Université de Sherbrooke
Faculté de génie
Département de génie électrique et de génie informatique

Co-design matériel / logiciel à l'aide de FPGA - Étude de faisabilité d'un gestionnaire dynamique

Mémoire de maîtrise es sciences appliquées
Spécialité : génie électrique

Jie YANG

Sherbrooke (Québec), Canada juin 2002

RÉSUMÉ

Dans les systèmes ordinés actuels, il est courant d'utiliser des circuits additionnels pour effectuer des tâches requérant une puissance de calcul dépassant les capacités du microprocesseur central. L'utilisation de FPGA (*Field Programmable Gate Array*) est une solution intéressante au problème de l'achat de multiples cartes numériques et de leur désuétude rapide.

Le présent projet de maîtrise porte sur l'utilisation de FPGA comme source de matériel dédié à la demande pour des applications s'exécutant sur des microprocesseurs. Un gestionnaire de RPU (*Re-configurable Processing Unit*) qui permet d'automatiser l'utilisation et le partage d'une carte FPGA (la carte H.O.T. de la compagnie Xilinx qui contient un RPU XC6216) entre plusieurs applications indépendantes a été mis au point. Ce gestionnaire divise la surface du RPU en quatre sections et alloue les sections tout comme l'allocation de mémoire paginée dans un système d'exploitation. Deux algorithme d'allocations, l'algorithme LRU (*Least Recently Used*) et FIFO (*First In First Out*), ont été testés et comparés.

Pour tester le fonctionnement de ce système, plusieurs simulations ont été effectuées. Les résultats de simulations sont présentés. Suivent des analyses de performance d'un tel système matériel / logiciel. Finalement des perspectives de travaux futurs sont présentées.

REMERCIEMENTS

Je remercie tout d'abord mon directeur de maîtrise, Monsieur Frédéric Mailhot, pour les conseils, les encouragements et le support scientifique qu'il m'a donnés, ainsi que pour le précieux temps qu'il m'a accordé.

C'est avec plaisir que je remercie également Monsieur Daniel Dalle qui m'a toujours apporté son soutien durant ces quatre années d'étude que j'ai passées à l'Université de Sherbrooke.

Merci sincèrement à tous les professeurs et chargés de cours pour les connaissances scientifiques qu'ils m'ont transmises.

Merci aux équipes de soutien technique pour l'aide qu'elles m'ont apportée face aux fréquentes difficultés que j'ai rencontrées.

Je veux surtout remercier mon époux, mon ami et compagnon, Mario Laplante, qui a su m'encourager et me soutenir fidèlement. Sa contribution pour ce mémoire dépasse de loin ce qui pourrait être écrit ici.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

CHAPITRE 1 - INTRODUCTION

Au cours des 50 dernières années, la technologie de l'électronique numérique s'est développée de la façon exponentielle. Suivant la célèbre loi de Gordon Moore, la performance des microprocesseurs double à tous les 18 mois alors que leur taille et leur coût ne cesse de diminuer. Le véritable premier ordinateur numérique a été conçu par un mécanicien anglais du nom Charles Babbage (1792 - 1871). M. Babbage a passé sa vie et dépensé sa fortune pour fabriquer cette machine d'analyse. Mais elle n'a jamais fonctionné correctement, la technologie de son époque ne permettant pas d'avoir le matériel de haute précision nécessaire.

C'est au 20^{e} siècle, à partir du milieu des années 40, que des ordinateurs vraiment fonctionnels ont commencé à voir le jour. On divise généralement l'évolution des ordinateurs en quatre générations. La première génération couvre les années 1945 à 1955. Ces premières machines à calculer utilisaient des lampes à vide. Elles étaient énormes et les dizaines de milliers de lampes qu'elles comportaient remplissaient des pièces entières. Malgré leur dimension, ces appareils étaient des millions de fois plus lents que les ordinateurs les moins coûteux d'aujourd'hui.

L'invention des transistors, au milieu des années 50, a permis le développement d'une deuxième génération d'ordinateurs. Cette génération s'étend sur à peu près dix ans, soit de l'année 1955 à l'année 1965. La fiabilité de ces appareils de même que leur dimension plus réduite leurs permettaient d'être vendus à des clients. L'un des systèmes les plus répandus de cette époque, l'IBM 1401, permettait de lire des données à partir de cartes perforées, de les copier sur des rubans magnétiques et d'imprimer les résultats. L'IBM 7094 était quant à lui destiné aux calculs plus complexes.

La troisième génération d'ordinateurs couvre les années allant de 1965 à 1980. L'apparition des circuits intégrés, qui remplaçaient les transistors individuels, permettait

de réduire considérablement la taille des ordinateurs. Le IBM 360 fut le premier modèle d'ordinateur commercialisé utilisant des circuits intégrés. Il a connu un grand succès.

C'est à l'arrivée des circuits LSI (*Large Scale Integration*) au début des années 80 que l'informatique grand public comme on la connaît a véritablement débuté. L'ordinateur personnel, le PC (*Personal Computer*) constitue la quatrième génération d'ordinateurs. Si l'architecture d'un PC n'est pas très éloignée de celle d'un mini-ordinateur de classe PDP-11 de la troisième génération, son prix est toutefois bien inférieur. Contrairement aux générations d'ordinateurs précédentes, le PC est accessible à tous. Quant à ses performances, inutile de dire qu'elles ont décuplé, la fréquence des microprocesseurs ayant passé, entre l'année 1983 et aujourd'hui, de 4.77 MHz à 2.2 GHz. Le prix et les performances des micro-ordinateurs sont sans doute les deux principaux facteurs qui permettent d'expliquer la présence actuelle des systèmes ordinés dans toutes les sphères d'activité.

Avec l'évolution constante de ses composantes, l'architecture d'ordinateurs doit être constamment réexaminée et réinventée. Au cours de la dernière décennie, les processeurs superscalaires (*superscalars*) ont été développés et sont devenus le standard des ordinateurs de bureau (*desktop computers*). Un processeur simple contient une seule unité fonctionnelle (*functional unit*) nommé ALU (*Arithmetic and Logic Unit*) qui peut exécuter une seule instruction à la fois. Les processeurs plus sophistiqués comme les processeurs superscalaires et les processeurs VLIW (*Very Long Instruction Word*) permettent quant à eux d'exécuter plusieurs instructions en même temps en utilisant plusieurs unités fonctionnelles (UF) de façon à améliorer la performance. La figure 1-1 présente un diagramme qui illustre la structure commune d'un processeur superscalaire et un processeur VLIW. Le fichier des registres constitue le plus haut niveau dans la hiérarchie de la mémoire. Il est utilisé pour enregistrer les valeurs intermédiaires qui circulent entre les unités fonctionnelles.

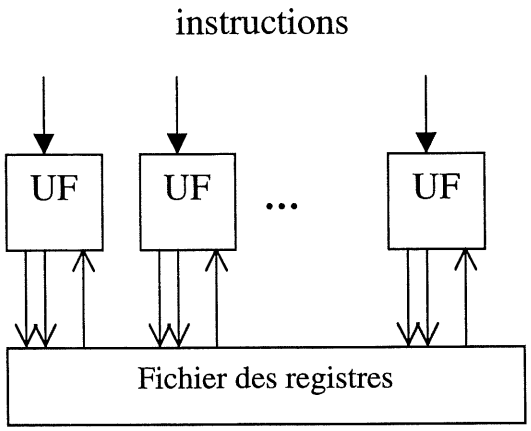

Figure 1-1 La structure commune d'un processeur superscalaire et VLIW

Les logiciels contiennent en gros trois catégories de parallélisme :

- Le parallélisme des processus allégés (*threads*) : il se trouve entre les processus allégés (*threads)* indépendants, chacun exécutant une séquence d'instructions séparée.
- Le parallélisme inter-itérations : il entre en jeu lorsque toutes les itérations d'une boucle sont indépendantes les unes des autres. Il est aussi appelé « parallélisme de données » (*data parallelism*) ou « parallélisme de vecteurs » (*vector parallelism*), qui est ciblé par les processeurs vectoriels.
- Le parallélisme au niveau des instructions (*ILP : Instruction Level Parallelism*) : il se manifeste lorsque des opérations sont exécutées à l'intérieur d'un flot de contrôle. Par exemple, dans l'expression : (a - b) * 3 + (a - c) * (b - c), il existe un ILP à 3-dimensions entre les trois soustractions, permettant à ces trois opérations de s'exécuter simultanément.

Les processeurs superscalaires et VLIW exploitent seulement l'ILP. Le principe du VLIW est de charger ses unités fonctionnelles des multiples opérations contenues dans une même instruction très longue. Le processeur superscalaire quant à lui, charge ses unités fonctionnelles de multiples instructions ordinaires. Par définition, un processeur superscalaire accepte une séquence d'instructions et découvre le parallélisme parmi ces

instructions de façon dynamique et automatique. Pour accroître la performance de ce type de processeur, il est possible d'augmenter le nombre d'unités fonctionnelles. Mais le temps requis par le système (*overhead*) pour détecter la dépendance entre les instructions augmente de façon quadratique avec le nombre d'unités fonctionnelles, indiquant que le superscalaire a déjà atteint ses limites. Les processeurs de type VLIW contournent ce problème en reléguant la tâche d'ordonnancement (*scheduling*) d'exécution des instructions aux compilateurs. En se chargeant de traiter les instructions ILP et de repérer à l'avance les fausses dépendances, les compilateurs permettent d'améliorer la performance des processeurs. Ces compilateurs sont toutefois complexes. Ils doivent non seulement tenir compte de la syntaxe et la sémantique d'un langage, mais ils doivent être adaptés aux caractéristiques techniques du processeur, de la mémoire et des unités fonctionnelles. Ce lien étroit entre les compilateurs et les caractéristiques du processeur fait en sorte qu'ils sont susceptibles d'être incompatibles avec les nouvelle versions de processeurs VLIW mis sur le marché (même si dans le sens conventionnel ces versions sont compatibles). Ces limites nous obligent à penser que le processeur de type VLIW ne constitue pas une solution d'avenir pour les ordinateurs personnels.

Les processeurs de type ASIC (*Application Specific Integrated Circuit*) constituent l'antithèse des processeurs génériques décrits plus haut. Ces processeurs, optimisés pour travailler avec une applications spécifique, permettent d'obtenir une excellente performance. Dans les systèmes ordinés actuels, il est courant d'utiliser des circuits additionnels pour effectuer des tâches requérant une puissance de calcul dépassant les capacités du microprocesseur central. Par exemple, l'ajout de cartes graphiques est devenu une opération essentielle pour utiliser efficacement les ordinateurs personnels. Dans les années qui viennent, la tendance à l'utilisation des cartes qui complémentent les capacités du microprocesseur se confirmera. Le décodage MPEG pour la lecture des DVDs, le chiffrement (*encryption*) et le déchiffrement (*decryption*), la compression de données pour l'utilisation des liens sans fils, sont autant des technologies qui requièrent déjà ou requérront dans un proche avenir des puissances de calcul qui doivent être soutenues par du matériel dédié approprié. Mais le coût de fabrication des ASIC étant de

plus en plus élevé, il ne serait pas réaliste de fabriquer un ASIC pour améliorer la performance de chaque application.

Les composants re-configurables constituent une autre alternative aux circuits ASIC et aux processeurs superscalaires et VLIW. Comme ils sont programmables, ils peuvent exécuter des instructions génériques au même titre que les processeurs. Contrairement à ces derniers toutefois, les instructions traitées par le matériel re-configurable sont entièrement chargées sur le matériel. À leur façon, les composants re-configurables imitent un circuit ASIC. Le chargement de configuration peut prendre de quelques cycles à des milliers de cycles d'horloge. La fameuse règle de 90-10 affirme que 90 % du temps d'exécution sont consommés par 10 % du code d'une application. Ces 10 % sont généralement constitués de boucles intérieures (*inner loops*). L'efficacité des systèmes ordinés augmente de façon appréciable lorsqu'on parvient à exécuter avec du matériel re-configurable toutes les boucles intérieures importantes d'une application. Ce type de matériel est particulièrement efficace lorsque les calculs à effectuer se répètent un nombre important de fois et que le temps de configuration est amorti par le temps d'exécution. Dans un contexte où la puissance de calculs exigée est importante, il offre une meilleure performance que les processeurs superscalaires et les processeurs VLIW. De plus, le matériel re-configurable permet non seulement d'exploiter l'ILP mais aussi le parallélisme des processus allégés et le parallélisme inter-itérations, ce qui lui promet un avenir encore plus prometteur que les processeurs superscalaires et VLIW.

Dans la famille des matériels re-configurable, les FPGA (*Field Programmable Gate Array*) sont les composants les plus utilisés dans l'industrie et dans le domaine de la recherche. La taille des FPGA nous permet actuellement de configurer une portion raisonnablement importante d'une application dans un seul FPGA. Cette possibilité ouvre les portes à un nouveau concept : un FPGA relié à un processeur pourrait, en théorie, servir de circuit spécifique pour chaque application ou même pour chaque étape de chaque application.

Le présent projet de maîtrise porte sur l'utilisation de FPGA comme source de matériel dédié à la demande pour des applications s'exécutant sur des microprocesseurs. Plus précisément, l'objet de cette étude consiste à mettre au point un système permettant d'automatiser l'utilisation et le partage de cartes FPGA entre plusieurs applications indépendantes.

La carte FPGA choisie est le Système H.O.T. (*Hardware Object Technology Developing System*) de la compagnie VCC (*Virtual Computer Company*). Elle contient une interface PCI, un élément configurable (XC6216 RPU), deux SRAM et un FPGA de la famille XC4000 qui gère la configuration en temps d'exécution du RPU (*Re-configurable Processing Unit*). Ce RPU comporte trois caractéristiques uniques:

- il a une architecture ouverte, qui permet le développement d'outils et de compilateurs ainsi que leur intégration;
- il est re-configurable dynamiquement et partiellement, cela permet de faire du matériel sur demande;
- il a une interface avec le microprocesseur, ce qui lui donne un temps de configuration avoisinant la milli-seconde.

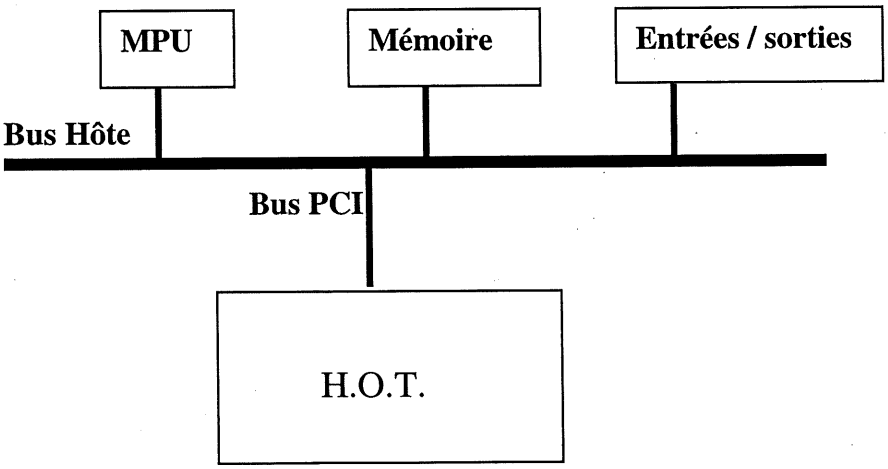

Figure 1-2 Connexion par bus PCI d'un PC au Système H.O.T.

De plus, la structure de la carte H.O.T. permet au système de fonctionner comme un co-processeur étroitement couplé avec le microprocesseur par le bus PCI, montré par la Figure 1-2. Ainsi, le Système H.O.T. possède toutes les caractéristiques nécessaires pour implémenter un système matériel / logiciel au niveau du co-processeur, ce qui permet à de multiples applications de partager une même puce FPGA de manière dynamique.

Au cours de ce projet, nous avons mis au point un mécanisme efficace de gestion de ce système H.O.T.. Ce mécanisme est rapide et simple d'utilisation, permettant à diverses applications d'utiliser de concert une seule carte FPGA re-programmable. Le système visé permet de partager diverses portions du RPU entre un ensemble de processus clients. Ce projet de recherche comprend les éléments suivants :

- Un gestionnaire de FPGA (analogue à un gestionnaire de mémoire virtuelle). Ce gestionnaire permet de diviser le RPU en plusieurs portions et de partager ces diverses portions entre plusieurs applications clientes. De plus, il peut s'adapter pour permettre aux applications requérant plus qu'une portion d'espace sur le RPU de s'exécuter en utilisant des blocs de taille plus importante sur le RPU.
- Une interface logicielle pour l'utilisation du gestionnaire par les applications.
- Un prototype du système comprenant le gestionnaire, et un ensemble de quelques processus clients pour démontrer le fonctionnement du système.
- Une étude de l'impact du système de partage de RPU sur la performance des processus clients.
- Une comparaison de quelques algorithmes de remplacement du matériel. Dans la mesure où le RPU est divisé en portions et que ces portions sont partagées entre de multiples processus clients, il y a des échanges de matériel. Tout comme dans le cas de la mémoire paginée il y a des fautes de page, le système utilisant un RPU produit des fautes de matériel. Ici on a utilisé des algorithmes de remplacement de matériel semblables aux algorithmes de remplacement de page. Une comparaison de l'algorithme LRU (*Least Recently Used*), FIFO (First In First Out) et deuxième chance est effectuée.

Ce mémoire est organisé de la façon suivante :

Le chapitre 2 présente une analyse de l'état de recherche dans le domaine de co-design matériel / logiciel à l'aide de FPGA. Nous commencerons par un bref historique du développement des FPGA. Nous traiterons ensuite des deux catégories de co-design matériel / logiciel. Finalement, nous ferons un survol des travaux effectués par plusieurs groupes de recherche dans ce domaine.

Nous expliquerons au chapitre 3 le fonctionnement du gestionnaire de FPGA implémenté dans notre projet. Nous commencerons par introduire le système H.O.T. de la compagnie VCC ciblé par notre gestionnaire, ensuite nous présenterons le JERC (*Java Environment for Re-configurable Computing*) basé sur lequel nous avons développé le gestionnaire. Finalement, nous démontrerons à l'aide d'exemples l'utilisation de ce gestionnaire.

Le chapitre 4 présente les résultats obtenus. Nous verrons ici comment se comparent les différents algorithmes de gestion de matériel que nous avons étudiés.

Le chapitre 5 enfin sera consacré à une analyse portant sur la possibilité d'utiliser l'architecture re-configurable comme source de matériel sur demande pour de multiples applications. Nous tenterons également de tracer quelques directions susceptibles d'être empruntées par cette technologie.

CHAPITRE 2 - FONDEMENTS ET ANALYSE DE L'ÉTAT DE LA RECHERCHE

2.1 Les différents type de composants programmables

Les FPGA (*Field Programmable Gate Arrays*) sont une classe de composants programmables et peuvent, à ce titre, être configurés pour une grande variété d'applications. Ils sont le résultat de plusieurs années de recherche.

Les PROMs (*Programmable Read Only Memory*) ont été les premiers composants programmables à avoir été couramment utilisés. Ils se sont développés sous deux formes : les EPROMs (*Erasable Programmable Read Only Memory*) et les EEPROMs (*Electrically Erasable Programmable Read Only Memory*). Dans les deux cas, il s'agit de puces constituées de mémoire morte effaçable et programmable. Ce qui distingue ces deux types de puces réside en grande partie dans le mode d'effacement de leur contenu. L'effacement des EPROMs est non sélective, demande plusieurs minutes et se fait au moyen d'un appareil produisant des rayons ultraviolets. Même si cette procédure n'était pas très efficace, ce type de puces a été très utilisé à une certaine époque. Les EEPROMs quant à eux s'effacent de façon sélective en quelques secondes par un procédé électrique. De nos jours, c'est ce type de mémoire qui est utilisé pour les BIOS (*Basic Input/Output System*) de la plupart des cartes mères et pour celui d'un certain nombre de contrôleurs SCSI (*Small Computer System Interface*).

Les PLD (*Programmable Logic Devices*), aussi nommés SPLD (*Simple Programmable Logic Devices*), ont constitué la deuxième génération de composants programmables. Ce sont les plus petits et les moins chers des circuits logiques programmables. Un PLD typique contient de 4 à 22 macro-cellules (*macrocell*) et chacune d'elles peut implémenter une fonction logique combinatoire de forme « somme de produits ». Ces macro-cellules sont généralement interconnectées de telle sorte qu'un PLD peut, à titre

d'exemple, remplacer plusieurs composants TTL de la série 7400. Ils sont destinés aux petits circuits logiques.

Au début des années 80, le développement des outils de synthèse logique permit de dessiner des circuits logiques de manière automatisée. À ce chapitre, le système LSS (*Logic Synthesis System*) de la compagnie IBM et le système SOCRATES de la compagnie GE (*General Electric*) ont été parmi les premiers outils de synthèse logique à connaître du succès. Ces outils rendent la synthèse de l'ASIC beaucoup plus facile. En 1987, les architectes de SOCRATES ont acheté les droits du produit qu'ils avaient mis au point chez GE et fondé la compagnie Synopsys, qui est depuis le leader mondial dans le domaine. Les outils de synthèse sont devenus des éléments essentiels dans le design des circuits actuels, des circuits qui comptent souvent des millions de portes logiques. C'est l'existence de ces outils de design automatisés qui a permis le développement des FPGAs. Xilinx, la compagnie qui, en 1984, introduisit les premiers FPGA, s'est en effet rapidement associée à des compagnies offrant des systèmes de synthèse automatisée pour la conception de ses produits. Cette association aura porté fruit puisque les FPGAs qu'elle a permis de concevoir ont connu de grands succès dans l'industrie et le secteur de l'enseignement.

2.2 Les différents types de FPGA

Selon leur mode d'interconnections et leur méthode de configuration, les FPGAs peuvent être classés sous quatre grandes catégories. On retrouve des FPGAs en rangées symétriques (*symmetrical array*), des FPGAs en ligne (*row-based*), des FGPAs à base de PLDs hiérarchiques (*hierarchical programmable logical unit*) et des FPGAs conçus sur la technologie des prédiffusés (*sea-of-gate*).

Les FPGAs peuvent également être classés selon le type de composants électroniques dont ils sont constitués. On en retrouve à base de cellules de RAM statique (SRAM), d'anti-fusibles (*anti-fuse*) ainsi que de transistors EPROMs et EEPROMs. Chacune de ces technologies répond à des applications spécifiques.

La technologie SRAM

Les connexions programmables que l'on retrouve dans les FPGAs de type SRAM sont réalisées au moyen de transistors de passage, des portes de transmission ou des multiplexeurs contrôlés par des cellules SRAMs. L'avantage de cette technologie est qu'elle permet la reconfiguration rapide des circuits. En revanche, les SRAMs utilisent de six à huit transistors par cellule de mémoire, ce qui oblige à fabriquer des puces d'une taille relativement importante et dont la capacité est moindre que celle d'autres types de FPGAs utilisant le même nombre de transistors.

La technologie anti-fusible

Un anti-fusible demeure à un état de haute impédance. Il peut être configuré à un état de basse impédance qui forme une connexion ou à un état « fused » qui coupe la connexion. La technologie anti-fusible ressemble à la technologie *fuse*. La différence est qu'au lieu de couper une connexion de métal par le passage d'un courant, elle établit une connexion, d'où vient le nom anti-fuse. Les anti-fuses sont des connexions soit de silicium amorphe, soit de métal. Ils sont généralement très petits et ont une résistance basse. Cette technologie est moins coûteuse que la technologie SRAM, mais le composant ne peut être programmé qu'une seule fois.

Les transistors EPROMs/EEPROMs

La technologie à la base des FPGAs à transistors EPROM et EEPROM est similaire à la technologie utilisée pour les puces de mémoire du même type. Les FPGAs qu'elle permet de fabriquer présentent l'avantage de pouvoir être reprogrammés sans stockage externe de la configuration. Mais le temps de configuration est assez long et il est impossible d'effacer l'information de façon sélective.

Ce sont les FPGAs à base de SRAM dont nous allons surtout traiter dans ce mémoire. Ce type de puces représente une alternative intéressante aux ASIC CMOS puisque leur coût et leur délai de fabrication sont beaucoup moindres. De plus, on peut les configurer un nombre illimité de fois par un simple chargement des données de configuration dans les

cellules de mémoire interne de la puce. Par contre, ils sont généralement plus lents que les ASIC CMOS et offrent une moins grande capacité.

Comme on le voit dans la figure 2-1, les FPGAs contiennent trois éléments configurables : les blocs de logique configurables (CLB), les blocs d'entrées / sorties (IOB) et les interconnexions. Les CLBs fournissent les éléments fonctionnels servant à construire la logique des usagers alors que les IOBs jouent le rôle d'interface entre les pattes de la puce et les signaux internes. Les ressources d'interconnexion programmables assurent quant à elles la connexion entre les entrées et les sorties des CLBs et des IOBs. La configuration s'effectue en programmant les cellules de mémoire interne. Ce sont ces cellules qui gèrent les fonctions logiques et les connexions internes de la puce.

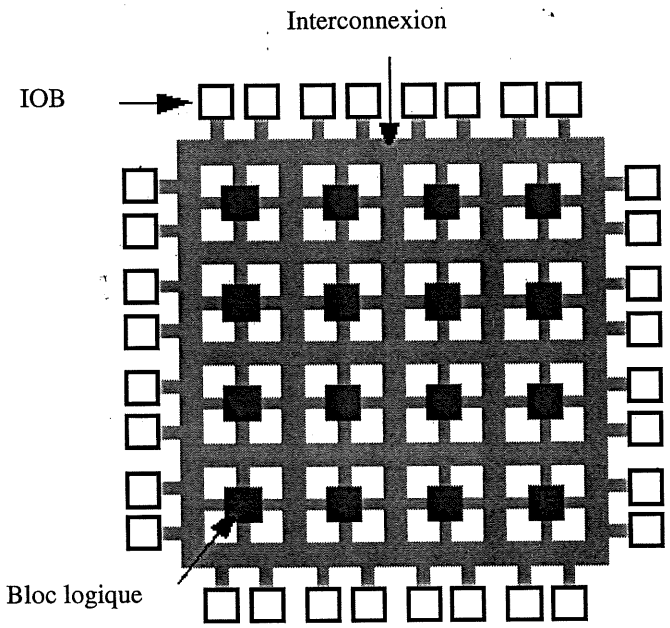

Figure 2-1 Le FPGA (tiré de [Xilinx])

2.3 L'historique du co-design matériel / logiciel

Depuis la fin des années 80, le co-design matériel / logiciel est un sujet de recherche populaire dans le domaine de la conception assistée par ordinateur (CAD : *Computer Aided Design*). Le co-design matériel / logiciel réfère à la prise en compte simultanée de l'aspect matériel et logiciel dans le processus de design d'un système. De façon générale, on cherche à combiner dans un même produit la flexibilité du logiciel à la performance du matériel. Jusqu'à présent, la plupart des systèmes matériel / logiciel ont été des systèmes embarqués. Mais, dans le secteur de la recherche, le co-design matériel / logiciel s'étend également aux systèmes génériques (*General Purpose System*) et aux systèmes DSP (*Digital Signal Processing system*).
Il existe quatre approches pour implémenter une application dans un système :

ASIC : *Application Specific Integrated Circuit* (Circuit Intégré à Application Spécifique)
ASIP : MPU spécifique + logiciel
MPU + ASIC : MPU générique + ASIC + logiciel
LOGICIEL : MPU générique + logiciel

L'approche ASIC implémente l'application entièrement sous forme de matériel. Cette approche est souvent utilisée pour des systèmes de haute performance, comme par exemple, les systèmes DSPs. Le transfert de l'application en matériel est réalisé par l'entremise d'outils de synthèse de haut niveau qui permettent de lire une description HDL (*Hardware Description Language*) et de produire l'architecture détaillée d'un circuit prêt à être fabriqué.

L'approche ASIP consiste à construire un processeur, appelé ASIP (*Application Specific Integrated Processor*), dont l'ensemble des instructions, le nombre de registres et la structure du processeur sont adaptés à une application spécifique. Ensuite, l'application est implémentée en logiciel pour ce processeur.

L'approche MPU + ASIC est de construire un co-processeur en ASIC pour une application spécifique. Un processeur ordinaire est utilisé comme processeur central (*core processor*). Certaines fonctions de l'application sont implémentées en matériel, avec un ou plusieurs ASIC, d'autres sont implémentées en logiciel. Le ou les ASIC sont typiquement connectés avec le processeur par un bus du système.

L'approche logicielle consiste à concevoir l'application entièrement sous forme de logiciel qu'un processeur ordinaire exécute.

TABLEAU 2-1 COMPARAISON DES DIFFÉRENTES APPROCHES DE DESIGN DE SYSTÈMES

Compromis	ASIC	ASIP	MPU+ASIC	Logiciel
Performance	La meilleure	Élevée	Moyenne	Moyenne
Puissance consommée	La plus faible	Moyenne à faible	Moyenne	Élevée
Flexibilité	Faible	Élevée	Élevée	Moyenne
Temps de design	Long	Le plus long	Moyen	Court

De Micheli [Mich94] a comparé ces quatre approches, comme le monte le tableau 2-1. L'approche ASIC offre la meilleure performance, mais la moins bonne flexibilité et son coût est relativement élevé. L'approche ASIP offre aussi une bonne performance, le processeur étant entièrement dédié à l'application. Mais la nécessité de dessiner le matériel et de concevoir un logiciel en fait une technologie très coûteuse. Le fait que seul le coprocesseur doit être dessiné avec l'approche MPU + ASIC permet de raccourcir le temps du design par rapport à l'approche ASIP mais elle ne permet par contre que d'obtenir une performance moyenne. L'approche logicielle permet d'obtenir une performance moyenne dans un temps de design court.

Des quatre approches présentées ci-dessus, seules les technologies ASIP et MPU + ASIC font partie du co-design matériel / logiciel. Ces deux approches impliquent en effet que les concepteurs ont à planifier la division entre les fonctions qui seront gérées par le matériel et celles qui seront gérées par le logiciel.

Il existe deux catégories de co-design matériel / logiciel : le co-design matériel / logiciel au niveau du processeur et le co-design matériel / logiciel au niveau du coprocesseur. Le co-design au niveau du processeur permet aux concepteurs de concevoir l'architecture du processeur en fonction d'une application spécifique. La technologie ASIP est un exemple de ce type de co-design. L'autre type de co-design matériel / logiciel (le co-design qui fait appel à un coprocesseur) oblige les concepteurs à utiliser un processeur noyau dont ils ne peuvent changer l'architecture. Cette technologie implique la conception d'un circuit intégré spécifique à l'application (ASIC) et utilisé comme un coprocesseur. L'approche MPU + ASIC est un exemple de ce type de co-design.

Dans le présent projet, nous nous intéresserons surtout au co-design matériel / logiciel faisant appel à une carte FPGA jouant le rôle d'un coprocesseur relié à un microprocesseur conventionnel. Lorsqu'elle est implémentée dans une carte FPGA, la partie matérielle d'un système co-design permet non seulement d'obtenir une performance qui se rapproche beaucoup de celle offerte par un processeur ASIC, mais le temps de configuration d'un tel système est beaucoup plus court que le temps de fabrication d'un ASIC. De plus, une carte FPGA a la caractéristique d'être re-programmable, ce qui permet aux différentes applications de partager la même puce.

La figure 2-2 illustre la méthodologie de base du co-design matériel / logiciel au niveau du co-processeur. Une spécification du comportement du système est divisée en matériel et logiciel pour satisfaire les exigences de performance. Le matériel et le logiciel sont ensuite développés, puis ils sont évalués pour vérifier si la performance du système est satisfaisante.

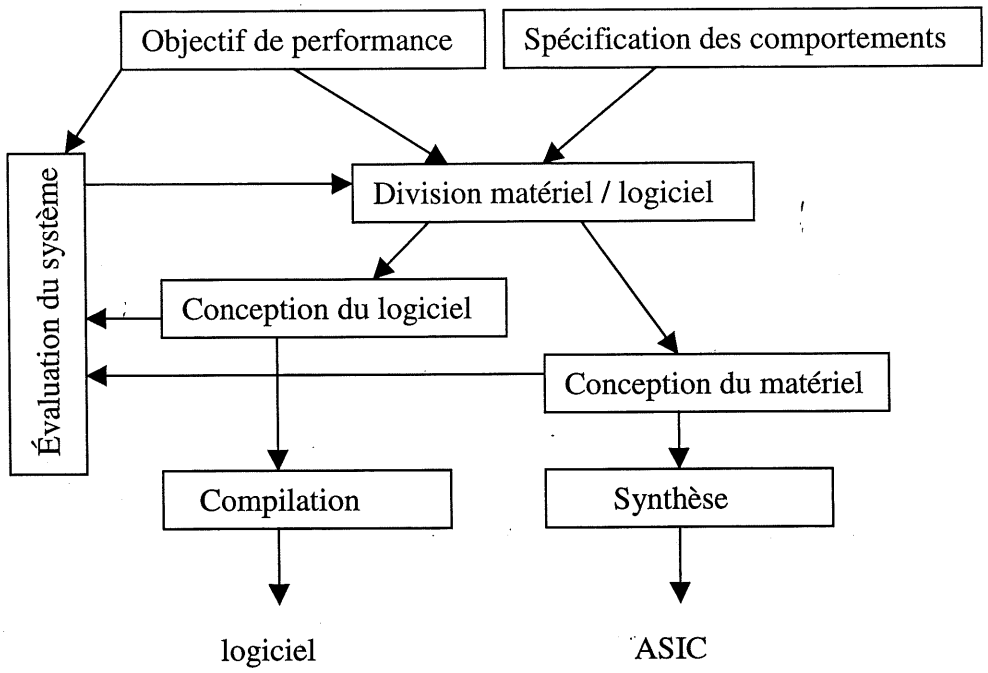

Figure 2-2 La méthodologie du co-design matériel / logiciel au niveau du co-processeur

La figure 2-3 illustre l'architecture de base d'un tel système. Comme on le voit, on y retrouve généralement un processeur ordinaire, des ASIC (co-processeurs) et de la mémoire. Toutes ces composantes sont connectées par un bus du système.

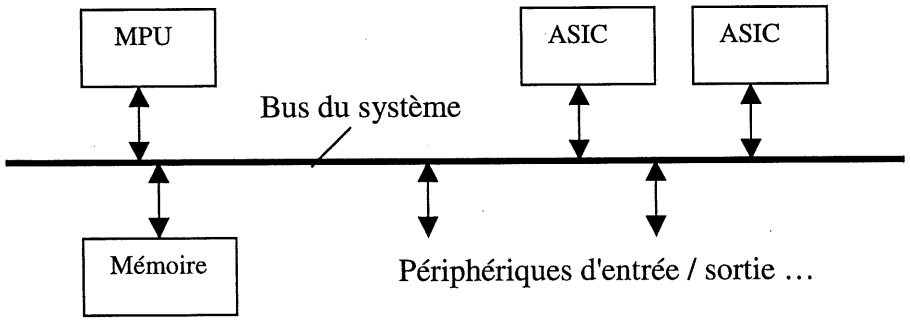

Figure 2-3 L'architecture d'un système co-design au niveau du co-processeur

2.4 Le co-design à base de FPGA - État de la recherche

Nous présentons ici un survol des principaux travaux de recherche qui ont marqué l'histoire récente du co-design matériel / logiciel faisant appel aux FPGAs.

Au début des années 90, l'équipe de co-design matériel / logiciel de l'Université de Berkeley en Californie a développé un système appelé POLIS, une plate-forme de co-design matériel / logiciel destinée aux systèmes embarqués [POLIS]. Le développement de ce système s'attaquait à trois des principaux problèmes du co-design des systèmes embarqués :

- Le manque de représentation unifiée du matériel / logiciel, qui rend difficile la vérification du système complet et l'incompatibilité à la frontière du matériel et du logiciel.
- La division prédéfinie entre les niveaux matériel et logiciel qui entraîne une conception sous-optimale.
- L'absence d'une méthodologie de design bien établie, qui rend la révision de la spécification difficile et qui allonge le temps de développement.

La plate-forme POLIS permet de faire du co-design avec un outil unifié et à partir d'une représentation unifiée du matériel et du logiciel. Cela permet de ne pas préjudicier l'implémentation de matériel ni l'implémentation de logiciel. Ce modèle est maintenu pendant toute la procédure de développement afin de préserver les caractéristiques formelles du design.

La représentation unifiée de POLIS est une machine à états finis du co-design (CFSM : *Codesign Finite State Machine*). Chaque élément du CFSM décrit une composante à modéliser. On trouvera ci-dessous les grandes étapes qu'implique le développement d'un système sous la plate-forme POLIS :

1. Spécification : les concepteurs écrivent une spécification en un langage de haut niveau (ESTEREL, FSM graphique, sous-ensemble de Verilog ou VHDL). Cette spécification est par la suite traduite en CFSM.
2. Vérification formelle.
3. Co-simulation du système.
4. Division matériel / logiciel et sélection de l'architecture et du répartiteur.
5. Synthèse de matériel : à l'aide du système de synthèse logique SIS, le sous-ensemble de la CFSM choisi pour le matériel est implémenté et optimisé.
6. Synthèse du logiciel : le sous-ensemble de la CFSM choisie pour le logiciel est implémenté, ce qui implique, entre autres, le développement d'un système d'exploitation spécifique doté d'un répartiteur et de pilotes d'entrée / sortie.
7. Implémentation d'interface : l'interface reliant le matériel et le logiciel est synthétisée de façon automatique.

L'arrivée des FPGAs a grandement favorisé le développement du co-design matériel / logiciel. La flexibilité des FPGAs et la rapidité de leur configuration ont intéressé de nombreux chercheurs de partout dans le monde et de plus en plus de prototypes de systèmes matériel / logiciel utilisant des FPGAs plutôt que des ASICs font leur apparition.

L'équipe de Fleischmann [FlBu97] fut l'une des premières à utiliser Java comme langage de spécification. Ses travaux ont conduit au développement d'une méthodologie de design d'un système matériel / logiciel qui a conduit à la conception d'un prototype d'une plate-forme de co-émulation où la partie logicielle s'exécutait sur une JVM (*Java Virtual Machine*) et la partie matérielle, sur une carte FPGA. Le choix de Java comme langage de spécification tenait au fait qu'il s'agit d'un langage *multi-thread* et qu'il peut exprimer la concurrence et les différents flots de contrôle dans un système. De plus, il satisfait à toutes les autres caractéristiques exigées pour un langage de spécification.

La première étape de cette méthodologie de design consiste à séparer les fonctions matérielles et logicielles sur la base d'une spécification de Java. Une description

d'interface est générée automatiquement pour chacune des fonctions à être implémentées en matériel. Ces fonctions sont par la suite synthétisées à l'aide d'un outil de synthèse de haut niveau, puis elles sont optimisées et des fichiers *netlist* sont générés. Finalement, les données de configuration du FPGA ciblé sont obtenues à l'aide des outils CAD de la compagnie Xilinx. Entre temps, les fonctions logicielles sont compilées par un compilateur conventionnel de Java. C'est le module RTS (*Run Time System*) qui se charge de gérer l'exécution et l'interaction entre les fonctions logicielles et matérielles. Les exécutables des fonctions logicielles sont stockés sous format *bytecode* de Java alors que les fonctions matérielles sont stockées sous forme de fichiers de bits de configuration.

Durant la co-simulation, le module RTS procède à la répartition des tâches entre les niveaux matériel et logiciel selon la table de partition courante. Lorsque le flot de contrôle dans un processus poids-plume (*thread*) rencontre une fonction à être gérée par le matériel, le module RTS détermine d'abord si le fichier de configuration est déjà chargé sur un FPGA. S'il ne l'est pas, le RTS détermine le FPGA disponible et démarre le mécanisme de configuration. S'il l'est, le RTS détermine l'adresse du FPGA qui contient cette fonction et démarre le transfert des données à être traitées par la fonction. Quand le processus est terminé, un signal d'interruption est envoyé au RTS. Un seul *thread* peut accéder au matériel à la fois. Ce *thread* est suspendu pendant qu'il est en attente de la réponse du matériel. Une étape de profilage permet de mesurer le temps d'exécution en matériel, en logiciel et le temps de communication entre ces deux niveaux. Les résultats sont retournés directement à l'étape de la partition pour d'éventuelles améliorations de la performance.

Dans le projet Hades [Ludw96], Ludwig et ses collaborateurs ont utilisé le FPGA CX6216 de la compagnie Xilinx pour implémenter un coprocesseur. Les FPGAs de la famille XC6200 sont des SRAMs programmables dotés de cellules de fine granulométrie. Chaque cellule peut être configurée pour effectuer n'importe quelle fonction logique impliquant deux entrées ou un multiplexeur. Ces cellules comportent également un registre permettant de les configurer pour satisfaire à des fonctions séquentielles. À partir

d'une machine hôte, on peut accéder au FPGA XC6200 comme s'il s'agissait d'une mémoire SRAM conventionnelle avec des signaux d'adresse, de donnée et de contrôle. La configuration s'effectue en faisant lire et écrire sur des registres de configuration. Il faut moins d'une milliseconde pour configurer la puce, ce qui est beaucoup plus rapide que le temps exigé pour la configuration de FPGAs ordinaires. C'est la principale raison pour laquelle Ludwig et son équipe ont opté pour le FPGA XC6216 dans leur projet de développement d'un coprocesseur. La machine hôte utilisée est un Ceres-2, une machine cadencée à 25 MHz. Ils ont conçu une carte permettant d'insérer tous les composants, incluant un FPGA XC6216, 256 Mo de mémoire locale SRAM et un décodeur générant les signaux de contrôle au FPGA et à la mémoire. Ce décodeur est muni de trois PALs (*Programmable Array Logic devices*) et il sert d'interface au FPGA. De cette manière, la carte coprocesseur (la carte Hades) agit, pour le Ceres-2, comme une mémoire supplémentaire et l'accès au FPGA est par conséquent aussi rapide que l'accès à une mémoire conventionnelle. La carte Hades occupe un mégaoctet dans l'espace mémoire du système d'exploitation de la machine hôte. Notons enfin que le système d'exploitation utilisé pour le projet Hades a été le *Oberon Operating System* dont on a modifié le noyau (*kernel*) pour supporter l'espace mémoire nécessaire.

Pour rendre l'utilisation des FPGAs aussi simple que n'importe quelle autre ressource d'un système, Brebner et al [Breb96] ont tenté de développer un système d'exploitation adapté à ce type de composant. Encore une fois, ce sont les FPGAs de la famille XC6000 de la compagnie Xilinx qui ont été sélectionnés pour ce projet de recherche. La possibilité de ne les configurer que partiellement et d'y avoir accès de la même manière qu'une mémoire conventionnelle constituaient pour les chercheurs des caractéristiques intéressantes.

La solution que Brebner et son groupe ont proposé pour gérer les FPGAs consiste à reprendre le principe de la mémoire virtuelle utilisé dans les systèmes d'exploitation conventionnels. Dans les deux cas, il s'agit de créer pour l'usager l'illusion qu'une ressource matérielle est beaucoup plus grande que dans la réalité, ce qui signifie que la ressource matérielle doit être à temps partagé. Les circuits utilitaires sont échangés entre

le FPGA et le disque selon le besoin. Des unités de logique SLU (*Swappable Logic Unit*) sont définies dans ce but. Les services de FPGA sont offerts par le système d'exploitation via une bibliothèque de fonctions programmées en langage C. On trouvera ci-dessous quelques-unes des principales fonctions de cette bibliothèque :

1) Une fonction pour déclarer l'existence d'un SLU.
2) Une fonction qui permet d'instancier un SLU sur le FPGA.
3) Une fonction qui permet aux usagers de charger les données d'entrée et de recevoir les données de sortie d'un SLU.

Des expériences avec une simple collection de SLUs ont été effectuées et les résultats préliminaires ont été encourageants. Le système développé par Brebner et son groupe fait en sorte que les FPGAs sont gérés par le système d'exploitation et qu'il permet de réaliser ses opérations sans que les usagers ne sachent où se situent exactement leurs circuits sur la puce.

CHAPITRE 3 - GESTIONNAIRE

Le présent projet de maîtrise porte sur l'utilisation de FPGA comme source de matériel dédié à la demande pour des applications s'exécutant sur des microprocesseurs. Nous avons choisi le système H.O.T. de la compagnie *Virtual Computer Company* (VCC) comme matériel et l'environnement JERC (*Java Environment for Re-configurable Computing*) comme l'outil logiciel. Un gestionnaire de RPU (*Re-configurable Processing Unit*) qui permet d'automatiser l'utilisation et le partage de la carte H.O.T. entre plusieurs applications indépendantes a été mis au point.

3.1 Le système H.O.T.

Le système H.O.T. fait partie de la famille des produits de type *configurable computing* de la compagnie *Virtual Computer Company* (VCC). Il se compose d'un RPU du modèle XC6200 et d'un FPGA du modèle XC4000 de la compagnie Xilinx. Il est conçu sous la forme d'une carte PCI que l'on peut insérer dans un PC. Il offre une plate-forme standard avec des composants re-configurables, ce qui donne la possibilité de créer et d'utiliser des circuits sur demande. Il est idéal pour faire du co-design matériel / logiciel.

3.1.1 Introduction du système H.O.T.

Les ordinateurs conventionnels sont des systèmes qui utilisent du matériel fixe et prédéterminé, basés sur un ou plusieurs microprocesseurs. Avec chaque nouvelle génération de microprocesseur, la performance des applications n'augmente pas aussi rapidement que celle des microprocesseurs. En effet, la capacité de la mémoire cache, le temps d'accès à la mémoire et la capacité limitée de paralléliser le code limitent l'augmentation de performance des applications.

Pour qu'une application ait une performance maximale, une ou plusieurs de ses procédures doivent être exécutées par des moyens matériels plutôt que logiciels. Cela est impossible avec les systèmes ordinés conventionnels, les microprocesseurs ordinaires ne possédant pas une architecture spécifique permettant de maximiser l'exécution d'une application particulière. Ils possèdent plutôt une architecture générale qui permet d'exécuter toutes les applications avec une performance moindre. Inversement, avec un système ordiné configurable, il est possible d'avoir du matériel spécifique pour chaque application et donc d'obtenir une meilleure performance.

Les systèmes ordinés configurables sont des plates-formes dont l'architecture peut être modifiée par un logiciel afin de l'adapter aux différentes applications. Dans le présent projet, ils sont composés de deux parties : la partie non configurable (généralement un ou plusieurs microprocesseurs conventionnels) et la partie configurable, qui peut être reconfigurée dynamiquement par un logiciel. La règle du 90/10 nous indique que dans une application typique, 90 % du code utilise 10 % du temps d'exécution, et 10 % de code consomme 90 % de temps d'exécution. Selon cette règle, une solution naturelle sera de faire exécuter ces 90% de code sur la partie non configurable d'un système ordiné, mais de confier les 10 % de code qui consomment 90 % de temps d'exécution à la partie re-configurable du système. Ainsi avec peu d'effort, on peut espérer obtenir une augmentation de performance considérable. Pour y parvenir, il faut que les deux parties du système puissent coopérer efficacement et que la configuration soit faite rapidement et de façon dynamique pendant l'exécution. En tant que composant re-configurable d'un système ordiné configurable, le système H.O.T. répond à tous ces critères.

La procédure qui consiste à faire alterner entre les composants matériels la logique configurable inscrite dans une application est appelée reconfiguration en temps d'exécution (*Run Time Reconfiguration*). Ce type de reconfiguration donne aux applications la possibilité d'utiliser les composants matériels des systèmes ordinés selon le principe du "matériel sur demande". Le système H.O.T. permet d'exploiter ce principe par l'entremise de la technologie des objets matériels (*Hardware Object Technology*),

Cette technologie permet aux concepteurs d'utiliser des designs matériels dans des programmes écrits en C++ ou en Java. Nous examinerons cette technologie un peu plus loin dans ce travail.

Le système H.O.T. est une extension de l'ordinateur hôte. Comme un microprocesseur, il peut effectuer des calculs complexes. Contrairement à un microprocesseur toutefois, il peut être modifié de façon à lui permettre de résoudre des problèmes spécifiques. Ce système se compose de deux parties : la partie matérielle, constituée de la carte H.O.T. et la partie logicielle, qui se présente sous la forme d'un protocole de communication permettant de faire communiquer la carte H.O.T. avec l'ordinateur hôte. Ensemble, ces deux parties forment une plate-forme prête à servir au co-design matériel / logiciel. On trouvera à la figure 3-1 la photo d'une carte H.O.T.. Nous aborderons l'architecture de cette carte dans la section suivante.

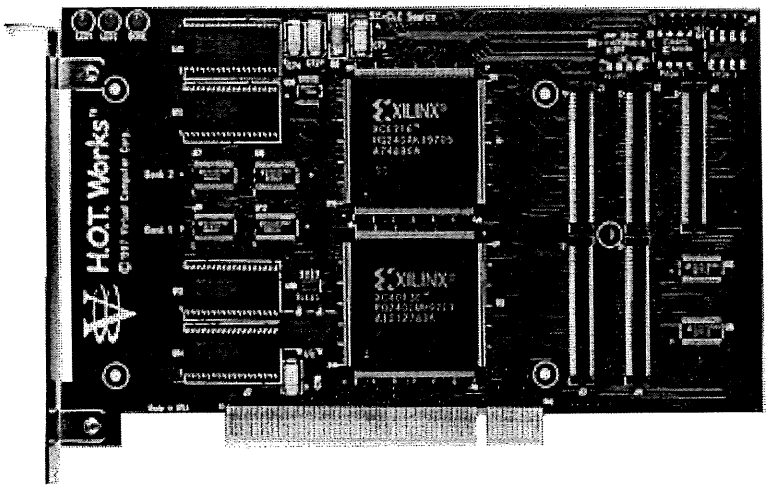

Figure 3-1 Une photo de la carte H.O.T. (tiré de [Xilinx])

3.1.2 La carte H.O.T.

La carte H.O.T. est un périphérique que l'on peut insérer dans une fente d'extension d'un PCI standard. L'interface avec le système hôte est réalisée par l'entremise d'un logiciel pilote exécuté par l'ordinateur et d'un circuit d'interface chargé dans le FPGA XC4000 de

la carte. La carte H.O.T. agit comme un co-processeur étroitement couplé au microprocesseur de l'ordinateur hôte. La figure 3-2 illustre la relation qui unit la carte H.O.T. au système hôte.

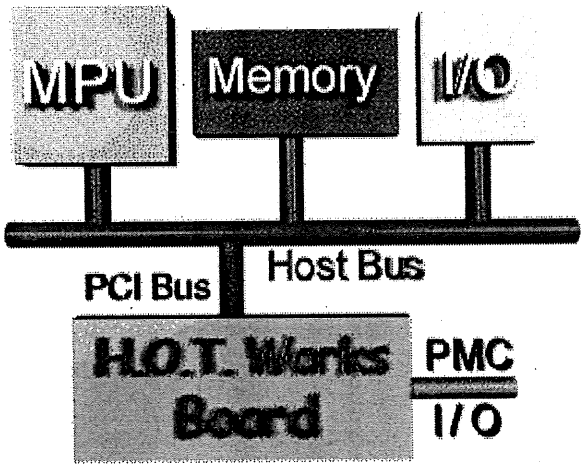

Figure 3-2 Le système H.O.T. et l'hôte (tiré de [Xilinx])

On trouvera à la figure 3-3 un diagramme illustrant l'architecture de la carte H.O.T. Le système H.O.T. consiste principalement en un FPGA XC4000 et un élément de calcul constitué d'un RPU XC6000, de quatre mémoires SRAM de huit bits chacune et de six contrôleurs de bus (ces contrôleurs ne sont pas illustrés à la figure 3-2). L'ordinateur hôte et la carte communiquent par le bus PCI à travers le FPGA XC4000. Ce FPGA contient l'interface de configuration de PCI/RPU. Cette interface est chargée automatiquement sur le FPGA à partir d'un PROM sur la carte au moment du démarrage du système H.O.T..

Les mémoires SRAM sont organisées en deux banques constituées chacune de deux SRAM de 512 Ko. Chacune de ces banques est accessible soit par l'interface PCI, soit par le RPU. Leur mode d'opération est configuré par les six contrôleurs de bus et elles possèdent chacune deux bus d'adresses séparés qui permettent de contrôler le signal d'écriture et de lecture de chacun des quatre SRAM. Cette architecture flexible offre aux usagers la possibilité d'implémenter une vaste variété d'algorithmes.

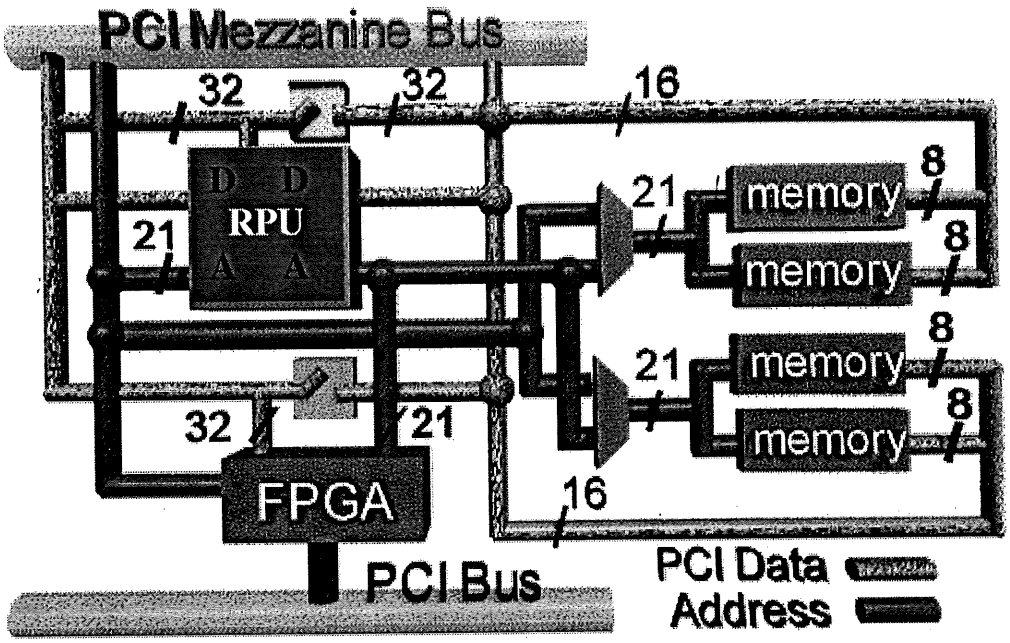

Figure 3-3 L'architecture de la carte H.O.T. (tiré de [Xilinx])

Les connecteurs « mezzanine » des cartes H.O.T. offrent la possibilité de connecter d'autres cartes filles au système. L'oscillateur programmable peut être utilisé par le FPGA et le RPU.

Le RPU est le composant principal de l'élément de calcul. L'architecture de la carte H.O.T. permet la configuration dynamique du RPU par l'interface PCI. Cette dernière rend également possible un accès direct aux cellules logiques du circuit configuré en permettant la lecture et l'écriture de données dans le registre flip-flop de chaque cellule. Nous examinerons ce mécanisme plus en détail dans la section suivante.

3.1.3 Le RPU XC6200

Les RPU (*Re-configurable Processing Unit*) de la famille XC6200 sont un type particulier de FPGA. Ils sont conçus pour coopérer étroitement avec un microprocesseur

ou un micro-contrôleur en offrant une exécution des fonctions qui sont normalement implémentées dans un ASIC. Un RPU est composé de milliers de cellules logiques configurables. Chacune de ces cellules contient un élément de calcul qui peut être configuré selon un ensemble de fonctions logiques et de ressources de routage qui permettent la connexion inter-cellules. Cette structure est simple, symétrique, hiérarchique et régulière. La configuration des RPU s'effectue par l'entremise d'une mémoire SRAM, ce qui permet de les configurer un nombre illimité de fois. Cette mémoire pouvant être représentée dans l'espace d'adressage du processeur hôte, cette configuration peut également être effectuée très rapidement. De plus, les RPU peuvent être partiellement reconfigurés sans entraver le fonctionnement en cours des autres sections, une caractéristique unique qui rend possible le partage du co-processeur entre plusieurs applications. Examinons maintenant les RPU d'un peu plus près.

Architecture

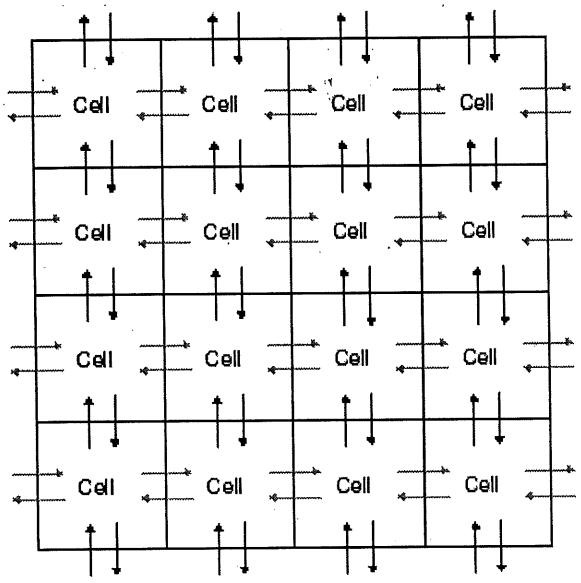

Figure 3-4 L'architecture *sea of gate* de la première génération avec seulement l'interconnexion entre les cellules voisines (tiré de [Xilinx])

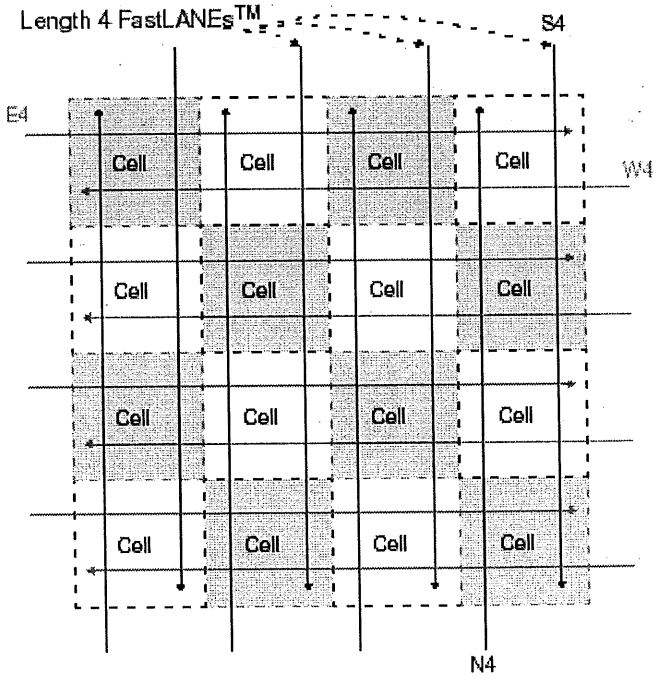

Figure 3-5 Un bloc de 4x4 cellules de l'architecture XC6200 (tiré de [Xilinx])

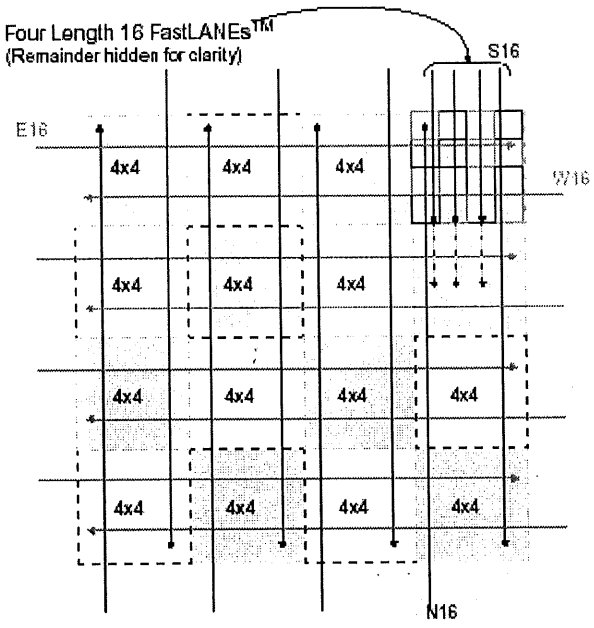

Figure 3-6 Un bloc de 16x16 cellules de l'architecture XC6200 (tiré de [Xilinx])

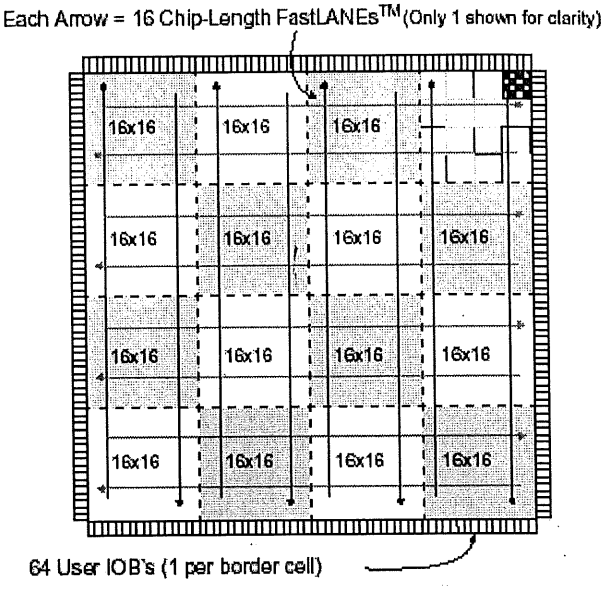

Figure 3-7 L'architecture du RPU XC6216 avec 64x64 cellules (tiré de [Xilinx])

L'architecture des RPU (souvent appelée *sea of gate*) peut être vue comme une hiérarchie. Au plus bas niveau de cette hiérarchie se trouve une grande matrice composée de cellules simples. La première génération de cette architecture assurait seulement l'interconnexion entre les cellules voisines sans aucun routage hiérarchique (figure 3-4). Les XC6200 font partie de la deuxième génération de RPU, une génération de puces qui utilisent une structure de cellules hiérarchiques par laquelle les cellules simples sont regroupées en blocs de 4x4 (figure 3-5). Ces blocs se comportent comme une unité pour la hiérarchie d'un niveau plus élevé. Ces unités de 4x4 cellules communiquent directement entre elles tout comme les cellules de niveau inférieur. Ces blocs sont à leur tour regroupés en blocs de 4x4 blocs (ou 16x16 cellules (figure 3-6)). Chacun de ces blocs de 16x16 cellules constitue une unité pour la hiérarchie du niveau supérieur. Une matrice de 4x4 blocs de 16x16 cellules forme un bloc de 64x64 cellules (figure 3-7) et ainsi de suite. La hiérarchie du plus haut niveau des RPU du modèle XC6216 qui seront utilisés dans le présent projet est une matrice de 4x4 blocs de 16x16 cellules faisant un total de 64x64 cellules.

Ressources de routage

L'une des caractéristiques les plus remarquables des RPU XC600 est l'abondance de leurs ressources de routage. Chaque niveau de leur hiérarchie possède des ressources de routage supplémentaires. Des fils faisant la longueur d'une cellule permettent d'interconnecter chacune des cellules simples. On notera à ce propos que les cellules utilisées pour réaliser cette connexion peuvent en même temps servir à l'implémentation de fonctions logiques. D'autres fils d'une longueur de quatre cellules permettent d'établir la connexion entre les blocs de 4x4 cellules. De même, des fils faisant la longueur de 16 cellules et de 64 cellules permettent de connecter les éléments des hiérarchies supérieures. Les fils dont la longueur dépasse la longueur d'une cellule simple sont appelés FastLANEs™. Ils sont tous directionnels et toujours nommés selon leur longueur et leur direction. Par exemple, un fil S4 est un fil de longueur de quatre cellules et la direction est du nord au sud (figures 3-5, 3-6 et 3-7). L'avantage de pouvoir compter sur de nombreuses ressources de routage réside dans le fait que les délais causés par le routage sont logarithmiques par rapport à la distance et non pas linéaires comme dans le cas des RPU de première génération dans lesquels on ne retrouvait que des ressources de routage n'assurant la connexion qu'entre cellules voisines. Les RPU de seconde génération possèdent quelques autres ressources de routage comme les « fils magiques » et les « fils globaux » (Consultez les spécifications du XC6000 [DATA97] pour en savoir plus).

Unité fonctionnelle

La figure 3-8 montre les détails d'une cellule simple d'un RPU de modèle XC6200. Ces cellules sont constituées d'une unité fonctionnelle (*Function Unit*) entourée de ressources de routage. Selon la direction d'où arrive le signal, les entrées qui proviennent des cellules voisines sont étiquetées N, S, E, W et celles qui proviennent des fils de longueur de quatre cellules (FastLANEs™) sont étiquetées N4, S4, E4, W4. Ce type de puce est également muni d'une entrée d'horloge (*Clk*) et d'un signal asynchrone (*Clr*) destinés au registre de l'unité fonctionnelle. La sortie de l'unité fonctionnelle est marquée "F". Une unité fonctionnelle peut établir environ 18 différents types de fonctions logiques incluant des registres du type D, des multiplexeurs de 2 à 1, des fonctions booléennes avec une ou

deux variables et les constantes 0 et 1. La figure 3-9 propose une représentation schématique de toutes les fonctions logiques gérées par une unité fonctionnelle.

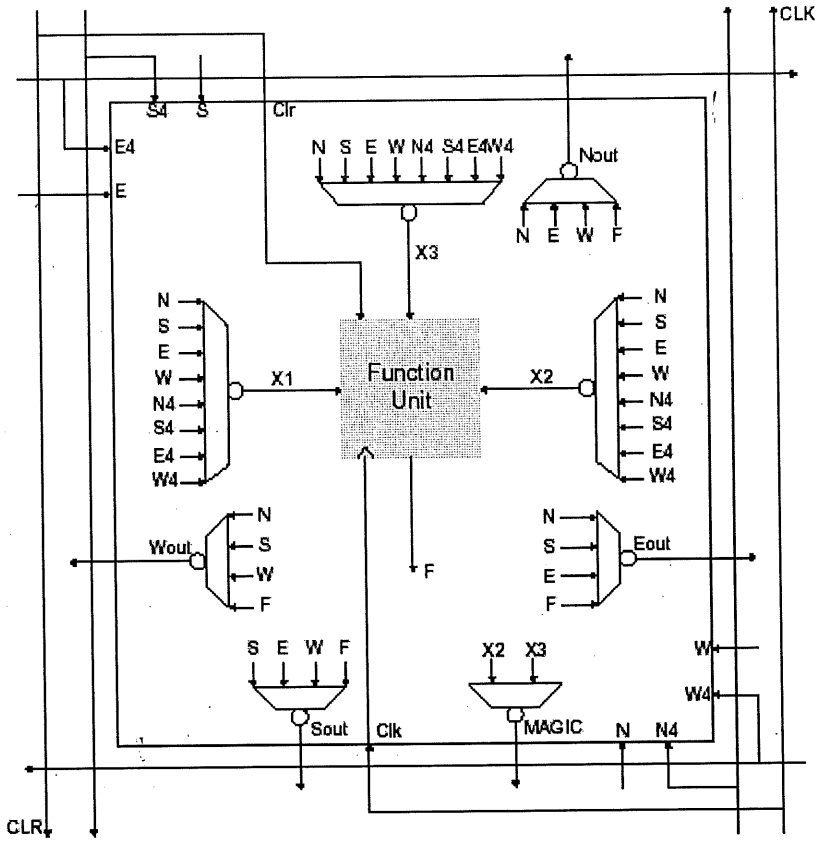

Figure 3-8 Une cellule simple des RPU XC6200 (tiré de [DATA97])

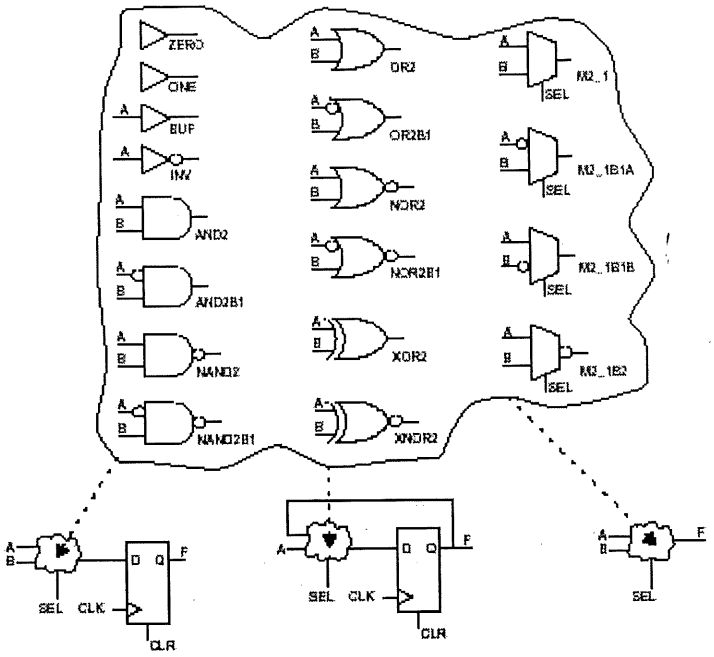

Figure 3-9 Les fonctions logiques qu'une unité fonctionnelle du RPU XC6200 peut implémenter (tiré de [DATA97])

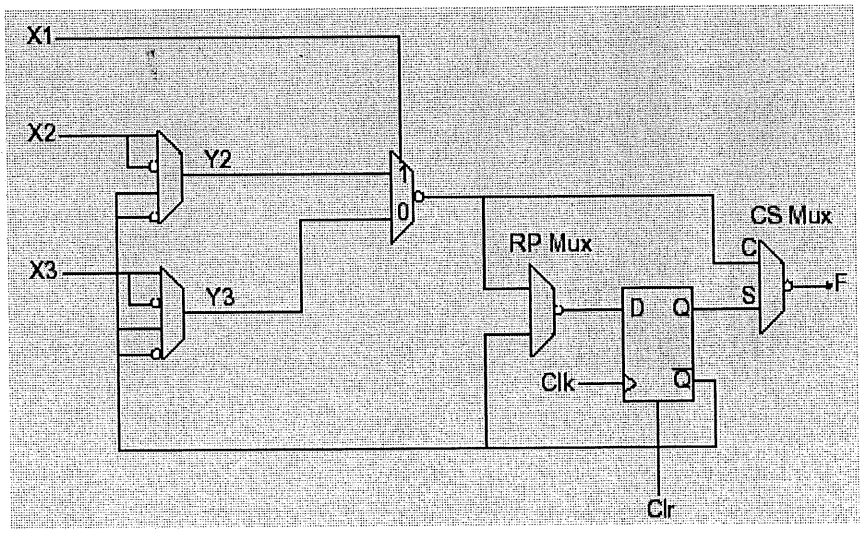

Figure 3-10 La structure de l'unité fonctionnelle dans une cellule des RPU XC6200 (tiré de [DATA97])

Pour comprendre comment l'unité fonctionnelle d'une cellule peut implanter toutes ces fonctions logiques, il faut étudier sa structure. Comme nous le voyons dans la figure 3-10, une cellule de RPU XC6200 comporte quelques multiplexeurs et un registre de type

D. Le principe de base qui préside au fonctionnement de ces cellules réside en ce que toutes les fonctions booléennes à deux variables peuvent être réalisées par un multiplexeur de 2 à 1 en choisissant bien les entrées et à la condition que les compléments des variables entrées de la fonction soient également présents parmi les entrées du multiplexeur. Par exemple, pour réaliser une fonction ET : A * B, il suffit de connecter le multiplexeur tel qu'indiqué dans la figure 3-11. Le multiplexeur au milieu de la figure 3-10 joue ce rôle. Les deux autres multiplexeurs qui sont à sa gauche se chargent de choisir les entrées. Le multiplexeur marqué CSMux décide quant à lui si la fonction est combinatoire ou séquentielle. En utilisant le registre D dans l'unité fonctionnelle, il est possible de définir des fonctions séquentielles.

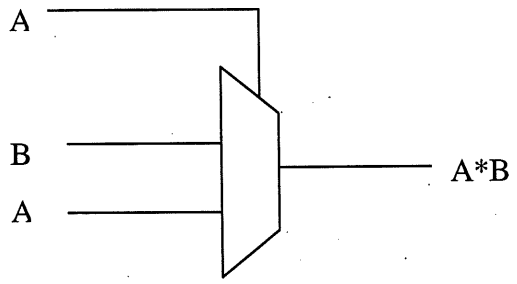

Figure 3-11 Réaliser la fonction ET avec un multiplexeur

Accès aux registres

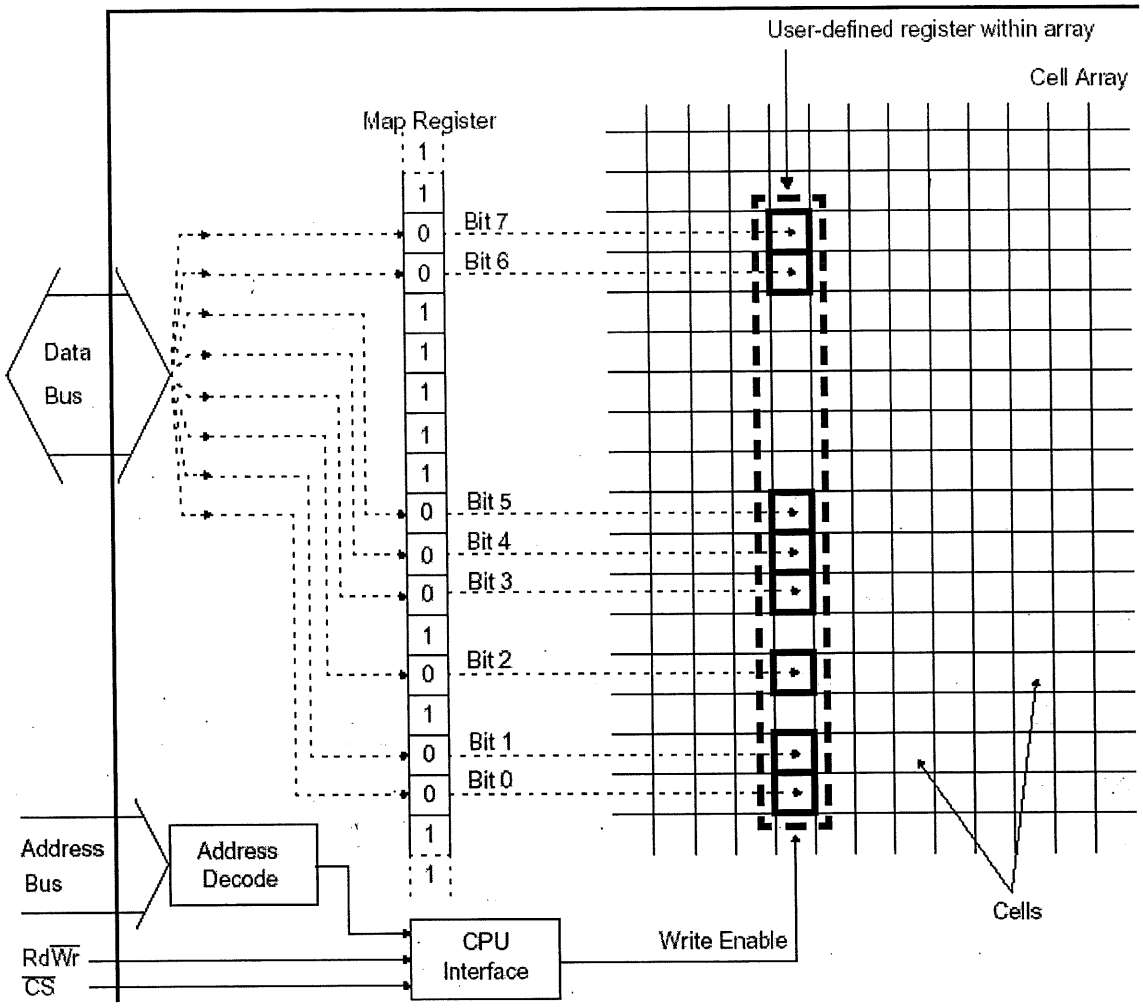

Figure 3-12 Accès aux registres internes du RPU (tiré de [DATA97])

Les RPU XC6200 supportent les accès directs à partir du processeur. Cette caractéristique fait en sorte que non seulement la sortie de l'unité fonctionnelle de n'importe quelle cellule peut être lue directement par le processeur mais si cette unité fonctionnelle est définie comme étant un registre d'un bit, on peut y écrire directement. Grâce à un circuit configurable des RPU XC6200, il est également possible d'acheminer directement aux cellules les signaux qui proviennent du CPU par l'intermédiaire de l'interface de la carte H.O.T.. Ce circuit est en mesure de détecter un tel accès et réagir en

conséquence, c'est-à-dire effectuer un calcul pour fournir une valeur à la sortie appropriée ou encore traiter la nouvelle valeur dans un registre d'entrée. Les RPU XC6200 disposent d'un mécanisme qui permet d'utiliser une bande passante de 8, 16 ou 32 bits pour le transfert des données. Il suffit de configurer une carte de registres (*Map Register*) avant le transfert. Il s'agit d'un registre de 64 bits (figure 3-12) où chacun des bits correspond à une rangée du RPU. Lorsque le bit est à 0, la rangée correspondante est activée. Les bits du bus de données provenant du processeur sont ainsi reliés aux différentes rangées du RPU. À l'aide de la mappe des registres, chacun des cycles de transfert de données peut s'effectuer sur des registres physiquement disjoints du RPU mais qui doivent être sur la même colonne. Le numéro de la colonne provient du signal sur le bus d'adresse. C'est un signal de 6 bits qui permet de désigner les colonnes (numérotées de 0 à 63). C'est précisément cette capacité d'accès direct au RPU qui nous assure la coopération étroite et l'échange d'information rapide entre le RPU et le processeur hôte.

Nous avons vu comment le système H.O.T. peut, à partir d'un ordinateur, servir à créer des éléments de matériel utilisables par des applications. Dans les sections qui suivent, nous verrons comment utiliser ce matériel à partir du logiciel.

3.2 JERC (Java Environment for Re-configurable Computing)

L'amélioration des performances qu'il apporte aux systèmes ordinés a engendré aux cours des dernières années un intérêt considérable pour le design matériel / logiciel. À ce chapitre, c'est une bonne cinquantaine de plates-formes qui ont été conçues pour explorer cette nouvelle approche. Tous ces systèmes permettent de combiner l'utilisation de matériel dédié (sous forme de puces ASIC ou de FPGA) et l'utilisation de logiciels fonctionnant sur des microprocesseurs ou des micro contrôleurs. Il se trouve toutefois que peu de logiciels ont été développés pour unifier et simplifier l'intégration du matériel et du logiciel. La plupart des systèmes ont recours à des techniques traditionnelles de conception de circuits dédiés que l'on tente par la suite d'interfacer au système hôte au moyen de langages de programmation standards. Cette approche par outils de synthèse conventionnels ne permet qu'une synthèse statique des circuits et n'offre pas par conséquent la possibilité d'effectuer la reconfiguration des coprocesseurs pendant l'exécution des programmes (*run time*).

Il existe depuis peu une nouvelle conception du co-design matériel / logiciel : le JERC (*Java Environment for Re-configurable Computing*). Il s'agit d'un environnement logiciel destiné aux applications utilisant des coprocesseurs re-configurables. Il se présente sous la forme d'un ensemble de bibliothèques de fonctions écrites en Java. Il était traditionnellement de mise de faire suivre l'étape de synthèse logique des circuits par une phase de placement et de routage (*place and route*) des portes logiques, obtenant ainsi un circuit matériel prêt à être utilisé. Or dans l'environnement JERC, à l'aide des bibliothèques JERC et le langage Java, le code de la configuration de FPGA et le logiciel interface au coprocesseur se trouve dans une même partie de code, c'est-à-dire la définition, le placement et le routage et la réalisation d'un circuit dédié sont effectués en une seule étape à l'aide de l'outil JERC. L'environnement JERC permet donc non seulement de réduire à environ une seconde le temps de compilation d'une application mais la configuration et la reconfiguration des coprocesseurs peuvent se faire aisément pendant l'exécution des applications.

3.2.1 La procédure du co-design matériel / logiciel

La conception d'une application sur les coprocesseurs re-configurables nécessite actuellement deux étapes distinctes. La première étape (qui est celle qui nécessite sans doute le plus d'effort), consiste à concevoir un circuit avec des outils CAD (*Computer Aided Design*) conventionnels. Typiquement, les phases de cette conception sont les suivantes : dessiner un circuit à l'aide d'un éditeur schématique ou d'un éditeur de langage HDL (*Hardware Description Language*); générer, à partir de ce dessin de circuit, une liste d'interconnexions (*netlist*) à l'aide d'un outil de synthèse; puis produire les données de configuration pour un FPGA ciblé à l'aide d'un outil de placement et de routage (*place and route*). Ce n'est qu'une fois les données de configuration produites que la deuxième étape peut débuter. Elle consiste à produire un logiciel qui servira d'interface au coprocesseur re-configurable. Cette méthode de design est illustrée dans la figure 3-13. Nous constatons que les deux étapes de conception qu'elle implique sont complètement isolées l'une de l'autre. En conséquence, le cycle de design est souvent très long. De plus, la plupart des outils traditionnels de design matériel ne permettent pas de créer des circuits dynamiquement re-configurables.

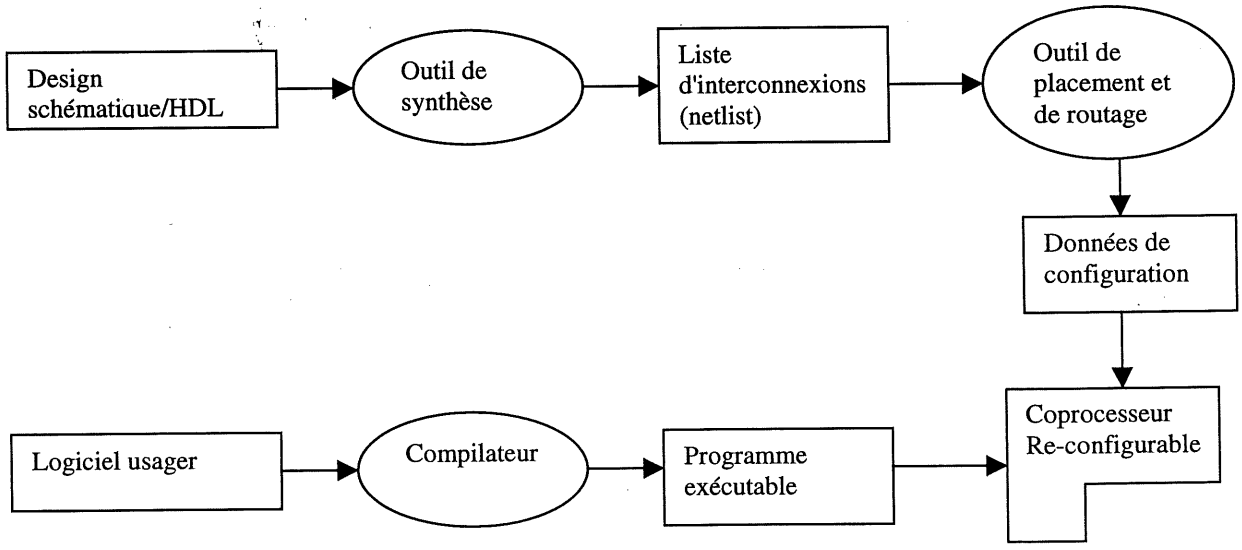

Figure 3-13 La procédure traditionnelle du co-design matériel / logiciel

Inversement, l'environnement JERC comporte une bibliothèque de fonctions qui permettent de spécifier l'aspect logique et le routage d'un design et de le réaliser sur un

composant re-configurable. Ainsi, dans un même programme hôte qui contient l'interface avec le coprocesseur, nous pouvons à la fois configurer le coprocesseur en appelant les fonctions de la bibliothèque JERC et interagir avec ce même coprocesseur. L'environnent JERC permet donc de combiner les deux étapes nécessaires au co-design matériel / logiciel. Cette approche simplifie non seulement la procédure de design (figure 3-14), mais elle rend possible la reconfiguration dynamique du coprocesseur.

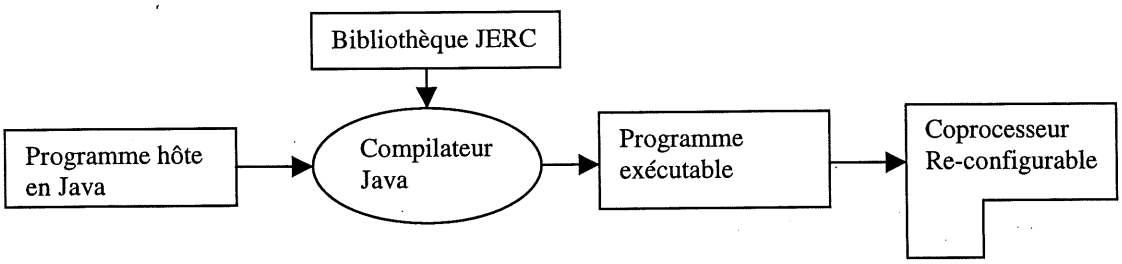

Figure 3-14 La procédure de design avec l'environnement JERC

3.2.2 L'abstraction du JERC

L'environnement JERC utilise une approche par couches pour créer une abstraction des composants re-configurables. Il définit principalement trois niveaux de code. Dans cette section, nous discuterons des différentes couches du JERC.

Au plus bas niveau, le niveau 0, le JERC utilise une représentation abstraite de toutes les ressources accessibles d'un composant re-configurable. Malgré son bas niveau, l'utilisation exhaustive des données constantes et des données symboliques le rend compréhensible et utilisable. La plate-forme de l'environnement JERC qui sera utilisée dans le présent projet est le système H.O.T. de la compagnie VCC sur lequel on retrouve le RPU XC6216. Dans ce cas précis, le niveau 0 de l'environnement JERC représente toutes les cellules et les commutateurs de routage du RPU XC6200, incluant le routage de l'horloge. Le code du niveau 0 du JERC est étroitement lié aux informations fournies dans les spécifications du XC6200. Le niveau 0 du JERC permet de programmer tous les aspects du composant XC6200 mais comme il se situe à un très bas niveau, il est

relativement fastidieux à utiliser directement et une telle programmation exige une bonne connaissance de l'architecture du composant ciblé. Habituellement, ce niveau de code n'est utilisé que comme base pour les niveaux supérieurs. En ce sens, le niveau 0 est en quelque sorte le langage assembleur de l'environnement JERC.

La couche de niveau 1 est bâtie au-dessus de la couche de niveau 0. La principale fonction de cette couche consiste à procéder à une abstraction des fonctions logiques et du routage du signal horloge et *clear*. Une portion importante du code est consacrée à la définition de la fonction des cellules du RPU, ce qui permet de configurer ces cellules en portes logiques standards. Actuellement les fonctions combinatoires prédéfinies sont : AND, NAND, OR, NOR, XOR, XNOR, BUFFER, et INVERTER (optionnellement, ces fonctions peuvent utiliser une sortie séquentielle). Les portes D flip-flop, toggle flip-flop et le registre sont aussi prédéfinis. Toutes ces fonctions logiques sont définies exclusivement à l'aide du niveau 0. Le niveau 1 pour sa part supporte les entrées et les sorties, ce qui permet de regrouper les cellules situées sur la même colonne et de les configurer comme port d'entrée / sortie. Une classe gabarit nommée HOG (*Hardware Object Generator*) est également prédéfinie. Il s'agit d'une classe abstraite qui permet de définir l'interface de tous les objets matériels qui seront implémentés sur le RPU. Elle contient des méthodes créateur / lecteur destinées à traiter les informations relatives aux objets matériels : leur hauteur et leur largeur (en terme de nombre de cellules) et leur position sur la puce XC6200. Elle comporte aussi une méthode abstraite "write()" qui réalise la configuration sur le RPU de l'objet. Cette classe abstraite fait en sorte que tous les objets matériels sont définis comme une instance paramétrée HOG.

Les objets matériels définis comme une sous-classe de la classe HOG constituent la troisième couche de l'environnement JERC. Cette couche est à un niveau d'abstraction plus élevé que la couche du niveau 1 car tous les objets sont construits avec les composants définis aux niveaux 1 et 0. Cette couche peut aussi être considérée comme une bibliothèque de circuits de base susceptibles d'être utilisés pour construire des circuits plus complexes. Dans cette bibliothèque, on retrouve des circuits simples comme un compteur, un demi-additionneur (*half adder*), un additionneur (*full adder*), etc.

3.2.3 Exemple d'un additionneur de *n* bits

On trouvera ci-dessous un exemple qui montre comment il est possible de construire un additionneur de n bits à l'aide de l'environnement JERC. La première étape consiste à trouver dans la bibliothèque de fonctions un demi-additionneur d'un bit.

```
public class HalfAdder extends HOG {

private static final int  WIDTH  = 3;
private static final int  HEIGHT = 2;

private  Logic     and;      /* The Logic elements */
private  Logic     xor;

public HalfAdder() {

   setWidth(WIDTH);      /* Set height and width */
   setHeight(HEIGHT);

   /* Half adder  */
   and = new Logic(Logic.AND, Logic.EAST, Logic.NORTH);
   xor = new Logic(Logic.XOR, Logic.EAST, Logic.SOUTH);

}

public int write(int  column, int  row, Pci6200 pci6200) {

   setRowStart(row);       /* Save some local variables */
   setColumnStart(column);

   /* Half adder 1 */
   xor.write(column, row, pci6200);
   and.write(column, (row+1), pci6200);

}  /* end write() */
```

Figure 3-15 Partie du code d'un demi-additionneur d'un bit (tiré de [JERC97])

La figure 3-15 présente une partie du code du demi-additionneur de la bibliothèque JERC. Nous constatons que cette classe est une sous-classe du HOG. Comme nous le savons déjà, tous les objets matériels étant représentés par la classe abstraite HOG, le demi-additionneur constitue une instance du HOG. Dans notre exemple, sa hauteur et sa largeur sont de 2x3 cellules. Comme il est construit avec des portes AND et XOR, il contient une instance de chacune et est défini comme membre donnée de cette classe. Les portes AND et XOR sont prédéfinies au niveau 1 du JERC, ce qui démontre bien la

structure hiérarchique de l'environnement JERC. La fonction membre *write*(), qui est définie comme une fonction abstraite dans la classe HOG et qui sert à la réalisation du circuit sur le RPU XC6200, est implémentée concrètement dans la classe *half adder*. Cette fonction fait appel à la même fonction de la classe *logic* qui se trouve au niveau 1 du JERC. À son tour, la fonction *write*() de la classe l*ogic* fait appel à la même fonction de la classe *cell* après avoir défini la fonction logique de la cellule. La classe *cell* se trouve au niveau 0 du JERC. Elle constitue une représentation abstraite d'une cellule du RPU XC6200. La fonction membre *write*() de cette classe interagit avec le RPU XC6200 grâce à des méthodes qui lui sont propres et qui réalisent réellement la configuration du circuit. L'additionneur (*full adder*) est défini d'une manière semblable à celle du demi-additionneur. La figure 3-16 reproduit une partie de son code. Nous pouvons constater qu'il contient deux portes XOR, deux portes AND, une porte OR et un BUFFER. Toutes ces portes logiques sont définies comme membres données de la classe et la méthode membre *write()* réalise la configuration du circuit sur le RPU.

```
public class FullAdder extends HOG {

private static final int  WIDTH  = 3;
private static final int  HEIGHT = 2;

/* The Logic elements */
private  Logic     and1;
private  Logic     and2;
private  Logic     xor1;
private  Logic     xor2;
private  Logic     or;

public FullAdder() {

   /* Half adder 2 */
   and1   = new Logic(Logic.AND, Logic.EAST, Logic.NORTH);
   xor1   = new Logic(Logic.XOR, Logic.EAST, Logic.SOUTH);

   /* Half adder 2 */
   and2   = new Logic(Logic.AND, Logic.EAST, Logic.WEST);
   xor2   = new Logic(Logic.XOR, Logic.NORTH, Logic.EAST);

   /* OR carrys from half adders (and use buffer to send north) */
   or     = new Logic(Logic.OR, Logic.NORTH, Logic.EAST);
}

public int write(int  column, int  row, Pci6200 pci6200) {

   /* Half adder 1 */
   xor1.write(column, row, pci6200);
   and1.write(column, (row+1), pci6200);

   /* Half adder 2 */
   xor2.write((column+2), row, pci6200);
   and2.write((column+1), row, pci6200);

   /* Carry */
   or.write((column+1), (row+1), pci6200);
   buffer.write((column+2), (row+1), pci6200);
}
```

Figure 3-16 Partie du code d'un additionneur d'un bit (tiré de [JERC97])

Maintenant que nous disposons d'un demi-additionneur et d'un additionneur d'un bit, nous sommes en mesure de construire un additionneur de *n* bits. À l'aide de la hiérarchie bien structurée du JERC, c'est une opération mineure. La figure 3-17 montre combien il est facile de construire un additionneur de *n* bits à l'aide d'un demi-additionneur d'un bit et de (*n-1*) fois d'un additionneur d'un bit.

```
public class Adder extends HOG {

private int  bits;

private  HalfAdder  halfAdder;
private  FullAdder  fullAdder;

public Adder(int _bits) {
   int  height;
   int  width;

   /* Half adder */
   halfAdder = new HalfAdder();

   /* Half adder */
   fullAdder = new FullAdder();

   /* Save a local variable */
   bits = _bits;
}

public int write(int  _column, int  _row, Pci6200 pci6200) {
   int  i;
   int  row;

   /* Write out half adder */
   halfAdder.write(_column, _row, pci6200);

   /* Write out full adders */
   for (i=0; i<(bits-1); i++) {
     row = getRowStart() + halfAdder.getHeight() + (i *
fullAdder.getHeight());
      fullAdder.write(_column, row, pci6200);
      }
   }
}
```

Figure 3-17 Partie du code d'un additionneur de *n* bits (tiré de [JERC97])

Cet exemple de codification d'un l'additionneur de *n* bits nous permet de constater la facilité avec laquelle l'outil JERC permet de définir des circuits différents à partir des circuits de base offerts par sa bibliothèque. À cet égard, la configuration dynamique de circuits situés sur un RPU peut être réalisée en définissant dans le code source d'une application l'ensemble des circuits chargés d'exécuter cette application. Les circuits pourront ainsi être instanciés tout simplement en appelant la méthode membre de chaque instance au moment approprié. La programmation dynamique du RPU est ainsi réalisée.

En conclusion, l'environnement JERC représente une nouvelle approche dans le domaine du co-design matériel / logiciel, une méthode qui permet d'effectuer une reconfiguration dynamique des coprocesseurs. En nous servant de cet environnement, nous avons conçu un gestionnaire du coprocesseur XC6200 du système H.O.T. destiné à gérer l'utilisation et le partage de ce composant re-programmable par une ou plusieurs applications. C'est de ce gestionnaire dont il sera question dans la section suivante.

3.3 Le gestionnaire

Nous présenterons dans ce chapitre un logiciel conçu pour interfacer les applications contenant des objets matériels du RPU XC6200 d'un système H.O.T.. L'une de ces principales caractéristiques consiste à offrir aux applications une API assurant la réalisation des objets matériels sur le RPU et les échanges de données entre ces objets et les applications hôtes. Ce logiciel permet de gérer les objets matériels de différentes applications en partageant le RPU entre ces applications de façon efficace et sécuritaire. Pour ce faire, le gestionnaire divise la surface du RPU en quatre sections, chacune d'entre elles pouvant être utilisée de façon indépendante grâce aux caractéristiques particulières du RPU. Cette manière de gérer le RPU s'apparente à la gestion de mémoire paginée dans un système d'exploitation, les sections du RPU correspondant aux cadres d'une mémoire physique. Nous utilisons l'algorithme LRU (*Least Recently Used*) pour la gestion de ces sections tout comme c'est le cas pour la gestion de cadres paginés dans un système d'exploitation. Chaque section possède un attribut « âge » qui fait en sorte que lorsque toutes les sections sont occupées, c'est la section la moins récemment utilisée qui est choisie comme « victime » et l'objet qui occupait cette section est remplacé par un nouveau si la section n'est pas verrouillée par une application. L'âge de chaque section est mis à jour chaque fois qu'un nouvel objet est réalisé ou lorsque des entrées sont écrites pour un objet occupant une section. Les sections sont verrouillées entre le moment où l'objet qui l'occupe reçoit des entrées et le moment où les sorties sont lues par l'application hôte. En supposant qu'une application ayant écrit des entrées à un objet matériel va lire les sorties sous peu, le mécanisme de verrouillage des sections empêche l'objet occupant cette section d'être remplacé avant que l'application hôte ne reçoive le résultat du calcul, ce qui assure une bonne performance. Dans les sections qui suivent, nous présenterons le logiciel gestionnaire en détail.

3.3.1 Vue globale du gestionnaire

On trouvera dans cette section une vue globale de notre logiciel gestionnaire du RPU XC6200. Nous aurons recours pour ce faire à un diagramme de « cas d'utilisation » (*use cases*) élaboré en langage UML (figure 3-18).

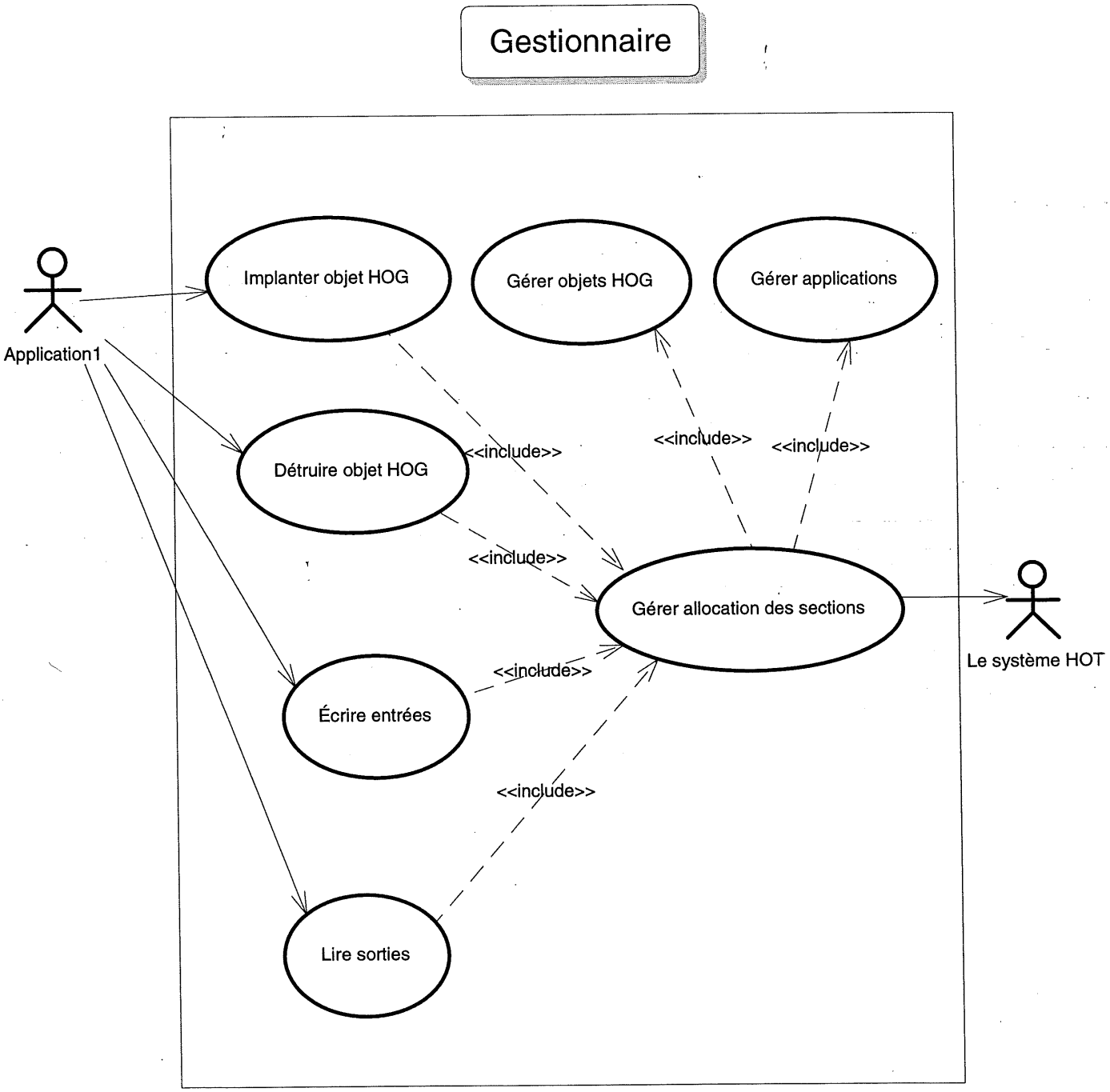

Figure 3-18 Vue globale du gestionnaire représentée par un diagramme de cas d'utilisation

Un diagramme de cas d'utilisation conçu en langage UML est constitué d'acteurs et de cas d'utilisation. Les acteurs sont représentés sous forme de petits personnages. Chaque acteur peut représenter l'une des deux catégories suivantes :

1- un rôle qu'un usager pourrait jouer en regard du système;
2- une entité qui est tout simplement en dehors du système.

En examinant la figure 3-18, nous constatons que pour notre gestionnaire, les applications sont des acteurs de la première catégorie. Ils représentent les usagers du gestionnaire. Quant au système H.O.T., il doit également être considéré comme un acteur, mais un acteur de la deuxième catégorie puisqu'il agit en tant qu'entité située en dehors de notre gestionnaire. Il réagit aux cas d'utilisation de notre gestionnaire.

Un cas d'utilisation est une séquence d'actions qu'un acteur effectue sur le système afin d'atteindre un but spécifique. Les cas d'utilisation sont représentés par des cercles ovales. Dans le cas de notre gestionnaire, quatre cas d'utilisation sont susceptibles d'être déclenchés par une application :

1- implanter un objet HOG;
2- détruire un objet HOG;
3- écrire des entrées;
4- lire les sorties.

Ces quatre cas d'utilisation peuvent être décrits comme étant des cas d'utilisation primaires, une application pouvant initier directement la création et la destruction d'un objet HOG de même que les échanges de données avec les objets HOG réalisés sur le RPU. Ces cas d'utilisation incluent le cas d'utilisation « gérer l'allocation des sections ». Le même cas d'utilisation (« gérer l'allocation des sections ») inclut les cas d'utilisation « gérer objets HOG » et « gérer applications ». Ces trois derniers cas d'utilisation doivent donc être considérés comme des cas d'utilisation secondaires puisqu'ils ne peuvent être déclenchés directement par un usager du système. Tous les trois servent à la

gestion de l'allocation du RPU. Nous verrons plus en détail leur fonctionnement dans la section 3.3.4. Il existe d'autres cas d'utilisation lesquels, pour des raisons de clarté, n'ont pas été intégré à notre diagramme.

Comme nous l'avons déjà mentionné, notre gestionnaire se présente comme une interface reliant les applications requérant des objets matériels et une carte H.O.T. équipée d'un RPU XC6200 pouvant réaliser des objets matériels. La figure 3-18 nous donne une vue d'ensemble du gestionnaire et de sa relation avec les applications et le système H.O.T.. Ce gestionnaire est codé en tant que classe Java. Une application écrite en langage Java peut facilement créer un objet de la classe gestionnaire et en faire un membre de donnée privé. À partir de ce membre de donnée, cette application peut avoir accès aux fonctions du gestionnaire et permettre la création d'un objet matériel et supporter les entrées / sorties effectuées sur l'objet. De plus, l'instance de gestionnaire étant utilisée dans le corps de l'application, les opérations relatives aux objets matériels sont effectuées dynamiquement pendant l'exécution, selon les besoins réels de l'application. Dans l'hypothèse où plusieurs applications veulent utiliser le RPU simultanément, il suffit de passer la même instance de gestionnaire dans toutes les applications. De cette façon, il n'y a qu'un seul gestionnaire qui gère le RPU, ce qui permet un partage centralisé de la ressource matérielle. En définissant plusieurs des fonctions du gestionnaire comme étant « synchronisées » (*synchronized*), le langage Java permet de gérer la synchronisation de l'accès au RPU, les fonctions faisant appel à la synchronisation étant définies comme étant synchronisées. On s'assure de cette façon un partage sécuritaire de la ressource matérielle. Nous étudierons le gestionnaire plus en détails dans les sections qui suivent.

3.3.2 La collaboration des classes

Nous présenterons dans la présente section le modèle de collaboration des classes que comporte notre logiciel gestionnaire. Pour ce faire, nous utiliserons un diagramme de classes élaboré en langage UML (figure 3-19). Ce type de diagramme nous donne une vue claire de la relation de coopération qui existe entre les classes et il nous aide par conséquent à visualiser la structure de notre logiciel gestionnaire. Dans cette figure, un

rectangle représente une classe et une ligne droite reliant deux de ces rectangles, une association entre classes. Chacune de ces associations possède une multiplicité. Cette dernière est marquée par le nombre ou les étoiles (*) situés sur la ligne reliant deux rectangles. La multiplicité sert à indiquer combien d'objets peuvent participer à une relation d'association donnée. Une seule étoile (*) signifie un nombre indéfini d'objets. Par exemple, un objet du gestionnaire peut être associé à un nombre indéfini d'objets de l'application. Les lignes se terminant par un triangle représentent quant à elles une relation de généralisation entre les classes reliées, le rectangle pointé par le triangle étant la classe mère et le côté opposé, la classe enfant. À titre d'exemple, la classe HOG, dans notre diagramme, constitue la classe mère de la classe *adder*.

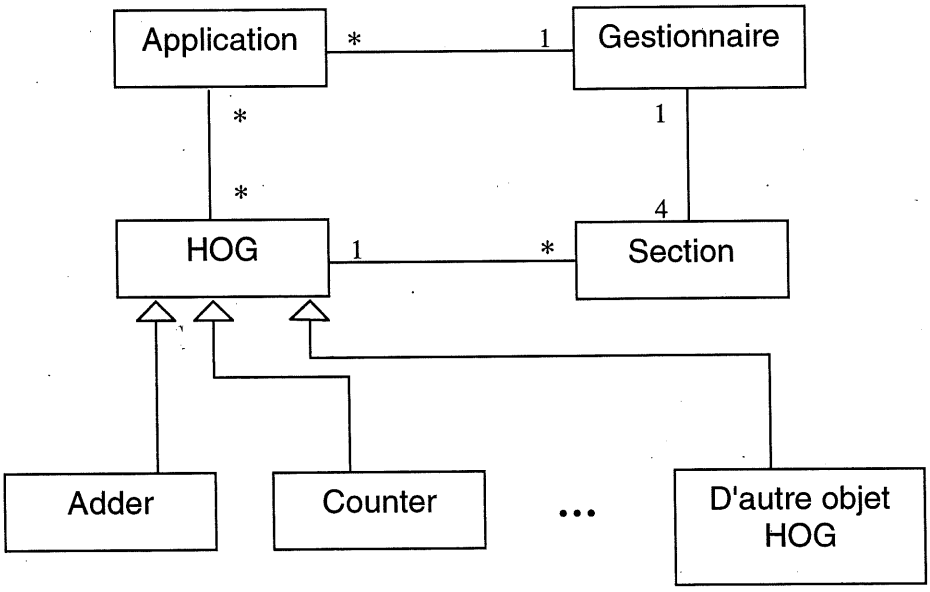

Figure 3-19 La collaboration des classes du gestionnaire présentée par un diagramme de classe

Dans le cadre de notre projet, un seul objet du gestionnaire sera présent dans le système. Cet objet est associé aux quatre objets de la classe « section », ce qui veut dire que l'objet « gestionnaire » est associé aux quatre sections du RPU. Ce même objet est aussi associé à 0 ou à plusieurs applications contenant des objets matériels. La première fois qu'une application tente d'implémenter un objet matériel sur le RPU, elle établit une relation associative avec l'objet « gestionnaire ». Chacune des applications est associée à un

certain nombre d'objets HOG car les objets matériels sont représentés par des objets HOG. La classe « HOG » étant une classe abstraite qui généralise tous les objets matériels, il existe des sous-classes de la classe HOG (i.e. *adder*, *counter*, etc.) qui représentent un objet matériel concret. Quand un objet matériel est réalisé sur le RPU, il est associé à la ou les sections qui lui sont allouées par le gestionnaire. Un objet peut occuper plusieurs sections si sa taille dépasse celle d'une section.

3.3.3 Les structures de données

Les structures de données utilisées dans notre logiciel gestionnaire du RPU XC6200 se présentent sous la forme de classes Java. Les plus importantes structures de données utilisées dans notre projet sont les classes Section et HOG.

Classe Section

La classe Section représente une section du RPU XC6200. Pour une utilisation efficace du RPU, nous avons décidé de diviser sa surface en quatre sections et de les utiliser indépendamment. Le RPU XC6200 supporte ce type de division puisqu'une configuration partielle de sa surface n'affecte pas l'ensemble du composant. Grâce à cette caractéristique, plusieurs circuits peuvent y résider simultanément et fonctionner indépendamment si la surface du RPU est suffisamment grande pour les contenir. C'est pour exploiter cette particularité du RPU XC6200 que nous avons décidé de diviser sa surface. On notera toutefois que le nombre de sections peut varier et que notre choix de diviser notre RPU en quatre sections relève avant tout de la surface du RPU dont nous disposions. Les RPU de plus grande taille peuvent évidemment être divisés en un plus grand nombre de sections.

La division des RPU en sections doit son origine au mécanisme de gestion de la mémoire paginée. Rappelons que dans un système d'exploitation doté d'une mémoire paginée, cette mémoire est divisée en un certain nombre de pages de même taille. L'unité d'allocation est une page. L'élaboration de ce concept répondait au besoin de résoudre le problème de fragmentation externe qu'impliquait l'allocation contiguë de blocs de

mémoire de tailles variables. Dans notre cas, bien qu'il soit possible de ne pas diviser le RPU et d'allouer des blocs de RPU d'une taille proportionnelle aux applications, cette façon de faire empêcherait d'utiliser le RPU de manière optimale, le problème de fragmentation externe qui affecte la mémoire paginée pouvant survenir. La figure 3-20 illustre la division de la surface du RPU pour laquelle nous avons optée. Chaque section est numérotée de 0 à 3 et elles sont représentées par la classe Section.

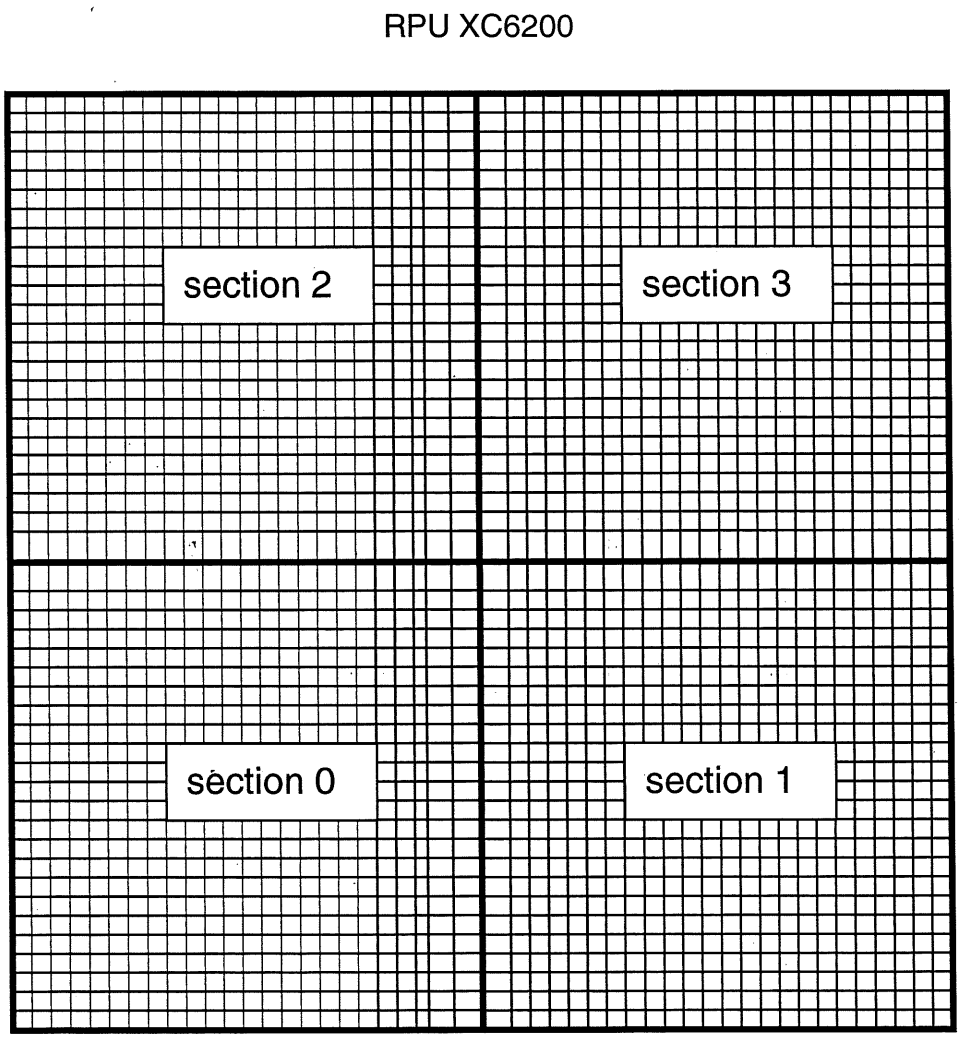

Figure 3-20 Les quatre sections du RPU XC6200

La figure 3-21 présente un tableau des éléments constituant la classe Section de notre gestionnaire. Il nous permet de constater qu'on y retrouve quatre membres données et cinq fonctions membre.

Membres données

- *occupied* indique si la section est occupée par un objet matériel;
- *idNo* est le numéro de la section qui varie de 0 à 3 comme illustré dans la figure 3-20;
- *hogObj* pointe vers l'objet matériel qui occupe la section;
- *age* est l'âge de la section. Il indique depuis combien de temps que la section n'est pas été utilisée. Ce champ est utilisé pour implémenter l'algorithme LRU.

Section
Occupied IdNo HogObj Age
getAge() setAge(age) makeOlder() setOccupied(occupied) getOcuppied()

Figure 3-21 Diagramme de la classe Section

Fonctions membres

- *getAge()* nous retourne l'âge de la section;
- *setAge(age)* peut donner un âge à la section, mais elle est surtout utilisée pour initialiser l'âge de la section à la valeur "0" au départ;
- *makeOlder()* est utilisé pour incrémenter l'âge de la section. Elle est appelée chaque fois qu'une autre section est utilisée;

- *setOccupied(occupied)* met à jour l'état de la donnée membre *occupied*;
- *getOcuppied()* nous retourne l'état d'occupation d'une section.

Classe HOG

La classe HOG (*Hardware Object Generator*) est une classe abstraite qui généralise tous les objets matériels. Il existe déjà un certain nombre de sous-classes de HOG dans la bibliothèque de l'environnement JERC. Les applications pourront définir leurs objets HOG et les y ajouter. La classe HOG contient les informations sur les opérations communes de tous les objets matériels. On retrouvera par exemple de l'information relative à leur dimension physique, à l'écriture des entrées et la lecture des sorties. Les applications définissent leurs objets matériels comme étant des objets de la classe HOG. La réalisation sur le RPU de l'objet HOG se fait en appelant sa fonction membre *write()*. La figure 3-22 est un diagramme de cette classe qui nous démontre en détails les membres données et les fonctions membres de la classe HOG :

Membres données

- *initialized* indique si l'objet est réalisé sur le PRU;
- *width* et *height* enregistrent la largeur et la hauteur de l'objet;
- *clolumnStart* et *rowStart* indiquent la position d'origine de l'objet sur le RPU;
- *sectionNo* est le numéro de la section où réside l'objet dans le RPU;
- *locked* indique si l'objet est verrouillé. Si sa valeur est "vrai", alors l'objet ne peut pas être enlevé du RPU;
- *InputEntered* note si l'application hôte a déjà écrit des entrées sur l'objet;
- *threadID* est le nom du *thread* (application) qui possède cet objet;
- *output* est utilisé pour enregistrer les valeurs de sorties de l'objet.

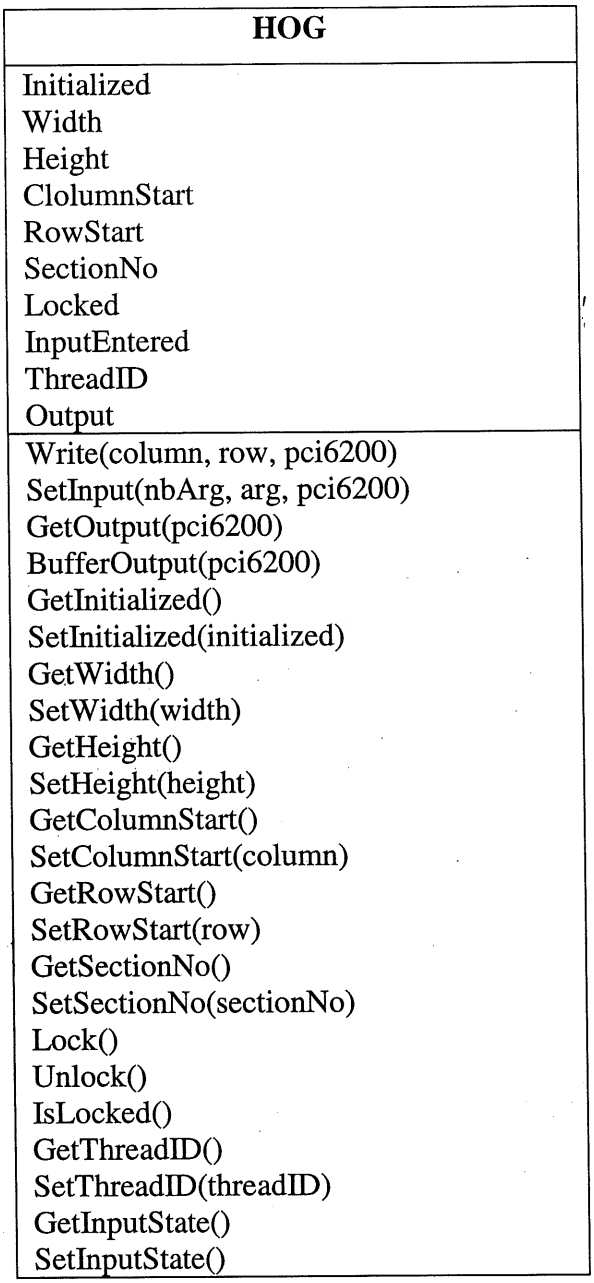

Figure 3-22 Diagramme de la classe HOG

Fonctions membres

La plupart des 23 fonctions membres de la classe HOG servent à manipuler les membres données. Leur fonctionnalité étant assez clairement décrite par leur nom, nous n'allons pas toutes les expliquer. Compte tenu de son importance, nous nous devons tout de même de dire un mot sur la fonction abstraite *write*(). Les sous-classes concrètes de la

classe HOG y font souvent appel. C'est la fonction *write*() qui, entre autres choses, est responsable de réaliser les objets matériels sur le RPU. De façon récursive, elle peut faire appel à cette même fonction lorsqu'elle appartient à d'autres composants du circuit. Par exemple, un additionneur d'un bit fera appel à la fonction *write*() de la porte ET et de la porte XOR dont il est dotées. De même, la fonction *write*() de la porte ET fera appel à la fonction *write*() de ses composants, et ainsi de suite. La position sur le RPU de chaque composant est transmise sous forme de paramètres à sa fonction membre *write*(). Ce mécanisme utilisé pour réaliser un circuit sur le RPU remplace ce que les outils de synthèse et les outils de placement et de routage (*place and route*) font dans la procédure traditionnelle du co-design matériel / logiciel.

Les fonctions abstraites *setInput*() et *getOutput*() sont également implémentées par les sous-classes concrètes de la classe HOG, les objets matériels possédant différentes entrées / sorties et diverses façons d'écrire les entrées et de lire les sorties. Pour les mêmes raisons, la fonction *bufferOutput*() est aussi une fonction abstraite. Lorsque les quatre sections sont occupées par un objet HOG verrouillé et qu'une nouvelle requête de section arrive, notre gestionnaire choisira la section la moins récemment utilisée et mettra la sortie de l'objet occupant dans un tampon au moyen de la fonction *bufferOutput*(). Nous évitons ainsi de bloquer les nouvelles requêtes de section dans le cas où toutes les sections seraient occupées par un objet HOG verrouillé.

3.3.4 La classe gestionnaire

Maintenant que nous connaissons les principales structures de données de notre système et sa structure globale, nous sommes prêts à étudier la classe gestionnaire. C'est dans cette classe que se trouve l'API du gestionnaire servant aux applications. Elle comporte également des fonctions utilitaires destinées à gérer l'ensemble des applications et l'ensemble des objets HOG. C'est dans cette classe également que l'algorithme LRU (*Least Recently Used*) qui sert à l'allocation des sections est implémenté. On aura compris que cette classe constitue le noyau de notre système de gestion du RPU.

L'API

L'API offert aux applications utilisant notre gestionnaire du RPU XC6200 est relativement simple. Il offre essentiellement des fonctions servant à réaliser et détruire un objet circuit sur la carte RPU de même que des fonctions reliées à la communication des objets circuits sur le RPU, notamment des fonctions destinées à écrire des entrées et lire des sorties. La gestion du RPU et des applications n'est pas visible pour les applications. L'API de notre logiciel gestionnaire comporte cinq fonctions publiques :

- *Load(hog)* prend un objet HOG comme argument, elle réalise l'objet circuit sur le RPU.
- *Unload(sectionNo)* fait le contraire de la fonction précédente. Elle prend le numéro d'une section comme paramètre puis elle enlève l'objet qui se trouve dans cette section.
- *SetInput(hog, nbArg, arg[])* écrit des entrées pour l'objet passé en paramètre (hog). Cette fonction vérifie d'abord si l'objet existe déjà dans la liste des objets HOG. S'il n'existe pas, elle génère un message d'erreur et le système s'arrête. Dans le cas contraire, elle vérifie si l'objet a été réalisé sur une des sections. Dans l'affirmative, les entrées relatives à l'objet sont écrites. Si l'objet n'a pas été réalisé, ledit objet est réalisé sur le RPU en appelant la fonction *load*() puis les entrées passées sont écrites sous la forme de paramètres (*arg*[]). À cet égard, le langage Java permet de faire passer en paramètres un tableau d'une taille variée d'entrées. Donc, peu importe le nombre d'entrées que possède un objet, il est possible de les passer à la fonction sous la forme d'un seul paramètre. Le nombre d'entrées est indiqué par le deuxième paramètre.
- *GetOutput(hog, nbOutput)* lit les sorties de l'objet circuit passé en paramètre. Cette fonction vérifie d'abord si l'objet existe dans la liste des objets HOG. S'il n'existe pas, elle génère un message d'erreur et le système s'arrête comme dans le cas de *setInput*(). Si l'objet existe, elle vérifie s'il a eu des entrées. Dans la négative, il sera traité comme un cas d'erreur. Si l'objet est bien actif sur le RPU, (et donc présente des entrées), cette fonction ira lire ses sorties. Encore une fois, le langage Java permet de retourner plusieurs sortie simultanément en inscrivant

ces sorties dans un tableau. Le deuxième argument de la fonction nous indique à combien de sorties l'objet a été soumis.

- *DestroyHOG(hog)* détruit définitivement un objet HOG en l'enlevant de la liste des objets HOG. Lorsqu'une application termine son exécution, notre gestionnaire fait disparaître tous les objets circuits qui lui appartiennent.
- *CleanUp(appthread)* permet de détruire tous les objets circuits qui appartenaient à l'application passée en paramètre. Cette fonction est appelée avant qu'une application ne termine son exécution. Elle permet de procéder à un nettoyage de système.

Il est à noter que dans ce projet, nous présumons que pour tous les objets HOG, le temps entre le moment d'écriture des entrées et le moment où la sortie est disponible représente invariablement un seul coup d'horloge. Cela signifie qu'une fois les entrées écrites, notre gestionnaire est en mesure de lire la sortie sans vérifier son état de préparation. Dans les applications réelles, il est possible de rencontrer des cas où la sortie des objets HOG exige des durées variables selon les objets. Il faudra dans ce cas qu'après avoir écrit des entrées, notre gestionnaire parvienne à vérifier l'état de préparation de la sortie avant de les lire. Il y a deux façons de résoudre ce problème :

1- L'application hôte de chaque objet HOG fournit l'information sur le délai de la sortie. À cet effet, il est très possible et facile d'ajouter un champ de données dans la classe HOG qui gardera cette information et un autre champ de données qui contiendra le moment d'écriture des entrées. Lorsque l'application hôte voudra lire la sortie de son objet HOG via le gestionnaire, ce dernier pourra faire un simple calcul pour vérifier si la sortie est prête.

2- L'autre possibilité consiste à implanter à l'objet HOG un registre d'un bit qui servira d'indicateur (*flag*). C'est la valeur « 0 » qui sera initialement donné à cet indicateur. Lorsque la sortie est prête, ce registre devra contenir la valeur '1'. À toutes les fois que de nouvelles entrées sont écrites, cet indicateur est remis à « 0 » de telle sorte que lorsque l'application veut lire la sortie de son objet HOG via le

gestionnaire, cet indicateur est vérifié et la sortie ne sera lue que si la valeur de l'indicateur est « 1 ».

Gestionnaire des applications et des HOG

Ce sont des fonctions utilitaires qui assurent la gestion de l'ensemble des applications et l'ensemble des objets HOG. Les applications et les objets HOG sont stockés dans deux listes séparées : la liste des applications et la liste des HOGs. Chaque fois qu'une nouvelle application a besoin des services de notre gestionnaire, elle est ajoutée à la liste des applications. Lorsqu'une application termine son exécution, elle est enlevée de la liste. De la même manière, chaque objet HOG réalisé sur le RPU est ajouté à la liste des objets HOGs. Lorsqu'une l'exécution d'une application est terminée, tous les objets HOGs qui lui appartenaient sont enlevés de la liste des objets HOGs. Ainsi, le gestionnaire garde toujours les informations sur les applications et les objets HOGs courants. Il est particulièrement utile de garder tous ces objets dans une liste. Comme il y a un maximum de quatre objets HOGs qui peuvent simultanément être réalisés sur le RPU, le fait de garder les objets qui sont temporairement enlevés du RPU dans une liste interne du gestionnaire nous assure de pouvoir les retrouver et les remettre sur le RPU rapidement sans affecter les applications hôtes. Il existe également deux fonctions permettant d'afficher la liste des applications et la liste des objets HOGs sur l'écran ou dans un fichier dans le cas ou il faudrait déverminer le système.

Gestionnaire de l'allocation des sections

C'est un ensemble de fonctions privées de la classe qui servent à l'allocation des sections du RPU. Pour gérer les quatre sections efficacement, nous avons implémenté l'algorithme LRU (*Least Recently Used*) en nous inspirant de la gestion de la mémoire paginée dans un système d'exploitation. C'est ainsi que lorsqu'une application fait une demande de section, nous vérifions d'abord s'il reste des sections qui sont libres. Dans l'affirmative, nous l'allouons. Dans l'hypothèse où toutes les sections seraient déjà occupées, on fera appel à la fonction *lru*() pour trouver la section qui a été la moins récemment utilisée parmi celles qui ne sont pas verrouillées. Dans le cas où les quatre sections seraient utilisées (c'est-à-dire que toutes les sections soient occupées par un objet

HOG verrouillés), notre gestionnaire sélectionnera la section la moins récemment utilisée parmi les quatre et la sortie de l'objet qui occupait cette section sera sauvegardée dans un tampon par l'entremise de la fonction *bufferOutput*() de l'objet HOG (voir dans la section précédente pour obtenir des détails sur les fonctions membres de la classe HOG). L'objet qui occupait la section choisie sera retiré du RPU mais restera dans la liste des objets HOG tant que l'exécution de l'application hôte ne sera pas terminée. Ainsi, lorsque l'application aura de nouveau besoin de cet objet, il pourra très rapidement être réalisé sur le RPU. Par la suite, le nouvel objet de l'application qui a fait la demande d'une section est réalisé sur la section ciblée. L'historique de l'utilisation des sections est exprimé par l'attribut « âge » de chaque section (voir section 3.3.3).

Avant d'opter pour l'algorithme LRU, nous avons fait l'essai d'un certain nombre d'autres algorithmes (en particulier les algorithmes FIFO (*First In First Out*) et Deuxième chance). L'algorithme LRU se sera toutefois révélé un bon compromis entre simplicité et efficacité.

Cette section nous aura permis d'introduire notre système gestionnaire du RPU XC6200. Il est destiné à la carte H.O.T. de la compagnie VCC (*Virtual Computer Corporation*). Il sert l'interface entre les applications contenant des objets matériels et le système H.O.T.. Il permet le partage du RPU entre plusieurs applications et entre les multiples objets circuits d'une même application. Dans la section qui suit, nous expliquerons le mode l'utilisation de notre gestionnaire.

3.4 L'utilisation du gestionnaire

Pour utiliser notre système gestionnaire du RPU XC6200, il suffit d'inclure, dans l'application que l'on désire faire exécuter, un objet instance de la classe gestionnaire et les objets circuits prédéfinis à l'aide de l'environnement JERC. À partir de l'objet gestionnaire, l'application est en mesure d'utiliser l'ensemble des fonctions API du gestionnaire de manière à réaliser ses objets sur la puce RPU et communiquer avec eux. Pour simuler le partage du RPU entre de multiples applications, nous avons implémenté ces applications en tant que *threads* Java. Nous avons également utilisé un programme de test basé sur la méthode *main*() pour lancer toutes les unités d'exécution des applications. La figure 3-23 présente un exemple d'une application simple. Pour des raisons de clarté, seul le code de la partie relative à l'utilisation de notre gestionnaire a été inclus. Pour les mêmes raisons, le code ne contient que peu de commentaires ; des explications détaillées suivront tout de suite après. On notera également que les applications ayant servi à cette simulation n'ont effectué que des opérations utilisant des objets matériels. Une application réelle impliquerait, bien sûr, d'autres types d'opérations.

```
1     import  Pci6200;
2     import  java.io.*;
3
4     public class Application1 extends Thread{
5           private GesRPU ges;
6           private Pci6200 pci6200;
7
8           public Application1(GesRPU _ges,Pci6200 _pci6200)
9           {
10                ges = _ges;
11                pci6200 = _pci6200;
12                this.setName("Thread_app1");
13                ges.addThread(this.getName());
14          }
15
16          public void run()
17          {
18
19                int [] input = new int [2];
20                AdderR adder1,adder2;
21                adder1 = new AdderR(16);
```

```
22              adder2 = new AdderR(14);
23              for(int i=0;i<5;i++)
24              {
25                    // 1st object
26                    adder1.setThreadID(this.getName());
27                    ges.load(adder1);
28                    input[0] = 55500;
29                    input[1] = 27;
30                    ges.setInput(adder1,2,input);
31        System.out.println("55500+27="+ges.getOutput(adder1,1)[0]);
32
33                    // 2nd object
34                    adder2.setThreadID(this.getName());
35                    ges.load(adder2);
36                    input[0] = 16300;
37                    input[1] = 27;
38                    ges.setInput(adder2,2,input);
39        System.out.println("16300+27="+ges.getOutput(adder1,1)[0]);
40                    input[0] = 32;
41                    input[1] = 29;
42                    ges.setInput(adder2,2,input);
43        System.out.println("32+29="+ges.getOutput(adder1,1)[0]);
44
45                    // 3rd object
46                    adder2 = new AdderR(4);
47                    adder2.setThreadID(this.getName());
48                    ges.load(adder2);
49                    input[0] = 6;
50                    input[1] = 7;
51                    ges.setInput(adder2,2,input);
52        System.out.println("6+7="+ges.getOutput(adder1,1)[0]);
53                    input[0] = 2;
54                    input[1] = 9;
55                    ges.setInput(adder2,2,input);
56        System.out.println("2+9="+ges.getOutput(adder1,1)[0]);
57              }
58
59              ges.displayHOG();
60              ges.displaySections();
61              ges.removeThread(this.getName());
61        }
63
64
65        protected void finalize()
66        {
67              ges.displayHOG();
68              ges.cleanUp(this.getName());
69              ges.displayHOG();
70        }
71  }
```

Figure 3-23 Une application utilisant notre gestionnaire implantée comme un *thread* en Java

Dans cet exemple, l'application est implémentée en tant que sous-classe de la classe *thread* (ligne 4). Elle contient une donnée membre « ges » du type gestionnaire représentant le gestionnaire et une donnée membre « pci6200 » du type Pci6200 représentant la puce RPU XC6200. On constate à la ligne 20 que dans la méthode *run*() de la classe, nous nous retrouvons deux objets matériels respectivement appelés adder1 et adder2. Ces deux objets sont du type adderR, un type d'objet prédéfini comme additionneur dans l'environnement JERC. De la même manière, nous pourrions prédéfinir n'importe quel objet matériel utile pour l'application et l'instancier dans cette dernière. Aux lignes 21 et 22, les objets adder1 et adder2 sont définis comme des additionneurs de 16 et 14 bits. Nous savons déjà que la flexibilité de l'environnement JERC permet de définir des objets paramétrés. Dans notre exemple, le nombre de bits peut donc être transmis sous forme de paramètres. La réalisation des objets sur le RPU est effectuée aux lignes 27 et 34 à l'aide de la fonction *load*(). Les lignes 28 et 29 servent à la préparation des entrées dans un tableau et la ligne 30 commande à l'application d'écrire les entrées de l'objet matériel adder1. À la ligne 31, on lit la sortie, la somme des deux entrées. On fait appel à la même procédure un peu plus loin dans le programme afin de simuler l'accès dynamique aux objets matériels dans une application.

Nous avons implémenté plusieurs applications semblables à celle qui est illustrée à la figure 3-23 avec quelques autres objets matériels. Pour simuler le partage du RPU par plusieurs applications, nous avons lancé tous les *threads* des applications à l'aide d'un programme de test contenant la fonction *main*(). La figure 3-24 présente ce programme de test :

```
1     import Pci6200;
2     import java.io.*;
3     public class TestGes
4     {
5       public static void main(String[] args)throws IOException
6       {
7         Pci6200 pci6200 = new Pci6200N(null);
8
9         pci6200.connect();
10        if (pci6200.isConnected() == false)
11        {
12          System.out.println("Could not connect to hardware");
13          System.exit(-1);
14        }
15
16         /* Reset the hardware */
17        pci6200.reset();
18        pci6200.setDeviceId(1);
19
20         /* define the output file */
21        FileOutputStream stream = new FileOutputStream("output");
22        PrintWriter writer = new PrintWriter(stream);
23
24        GesRPU ges = new GesRPU(pci6200,writer);
25
26        Application1 app1 = new Application1(ges,pci6200,writer);
27        Application2 app2 = new Application2(ges,pci6200,writer);
28        app1.setPriority(Thread.NORM_PRIORITY-1);
29        app1.start();
30        app2.start();
31    }
32   }
```

Figure 3-24 Le programme test qui lance toutes les applications

L'une des premières étapes de ce programme consiste à instancier un objet de la classe Pci6200 qui représentera la puce RPU XC6200. À la ligne 9, nous essayons d'établir le contact avec cette puce. Si la connexion réussit, nous effectuons quelques opérations d'initialisation de la puce aux lignes 17 et 18. Nous créons ensuite (ligne 24) une instance d'objet de notre gestionnaire puis nous passons cette instance aux applications app1 et app2 (lignes 26 et 27). Finalement, nous lançons les applications (lignes 29 et 30). Les applications étant implémentées sous forme de *threads* Java, elles s'exécuteront simultanément de façon à simuler des applications concurrentes partageant les ressources matérielles.

Ceci termine le troisième chapitre de notre mémoire. Il nous a permis de présenter notre logiciel conçu pour la gestion d'un RPU XC6200. Après avoir décrit le système H.O.T. de la compagnie VCC (*Virtual Computer Corporation*), nous avons exposé les principales caractéristiques du RPU XC6200 que ce système comporte. Par la suite, nous avons présenté l'environnement JERC (*Java Environment for re-configurable Computing*) avec lequel notre gestionnaire a été développé. Enfin, nous avons expliqué en détail le fonctionnement de notre gestionnaire et avons décrit des programmes permettant de simuler son utilisation dans un environnement Java. On trouvera au chapitre suivant les résultats de ces simulations.

CHAPITRE 4 - RÉSULTATS

Pour valider le logiciel gestionnaire présenté au troisième chapitre, nous avons procédé à un certain nombre de simulations. Pour ce faire, nous avons créé deux applications dans lesquelles plusieurs objets matériels sont instanciés et implémentés sur le RPU XC6200. Ces deux applications sont implémentées comme étant deux *threads* concurrents qui accèdent au RPU par la même instance du gestionnaire. Les résultats obtenus prouvent que notre gestionnaire est tout à fait fonctionnel. Il a permis le partage de ressources matérielles entre plusieurs applications comme prévu.

4.1 La validité du design

Pour prouver la validité du concept de notre gestionnaire de RPU, nous avons effectué des simulations avec plusieurs applications. Dans la section 3.4 nous avons déjà présenté le squelette d'une des applications nommée « application1 » (voir la figure 3-23). Cette application contient trois objets matériels : un additionneur de 16 bits, un additionneur de 14 bits et un additionneur de 4 bits. Répétitivement, ces trois objets sont implémentés sur le RPU par l'entremise d'une boucle *for*. Des opérations d'entrée / sorties sont également effectuées sur ces objets. Pour vérifier si notre gestionnaire permet le partage du RPU, nous avons créé une deuxième application que nous avons nommée « application2 ». Comme dans le cas de la première application, elle comporte trois objets matériels : un additionneur de 10 bits, un compteur de 10 bits et un compteur de 8 bits. Ces trois objets sont implémentés répétitivement de la même manière que dans l'application1 et des opérations d'entrées / sorties sont effectuées sur chacun d'eux. Les deux applications étant implémentées sous forme de sous-classe de la classe *thread* Java, elles se comporteront comme deux *threads* concurrents (voir la figure 3-24).

TABLEAU 4-1 SOMMAIRE DE LA SIMULATION - APPLICATIONS AVEC PRIORITÉS DIFFÉRENTES

	Application1	Application2
Nombre de configurations de matériel	15	15
Nombre d'écriture d'entrées	25	30
Nombre de lecture de sorties	25	30
Nombre de fautes de matériel	26	
Nombre de succès de matériel	4	

Au cours de cette simulation, les deux applications ont chacune effectué cinq itérations d'une boucle *for* contenant le code des implémentations et des entrées / sorties de trois objets matériels. Le tableau 4-1 présente les statistiques de cette simulation dont le fichier journal de la simulation se trouve à l'annexe B. Nous observons qu'il y a eu au total 30 configurations de matériel parmi lesquels il y a eu 4 succès : Il est arrivé quatre fois que l'objet à configurer était déjà présent sur le RPU). À noter que nous avons augmenté la priorité de l'application2 par rapport à celle de l'application1. On peut remarquer dans l'appendice B qu'il y a plus d'activité de l'application2 que de l'application1 au début.

TABLEAU 4-2 SOMMAIRE DE LA SIMULATION - APPLICATIONS AVEC MÊMES PRIORITÉS

	Application1	Application2
Nombre de configurations de matériel	15	15
Nombre d'écriture d'entrées	25	30
Nombre de lecture de sorties	25	30
Nombre de fautes de matériel	30	
Nombre de succès de matériel	0	

Nous avons également procédé à une simulation impliquant le même niveau de priorité pour les deux applications (on trouvera le fichier journal de cette simulation à l'annexe C

et le sommaire des résultats obtenus lors de cette deuxième simulation se retrouve au tableau 4.2). Contentons-nous ici de dire que lorsque les deux applications ont la même priorité, le taux de succès matérielles tombe à zéro. Ce résultat peut être expliqué par le trop grand nombre d'objets à implanter par rapport au nombre de sections disponibles sur le RPU. En donnant des niveaux de priorité différents entre les deux applications, on fait diminuer le nombre d'objets à implanter sur le RPU pendant un moment.

En vérifiant le contenu du fichier journal des simulations, nous voyons que les deux applications partagent le RPU de concert. Nous pouvons vérifier que les lignes qui marquent une lecture de sortie nous donnent les bons résultats du calcul. Ceci démontre que notre gestionnaire fonctionne bien : il gère l'utilisation croisée du matériel par plusieurs processus, le bon bloc logique étant implémenté au bon moment.

4.2 La performance

L'accroissement de la performance des systèmes ordinés est l'un des principaux buts recherchés lorsqu'il s'agit de concevoir un système matériel / logiciel. Malheureusement, certaines propriétés de ces systèmes peuvent affecter leur performance et remettre en cause l'intérêt d'y avoir recours. Parmi ces facteurs, le temps de configuration du RPU est sans doute celui dont il faut le plus tenir compte. En terme de performance, est-il préférable d'implanter une fonction sur le RPU plutôt que de l'implanter en logiciel ? Pour le savoir, nous allons procéder à une analyse mathématique.

Considérons le cas où une fonction F contenant une séquence de N instructions qui vont être exécutées M fois sur un CPU et supposons que chaque instruction prenne un cycle d'horloge du CPU, noté t_{cpu}, alors le temps d'exécution T_{cpu} du CPU sera :

$$T_{cpu} = (N * M * t_{cpu}) \tag{4.1}$$

Si cette fonction est implantée sur le RPU, avec un temps de configuration T_r et que les N instructions peuvent être exécutées dans un cycle d'horloge du RPU t_{rpu}, alors le temps d'exécution T_{rpu} du RPU est :

$$T_{rpu} = T_r + (M * t_{rpu}) \tag{4.2}$$

Pour qu'il soit profitable d'implanter cette fonction sur le RPU, il faut que T_{rpu} soit plus petit que T_{cpu}. Alors on obtient alors la condition suivante :

$$M \geq \frac{T_r}{(N * t_{cpu}) - t_{rpu}} \tag{4.3}$$

L'équation 4.3 peut être simplifiée en considérant la relation entre un cycle d'horloge du CPU et un cycle d'horloge du RPU. Dans notre cas, le microprocesseur de la machine hôte utilisé est un Pentium II cadencé à 350 MHz, tandis que l'horloge du RPU varie

entre 33 et 100 MHz. En considérant que T_{rpu} est approximativement quatre fois T_{cpu}, nous obtenons l'équation suivante :

$$M \geq \frac{4*R}{N-4} \tag{4.4}$$

Dans l'équation 4.4, R est le nombre de cycles nécessaires pour configurer la fonction sur le RPU. Cette équation nous indique approximativement la valeur minimum du nombre de fois que la fonction F devra être exécutée pour qu'il soit profitable de l'implanter sur le RPU. Plus la valeur de M est grande plus il est profitable de l'implanter sur le RPU. De même, plus la valeur de N est grande (N est le nombre d'instructions en logiciel remplacées par l'implantation sur le RPU), moins la valeur minimum de M est élevée. Cela signifie que le gain de performance d'une implémentation sur un RPU augmente à mesure que la fonction à exécuter est complexe. (On notera que nous n'avons pas considéré le temps d'entrées / sorties sur l'objet implémenté sur le RPU puisque pour une fonction complexe, le temps d'entrées / sorties est négligeable par rapport au temps de configuration.)

Dans le présent projet, nous avons divisé la surface du RPU XC6200 en quatre sections et notre système n'a été testé surtout que sur des petits circuits. Cela explique le peu de gain en performance que nous avons obtenu par rapport à des systèmes ne faisant appel qu'à des logiciels. La configuration d'une section du RPU a demandé en moyenne 13 millisecondes. Le temps d'exécution des mêmes fonctions implémentées en logiciel étant lui aussi négligeable, c'est ce temps de configuration qui fait la différence entre les deux modes d'exécution. Cela n'a pas un impact majeur sur les résultats de ce travail, notre projet ayant surtout pour but de concevoir et de vérifier le fonctionnement d'un prototype d'un système de gestion de matériel, le gain en performance étant souhaitable mais accessoire. Les recherches dans le domaine nous démontrent à cet égard que l'utilisation du RPU XC6200 peut améliorer considérablement la performance des applications de traitement d'images [KeDu97]. Dans une note d'application de la compagnie VCC [VCC97] par exemple, on présente les résultats obtenus avec un corrélateur d'images

implémenté sur un RPU XC6216. On rapporte qu'au moment où le système a été utilisé (1996), l'utilisation du RPU améliorait la performance de 20 % par rapport à celle d'un microprocesseur haut de gamme de cette époque (Pentium 133 MHz). Pour une image de 512 * 512 pixels, le temps de corrélation est approximativement de 30 ms avec le corrélateur matériel, avec un temps de configuration de moins de 0.2 ms. La même opération effectuée par un corrélateur purement logiciel dure environ 38 ms.

4.3 L'algorithme d'allocation de sections

Nous avons déjà mentionné que nous avons divisé la surface du RPU en quatre sections. Nous avons également dit que notre gestionnaire gérait ces sections de la même manière qu'un système d'exploitation gère la mémoire paginée. Quand toutes les sections sont occupées, le gestionnaire doit choisir une section et remplacer l'objet qui l'occupe par le nouvel objet. Plusieurs algorithmes sont susceptibles de gérer cette allocation de sections. Nous avons fait l'essai des algorithmes LRU (*Least Recently Used*) et FIFO (*First In First Out*). Les résultats obtenus nous indiquent que l'algorithme LRU est légèrement plus efficace que l'algorithme FIFO : pour un total de 60 reconfigurations de sections, l'algorithme LRU nous a donné en moyenne 5 % de taux de succès de plus que ce dernier. On notera toutefois que dans la simulation qui a servi à nos tests, les applications créaient les objets sur le RPU aléatoirement, ce qui n'est pas le cas dans un système réel. Nous sommes consciente que pour bien choisir l'algorithme d'allocation de sections, il est essentiel de procéder à une étude portant sur la tendance habituelle d'utilisation des objets dans les cas réels et choisir ou créer un algorithme en conséquence.

Le chapitre 4 se termine ici. Nous y avons présenté les résultats obtenus lors de plusieurs simulations effectuées sur notre gestionnaire de RPU. Ces simulations nous ont prouvé que ce gestionnaire est tout à fait fonctionnel dans un contexte de simulation. Nous avons par la suite discuté des facteurs qui influencent les gains de performance des matériel / logiciel et des conditions pour lesquelles il est profitable d'implanter des fonctions sur un RPU. Enfin, nous avons comparé l'efficacité de deux algorithmes d'allocation de sections. Le prochain chapitre est la conclusion de ce mémoire.

CHAPITRE 5 - CONCLUSION ET TRAVAUX FUTURS

Ce mémoire a présenté un système de gestionnaire du RPU XC6200. La raison d'être d'un tel gestionnaire est de permettre l'utilisation de matériel dédié dynamiquement afin d'améliorer la performance et la flexibilité des systèmes logiciels. À l'aide de ce gestionnaire, nous permettons également le partage de matériel entre plusieurs applications. Notre gestionnaire a été conçu pour être exécuté sur le système H.O.T. de la compagnie Xilinx, le RPU XC6200 sur lequel il est basé comportant des caractéristiques qui nous ont paru intéressantes : il supporte la programmation partielle et sa configuration est plus rapide que les FPGA génériques. C'est l'outil JERC qui nous a servi d'environnement de programmation. Il avait l'avantage de permettre l'intégration dans une même partie de code des diverses étapes de la conception d'un système de co-design matériel / logiciel. Le gestionnaire de RPU ainsi conçu permet aux applications de créer des objets matériels, de les programmer sur le RPU et de les communiquer de façon dynamique. Il peut, de plus, permettre à plusieurs applications de partager le même matériel de manière transparente et sécuritaire. Notre gestionnaire gère le RPU de la même façon qu'un système d'exploitation gère la mémoire paginée : il divise le RPU en quatre sections puis il gère l'allocation des sections aux applications en utilisant l'algorithme LRU. Notre gestionnaire permet également de sauvegarder dans un tampon tous les objets créés par les applications et toutes leurs unités d'exécution, ce qui permet un échange rapide des objets entre la mémoire et le RPU.

Les résultats obtenus sont assez prometteurs : les simulations effectuées démontrent que notre gestionnaire permet le partage du RPU entre plusieurs applications de façon sécuritaire. La simplicité des fonctions implémentées sur le matériel ne nous a toutefois pas permis d'observer une amélioration de la performance par rapport à un système basé uniquement sur un logiciel. Mais des études dans le domaine ont démontré que l'augmentation de la complexité des fonctions implémentées sur le RPU s'accompagne d'un gain en performance.

La prochaine étape de ce projet consistera à construire un compilateur qui permettra de diviser automatiquement les applications en parties logicielles et matérielles. À partir d'une spécification, le compilateur devra pouvoir identifier les sections du type « calcul intensif » qui consomment beaucoup de temps d'exécution, et les remplacer par du code qui permettra d'implémenter un objet matériel sur le co-processeur ciblé et de communiquer avec lui.

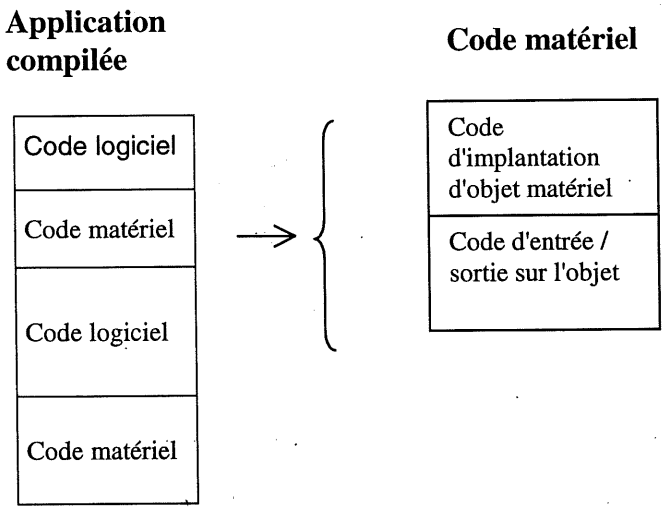

Figure 5-1 L'image d'une application compilée par un compilateur co-design

La figure 5-1 présente le schéma d'une application compilée par ce type de compilateur. L'application y est divisée en sections de code logiciel et de code matériel. Les sections de code matériel incluent le code servant à l'implémentation de l'objet matériel sur le RPU et celui permettant de communiquer avec l'objet. Le système complet devrait ressembler à la figure 5-2.

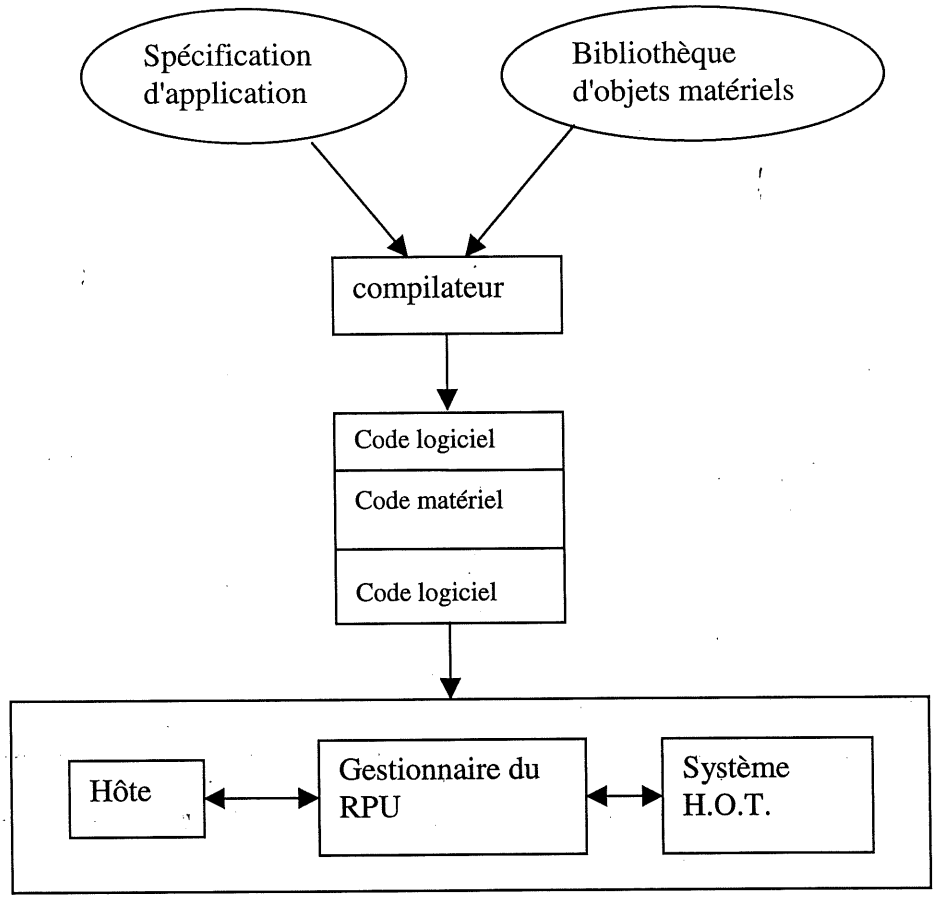

Figure 5-2 Le système co-design complet avec un compilateur co-design

Dans l'état actuel de la recherche, il manque encore d'outils de synthèse automatisés qui parviendraient à créer un circuit logique à partir d'une description écrite en langage de programmation standard (i.e. VHDL ou C) et généreraient les données d'implémentation requises pour le RPU. L'environnement JERC permet d'implanter des circuits directement sur le RPU sans passer par les étapes de placement et de routage. Mais pour ce faire, une bonne connaissance de l'architecture et des détails techniques du RPU sont nécessaires. De plus, ce n'est pas une tâche facile car chaque composant du circuit doit être programmé de façon très détaillée. Cela ne représente pas un obstacle majeur lorsqu'il s'agit d'un circuit relativement simple ou d'un circuit ayant une structure

répétitive. L'utilisation de l'environnement JERC devient toutefois vite fastidieuse lorsque la taille et la complexité du circuit augmentent. En effet, la bibliothèque de fonctions actuellement définie dans l'outil JERC ne contient pas beaucoup de composants de haut niveau. Pour réaliser les circuits que nous avons testés dans le présent projet, nous avons dû créer plusieurs composants de bas niveau et construire des circuits de haut niveau à partir de ces composants. C'est en partie pour cette raison que nous n'avons pu véritablement tester notre gestionnaire avec des circuits de plus grande complexité. Pour que l'outil JERC puisse être utilisé sur une plus grande échelle, il faudra établir une bibliothèque beaucoup plus sophistiquée, qui comprendra davantage de composants de haut niveau.

Bien que les activités de recherche dans le domaine de l'informatique re-configurable aient débuté il y a déjà plus d'une décennie, cette technologie n'en est encore qu'à ses premiers balbutiements. Nous avons voulu, dans ce projet, examiner la possibilité de diviser une puce RPU en plusieurs sections et de gérer l'allocation des sections avec un algorithme inspiré de la gestion de la mémoire paginée. Nous pensons que beaucoup d'autres notions et mécanismes destinés initialement aux systèmes informatiques conventionnels pourraient être réutilisés dans le domaine de l'informatique re-configurable. Il n'est qu'à penser ici au mécanisme de gestion des systèmes de mémoire cache qui pourrait servir à conserver les données de configuration dans le cas de ressources matérielles devant être reconfigurées fréquemment et rapidement. Une hiérarchie de mémoires cache rendrait sans doute cette reconfiguration encore plus rapide... mais le prototype d'un tel système reste encore à concevoir.

La plupart des activités de recherche dans le domaine de l'informatique re-configurable sont basées sur des FPGA génériques dont l'architecture n'est pas toujours optimisée pour ce type d'application. Le besoin de configuration dynamique est l'une des caractéristiques les plus marquées du domaine de l'informatique re-configurable. Mais elle est aussi la caractéristique la moins bien supportée par les FPGA génériques. Les RPU XC6200 permettent quant à eux non seulement de faire de la configuration dynamique mais ils supportent la configuration partielle. À cet égard, ils apparaissent

comme étant les mieux adaptés à l'informatique re-configurable. L'utilisation commerciale de ce type d'informatique est encore très limitée et, pour l'instant, on ne la retrouve surtout que dans les laboratoires de recherche. Il ne fait aucun doute que cette situation devrait changer avec l'avènement de nouveaux RPU plus performants et l'émergence de systèmes logiciels de gestion qui faciliteront leur utilisation. Le RPU XC6216 qui a servi à ce projet ne possède que l'équivalent d'environ 16 000 portes logiques. Si ce type de puce est adéquat pour le prototypage de systèmes de gestion du matériel comme le nôtre, son utilisation dans des systèmes réels ne sera réaliste que lorsque les RPU atteindront la puissance des FPGA standards qui comptent aujourd'hui plusieurs millions de portes logiques.

ANNEXE A - LISTE DES SYMBOLES ET DES ABREVIATIONS

ALU	: Arithmetic and Logic Unit
ASIC	: Application Specific Integrated Circuit
ASIP	: Application Specific Integrated Processor
BIOS	: Basic Input / Output System
CAD	: Computer Aided Design
CFSM	: Co-design Finite State Machine
CLB	: Configurable Logic Bloc
CMOS	: Complementary Metal Oxide Semiconductor
CPU	: Central Processing Unit
DSP	: Digital Signal Processing system
EEPROM	: Electrical Erasable Programmable Read Only Memory
EPROM	: Erasable Programmable Read Only Memory
FIFO	: First In First Out
FPGA	: Field Programmable Gate Array
GE	: General Electric
HDL	: Hardware Description Language
HOG	: Hardware Object Generator
H.O.T.	: Hardware Object Technology
ILP	: Instruction Level Parallelism
IOB	: Input / Output Bloc
JERC	: Java Environment for Re-configurable Computing
JVM	: Java Virtual Machine
LRU	: Least Recently Used
LSI	: Large Scale Integration
LSS	: Logic Synthesis System
MPEG	: Moving Picture Expert Group
MPU	: Micro-Processing Unit
PAL	: Programmable Array Logic device

PC : Personal Computer

PCI : Peripheral Component Interconnect

PLD : Programmable Logic Device

PROM : Programmable Read Only Memory

RAM : Random Access Memory

RPU : Re-configurable Processing Unit

RTS : Run Time System

SCSI : Small Computer System Interface

SLU : Swappable Logic Unit

SPLD : Simple Programmable Logic Device

SRAM : Statique RAM

TTL : Transistor-Transistor Logic

UF : Unités Fonctionnelles

UML : Unified Modeling Language

VCC : Virtual Computer Corporation

VLIW : Very Long Instruction Word

ANNEXE B - JOURNAL DE SIMULATION - PRIORITÉS DIFFÉRENTES

Ce fichier journal rend compte de la simulation de notre gestionnaire que nous avons effectuée. Les lignes débutant par la séquence « --> load » enregistrent une configuration du RPU. Dans notre cas, il s'agit d'une configuration partielle sur une des quatre sections du RPU. L'enregistrement de la simulation contient aussi des informations sur le nom de l'objet et l'hôte de l'objet. Par exemple, la ligne « --> load : adder of 10 of Thread_app2 » nous indique que l'application1 a configuré un additionneur de 10 bits sur le RPU.

Les lignes débutant par « >>>>> setInput » signalent une écriture d'entrée sur un objet implémenté sur le RPU. Elles permettent également de connaître l'identité de l'application et de l'objet impliqués par cette écriture. Par exemple, la ligne « >>>>> setInput : counter of 8 of Thread_app2 » signifie que l'application2 a écrit une entrée sur son objet, le compteur de 8 bits.

Les lignes débutant par « <<<<< getOutput <<<<< getOutput app1 : » ou « <<<<< getOutput <<<<< getOutput app2 : » rendent compte d'une lecture de sortie. Par exemple, la ligne « <<<<< getOutput <<<<< getOutput app1 : adder(4) : 2+9=11 » signifie que l'application1 a lu la sortie de son objet, l'additionneur de 4 bits, et que le résultat lu est 11 (les entrées étant 2 et 9)

```
--> load : adder of 10 of Thread_app2
>>>>> setInput : adder of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : 930+27=957
>>>>> setInput : adder of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : 32+29=61
--> load : counter of 10 of Thread_app2
>>>>> setInput : counter of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(10) until 11 11
>>>>> setInput : counter of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(10) another 21 32
--> load : counter of 8 of Thread_app2
>>>>> setInput : counter of 8 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(8) until 11 11
```

--> load : adder of 16 of Thread_app1
>>>>> setInput : adder of 16 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(16) : 55500+27=55527
>>>>> setInput : counter of 8 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(8) another 21 32
--> load : adder of 14 of Thread_app1
--> load : adder of 10 of Thread_app2
>>>>> setInput : adder of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : 930+27=957
>>>>> setInput : adder of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : 32+29=61
--> load : counter of 10 of Thread_app2
>>>>> setInput : counter of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(10) until 11 11
>>>>> setInput : counter of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(10) another 21 32
--> load : counter of 8 of Thread_app2
>>>>> setInput : counter of 8 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(8) until 11 11
>>>>> setInput : counter of 8 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(8) another 21 32
--> load : adder of 10 of Thread_app2
>>>>> setInput : adder of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : 930+27=957
>>>>> setInput : adder of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : 32+29=61
--> load : counter of 10 of Thread_app2
>>>>> setInput : counter of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(10) until 11 11
>>>>> setInput : counter of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : c ount(10) another 21 32
--> load : counter of 8 of Thread_app2
>>>>> setInput : counter of 8 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(8) until 11 11
>>>>> setInput : counter of 8 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(8) another 21 32
--> load : adder of 10 of Thread_app2
>>>>> setInput : adder of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : 930+27=957
>>>>> setInput : adder of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : 32+29=61
--> load : counter of 10 of Thread_app2
>>>>> setInput : counter of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(10) until 11 11
>>>>> setInput : counter of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(10) another 21 32
--> load : counter of 8 of Thread_app2
>>>>> setInput : counter of 8 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(8) until 11 11
>>>>> setInput : counter of 8 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(8) another 21 32

--> load : adder of 10 of Thread_app2
>>>>> setInput : adder of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : 930+27=957
>>>>> setInput : adder of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : 32+29=61
--> load : counter of 10 of Thread_app2
>>>>> setInput : counter of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(10) until 11 11
>>>>> setInput : counter of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(10) another 21 32
--> load : counter of 8 of Thread_app2
>>>>> setInput : counter of 8 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(8) until 11 11
>>>>> setInput : counter of 8 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(8) another 21 32

***end of the application2 : *** run time is 660ms

>>>>> setInput : adder of 14 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(14) : 16300+27=0
--> load : adder of 14 of Thread_app1
>>>>> setInput : adder of 14 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(14) : 32+29=61
--> load : adder of 4 of Thread_app1
>>>>> setInput : adder of 4 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(4) : 6+7=13
>>>>> setInput : adder of 4 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(4) : 2+9=11
--> load : adder of 16 of Thread_app1
>>>>> setInput : adder of 16 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(16) : 55500+27=55527
--> load : adder of 14 of Thread_app1
>>>>> setInput : adder of 14 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(14) : 16300+27=16327
>>>>> setInput : adder of 14 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(14) : 32+29=61
--> load : adder of 4 of Thread_app1
>>>>> setInput : adder of 4 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(4) : 6+7=13
>>>>> setInput : adder of 4 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(4) : 2+9=11
--> load : adder of 16 of Thread_app1
>>>>> setInput : adder of 16 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(16) : 55500+27=55527
--> load : adder of 14 of Thread_app1
>>>>> setInput : adder of 14 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(14) : 16300+27=16327
>>>>> setInput : adder of 14 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(14) : 32+29=61
--> load : adder of 4 of Thread_app1
>>>>> setInput : adder of 4 of Thread_app1

<<<<< getOutput <<<<< getOutput app1 : adder(4) : 6+7=13
>>>>> setInput : adder of 4 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(4) : 2+9=11
--> load : adder of 16 of Thread_app1
>>>>> setInput : adder of 16 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(16) : 55500+27=55527
--> load : adder of 14 of Thread_app1
>>>>> setInput : adder of 14 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(14) : 16300+27=16327
>>>>> setInput : adder of 14 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(14) : 32+29=61
--> load : adder of 4 of Thread_app1
>>>>> setInput : adder of 4 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(4) : 6+7=13
>>>>> setInput : adder of 4 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(4) : 2+9=11
--> load : adder of 16 of Thread_app1
>>>>> setInput : adder of 16 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(16) : 55500+27=55527
--> load : adder of 14 of Thread_app1
>>>>> setInput : adder of 14 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(14) : 16300+27=16327
>>>>> setInput : adder of 14 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(14) : 32+29=61
--> load : adder of 4 of Thread_app1
>>>>> setInput : adder of 4 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(4) : 6+7=13
>>>>> setInput : adder of 4 of Thread_app1
<<<<< getOutput <<<<< getOutput app1 : adder(4) : 2+9=11

***end of the application1 : *** run time is 660ms

ANNEXE C - JOURNAL DE SIMULATION - MÊMES PRIORITÉS

--> load : adder of 16 of Thread_app1

The active HOG objects are :
adder of 16

The HOGs in the 4 sections are :
section 0 : age : 1 adder of 16
section 1 : age : 0 ***not occupied***
section 2 : age : 0 ***not occupied***
section 3 : age : 0 ***not occupied***

--> load : adder of 10 of Thread_app2

The active HOG objects are :

>>>>> setInput : adder of 16 of Thread_app1
<<<<< getOutput app1 : adder(16) : 55500+27=55527
adder of 16 adder of 10

The HOGs in the 4 sections are :
section 0 : age : 2 adder of 16
section 1 : age : 1 adder of 10
section 2 : age : 1 ***not occupied***
section 3 : age : 1 ***not occupied***

>>>>> setInput : adder of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : 930+27=957

>>>>> setInput : adder of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : 32+29=61

--> load : adder of 14 of Thread_app1

The active HOG objects are :
adder of 16 adder of 10 adder of 14

The HOGs in the 4 sections are :
section 0 : age : 5 adder of 16
section 1 : age : 2 adder of 10
section 2 : age : 1 adder of 14
section 3 : age : 3 ***not occupied***

>>>>> setInput : adder of 14 of Thread_app1
<<<<< getOutput app1 : adder(14) : 16300+27=16327

>>>>> setInput : adder of 14 of Thread_app1
<<<<< getOutput app1 : adder(14) : 32+29=61

--> load : counter of 10 of Thread_app2

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10

The HOGs in the 4 sections are :

section 0 :	age : 8	adder of 16
section 1 :	age : 5	adder of 10
section 2 :	age : 2	adder of 14
section 3 :	age : 1	counter of 10

--> load : adder of 4 of Thread_app1

>>>>> setInput : counter of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(10) until 11 11

>>>>> setInput : counter of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(10) another 21 32

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4

The HOGs in the 4 sections are :

section 0 :	age : 3	adder of 4
section 1 :	age : 8	adder of 10
section 2 :	age : 5	adder of 14
section 3 :	age : 1	counter of 10

>>>>> setInput : adder of 4 of Thread_app1
<<<<< getOutput app1 : adder(4) : 6+7=13

>>>>> setInput : adder of 4 of Thread_app1
<<<<< getOutput app1 : adder(4) : 2+9=11

--> load : counter of 8 of Thread_app2

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :

section 0 :	age : 2	adder of 4
section 1 :	age : 1	counter of 8
section 2 :	age : 8	adder of 14
section 3 :	age : 4	counter of 10

>>>>> setInput : counter of 8 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(8) until 11 11

>>>>> setInput : counter of 8 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(8) another 21 32

--> load : adder of 16 of Thread_app1

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :

section 0 :	age : 5	adder of 4
section 1 :	age : 2	counter of 8
section 2 :	age : 1	adder of 16
section 3 :	age : 7	counter of 10

>>>>> setInput : adder of 16 of Thread_app1
<<<<< getOutput app1 : adder(16) : 55500+27=55527

--> load : adder of 10 of Thread_app2

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :

section 0 :	age : 7	adder of 4
section 1 :	age : 4	counter of 8
section 2 :	age : 2	adder of 16
section 3 :	age : 1	adder of 10

--> load : adder of 14 of Thread_app1

>>>>> setInput : adder of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : 930+27=957

>>>>> setInput : adder of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : 32+29=61

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :

section 0 :	age : 3	adder of 14
section 1 :	age : 7	counter of 8
section 2 :	age : 5	adder of 16
section 3 :	age : 1	adder of 10

>>>>> setInput : adder of 14 of Thread_app1
<<<<< getOutput app1 : adder(14) : 16300+27=16327

--> load : counter of 10 of Thread_app2

>>>>> setInput : adder of 14 of Thread_app1
<<<<< getOutput app1 : adder(14) : 32+29=61

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :

section 0 :	age : 1	adder of 14
section 1 :	age : 2	counter of 10
section 2 :	age : 8	adder of 16
section 3 :	age : 4	adder of 10

>>>>> setInput : counter of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(10) until 11 11

--> load : adder of 4 of Thread_app1

The active HOG objects are :

>>>>> setInput : counter of 10 of Thread_app2
<<<<< getOutput <<<<< getOutput app2 : count(10) another 21 32
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :

section 0 :	age : 4	adder of 14
section 1 :	age : 1	counter of 10
section 2 :	age : 2	adder of 4
section 3 :	age : 7	adder of 10

>>>>> setInput : adder of 4 of Thread_app1
<<<<< getOutput app1 : adder(4) : 6+7=13

>>>>> setInput : adder of 4 of Thread_app1
<<<<< getOutput app1 : adder(4) : 2+9=11

--> load : counter of 8 of Thread_app2

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :

section 0 :	age : 7	adder of 14
section 1 :	age : 4	counter of 10
section 2 :	age : 2	adder of 4
section 3 :	age : 1	counter of 8

>>>>> setInput : counter of 8 of Thread_app2
<<<<< getOutput app2 : count(8) until 11 11

>>>>> setInput : counter of 8 of Thread_app2
<<<<< getOutput app2 : count(8) another 21 32

--> load : adder of 16 of Thread_app1

```
The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :
section 0 :      age : 1          adder of 16
section 1 :      age : 7          counter of 10
section 2 :      age : 5          adder of 4
section 3 :      age : 2          counter of 8

>>>>> setInput : adder of 16 of Thread_app1
<<<<< getOutput app1 : adder(16) : 55500+27=55527

--> load : adder of 10 of Thread_app2

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :
section 0 :      age : 2          adder of 16
section 1 :      age : 1          adder of 10
section 2 :      age : 7          adder of 4
section 3 :      age : 4          counter of 8

>>>>> setInput : adder of 10 of Thread_app2
<<<<< getOutput app2 : 930+27=957

>>>>> setInput : adder of 10 of Thread_app2
<<<<< getOutput app2 : 32+29=61

--> load : adder of 14 of Thread_app1

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :
section 0 :      age : 5          adder of 16
section 1 :      age : 2          adder of 10
section 2 :      age : 1          adder of 14
section 3 :      age : 7          counter of 8

>>>>> setInput : adder of 14 of Thread_app1
<<<<< getOutput app1 : adder(14) : 16300+27=16327

>>>>> setInput : adder of 14 of Thread_app1
<<<<< getOutput app1 : adder(14) : 32+29=61

--> load : counter of 10 of Thread_app2

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8
```

The HOGs in the 4 sections are :

section 0 :	age : 8	adder of 16
section 1 :	age : 5	adder of 10
section 2 :	age : 2	adder of 14
section 3 :	age : 1	counter of 10

>>>>> setInput : counter of 10 of Thread_app2
<<<<< getOutput app2 : count(10) until 11 11

>>>>> setInput : counter of 10 of Thread_app2
<<<<< getOutput app2 : count(10) another 21 32

--> load : adder of 4 of Thread_app1

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :

section 0 :	age : 1	adder of 4
section 1 :	age : 8	adder of 10
section 2 :	age : 5	adder of 14
section 3 :	age : 2	counter of 10

>>>>> setInput : adder of 4 of Thread_app1
<<<<< getOutput app1 : adder(4) : 6+7=13

>>>>> setInput : adder of 4 of Thread_app1
<<<<< getOutput app1 : adder(4) : 2+9=11

--> load : counter of 8 of Thread_app2

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :

section 0 :	age : 2	adder of 4
section 1 :	age : 1	counter of 8
section 2 :	age : 8	adder of 14
section 3 :	age : 5	counter of 10

>>>>> setInput : counter of 8 of Thread_app2
<<<<< getOutput app2 : count(8) until 11 11

>>>>> setInput : counter of 8 of Thread_app2
<<<<< getOutput app2 : count(8) another 21 32

--> load : adder of 16 of Thread_app1

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :
section 0 : age : 5 adder of 4
section 1 : age : 2 counter of 8
section 2 : age : 1 adder of 16
section 3 : age : 8 counter of 10

>>>>> setInput : adder of 16 of Thread_app1
<<<<< getOutput app1 : adder(16) : 55500+27=55527

--> load : adder of 10 of Thread_app2

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :
section 0 : age : 7 adder of 4
section 1 : age : 4 counter of 8
section 2 : age : 2 adder of 16
section 3 : age : 1 adder of 10

>>>>> setInput : adder of 10 of Thread_app2
<<<<< getOutput app2 : 930+27=957

>>>>> setInput : adder of 10 of Thread_app2
<<<<< getOutput app2 : 32+29=61

--> load : adder of 14 of Thread_app1

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :
section 0 : age : 1 adder of 14
section 1 : age : 7 counter of 8
section 2 : age : 5 adder of 16
section 3 : age : 2 adder of 10

>>>>> setInput : adder of 14 of Thread_app1
<<<<< getOutput app1 : adder(14) : 16300+27=16327

>>>>> setInput : adder of 14 of Thread_app1
<<<<< getOutput app1 : adder(14) : 32+29=61

--> load : counter of 10 of Thread_app2

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :
section 0 : age : 2 adder of 14
section 1 : age : 1 counter of 10

```
section 2 :     age : 8         adder of 16
section 3 :     age : 5         adder of 10

--> load : adder of 4 of Thread_app1

The active HOG objects are :
adder of 16
>>>>> setInput : counter of 10 of Thread_app2
<<<<< getOutput app2 : count(10) until 11 11

>>>>> setInput : counter of 10 of Thread_app2
<<<<< getOutput app2 : count(10) another 21 32
adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :
section 0 :     age : 5         adder of 14
section 1 :     age : 1         counter of 10
section 2 :     age : 3         adder of 4
section 3 :     age : 8         adder of 10

>>>>> setInput : adder of 4 of Thread_app1
<<<<< getOutput app1 : adder(4) : 6+7=13

>>>>> setInput : adder of 4 of Thread_app1
<<<<< getOutput app1 : adder(4) : 2+9=11

--> load : counter of 8 of Thread_app2

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :
section 0 :     age : 8         adder of 14
section 1 :     age : 4         counter of 10
section 2 :     age : 2         adder of 4
section 3 :     age : 1         counter of 8

>>>>> setInput : counter of 8 of Thread_app2
<<<<< getOutput app2 : count(8) until 11 11

--> load : adder of 16 of Thread_app1

>>>>> setInput : counter of 8 of Thread_app2
<<<<< getOutput app2 : count(8) another 21 32

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :
section 0 :     age : 2         adder of 16
section 1 :     age : 7         counter of 10
```

```
section 2 :     age : 5         adder of 4
section 3 :     age : 1         counter of 8

>>>>> setInput : adder of 16 of Thread_app1

--> load : adder of 10 of Thread_app2
<<<<< getOutput app1 : adder(16) : 55500+27=55527

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :
section 0 :     age : 2         adder of 16
section 1 :     age : 1         adder of 10
section 2 :     age : 7         adder of 4
section 3 :     age : 3         counter of 8

>>>>> setInput : adder of 10 of Thread_app2
<<<<< getOutput app2 : 930+27=957

>>>>> setInput : adder of 10 of Thread_app2
<<<<< getOutput app2 : 32+29=61

--> load : adder of 14 of Thread_app1

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :
section 0 :     age : 5         adder of 16
section 1 :     age : 2         adder of 10
section 2 :     age : 1         adder of 14
section 3 :     age : 6         counter of 8

>>>>> setInput : adder of 14 of Thread_app1
<<<<< getOutput app1 : adder(14) : 16300+27=16327

>>>>> setInput : adder of 14 of Thread_app1
<<<<< getOutput app1 : adder(14) : 32+29=61

--> load : counter of 10 of Thread_app2

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :
section 0 :     age : 8         adder of 16
section 1 :     age : 5         adder of 10
section 2 :     age : 2         adder of 14
section 3 :     age : 1         counter of 10
```

```
>>>>> setInput : counter of 10 of Thread_app2
<<<<< getOutput app2 : count(10) until 11 11

>>>>> setInput : counter of 10 of Thread_app2
<<<<< getOutput app2 : count(10) another 21 32

--> load : adder of 4 of Thread_app1

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :
section 0 :     age : 1         adder of 4
section 1 :     age : 8         adder of 10
section 2 :     age : 5         adder of 14
section 3 :     age : 2         counter of 10

>>>>> setInput : adder of 4 of Thread_app1
<<<<< getOutput app1 : adder(4) : 6+7=13

>>>>> setInput : adder of 4 of Thread_app1
<<<<< getOutput app1 : adder(4) : 2+9=11

--> load : counter of 8 of Thread_app2



***end of the application1 : *** run time is 940ms



The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :
section 0 :     age : 2         adder of 4
section 1 :     age : 1         counter of 8
section 2 :     age : 8         adder of 14
section 3 :     age : 5         counter of 10

>>>>> setInput : counter of 8 of Thread_app2
<<<<< getOutput app2 : count(8) until 11 11

>>>>> setInput : counter of 8 of Thread_app2
<<<<< getOutput app2 : count(8) another 21 32

The active HOG objects are :
adder of 16 adder of 10
```

***end of the application2 : *** run time is930ms

adder of 14 counter of 10 adder of 4 counter of 8

The active HOG objects are :
adder of 16 adder of 10 adder of 14 counter of 10 adder of 4 counter of 8

The HOGs in the 4 sections are :
section 0 : age : 4 adder of 4
section 1 : age : 1 counter of 8
section 2 : age : 10adder of 14
section 3 : age : 7 counter of 10

The HOGs in the 4 sections are :
section 0 : age : 4 adder of 4
section 1 : age : 1 counter of 8
section 2 : age : 10adder of 14
section 3 : age : 7 counter of 10

BIBLIOGRAPHIE

[Axel97] Axelsson, J.: *Architecture Synthesis and Partitioning of Real-Time Systems: A Comparison of Three Heuristic Search Strategies,* In Proceedings of the Fifth Int. Workshop on Hardware/Software Codesign, 1997.

[BeRo97] Benner, T., Ernst, R.: *An Approach to Mixed Systems Co-Synthesis,* In Proceedings of the Fifth Int. Workshop on Hardware/Software Codesign, 1997.

[Breb96] Brebner, G.: *A Virtual Hardware Operating System for the Xilinx XC6200,* Lecture Note in Computer Science : Field Programmable Logic FPL'96, 1996, Page 327-336.

[CMM97] Carchoilo, V., Malgeri, M., Mangioni, G.: *An Approach to the Synthesis of HW and SW in CoDesign,* In Proceedings of the Fifth Int. Workshop on Hardware/Software Codesign, 1997.

[CRC98] Passerone, C., Passerone, R., Sansoè, C. : *Modeling Reactive System in Java,* In Proceedings of the Sixth Int. Workshop on Hardware/Software Codesign, 1998.

[DATA97] *Xilinx XC6200 Filed Programmable Gate Arrays Data Sheet*, 1997, http://www.xilinx.com.

[Edwa97] Edwards, M.: *Software Accelerating Using Coprocessors: Is it Worth the Effort?*, In Proceedings of the Fifth Int. Workshop on Hardware/Software Codesign, 1997.

[FDI97] Freund, L., Dupont, D., Israël, M.: *Interface Optimization During Hardware-Software Partitioning,* In Proceedings of the Fifth Int. Workshop on Hardware/Software Codesign, 1997.

[FlBu97] Fleischmann, J., Buchenrieder, K.: *Co-Design of Hardware/Software System based on Java Specifications,* Tech. Report TUM-LRE-97-4, Tech. Univ. Of Munich, 1997.

[FlBu98] Fleischmann, J., Buchenrieder, K.: *A Hardware/Software Prototyping Environment for Dynamically Reconfigurable Embedded Systems,* In Proceedings of the Sixth Int. Workshop on Hardware/Software Codesign, 1998.

[GuLe99] Steven A. Guccione and Delon Levi. *The Advantages of Run-Time Reconfiguration.* In John Schewel et al., editors, Reconfigurable Technology: FPGAs for Computing and Applications, Proc. SPIE 3844, Bellingham, WA, September 1999, pages 87-92.

[HBG98] Hollstein, T., Becker, J., Kirschbaum, A.: *HiPART: A New Hierarchical Semi-Interactive HW/SW Partitioning Approach with Fast Debugging for Real-Time Embedded Systems,* In Proceedings of the Sixth Int. Workshop on Hardware/Software Codesign, 1998.

[HeOl97] Helaihel, R., Olukotun, K.: *Java as a Specification Language for Hardware/Software Systems,* In Proceedings Int. Conf. On Computer-Aided Design (ICCAD), 1997.

[JERC97] Lechner, E and Guccione, S. A. : *The Java Environment for Re-configurable Computing,* In Proceeding of Seventh International Workshop on Field Programmable Logic and Applications, Springer LNCS 1304, 1997, pages 284-293.

[JMP97] Jensen, D.C.R., Masen, J. Pedersen, S.: *The Importance of Interfaces: A HW/SW Codesign Case Study,* In Proceedings of the Fifth Int. Workshop on Hardware/Software Codesign, 1997.

[KeDu97] Kean, T., Duncan,A. : *A 800 Pixel/sec Reconfigurable Image Correlator on XC6216.* In Proceedings Int. Workshop on Field programmable logic and applications, 1997.

[KnMa98] Knudsen, P. V., Madsen, J.: *Communication Estimation for Hardware/Software Codesign,* In Proceedings of the Sixth Int. Workshop on Hardware/Software Codesign, 1998.

[KuRo97] Kumar, S., Rose, F.: *A Codesign Environment Supporting Hardware/Software Modeling at Different Levels of Detail,* In Proceedings of the Fifth Int. Workshop on Hardware/Software Codesign, 1997.

[KYI97] Kimura, S., Yukishita, M., Itou, Y.: *A Hardware/Software CoDesign Method for a General Purpose Reconfigurable Co-Processor,* In Proceedings of the Fifth Int. Workshop on Hardware/Software Codesign, 1997.

[LRD98] Lajolo, M., Raghunathan, A., Dey, S.: *A Case Study on Modeling Shared Memory Access Effects During Performance Analysis of HW/SW Systems,* In Proceedings of the Sixth Int. Workshop on Hardware/Software Codesign, 1998.

[Ludw96] Ludwig, S.H.M., : *The Design of a Coprocessor Board Using Xilinx's XC6200 FPGA – An Experience Report*, Lecture Note in Computer Science : Field Programmable Logic FPL'96, 1996, Pages 77-86.

[Mich94] De Micheli, G.: *Computer-Aided Hardware-Software Codesign.* IEEE Micro, Vol. 14, No. 4, Sep. 1994, pages 10-16,

[NiGu97] Nisbet, S., Guccione S.A.: *The XC6200DS Development System,* In Proceedings Int. Workshop on Field-programmable Logic and Applications, 1997.

[POLIS] http://www-cad.eecs.berkeley.edu/Respep/Research/hsc/abstract.html.

[ShCh97] Shin, Y., Choi, K.: *Enforcing Schedulability of Multi-Task Systems by Hardware-Software CoDesign,* In Proceedings of the Fifth Int. Workshop on Hardware/Software Codesign, 1997.

[SKH97] Suzuki, F., Koizumi, H., Hiramine, M.: *A HW/SW Co-design Environment for Multi-media Equipments Development using Inverse Problem,* In Proceedings of the Fifth Int. Workshop on Hardware/Software Codesign, 1997.

[TBT97] Teich, J., Blickle, T., Thiele, L.: *An Evolutionary Approach to System-Level Synthesis,* In Proceedings of the Fifth Int. Workshop on Hardware/Software Codesign, 1997.

[Vahi97] Vahid, F.: *Modifying Min-Cut for Hardware and Software Functional Partitioning,* In Proceedings of the Fifth Int. Workshop on Hardware/Software Codesign, 1997.

[Varh97] Varhol, P.: *Adapting Java for Embedded Systems,* Computer design October. 1997.

[VaTa97] Vahid, F., Tauro, L.: *An Object-Oriented Communication Library for Hardware-Software CoDesign,* In Proceedings of the Fifth Int. Workshop on Hardware/Software Codesign, 1997.

[VCC97] *H.O.T. Works User's Guide*, by Virtual Computer Corporation, 1997.

[Xilinx] http://www.xilinx.com.