec les Virtex 4 les plus rapides [XILI04]

[6] Si le FPGA cible possède les fonctions DSP nécessaires. [ALTE04b] et [XILI04]

1.3. Interconnexion

Afin que ces processeurs, autant les pré-synthétisés que les autres, puissent communiquer avec le reste des modules présents dans le système, il semble important de choisir un protocole de communication adéquat. Actuellement, sur le marché des systèmes-sur-puce, les structures de bus représentent le moyen le plus répandu pour communiquer entre un processeur et ses périphériques. Évidemment, on aurait pu choisir tout autre réseau sur puce (NOC) plus efficace pour l'application utilisée dans ce projet [DESL05], mais le but du projet réside dans l'aisance de la création d'une architecture efficace, et les protocoles de bus se montrent plus répandus et souvent offerts avec les outils de développement des FPGA. De plus, la majorité des modules fournis possèdent déjà une interface compatible avec certains protocoles de bus. Le choix du FPGA n'est donc pas seulement un choix de technologie physique mais aussi le choix du processeur et de ses interconnexions. Bien entendu, il reste possible de déroger à la tendance d'utiliser les interfaces de communication, les périphériques et le processeur généralement optimisé pour un FPGA, mais le gain possible ne vaut pas les complications attendues. Les structures de bus suivantes auraient pu être utilisées dans le projet : AMBA, CoreConnect, Avalon

1.3.1. AMBA

Conçu par ARM, le protocole AMBA [ARM04] est une spécification de bus de communication pour système sur puce que l'entreprise propose comme norme pour les SOC. L'entièreté de cette spécification comprend trois protocoles qui permettent une utilisation différente, dépendamment des besoins du système. Le plus connu des protocoles AMBA est le AHB « Advanced High-Speed Bus ». Celui-ci est particulièrement utilisé pour créer des systèmes très performants contenant généralement des processeurs rapides, des DMA, et des DSP. Le ASP, quant à lui, représente l'acronyme de « Advanced System Bus » et se trouve actuellement remplacé par le AHB. Le dernier, l'APB, pour «Advanced Peripheral Bus », permet l'utilisation de

périphériques et modules plus lents. Son protocole, plus simple que les deux autres, rend aisée la génération de nouveaux modules, son interface étant plus facile à comprendre et à tester.

Ce protocole AHB, largement utilisé, possède plusieurs caractéristiques intéressantes, comme le support de plusieurs maîtres, les transactions en rafales et les transactions partagées dites « split ». Il offre aussi un mécanisme de protection des accès privilégiés ainsi qu'un algorithme d'arbitrage pour éviter les conflits sur des système multi-maîtres. Un pont AHB-APB reste disponible pour assurer la communication entre les modules à haut débit et les périphériques plus lents.

AHB dispose aussi d'un nouveau modèle qui se nomme AHB-CLI pour « cycle level interface », qui consiste en fait en un modèle RTL pour SystemC du protocole de bus.

1.3.2. CoreConnect

Très semblable à AMBA, cette spécification est aussi divisée en trois protocoles distincts. Le premier nommé PLB pour « Processor Local Bus », se montre semblable au AHB d'AMBA : il est utilisé dans des systèmes à haut débit et pour séparer la majeure partie des périphériques du processeur utilisé. Souvent en conjonction avec sont homologue, l'OPB, « On-Chip Peripheral Bus », s'avère beaucoup plus simple à comprendre et les modules qui lui sont connectés sont plus faciles à tester et à intégrer au système. Ce protocole est utilisé pour contenir les périphériques lents, comme le APB de AMBA. Finalement, le dernier des trois protocoles offerts par CoreConnect se nomme le DCR, pour « Device Control Register ». Celui-ci se connecte en anneau avec le PowerPc, permettant d'aligner plusieurs esclaves en série sur cette interconnexion. Ce type de connexion semble intéressant pour les modules très lents : il est principalement utilisé pour connecter les contrôleurs d'interruptions au processeur.

Pour sa part, le « Microblaze » possède une interface qui n'est compatible qu'avec l'OPB. Par contre, le PowerPc 405 de IBM utilise une interface PLB. Certains considèrent CoreConnect comme trop lourd et trop compliqué pour intégrer à un SOC [USSE01]; Xilinx, par contre, tente de contrer cette croyance en intégrant ce protocole et une quantité intéressante d'IP compatibles à ses outils de développement de FPGA. Effectivement, CoreConnect, avec son Microblaze, peut être instancié sur tous les FPGA de Xilinx possèdant assez de ressources configurables.

Puisque la création d'interfaces pour ces protocoles reste souvent inévitable et fastidieuse, IBM offre certains outils de vérification et de génération de tests, afin d'assurer la compatibilité du protocole. Ainsi, il devient plus simple de connecter son propre périphérique au système.

1.3.3. Avalon

Le dernier protocole présenté est particulièrement simple. En effet, cette particularité est l'objectif principal que les architectes ont voulu atteindre. Effectivement, ce protocole de bus, créé par Altera et complètement dédié aux systèmes utilisant un « Nios » ou un « Nios II », vise principalement à rester facile d'apprentissage pour les concepteurs de systèmes sur puce. De plus, il est optimisé pour l'utilisation de « LE », principal élément dans les FPGA de Altera.

Comme les deux premiers protocoles, il se base sur des transactions maîtres-esclaves et peut supporter plusieurs maîtres. Afin d'éviter les problèmes d'accès simultanés, un arbitre esclaves est directement intégré dans le protocole. Ce type d'arbitrage détermine quel maître peut communiquer un esclave si les requêtes des maîtres sont émises simultanément pour un même module.

Finalement, les outils d'Altera procurent un assistant permettant de générer facilement et automatiquement le bus à partir des paramètres spécifiés par l'utilisateur. [ALTE04a]

1.4. Multi-Processus matériel.

Depuis plusieurs décennies, des ingénieurs du monde entier tentent, en vain, de maintenir la cadence de développement face à rapidité de l'évolution de la loi de Moore au niveau de la performance des systèmes informatiques. De ce fait, plusieurs recherches portant sur des architectures multiprocesseurs émergent chaque année. Ces systèmes à grande échelle utilisant des mémoires partagées génèrent des latences de communications de plusieurs dizaines, voire centaines de cycles [GHGM91, KHKA92]. Ce problème de latence peut facilement faire écrouler les performances du système utilisé. Pour remédier à cet obstacle, plusieurs mécanismes sont encore étudiés. Parmi ceux-ci on retrouve notamment les protocoles de cohérence de caches [GHGM91, BORA92, TUEG93], la prélecture de données [GHGM91, BORA92, TUEG93] ou encore des architectures basées sur les flots de données [GHGM91, PDWO92, HAFU88, KHKA92, BORA92].

Toutes ces techniques comportent leurs avantages et leurs inconvénients. En fait, elles peuvent réduire significativement la latence de communication, sans toutefois l'éliminer. Individuellement, ces mécanismes peuvent produire globalement des gains variant d'un facteur 1 à 2. Cependant, la meilleure solution consisterait en une composition de plusieurs techniques. Effectivement, une combinaison de mécanismes de réduction/masquage de la latence peut générer une augmentation de performance d'un facteur 4 à 7, dépendamment de l'application utilisée [GHGM91].

Le but ultime de ces mécanismes consiste à augmenter le taux d'utilisation des processeurs inclus dans les systèmes [EEL98]. Les prochaines sections présentent d'ailleurs des mécanismes tirés des architectures orientées données.

1.4.1. Processeurs RISC

Au début des années 80, Berkley, Stanford et IBM se sont penchés, chacun de leur coté, sur une nouvelle architecture afin de faciliter le support de langages sur le matériel. De ces recherches, il en résulte, quelques années plus tard, l'apparition de l'architecture RISC [HEPA90].

Cette philosophie architecturale se base principalement sur la simplification des instructions. En effet, seules les instructions « load » et « store » utilisent la mémoire, toutes les autres instructions travaillent de registres à registres. Cette technique simplifie le décodage et uniformise le format des instructions [PATT85]. De plus, cette architecture utilise un pipeline afin que les instructions se recouvrent pendant leur exécution. Ce mécanisme augmente ainsi le taux d'utilisation du processeur en exécutant une instruction par cycle [HEPA90]. Cependant, ces performances ne s'atteignent que dans les conditions idéales : un code possédant le moins de branchements possible et très peu de dépendances de données. Si un seul de ces deux facteurs est présent, des bulles s'installent dans le pipeline et diminuent le nombre d'instructions par cycle. Dans les processeurs actuels, plusieurs solutions sont apportées pour éviter les bulles ou pour diminuer l'impact des pénalités dans le pipeline, dont la prédiction de branchement.

Parmi les processeurs embarqués, le ARM, le PowerPc, les « Nios I/II » et le « Microblaze » font partie des noyaux dit RISC.

Évidemment, ces processeurs ne possèdent aucun mécanisme matériel de multi-processus. Pour y arriver, ces processeurs doivent utiliser des bibliothèques logicielles prévues à cet effet. De plus, aucun parallélisme matériel ne s'exploite dans ce type de processeurs.

1.4.2. Processeurs superscalaires

Au milieu-fin des années 80, on remarque l'apparition des premiers prototypes de processeurs superscalaires. Certains y voient une évolution directe des processeurs RISC [SMSO95]. Cette architecture possède la capacité d'initier plusieurs instructions par cycle [LEEL97,SMWE94,SMSO95,LOBA96]. Intégré pour la première fois au sein d'un processeur, ce type de parallélisme est appelé ILP « instruction level parallelism » ou encore parallélisme horizontal. Évidemment, pour accomplir cet exploit, ce type de processeur doit analyser les dépendances de contrôles et de données, exécuter les instructions indépendantes parallèlement et remettre le tout dans l'ordre, comme si l'exécution se déroulait séquentiellement [SMSO95].

Parmi les processeurs sur le marché, l'ALPHA 21064 et le PowerPc 601 [SMWE94] se conforment à ce type d'architectures.

1.4.3. Multi-processus Matériel

Le multiprocessus matériel se voit comme une autre alternative au parallélisme permettant de masquer ou tolérer la latence [GHGM91, BORA92, EEL98]. En effet, tel qu'illustré à la Figure 1.9, cette technique permet de changer de flots d'exécution quand le processus courrant se bloque. Cette technique aussi nommée HMT, ou « Hardware Mutithreading », semble généralement réalisée par la duplication des registres nécessaires à l'exécution d'un processus [BYHO95]. Cet ensemble de registres, appelé contexte matériel d'un flot, permet de garder l'état courrant d'un processus à tout moment, et ce, en conservant généralement le registre de statut, le compteur ordinal ainsi que l'état des registres d'usage général [BYHO95, HAFU88]. Ce type de parallélisme au niveau des flots s'appelle le TLP « Thread Level Parallelism », ou parallélisme vertical.

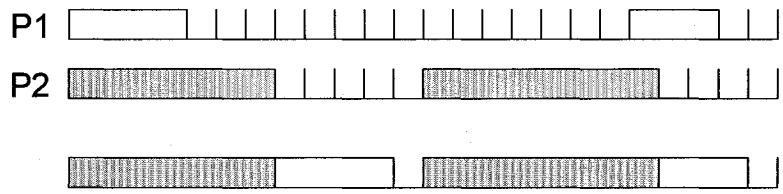

Figure 1.9 : Cacher la latence

En pratique, au moins un flot d'exécution logiciel se joint à un contexte matériel. Cette technique permet de réduire drastiquement le coût des changements de contexte, et ainsi d'obtenir un système plus performant [BYHO95]. On constate qu'il peut y avoir deux niveaux d'abstraction de changement de contexte. Effectivement, si on dispose de plusieurs flots logiciels pour un contexte matériel, on obtient des changements de contextes logiciels pour accéder aux contextes matériels, ce qui permet un système très flexible tant du coté matériel que logiciel.

Le multiprocessus matériel tente d'augmenter le taux d'utilisation du processeur. La base de cette métrique s'avère simple; ce taux processeur se traduit par l'équation suivante : U = C/T [EEL98], où C est le temps de calcul utile, alors que T est le temps total. On peut donc séparer T en trois parties (T = C+I+S): C pour le temps de calcul, I pour le temps d'inactivité et S pour le temps de changement de contexte. D'après cette relation, on remarque que plus I et S sont petits, plus notre système devient performant. L'objectif vise à diminuer au maximum I, sans toutefois hausser la valeur de S. Pour ce faire, il est important de bien choisir le type de multiprocessus matériel. En effet, on peut distinguer deux types de HMT : celui à grains fins et celui à gros grains.

1.4.3.1. Grains fins

Qualifié de multiprocessus parrallèle par [HKNM92] ou de modèle « switch-every-cycle » [BORA92], ce mécanisme est le premier à avoir vu le jour. En effet, certaines machines populaires comme le HEP [SMIT81], et le MASA [HAFU88] utilisent ce type

de HMT. Les systèmes à grains fins changent de contexte périodiquement, en général, à chaque cycle [HAFU88, BORA92, HKNM92, SMIT95, EEL98, BYHO95]. Ces noyaux ont donc souvent autant de contextes matériels que de flots logiciels et sont généralement associés un à un [BYHO95]. Ce type de système doit posséder un bon degré de parallélisme et surtout un coût de changement de contexte extrêmement léger, voire nul. Par contre, le coût se situe dans le matériel nécessaire pour supporter tous ces contextes. De plus, si le système veut masquer de longues latences, il faut autant de contextes que de cycle d'attente. Par exemple, le HEP peut cacher une latence de 128 cycles car il contient 128 contextes matériels [SMIT95].

Un des avantages très intéressant de ce type de HMT se situe au niveau de l'ordonnancement. Effectivement, comme le contexte change à chaque cycle, aucun mécanisme pour la dépendance de données, ni de détection de blocage dû au branchement ne doit être considéré dans le pipeline. Ainsi, l'ordonnanceur devient extrêmement facile à réaliser [EEL98, BYHO95]. La figure suivante illustre une version simpliste d'un ordonnanceur pour multiprocessus matériel à grains fins. La sélection pourrait se faire à l'aide d'un simple compteur se limitant au nombre de contextes matériels supportés.

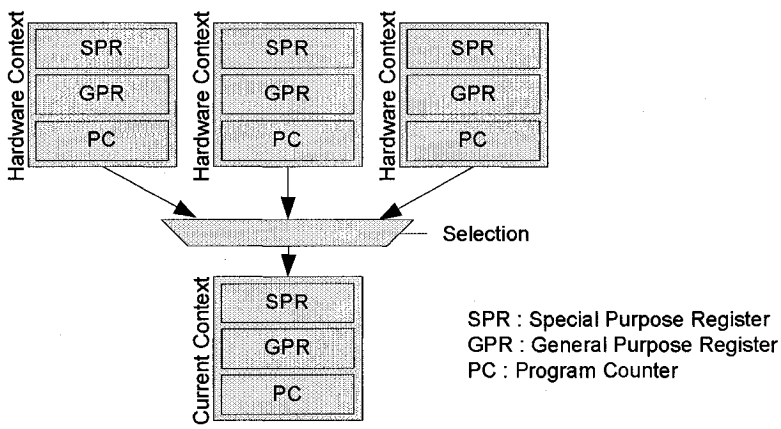

Figure 1.10 : Ordonnanceurs à grains fins.

1.4.3.2. Gros grains

Le multiprocessus concurrent [HKNM92] possède une multitude de mécanismes possibles pour effectuer ses changements de contexte. Par opposition aux grains fins, ce type de HMT change de contexte au moment jugé approprié. Plusieurs modèles sont proposés dans [BORA92], afin de trouver les meilleurs instants pour changer de contextes. Parmi ceux-ci figurent les modèles « switch-on-load » et « switch-on-miss » qui suggèrent respectivement de changer de contexte lorsqu'une instruction « load » est initiée ou bien lors d'un défaut de cache [BORA92]. Le processeur RISC APRIL [ALKK90] figure parmi les architectures qui supportent le HMT à gros grains. Effectivement, ce noyau peut supporter 4 processus exécutant des rafales de 50 à 100 cycles suivies d'un changement de contexte consommant généralement 4 à 10 cycles d'horloge.

Ce type de HMT tente de diminuer au maximum le nombre de changements. De ce fait, ils peuvent se permettre d'avoir des changements plus lourds que les systèmes à grains fins. Les systèmes qui utilisent le HMT à gros grains ont généralement un degré de parallélisme moins important que ceux à grains fins. Cependant, ils nécessitent beaucoup moins de matériel étant donné qu'ils possèdent généralement moins de contextes matériels. Comme les processeurs à pipeline, ces noyaux doivent s'attendre à obtenir des pénalités dues aux aléas de données et aux branchements [BYHO95].

1.4.4. Multiprocessus avancés.

Dans sa forme primitive, le multiprocessus procure du parallélisme de type TLP. Ce type de concurrence permet de diminuer les trous dans les pipelines classiques en changeant de contexte losque l'unité d'exécution courante se bloque. De ce fait, on dit que le HMT s'attaque au gaspillage vertical dans le pipeline [LEEL97, TEL95]. Le pipeline

superscalaire, quant à lui, permet de réduire le gaspillage horizontal en augmentant le parallélisme au niveau instruction (ILP).

Le multiprocessus superscalaire combine les deux mécanismes de concurrence et offre ainsi la possibilité de réduire autant le gaspillage vertical qu'horizontal. Ce mécanisme, fusion de deux techniques éprouvées, offre la possibilité à un seul processus d'exécuter plusieurs instructions en un cycle dans la mesure où celles-ci ne possède pas de vraies dépendances de données entre-elles. Cette amélioration semble très intéressante. Cependant, pour devenir vraiment optimale, il faut que les processus présents dans le système possèdent un très bon taux de concurrence afin d'utiliser le plus grand nombre d'unités fonctionnelles du pipeline superscalaire.

Le multiprocessus simultané se qualifie comme un dérivé plus performant que la version superscalaire. En effet, cette technique permet de remplir les unités fonctionnelle du pipeline superscalaire par les instructions des processus prêt a s'exécuter. Ainsi, au même cycle, il est possible d'exécuter plusieurs instructions de plusieurs processus, et ce, de façon concurrente. De ce fait, le gaspillage horizontal réduit de façon optimale tandis que la partie multiprocessus se charge du vertical [TEL95, LEEL97]

CHAPITRE 2

Ressources logicielles et matérielles utilisées

Afin de réaliser le modèle de multi processus désiré, il paraît important de l'intégrer dans un environnement adéquat. Effectivement, ce projet nécessite une quantité minimale de modules afin que le modèle voulu puisse être fonctionnel, stable, simulable et implémentable. En ce sens, ce chapitre exposera le choix des ressources matérielles, ainsi qu'un détail de leur utilité dans le système. De plus, une brève revue des ressources logicielles utilisées pour la création d'un environnement de développement et de simulation sera expliquée vers la fin de cette section.

2.1. Choix architecturaux

Étant donné que le projet vise une plateforme FPGA, il semble important de faire des choix architecturaux judicieux, afin d'assurer une compatibilité matérielle et logicielle. En effet, chacun des choix influence les suivant de façon significative.

2.1.1. Plateforme reconfigurable

Pendant la période correspondant au début du projet, les plus performants et les plus intéressants FPGA disponibles sur le marché constituaient une toute nouvelle et ingénieuse famille de Xilinx. Cette famille qui sortait de l'ordinaire avec ses ajouts incroyables, dont le PowerPc directement intégré au sein de la puce, est le Virtex 2 pro. De ce fait, la plateforme configurable à été choisie dans l'optique d'y intégrer un tel FPGA. Pour leur compétence et leur expertise avec les puces de Xilinx, le choix de la plateforme configurable pour ce projet s'est arrêté sur un produit qu'offre la société Amirix[AMIR04]. En effet, la plateforme AP100[AMIR05] développée par cette société satisfait parfaitement les exigences de ce projet.

Effectivement, ce projet ne nécessite que très peu d'éléments sur la carte de développement, dont un port série, de la mémoire, quelques « leds », des boutons pour faire des « reset », et une interface JTAG pour l'envoi de la configuration et le débogage. Comme cette plateforme nous offre beaucoup de ressources, elle pourra être facilement utilisée pour d'autres projets ou pour les changements et ajouts à venir pour le projet courant. Cependant, il est important de noter que cette carte possède deux holorges indépendantes, une s'exécutant a 40MHz alors que l'autre est à 80MHz. De ce fait, si une autre fréquence est voulue, il s'avère nécessaire d'utiliser un DCM pour l'obtenir. Le restes des détails sur la description de la cartes et des ces composants sont disponibles à la référence [AMIR05].

2.1.2. Processeur

Parmi les processeurs disponibles pour ce projet, le ARM, le Nios, le Power PC, ainsi que le Microblaze ont été étudiés, afin d'utiliser le processeur le mieux adapté au projet. Cependant, suite au choix du FPGA, il serait non fondé d'utiliser un processeur que Xilinx ne supporte pas. De ce fait, le choix se restreint désormais au PowerPc d'IBM et au Microblaze de Xilinx. Il est toutefois à noter que le projet nécessite une architecture basée sur 4 processeurs. Ainsi, considérant que les Virtex 2 Pro disponibles pour le design ne contiennent que 2 PPC 405 [XILI05], le Microblaze se montre préferable, et ce malgré la performance supérieure du PPC. En ce sens, le choix devient donc le Microblaze : processeur RISC 32 bits de Xilinx, construit sur une architecture Harvard. Les impacts de ce choix semblent mineurs a priori. S'il est vrai que ce choix restreint l'aspect des communications ainsi que celui de la vitesse d'exécution, sa maigre utilisation de CLB permet une multitude d'options intéressantes.

2.1.3. Communication

Les technologies étant fortement liées, le choix des communications au sein de la puce devient très simple. Effectivement, IBM procure des IP des différents bus du protocole

CoreConnect accessible dans l'outil de conception de Xilinx. De ce fait, ce serait autant une perte de temps que d'argent de tenter d'utiliser d'autres protocoles que ceux distribués avec le FPGA. Comme le choix du processeur s'est arrêté sur le Microblaze et qu'il possède également une interface OPB, le choix le plus judicieux pour le bus est donc l'OPB.

Malgré l'avantage de la compatibilité, il reste important de noter que ce bus n'est pas le plus rapide. Effectivement, comme son nom complet l'indique « On-Chip Peripheral Bus », ce bus sert généralement à relier les périphériques plus ou moins lents du système. De ce fait, une transaction typique entre un processeur (maître) et sa mémoire (esclave) prend généralement 3-4 cycles d'horloges dépendamment du protocole d'arbitrage.

2.1.3.1. Transactions & arbitration

Afin de bien comprendre cette latence de 3-4 cycles pour une seule opération de lecture ou encore d'écriture, les transactions typiques d'un système basé sur les communications via l'OPB sont détaillées dans la sous section suivante. Cependant, avant d'aborder ce type de transactions sur un bus OPB, il est important de connaître le protocole d'arbitration présent dans ce type de communication.

Protocole d'arbitration typique

Contrairement aux autres bus (APB, wishbone) [USSE01], l'OPB possède un arbitre directement intégré au sein de son module de communication. La raison de cet ajout réside dans le fait que le processeur le plus souvent présent sur ce type de communication suit une architecture Harvard. En effet, comme mentionné précédemment, le Microblaze possède deux interfaces maîtres compatibles avec les signaux de l'OPB. Ces deux maîtres sont normalement transparents à l'utilisateur, tout comme l'arbitre interne à ce bus qui gère l'accès de ces deux instances au bus. Dans un système normal basé sur l'utilisation d'un Microblaze, où les seuls maîtres sont les bus de données et

d'instruction, les transactions ressemblent généralement au scénario présenté à la figure suivante.

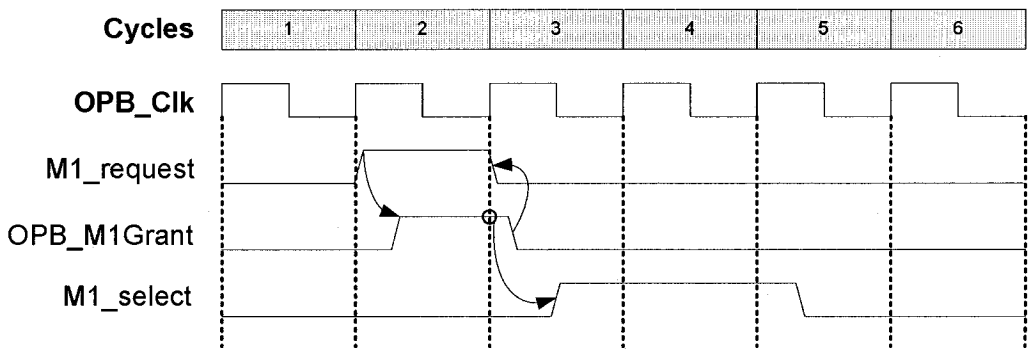

Figure 2.1 : Protocole d'arbitration typique de l'OPB [IBM02]

Cette figure illustre le protocole d'arbitration utilisé par le bus OPB. Ainsi, sur la Figure 2.1, on remarque que ce protocole suit les étapes suivantes :

1. Le maître initie la transaction en montant son signal REQUEST à 1 pour signaler son intention de débuter une requête.
2. L'arbitre reçoit ce signal et monte un signal GRANT (propre à chaque maître) qui désigne le maître choisi pour la transaction en cours.
3. Chaque maître lit son signal GRANT. Seul celui avec une valeur 1 peut débuter sa transaction.

Dans le cas où plusieurs maîtres utilisent le même bus, celui qui effectue la transaction en cours peut garder sont signal REQUEST à 1 aussi longtemps que l'arbitre lui envoie des GRANT. Si un maître qui a une priorité plus élevée demande à faire une transaction sur ce bus, l'arbitre accordera l'autorisation au maître prioritaire.

Plusieurs options de cet arbitre sont configurables, cependant aucune d'entres-elles ne sera détaillée dans le projet actuel.

Transaction typique

Maintenant que nous avons discuté que les maîtres peuvent acquérir le droit d'effectuer des transactions sur un bus partagé, les échanges typiques entre un maître et un esclave peuvent être détaillés. De ce fait, cette sous-section illustre une transaction typique sur un canal de communication OPB, ainsi que 2 cas spéciaux auxquels ce projet fait référence dans les prochaines sections.

Pour commencer, les transactions de bus sont généralement semblables d'un protocole de bus à l'autre. Effectivement, comme les interfaces de bus se ressemblent, il semble normal que les échanges typiques entre un maître et un esclave sur l'OPB utilisent les mêmes types de signaux que dans les protocoles de bus tels que AMBA, Wishbone, ... [USSE01]. Dans l'exemple suivant, un simple transfert de données sera utilisé pour illustrer cet échange. Dans la figure suivante, un maître initie la transaction en effectuant les étapes nécessaires pour obtenir le droit d'émettre des signaux sur le bus.

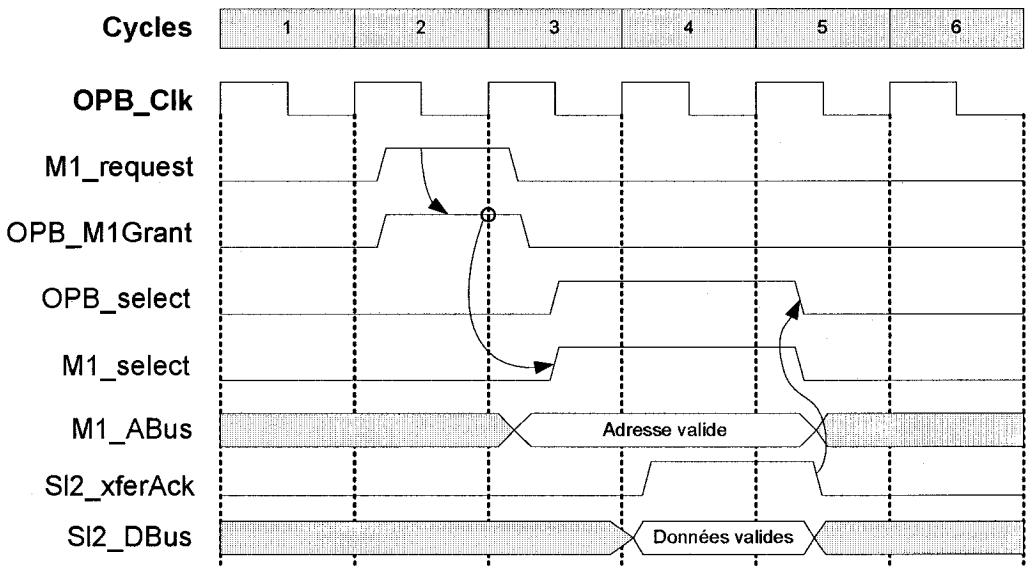

Figure 2.2 : Transaction de bus typique [IBM02]

Une fois le protocole d'arbitration terminé, et le droit d'accès au bus acquis, le maître choisi par l'arbitre incrémente le « M_SELECT » pour acquérir son droit de transiger, et dépose sur le bus :

- l'adresse de l'esclave à contacter « M_Abus »,
- un signal signifiant si la requête est une lecture ou une écriture « M_RNW »,
- la donnée en cas d'écriture « M_DBus »

Une fois ces étapes accomplies, l'esclave convoqué effectue la transaction (écriture ou lecture) et répond à l'aide des signaux suivants :

- la donnée lue (en cas de lecture) « Sl_DBus »,
- une confirmation de fin de transaction « Sl_XferAck » (en cas de réussite seulement)

Certains autres signaux moins importants font aussi partie de la transaction, mais ils ne sont pas nécessaires à la réussite de celle-ci

Quelques cas spéciaux

Outre les transactions typiques, les modèles de bus contenus dans la suite d'outils CoreConnect possèdent aussi des signaux conçus pour la détection et la correction d'erreurs qui pourraient survenir dans le design. Cependant, certains scénarios d'erreurs élaborés dans le cadre d'un système normal ne sont pas spécialement adaptés au type de système conçu dans ce projet. Bien que celui-ci ne contienne aucun mécanisme permettant la gestion d'erreur, il faut s'assurer que le comportement normal dans ce projet n'induise aucun problème ou erreur dans le bus qui se considère dans un système typique. Effectivement, il y a au moins deux cas problématiques qui démontrent une incompatibilité dans la gestion d'erreur pour notre système.

Transaction expirée

Le premier de ces deux cas apparaît lorsqu'on ajoute des latences au système. Effectivement, afin de démontrer que ce système est robuste et efficace, il semble important d'utiliser de longues latences de communication. Or, l'OPB génère un signal d'expiration (« OPB_timeout ») lorsqu'aucune réponse n'a été engagée après un délai de

15 coups d'horloge. La figure suivante illustre justement ce comportement. En effet, on remarque qu'après un délai de 15 coups d'horloge, le signal d'expiration du bus monte à la valeur logique 1. Dans un système typique le module maître va réagir en annulant la requête. Cette annulation est générée en mettant le signal de sélection du module maître («M1_select ») et du bus («OPB_select ») à 0.

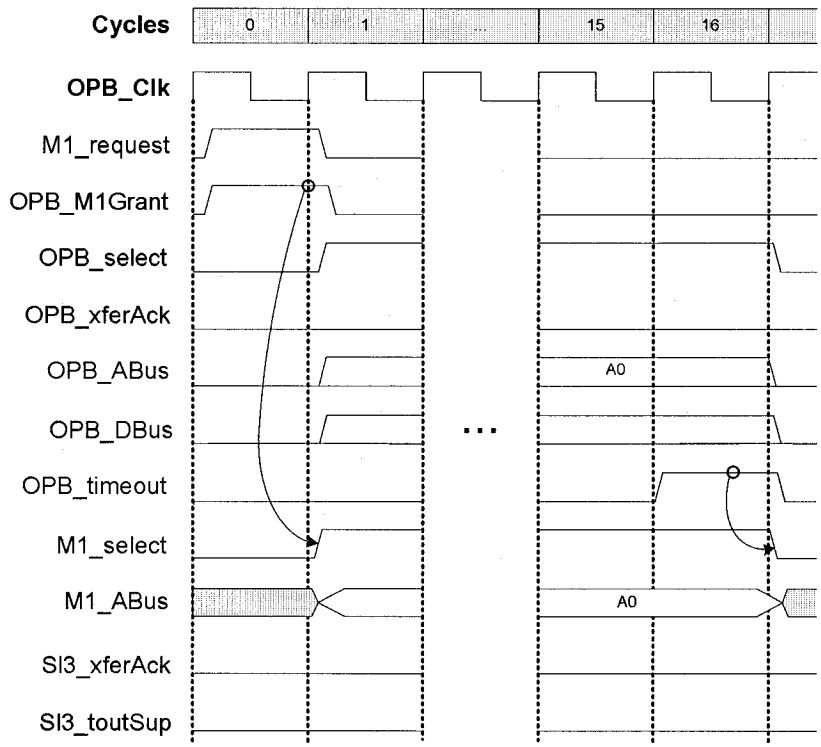

Figure 2.3 : Transaction trop longue : génération d'un timeout [IBM02]

Pour éviter tout problème, plusieurs solutions sont possibles :

1) Ce comportement peut être modifiable étant donné que les maîtres impliqués dans le système ont été interfacés par un module propriétaire. Il est donc facile de modifier le comportement de ceux-ci lors de la réception d'un signal d'expiration. Le comportement le plus simple serait simplement d'ignorer le signal. Malheureusement, si un autre maître non-propriétaire s'ajoute au design, les comportements face à ce signal peuvent être problématiques.

2) Les esclaves ont des mécanismes qui permettent d'avoir des transactions plus longues que 15 cycles. Ils utilisent le signal « toutSup », aussi représentée sur la figure, qui permet d'indiquer au maître de ne pas tenir compte du signal d'expiration, et ainsi aucun problème ne devrait arriver. Bien que cette solution soit la plus élégante parce qu'elle est tout-à-fait conforme au protocole, l'utilisation de celle-ci nécessite plus de logique dans l'interface du processeur et dans chaque esclave.
3) Tout simplement tenter d'obtenir des latences de moins de 15 cycles.

Étant donné, que ce projet ne nécessite pas spécialement de très longs délais, la solution qui préconise l'utilisation de latences de moins de 15 cycles seulement est utilisée. De plus, l'interface du « Microblaze » ignore le signal d'expiration pour éviter qu'une erreur inconnue ne survienne.

Transaction annulée

Le second cas à considérer est beaucoup plus important qu'une transaction expirée car il est directement en conflit avec une des règles que ce projet suit. En effet, la façon utilisée pour annuler une transaction, selon le protocole que suit l'OPB, consiste à mettre le signal de sélection à 0 avant que la requête ne soit complètement terminée par l'esclave. Ainsi tous les esclaves fournis avec les outils de Xilinx et leurs bibliothèques annulent leurs transactions quand ce signal descend à 0. La Figure 2.4 illustre ce scénario, on remarque que suite à la désassertion du signal de selection, l'esclave n'envoi pas de signal de réponse obtenue (« xferAck ») indiquant que la transaction a été réussie.

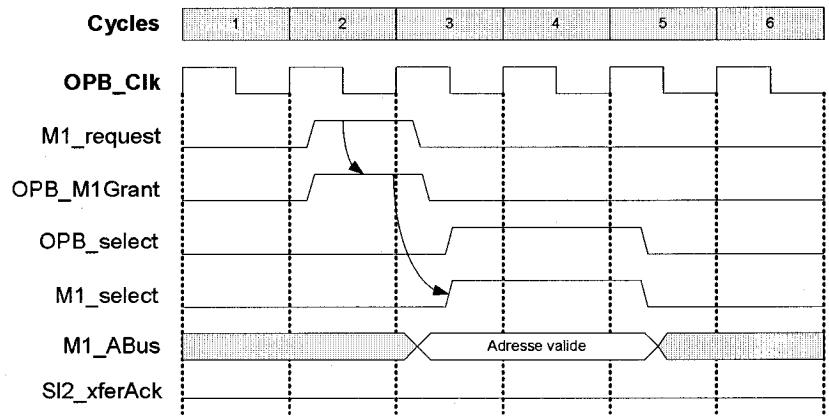

Figure 2.4 : Annulation de requête [IBM02]

Le problème de ce scénario provient du fait que le système de multiprocessus matériel utilisé dans ce projet doit effectuer une transaction par cycle. De ce fait, le select ne doit absolument pas durer plus d'un cycle. Ainsi, lorsque le maître remet ce signal à 0, les esclaves croient que celui-ci vient, tout simplement, d'annuler la requête demandée.

Ce cas très problématique doit absolument être tenu en compte pour s'assurer du bon fonctionnement du système. Pour ce faire, des modifications doivent être apportées aux esclaves afin qu'ils ne considèrent pas ce scénario comme un cas problème, mais plutôt comme un cas typique. Ces modifications peuvent être retrouvées dans la section 3.3.4.

2.2. Outils logiciels

Cette sous section décrit principalement mais brièvement les multiples logiciels utilisés pour obtenir un environnement de travail pour l'élaboration du projet. La majorité de ces outils sont directement reliées aux choix des ressources matérielles, étant donné qu'ils sont principalement développés et distribués avec le FPGA intégré sur la plateforme cible. D'autres outils de tierces parties auraient pu être utilisés et explorés. Cependant, la quantité de travail pour adapter le matériel au logiciel reste significativement plus grande que l'utilisation d'applications propriétaires.

2.2.1. Xilinx EDK ("Embedded Developpment Kit")

Principal outil pour la réalisation de ce projet, EDK permet au concepteur de travailler à très haut niveau, procurant des modules propres à Xilinx. Il offre aussi la possibilité, pour un utilisateur averti, de créer des modules personnalisés. Ce logiciel, qui est en fait un cadre de travail pour la création de systèmes embarqués, offre une suite d'outils permettant de créer un système sur puce à partir d'une simple spécification, en passant par la synthèse, la création du « bitstream », le transfert sur le FPGA et le débogage logiciel et matériel en temps réel. Pour ce faire, EDK utilise plusieurs outils propriétaires dont platgen, XST, iMPACT, Chipscope Pro, ainsi que des interfaces d'outils de tierces parties dont les outils de compilation, de débuggage et les utilitaires binaires de Gnu [GNU04], cygwin[CYGW04], et Modelsim. [MODE04]

Les versions de cet outil ont varié au cours du projet. En effet, la première version utilisée pour ce mémoire fut la primitive 3.2, qui au fil du temps s'est améliorée en 8.1. Plusieurs changements positifs permettent à cette dernière version de se qualifier comme interface facile et efficace pour la conception de système sur puce. Cependant, il est nécessaire d'avoir un ordinateur performant (i.e : P4 2.4 GHz, 1 Go Ram.) afin d'obtenir une compilation assez rapide (< 10 mins.) pour un système d'envergure moyenne (i.e : 4 Microblazes, 5 périphériques, 1 bus).

Xilinx offre, en plus des processeurs, une bonne quantité de modules intéressants avec cet outil. Ces modules interfacent les protocoles OPB, PLB, DCR ou FSL. Cependant, certains d'entre eux, tels les gestionnaires d'horloges, sont indépendants puisqu'ils ne se connectent pas via un bus. Le tableau suivant offre un petit aperçu des modules importants pour notre projet et des processeurs disponibles avec EDK 6.3. Les modules accompagnés d'un astérisque sont aussi disponibles pour le PLB.

Tableau 2.1 : Quelques modules disponibles avec EDK

Modules divers	Modules OPB
MicroBlaze Soft Processor	OPB Arbiter*
PowerPC Embedded Processor	OPB Microblaze Debug Module
On-Chip Peripheral Bus (OPB)	OPB to PLB Bridge
Processor Local Bus (PLB)	OPB to OPB Bridge (Lite Version)
Fast Simplex Link (FSL) Bus	OPB to DCR Bridge
Device Control Register Bus (DCR)	OPB IPIF*
Local Memory Bus (LMB)	OPB External Memory Controller (EMC)
BRAM	OPB SDRAM/DDR SDRAM Controller
LMB BRAM Interface Controller	OPB BRAM Interface Controller
DCR Interrupt Controller	OPB Interrupt Controller
PLB 1-Gigabit EMAC	OPB 16450*/16550*/Jtag/Lite UART
PCI Core Bridge	OPB GPIO*
Chipscope Pro ILA	OPB EMAC*
	OPB Central DMA Controller

De plus, il est important de noter que le Microblaze ne contient qu'une interface OPB et ne peut donc pas être utilisé sur un PLB, à moins d'une modification spécifique majeure de son interface.

CHAPITRE 3

Organisation architecturale et application

Ce chapitre détaille l'architecture et les applications utilisées dans la réalisation de ce travail. Tel qu'illustré à la figure suivante, ce travail tente de démontrer que quatre processeurs s'exécutant au quart de la vitesse d'un processeur standard peuvent s'acquitter de leur tâche beaucoup plus rapidement que le processeur de référence.

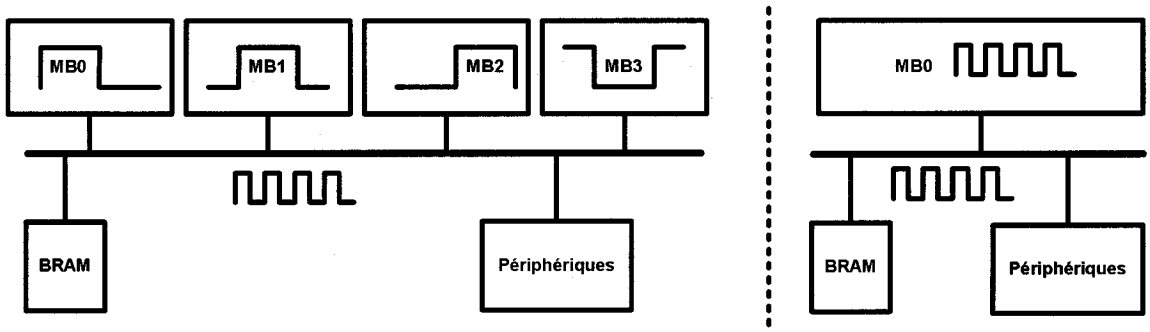

Figure 3.1 : Système à quatre processeurs plus lent versus un seul processeur rapide.

3.1. Vue d'ensemble

Le système présenté dans ce projet correspond à un agencement de quatre processeurs accédant aux périphériques inclus dans le design via un système d'interconnexion partagé et synchronisé. La figure suivante offre un bref aperçu du design implémenté.

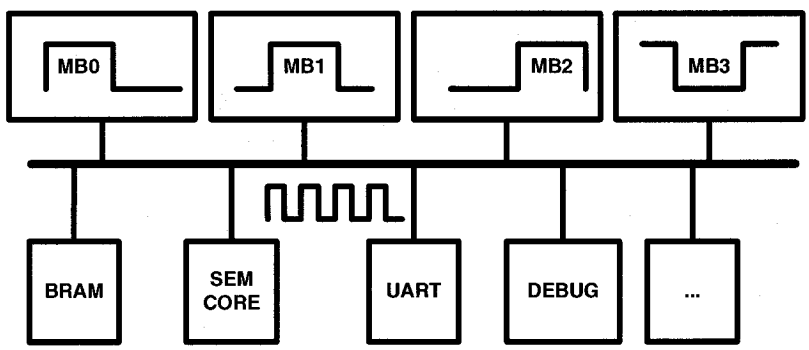

Figure 3.2: Vue d'ensemble du système

Comme le démontre l'illustration, les périphériques utilisés dans ce projet sont majoritairement des modules de base offrant des comportements et des fonctionnalités nécessaires à tout projet. Parmi ces composantes, on retrouve notamment :

- mémoire (BRAM),
- port série (UART),
- module de sémaphores,
- module d'identification
- modules de débogage,
- module de contrôle d'E/S

Essentielle à tous projets, la mémoire se retrouve sous forme de BRAM dans le FPGA utilisé. De ce fait, la mémoire utilisée provient notamment de bloques intégrés dans le système sur puce. Outre la mémoire, l'UART permet d'interagir avec l'usager via le port série inclus sur la carte de développement, le module de sémaphore protège de la corruption les données partagées, tandis que les modules de débogage offrent une interface permettant de retracer des problèmes, tant au niveau logiciel que matériel. Finalement, d'autres petits modules spécialisés ont été ajoutés au design pour rajouter de la fonctionnalité à l'application choisie.

Tel que mentionné précédemment, le protocole de bus utilisé dans ce projet est l'OPB. De ce fait, la base de certains des modules utilisés provient directement d'une bibliothèque fournie avec l'outil de conception de projet de Xilinx. Cependant, certains autres périphériques sont spécifiques au projet, leur implémentation est donc détaillée à la section 3.3.

Évidemment, pour assurer le bon fonctionnement d'un tel agencement de processeurs sur un seul bus, il est impératif d'utiliser une méthode de synchronisation fiable et robuste. Pour ce faire, le canal de communication partagé, ainsi que tous les périphériques

présents dans le système, utilisent une fréquence rapide, alors que les processeurs qui y sont connectés s'exécutent à une fréquence diminuée du quart de la rapide. Afin d'assurer l'exclusion mutuelle du bus, chaque processeur utilise une horloge déphasée de 90° par rapport au processeur précédant. Ainsi, cette organisation rend possible la création d'une fenêtre d'opération permettant à chaque processeur l'accès aux canaux de communication et aux périphériques de façon exclusive pendant une période d'horloge rapide (celle du canal). La Figure 3.3 illustre un chronogramme qui démontre la fonctionnalité de la synchronisation des accès. Sur cette figure, les colonnes ombragées représentent les fenêtres d'opération spécifiques à chaque processeur. Pour initier un accès à un périphérique, chaque processeur doit utiliser la période rapide incluse dans sa fenêtre afin d'éviter les collisions. De la même façon, pour recevoir une réponse, le processeur écoute sur le bus seulement pendant cette période rapide.

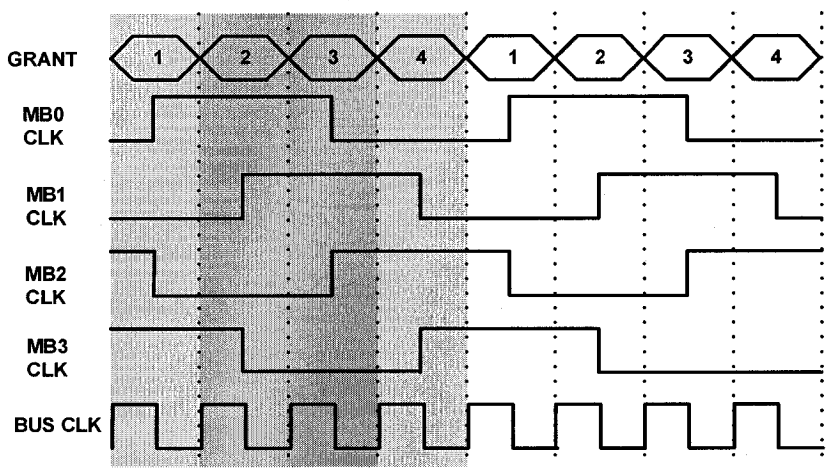

Figure 3.3: Fenêtre d'opération et domaines de fréquence

Bien que plus lent que le reste du système, ce mécanisme offre la possibilité à tous les processeurs de s'exécuter en même temps, tout en évitant les collisions grâce aux accès à tour de rôle sur le canal de communication. De plus, cette technique permet aux processeurs d'effectuer un accès au bus à tous les fronts montants d'horloge. Ainsi, chaque processeur ne se voit jamais bloqué par un accès d'un processeur concurrent.

3.1.1. Prérequis

Évidemment, pour faire une telle implantation d'un système multiprocessus, certaines conditions doivent être présentes dans le design. Ces pré requis sont spécifiques à ce type de multiprocessus, compte tenu des mécanismes particuliers de synchronisation et, de ce fait, ils ne peuvent être utilisé que pour les autres modèles existants de HMT sans un minimum de modifications.

Afin que tous les éléments du système soient bien synchronisés, il est impératif que chaque périphérique présent puisse pipeliner ses accès. En effet, comme il est possible qu'un périphérique reçoive plusieurs accès consécutifs et ce, à chaque front montant de l'horloge rapide, les accès successifs au premier pourraient malencontreusement écraser les précédents. Pour la même raison, il reste intéressant d'utiliser des périphériques qui possèdent une latence d'exécution d'un seul cycle. Dans le cas contraire, un FIFO doit être rajouté afin d'emmagasiner les requêtes suivantes pendant l'exécution du périphérique.

De plus, l'utilisation de ce bus présente certains inconvénients. Comme le chapitre précédent le mentionne, le protocole OPB offre plusieurs scénarios. Certains de ceux-ci sont directement contradictoires avec l'idée principale du HMT. En effet, notre modèle tend à cacher la latence de communication. Or, selon les spécifications du protocole, l'OPB génère un signal d'expiration après une latence de 15 cycles. Il faudra donc que ce projet s'en tienne à des latences maximales de 15 cycles pour des fins de test afin d'éviter des comportements non désirés. Dans le même ordre d'idée, la spécification de l'OPB nécessite que le signal « SELECT » soit maintenu pendant au moins deux cycles. Dans le cas contraire, l'accès courant est considéré comme annulé par le demandeur. Ce scénario n'est pas acceptable pour le projet courant étant donné qu'il se base sur le fait que les processeurs ne doivent absolument pas travailler hors de leur fenêtre d'opération et celle-ci dure exactement un coup d'horloge. Finalement, dans le modèle de bus utilisé, un arbitre de base y est inclus. Ce type de synchronisation n'est toutefois pas nécessaire

au bon fonctionnement du projet, d'autant plus qu'il est possible que ce module génère des collisions non désirées sur le bus.

À la lumière de ces informations, il va sans dire que des modifications doivent être apportées dans les modules appropriés afin d'éviter ce type de problème.

3.2. Organisation de la mémoire

Dans tout projet impliquant plusieurs processeurs, l'organisation de la mémoire attachée à ces noyaux devient un enjeu majeur. De plus, avec les restrictions de ressources de mémoire dans les FPGA, cet enjeu devient encore plus important dans ce type de SOC. De ce fait, il semble très important de trouver une organisation audacieuse mais simple qui permettra au système de partager des zones de stockage, en restant efficace et en économisant le peu de mémoire disponible pour l'ensemble du système.

Pour ce projet, l'organisation doit offrir aux processeurs impliqués :

- une zone partagée : pour les communications inter-processeurs, ou autres.
- une zone privée : pour les données privées et pour la pile d'exécution,
- une zone application : contenant le code à exécuter.

Afin de réduire l'usage de la mémoire de chacun des processeurs, la zone de code peut être identique à tous les noyaux. Malgré ce fait, les processeurs possèdent un flot d'exécution unique grâce à leurs zones privées qui leur permettent de conserver des variables locales uniques, et ainsi de distinguer leur exécution des autres processeurs. Le schéma global de cette organisation s'illustre par la figure suivante :

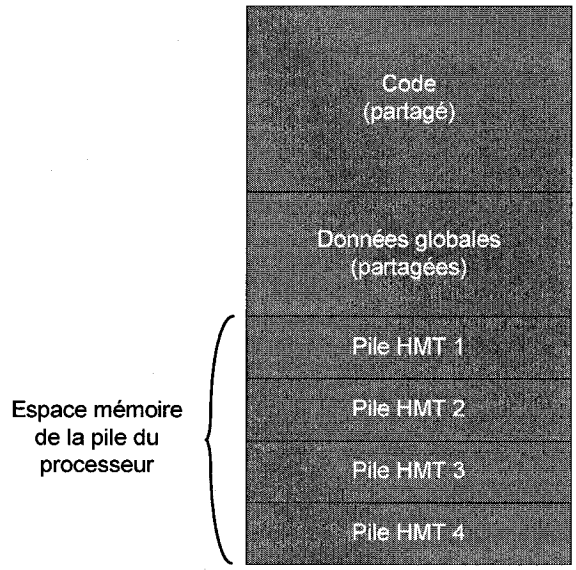

Figure 3.4: Organisation de la mémoire

Comme la figure le démontre, les zones réservées aux piles d'exécution sont contiguës et de même grandeur. Cette caractéristique est obligatoire car l'implémentation du système en dépend. Dans le système actuel, la grandeur des 4 segments peut atteindre 16Kb.

3.3. Détails d'implantation

Pour atteindre les objectifs du projet et les spécifications du système décrit dans la section précédente, des modifications s'imposent dans la majorité des modules impliqués dans cette architecture. De ce fait, cette sous-section fait état des changements apportés à l'implémentation originale des modules fournis par Xilinx ou par tout autre fournisseur. Pour ce faire, le premier point décrit les principaux ajouts apportés au processeur utilisé. Ensuite, les modifications sur l'implémentation du bus de communication sont élaborées pour finalement terminer avec le détail des changements sur les périphériques existants et la description des nouveaux inclus dans le projet.

3.3.1. Horloges

Tel que mentionnée précédemment et illustré à la Figure 3.3, ce projet nécessite quatre horloges décalées de 90 degrés pour les processeurs ainsi qu'une horloge quatre fois plus lente pour les périphériques. Comme la carte utilisée nous offre deux holorges (40MHz et 80MHz) synchronisées (sans décalage) pour le FPGA, l'utilisation d'un module de gestion d'horloges est indispensable pour la génération des signaux requis. Ces modules de gestion sont disponibles dans les bibliothèques offertes par Xilinx sous le nom de DCM. Comme le démontre la Figure 3.5, représentant le schéma bloc d'un DCM, ce type de module offre un multitude d'options permettant de générer une kyrielle de signaux différents dérivant d'un signal de base. En plus, ce type de module tente de minimiser les délais entre les différentes horloges générées.

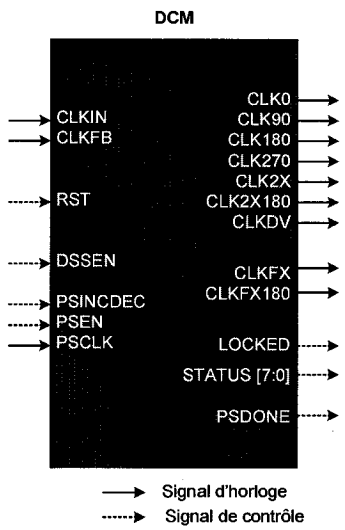

Figure 3.5: Shema bloc d'un DCM [XILI04]

Ainsi, il est possible à l'aide d'un seul de ces module de générer une horloge à 160MHz et 4 horloges de 40MHz déphasées de 90 degrés à partir de l'horloge de base à 40MHz disponible sur la carte. Malheureusement, une telle configuration de ce module n'est pas n'est pas adéquate au projet. En effet, le bus de communication ne supporte pas une fréquence dépassant 150MHz. Ainsi, il est primordial de diminuer la fréquence de base de l'horloge afin de générer une horloge rapide qui ne dépasse pas les limitations des modules inclus dans le projet. Une façon simple consiste en la sérialisation de plusieurs

DCM. Le premier divise la fréquence de base pendant que le second la multiplie par 4 (afin d'obtenir la partie rapide) et décale l'horloge divisée pour obtenir les signaux mutuellement exclusif.

3.3.2. Processeur

Afin que le processeur soit compatible avec la synchronisation, le Microblaze doit subir quelques modifications. Pour que ces changements n'affectent pas le comportement du processeur, celui-ci est encapsulé dans un module qui se charge d'être compatible avec le système. Cette nouvelle structure, baptisé le « Mortiblaze », est composée de l'enveloppe, qui permet d'ajouter des fonctionnalités au processeur, ainsi que du processeur qu'il contient. La figure suivante représente en schéma bloc les différents modules intégrés au Microblaze l'obtention du comportement du « Mortiblaze ».

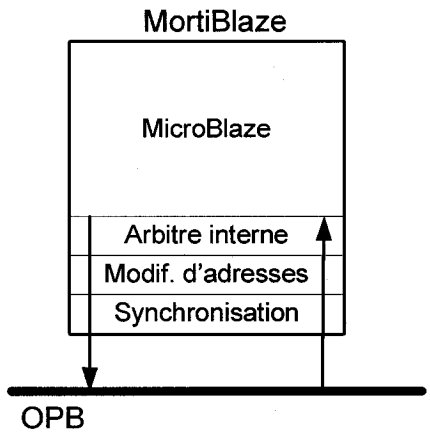

Figure 3.6: Shema bloc d'un « Mortiblaze »

Ainsi, comme le démontre la figure 3.5, ce module offre au processeur trois nouveaux services majeurs permettant son bon fonctionnement ainsi que sa compatibilité au sein du projet. Cette structure est représentée en couche de la même façon qu'une pile de protocole. Cette représentation consiste au fait que tous les signaux doivent passer au travers de chacune des couches avant de rejoindre la couche la plus élevée : le processeur lui-même.

- Le premier ajout consiste à intégrer un arbitre à l'intérieur du nouveau processeur « Mortiblaze ». Effectivement, comme le processeur suit une architecture Harvard, il nécessite la présence d'un arbitre pour permettre l'utilisation des deux interfaces maîtres qu'il possède. Contrairement aux architectures communes, cet arbitre ne peut pas être intégré au niveau du bus étant donné que tous les maîtres connectés utilisent une synchronisation à temps partagé. L'arbitre intégré au processeur s'exécute selon un algorithme permier arrivé, premier servi (Round Robin). Cet arbitre utilise d'ailleurs la même interface d'arbitration du Microblaze (Request-Grant-Select).
- Le second élément définit l'instant de départ de la fenêtre d'opération ainsi que son délai. En d'autres mots, il offre au processeur la capacité d'utiliser le bus à temps partagé avec les autres processeurs présents dans le système. Finalement, cet élément de synchronisation permet aussi d'obtenir les résultats des requêtes au moment opportun. Effectivement, tel que montré dans la section 3.1, les résultats doivent être retournés par les périphériques au début d'une fenêtre d'opération succédant celle qui a permis l'envoi de la requête par le processeur. De ce fait, cette machine s'assure que les résultats retournés appartiennent bel et bien au processeur.
- Le dernier ajout à ce nouveau processeur consiste au décalage des adresses lors des transactions dans la pile d'exécution. En temps normal, comme tous les processeurs exécutent le même code, toutes les références à la pile d'exécution dans ce code sont identiques. De ce fait, si tous les processeurs exécutent exactement le même code, toutes les variables, même les locales, seront identiques pour chaque flot d'exécution. Par exemple, pour une application de traitement de paquets, si tous les processeurs utilisent la même pile, ceux-ci traiteront toujours le même paquet. Cependant, puisque le but du projet consiste, justement, à l'obtention de quatre flots d'exécution sur quatre processeurs ayant des parties de mémoire partagées, il devenait nécessaire d'ajouter le décalage mentionné.

D'un coté plus technique, ces trois services ajoutés dans le « Mortiblaze » sont implémentés sous forme de machine à état qui s'exécute à la fréquence de l'horloge rapide, donc celle du bus. La machine à état utilisée au sein du « Mortiblaze » est illustrée à la Figure 3.7. Cette machine possède 6 états d'opération et 6 états d'attente.

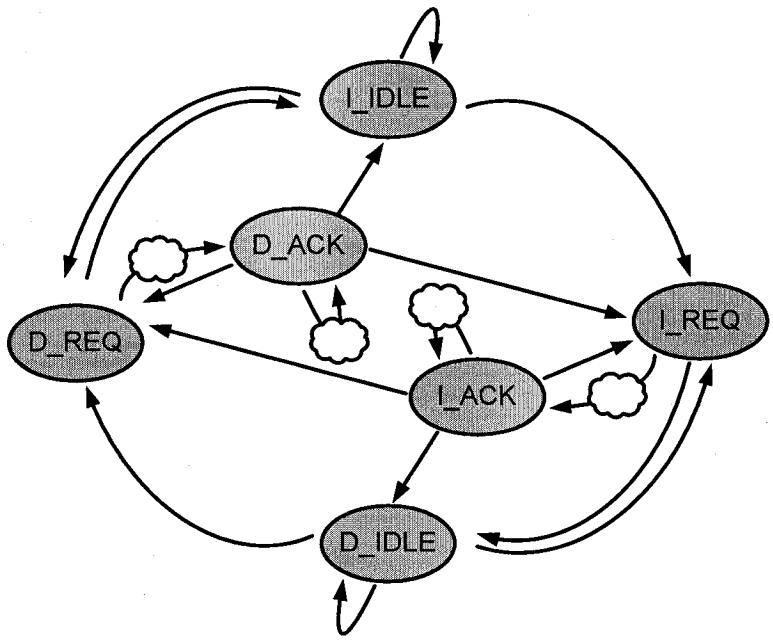

Figure 3.7: Machine à état du « Mortiblaze »

Le diagramme représenté sur cette figure, possède des états et des nuages. Les nuages représentent des états d'attente qui permettent de perdre un certain nombre de coups d'horloge (3 coups dans le cas de 4 processeurs) afin d'arriver dans l'état de réception des résultats avec une fenêtre d'opération valide.

De plus, la structure symétrique du diagramme démontre que le comportement est équivalent pour les données et les instructions. De ce fait, seuls les états pour les instructions seront détaillés.

- La machine débute dans l'état d'attente « I_IDLE ». Le préfixe « I_ » illustre que la priorité est accordée au coté instruction. Ainsi, si les deux maîtres[7] font une requête simultanément, le droit sera accordé au bus d'instruction. Cet état va basculer vers l'état de « I_REQ » ou « D_REQ », dépendamment du maître émetteur du signal « REQUEST ». Lors de la transition vers l'état « I_REQ », le signal « GRANT » est envoyé au processeur pour qu'il puisse débuter sa requête.
- L'état de requête « I_REQ » attend que le processeur ait débuté une transaction et que la fenêtre d'opération soit valide pour envoyer les signaux sur le bus. Une fois ces deux conditions atteintes, la transaction est acheminée au bus et l'état bascule vers « I_ACK ». En cas d'erreur, l'état change à l'état « D_IDLE » afin que les deux maîtres puissent faire des requêtes à tour de rôle.
- Le dernier état attend la confirmation de l'esclave pour acheminer les résultats au processeur. Une fois les résultats acheminés, si le processeur désire continuer sa transaction l'état bascule à « I_REQ », sinon, il bascule vers « D_REQ » ou vers « D_IDLE » dépendamment des signaux de requêtes du côté des données.

La fenêtre d'opération, quant à elle, consiste en un simple signal qui se génère à partir de l'horloge du bus et celle du processeur. Lorsque ce signal est actif, les signaux peuvent être acheminés, dans le cas contraire, ils sont bloqués. De plus, ce signal doit rester actif pendant un seul coup d'horloge rapide et doit être présent au moment de la transition montante de l'horloge du processeur. Ainsi, comme la figure le démontre, un détecteur de front conjugué avec un « reset » synchrone provenant de l'horloge lente permet de créer ce signal si important.

---

[7] Les processeurs possédant une architecture dites Hardvard possèdent deux bus (instruction et donnée) considérés ici comme deux maîtres.

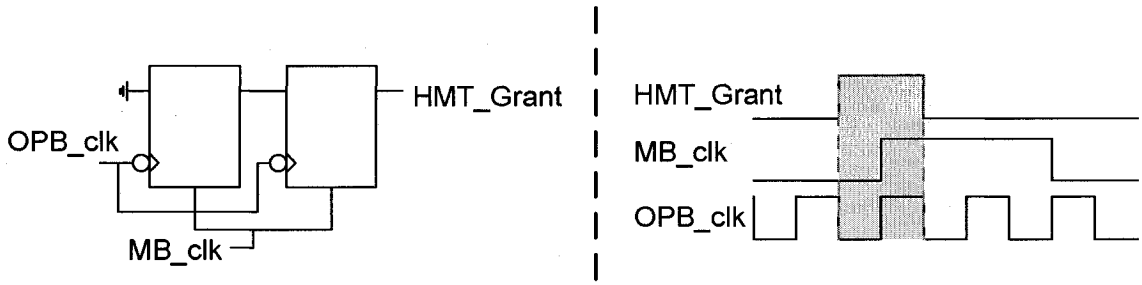

Figure 3.8 : Génération du signal de synchronisation.

La dernière utilité importante de cette encapsulation du processeur permet de transformer l'exécution d'un code unique en un flot d'exécution différent sur chaque processeur. En effet, une partie importante du projet sert à permettre aux processeurs impliqués dans le système d'exécuter un flot unique à eux-mêmes, et ainsi à pouvoir traiter des données différentes sur chaque unité de traitement, tout en ayant le même code d'entrée. Cette technique permet donc l'économie de mémoire car le code d'exécution n'y est emmagasiné qu'une seule fois et à une seul endroit pour tous les processeurs.

Afin d'y arriver, un numéro d'identification statique (au niveau du code VHDL) est attribué à chaque processeur. Ce numéro permet d'identifier la zone de mémoire privée à chacun d'eux. Ainsi, quand l'unité de traitement numéro 1 doit accéder à l'adresse de la pile dans le code, il sait en fait qu'il doit accéder à sa propre pile. Tous les processeurs s'exécutant ainsi possèdent donc une pile distincte et peuvent alors obtenir leur propre flot d'exécution.

Bien que cette technique semble lourde à première vue, l'organisation de la mémoire a été pensée pour réduire le plus possible le traitement pour l'obtention de l'adresse de la pile. En effet, il ne suffit que d'identifier les appels à la pile et de remplacer les bits 17 et 18 par le numéro d'identification pour que l'adresse soit traduite correctement. Ainsi, on obtient les plages d'adresses suivantes pour les quatre processeurs du système.

Tableau 3.1 : Adresses des piles des différents processeurs

Numéro d'identification	Adresse de la pile
00	0xC000
01	0xD000
10	0xE000
11	0xF000

Le code complet du « Mortiblaze » se trouve à l'annexe 1.

3.3.3. OPB

Tel que mentionné dans la section précédente, le processeur de notre système possède maintenant son propre arbitre afin de s'assurer que chacune de ses deux interfaces maîtres reçoivent ses requêtes. Cependant, le modèle d'interconnexion choisit pour ce projet possède lui aussi un arbitre intégré en son sein. Ainsi, ces deux modules sont directement en compétition pour établir la priorité des opérations. Malheureusement, l'arbitre de l'OPB n'est pas du tout conçu pour le type de synchronisation suivit dans le système étudié et viendrait perturber des signaux importants au bon fonctionnement du système.

Ainsi, le module spécifiant le modèle de l'OPB, a du subir une amputation majeure afin qu'il puisse être insérer dans le projet sans troubler la synchronisation des périphériques et des processeurs. Effectivement, l'arbitre intégré à ce module a été drastiquement enlevé de l'implémentation de l'OPB.

3.3.4. Modification des autres périphériques existants

Comme la majorité des périphériques utilisés proviennent de la bibliothèque fournie avec EDK, il a fallu les ajuster afin qu'ils puissent être aisément incorporés dans le système créé. En effet, chacun des périphériques doit malheureusement subir quelques modifications afin que la synchronisation des données puisse être respectée. Dans le cas

contraire, en plus de posséder un périphérique non-fonctionnel au sein du système, celui-ci pourrait rendre le design complètement inutilisable. Par conséquent, dans le cas où le périphérique émetterait des réponses au mauvais moment, la réponse finale sera le résultat d'une opération logique « ou » utilisant les bits de la réponse érronée combinés à ceux de la réponse du périphérique en droit de soumettre son résultat.

Afin d'expliquer plus clairement les brèves modifications à apporter à un périphérique existant que l'on veut instancier dans le système, les changements effectués sur le module de la mémoire seront détaillés. Ce module que l'on retrouve dans la bibliothèque de Xilinx sous le nom "BRAM", possède une interface OPB qui devra désormais supporter le mécanisme de synchronisation de notre système. Avant d'expliquer ces changements, il est important de vérifier si le module que l'on désire utiliser répond en un seul cycle. Si ce n'est pas la cas, il faudra, tel que mentionné précédemment, emmagasiner les requêtes dans une file et répondre au moment opportun. De telles adaptations sont détaillé dans [SATH05]. Cependant, ce n'est pas le cas de notre mémoire. Heureusement, comme tous les périphériques utilisés dans ce système, son traitement prend seulement un cycle. Ainsi, seule la synchronisation et certains détails du protocole OPB devront être modifiés.

Comme il a été mentionné précédemment dans les cas spéciaux du protocole OPB, l'arrêt de la transaction dû à un cas d'annulation de requête peut survenir. De ce fait, il est nécessaire de désactiver l'utilisation du module qui génère ces cas afin que notre périphérique ne suive que le protocole qui nous intéresse. En effet, dans la majorité des modules inclus dans la bibliothèque fournie, un module est inséré dans le flot de données pour contrôler les différents cas du protocole OPB. Ainsi, pour éviter tout problème d'inconsistance entre la synchornisation utilisée dans ce système et les spécifications du protocole de bus, il est nécessaire de désactiver le module en charge de vérifier l'utilisation exacte du protocole.

De plus, il faut pipeliner la sortie (ou l'entrée) afin de s'assurer que le résultat sorte au bon moment. La meilleure technique, pour un module qui n'a qu'un seul cycle de latence, consiste à ajouter 3 bascules-D à la sortie du périphérique afin que la réponse sorte exactement au bon moment. Ainsi, la synchronisation est respectée.

3.3.5. Nouveaux périphériques

Tout au long du projet, plusieurs périphériques ont été conçus pour des fins de test ou encore pour l'obtention du système final. Ces différents périphériques sont présentés brièvement dans les sous-sections suivantes.

3.3.5.1. Module identifieur / compteur

Parmi les plus importants périphériques de ce projet, ce module est utilisé de différentes façons. Tout d'abord, son utilisation permet aux processeurs de s'identifier dynamiquement. De plus, le module parvient à compter le nombre de paquets traités par les quatre processeurs lors de l'exécution d'une application.

L'identification que procure ce module est dite *dynamique* car c'est pendant l'exécution que le processus d'identification intervient. En fait, chaque processeur possède déjà une identification statique pour définir ses zones de mémoire privée. Cependant, ce périphérique offre la possibilité de se distinguer dans son flot d'exécution. Ainsi, une lecture sur ce périphérique renvoi un « id » au processeur qui, par la suite, peut utiliser ce « id » dans le code afin de détourner son exécution du flot normal. Sans cette fonctionnalité, tous les processeurs seraient contraints de toujours utiliser le même chemin d'exécution.

L'écriture sur ce périphérique engendre l'incrémentation d'un compteur qui permet de garder une trace de l'exécution. Dans le cas présent, ce compteur permet de comptabiliser le nombre de paquets traités.

Les deux fonctionnalités sont implémentées sous forme de compteur qui possède une interface esclave OPB. Le code source de ce périphérique se trouve en annexe.

3.3.5.2. Module sémaphore

Bien que son nom nous en donne une bonne idée, ce module procure un mécanisme de protection multiprocesseurs au code exécuté dans le système. En d'autres mots, ce périphérique permet de sérialiser une partie de code qui pouvait être exécutée de façon parallèle dans les différents processeurs. De façon plus rigoureuse, on devrait l'appeler « Mutex Engine » car sa fonctionnalité de base procure une exclusion mutuelle des périphériques au processeur demandant. Cependant, il est simple d'utiliser ce mécanisme de base pour générer une bibliothèque comprenant les fonctionnalités des sémaphores compteurs.

La séparation des flots d'exécution des différents processeurs impliqués dans le système devient donc triviale grâce à ce module. En effet, il suffit de prendre un mutex sur une petite partie du code au début de l'exécution. Ainsi, cette portion sera exécutée de façon sérielle par les différents processeurs et ceux-ci seront désormais décalés dans leur exécution par rapport aux autres. De plus, ce module permet aussi de simplifier l'utilisation des périphériques qui prennent plus d'un cycle à traiter une donnée. De ce fait, l'usage de mutex permet de minimiser les modifications nécessaires à un périphérique existant. Désormais, seules les modifications de synchronisation sont obligatoires.

Évidemment, l'usage de ce périphérique augmente les latences de communications et ainsi peut réduire les performances observées de façon significative. De ce fait, il faut limiter son utilisation dans des systèmes visant la performance.

3.3.5.3. Périphérique lent

Comme le projet tente de prouver que le multi-processus matériel rend efficace les systèmes possédant des mécanismes de communications lourds ou la latence est très grande, il semble important de pouvoir tester ce type de systèmes pour l'obtention des résultats. Ce module, simule un périphérique avec une longue latence. De ce fait, une lecture sur l'adresse de ce périphérique entraîne automatiquement une attente d'un nombre pré-determiné de coups d'horloge. Ce temps d'attente ou la grandeur de la latence gaspillée par ce module se paramétrise directement dans le module avant sa compilation. Cependant, par défaut celui-ci est paramétré à 10 coups d'horloge avant de renvoyer une réponse.

Évidemment l'utilisation de ce module n'entraîne pas spécialement une perte significative de performance. Effectivement, ce type de module ralentisseur suit aussi la loi d'Amhdal tout comme les modules d'optimisation,. Ainsi, une seule lecture sur ce module n'aura presqu'aucun effet sur l'exécution totale du projet. De ce fait, afin d'avoir un ralentissement significatif, il faut appliquer celui-ci à une portion importante de l'exécution de l'application.

CHAPITRE 4
Analyse des résultats

Les résultats détaillés dans ce chapitre proviennent de simulations logicielles du système. Évidemment, comme il était impossible de créer tous les modules nécessaires à l'élaboration de ce projet, il est important d'avoir accès à des modèles simulables d'IPs commerciaux utilisés dans ce système. Heureusement, de tels modèles simulables sont disponibles et offerts par les sociétés propriétaires dans leurs trousses de développement.

Afin d'obtenir des résultats de simulation, un design spécifique a dû être créé pour la génération des horloges. Malheureusement, l'usage de deux DCM en série, tel qu'expliqué dans la section 3.3.1, n'était pas un comportement simulable avec les outils utilisés lors de la collecte des données. L'horloge de base employée pour la simulation est déterminée au niveau du banc d'essai et constitue un signal de 20MHz. De ce fait, un seul DCM est utilisé afin de générer les signaux décalés et le signal multiplié.

Afin de démontrer que chaque partie du projet fonctionne, les résultats de simulation de divers modules seront détaillés. Ainsi, il sera facile de comprendre comment les nombreuses contraintes, préalablement expliquées, ont été réglées. Ensuite, les résultats des diverses applications utilisées sur la plateforme seront présentés et analysés. Cependant, avant d'expliquer les résultats, il est important de détailler l'environnement de test, ainsi que les applications employées pour obtenir de tels chiffres.

4.1. Environnement et application test

Le système utilisé durant l'expérimentation est doté de quatre processeurs Microblaze qui s'exécutent à une vitesse d'horloge de 20MHz. Le choix de la magnitude de cette horloge provient des horloges disponibles sur la carte de développement. Les tests seront exécutés sur deux plateformes. L'une, dite HMT, contient plusieurs périphériques

spécialisés, en plus de la mémoire, dont le périphérique lent, le module identificateur ainsi que le module sémaphore. Ce système permet donc de représenter un modèle contenant un seul processeur utilisant quatre flots d'exécution matériels.

Afin de tester ses performances, le design HMT sera opposé au design de référence qui, quant à lui, est un système possédant un seul Microblaze s'exécutant à 80 MHz. Évidemment, tous les périphériques accompagnant ce design de référence sont identiques à ceux contenus dans le design HMT

Les deux systèmes sur puce ont été conçus à l'aide de l'outil de développement de Xilinx présenté précédemment, appelé EDK. De plus, les traces de simulation obtenues ont été récoltées à l'aide de l'outil Modelsim 6.0 [MODE04]. Les résultats proviennent directement de ces traces. Comme ces chiffres sont issus de simulations, les résultats ne pourront que servir de preuves de concept et ainsi, de plus amples recherches pourraient établir un modèle de référence pour du multi-processus matériel sur puce.

De plus, chacun des tests effectués sera exécuté selon deux configurations différentes du périphérique lent. La première configuration de ce périphérique consiste à répondre dans un délai normal d'un contexte de l'OPB, soit un minimum de 3 cycles d'horloges entre le départ de la requête et l'arrivée de la réponse. La seconde configuration ajoutera une latence modifiant le délai d'échange de données à 10 cycles. Ce type de latence pourrait simuler, par exemple, l'accès à une ressource externe ou éloignée comme une mémoire ou tout autre périphérique plus lent. Évidemment, l'ajout de ce délai est soumis à la loi d'Amdhal, qui s'applique autant pour une accélération que pour une décélération. De ce fait, la perte de performance associée à cet ajout dépend directement de la fréquence d'utilisation de ce module dans chacun des tests.

4.2. Applications

Afin d'évaluer de façon efficace le potentiel de ce projet, quatre applications simples sont développées, intégrées et simulées pour obtenir une multitude de résultats permettant d'analyser le comportement du système sous différents angles. Cette section détaille chacune des applications, leur fonctionnement, leur but ainsi que leur implémentation. Les résultats obtenus sont analysés et expliqués dans la section suivante.

4.2.1. Accès mémoire

La première des trois applications présentées pour évaluer le projet permet tout simplement de quantifier les accès à la mémoire du système. En effet, cette petite et simple application n'est en fait qu'une série d'écritures ainsi qu'un nombre d'appels au module de latence à chaque huit requêtes à la mémoire.

Cette application devrait permettre d'évaluer le gain brut obtenu pour des accès mémoire purs. Elle simule le gain possible avec une application purement axée sur les données. Le désavantage de cette application réside dans le fait que tous les processeurs exécutent exactement le même code. De ce fait, à tout moment, chacune des quatre exécutions devrait se situer à quelques instructions près. Donc, le temps d'exécution devrait varier linéairement selon le nombre de paquets traités.

4.2.2. Somme de contrôle

Basé sur le RFC 1071, cette application calcule les sommes de contrôles (« checksum ») de paquets IPv4 prédéfinis dans la mémoire. L'application traite les paquets différemment selon l'exactitude des sommes obtenues. Chacun des paquets est sélectionné et traité au hasard. Il est important de noter que la mémoire contient une proportion équivalente a 3/8 de paquets contenant une mauvaise somme de contrôle. Ainsi, il arrivera, en moyenne 3 fois sur 8, lors de l'exécution de l'application que la vérification de la somme de contrôle soit invalidée.

Cette application nous permet d'évaluer le gain du multiprocessus avec une application orientée donnée et contrôle. Effectivement, même si l'application traite des paquets en mémoire, il semble important de noter que la majeure partie du code exécuté effectue un calcul à l'aide d'opérations arithmétiques et logiques. Pour les mêmes raisons que l'application des accès mémoire, le gain ne devrait pas varier en changeant le nombre de paquets. Bien qu'il y ait une variable hasardeuse, sur un grand nombre de paquets le rapport de bonne somme de contrôle se situera en moyenne à 3/8 pour les quatre processeurs, et de ce fait, le temps d'exécution devrait être proportionnel au nombre de paquets traités.

4.2.3. Somme de contrôle partagée

Afin de permettre aux processeurs de séparer leur exécution des autres processeurs, une troisième application est intégrée au projet. Cette application n'est en réalité qu'une copie de la précédente. Cependant, elle utilise une variable partagée ainsi qu'un module de sémaphore binaire (ou mutex), afin d'assurer l'exclusion mutuelle sur les sections critiques de leur exécution. De ce fait, chacune des exécutions devra se distinguer des autres lors de l'accès aux objets partagés.

Ainsi, on peut tester l'efficacité d'objets partagés dans le système multiprocessus, en plus de pouvoir quantifier le gain selon plusieurs nombres de paquets traités. Effectivement, contrairement aux deux autres applications, le gain devrait croître en augmentant le nombre de paquets, minimisant ainsi l'effet de l'attente du début de l'exécution. Cette application permet de simuler encore une application orientée donnée et contrôle possédant un mécanisme de partage de ressources. Malheureusement, ce test ne peut être opposé au design de référence. En effet, le design de référence ne contient qu'un seul processeur : l'utilisation du mutex ne sera ainsi qu'une perte de trois cycles. Il n'y a donc aucun but au test.

4.3. Simulation

Avant de discuter des résultats de simulation obtenus, il est important de vérifier certains comportements précédemment mentionnés.

4.3.1. Fenêtre d'opération

Telle qu'expliqué précédemment, chacun des processeurs possède une horloge quatre fois plus lente que celles du reste du système. De plus, une période spécifique baptisé la la fenêtre d'opération du processeur est attribuée a l'unité centrale pour effectuer ces opérations sur le reste du système. Cette période spécifique débute par le front descendant de la période rapide juste avant le front montant de la période lente et se termine après un délai d'exactement une période d'horloge rapide. La simulation suivante représente les 5 horloges mentionnées ainsi que la fenêtre d'opération du processeur 0 représentée par le signal nommé « hmtgrant ».

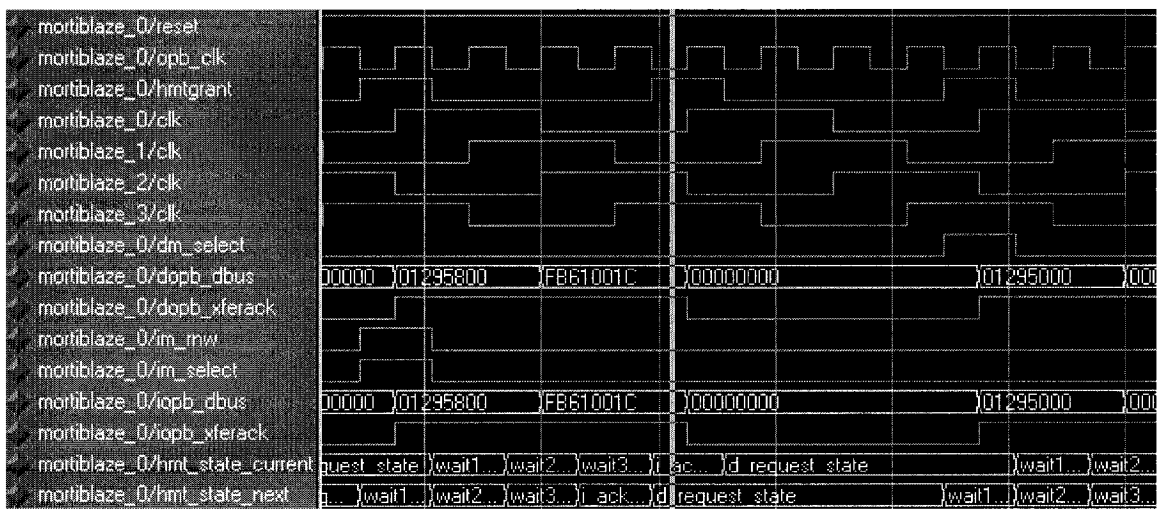

Figure 4.1 : Simulation de la fenêtre d'opération du Microblaze 0

Dans cette même figure, il est intéressant de remarquer que les signaux internes aux processeurs continuent de changer au gré de l'horloge lente. Cependant, la machine à état, qui s'occupe de l'interface entre les signaux du processeur et ceux du reste du système, suit la fréquence rapide. Finalement, certains signaux illustrés, tels que le « im_rnw » et le « im_select », suivent quant à eux exactement la fenêtre d'opération du processeur 0.

4.3.2. Flot d'exécution

Sans décalage explicite, les flots d'exécution des processeurs se suivent indéfiniment. Effectivement, comme la zone mémoire d'instruction est identique, chacun des processeurs exécute toujours le même code au même moment. Ainsi, tel qu'on l'observe sur la figure suivante, chacun des processeurs se retrouve dans un même état au même instant. En ce sens, la figure ci-dessous illustre les compteurs ordinaux des quatre Microblaze en execution. Il est intéressant de voir que ces compteurs se suivent continuellement.

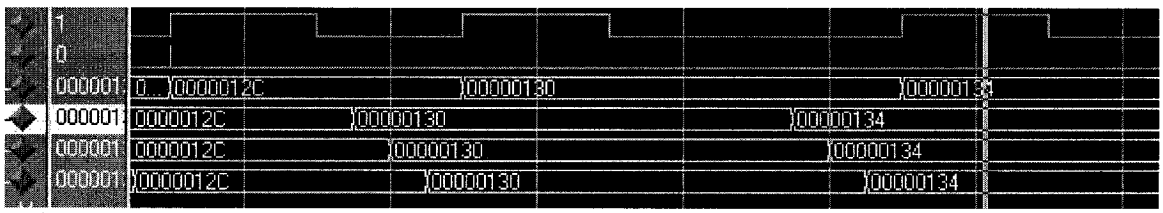

Figure 4.2 : Flots d'exécution des Microblazes

Bien que pratique pour le déboggage du système, ce type d'exécution devient gênant étant donné que tous les processeurs utilisent les mêmes périphériques à un instant donné. Or, le but de ce type de HMT est aussi d'obtenir plusieurs flots d'exécution distincts, permettant ainsi une utilisation uniforme des ressources disponibles. Afin d'obtenir ce type de comportement, la troisième application utilise le module de sémaphore. Tel que mentionné précédemment, ce module, en plus de protéger les zones de mémoires partagées, permet de séparer les flots d'exécution, et ainsi d'uniformiser l'utilisation des

ressources. La figure suivante illustre un bel exemple d'un flot qui se détache du peloton grâce au module de sémaphore.

Figure 4.3 : Détachement de flot d'exécution

4.4. Résultats

La présente section expose et explique les résultats des simulations des trois applications test définies précédemment, et ce dans leurs différentes configurations.

4.4.1. Test 1 – Accès mémoires, application orientée données

Le tableau 4.1 présente les résultats du test 1 dans ses deux configurations et sur deux design différents.

Tableau 4.1: Résultats du test sur les accès mémoires.

Design	Latence standard		Avec une latence de 10 cycles	
	Cycles/ paquet	Gain	Cycles/ paquet	Gain
Référence	51	-	55	-
HMT	25	2.04	26	2.11

La Figure 4.4 illustre à l'aide d'un histogramme l'optimisation obtenue avec le design HMT.

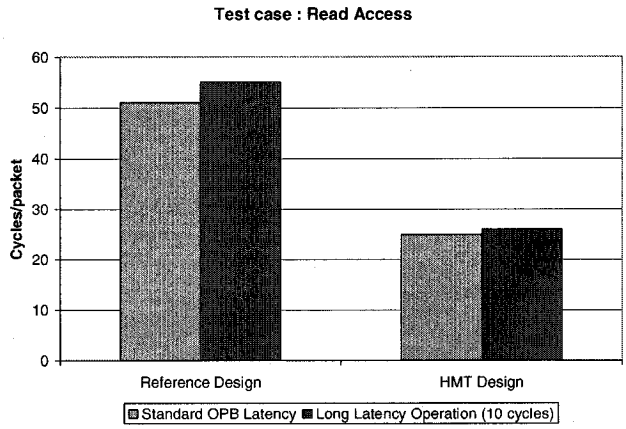

Figure 4.4 : Comparaison des gains du test1

De ces données, on peut remarquer que le nombre de cycles par paquet a littéralement diminué de moitié sur la plateforme HMT. Ce gain est directement attribuable au fait que les accès à la mémoire passent de 3 cycles sur un design standard à 1 seul cycle sur la plateforme HMT, ce qui devrait apporter un gain de 3. Cependant, comme la loi d'Amhdal le dicte, le gain total obtenu dépend directement de la portion accélérée. Or, l'application utilisée pour ce test ne contient pas uniquement que des accès mémoires. En effet, d'après le gain obtenu, on remarque que les accès mémoire constituent environ 66% des instructions de l'application pour le test1. On constate d'ailleurs une légère augmentation du gain lors de l'utilisation du périphérique lent. Cette trop faible augmentation est encore une fois régie par la loi d'Amdhal. Afin d'augmenter son effet sur le gain, ce périphérique aurait dû être utilisé beaucoup plus qu'une fois tous les 8 paquets.

4.4.2. Test 2 – Calcul des sommes de contrôle

Pour le test1, un gain d'un peu plus de 2 est obtenu. Puisque l'application du test 2 contient une moins grande proportion d'accès mémoire, on peut s'attendre à un gain un peu plus faible.

Le tableau 4.2 et la Figure 4.5 présentent les résultats obtenus en calculant et vérifiant le checksum 4000 paquets ipv4

Tableau 4.2: Résultats du test du calcul des "checksum"

Design	Latence standard		Avec une latence de 10 cycles	
	cycles/ paquet	Gain	Cycles/ paquet	Gain
Référence	2712	-	2725	-
HMT	1380	1.96	1381	1.97

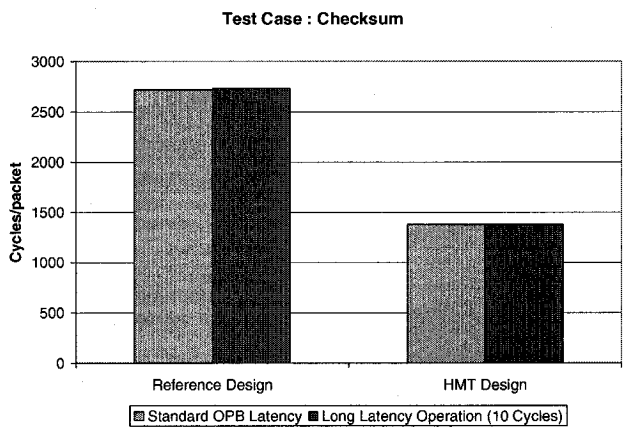

Figure 4.5 : Comparaison des gains du test2

Tel que mentionné précédemment, le gain obtenu pour le calcul du « checksum » est plus faible que celui pour le test des accès mémoire. Comme la majorité du gain provient des accès mémoire et que la proportion de ce type d'instructions est beaucoup plus faible dans la seconde application, il est normal de noter une diminution du gain. On remarque à nouveau une faible hausse du gain lors de l'ajout du périphérique à accès lent. Encore ici, il aurait été intéressant d'augmenter la proportion d'utilisation de ce périphérique pour observer pleinement l'avantage d'une plateforme HMT.

4.4.3. Test 3 – Calcul du « checksum » partagé

Finalement, ce dernier test, ne peut pas être comparé comme les autres. En fait, cette simulation permet simplement d'appuyer la thèse selon laquelle le nombre de cycles/paquets n'est plus constant lorsque les flots d'exécution des processeurs se détachent.

Pour ce faire, le périphérique de sémaphore est utilisé dans l'application du troisième test.

Le tableau suivant présente les résultats obtenus lors du traitement de 1000, 5000 et 10000 paquets. Les résultats exposés dans ce tableau sont aussi en nombre de cycles/paquets.

Tableau 4.3: Resultats du test sur le "checksum" partagé

Design	400 paquets	1000 paquets	5000 paquets	10000 paquets
HMT	1385	1381	1380	1380

À première vue, le nombre de cycles par paquet semble à peu près constant en augmentant le nombre de paquets traités, mais pourtant, ce n'est pas le cas. Effectivement, l'utilisation du périphérique de sémaphore bloque l'exécution de trois des processeurs, ne laissant le champ libre qu'au processeur détenteur du mutex. Une fois le mutex libéré, un des trois processeurs en attente en prendra possession pour continuer son exécution, et ainsi de suite. Ce mécanisme apporte un certain supplément de cycles dès le début de l'application. Une fois les flots parfaitement synchronisés, les processeurs n'attendront plus pour le mutex et ainsi, les traitements seront considérés de la même façon que pour le test 2. En ce sens, c'est ce supplément distribué sur 400 paquets qui donne la différence de 4 cycles par rapport au traitement de 1000 paquets et de 5 cycles par rapport au traitement de 5000 et 10000. En réalité, lorsque beaucoup de paquets sont traités, ce supplément devient négligeable par rapport au reste de l'exécution.

Conclusion et travaux futurs

Après avoir passé en revue l'évolution des processeurs et surtout des solutions pour augmenter leur efficacité, nous avons décidé de démontrer qu'un des mécanismes identifiés, le mutiprocessus matériel, pouvait être reproduit à l'aide de plusieurs plus petits processeurs remplaçant les contextes matériels d'un processeur HMT, et ce en déployant un effort relativement restreint par rapport à l'élaboration d'un processeur HMT. Pour ce faire, ce travail tente de démontrer, tel qu'illustrer sur la figure suivante, que quatre processeurs s'exécutant au quart de la vitesse d'un processeur standard peuvent s'acquitter de leur tâche beaucoup plus rapidement que le processeur de référence. Évidemment, malgré l'optimisation de l'exécution, la taille d'un tel processeur quadruple presque par rapport à un processeur HMT avec quatre contextes matériels. Cependant, suite à l'écart actuel de la loi de Moore, l'espace nécessaire coûte beaucoup moins cher que la différence de l'effort à déployer pour créer un processeur embarqué purement HMT, par rapport au modèle proposé.

Bien que la théorie semble simple, pour arriver au résultat pratique, les processeurs doivent être synchronisés de façon adéquate, afin d'obtenir un modèle semblable au HMT. En effet, la technique du mutiliprocessus matériel à grains fins souligne qu'à chaque coup d'horloge, un contexte matériel exécute une instruction. En ce sens, le modèle proposé dans ce travail ne peut guère avoir de délai de synchronisation. Ainsi, la synchronisation utilisée s'apparente à un mécanisme de partage de temps, où chaque processeur s'exécute à tour de rôle pendant un coup d'horloge et à un instant précis. Ainsi, les collisions sont inexistantes et les transferts de données, optimisés

Comme de tels processeurs embarqués n'existent pas sur le marché, le Microblaze de la société Xilinx a été ciblé pour une métamorphose, afin qu'il puisse être inséré dans un système réalisant du HMT sur puce. Le Nios d'Altera ainsi que le PowerPC d'IBM faisaient partie des candidats disponibles.

Afin d'adapter un processeur comme celui choisi, il est nécessaire de le transformer pour qu'il réponde exactement aux attentes de la synchronisation. Ainsi, les signaux de sorties et ceux des entrées ont été inhibés, enregistrés, traités et/où décalés par un module enveloppe permettant ainsi de respecter la synchronisation adéquate pour ce projet. Une fois la métamorphose acquittée, le processeur s'exécute seulement durant une période distincte appelée la fenêtre d'exécution du processeur. De plus, si comme dans le cas du Microblaze, on retrouve un arbitre interne au processeur, il importe particulièrement d'en inhiber l'effet, afin d'éviter toute collision possible.

Finalement, la dernière étape avant les résultats consiste à adapter le canal de communication ainsi que tous les périphériques inclus dans le système, à ce mécanisme de synchronisation. Bien que cruciale, cette étape est plus ou moins triviale puisqu'il s'agit simplement de s'assurer que le délai entre l'entrée d'une requête et sa réponse respective soit de 4 cycles. Un simple pipelinage du flot de données permet de s'acquitter de cette tâche.

L'utilisation de multiprocessus matériel et surtout la substitution du mécanisme d'arbitration par une fenêtre d'opération, ont tout d'abord permis, dans un premier temps, d'optimiser drastiquement les accès à la mémoire du processeur. Effectivement, le fait que cette technique cache la latence, chaque accès à la mémoire et à différents périphériques est passé de 3 coups d'horloges à un seul coup. En réalité, il est à noter que ce nombre a plutôt augmenté d'un coup d'horloge, étant donné le pipelinage des périphériques, pour obtenir une latence totale de 4 coups rapides pour une transaction. Par contre, du point de vue du processeur, qui s'exécute 4 fois plus lentement, la réponse aux transactions apparaît au coup d'horloge suivant la requête, d'où la perception d'un seul coup.

Dans un second temps, ce mécanisme a permis d'observer des optimisations intéressantes lors de l'utilisation de périphériques lents ou physiquement éloignés. Ainsi, lors de

transactions apportant de longues latences de communication, le multiprocessus se distingue particulièrement au niveau de l'efficacité. Effectivement, de par sa nature qui tend à cacher les latences, les processeurs ne perçoivent qu'un quart du délai de la transaction atypique. Pendant ce temps d'attente, les trois autres processeurs poursuivent leur exécution de façon transparente.

À la lumière de cette information, trois tests ont été simulés à partir du modèle développé pour ce projet. Deux de ces tests ont été comparés avec un modèle de référence quasi identique à son opposant. Leurs seules différences résident dans le fait que le processeur inclus dans le système de référence n'utilise aucune forme de multiprocessus matériel.

Les deux premiers tests impliquent deux applications complètement différentes, permettant ainsi de voir l'effet du type d'application sur le gain obtenu par rapport au design de référence. En effet, comme on sait que le HMT optimise les transactions de données, une application purement orientée contrôle devrait moins subir l'effet de l'optimisation des transactions. Ainsi, pour une application typiquement orientée donnée, un gain d'un peu plus de 2 a été observé. L'ajout d'accès à un périphérique lent, a permis au gain de gagner quelques dixièmes. Bien que, pour l'application orientée contrôle, le gain se soit, comme prévu, montré un peu moindre, le comportement lors de l'ajout du périphérique lent est restée identique. En fait, les accès à ce périphérique auraient dû être beaucoup plus fréquents pour observer un gain plus significatif. En résumé, les gains obtenus lors de ces deux tests sont directement attribuables à l'optimisation des accès mémoires expliqués précédemment.

Bien que les deux objectifs fixés aient été atteints, il serait intéressant de mettre plus de temps sur l'intégration au FPGA. En fait, durant ce projet, un peu de temps a été utilisé pour l'intégration sur une puce physique. Cependant, après une première tentative succincte et vaine, il semble que les processeurs ne s'exécutaient pas. Ainsi, le problème provient sûrement des domaines de fréquences. De ce fait, avec l'utilisation d'outils plus

sophistiqués de débogage ainsi que plus de temps, il serait intéressant d'intégrer cette architecture au FPGA visé. De plus, il serait aussi intéressant d'évaluer une vraie application tel que nano click [QUIN03], qui permettrait de générer une application IPV4 pour la plateforme. Ainsi, on pourrait apercevoir les gains non pas pour une application de tests, mais pour une véritable application qu'on pourrait utiliser dans un contexte de système embarqués.

Finalement, ce projet pourrait être modifié de façon à évaluer les performances du HMT en se basant sur d'autres IPs. Cependant, il est important de noter que même avec le code source du Microblaze, les modules doivent être modifiés du a certains cas spéciaux du fonctionnement interne de l'OPB dont l'annulation et l'expiration. Ainsi, il serait intéressant de comparer une version de l'architecture composée de Nios II d'Altera sur un réseau sur puce complètement différent. Plusieurs combinaisons de modules pourraient ainsi amener cette architecture à un système mature, spécialisé et performant.

Références

[ACTE05] Voir le site Web de Actel : http://www.actel.com/

[ALTE99] ALTERA INC. 1999. "A View of the System-on-Chip World From a Programmable Logic Perspective". 7p. http://kontoret.webmaster.se/dockeeperfiles/340/998/AlteraAbstract.pdf (Page consultée le 2 septembre 2004).

[ALTE03] ALTERA INC. February 2003. "The Truth About Die Size: Comparing Stratix & Virtex-II Pro FPGAs". 6p. http://www.altera.com/literature/wp/wp_stx_compare.pdf (Page consultée le 7 septembre 2004).

[ALTE03a] ALTERA INC. 2003. "HardCopy device Handbook". 188p. http://www.altera.com/literature/hb/hrd/hc_h51001.pdf (Page consultée le 18 septembre 2004).

[ALTE04] Voir le site Web de Altera : http://www.altera.com/

[ALTE04a] ALTERA INC. "Avalon Bus Specification Reference Manual". 106p. http://www.altera.com/literature/manual/mnl_avalon_bus.pdf (page consultée le 18 octobre 2004).

[ALTE04b] ALTERA INC. " Nios 2 Reference Handbook Chapter 16-17", 136p. http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1_03.pdf (page consultée le 19 octobre 2004).

[ALTE04c] ALTERA INC. "Cyclone II Device Backgrounder". 5p. http://www.altera.com/literature/wp/wp-exc_backgrounder.pdf (page consultée le 4 septembre 2004).

[ALTE04d] ALTERA INC., "Excalibur Device Backgrounder ", 5p.,

http://www.altera.com/literature/wp/wp-cyc2_backgrounder.pdf (page consultée le 4 septembre 2004).

[ALKK90] ARGAWAL, A., LIM, B-H., KRANZ, D. and KUBIATOWICZ, J. 1990. "APRIL: A Processor Architecture for Multiprocessing". *IEEE*. 104-114.

[AMIR04] Voir le site web de Amirix : http://www.amirix.com/

[AMIR05] AMIRIX INC. "AP100 PCI Platform FPGA Development Board". 2p. http://www.amirix.com/downloads/developboard.pdf (page consultée le 20 décembre 2005).

[BEHR03] BEHROOZ Zahiri, 2003, " Structured ASICs: Opportunities and Challenges", Proceedings of the 21st International Conference on Computer Design 03.

[BERT03] BERTOLA Marc, 2003, "Conception, réalisation et étude d'une plate-forme générique basée sur le protocole AMBA AHB", Mémoire de maîtrise, Département de génie électrique, École Polytechnique de Montréal.

[BORA92] BOOTHE, B. and RANADE, A. 1992, "Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessors". *ACM*, 214-223.

[BYHO95] BYRD, G. T. and HOLLIDAY, M. A. 1995. "Multithreaded Processor Architectures". *IEEE*. 38-46.

[CYGW04] Voir le site Web de Cygwin : http://www.cygwin.com/

[DESL05] DESLAURIERS François, 2005, « Création d'un environnement de test pour l'évaluation de performance de réseaux intégrés sur puce », Mémoire de maîtrise, Département de génie informatique, École Polytechnique de Montréal.

[EEL98] EL-KHARASHI, Watheq. M., ELGUIBALY, Fayez., LI, Kin. F. 1998. "Multithreaded processors: the upcoming generation for multimedia chips", *IEEE Symposium on Advances in Digital Filtering and Signal Processing*, 111-115.

[EELS97] EGGERS, S. J., EMER, J. S., LEBY, H. M., STAMM, R. L., and TULLSEN, D. M. 1997. "Simultaneous multithreading: a platform for next-generation processors". *Micro, IEEE.* 17:5. 12-19.

[FAWC94] FAWCETT B.K. 1994, "Extending FPGA architectures to address new applications", WESCON/94. 4 pp.

[GHGM91] GUPTA, A., HENNESSY, J., GHARACHORLOO, K., MOWRY, T. and WEBER, W. D. 1991. "Comparative Evaluation of Latency Reducing and Tolerating Techniques". *ACM.* 254-263.

[GNU04] Voir le site Web de GNU : http://www.gnu.com/

[HAFU88] HALSTEAD, R. H. Jr. and FUJITA, T. 1988. "MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing". *IEEE.* 443-451.

[HEPA96] HENNESSY, J.L., PATTERSON, D. A. 1996. "Architecture des ordianteurs – Une approche quantitative 2e édition". Paris : Morgan Kaufmann Publishers, 72-76, 122-125.

[HKNM92] HIRATA, H., KIMURA, K., NAGAMINE, N., MOCHIZUKI, Y., NISHIMURA, A., NAKASE, Y. and NISHIZAWA, T. 1992. "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads". *ACM.* 136-145.

[IBM02] IBM, 2002, "On-Chip Peripheral Bus - Architecture Specifications - Version 2.1", 118p., http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/9A7AFA74DAD200D087256AB30005F0C8/$file/OpbBus.pdf (Page consultée le 20 avril 2005).

[IBM99] IBM, 1999, "The CoreConnect™ Bus Architecture", 8p., http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF77852569910050C0FB/$file/crcon_wp.pdf (Page consultée le 24 mai 2004).

[JERR04] JERRAYA A.A. 2004. "Long term trends for embedded system design". Digital System Design, 2004. Euromicro Symposium. 136-145.

[KHKA92] KIMURA, K., HIRATA, H., KIYOHARA, T., ASAHARA, S., SAGISHIMA, T., ONOYE, T. and SHIRAKAWA, I. 1992. "Evaluation Method of Microarchitecture for Multithreaded Processor". *IEEE*. 53-58.

[LEEL97] LO, J. L., EGGERS, S. J., EMER, J. S., LEVY, H. M., STAMM, R. L. and TULLSEN, D. M. 1997. "Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading". *ACM*. 322-354.

[LOBA96] LOIKKANEN, M. and BAGHERZADEH, N. 1996. "A Fine-Grain Multithrreaded Architecture". *IEEE*. 163-1.

[LYSA02] LYSAGHT Patrick, 2002. "FPGAs as Meta-platforms for Embedded Systems", Proceedings. 2002. IEEE International Conference. 7-12.

[MODE04] Voir le site Web de Model : http://www.model.com/

[PDWO92] PAPADOPOULOS, G. M., DAHBURA, A. T., WILLEM BÖHM, A. P. and OLDEHOEFT, R. R. 1992. "Minisymposium: Multithreaded Computer Systems". *IEEE*. 772-775.

[PATT85] PATTERSON, D. A., 1985, " Reduced instruction set computers ", Comm. *ACM*. 28:1, 8-21.

[QUIN03] QUINN, David, 2003, " Exploration architecturale pour la conception de processeurs réseaux basée sur l'utilisation de processeurs configurables",

Mémoire de maîtrise, Département de génie électrique, École Polytechnique de Montréal.

[SATH05] SAMSON, Patrick, THIBEAULT, Jean-Francois. 2005. " Adaptation d'une mémoire SDRAM sur une plateforme multi-processus materiel ".

[SILI02] SILICORE, 2002, " WISHBONE System-on-Chip (SoC)Interconnection Architecture for Portable IP Cores ", révision B.3, 14p. http://www.opencores.org/projects.cgi/web/wishbone/wbspec_b3.pdf (Page consultée le 13 août 2004)

[SMIT95] SMITH James E., 1995, " The Microarchitecture of Superscalar Processors ", Proceesings of the IEEE, vol. 83, no 12.

[SMSO95] SMITH, J. E. and SOHI, G. S. 1995. "The Microarchitecture of Superscalar Processors". *IEEE*. 1609-1624.

[SMWE94] SMITH, J. E. and WEISS, S. 1994. "PowerPc 601 and Alpha 21064: A Tale of Two RISCs". *IEEE*. 46-58.

[TUEG93] TULLSEN, D.M. and EGGERS, S. J.1993. "Limitation of Cache Prefetching on a Bus-Based Multiprocessor". *IEEE*. 278-288.

[TEL95] TULLSEN, D. M., EGGERS, S. J. and LEVY, H. M. 1995. "Simultaneous Multithreading: Maximizing On-Chip Parallelism". *ACM*. 392-403.

[THDS05] THIBEAULT, J.-F., HUBIN, M., DESLAURIERS F., SAMSON, P., BOIS, G. 2005, "A Reprogrammable SoC Design for a Real-Time Control Application". *MSE 2005*: 73-74

[USSE01] USSELMAN Rudolf, 2001, " OpenCores SOC Bus Review ", 13p. http://www.opencores.org/projects.cgi/web/wishbone/soc_bus_comparison.pdf (Page consultée le 13 août 2004)

[XILI04] Voir le site Web de Xilinx : http://www.xilinx.com/

[XILI04a] XILINX INC., " Virtex 4 : One family, multiple platforms ", 8p, http://www.xilinx.com/publications/prod_mktg/V4_brochure.pdf (page consultée le 18 septembre 2004).

[XILI04b] XILINX INC., " Microblaze Processor Reference guide ", 136p, http://www.eecg.toronto.edu/~pc/courses/432/2004/handouts/doc/mb_ref_guide.pdf (page consultée le 18 septembre 2004).

[XILI04c] XILINX INC., " Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link (FSL) Channel ", 12p, http://direct.xilinx.com/bvdocs/appnotes/xapp529.pdf (page consultée le 18 septembre 2004).

[XILI05] XILINX INC., "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet", 429 p, http://direct.xilinx.com/bvdocs/publications/ds083.pdf, page consultée le 20 décembre 2005.

[XILI06] XILINX INC., Virtex-5 Multi-Platform FPGA, http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex5/index.htm, page consultée le 24 juin 2006.

Annexes

ANNEXE A
Code du Mortiblaze

```
-------------------------------------------------------------------------------
-- mortiblaze.vhd
-------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_arith.all;
--use IEEE.std_logic_arith.all;
--use IEEE.std_logic_signed.all;
use IEEE.std_logic_misc.all;

-- synthesis translate_off
library UNISIM;
use UNISIM.VCOMPONENTS.ALL;

-- synthesis translate_on

library microblaze_v2_10_a;
use microblaze_v2_10_a.all;

entity mortiblaze is
    generic (
      C_FAMILY : string:="virtex2p";
      C_INSTANCE : string:="microblaze_0";
      C_INSTANCE_ID : integer range 0 to 3 := 1;
      C_D_OPB        : integer:=1;
      C_D_LMB        : integer:=0;
      C_I_OPB         : integer:=1;
      C_I_LMB : integer:=0;
      C_USE_BARREL : integer:=0;
      C_USE_DIV : integer:=0;
      C_DEBUG_ENABLED : integer:=1;
      C_NUMBER_OF_PC_BRK : integer:=2;
      C_NUMBER_OF_RD_ADDR_BRK : integer:=1;
      C_NUMBER_OF_WR_ADDR_BRK : integer:=1;
      C_INTERRUPT_IS_EDGE : integer:=0;
      C_EDGE_IS_POSITIVE : integer:=1;
      C_FSL_LINKS : integer:=0;
      C_FSL_DATA_SIZE : integer:=32;
      C_ICACHE_BASEADDR : std_logic_vector:=X"00000000";
      C_ICACHE_HIGHADDR : std_logic_vector:=X"3FFFFFFF";
      C_USE_ICACHE : integer:=0;
      C_ALLOW_ICACHE_WR : integer:=1;
      C_ADDR_TAG_BITS : integer:=7;
      C_CACHE_BYTE_SIZE : integer:=8192;
      C_DCACHE_BASEADDR : std_logic_vector:=X"00000000";
      C_DCACHE_HIGHADDR : std_logic_vector:=X"3FFFFFFF";
      C_USE_DCACHE : integer:=0;
      C_ALLOW_DCACHE_WR : integer:=1;
```

```
    C_DCACHE_ADDR_TAG : integer:=7;
    C_DCACHE_BYTE_SIZE : integer:=8192
  );
port (
  CLK : in std_logic;
  OPB_CLK : in std_logic;
  PHASE_CLK : in std_logic;
  OPB_RST : in std_logic;
  RESET : in std_logic;
  INTERRUPT : in std_logic;
  DEBUG_RST : in std_logic;
  EXT_BRK : in std_logic;
  EXT_NM_BRK : in std_logic;
  DBG_STOP : in std_logic;
  INSTR : in std_logic_vector(0 to 31);
  I_ADDRTAG : out std_logic_vector(0 to 3);
  IREADY : in std_logic;
  IWAIT : in std_logic;
  INSTR_ADDR : out std_logic_vector(0 to 31);
  IFETCH : out std_logic;
  I_AS : out std_logic;
  DATA_READ : in std_logic_vector(0 to 31);
  DREADY : in std_logic;
  DWAIT : in std_logic;
  DATA_WRITE : out std_logic_vector(0 to 31);
  DATA_ADDR : out std_logic_vector(0 to 31);
  D_ADDRTAG : out std_logic_vector(0 to 3);
  D_AS : out std_logic;
  READ_STROBE : out std_logic;
  WRITE_STROBE : out std_logic;
  BYTE_ENABLE : out std_logic_vector(0 to 3);
  DM_ABUS : out std_logic_vector(0 to 31);
  DM_BE : out std_logic_vector(0 to 3);
  DM_BUSLOCK : out std_logic;
  DM_DBUS : out std_logic_vector(0 to 31);
  DM_REQUEST : out std_logic;
  DM_RNW : out std_logic;
  DM_SELECT : out std_logic;
  DM_SEQADDR : out std_logic;
  DOPB_DBUS : in std_logic_vector(0 to 31);
  DOPB_ERRACK : in std_logic;
  DOPB_MGRANT : in std_logic;
  DOPB_RETRY : in std_logic;
  DOPB_TIMEOUT : in std_logic;
  DOPB_XFERACK : in std_logic;
  IM_ABUS : out std_logic_vector(0 to 31);
  IM_BE : out std_logic_vector(0 to 3);
  IM_BUSLOCK : out std_logic;
  IM_DBUS : out std_logic_vector(0 to 31);
  IM_REQUEST : out std_logic;
  IM_RNW : out std_logic;
  IM_SELECT : out std_logic;
  IM_SEQADDR : out std_logic;
  IOPB_DBUS : in std_logic_vector(0 to 31);
```

```
IOPB_ERRACK : in std_logic;
IOPB_MGRANT : in std_logic;
IOPB_RETRY : in std_logic;
IOPB_TIMEOUT : in std_logic;
IOPB_XFERACK : in std_logic;
DBG_CLK : in std_logic;
DBG_TDI : in std_logic;
DBG_TDO : out std_logic;
DBG_REG_EN : in std_logic_vector(0 to 4);
DBG_CAPTURE : in std_logic;
DBG_UPDATE : in std_logic;
VALID_INSTR : out std_logic;
PC_EX : out std_logic_vector(0 to 31);
REG_WRITE : out std_logic;
REG_ADDR : out std_logic_vector(0 to 4);
MSR_REG : out std_logic_vector(0 to 7);
NEW_REG_VALUE : out std_logic_vector(0 to 31);
PIPE_RUNNING : out std_logic;
INTERRUPT_TAKEN : out std_logic;
JUMP_TAKEN : out std_logic;
PREFETCH_ADDR : out std_logic_vector(0 to 3);
MB_Halted : out std_logic;
Trace_Branch_Instr : out std_logic;
Trace_Delay_Slot : out std_logic;
Trace_Data_Address : out std_logic_vector(0 to 31);
Trace_AS : out std_logic;
Trace_Data_Read : out std_logic;
Trace_Data_Write : out std_logic;
Trace_DCache_Req : out std_logic;
Trace_DCache_Hit : out std_logic;
Trace_ICache_Req : out std_logic;
Trace_ICache_Hit : out std_logic;
Trace_Instr_EX : out std_logic_vector(0 to 31);
FSL0_S_CLK : out std_logic;
FSL0_S_READ : out std_logic;
FSL0_S_DATA : in std_logic_vector(0 to 31);
FSL0_S_CONTROL : in std_logic;
FSL0_S_EXISTS : in std_logic;
FSL0_M_CLK : out std_logic;
FSL0_M_WRITE : out std_logic;
FSL0_M_DATA : out std_logic_vector(0 to 31);
FSL0_M_CONTROL : out std_logic;
FSL0_M_FULL : in std_logic;
FSL1_S_CLK : out std_logic;
FSL1_S_READ : out std_logic;
FSL1_S_DATA : in std_logic_vector(0 to 31);
FSL1_S_CONTROL : in std_logic;
FSL1_S_EXISTS : in std_logic;
FSL1_M_CLK : out std_logic;
FSL1_M_WRITE : out std_logic;
FSL1_M_DATA : out std_logic_vector(0 to 31);
FSL1_M_CONTROL : out std_logic;
FSL1_M_FULL : in std_logic;
FSL2_S_CLK : out std_logic;
```

```
FSL2_S_READ : out std_logic;
FSL2_S_DATA : in std_logic_vector(0 to 31);
FSL2_S_CONTROL : in std_logic;
FSL2_S_EXISTS : in std_logic;
FSL2_M_CLK : out std_logic;
FSL2_M_WRITE : out std_logic;
FSL2_M_DATA : out std_logic_vector(0 to 31);
FSL2_M_CONTROL : out std_logic;
FSL2_M_FULL : in std_logic;
FSL3_S_CLK : out std_logic;
FSL3_S_READ : out std_logic;
FSL3_S_DATA : in std_logic_vector(0 to 31);
FSL3_S_CONTROL : in std_logic;
FSL3_S_EXISTS : in std_logic;
FSL3_M_CLK : out std_logic;
FSL3_M_WRITE : out std_logic;
FSL3_M_DATA : out std_logic_vector(0 to 31);
FSL3_M_CONTROL : out std_logic;
FSL3_M_FULL : in std_logic;
FSL4_S_CLK : out std_logic;
FSL4_S_READ : out std_logic;
FSL4_S_DATA : in std_logic_vector(0 to 31);
FSL4_S_CONTROL : in std_logic;
FSL4_S_EXISTS : in std_logic;
FSL4_M_CLK : out std_logic;
FSL4_M_WRITE : out std_logic;
FSL4_M_DATA : out std_logic_vector(0 to 31);
FSL4_M_CONTROL : out std_logic;
FSL4_M_FULL : in std_logic;
FSL5_S_CLK : out std_logic;
FSL5_S_READ : out std_logic;
FSL5_S_DATA : in std_logic_vector(0 to 31);
FSL5_S_CONTROL : in std_logic;
FSL5_S_EXISTS : in std_logic;
FSL5_M_CLK : out std_logic;
FSL5_M_WRITE : out std_logic;
FSL5_M_DATA : out std_logic_vector(0 to 31);
FSL5_M_CONTROL : out std_logic;
FSL5_M_FULL : in std_logic;
FSL6_S_CLK : out std_logic;
FSL6_S_READ : out std_logic;
FSL6_S_DATA : in std_logic_vector(0 to 31);
FSL6_S_CONTROL : in std_logic;
FSL6_S_EXISTS : in std_logic;
FSL6_M_CLK : out std_logic;
FSL6_M_WRITE : out std_logic;
FSL6_M_DATA : out std_logic_vector(0 to 31);
FSL6_M_CONTROL : out std_logic;
FSL6_M_FULL : in std_logic;
FSL7_S_CLK : out std_logic;
FSL7_S_READ : out std_logic;
FSL7_S_DATA : in std_logic_vector(0 to 31);
FSL7_S_CONTROL : in std_logic;
FSL7_S_EXISTS : in std_logic;
```

```
    FSL7_M_CLK : out std_logic;
    FSL7_M_WRITE : out std_logic;
    FSL7_M_DATA : out std_logic_vector(0 to 31);
    FSL7_M_CONTROL : out std_logic;
    FSL7_M_FULL : in std_logic
  );
end mortiblaze;

architecture STRUCTURE of mortiblaze is

  component microblaze is
    generic (
      C_FAMILY : string;
      C_INSTANCE : string;
      C_D_OPB : integer;
      C_D_LMB : integer;
      C_I_OPB : integer;
      C_I_LMB : integer;
      C_USE_BARREL : integer;
      C_USE_DIV : integer;
      C_DEBUG_ENABLED : integer;
      C_NUMBER_OF_PC_BRK : integer;
      C_NUMBER_OF_RD_ADDR_BRK : integer;
      C_NUMBER_OF_WR_ADDR_BRK : integer;
      C_INTERRUPT_IS_EDGE : integer;
      C_EDGE_IS_POSITIVE : integer;
      C_FSL_LINKS : integer;
      C_FSL_DATA_SIZE : integer;
      C_ICACHE_BASEADDR : std_logic_vector;
      C_ICACHE_HIGHADDR : std_logic_vector;
      C_USE_ICACHE : integer;
      C_ALLOW_ICACHE_WR : integer;
      C_ADDR_TAG_BITS : integer;
      C_CACHE_BYTE_SIZE : integer;
      C_DCACHE_BASEADDR : std_logic_vector;
      C_DCACHE_HIGHADDR : std_logic_vector;
      C_USE_DCACHE : integer;
      C_ALLOW_DCACHE_WR : integer;
      C_DCACHE_ADDR_TAG : integer;
      C_DCACHE_BYTE_SIZE : integer
    );
    port (
      CLK : in std_logic;
      RESET : in std_logic;
      INTERRUPT : in std_logic;
      DEBUG_RST : in std_logic;
      EXT_BRK : in std_logic;
      EXT_NM_BRK : in std_logic;
      DBG_STOP : in std_logic;
      INSTR : in std_logic_vector(0 to 31);
      I_ADDRTAG : out std_logic_vector(0 to 3);
      IREADY : in std_logic;
      IWAIT : in std_logic;
      INSTR_ADDR : out std_logic_vector(0 to 31);
```

```
IFETCH : out std_logic;
I_AS : out std_logic;
DATA_READ : in std_logic_vector(0 to 31);
DREADY : in std_logic;
DWAIT : in std_logic;
DATA_WRITE : out std_logic_vector(0 to 31);
DATA_ADDR : out std_logic_vector(0 to 31);
D_ADDRTAG : out std_logic_vector(0 to 3);
D_AS : out std_logic;
READ_STROBE : out std_logic;
WRITE_STROBE : out std_logic;
BYTE_ENABLE : out std_logic_vector(0 to 3);
DM_ABUS : out std_logic_vector(0 to 31);
DM_BE : out std_logic_vector(0 to 3);
DM_BUSLOCK : out std_logic;
DM_DBUS : out std_logic_vector(0 to 31);
DM_REQUEST : out std_logic;
DM_RNW : out std_logic;
DM_SELECT : out std_logic;
DM_SEQADDR : out std_logic;
DOPB_DBUS : in std_logic_vector(0 to 31);
DOPB_ERRACK : in std_logic;
DOPB_MGRANT : in std_logic;
DOPB_RETRY : in std_logic;
DOPB_TIMEOUT : in std_logic;
DOPB_XFERACK : in std_logic;
IM_ABUS : out std_logic_vector(0 to 31);
IM_BE : out std_logic_vector(0 to 3);
IM_BUSLOCK : out std_logic;
IM_DBUS : out std_logic_vector(0 to 31);
IM_REQUEST : out std_logic;
IM_RNW : out std_logic;
IM_SELECT : out std_logic;
IM_SEQADDR : out std_logic;
IOPB_DBUS : in std_logic_vector(0 to 31);
IOPB_ERRACK : in std_logic;
IOPB_MGRANT : in std_logic;
IOPB_RETRY : in std_logic;
IOPB_TIMEOUT : in std_logic;
IOPB_XFERACK : in std_logic;
DBG_CLK : in std_logic;
DBG_TDI : in std_logic;
DBG_TDO : out std_logic;
DBG_REG_EN : in std_logic_vector(0 to 4);
DBG_CAPTURE : in std_logic;
DBG_UPDATE : in std_logic;
VALID_INSTR : out std_logic;
PC_EX : out std_logic_vector(0 to 31);
REG_WRITE : out std_logic;
REG_ADDR : out std_logic_vector(0 to 4);
MSR_REG : out std_logic_vector(0 to 7);
NEW_REG_VALUE : out std_logic_vector(0 to 31);
PIPE_RUNNING : out std_logic;
INTERRUPT_TAKEN : out std_logic;
```

```
JUMP_TAKEN : out std_logic;
PREFETCH_ADDR : out std_logic_vector(0 to 3);
MB_Halted : out std_logic;
Trace_Branch_Instr : out std_logic;
Trace_Delay_Slot : out std_logic;
Trace_Data_Address : out std_logic_vector(0 to 31);
Trace_AS : out std_logic;
Trace_Data_Read : out std_logic;
Trace_Data_Write : out std_logic;
Trace_DCache_Req : out std_logic;
Trace_DCache_Hit : out std_logic;
Trace_ICache_Req : out std_logic;
Trace_ICache_Hit : out std_logic;
Trace_Instr_EX : out std_logic_vector(0 to 31);
FSL0_S_CLK : out std_logic;
FSL0_S_READ : out std_logic;
FSL0_S_DATA : in std_logic_vector(0 to C_FSL_DATA_SIZE-1);
FSL0_S_CONTROL : in std_logic;
FSL0_S_EXISTS : in std_logic;
FSL0_M_CLK : out std_logic;
FSL0_M_WRITE : out std_logic;
FSL0_M_DATA : out std_logic_vector(0 to C_FSL_DATA_SIZE-1);
FSL0_M_CONTROL : out std_logic;
FSL0_M_FULL : in std_logic;
FSL1_S_CLK : out std_logic;
FSL1_S_READ : out std_logic;
FSL1_S_DATA : in std_logic_vector(0 to C_FSL_DATA_SIZE-1);
FSL1_S_CONTROL : in std_logic;
FSL1_S_EXISTS : in std_logic;
FSL1_M_CLK : out std_logic;
FSL1_M_WRITE : out std_logic;
FSL1_M_DATA : out std_logic_vector(0 to C_FSL_DATA_SIZE-1);
FSL1_M_CONTROL : out std_logic;
FSL1_M_FULL : in std_logic;
FSL2_S_CLK : out std_logic;
FSL2_S_READ : out std_logic;
FSL2_S_DATA : in std_logic_vector(0 to C_FSL_DATA_SIZE-1);
FSL2_S_CONTROL : in std_logic;
FSL2_S_EXISTS : in std_logic;
FSL2_M_CLK : out std_logic;
FSL2_M_WRITE : out std_logic;
FSL2_M_DATA : out std_logic_vector(0 to C_FSL_DATA_SIZE-1);
FSL2_M_CONTROL : out std_logic;
FSL2_M_FULL : in std_logic;
FSL3_S_CLK : out std_logic;
FSL3_S_READ : out std_logic;
FSL3_S_DATA : in std_logic_vector(0 to C_FSL_DATA_SIZE-1);
FSL3_S_CONTROL : in std_logic;
FSL3_S_EXISTS : in std_logic;
FSL3_M_CLK : out std_logic;
FSL3_M_WRITE : out std_logic;
FSL3_M_DATA : out std_logic_vector(0 to C_FSL_DATA_SIZE-1);
FSL3_M_CONTROL : out std_logic;
FSL3_M_FULL : in std_logic;
```

```
      FSL4_S_CLK : out std_logic;
      FSL4_S_READ : out std_logic;
      FSL4_S_DATA : in std_logic_vector(0 to C_FSL_DATA_SIZE-1);
      FSL4_S_CONTROL : in std_logic;
      FSL4_S_EXISTS : in std_logic;
      FSL4_M_CLK : out std_logic;
      FSL4_M_WRITE : out std_logic;
      FSL4_M_DATA : out std_logic_vector(0 to C_FSL_DATA_SIZE-1);
      FSL4_M_CONTROL : out std_logic;
      FSL4_M_FULL : in std_logic;
      FSL5_S_CLK : out std_logic;
      FSL5_S_READ : out std_logic;
      FSL5_S_DATA : in std_logic_vector(0 to C_FSL_DATA_SIZE-1);
      FSL5_S_CONTROL : in std_logic;
      FSL5_S_EXISTS : in std_logic;
      FSL5_M_CLK : out std_logic;
      FSL5_M_WRITE : out std_logic;
      FSL5_M_DATA : out std_logic_vector(0 to C_FSL_DATA_SIZE-1);
      FSL5_M_CONTROL : out std_logic;
      FSL5_M_FULL : in std_logic;
      FSL6_S_CLK : out std_logic;
      FSL6_S_READ : out std_logic;
      FSL6_S_DATA : in std_logic_vector(0 to C_FSL_DATA_SIZE-1);
      FSL6_S_CONTROL : in std_logic;
      FSL6_S_EXISTS : in std_logic;
      FSL6_M_CLK : out std_logic;
      FSL6_M_WRITE : out std_logic;
      FSL6_M_DATA : out std_logic_vector(0 to C_FSL_DATA_SIZE-1);
      FSL6_M_CONTROL : out std_logic;
      FSL6_M_FULL : in std_logic;
      FSL7_S_CLK : out std_logic;
      FSL7_S_READ : out std_logic;
      FSL7_S_DATA : in std_logic_vector(0 to C_FSL_DATA_SIZE-1);
      FSL7_S_CONTROL : in std_logic;
      FSL7_S_EXISTS : in std_logic;
      FSL7_M_CLK : out std_logic;
      FSL7_M_WRITE : out std_logic;
      FSL7_M_DATA : out std_logic_vector(0 to C_FSL_DATA_SIZE-1);
      FSL7_M_CONTROL : out std_logic;
      FSL7_M_FULL : in std_logic
    );
  end component;

  component FDC is
  port (
    Q : out std_logic;
    C : in  std_logic;
    D : in  std_logic;
    CLR : in  std_logic
  );
end component FDC;
```

```
type HMT_State_Machine is (I_Idle, I_Request_State, I_Ack_State,
Wait1_State, Wait2_State, Wait3_State, D_Idle, D_Request_State,
D_Ack_State, AckWait1_state, AckWait2_state, AckWait3_state);
signal HMT_state_current: HMT_State_Machine;
signal HMT_state_next   : HMT_State_Machine;
signal HMT_wait_next    : HMT_State_Machine;
signal HMT_wait_ack                : HMT_State_Machine;

-- Bus Parking here. Force MGrant to 1;
signal M_RESET                : std_logic;
signal MGrant                 : std_logic;
signal I_Grant                : std_logic;
signal D_Grant                : std_logic;

-- HMTGrant. For HMT Sync.
signal HMTGrant         : std_logic;
signal HMTGrant_edge    : std_logic;
signal HMTGrant_reset   : std_logic;

signal DM_ABUS_S        : std_logic_vector(0 to 31);
signal DM_ABUS_T        : std_logic_vector(0 to 31);
signal DM_DBUS_S        : std_logic_vector(0 to 31);
signal DM_BE_S          : std_logic_vector(0 to 3);
signal DM_BUSLOCK_S     : std_logic;
signal DM_RNW_S         : std_logic;
signal DM_SELECT_S      : std_logic;
signal DM_SEQADDR_S     : std_logic;
signal DM_REQUEST_S     : std_logic;
signal DOPB_DBUS_S      : std_logic_vector(0 to 31);
signal DOPB_DBUS_LATCH : std_logic_vector(0 to 31);
signal DOPB_ERRACK_S    : std_logic;
signal DOPB_TIMEOUT_S   : std_logic;
signal DOPB_RETRY_S     : std_logic;
signal DOPB_XFERACK_S   : std_logic;
signal DOPB_XFERACK_LATCH : std_logic;

signal IM_ABUS_S        : std_logic_vector(0 to 31);
signal IM_ABUS_T        : std_logic_vector(0 to 31);
signal IM_DBUS_S        : std_logic_vector(0 to 31);
signal IM_BE_S          : std_logic_vector(0 to 3);
signal IM_BUSLOCK_S     : std_logic;
signal IM_RNW_S         : std_logic;
signal IM_SELECT_S      : std_logic;
signal IM_SEQADDR_S     : std_logic;
signal IM_REQUEST_S     : std_logic;
signal IOPB_DBUS_S      : std_logic_vector(0 to 31);
signal IOPB_DBUS_LATCH : std_logic_vector(0 to 31);
signal IOPB_ERRACK_S    : std_logic;
signal IOPB_TIMEOUT_S   : std_logic;
signal IOPB_RETRY_S     : std_logic;
signal IOPB_XFERACK_S   : std_logic;
signal IOPB_XFERACK_LATCH  : std_logic;

begin
```

```
microblaze_0 : microblaze
  generic map (
    C_FAMILY => C_FAMILY,
    C_INSTANCE => C_INSTANCE,
    C_D_OPB => C_D_OPB,
    C_D_LMB => C_D_LMB,
    C_I_OPB => C_I_OPB,
    C_I_LMB => C_I_LMB,
    C_USE_BARREL => C_USE_BARREL,
    C_USE_DIV => C_USE_DIV,
    C_DEBUG_ENABLED => C_DEBUG_ENABLED,
    C_NUMBER_OF_PC_BRK => C_NUMBER_OF_PC_BRK,
    C_NUMBER_OF_RD_ADDR_BRK => C_NUMBER_OF_RD_ADDR_BRK,
    C_NUMBER_OF_WR_ADDR_BRK => C_NUMBER_OF_WR_ADDR_BRK,
    C_INTERRUPT_IS_EDGE => C_INTERRUPT_IS_EDGE,
    C_EDGE_IS_POSITIVE => C_EDGE_IS_POSITIVE,
    C_FSL_LINKS => C_FSL_LINKS,
    C_FSL_DATA_SIZE => C_FSL_DATA_SIZE,
    C_ICACHE_BASEADDR => C_ICACHE_BASEADDR,
    C_ICACHE_HIGHADDR => C_ICACHE_HIGHADDR,
    C_USE_ICACHE => C_USE_ICACHE,
    C_ALLOW_ICACHE_WR => C_ALLOW_ICACHE_WR,
    C_ADDR_TAG_BITS => C_ADDR_TAG_BITS,
    C_CACHE_BYTE_SIZE => C_CACHE_BYTE_SIZE,
    C_DCACHE_BASEADDR => C_DCACHE_BASEADDR,
    C_DCACHE_HIGHADDR => C_DCACHE_HIGHADDR,
    C_USE_DCACHE => C_USE_DCACHE,
    C_ALLOW_DCACHE_WR => C_ALLOW_DCACHE_WR,
    C_DCACHE_ADDR_TAG => C_DCACHE_ADDR_TAG,
    C_DCACHE_BYTE_SIZE => C_DCACHE_BYTE_SIZE
  )
  port map (
    CLK => CLK,
    RESET => M_RESET,
    INTERRUPT => INTERRUPT,
    DEBUG_RST => DEBUG_RST,
    EXT_BRK => EXT_BRK,
    EXT_NM_BRK => EXT_NM_BRK,
    DBG_STOP => DBG_STOP,
    INSTR => INSTR,
    I_ADDRTAG => I_ADDRTAG,
    IREADY => IREADY,
    IWAIT => IWAIT,
    INSTR_ADDR => INSTR_ADDR,
    IFETCH => IFETCH,
    I_AS => I_AS,
    DATA_READ => DATA_READ,
    DREADY => DREADY,
    DWAIT => DWAIT,
    DATA_WRITE => DATA_WRITE,
    DATA_ADDR => DATA_ADDR,
    D_ADDRTAG => D_ADDRTAG,
    D_AS => D_AS,
```

```
READ_STROBE => READ_STROBE,
WRITE_STROBE => WRITE_STROBE,
BYTE_ENABLE => BYTE_ENABLE,
DM_ABUS => DM_ABUS_T,
DM_BE => DM_BE_S,
DM_BUSLOCK => DM_BUSLOCK_S,
DM_DBUS => DM_DBUS_S,
DM_REQUEST => DM_REQUEST_S,
DM_RNW => DM_RNW_S,
DM_SELECT => DM_SELECT_S,
DM_SEQADDR => DM_SEQADDR_S,
DOPB_DBUS => DOPB_DBUS_S,
DOPB_ERRACK => DOPB_ERRACK_S,
DOPB_MGRANT => D_GRANT,
DOPB_RETRY => DOPB_RETRY_S,
DOPB_TIMEOUT => DOPB_TIMEOUT_S,
DOPB_XFERACK => DOPB_XFERACK_S,
IM_ABUS => IM_ABUS_T,
IM_BE => IM_BE_S,
IM_BUSLOCK => IM_BUSLOCK_S,
IM_DBUS => IM_DBUS_S,
IM_REQUEST => IM_REQUEST_S,
IM_RNW => IM_RNW_S,
IM_SELECT => IM_SELECT_S,
IM_SEQADDR => IM_SEQADDR_S,
IOPB_DBUS => IOPB_DBUS_S,
IOPB_ERRACK => IOPB_ERRACK_S,
IOPB_MGRANT => I_GRANT,
IOPB_RETRY => IOPB_RETRY_S,
IOPB_TIMEOUT => IOPB_TIMEOUT_S,
IOPB_XFERACK => IOPB_XFERACK_S,
DBG_CLK => DBG_CLK,
DBG_TDI => DBG_TDI,
DBG_TDO => DBG_TDO,
DBG_REG_EN => DBG_REG_EN,
DBG_CAPTURE => DBG_CAPTURE,
DBG_UPDATE => DBG_UPDATE,
VALID_INSTR => VALID_INSTR,
PC_EX => PC_EX,
REG_WRITE => REG_WRITE,
REG_ADDR => REG_ADDR,
MSR_REG => MSR_REG,
NEW_REG_VALUE => NEW_REG_VALUE,
PIPE_RUNNING => PIPE_RUNNING,
INTERRUPT_TAKEN => INTERRUPT_TAKEN,
JUMP_TAKEN => JUMP_TAKEN,
PREFETCH_ADDR => PREFETCH_ADDR,
MB_Halted => MB_Halted,
Trace_Branch_Instr => Trace_Branch_Instr,
Trace_Delay_Slot => Trace_Delay_Slot,
Trace_Data_Address => Trace_Data_Address,
Trace_AS => Trace_AS,
Trace_Data_Read => Trace_Data_Read,
Trace_Data_Write => Trace_Data_Write,
```

```
Trace_DCache_Req => Trace_DCache_Req,
Trace_DCache_Hit => Trace_DCache_Hit,
Trace_ICache_Req => Trace_ICache_Req,
Trace_ICache_Hit => Trace_ICache_Hit,
Trace_Instr_EX => Trace_Instr_EX,
FSL0_S_CLK => FSL0_S_CLK,
FSL0_S_READ => FSL0_S_READ,
FSL0_S_DATA => FSL0_S_DATA,
FSL0_S_CONTROL => FSL0_S_CONTROL,
FSL0_S_EXISTS => FSL0_S_EXISTS,
FSL0_M_CLK => FSL0_M_CLK,
FSL0_M_WRITE => FSL0_M_WRITE,
FSL0_M_DATA => FSL0_M_DATA,
FSL0_M_CONTROL => FSL0_M_CONTROL,
FSL0_M_FULL => FSL0_M_FULL,
FSL1_S_CLK => FSL1_S_CLK,
FSL1_S_READ => FSL1_S_READ,
FSL1_S_DATA => FSL1_S_DATA,
FSL1_S_CONTROL => FSL1_S_CONTROL,
FSL1_S_EXISTS => FSL1_S_EXISTS,
FSL1_M_CLK => FSL1_M_CLK,
FSL1_M_WRITE => FSL1_M_WRITE,
FSL1_M_DATA => FSL1_M_DATA,
FSL1_M_CONTROL => FSL1_M_CONTROL,
FSL1_M_FULL => FSL1_M_FULL,
FSL2_S_CLK => FSL2_S_CLK,
FSL2_S_READ => FSL2_S_READ,
FSL2_S_DATA => FSL2_S_DATA,
FSL2_S_CONTROL => FSL2_S_CONTROL,
FSL2_S_EXISTS => FSL2_S_EXISTS,
FSL2_M_CLK => FSL2_M_CLK,
FSL2_M_WRITE => FSL2_M_WRITE,
FSL2_M_DATA => FSL2_M_DATA,
FSL2_M_CONTROL => FSL2_M_CONTROL,
FSL2_M_FULL => FSL2_M_FULL,
FSL3_S_CLK => FSL3_S_CLK,
FSL3_S_READ => FSL3_S_READ,
FSL3_S_DATA => FSL3_S_DATA,
FSL3_S_CONTROL => FSL3_S_CONTROL,
FSL3_S_EXISTS => FSL3_S_EXISTS,
FSL3_M_CLK => FSL3_M_CLK,
FSL3_M_WRITE => FSL3_M_WRITE,
FSL3_M_DATA => FSL3_M_DATA,
FSL3_M_CONTROL => FSL3_M_CONTROL,
FSL3_M_FULL => FSL3_M_FULL,
FSL4_S_CLK => FSL4_S_CLK,
FSL4_S_READ => FSL4_S_READ,
FSL4_S_DATA => FSL4_S_DATA,
FSL4_S_CONTROL => FSL4_S_CONTROL,
FSL4_S_EXISTS => FSL4_S_EXISTS,
FSL4_M_CLK => FSL4_M_CLK,
FSL4_M_WRITE => FSL4_M_WRITE,
FSL4_M_DATA => FSL4_M_DATA,
FSL4_M_CONTROL => FSL4_M_CONTROL,
```

```
      FSL4_M_FULL => FSL4_M_FULL,
      FSL5_S_CLK => FSL5_S_CLK,
      FSL5_S_READ => FSL5_S_READ,
      FSL5_S_DATA => FSL5_S_DATA,
      FSL5_S_CONTROL => FSL5_S_CONTROL,
      FSL5_S_EXISTS => FSL5_S_EXISTS,
      FSL5_M_CLK => FSL5_M_CLK,
      FSL5_M_WRITE => FSL5_M_WRITE,
      FSL5_M_DATA => FSL5_M_DATA,
      FSL5_M_CONTROL => FSL5_M_CONTROL,
      FSL5_M_FULL => FSL5_M_FULL,
      FSL6_S_CLK => FSL6_S_CLK,
      FSL6_S_READ => FSL6_S_READ,
      FSL6_S_DATA => FSL6_S_DATA,
      FSL6_S_CONTROL => FSL6_S_CONTROL,
      FSL6_S_EXISTS => FSL6_S_EXISTS,
      FSL6_M_CLK => FSL6_M_CLK,
      FSL6_M_WRITE => FSL6_M_WRITE,
      FSL6_M_DATA => FSL6_M_DATA,
      FSL6_M_CONTROL => FSL6_M_CONTROL,
      FSL6_M_FULL => FSL6_M_FULL,
      FSL7_S_CLK => FSL7_S_CLK,
      FSL7_S_READ => FSL7_S_READ,
      FSL7_S_DATA => FSL7_S_DATA,
      FSL7_S_CONTROL => FSL7_S_CONTROL,
      FSL7_S_EXISTS => FSL7_S_EXISTS,
      FSL7_M_CLK => FSL7_M_CLK,
      FSL7_M_WRITE => FSL7_M_WRITE,
      FSL7_M_DATA => FSL7_M_DATA,
      FSL7_M_CONTROL => FSL7_M_CONTROL,
      FSL7_M_FULL => FSL7_M_FULL
    );


DM_ABUS_S <= DM_ABUS_T(0 to 17) & conv_std_logic_vector(C_INSTANCE_ID,
2) & DM_ABUS_T(20 to 31)
      WHEN (DM_ABUS_T(16) = '1' AND DM_ABUS_T(17) = '1' AND
or_reduce(DM_ABUS_T(0 to 15)) = '0')
else DM_ABUS_T;

IM_ABUS_S <= IM_ABUS_T(0 to 17) & conv_std_logic_vector(C_INSTANCE_ID,
2)(0 to 1) & IM_ABUS_T(20 to 31)
     WHEN (IM_ABUS_T(16) = '1' AND IM_ABUS_T(17) = '1' AND
or_reduce(IM_ABUS_T(0 to 15)) = '0')
     else IM_ABUS_T;

MGrant <= '1';

HMT_SM_COMB_PROC: process(HMTGrant, HMT_state_current, DM_SELECT_S,
IM_SELECT_S, DM_REQUEST_S, IM_REQUEST_S, IM_ABUS_S, IM_RNW_S, IM_BE_S,
DM_ABUS_S, DM_RNW_S, DM_BE_S, DOPB_ERRACK, DOPB_TIMEOUT, DOPB_RETRY,
DOPB_XFERACK_LATCH, DOPB_DBUS, IOPB_ERRACK, IOPB_TIMEOUT, IOPB_RETRY,
IOPB_XFERACK_LATCH, IOPB_DBUS) is
  Begin
```

```
HMT_state_next <= HMT_state_current;
DM_ABUS <= (others => '0');
DM_DBUS <= (others => '0');
DM_BE <= (others => '0');
DM_BUSLOCK <= '0';
DM_RNW <= '0';
DM_SELECT <= '0';
DM_SEQADDR <= '0';

DOPB_DBUS_S <= (others => '0');
DOPB_ERRACK_S <= '0';
DOPB_TIMEOUT_S <= '0';
DOPB_RETRY_S <= '0';
DOPB_XFERACK_S <= '0';

IM_ABUS <= (others => '0');
IM_DBUS <= (others => '0');
IM_BE <= (others => '0');
IM_BUSLOCK <= '0';
IM_RNW <= '0';
IM_SELECT <= '0';
IM_SEQADDR <= '0';

IOPB_DBUS_S <= (others => '0');
IOPB_ERRACK_S <= '0';
IOPB_TIMEOUT_S <= '0';
IOPB_RETRY_S <= '0';
IOPB_XFERACK_S <= '0';



case HMT_state_current is
   when I_Idle =>
      if IM_REQUEST_S = '1' AND DM_SELECT_S = '0' then
                  HMT_state_next <= I_Request_State;
                  D_GRANT <= '0';
                  I_GRANT <= '1';
      elsif IM_SELECT_S = '1'  AND DM_SELECT_S = '0' then
                  HMT_state_next <= I_Request_State;
      elsif DM_REQUEST_S = '1' AND IM_SELECT_S = '0' then
                  HMT_state_next <= D_Request_State;
                  D_GRANT <= '1';
      elsif DM_SELECT_S = '1'  AND IM_SELECT_S = '0' then
                  HMT_state_next <= D_Request_State;
      else
                  HMT_state_next <= I_Idle;
      end if;

   when I_Request_State =>
      if IM_SELECT_S = '1' AND HMTGrant = '1' then
                  IM_ABUS <= IM_ABUS_S;
                  IM_DBUS <= IM_DBUS_S;
                  IM_BE <= IM_BE_S;
```

```
                    IM_BUSLOCK    <= IM_BUSLOCK_S;
                    IM_RNW <= IM_RNW_S;
                    IM_SEQADDR <= IM_SEQADDR_S;
                    IM_SELECT <= IM_SELECT_S;
                    I_GRANT <= '0';
                    HMT_wait_next <= I_Ack_State;
                    HMT_state_next <= Wait1_State;
    elsif IM_SELECT_S = '0' then
                    HMT_state_next <= D_Idle;
    end if;

when Wait1_State => HMT_state_next <= Wait2_State;
when Wait2_State => HMT_state_next <= Wait3_State;
when Wait3_State => HMT_state_next <= HMT_wait_next;
when AckWait1_state => HMT_state_next <= AckWait2_state;
when AckWait2_state => HMT_state_next <= AckWait3_state;
when AckWait3_state => HMT_state_next <= HMT_wait_ack;
when I_Ack_State =>
    if IOPB_XFERACK_LATCH = '1' then
      IOPB_XFERACK_S <= IOPB_XFERACK_LATCH;
      IOPB_DBUS_S <= IOPB_DBUS;
      if DM_REQUEST_S = '1' then
          HMT_state_next <= D_Request_State;
          D_GRANT <= '1';
      elsif DM_SELECT_S = '1' then
          HMT_state_next <= D_Request_State;
      elsif IM_REQUEST_S = '1' AND DM_SELECT_S = '0' then
          HMT_state_next <= I_Request_State;
          I_GRANT <= '1';
      elsif IM_SELECT_S = '1'  AND DM_SELECT_S = '0' then
          HMT_state_next <= I_Request_State;
      else
          HMT_state_next <= D_Idle;
      end if;
    else
      HMT_wait_ack <= I_Ack_State;
      HMT_state_next <= AckWait1_state;
    end if;

when D_Idle =>
    DOPB_DBUS_S <= (others => '0');
    DOPB_ERRACK_S <= '0';
    DOPB_TIMEOUT_S <= '0';
    DOPB_RETRY_S <= '0';
    DOPB_XFERACK_S <= '0';
    D_GRANT <= '0';
    if DM_REQUEST_S = '1' AND IM_SELECT_S = '0' then
       HMT_state_next <= D_Request_State;
       D_GRANT <= '1';
       I_GRANT <= '0';
    elsif DM_SELECT_S = '1' AND IM_SELECT_S = '0'then
       HMT_state_next <= D_Request_State;
    elsif IM_REQUEST_S = '1' AND DM_SELECT_S = '0' then
       HMT_state_next <= I_Request_State;
```

```
                        I_GRANT <= '1';
                     elsif IM_SELECT_S = '1'  AND DM_SELECT_S = '0' then
                        HMT_state_next <= I_Request_State;
                     else
                        HMT_state_next <= D_Idle;
                     end if;

                  when D_Request_State =>
                     if DM_SELECT_S = '1' AND HMTGrant = '1' then
                        DM_ABUS <= DM_ABUS_S;
                        DM_DBUS <= DM_DBUS_S;
                        DM_BE   <= DM_BE_S;
                        DM_BUSLOCK <= DM_BUSLOCK_S;
                        DM_RNW                        <= DM_RNW_S;
                        DM_SEQADDR <= DM_SEQADDR_S;
                        DM_SELECT     <= DM_SELECT_S;
                        D_GRANT <= '0';
                        HMT_wait_next <= D_Ack_State;
                        HMT_state_next <= Wait1_State;
                     elsif DM_SELECT_S = '0' then
                        HMT_state_next <= I_Idle;
                     end if;
                  when D_Ack_State =>
                     if DOPB_XFERACK_LATCH = '1' then
                        DOPB_XFERACK_S <= DOPB_XFERACK_LATCH;
                        DOPB_DBUS_S <= DOPB_DBUS;
                       if IM_REQUEST_S = '1' then
                          HMT_state_next <= I_Request_State;
                          I_GRANT <= '1';
                       elsif IM_SELECT_S = '1' then
                          HMT_state_next <= I_Request_State;
                       elsif DM_REQUEST_S = '1' AND IM_SELECT_S = '0' then
                          HMT_state_next <= D_Request_State;
                          D_GRANT <= '1';
                       elsif DM_SELECT_S = '1'  AND IM_SELECT_S = '0' then
                          HMT_state_next <= D_Request_State;
--                        D_GRANT <= '1';
                       else
                          HMT_state_next <= I_Idle;
                       end if;
                     else
                        HMT_wait_ack <= D_Ack_State;
                        HMT_state_next <= AckWait1_state;
                     end if;
            end case;

   end process HMT_SM_COMB_PROC;

   M_RESET <= (not RESET) OR OPB_RST;
   HMT_SM_SEQ_PROC: process (OPB_Clk) is
   begin
      if OPB_Clk'event and OPB_Clk='0' then
         if M_RESET='1' then
            HMT_state_current <= I_Idle;
```

```
            else
               DOPB_XFERACK_LATCH <= DOPB_XFERACK;
               IOPB_XFERACK_LATCH <= IOPB_XFERACK;
               DOPB_DBUS_LATCH  <= DOPB_DBUS;
               IOPB_DBUS_LATCH  <= IOPB_DBUS;
               HMT_state_current <= HMT_state_next;
            end if;
         end if;
      end process HMT_SM_SEQ_PROC;


      HTMGrant_PROC: process(opb_clk, CLK) is
      begin
         if opb_clk'event and opb_clk = '0' then
            HMTGrant_edge <= '1';
            HMTGrant <= HMTGrant_edge;
            if CLK = '1' then
               HMTGrant_edge <= '0';
               HMTGrant <= '0';
            end if;
         end if;
      end process HTMGrant_PROC;

      DM_REQUEST <= '0';
      IM_REQUEST <= '0';

   end architecture STRUCTURE;
```