UNIVERSITÉ DE MONTRÉAL

MÉTHODES DE RAFFINEMENT DES COMMUNICATIONS POUR PASSER D'UNE PLATE-FORME SYSTEMC À UN SYSTÈME REPROGRAMMABLE

AHMED FAIZ
DÉPARTMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
AOÛT 2007

Your file Votre référence
ISBN: 978-0-494-53901-9
Our file Notre référence
ISBN: 978-0-494-53901-9

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

MÉTHODES DE RAFFINEMENT DES COMMUNICATIONS POUR PASSER D'UNE PLATE-FORME SYSTEMC À UN SYSTÈME REPROGRAMMABLE

présenté par : FAIZ Ahmed
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

M. LANGLOIS Pierre, J.M., Ph.D., président
M. BOIS Guy, Ph.D., membre et directeur de recherche
M. ABOULHAMID El Mostapha, Ph.D., membre

Remerciements

Je tiens d'abord à remercier chaleureusement mon directeur de recherche, M. Guy Bois, pour sa compréhension, son support technique et financier, et ses conseils précieux durant toute la période de ma maîtrise.

Je remercie les membres du jury, M. Mostapha Aboulhamid et M. Pierre Langlois, pour avoir pris de leur temps afin d'assister à la soutenance de mes travaux.

Je remercie également mes collègues du laboratoire Circus de l'École Polytechnique de Montréal pour leur aide et leur fraternité.

Je profite de cette occasion pour remercier mon épouse Hassnaa pour son aide morale et pour sa patience.

Résumé

Le temps de commercialisation et la productivité sont parmi les critères les plus importants de l'industrie des systèmes numériques et des systèmes embarqués. Auparavant, ces systèmes étaient totalement conçus en matériel. Or, l'introduction des processeurs embarqués a changé la donne, et les systèmes embarqués sont présentement composés dans la majorité des cas d'une partie logicielle et d'une autre matérielle. Ils sont donc devenus des systèmes sur puce, ou *system-on-chips* (SoC).

Cette coexistence entre le logiciel et le matériel nécessite la mise à jour des méthodologies de conception et deux solutions sont possibles : 1) séparer les spécifications du logiciel et du matériel dès le départ, 2) travailler sur une seule spécification et raffiner le système au fur et à mesure jusqu'à ce qu'il faille diviser le système en deux partitions : une logicielle et une matérielle.

La nature de l'application est un facteur important dans les décisions de partitionnement en logiciel et en matériel d'un architecte système. Le logiciel, par sa flexibilité, convient parfaitement aux parties de l'application nécessitant du contrôle informatique, tandis que le matériel (généralement plus ardu à implémenter que le logiciel) convient mieux aux parties de l'application orientées flot de données.

Plusieurs groupes de recherche et sociétés dans le domaine des SoC travaillent sur la mise en œuvre de langages de programmation et de plates-formes afin d'augmenter le niveau d'abstraction des spécifications lors de la phase de conception, et de pouvoir travailler sur une spécification commune au logiciel et au matériel. Parmi ces langages, nous nous intéressons particulièrement à SystemC [2] qui sera présenté au deuxième chapitre.

L'utilisation des langages de haut niveau tels que le C ou le C++ pour concevoir des applications destinées à l'implantation sur puce amène des questions telles que le choix

du partitionnement logiciel/matériel, le passage d'une spécification haut niveau vers l'implémentation physique et la synthèse des communications entre le logiciel et le matériel.

Notre travail se base sur la plate-forme Space Codesign™, un environnement SystemC servant à la simulation et à la validation de systèmes décrits à haut niveau d'abstraction. Nous proposons d'abord, dans une méthodologie, le raffinement des communications logicielles/matérielles pour les systèmes orientés bus, pour ensuite effectuer le passage de façon semi automatique d'une spécification haut niveau vers une implémentation physique de type FPGA. Pour atteindre cet objectif, nous concevons en VHDL des adaptateurs pour assurer la communication matériel-matériel, matériel-logiciel et logiciel-matériel. Ces adaptateurs permettent la conversion du protocole de communication utilisé à haut niveau vers le protocole de communication de l'architecture cible au niveau implémentation.

Abstract

Time-to-market and productivity are amongst the most important criteria when speaking of digital and embedded systems. Originally, these systems were designed in hardware only. Nowadays, the introduction of embedded processors changed this situation, and embedded systems are most of the time composed of a software part and a hardware part, which have now become systems-on-chips.

This coexistence between software and hardware requires the adaptation of design methodologies. In this case, two solutions are possible: 1) to separate software and hardware specifications at the early stages of the design cycle, 2) to work out from a unique specification and to progressively refine it until it becomes necessary to divide the system into two partitions: software and hardware.

Also, the nature of the embedded application is an important factor in the of software/hardware partitioning decision. Because of its flexibility, software is appropriate for the parts of the application requiring control, while hardware (generally more difficult to implement than the software) is appropriate for the parts of the application processing enormous quantity of data.

In the most specified field of system-on-chips, several research groups and companies work on the implementation of platforms and new programming languages to increase the level of abstraction in the early design stages, One advantageous outcome of these implementations work led to use a common specification to describe both software and hardware. Among these languages is SystemC [2] on which we focused, which will be presented in the Second Chapter.

The use of high level languages such as C or C++ to design applications intended for on-chip implementations comes with many questions such as making decisions on software/hardware partitioning, the migration from a high level specification to a physical implementation and the synthesis of the communications between software and hardware. Starting with the SystemC-based Space Codesign™ platform we used for simulating and validating high level systems, our work consisted in proposing a bus-based, hardware/software communication refinement methodology, then in carrying out in by an automated technique the transfer from a high level specification to an FPGA physical implementation.

Table des matières

Liste des tableaux

Liste des figures

Liste des acronymes

AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
BCA	Bus Cycle Accurate
BRAM	Block RAM
DDR	Double Data Rate
DMA	Direct Memory Access
DSOCM	Data Side OCM
EDK	Embedded Development Kit
ESL	Electronic System Level
FIFO	First In First Out
FPGA	Field Programmable Gate Array
IP	Intellectual Property
IPIF	IP InterFace
ISS	Instruction Set Simulator
LUT	Look Up Table
OCM	On Chip Memory
OCP	Open Core Protocol
OPB	On Chip Periphal Bus
OSCI	Open SystemC Initiative
PIC	Programmable Interrupt Controller
PLB	Processor Local Bus
RAM	Random Access Memory
RTL	Register Transfer Level
RTOS	Real-Time Operating System
SDRAM	Synchronous Dynamic RAM
SHIP	SystemC High-Level Interface Protocol

SoC	System **O**n **C**hip
SRAM	Static **RAM**
TF	**T**imed **F**unctional
TLM	**T**ransactional **L**evel **M**odeling
UART	**U**niversal **A**synchronous **R**eceiver **T**ransmitter
UTF	**U**ntimed **F**unctional
VHDL	**V**ery **H**igh Speed Integrated Circuit **H**ardware **D**escription **L**anguage
XPS	**X**ilinx **P**latform **S**tudio
ZBT	**Z**ero-**B**us **T**urnaround

Liste des Annexes

Introduction

La complexité des circuits à concevoir, en particulier pour les SoC augmente rapidement avec l'augmentation de la densité d'intégration [1]. Les données à traiter au niveau RTL augmentent de manière exponentielle, d'où la nécessité d'élever le niveau d'abstraction lors de la conception de tels circuits.

Une récente étude [1] menée par la Société du marché Internationale Business Stratégies auprès des ingénieurs de conception des SoC, montre que le développement du logiciel prend une part de plus en plus importante dans l'effort global de conception des SoC. Il devient donc important de démarrer le développement du logiciel en même temps que celui du matériel.

Le TLM (Transaction Level Modeling) permet d'augmenter le niveau d'abstraction de la conception des SoC. Il consiste à modéliser uniquement les échanges de données entre les différents modules d'une architecture du système à concevoir. À ce niveau le système peut être divisé en partitions logicielles et /ou matérielles et simulé sur une plate-forme au sein de laquelle les différentes composantes du système échangent des données via des médiums de communication (e.g. de type bus).

L'objectif principal du TLM est de représenter le comportement global du système à concevoir, en mettant l'accent sur l'échange de données et en ne se souciant pas des détails au niveau signaux.

De plus en plus de compagnies dans le domaine de conception des systèmes numériques et des systèmes embarqués, développent des outils de conception [1], de vérification et même de synthèse pour la conception au niveau système. Toutefois, de nos jours il existe encore des réticences [1] vis-à-vis ces outils. Les raisons sont prétendument techniques, liées à la maturité de ces outils, ou encore dues à la résistance au changement et à la protection d'un savoir faire accumulé depuis plusieurs années.

Problématique

L'utilisation des langages de programmation et de plates-formes à haut niveau d'abstraction pour la conception d'un système destiné à l'implémentation sur une puce permettent au concepteur de valider son système et ses choix architecturaux de manière plus rapide que de travailler directement au niveau RTL. Contrairement à l'approche traditionnelle qui consistait à concevoir un prototype RTL et à démarrer ensuite la conception du logiciel, désormais les concepteurs peuvent travailler sur une spécification commune du logiciel et du matériel.

Une fois le système validé à haut niveau, il faut l'implémenter physiquement. Idéalement, le concepteur devra disposer d'outils qui lui permettent de passer automatiquement d'une spécification de haut niveau vers la synthèse de son application sur l'architecture cible. Or, plusieurs plates-formes de conception à haut niveau utilisent des spécifications basées sur des langages (e.g. SystemC TLM et System Verilog) qui ne peuvent pas être synthétisées totalement.

La plate-forme Space Codesign™ (abrégé par Space) est une plate-forme basée sur SystemC. Elle est développée au sein du laboratoire de codesign de l'École Polytechnique de Montréal. Elle permet la conception et la simulation des systèmes à haut niveau, fournit plusieurs IP standards en SystemC afin d'augmenter la réutilisation, fournit le support d'un RTOS au niveau du logiciel d'un simulateur de niveau jeu d'instructions (ISS) pour simuler le logiciel, et fournit un modèle TLM pour une architecture de bus standard à savoir le bus OPB.

Toutefois, pour étendre la méthodologie de conception de Space jusqu'à l'implémentation physique, il faut pouvoir assurer une génération (semi-automatique) des composantes suivantes : exécutable du logiciel sur le processeur embarqué, architecture cible au niveau de la puce, modules matériels de l'application, et interfaces de communication entre le logiciel et le matériel.

Objectif du travail

Nous travail consiste à proposer une méthode pour passer d'une spécification basée sur le niveau TLM vers l'implémentation physique sur puce en utilisant l'architecture cible. Cet objectif se divise à son tour en 3 objectifs secondaires :

1) Conception d'un adaptateur d'un module matériel lui permettant de connecter à un bus d'une architecture standard.
2) Conception d'un adaptateur pour assurer les communications entre le matériel et le logiciel.
3) Génération du code exécutable pour la partition logicielle.

Méthodologie

Nous utiliserons deux technologies pour réaliser nos objectifs :

1) La technologie Space qui inclut un niveau TLM et un mécanisme de communication par échanges de messages, à partir desquels nous procéderons au raffinement.
2) La technologie Virtex2 Pro de Xilinx qui inclut le processeur MicroBlaze et le standard de bus OPB de CoreConnect.

Par conséquent, nous procéderons à la conception et à l'implémentation : 1) d'un adaptateur pour un module matériel et 2) d'un adaptateur pour la communication entre partitions. Ceci permettra d'une part d'envoyer et recevoir des données à travers le bus OPB en utilisant le protocole de communication et ainsi d'assurer les différents types de communication requis : 1) communication logiciel/logiciel (e.g. deux partitions logicielles sur un même processeur), 2) logiciel/matériel (e.g. entre un processeur et un coprocesseur) et 3) matériel/matériel (e.g. entre deux coprocesseurs distincts)[1]. Finalement, nous procéderons à la génération du logiciel qui sera compilé et exécuté à bas niveau par le processeur embarqué, avec le minimum voire aucune modification par rapport au logiciel exécuté sur l'ISS du processeur au niveau TLM. Notez que la

[1] La communication matériel/matériel sur un même coprocesseur est pour l'instant impossible dans Space, puisque l'on suppose un processus par coprocesseur.

génération matérielle à partir d'une description comportementale ne fait pas l'objet de ce travail. Nous supposerons que cette génération se fait manuellement ou se fait à l'aide d'un outil d'un tiers. C'est d'ailleurs pour cette raison que nous utiliserons dans ce travail le terme *génération semi-automatique* plutôt que *génération automatique*.

Contribution

Trois niveaux de spécification d'un système sont actuellement possible dans Space. Ces niveaux vont de la vérification fonctionnelle jusqu'à la validation au cycle près (BCA de l'anglais *Bus Cycle Accurate*) du système à concevoir. Notre travail permettra l'ajout d'un niveau d'abstraction qui est celui du RTL et d'une passerelle semi-automatique pour passer du niveau BCA au niveau RTL. Ce dernier étant beaucoup plus proche de l'implémentation que le niveau BCA, nous pourrons aussi valider la précision de notre modèle TLM (e.g. au niveau des latences prises en compte).

Enfin, notre travail permet de définir les bases d'un travail futur de passage complètement automatique d'une spécification TLM vers une implémentation physique et la possibilité d'utiliser des outils commerciaux capables de synthétiser du SystemC TLM, ou d'effectuer une conversion vers un langage HDL synthétisable.

Distribution des chapitres

Ce travail est organisé en quatre chapitres. Le chapitre 1 présente quelques concepts de la méthodologie de codesign, ainsi que la revue des travaux de raffinement des systèmes numériques conçus à haut niveau. Le chapitre 2 introduit brièvement SystemC, puis décrit les niveaux de raffinements et le protocole de communication de la plate-forme Space. Le chapitre 3 présente en détail l'implémentation à bas niveau de l'adaptateur d'un module matériel, l'implémentation de l'adaptateur qui gère les communications entre le matériel et le logiciel, et l'interface entre un module matériel et son adaptateur. Le chapitre 4 présente les résultats obtenus des latences de communications (permettant du même coup de valider l'exactitude du modèle TLM), des ressources matérielles

utilisées par les adaptateurs de communication, et un exemple de vérification du fonctionnement du protocole de communication de Space à bas niveau.

Chapitre 1 Revue du raffinement des communications pour systèmes sur puce

Un système embarqué peut être composé de partitions logicielles et/ou matérielles. La méthodologie de conception employée devrait assurer des mécanismes de communications entre les différentes composantes du système. Plusieurs modèles pour la communication sont possibles, et vont d'une communication point à point, jusqu'à une communication basée sur un réseau d'interconnexions (en anglais Network on chip). Ce chapitre présente l'approche de conception codesign. Nous nous intéressons aux étapes de génération d'un modèle du système au niveau RTL à partir d'un modèle conçu au niveau système. Plusieurs mécanismes de communication entre les parties logicielle et matérielle d'un SoC seront présentés. Par conséquent, nous parcourons quelques travaux de recherche et leurs approches pour le raffinement d'un système conçu à haut niveau vers une implémentation physique, en particulier nous mettons l'accent sur les techniques utilisées pour générer les interfaces de communication.

1.1 Le modèle TLM

Le modèle TLM décrit un système numérique complexe à un haut niveau d'abstraction. Il permet aux concepteurs d'explorer différentes architectures du système avant d'atteindre l'étape détaillée de l'implémentation finale du système sur une puce [32]. La vérification fonctionnelle est effectuée par des bancs d'essais (*testbenches*) basés sur les transactions de données.

Une transaction réfère à un échange d'une donnée entre deux composantes du système. Cette donnée peut être représentée par un mot, une série de mots, ou par une structure plus complexe [32]. Le modèle TLM, contrairement à d'autres modèles (e.g le modèle RTL) ne s'intéresse pas aux détails du protocole qui a permet d'effectuer la transaction.

Dans un modèle TLM, les détails de la communication sont séparés des détails de calcul. Les communications sont modélisées par des canaux, et les transactions ont lieu par l'appel des fonctions offertes par l'interface du canal en question [42].

Le modèle TLM est rapide à réaliser et à valider puisqu'il est moins détaillé qu'une réalisation complète du système au niveau RTL. Une fois ce modèle est finalisé, il pourra être raffiné progressivement vers le niveau RTL en y ajoutant plus de détails à chaque niveau de raffinement.

Selon la littérature, les niveaux d'abstraction diffèrent légèrement les uns des autres, mais en général, le premier niveau d'un modèle TLM permet de valider la fonctionnalité du système sans aucun détail sur la notion de temps, le deuxième niveau intègre la notion de temps pour simuler les délais que peuvent avoir certaines composantes du système durant une transaction, et un troisième niveau est détaillé au cycle près.

La figure 1.1 présente différents niveaux de raffinement d'un modèle TML tel que défini dans [42].

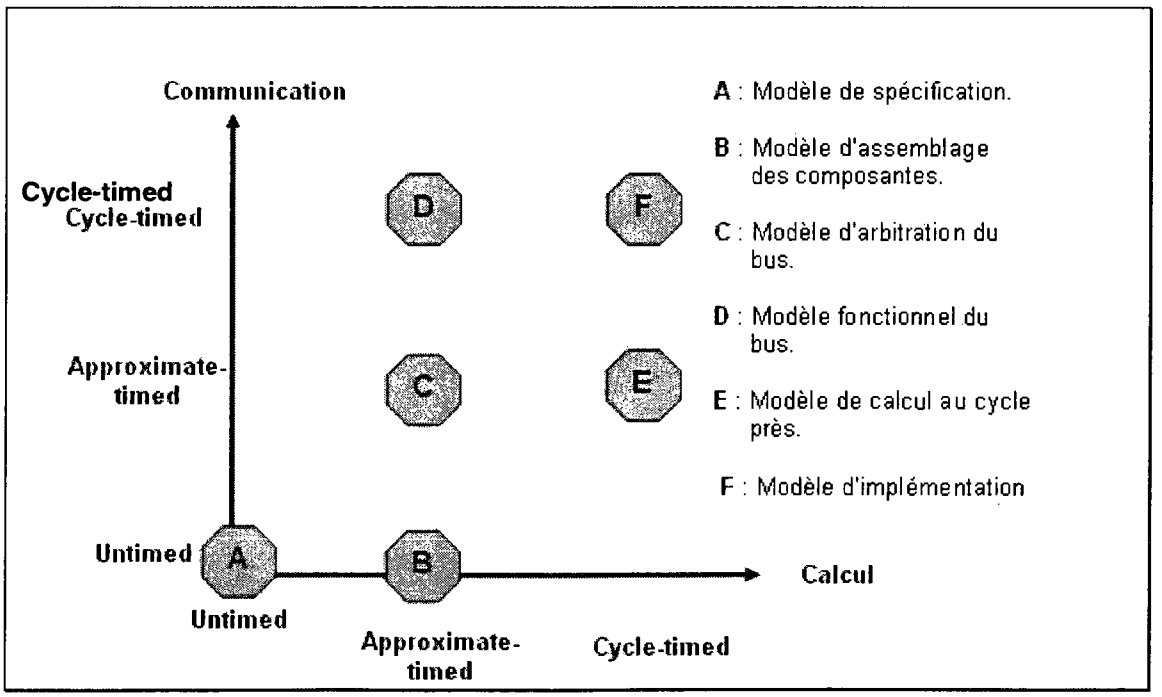

Figure 1. 1 Les différents niveaux de raffinement d'un modèle TLM

La figure 1.2 représente une requête de lecture d'une mémoire faite par un DMA dans un niveau d'abstraction TLM, elle pourra être vue comme une transaction de lecture en précisant simplement l'adresse de la mémoire à lire. Le DMA appelle une fonction ***read*** que le médium de communication implémente.

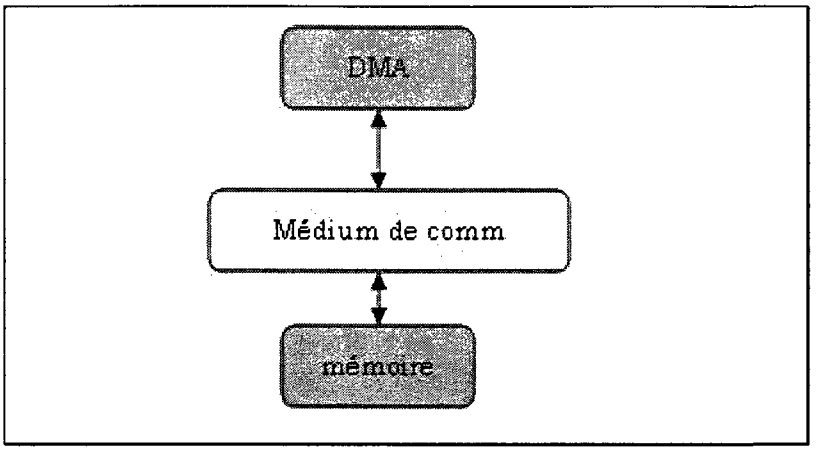

Figure 1. 2 Transaction de lecture par un DMA vers une mémoire au niveau TLM

1.2 Les systèmes sur puce

1.2.1 Évolution

Les dernières années ont connu de grandes avancées en ce qui concerne la vitesse, la puissance et la complexité des circuits intégrés tels que les mémoires (RAM), les microprocesseurs et autres. L'évolution de la densité d'intégration a rendu possible la mise en œuvre des systèmes sur puce (SoC). Ces derniers sont construits en utilisant des modèles pré-conçus de fonctions complexes connues sous le nom de IP utilisées dans diverses applications. Un SoC intègre donc sur une même puce une combinaison de composantes IP de différentes fonctions telles des microprocesseurs, des mémoires, des contrôleurs de vidéo, modem, des contrôleurs 2D, des fonctions de DSP, etc.

L'avancement de la technologie des FPGA a permet de construire des SoC basés sur cette technologie dont la flexibilité de re-programmation permet de reconfigurer la puce avec l'ensemble de IP sélectionnées pour obtenir la fonctionnalité désirée.
La présence d'un processeur et de différentes composantes IP sur la même puce a permis le développement d'applications dont une partie est implémenté en matériel, et l'autre partie en logiciel. Le pourcentage d'utilisation du logiciel dans de telles applications dépend la nature de l'application, et ce pourcentage frôle les 35 % dans certains cas [1]. Certains noyaux de systèmes d'exploitation en temps réel ont été portés pour rouler sur les processeurs embarqués des SoC, permettant d'avoir un système multitâches en logiciel. Ceci permet dont de concevoir et d'implémenter des applications à temps réel pour des systèmes embarqués.

1.2.2 Approche de conception par codesign

La méthode classique de conception des SoC, telle que présentée à la figure 1.3 [24], consiste à séparer le développement du logiciel et celui du matériel. Les partitions logicielles et matérielles sont décidées à l'avance, et des équipes de concepteurs travaillent séparément sur la spécification logicielle ou matérielle selon leurs spécialités.

Cette approche souffre des faiblesses suivantes

- Manque d'une spécification permettant une représentation unifiée du matériel et du logiciel, ce qui introduit des difficultés pour vérifier le système en entier.
- La complexité croissante des applications entraîne une augmentation considérable de la durée de développement.
- Possibles incompatibilités entre les frontières du logiciel et du matériel (interfaces).
- La configuration logicielle / matérielle choisie dès le début du flot de conception n'est pas toujours la plus optimale.
- Manque d'un flot de conception bien défini qui rend les spécifications (logicielle et matérielle) difficiles à réviser et peut augmenter le temps de mise en marché.

Pour pallier à ces faiblesses, de nouvelles méthodologies de conception sont recherchées. La conception au niveau système appelée conception conjointe logiciel/matériel (de l'anglais *codesign*) essaie de répondre à ce problème. Son but est de couvrir tout le cycle de conception de la spécification jusqu'au prototype de l'application à réaliser.

Le codesign propose une nouvelle méthodologie. Dans la littérature, cette dernière varie selon les besoins de l'application, mais tous les chercheurs s'entendent sur le principe du développement conjoint du logiciel et du matériel devenu essentiel.

La spécification du système à haut niveau est une des techniques utilisées par la méthodologie de codesign, elle permet au concepteur de spécifier les fonctionnalités du système dans un langage haut niveau tel que C, C++ ou SystemC. La simulation du système à haut niveau est plus rapide par rapport à une simulation au niveau RTL. Plusieurs variantes de la méthodologie de codesign intègrent le modèle TLM dés les premières phases de la conception.

La figure 1.4 [8] représente les différentes étapes de la conception conjointe d'un système.

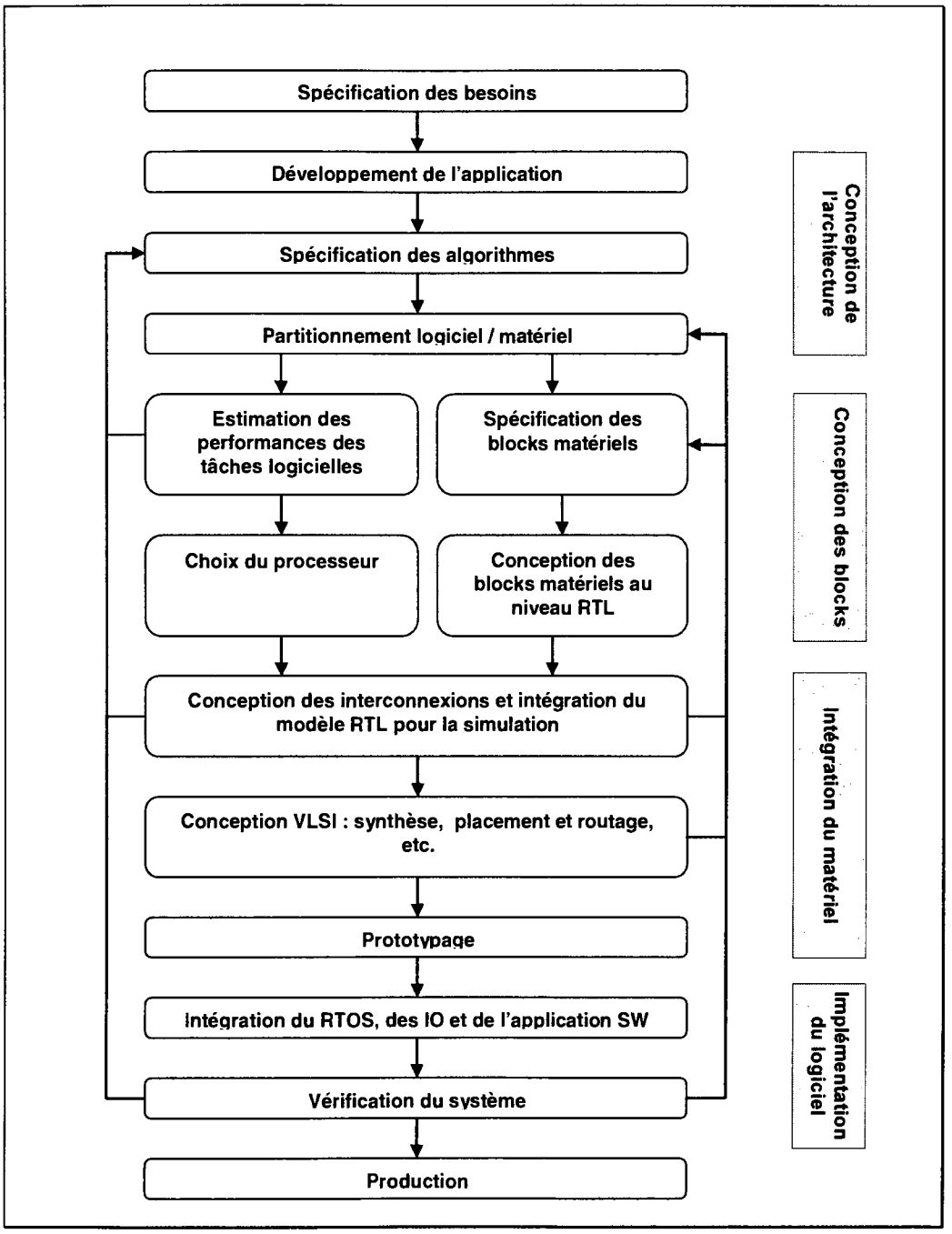

Figure 1. 3 Flot de conception classique de SoC

1.2.3 Langages de programmation à haut niveau et les SoC

Les langages de programmation utilisés durant le flot de conception peuvent être classifiés selon l'étape de conception à laquelle ils sont utilisés [24]. Une manière générale est de les classifier en langages de description du matériel et en langages de description du logiciel.

La majorité des langages de programmation ont été utilisé pour la description des systèmes, spécialement le langage C avec des extensions pour la gestion de la concurrence et du temps. HardwareC [19], HandelC [20], SpecC [21], et SystemC [2] (qui sera décrit plus en détail au chapitre 2) sont les langages les plus connus. Ils dérivent du C ou du C++ et ils sont utilisés pour la spécification au niveau système.

SystemC utilise le même modèle de calcul celui des langages VHDL et Verilog. Il est basé sur les événements discrets et sa force réside dans son support pour la modélisation des systèmes au niveau TLM, afin de faciliter l'exploration architectural et la validation d'un système à haut niveau. L'objectif n'est pas que SystemC remplace VHDL ou Verilog, mais plutôt de fournir un nouvel environnement pour la simulation au niveau système.

D'autre part, il existe également des langages synchrones tels que Esterel, Lustre, StateCharts, et Signal sont utilisés pour décrire des systèmes réactifs qui interagissent avec leurs environnements à temps réel. Le langage Esterel est utilisé pour la spécification et la validation d'un système [22,23].

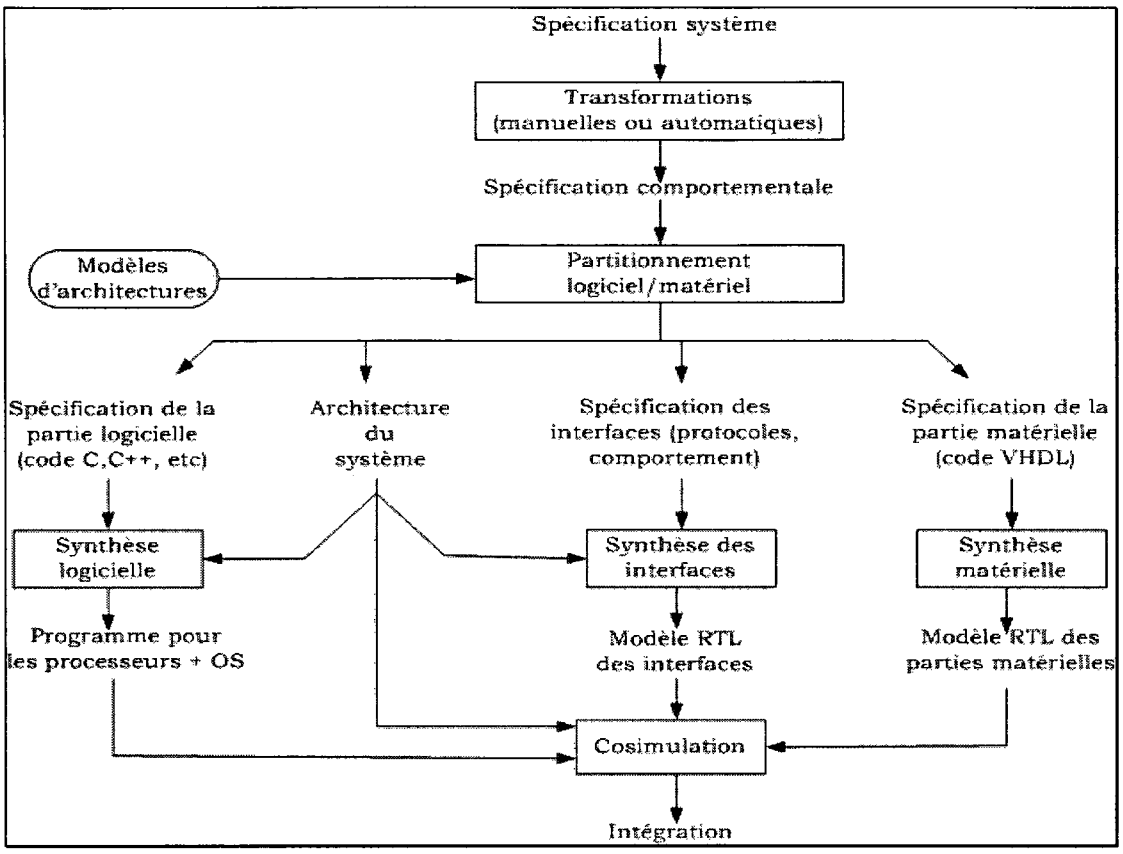

Figure 1. 4 Flot de conception de SoC au niveau système

1.3 Modèles de communication pour un SoC

Les mécanismes de communication dans un SoC dépendent des parties du système mises en jeu. On distingue la communication inter logiciel, la communication inter matériel, et la communication entre le logiciel et le matériel.

1.3.1 La communication inter logiciel

Dans [9], les modèles de communication entre les différentes tâches d'un système logiciel sont classés en deux catégories : la communication explicite dans laquelle le transfert des données entre les tâches est effectué par des méthodes du type ***send*** et ***receive***, et la communication implicite dans laquelle le transfert des données est effectué à travers un mécanisme de mémoire partagée.

La communication explicite entre les tâches est modélisée par un mécanisme de passage de messages. Ce modèle implique l'utilisation des mémoires de type queue ou FIFO, et la communication peut être bloquante ou non.

La communication implicite entre les tâches est modélisée par un mécanisme de mémoire partagée. Dans ce mode une partie de la mémoire, allouée pour les données, est visible à une ou plusieurs tâches. Un arbitrage est donc requis pour accéder à cette mémoire partagée. Les opérations de lecture et d'écriture vers cette mémoire devront être gérées de façon à assurer la cohérence des données. Les langages de programmation du logiciel embarqué sont munis d'extensions pour faciliter l'utilisation de la mémoire partagée. Par exemple, en langage C une variable dans une mémoire partagée se verra attribuée le mot clé ***volatile.***

L'utilisation de la mémoire partagée, généralement utilisée pour assurer l'échange de données dans un système muti-processeur [15], nécessite d'autres mécanismes pour assurer la cohérence des données. Par exemple, un mécanisme de synchronisation est utilisé dans le cas d'accès simultanés à la mémoire partagée par plusieurs tâches.

Toutefois, les architectures multi-processeurs de SoC dépassent le cadre de notre travail et ne seront pas abordées.

1.3.2 La communication inter matériel

Le choix d'un mécanisme de communication pour une partition matérielle a un impact sur la performance, et sur le coût de la puce. Les modèles de communication par passage de messages ou par mémoire partagée utilisés pour le logiciel, peuvent aussi être modélisés pour les communications entre les différentes composantes matérielles du système en utilisant un des trois types de communication suivant : ..par bus, point à point et par réseaux de connexions (de l'anglais *network on chip*). À l'exception des réseaux de connexions qui dépassent le cadre de ce travail, dans ce qui suit nous présentons les deux premiers types de communication.

Communication par bus

Le bus permet la communication entre plusieurs blocs matériels du système, ces blocs peuvent être des maîtres (tels que les processeurs) ou des esclaves (telles que les mémoires). Un bus est en général muni d'un mécanisme d'arbitrage basé sur différentes politiques telles que l'arbitrage en mode FIFO et l'arbitrage basé sur les priorités.

La conception d'un bus utilise plusieurs stratégies afin de répondre aux objectifs générales de l'architecture dans laquelle le bus sera utilisé. Les décisions de conception du bus sont basées sur trois paramètres [9]:

- Largeur du bus et fréquence: ces paramètres déterminent la fréquence de transfert de données à travers le bus, et ont un impact sur le coût, la consommation d'énergie, et sur les requis de la technologie.
- Arbitrage: la politique d'arbitrage affecte directement le taux d'utilisation du bus et la latence de chaque maîtres du bus. *L'arbitrage de type FIFO ou Round-robin* en anglais permet aux maîtres d'avoir des chances égales d'accéder au bus, mais présente l'inconvénient de faire attendre une requête de plus haute priorité. *L'arbitrage basé sur la priorité* donne accès au bus au maître qui a la plus haute priorité.
- Type de transfert : un bus simple implémente juste quelques types de transfert tels que des lectures et des écritures de données pouvant aller de 8 à 32 bits de taille. Par contre, des bus plus complexes implémentent d'autres types de transfert plus avancés tels que :

  ➢ Transfert de blocs fixes: en général des blocs de données dont la taille est une puissance de deux. Ce type de transfert est utilisé par exemple pour les mémoires caches.

  ➢ Transfert de type différé : dans ce mode de transfert (en anglais *split transaction*), l'accès au bus peut passer à un autre maître en attendant que la requête du maître qui avait le bus soit complétée. Ce type de transfert est particulièrement intéressant dans le cas de mémoires lentes.

➢ Transfert atomique : ce type de transfert est utilisé dans le cas ou plusieurs maîtres accèdent à une mémoire partagée. Il renforce la politique d'arbitrage utilisée par *un mécanisme de réservation du bus* (en anglais *bus lock*). Un maître peut se servir de ce mécanisme afin de transférer plusieurs données successivement sans avoir à redemander le bus fréquemment, c'est ce qu'on appelle le transfert en rafale (en anglais *burst*).

La connexion point à point

La connexion point à point (aussi appelée *directe*) entre les blocs matériels réduit les coûts et la latence de communication. Elle permet l'envoi direct des données d'un bloc à un autre. Ce type de communication se trouve rapidement limité dans le cas de composantes devant communiquer avec plusieurs composantes, de même il ne permet pas la réutilisation de l'architecture de base, et reste spécifique au type de l'application.

Dans un système de type un producteur et plusieurs consommateurs, le producteur envoi la donnée qui est acheminée vers la FIFO de réception du consommateur concerné. Un mécanisme de décodage d'adresses pourra être utilisé afin de choisir la bonne FIFO parmi celles des blocs consommateurs.

Ce mécanisme permet la lecture et l'écriture bloquantes de manière implicite, le bloc effectuant l'écriture bloque si la FIFO de réception est pleine, le bloc effectuant la lecture bloque si sa FIFO de réception est vide. Il est aussi possible de réaliser ces opérations de manière non bloquante, chaque bloc devra vérifier de son côté si la FIFO est pleine, ou s'elle est vide.

1.3.3 La communication entre le logiciel et le matériel

La communication entre le logiciel et le matériel est une communication entre le processeur qui exécute le logiciel et le reste de la plateforme matériel. Pour les SoC basés sur une architecture de bus, le processeur communique avec les autres composantes matérielles à travers le bus commun. Il peut y avoir une communication

directe entre le processeur et une composante du système et ce à travers une liaison dédiée basée en générale sur le mécanisme de FIFO telle que la liaison FSL [25] sur un processeur Microblaze [26].

Pour assurer le transfert des données d'un bloc matériel vers système logiciel multitâches, le mécanisme d'interruption est souvent utilisé pour éviter l'attente active (de l'anglais *polling*) [3].

1.4 Éléments du raffinement de SoC

La figure 1.4 présente le flot de conception d'un SoC au niveau système. Nous nous intéressons dans cette section aux étapes de choix de l'architecture du système et aux différentes synthèses effectuées afin de transformer les spécifications vers un niveau plus bas, généralement le niveau RTL.

1.4.1 L'architecture cible

L'architecture cible définit le support du système [8]. Elle peut être déduite après partitionnement ou imposée avant le partitionnement. Pour cet architecture, il faut déterminer le type de processeur (général, DSP, spécifique) qui convient le mieux à l'application. Il faut aussi décider de la structure de l'ensemble : nombre de bus, mémoires, entrées/sorties, etc.

Pour faire le lien avec l'architecture cible, on utilise en général l'approche dite *plateforme*, dans laquelle la correspondance est faite entre l'architecture à haut niveau et une architecture standard cible constituée en général d'un standard de bus et de ses périphériques principales. La figure 1.5 représente une architecture type d'un bus pour un SoC.

Lors du raffinement de l'approche plate-forme, les blocs du système partitionnés en logiciel et en matériel devront être respectivement compilés et synthétisés. Le bus de l'architecture choisie assure la communication entre les blocs. Des interfaces de communication entre le logiciel et le matériel peuvent s'avérer nécessaires dans le cas de

protocoles spécifiques. Et enfin, une couche d'adaptateurs est aussi nécessaire afin de réduire la complexité de connexion des blocs matériels du système sur le bus.

Les architectures standards de bus utilisées par les SoC telles que CoreConnect de IBM [10], Avalon de Altera [12], et AMBA de ARM [11], peuvent représenter une solution pour une approche de raffinement plateforme. Elles fournissent aussi des solutions partielles à la génération des interfaces de communication entre les différentes composantes de la plateforme. Par exemple, les librairies de IP fournies avec les outils de la compagnie Xilinx incluent une composante appelée IPIF [29]. D'une part, cette dernière permet de normaliser l'interface de connexion de chaque composante du système (uart, timer, mémoire) sur le bus (OPB ou PLB) et d'autre part elle facilite l'ajout de blocs matériels par l'utilisateur en lui fournissant l'interface de connexion sur le bus en question.

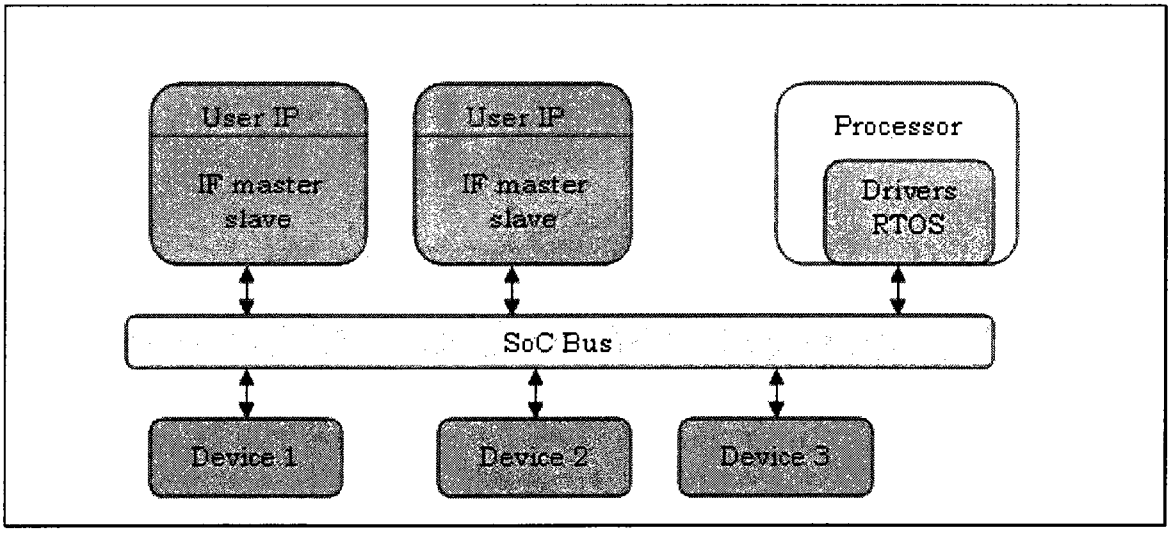

Figure 1. 5 Architecture type d'un bus pour un SoC

1.4.2 Les différentes synthèses

L'étape de synthèse dans un flot de conception de SoC (figure 1.4) est effectuée après les décisions de partitionnement. Elle comporte trois phases de synthèse: logicielle, matérielle et communications.

Synthèse logicielle

Elle consiste à générer le code binaire qui sera exécuté par le processeur de l'architecture cible. L'idéal est d'utiliser le même code source que celui utilisé à haut niveau dans lequel la partie logicielle du système est généralement validée sur un émulateur de processeur. En effet, des compagnies telles que IBM et ARM ont développé des ISS pour leurs processeurs embarqués respectifs (respectivement PowerPC et ARM).

Également, un système d'exploitation temps réel est généralement associé avec la partie logicielle dans le cas ou plusieurs tâches doivent être exécutés sur le même processeur.

Synthèse matérielle

Elle consiste à transformer la spécification de la partie matérielle du système en un circuit électronique. C'est aussi la possibilité de réutiliser des fonctions existantes (IP). Cette synthèse débute habituellement avec une spécification au niveau RTL décrite dans un langage HDL tel VHDL ou Verilog. Elle peut aussi débuter à un plus haut niveau d'abstraction, c'est-à-dire au niveau comportemental. Selon le langage utilisé à haut niveau (e.g. SystemC, C/C++), plusieurs cas de figures sont possibles pour le passage de la spécification à haut niveau vers le niveau RTL ou vers le niveau portes logiques:

- Passage du haut niveau vers le niveau des portes logiques, se traduisant par la génération d'une liste d'interconnexions (en anglais *netlist*).
- Translation de la spécification haut niveau vers une spécification au niveau RTL. Des exemples d'outils commerciaux sont Cynthesizer de Fortes Design et Catapult de Mentor Graphics.

- Translation manuelle de la spécification du langage haut niveau vers un langage HDL synthétisable. À priori, on peut penser que cette manière engendre un double effort durant le cycle de développement, mais le facteur de facilité de modélisation à haut niveau dans un langage tel que C ou SystemC est important.

Synthèse des communications

La synthèse des communications est la réalisation des interfaces de communication entre les différentes composantes du système. Nous avons mentionné que le choix de l'architecture cible impose un certain protocole de communication auquel les blocs matériels de l'application, ainsi que les tâches logicielles, devront être adaptés. En considérant les différents types de communication, quatre possibilités doivent être considérées:

- La communication entre les tâches du logiciel dans le cas d'un système multitâches.
- La communication entre les blocs matériels.
- La communication du logiciel vers le matériel.
- La communication du matériel vers le logiciel.

1.5 Revue des travaux sur le raffinement

La conception des SoC au niveau système est une approche récente, plusieurs groupes de recherche et compagnies du domaine EDA s'y intéressent. La majeure partie des travaux de recherche se limite à la validation du système à haut niveau [6, 13, 14, 27, 28] ou traite une partie de la problématique à savoir comment effectuer le raffinement final qui produira une spécification synthétisable à partir des spécifications haut niveau. Les compagnies quand à elles proposent des outils pour automatiser le flot de conception au complet en partant d'une spécification de haut niveau jusqu'à l'implémentation du système sur puce.

Pour des systèmes monoprocesseurs, les étapes d'exploration et de validation diffèrent principalement au niveau des termes utilisés pour décrire les étapes de raffinement et

des langages de programmation utilisés pour la spécification du système (SystemC, System Verilog, Esterelle, C/C++, SpecC, etc.).

On distingue principalement deux approches pour passer d'un modèle niveau système à un prototype au niveau RTL synthétisable. La première approche [3, 4, 7, 16, 17, 18] utilise des adaptateurs pour les blocs matériels pour des fins de compatibilité avec le protocole de communication de l'architecture cible à bas niveau et utilise une interface matériel entre le processeur embarqué et le reste de l'architecture cible afin de gérer les communications entre la partie logicielle et la partie matérielle. Cette interface que l'on peut nommer adaptateur de processeur peut être directement connectée au processeur via des liaisons rapides ou à travers le bus de l'architecture cible.

Dans la deuxième approche [5, 30], la partie logicielle du système est exécutée par le processeur de l'architecture cible, tandis que la partie matérielle est divisé en plusieurs fonctionnalités que chacune est implémentée comme étant un coprocesseur directement connecté au processeur.

Dans ce qui suit nous examinons plus en détail un exemple de chacune des deux approches.

1.5.1 Raffinement par l'utilisation d'adaptateurs de communication

Dans [3] on propose une méthodologie de raffinement des communications d'un modèle TLM vers un modèle RTL. Dans cette méthodologie, le protocole de communication, appelé SHIP (de l'anglais *SystemC High-Level Interface Protocol*), est basé sur SystemC et défini un canal de communication point à point pour le passage des messages entre les entités du système. Une entité peut être soit maître, soit esclave. La communication est définie en termes de fonctions *send/request* pour un maître, et *recv/reply* pour un esclave. La figure 1.6 présente le modèle de communication à haut niveau tel que présenté dans [3]. Le flot de conception proposé dans [3] et représenté à la figure 1.7. Il est composé de plusieurs niveaux d'abstractions pour obtenir à la dernière étape un modèle raffiné au niveau RTL synthétisable. Le flot en question est considéré le

point d'entrée pour la génération d'un prototype du système qui pourra être implémenté sur une puce. Les étapes de raffinement proposées se résument à :

- Validation du système au niveau fonctionnel en utilisant l'architecture basée sur un canal SHIP (figure 1.6).
- Remplacement du canal SHIP par un mécanisme de communication précis au niveau cycle appelé CASM (communication architecture simulation model en anglais).
- Partitionnement du système en logiciel et matériel.
- Connexion du module en logiciel au CASM par un mécanisme appelé *HwSwChannel* afin d'assurer la communication entre le matériel et le logiciel.
- Raffinement des ports SHIP d'un module matériel vers un niveau d'abstraction plus bas niveau, basé sur le protocole OCP [33].

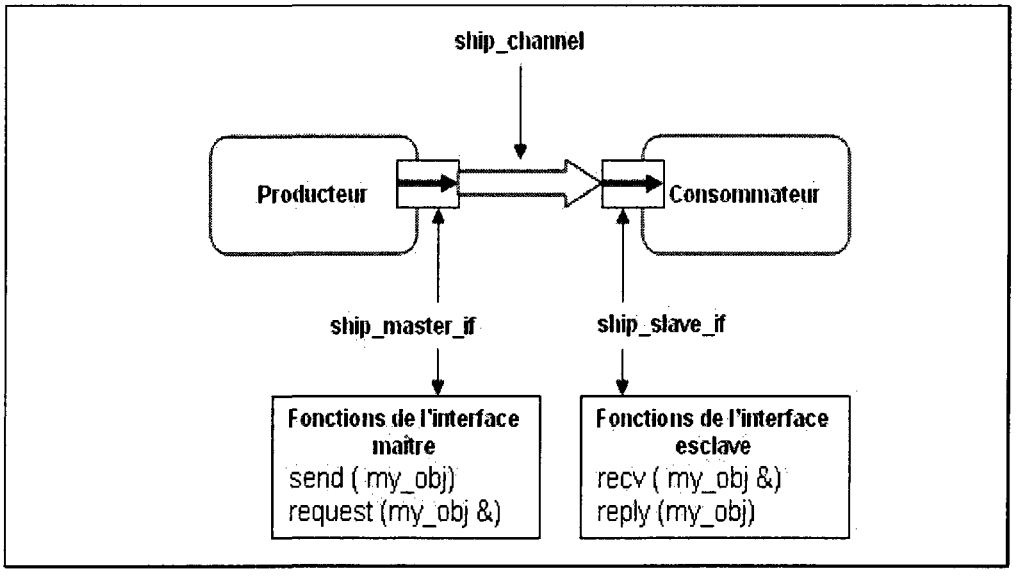

Figure 1. 6 Canal, ports et interfaces du modèle SHIP

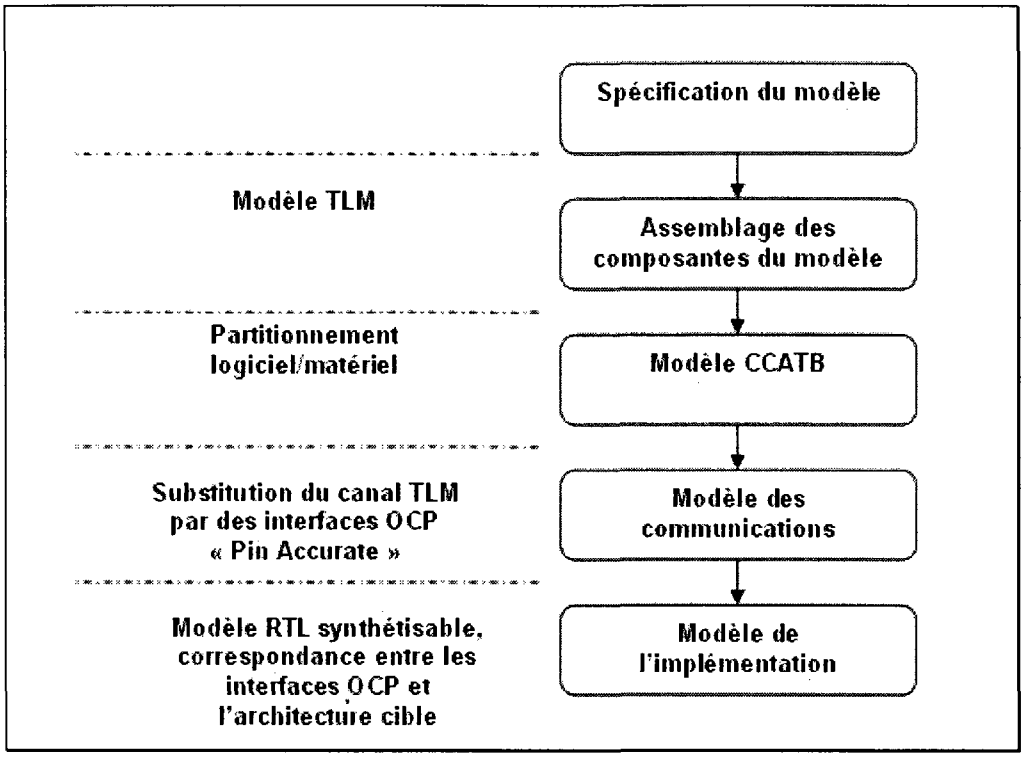

Figure 1. 7 Flot de conception proposé dans [3]

Un exemple de système raffiné est représenté à la figure 1.8

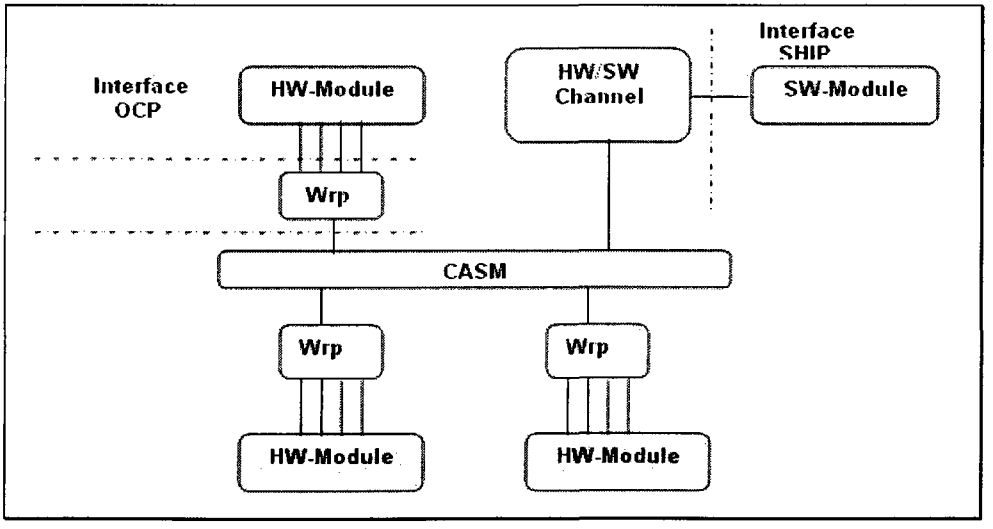

Figure 1. 8 Le modèle raffiné proposé dans [3]

D'autre part, les phases de la synthèse proposées sont :

- Choix de l'architecture de communication: bus ou réseau de connexion sur puce.
- Connexion d'un module matériel à l'architecture de communication par un mécanisme de *wrapper appelé Accessor* (figure 1.9), implémenté au niveau RTL, qui effectue la correspondance entre le protocole OCP et le protocole de l'architecture de communication via un mécanisme appelé PSM (Protocol State Machine).

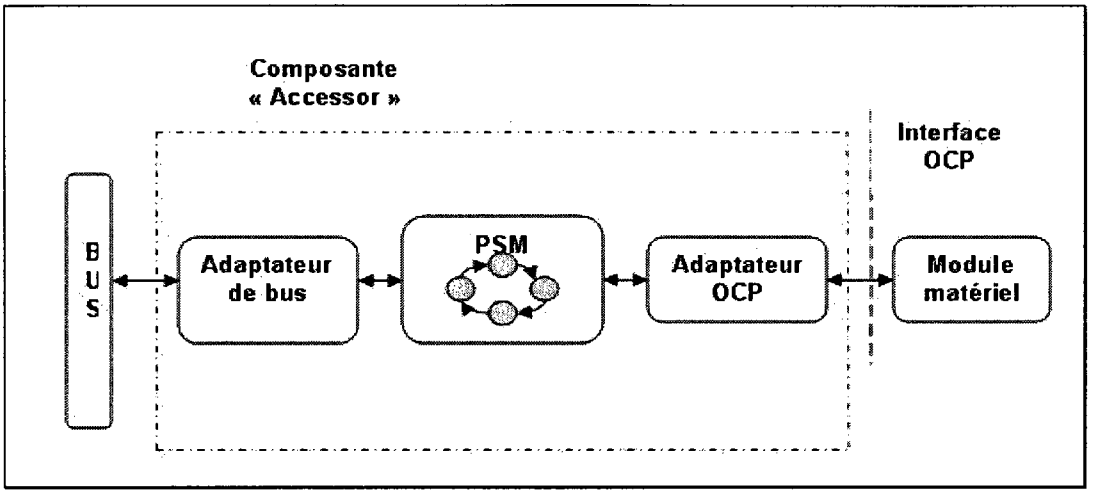

Figure 1. 9 Architecture d'un wrapper tel que proposé dans [3]

- Utilisation d'un adaptateur appelé *HW/SW-Controller* (figure 1.10) qui assure la communication entre le logiciel et le matériel. Ce contrôleur s'occupe de l'envoi des données du logiciel vers les autres composantes du système, il s'occupe aussi de la réception des données destinées au logiciel. Le contrôleur est connecté au processeur via les interfaces rapides disponibles pour certains processeurs telles que les interfaces DSOCM du PowerPC405 [34]. Un mécanisme d'interruption est utilisé entre le contrôleur et le processeur afin de traiter les requêtes destinées au logiciel. Enfin, le contrôleur est connecté sur l'architecture de communication (un bus en général) et dispose d'une FIFO pour y stocker les données en attendant d'avoir fini le traitement de la requête encours.

La figure 1.11 représente le modèle d'implémentation obtenu à la fin des étapes de synthèse.

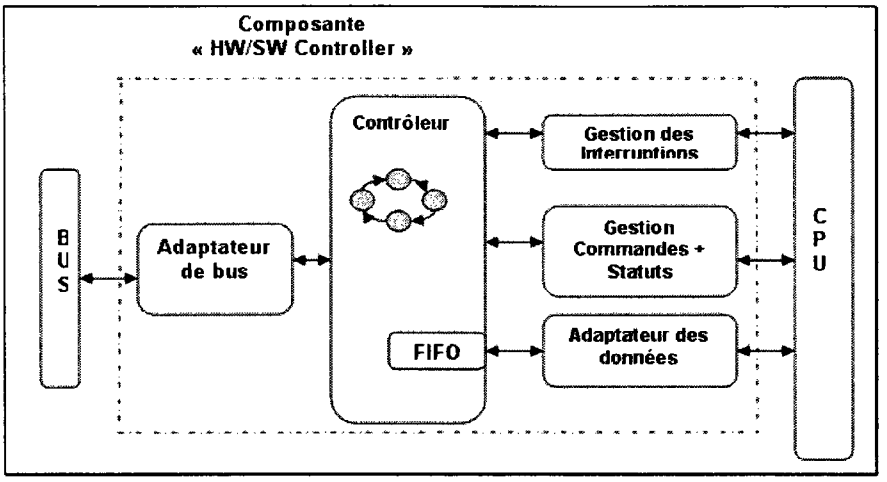

Figure 1. 10 Hw/Sw adaptateur tel que proposé par [3]

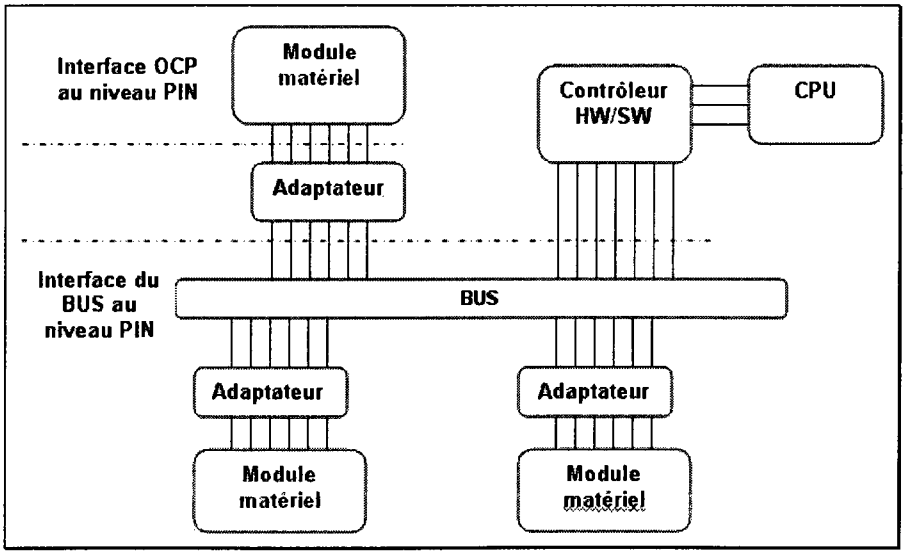

Figure 1. 11 Le modèle obtenu après synthèse tel que proposé par [3]

1.5.2 Raffinement par processeur / coprocesseur

La méthodologie proposée dans [5] part d'une spécification du système à haut niveau basée sur le langage C. Les fonctions en C qui exigent des calculs intenses sont implémentées en matériel et le reste du système est implémenté en logiciel.

Pour implémenter le système en logiciel et en matériel, l'approche processeur et coprocesseur est utilisée. Chaque fonction partitionnée en matériel sera implémentée en VHDL sur un coprocesseur. Les fonctions identifiées pour être implémentées en matériel sont converties en VHDL, et liées aux coprocesseurs.

Notons également une approche proposée dans [31], comparable à [5], qui consiste à spécifier le système en C/C++. Le système est ensuite partitionné en logiciel et en matériel. La partie matérielle contient les fonctions nécessitant une accélération matérielle. Chacune de ces fonctions est mappée sur un petit processeur spécialisé (figure 1.12) conçu en SystemC RTL synthétisable.

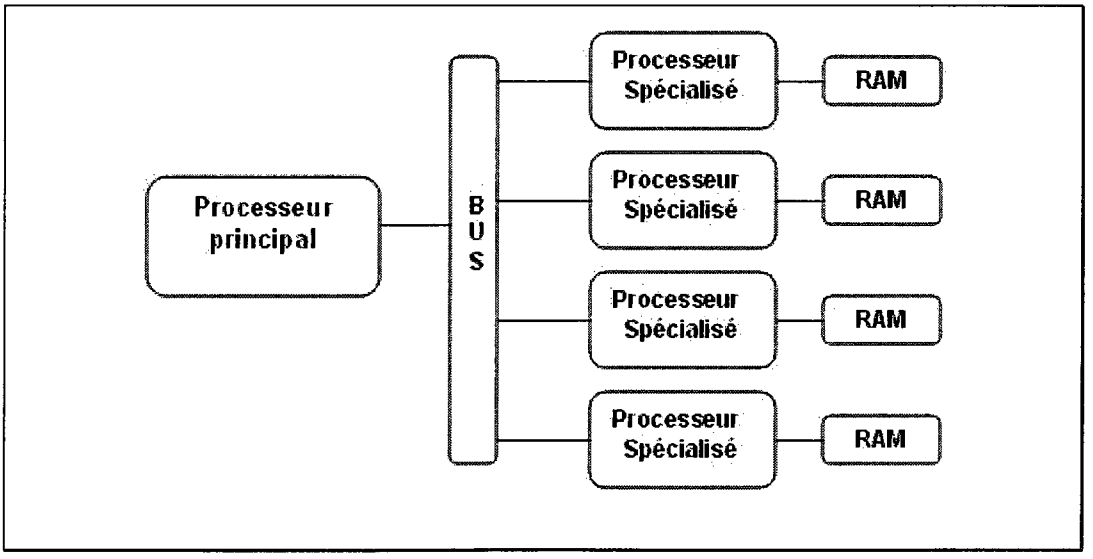

Figure 1. 12 Architecture de SoC, composée d'un processeur principal et des processeurs spécialisés

Chapitre 2 La plateforme Space

La plateforme Space est le résultat du travail de plusieurs étudiants, elle a été développée au laboratoire de codesign de l'École Polytechnique de Montréal. La plateforme est bâtie autour de SystemC et permet de concevoir des applications au niveau système. Dans une approche de haut vers le bas, la méthodologie de conception, basée sur Space, offre trois niveaux de raffinement progressif incluant un processus de partitionnement logiciel/matériel.

Ce chapitre est divisé en deux parties. La première partie propose une description non exhaustive de la librairie SystemC, alors que la deuxième présente une description de l'architecture et la philosophie de la plateforme Space. Nous mettons l'accent sur le protocole de communication entre les différentes composantes d'un système développé dans Space afin d'établir le lien avec notre travail à savoir le raffinement des communications de la plateforme vers un système reprogrammable de type FPGA.

2.1 La librairie SystemC

2.1.1 Introduction

SystemC est une librairie basée sur le langage de programmation orienté objet C++. On désigne généralement SystemC comme étant un langage pour la modélisation à haut niveau d'abstraction du comportement matériel des composantes d'un système. SystemC est souvent comparé aux langages HDL tels que VHDL et Verilog. L'extrait suivant tiré de [35] résume bien l'origine de SystemC :

« En 1989, Synopsys met son outil commercial Scenic dans le domaine libre, et crée la version 0.9 de SystemC. Une première contribution de Frontier Design donne lieu à la version 1.0, et une autre de CoWare aboutit en 2000 à la version 1.1, première version officielle de SystemC. L'OSCI (Open SystemC Initiative) est alors créé, rassemblant une

multitude de sociétés et laboratoires de recherche. Cette organisation est responsable de diffuser, promouvoir et rédiger les spécifications de SystemC

Les spécifications de SystemC ont été étendues en 2001 à la modélisation de systèmes abstraits (de très haut niveau, avant partitionnement matériel/logiciel), aboutissant à la version 2.0 ».

En 2005, SystemC a été standardisé IEEE 1666-2005. La version courante étant 2.2, on parle déjà de la version 3.0 qui devrait supporter la modélisation de logiciel embarqué.

L'objectif principal de SystemC consiste à élever le niveau d'abstraction durant la modélisation d'un système. Au lieu de démarrer la conception à un niveau bas tel que le niveau RTL, SystemC permet de valider la spécification au niveau transactionnel sans se soucier de tous les détails de l'implémentation finale.

Malgré qu'on parle de SystemC comme étant un langage pour la modélisation d'un système composé de matériel et/ou logiciel, à la base SystemC est beaucoup plus orienté pour la modélisation du comportement matériel. En effet, les versions actuelles et antérieures, de SystemC ne fournissent pas un support complet pour la modélisation du comportement logiciel.

2.1.2 Architecture générale de SystemC

La figure 2.1 présente l'architecture générale de la librairie SystemC. Elle est basée sur le langage C++ et fournit une collection de classes représentant des modèles en matériel, des mécanismes de communication, des éléments de synchronisation et le support de plusieurs types de données. Ces classes de base permettent la modélisation d'un système. La librairie inclut aussi un simulateur événementiel.

Dans ce qui suit, nous décrirons brièvement dans ce qui suit les éléments de chaque couche. Notez que la première couche (couche du haut sur la Figure 2.1) ne fait pas partie des éléments de base de la librairie SystemC. Elle n'est donc pas présentée dans ce qui suit, toutefois le lecteur peut référer à [2] pour plus de détails.

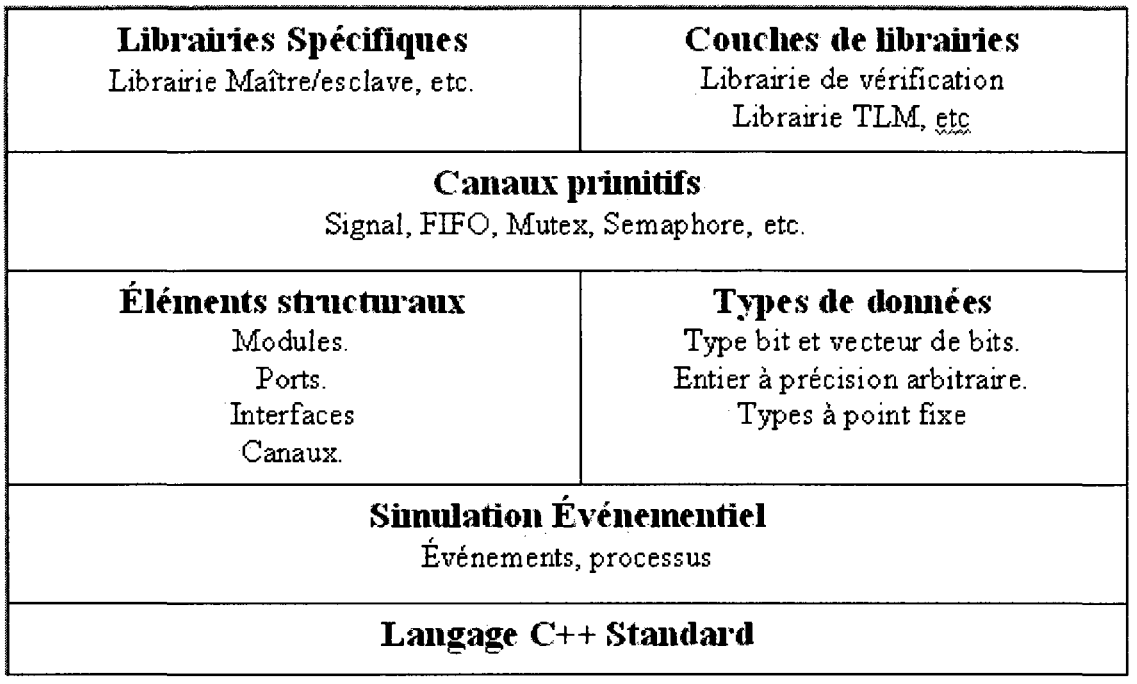

Figure 2. 1 Architecture de la librairie SystemC

Canaux primitifs

Les canaux primitifs sont des éléments de communication ou de synchronisation. Ils sont élémentaires et font partie de la construction de base d'un modèle de système basé sur SystemC. Selon le type du canal primitif, ce dernier se prête à une utilisation en matériel ou en logiciel. Par exemple, un signal est plus utilisé en matériel, et une correspondance directe peut être faite avec un signal décrit en VHDL. Par contre, les types mutex et sémaphore sont plus utilisés pour représenter un comportement en logiciel.

Éléments structuraux

Module

Un système est généralement divisé en plusieurs blocs, chaque bloc réalisant une fonctionnalité donnée. Un bloc est représenté en SystemC par un module qui est une classe décrivant le comportement d'un tel bloc. En général, un module contient :

- Un ou plusieurs ports à travers lesquels communique le module avec le reste du système.
- Un ou plusieurs processus **où** chaque processus décrit un comportement
- Des données et canaux primitifs internes afin d'assurer la communication et la synchronisation entre les processus du module.
- D'autres (sous) modules à travers une hiérarchie.

Interface

Une interface est un ensemble des fonctions (ou méthodes) qui peuvent être utilisées à travers un port. Une interface ne contient pas de code, ce n'est qu'une déclaration de fonctions. Les fonctions de ces interfaces sont implémentées dans les canaux.

Toutes les interfaces en SystemC dérivent de la classe de base ***sc_interface***

Canaux

Les canaux assurent la communication entre les différents modules d'un système à travers le port. Ils ne sont pas limités à une communication point à point comme c'est le cas d'un signal ou d'une FIFO. Les interfaces déclarent les fonctions disponibles pour la communication à travers un port. L'utilisation des interfaces permet donc d'implémenter différemment les fonctions de communications dans différents canaux.

On distingue les canaux primitifs qui sont atomiques et les canaux hiérarchiques avec lesquels il est possible de modéliser le comportement d'un médium de communication plus complexe tel qu'un bus dans un système sur puce.

Ports

En SystemC, les ports sont des objets qui permettent aux composantes d'un modèle de communiquer entre elles par l'intermédiaire des fonctions de l'interface implémentées par le canal.. Chaque port est spécifié avec le type d'interface auquel il correspond. La composante qui hérite de l'interface spécifiée par un port, devra donc implémenter les fonctions de l'interface correspondante au port en question.

Un port est spécifié par la déclaration **sc_port < interface_type>**.

Moteur de simulation événementiel

Processus

Un processus est l'unité de base pour représenter une fonctionnalité donnée. Un module SystemC peut avoir un ou plusieurs processus pour modéliser le comportement global du module en question. Contrairement aux langages de programmation séquentiels, SystemC permet la simulation d'un comportement concurrentiel grâce aux processus.

Les processus ont une liste de sensibilité sur un ou plusieurs paramètres tels que l'horloge ou d'autres signaux.

SystemC fournit trois types de processus : SC_METHOD, SC_THREAD et SC_CTHREAD. Le **SC_METHOD** s'exécute entièrement du début à la fin de son code suite à un changement dans sa liste de sensibilité et ne peut pas être suspendue explicitement par un appel à la fonction **wait().** Le **SC_THREAD** peut être interrompu à n'importe quel endroit de son code et quand son exécution reprend il peut restituer l'état juste avant la suspension et continuer l'exécution à partir de cet état. Finalement, le processus **SC_CTHREAD** est identique au SC_THREAD, mais sensible uniquement sur les fronts de l'horloge.

Événements

Un événement est un objet de la classe **sc_event**. Il représente en général une certaine condition sous laquelle l'exécution d'un processus s'arrête ou redémarre. Il maintient une liste de processus dont la liste de sensibilité contient l'événement en question. Le

changement d'une certaine condition est notifié à travers l'événement, ce qui permet de réordonnancer les processus sensibles à cet événement.
Un événement pourra être directement utilisé par un processus pour contrôler un autre processus en particulier si les deux processus en question ont besoin de synchroniser leurs exécutions sur une certaine condition.

Le non-déterminisme

Le non-déterminisme du comportement d'un système en SystemC lors de la simulation est un facteur important à prendre en considération [37]. En effet, l'ordre d'exécution des processus n'est pas spécifié particulièrement au démarrage ou au cours de la simulation quand plusieurs processus sont prêts simultanément.
Malgré cet aspect de non-déterminisme de SystemC, globalement un système se comporte de la même manière entre deux simulations effectuées sur une même machine et avec les mêmes entrées, puisque l'Ordonnanceur effectue le même choix du processus à exécuter entre deux simulations [36].

2.1.3 Les niveaux de raffinement de SystemC

SystemC offre plusieurs niveaux de raffinement pour le système à concevoir. Ces niveaux diffèrent principalement au niveau de la granularité des détails liés au fonctionnement du système. Ainsi d'un niveau à l'autre, des latences liées à la communication ou au calcul pourront être ajoutées à différentes composantes du système et en particulier au médium de communication. SystemC permet donc de passer d'une spécification purement transactionnelle (niveau TLM), à une spécification plus proche du niveau RTL (au niveau des signaux).
Les niveaux de raffinement supportés par SystemC sont comme suit [35] :

- **UTF (Untimed Functional) :** le modèle ne comporte aucune notion de durée d'exécution, mais seulement un ordre éventuel dans l'exécution des événements. Chaque requête s'exécute en un temps nul. Seul compte l'ordonnancement des événements.

- **TF (Time Functional)** Le modèle comporte des notions de durée d'exécution (e.g. temps d'exécution des processus et latence des communications).
- **BCA (Bus Cycle Accurate)** Ce niveau signifie que la modélisation des transactions sur l'interface est précise au cycle près. Un modèle BCA n'apporte aucune information sur les bits (signaux) de l'interface.
- **BA (Bit Accurate)** Ce niveau signifie que la modélisation des transactions sur l'interface est de niveau BCA et considère aussi sur les signaux de l'interface. La modélisation est donc précise au bit près. On désigne souvent ce niveau par le terme CABA (cycle accurate / bit accurate).
- **RTL (Register Transfert Level)** Chaque bit, chaque cycle et chaque registre du système sont modélisés.

2.1.4 Exemple d'une bascule avec SystemC

Pour illustrer la syntaxe de SystemC, nous présentons à la figure 2.2 un exemple simple d'une bascule D avec un signal de remise à zéro (*reset*) asynchrone.

Dans cet exemple, SC_MODULE est une macro qui déclare un module dont le nom est **diffa** et qui a trois ports d'entrées (clock, reset, et din) et un port de sortie (dout). Le module a un processus de type SC_METHOD qui est sensible au front montant de l'horloge *clock*, et sur le niveau du signal Reset.

```
#include "systemc.h"
SC_MODULE (dffa)
{
        sc_in<bool> clock , reset, din;
        sc_out<bool> dout;
        void do_ffa()
        {
                if (reset) {
                        dout = false;
                } else if (clock.event()) {
                        dout = din;
                }
        };
        SC_CTOR(dffa)
        {
                SC_METHOD (do_ffa);
                sensitive (reset);
                sensitive_pos (clock);
        }
};
```

Figure 2. 2 Exemple simple d'une bascule D en SystemC

2.2 La plateforme Space

Space est une plateforme bâtie autour de SystemC pour permettre la modélisation, la validation et l'exploration architecturale à haut niveau d'un système matériel et/ou logiciel. Elle est composée d'un ensemble de librairies représentant différentes composantes (bus, uart, timer, ram, iss, ou wrapper, etc.) à haut niveau, d'un ensemble d'outils d'aide au développement tel qu'une interface graphique et de logiciels nécessaires à la compilation et la production d'un exécutable.

La plate-forme Space bâtie sur SystemC permet à ce dernier d'ajouter 3 éléments importants:

- **Augmention de la réutilisation** Ceci permet à un utilisateur de la plateforme Space d'avoir une collection de composantes standards nommées IP qu'il peut réutiliser dans différentes applications. Ce concept existe présentement pour la conception SoC à bas niveau. Il permet au concepteur de se concentrer sur l'application en mettant à sa disposition une architecture et un ensemble de IPs.

- **Un protocole de communication générique** Les communications (bloquantes ou non bloquantes) entre les composantes d'un système conçu avec Space se basent sur le mécanisme de passage de message (message passing). Les composantes du système (modules et périphériques) sont identifiées chacune par un identificateur numérique qui est converti en une adresse lors du l'initiation d'une requête de lecture ou d'écriture.
- **Support avancé pour la partie logicielle** SystemC dans sa version actuelle est très limité quand à la modélisation de la partie logicielle d'un système composé de logiciel et de matériel. La plateforme Space apporte davantage de support pour la conception et la simulation de cette partie en intégrant un RTOS afin de modéliser des applications avec des contraintes de temps réel. Space fournit aussi un ISS pour un processeur donné (actuellement deux ISS sont fournis pour les processeurs ARM, et Microblaze) afin de simuler le comportement du logiciel.

Trois niveaux de raffinement sont possibles dans Space : UTF, TF et BCA. Le partitionnement d'un système en logiciel et/ou en matériel est encore manuel et intuitif. Le but à court terme est de l'automatiser ou du moins de fournir une aide aux prises de décisions grâce au profilage de la partie logicielle et à l'aide de métriques sur les communications entre le matériel et le logiciel.

2.2.1 Vue d'ensemble

Afin de modéliser un système à haut niveau dans Space, un utilisateur dispose d'un ensemble de librairies qui lui permettent de construire l'architecture de base de son système. L'utilisateur aura pour seule tâche la conception des modules spécifiques de son système. Une représentation globale d'un système typique conçu dans Space est présentée à la figure 2.3.

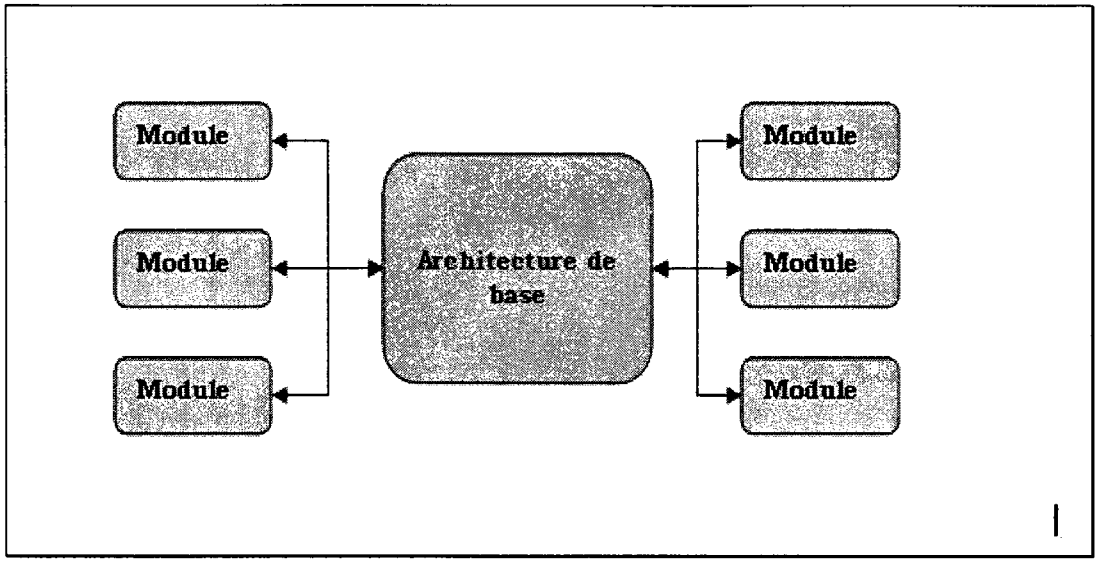

Figure 2. 3 Vue d'ensemble d'un système conçu avec Space

La figure 2.3 représente un système découpé en plusieurs modules qui communiquent entre eux via une architecture de base à laquelle ils sont attachés. L'architecture de base diffère d'un niveau de raffinement à un autre. Alors que seulement des notions temporelles sont ajoutées ou retirées des modules quand le niveau de raffinement change. Les modules peuvent être déplacés de la partition logicielle à la partition matérielle et vis-versa sans aucune modification à leur code. Dans ce qui suit nous décrirons plus en détail chacune de ces composantes.

Module

Un module dans Space, représente composante de base pour la conception d'un système, qui est en général divisé en plusieurs modules représentant chacun un comportement ou une fonctionnalité précise du système. La classe **SpaceBaseModule** est la classe de base dont tous les modules conçus par l'utilisateur héritent. Cette classe dérive elle-même de la classe **SC_MODULE** de SystemC. Un module peut avoir un ou plusieurs processus.

Selon le niveau de raffinement utilisé, le module peut être en matériel ou en logiciel. Tel que mentionné ci-haut, le passage d'une partition à une autre n'implique aucun changement quant au code du module.

Chaque module est identifié par un identificateur numérique permettant d'abstraire le niveau d'adressage. En effet, quand un module effectue une requête de lecture ou d'écriture vers une autre composante du système (module ou périphérique), l'identificateur de la composante destinatrice est converti en adresse interne. La génération de l'adresse à partir de l'identificateur est décrite à la fin de ce chapitre.

Un module matériel est connecté à l'architecture de base, en l'occurrence au bus de l'architecture via un adaptateur (*wrapper*) qui assure principalement deux rôles :

- **Découplage de l'interface entre le module et le bus :** selon le niveau de raffinement, un ou l'autre des bus UTF, TF ou BCA est utilisé dans l'architecture de base. La présence d'un adaptateur entre le module et le bus permet d'éliminer la nécessité d'avoir plusieurs interfaces différentes au niveau de la classe SpaceBaseModule pour connecter un module sur un bus.
- **Encapsulation des mécanismes de communication :** les mécanismes de communication utilisés dans Space sont gérés au niveau de l'adaptateur du module et au niveau du bus, ce qui permet de séparer la communication du calcul au niveau du module. Les données à envoyer par le module sont acheminées à travers l'adaptateur vers le bus, alors que les données reçues sont provisoirement stockées au niveau de l'adaptateur jusqu'à ce que le module soit prêt pour les lire. Par conséquent, les opérations de lecture sont locales entre le module et son adaptateur, et ne requièrent par les services du bus.

Un module logiciel est converti dans la partie logicielle du système en une tâche logicielle gérée au niveau du RTOS utilisé. Nous verrons au paragraphe suivant les détails de la partie logicielle de l'architecture de base.

Architecture de base

L'architecture de base contient des composantes standards formant l'infrastructure de base pour concevoir un système utilisant un bus comme médium de communication. Chaque composante modélise un comportement au sein d'un SoC et est disponible à l'utilisation sous forme d'une librairie. Certaines composantes telles que le bus d'une architecture de base diffèrent d'un niveau de raffinement à un autre.

Ces éléments de l'architecture sont conçus en SystemC pour modéliser le fonctionnement d'une partie du système, l'autre partie est modélisée par les modules. La figure 2.4 représente une architecture de base au niveau de raffinement BCA dans Space.

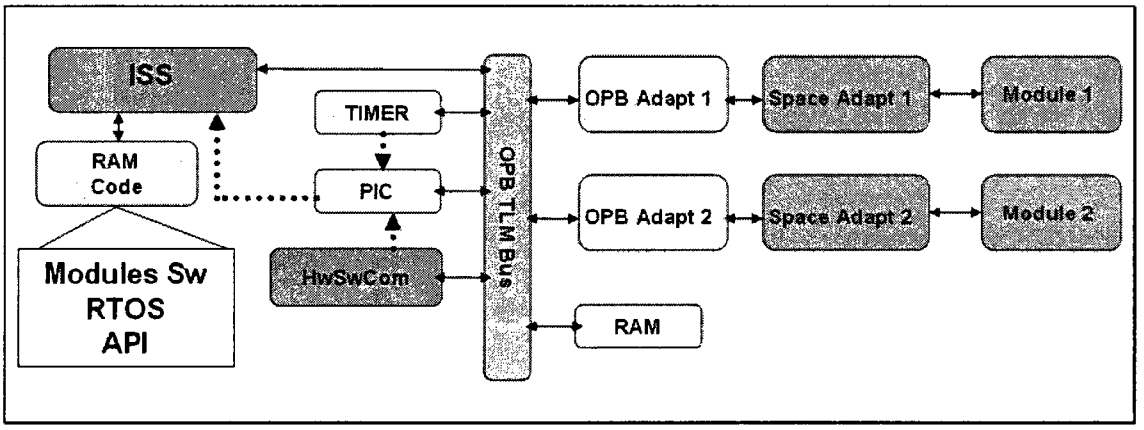

Figure 2. 4 Architecture de base au niveau de raffinement BCA de Space

Les composantes *OPB TLM Bus*, *Timer, RAM*, *PIC* (gestionnaire d'interruptions), *OPB adaptateur*, *Space adaptateur*, *HwSwCom* (adaptateur de communication matériel/logiciel), *ISS*, et *RAM code* forment l'architecture de base du système de niveau BCA, et sont fournies par la plateforme Space.

Le code binaire chargé dans la composante « RAM code » modélise le comportement du logiciel.

Les modules 1 et 2 connectés au bus à travers des adaptateurs modélisent le comportement matériel. La présence de deux adaptateurs est justifiée par la différence

entre l'interface du bus utilisé à ce niveau de raffinement et l'interface de chacun des bus utilisés aux autres niveaux (TF et UTF) de raffinement.

À ce niveau de raffinement, le système est partitionné en logiciel et en matériel. La partition logicielle du système est représentée par les modules logiciels originalement décrits en SystemC. Ces modules sont convertis en tâches logicielles gérées et ordonnancés par le RTOS choisi. La conversion des appels SystemC à des appels au niveau du RTOS est effectuée par une API qui, lors de compilation, effectue la correspondance directe entre les éléments de SystemC et ceux du RTOS. Le tableau 2.1 présente quelques correspondances entre les appels SystemC et ceux de MicroC-OS II qui est le RTOS utilisé dans Space.

Tableau 2. 1 Correspondances entre les appels SystemC et ceux de MicroC-OS II

Appels SystemC	Appels MicroC-OS II	Désignation
Sc_start ()	OSStartScheduler ()	Démarrage de l'Ordonnanceur du RTOS
Sc_stop ()	OSStopScheduler ()	Arrêt de l'Ordonnanceur du RTOS
wait ()	OSTaskDelay ()	Attente pour un certain nombre de cycles
sc_event ()	OSSemBCreate ()	Création d'un événement en SystemC correspond à un sémaphore au niveau du RTOS.
notifie ()	OSSemBGive ()	Notification de l'événement.

Cette conversion, entre SystemC et un RTOS donné, permet d'apporter le support temps réel au système avec tous les mécanismes fournis par le RTOS à savoir l'ordonnancement préemptif et la gestion des tâches basée sur les priorités.

2.2.2 Les niveaux de raffinement dans Space.

Tel que mentionné précédemment, trois niveaux de raffinement sont possibles dans Space. Ils diffèrent principalement au niveau du canal ou bus de communication utilisé. Les modules changent légèrement par l'ajout ou le retrait de notions temporelles selon le

niveau de raffinement. Les autres composantes standards peuvent ou non être utilisées par l'architecture de base selon le niveau de raffinement.

Le niveau UTF

Le niveau UTF permet de valider le système au niveau fonctionnel sans aucune notion de latence des communications. La simulation du système à ce niveau est équivalente à une simulation comportementale d'un niveau RTL. Les opérations de lecture et d'écriture sont modélisées par des appels de fonctions qui seront décrites au paragraphe 2.2.3. L'appel d'une fonction de lecture ou d'écriture au niveau du module se propage jusqu'au destinataire à travers l'adaptateur du module source et à travers le bus.

À ce niveau de raffinement, le système n'est pas partitionné, la validation est faite au niveau système. Le bus ou le canal utilisé à ce niveau ne correspond à un aucun standard de bus, et sert simplement à véhiculer la requête de la source à la destination. La figure 2.5 présente un système composé de quatre modules connecté chacun au canal UTF à travers un adaptateur.

Le niveau TF

Le niveau TF permet de valider le système au niveau fonctionnel, la différence par rapport au niveau UTF réside dans le fait que la communication à travers le bus TF a une latence d'un cycle pour une donnée de 4 octets, en plus des latences liées au décodage d'adresses et à l'attente de l'acquittement. Ces deux dernières latences sont configurables par l'utilisateur ou sont définies selon un protocole de communication choisi par l'utilisateur tel que le protocole du bus OPB ou celui du bus AMBA AHB.
À ce niveau un arbitrage de type FIFO (Round Robin) est aussi ajouté par rapport au niveau UTF. Pour représenter le système de la figure 2.5 au niveau TF, nous remplaçons uniquement le bus UTF par le bus TF.

Le niveau BCA

La figure 2.4 représente un système modélisé dans Space au niveau du raffinement BCA. À ce niveau le protocole de communication implémenté par le bus est précis au cycle près et l'architecture de base du système à haut niveau est identique à une architecture standard que l'on peut trouver au niveau RTL sur un FPGA. À ce jour, la plateforme Space fournit un modèle transactionnel pour le bus OPB du standard CoreConnect [10]. Ce modèle implémente à haut niveau la plupart des options (*features*) du protocole de communication du bus OPB au niveau RTL. Ainsi, le modèle calcule à chaque cycle d'exécution la valeur de chaque signal de son interface. À haut niveau, ces signaux sont modélisés par une variable de type booléenne (comparativement à l'utilisation d'un signal binaire pour le niveau RTL).

Le système à ce stade peut être partitionné en logiciel et/ou en matériel. Ce qui permet de valider les différentes parties du système final (à savoir le logiciel, le matériel et les interfaces de communications), et d'explorer plusieurs configurations en déplaçant les modules du logiciel au matériel et vis-versa.

Nous rappelons que l'objectif de ce projet de recherche est de construire un pont entre ce niveau de raffinement et l'implémentation d'un système sur une puce.

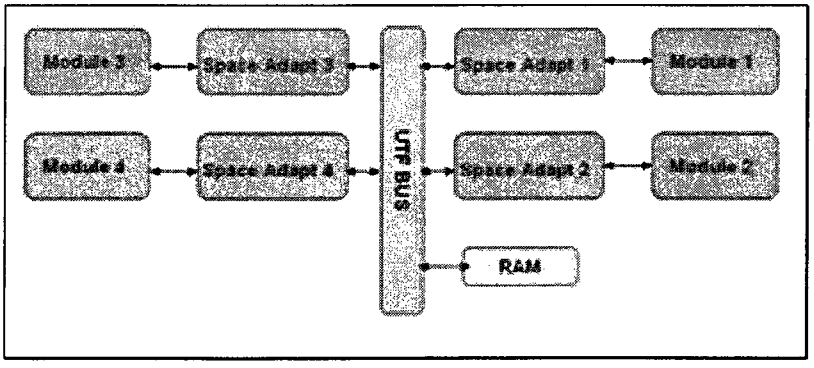

Figure 2. 5 Un système composé de quatre modules au niveau UTF

2.2.3 Le protocole de communication dans Space.

Nous avons vu qu'un système conçu avec la méthodologie de la plateforme Space est constitué d'un ensemble de modules partitionnés en matériel et/ou en logiciel et d'une architecture de base composée de plusieurs composantes fournies par la plateforme. Pour un tel système, il faut assurer la communication entre les modules et les différentes composantes.

Les fonctions de communication

Les communications sont assurées par le bus et utilisent les mécanismes implémentés au niveau des adaptateurs pour les modules matériels ou des mécanismes logiciels (FIFO, événements, synchronisation, etc.) pour les modules logiciels. Un module utilise une fonction parmi quatre pour communiquer soit avec un module ou soit un périphérique de l'architecture. Ces fonctions sont décrites ci-dessous :

- **ModuleWrite () :** l'appel de cette fonction par un module permet d'envoyer un message d'un module vers un autre module. Le message est acheminé à travers le bus jusqu'à l'adaptateur du module destinataire où il sera stocké dans une FIFO jusqu'à ce que le module destinataire puisse consommer le message. Il faut noter que les messages envoyés par un module matériel vers un module logiciel sont acheminés vers la composante HwSwCom qui par un mécanisme d'interruption signale au processeur que des données destinées à un module logiciel ont été reçues.
- **ModuleRead () :** l'appel de cette fonction par un module permet de lire un message qui a été envoyé par un autre module et stocké dans une FIFO de réception au niveau de l'adaptateur pour un module matériel ou dans une FIFO en logiciel pour un module logiciel. Cette opération est locale, entre un module matériel et son adaptateur ou sur la FIFO logicielle pour un module logiciel et ne requiert donc pas l'accès au bus.

- **DeviceWrite () :** l'appel de cette fonction par un module permet d'écrire un message à un périphérique. Le message est acheminé par le bus.
- **DeviceRead () :** l'appel de cette fonction par un module permet de lire un message à partir d'un périphérique. Le message est acheminé par le bus.

Le Tableau 2.2 présente les paramètres que prennent les quatre fonctions présentées ci-dessus.

Tableau 2. 2 Arguments des fonctions de communications

Paramètre	Signification	A	B	C	D
OrgID	L'identificateur numérique du module source de la requête	x	x	x	x
DestID	L'identificateur numérique du module ou périphérique destinataire de la requête	x	x	x	x
Priority	Priorité du message durant la requête courante	x	x	x	x
*Data	Un pointeur sur le message à transférer	x	x	x	x
DataSize	La taille du message à transférer.	x	x	x	x
TimeOut	Paramètre pour indiquer si l'opération est bloquante ou non	x	x		
DevOffset	Déplacement par rapport à l'adresse de début du périphérique destinataire.			x	x

A: ModuleWrite, B: ModuleRead, C: DeviceWrite, D: DeviceRead
x: pour indiquer si le paramètre en question est utilisé ou non.

Les différents types de communication dans Space

Un module est un maître sur le bus, et peut initier des requêtes de lecture ou d'écriture, alors que les périphériques tels qu'une mémoire RAM sont esclaves et se contentent de répondre aux requêtes d'un maître via le bus.

On distingue la communication module à module et module à périphérique. La gestion de ces types de communication diffère selon la partition dans laquelle un module a été instancié et selon le niveau de raffinement. Nous décrirons ci-dessous les communications au niveau du raffinement BCA dans lequel le système est en général

partitionné en logiciel et en matériel. Notez que pour la communication entre modules, il faut toujours garder en tête le modèle de communication par rendez-vous, c'est-à-dire que lorsqu'un module A demande une requête de lecture (e.g. ModuleRead) à un module B, ce dernier répondra par une requête d'écriture (ModuleWrite). De façon symétrique, lorsqu'un module A demande une requête d'écriture (e.g. ModuleWrite) à un module B, ce dernier répondra par une requête de lecture (ModuleRead).

- **Communication matériel-matériel** Il s'agit d'une communication entre deux modules matériels ou entre un module matériel et un périphérique. La figure 2.6 représente le cheminement d'une requête de lecture ou d'écriture pour une communication matériel-matériel. Le premier cas (figure 2.6A) représente l'écriture d'un message d'un module matériel M1 à un autre module matériel M2 alors que le deuxième cas (figure 2.6B) représente la lecture d'un message. Dans ce dernier cas, la lecture est toujours locale, c'est-à-dire que le module M1 lira les messages, qui lui ont été envoyées, au niveau de son adaptateur sans utiliser le bus. Le troisième cas (figure 2.6C) représente une requête de lecture ou d'écriture du module matériel M1 vers un périphérique D. Dans ce dernier cas la requête utilise le bus pour la lecture et pour l'écriture.

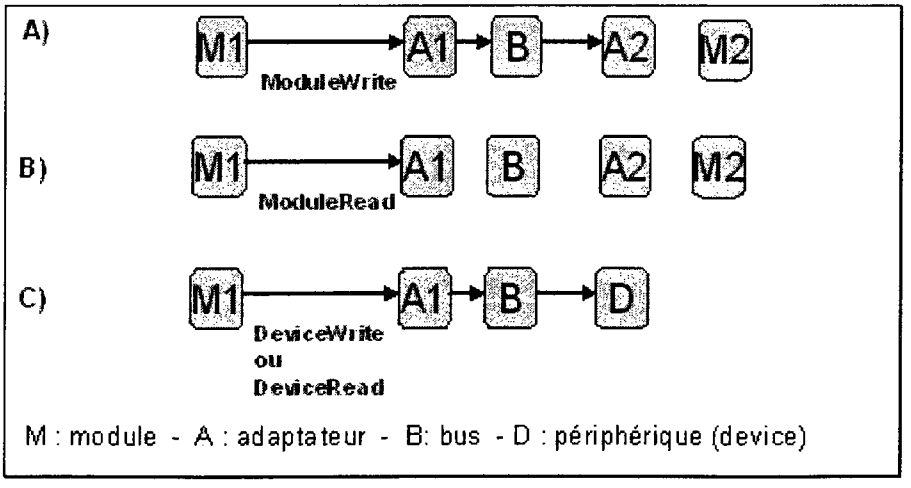

Figure 2. 6 Illustration de la communication de matériel à matériel

- **Communication matériel-logiciel** Il s'agit d'une communication entre un module matériel et un module logiciel. La figure 2.7 illustre ce type de communication au niveau raffinement BCA. Le message envoyé par le module matériel M1 au module logiciel M2 est acheminé à travers le bus vers la composante HwSwCom qui est un adaptateur entre le matériel et le logiciel. Cette composante utilise un mécanisme d'interruption pour signaler à l'émulateur du processeur (ISS de l'anglais Instruction Set Simulator) que des données ont été reçues de la part d'un module matériel. À travers une sous-routine d'interruption, l'ISS initie une requête de lecture vers l'adaptateur HwSwCom et achemine les données à la FIFO dédiée au module logiciel M2. Ce dernier effectue ses requêtes de lecture au niveau de sa FIFO. Le cas d'une lecture à partir du module M1 est couvert dans le point suivant.

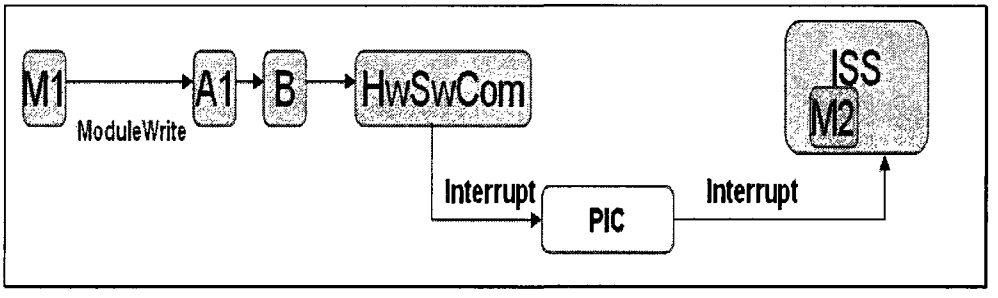

Figure 2. 7 Illustration de la communication de matériel à logiciel

- **Communication logiciel-matériel (figure 2.8)** L'écriture d'un message par un module logiciel vers un module matériel ou un périphérique s'effectue directement à travers le bus. L'ISS demande le bus et initie la requête quand l'accès au bus est autorisé. Encore une fois, pour le module M1 la lecture est locale, c'est-à-dire que le module M1 lira les messages, qui lui ont été envoyées par M2, au niveau de son adaptateur sans utiliser le bus.

La lecture de données par un module logiciel à partir d'un périphérique s'effectue de la même manière que l'écriture.

- **Communication logiciel-logiciel :** cette communication est interne au logiciel, et se traduit par une communication entre les tâches crées par correspondance entre les appels SystemC et ceux au niveau du RTOS. Chaque tâche logicielle représente un module et utilise une FIFO dans laquelle sont stockés les messages qui lui sont destinés. L'écriture et la lecture d'un message par un module logiciel vers un autre se traduisent respectivement par une écriture et une lecture sur une FIFO.

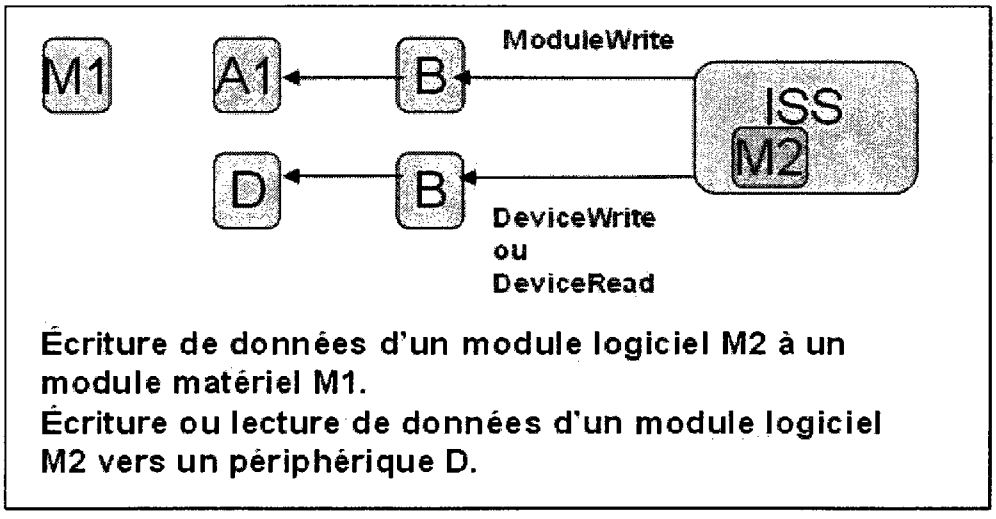

Figure 2. 8 Illustration de la communication du logiciel au matériel

Nous venons de présenter les différents types de communication pour le niveau BCA. Qu'en est-il maintenant des niveaux de raffinement UTF et TF ? En fait, ces niveaux sont beaucoup plus simples à modéliser car ils se ramènent simplement au cas d'une communication matériel-matériel (1er cas ci-haut).

Notez également que la couche des adaptateurs (désignés par la lettre A aux figures 2.6, 2.7 et 2.8) est en fait composée d'un seul ou de deux adaptateurs selon le niveau de raffinement ciblé. Plus précisément, deux adaptateurs sont utilisés pour le niveau de raffinement BCA (figure 2.4) : le premier est connecté au module alors que le deuxième est connecté au modèle SystemC du bus OPB. Par contre, un seul adaptateur connecté au module est requis pour le niveau UTF et TF (figure 2.5).

Communication bloquante versus non bloquante

Un autre paramètre important en ce qui concerne les communications entre modules est l'aspect bloquant versus non bloquant (voir description plus loin dans cette section). La valeur de l'argument **timeOut** des fonctions **ModuleWrite** et **ModuleRead** permet de distinguer la nature de la communication : 0 signifie non bloquante, 255 signifie bloquante de manière indéfinie. Entre ces deux valeurs ($0 < x > 255$), on a une communication bloquante avec un délai d'attente (timeout) égal à x. Les comportements possibles sont donc les suivants :

- **Écriture non bloquante** Un module qui effectue une écriture non bloquante peut poursuivre son exécution immédiatement après le traitement de sa requête.
- **Lecture non bloquante** Un module qui effectue une lecture non bloquante peut poursuivre son exécution immédiatement après avoir vérifié si des données étaient stockées dans sa FIFO de réception.
- **Écriture bloquante** Un module qui effectue une écriture bloquante attend un acquittement **ACK** de la part du module destinataire. Cet acquittement lui sera envoyé par le module destinataire dès que ce dernier consomme le message en question. Le paramètre **timeOut** passé comme argument de la fonction d'écriture donne le temps maximal (en cycles) pour lequel le module doit attendre cet acquittement.
- **Lecture bloquante** Dans ce cas le module bloquera s'il ne trouve pas le message qu'il désire lire dans la FIFO de réception. Encore une fois, le paramètre **timeOut** passé comme argument donne le temps maximal d'attente.

Dans les quatre cas une valeur de retour de la fonction d'écriture ou de lecture permet de vérifier le statut de la requête.

Le format d'un message au niveau BCA

Au niveau BCA, le protocole de communication du bus OPB est modélisé au niveau transactionnel. Pour la communication entre modules, un message est construit à partir des paramètres des fonctions de communication. Il est composé d'un en-tête , de quatre octets et d'une donnée (figure 2.9).

La donnée peut avoir une taille variant de 4 à 256 octets, et elle est sérialisée par l'adaptateur du bus OPB selon la largeur allouée à la donnée.

L'en-tête est composé de 4 champs de un octet chacun :

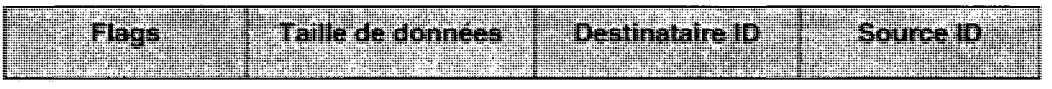

Figure 2. 9 Format de l'en-tête d'un message

Les champs Destinataire ID et source ID permettent de distinguer le module source et le module destinataire de la requête. Le champ Destinataire ID est utile lorsque plusieurs modules sont connectés au même adaptateur, alors que le champ Source ID est utile puisqu'un module peut recevoir des messages de plusieurs autres modules. Il permet aussi de reconstruire l'adresse à laquelle un ACK est envoyé dans le cas d'une écriture bloquante.

Le champ de drapeaux (Flags) sert à indiquer certaines options concernant le message, telles que l'écriture bloquante et l'acquittement.

Notons finalement que dans le cas de communication entre modules et périphériques le message contient uniquement la donnée.

La génération de l'adresse d'une composante ou d'un module

Pour tous les niveaux de raffinement, le bus utilise des adresses de 4 octets. Les périphériques, les modules et leurs adaptateurs au sein d'un système sont identifiés chacun par un identificateur numérique.

À l'instanciation des composantes du système au niveau de la fonction **sc_main** de SystemC qui est l'entrée principale de l'exécutable, et avant le lancement de la simulation, les périphériques et les modules forment chacun son adresse de manière dynamique et s'enregistrent sur une table globale faisant la correspondance entre identificateur et adresse.

Pour acheminer les données à travers le bus il faudra retrouver l'adresse de destination à partir de la table globale. Cette manière de générer les adresses à partir des identificateurs, est utilisée dans le but de :

- Augmenter le niveau d'abstraction lors de l'implémentation du code d'un module puisqu'il est plus simple de travailler avec un identificateur de un octet que de travailler avec des adresses de 4 octets.
- Réduire la taille de l'en-tête en passant les ID destination et source de un octet chacun au lieu de passer deux adresses chacune sur 4 octets.
- Faciliter le passage d'un module de la partition logicielle à la partition matérielle et vis-versa.

Le Tableau 2.3 présente la composition d'une adresse à partir de l'identificateur du périphérique ou du module.

Tableau 2. 3 Génération d'une adresse à partir d'un identificateur

<table>
<tr><th></th><th>Bits 30 à 31</th><th>Bits 22 à 29</th><th>Bits 14 à 21</th><th>Bits 0 à 13</th></tr>
<tr><td>Adresse d'un module</td><td rowspan="2">Bits prévus pour la gestion des systèmes muti-bus</td><td>Identificateur de l'adaptateur</td><td>Identificateur du module</td><td>Non utilisé ou espace mémoire dans le module</td></tr>
<tr><td>Adresse d'un périphérique</td><td>Identificateur du périphérique</td><td colspan="2">Espace mémoire du périphérique (maximum de 4 Moctets).</td></tr>
</table>

Chapitre 3 Raffinement des communications de Space

Ce chapitre décrit la méthodologie utilisée pour implémenter le protocole de communication de Space sur un circuit reprogrammable de type FPGA afin d'assurer le passage du niveau BCA au niveau RTL. Nous avons choisi le FPGA Virtex2 de la compagnie Xilinx avec un processeur embarqué de type Microblaze et une architecture de bus basée sur le standard CoreConnect OPB [10].

Une application conçue avec la plateforme Space est soit complètement matérielle, soit complètement logicielle ou partitionnée en logiciel et en matériel. Dans les trois cas, le protocole de Space, à haut niveau met en jeu différents mécanismes de gestion des communications entre les différentes composantes de l'application (maîtres, esclaves, bus, logiciel, etc.). Nous avons considéré le cas général dans lequel le système à concevoir contient une partition logicielle et une partition matérielle.

Notre travail consiste donc à passer d'une spécification d'un système à haut niveau basée sur la plateforme Space vers une spécification au niveau RTL synthétisable. En particulier nous implémentons les mêmes mécanismes d'adaptateurs vus au chapitre 2 pour gérer les communications du matériel-matériel, matériel –logiciel et logiciel-matériel.

Nous avons utilisé les outils de développement de la compagnie Xilinx, en particulier l'outil EDK et la carte multimédia de Xilinx pour le prototypage rapide de SoC. Nous commençons ce chapitre par une brève description de ces outils avant d'aborder la méthodologie utilisée pour implémenter les mécanismes de communications de Space sur le circuit FPGA de la carte de prototypage utilisée.

3.1 Les outils de développement

3.1.1 EDK

Pour faciliter le développement de systèmes embarqués sur FPGA de Xilinx, EDK permet de spécifier le système de manière simple basée sur des interfaces graphiques qui guident le concepteur dans les différentes étapes de spécification de l'application. L'outil EDK vient avec un ensemble de composantes IP (uart, timer, bus, etc.) et deux types de processeurs Microblaze et PowerPC 405. L'ensemble de ces composantes permet de bâtir l'architecture du système, d'intégrer la partie logicielle de ce système et d'incorporer au système de nouvelles composantes non fournies par les librairies de EDK. Le menu « Create/Import Peripheral » dans EDK permet de générer une nouvelle composante disposant de l'interface nécessaire pour être connectée sur un des bus de l'architecture CoreConnect.

Le système en entier est spécifié dans un fichier MHS (Microprocessor Hardware Specification) dans lequel chaque composante matérielle du système est configurée avec l'ensemble de ses paramètres et se voit attribuer une adresse si elle est connectée au bus de l'architecture. Les ports locaux et externes sont aussi spécifiés. Le fichier MHS est utilisé par l'outil PlateGen (Plateform Generator) pour générer le système embarqué sous forme d'une liste d'interconnexions.

Un autre fichier MSS (Microprocessor Software Specification) est aussi généré automatiquement par EDK dans lequel sont spécifiés les pilotes logiciels des différentes composantes du système et certains paramètres liés au fonctionnement du logiciel du système embarqué tel que les ports et les gestionnaires d'interruptions, la fréquence du processeur, etc. Le fichier MSS est utilisé par un outil générateur de librairies pour la compilation et l'édition des liens avec le code logiciel de l'application.

Un ensemble d'autres outils est ensuite utilisé pour le placement des fonctions logiques sur les blocs CLB du FPGA et pour le routage des connexions. Toutes ces étapes permettent de générer un fichier de bits (bitstream) utilisé pour programmer le FPGA.

3.1.2 FPGA Virtex-II de Xilinx

La technologie des FPGA permet aux concepteurs de procéder à la vérification des systèmes durant le cycle de développement. Un FPGA est un circuit intégré générique composé de blocs logiques (CLBs) et d'interconnexions programmables. Le système conçu est implémenté sous forme de fonctions logiques simples sur chaque bloc. Les FPGA modernes contiennent suffisamment de logique pour implémenter un SoC ou d'autres systèmes plus complexes.

Le concepteur décrit son système dans un langage HDL tel que VHDL ou Verilog. Un outil de synthèse est utilisé afin de convertir cette description en bitstream pour programmer le circuit FPGA. Ce type de circuit est reprogrammable plusieurs fois, ce qui implique une grande flexibilité de mise à jour et de corrections du système conçu.

Un FPGA de la famille Virtex-II [41] est constitué de blocs d'entrée/sortie (IOBs), et de blocs logiques internes. Les blocs IOBs assurent l'interface entre les pins externes et la logique interne.

Un bloc de logique interne est constitué principalement de quatre éléments :

- Élément fonctionnel : utilisé pour la logique combinatoire et séquentielle.
- Élément de stockage large de 18Kbits : fourni par des mémoires RAM à double port.
- Multiplicateurs dédiés de 18x18 bits.
- Gestionnaire d'horloge numérique (DCM : Digital clock manager).

Un CLB consiste en 4 tranches (en anglais *slices*), qui sont la base pour déterminer l'utilisation des ressources après la synthèse. Chaque slice est composée de 2 générateurs de fonctions à 4 entrées, 2 éléments de stockage, une unité arithmétique et des multiplexeurs. La structure de LUT est généralement utilisée pour les générateurs de fonctions, les registres de décalage de 16 bits ou des éléments de mémoire distribuée RAM de 16 bits. L'utilisation d'une LUT, permet d'implémenter n'importe quelle fonction logique à 4 entrées. Les multiplexeurs d'une slice permettent d'implémenter la majorité des fonctions logiques à 8 entrées.

La figure 3.1 présente une tranche d'un CLB du FPGA Virtex 2.

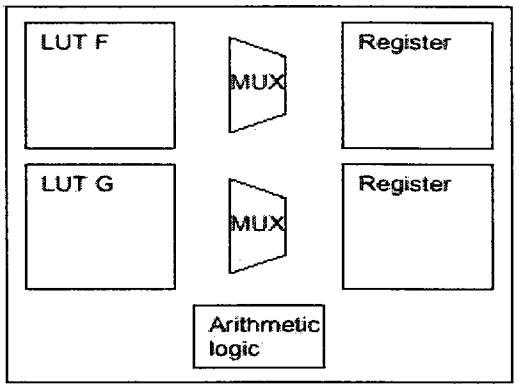

Figure 3. 1 Slice d'un CLB. FPGA Virtex-II

3.2 Méthodologie

Le raffinement d'un système conçu à haut niveau dans Space et que l'on désire implémenter sur un circuit reprogrammable peut être divisé en quatre sous parties (figure 3.2).

- **Génération du logiciel** Nous avons vu au chapitre 2 qu'un système conçu dans Space peut être partitionné en logiciel et en matériel au niveau de raffinement BCA. À ce niveau un émulateur de processeur permet de simuler le logiciel composé d'une API de conversion des appels SystemC en appel au niveau du RTOS choisi, Mis à part le code des modules logiciels, le reste est compilé sous forme de librairies.

  Au niveau de l'implémentation sur FPGA, toutes les librairies utilisées au niveau BCA et le code des modules sont repris sans aucune modification et compilés par l'outil de développement EDK pour produire l'exécutable de la partie logicielle du système à implémenter sur FPGA.

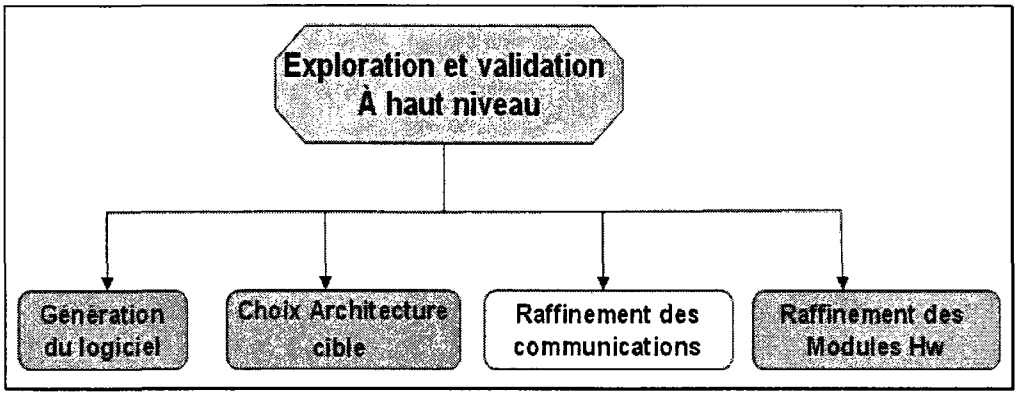

Figure 3. 2 Méthodologie de raffinement d'un système conçu avec Space

- **Choix de l'architecture cible** Au chapitre 2, nous avons décrit un système conçu dans Space comme étant composé : 1) d'un ensemble de modules logiciel ou matériel et 2) d'une architecture de base construite à partir de plusieurs composantes que l'on retrouve dans différentes applications telles qu'un bus, une mémoire, un uart, etc. Pour passer d'un système à haut niveau vers le niveau de l'implémentation physique, une correspondance (mapping) est effectuée entre les composantes de Space à haut niveau et des IP d'une architecture cible choisie. Pour notre projet, nous avons choisi l'architecture CoreConnect implémentée sur les FPGA de Xilinx. Cette architecture fournit les fondations de base pour bâtir un système embarqué utilisant une architecture de bus en l'occurrence le bus OPB. Les composantes IPIF fournies par Xilinx facilitent l'ajout et l'adaptation de nouvelles composantes maîtres et/ou esclaves sur le bus OPB. Nous avons d'ailleurs utilisé ces composantes afin de concevoir l'adaptateur d'un module matériel et l'adaptateur des communications entre le matériel et le logiciel (composante HwSwCom) que nous allons décrire respectivement aux sections 3.3..3 et 3.3.4

  La figure 3.3 présente une architecture type basée sur le standard CoreConnect. Nous décrirons brièvement le protocole et l'interface du bus OPB à l'annexe A et le processeur Microblaze à l'annexe B.

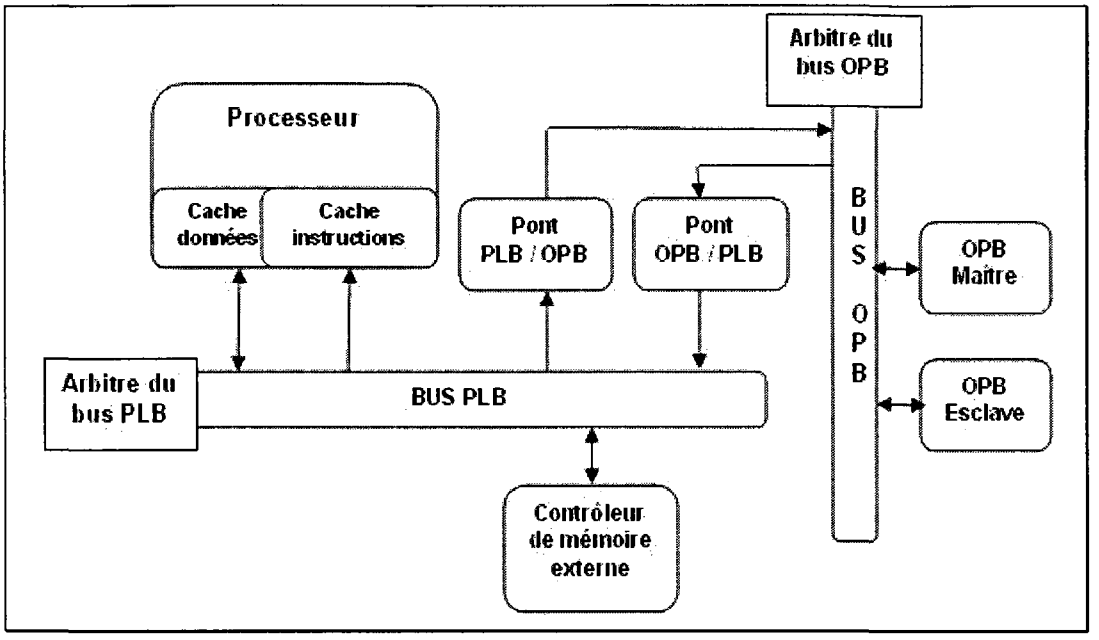

Figure 3. 3 Architecture type basée sur le standard CoreConnect de IBM

- **Raffinement des communications** Cette partie de la démarche constitue une des contributions centrales de ce travail. Il s'agit d'implémenter au niveau FPGA le même protocole de communication, par échange de messages, utilisé à haut niveau. Nous avons vu à la section 2.2.3 que les modules communiquent à travers les adaptateurs, le bus et éventuellement la composante HwSwCom. À bas niveau, nous avons conçu ces adaptateurs afin d'interfacer les modules sur le bus OPB en utilisant la composante IPIF fournie par Xilinx. Nous décrivons la composante IPIF et l'implémentation des adaptateurs respectivement à l'annexe C et à la section 3.3
- **Raffinement des modules matériels** Cette partie du raffinement d'un système conçu à haut niveau dans Space ne fait pas partie des objectifs de notre travail. Elle nécessitera des outils de translation d'une spécification basée sur SystemC vers une spécification en langage HDL synthétisable. Néanmoins, pour valider le fonctionnement de l'implémentation des adaptateurs au niveau RTL, nous fournissons une interface entre un adaptateur et un module matériel, et nous retranscrivons manuellement le code des modules Space en VHDL.

Nous présentons à la section 3.3.1 cette interface, et nous décrivons l'équivalent au niveau RTL des appels des fonctions de communications vues au chapitre 2.

3.3 Implémentation au niveau RTL

Au paragraphe 3.2 nous avons expliqué les grandes lignes pour raffiner un système, conçu à haut niveau dans Space. En particulier, nous avons vu que le logiciel simulé à haut niveau par un ISS est réutilisable à bas niveau sans aucune modification. Nous avons également présenté l'architecture de base ciblant le standard de bus OPB avec tous ses périphériques nécessaires fournis sous forme de IP avec les outils de Xilinx. Il ne reste donc qu'à concevoir et implémenter les mécanismes de communications et à proposer une interface au niveau RTL pour les modules utilisateur.

La communication est assurée à travers les adaptateurs des modules matériels (description détaillée à la section 3.3.3) et à travers la composante HwSwCom (description détaillée à la section 3.3.4). Nous décrivons dans cette section l'architecture et le fonctionnement de ces adaptateurs, ainsi que **l'approche de raffinement manuelle** d'un module de Space du niveau BCA au niveau RTL.

Nous avons implémenté les adaptateurs au niveau RTL en VHDL.

3.3.1 Interface d'un module matériel

Pour établir l'interface entre un module matériel et son adaptateur, nous nous sommes basés sur les types de communication représentés à haut niveau par les fonctions de communication décrites au paragraphe 2.2.3 et sur les arguments passés à ces fonctions. Le type d'opération est défini par la valeur du signal **Mod2Adapt_xferType** et chaque argument des fonctions à haut niveau correspond à un signal au niveau RTL. La figure 3.4 présente l'interface entre un module matériel et son adaptateur.

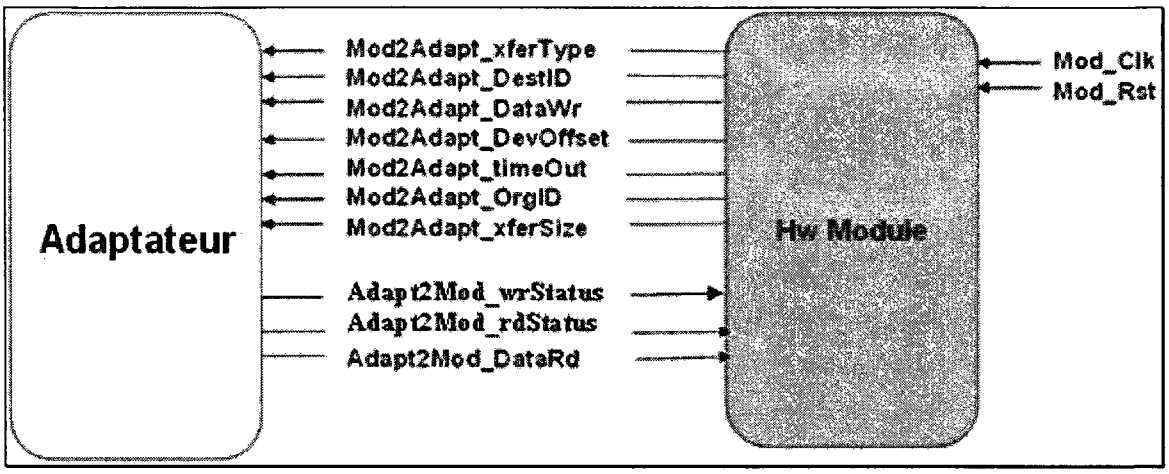

Figure 3. 4 Interface entre un module matériel et son adaptateur au niveau RTL

Nous présentons au tableau 3.1 la description des signaux de l'interface de la figure 3.4.

Tableau 3. 1 Description des signaux de l'interface d'un module matériel

Signal	Taille en bits	Description
Mod_Clk	1	Signal de l'horloge
Mod_Rst	1	Signal de mise à zéro (Reset)
Mod2Adapt_xferType	3	Signal pour choisir le type de requête.
Mod2Adapt_DestID	8	Identificateur numérique du destinataire de la requête.
Mod2Adapt_OrgID	8	Identificateur numérique de la source de la requête.
Mod2Adapt_DevOffset	22	Le déplacement (offset) à partir de l'adresse de départ du périphérique destinataire de la requête.
Mod2Adapt_timeOut	8	Signal pour préciser si la communication est bloquante ou non.
Mod2Adapt_xferSize	5	Le nombre de transferts.
Adapt2Mod_wrStatus	4	Le statut de la requête d'écriture
Adapt2Mod_rdStatus	4	Le statut de la requête de lecture
Adapt2Mod_DataRd	32	La donnée lue par le module.
Mod2Adapt_DataWr	32	La donnée écrite par le module vers un autre module ou un périphérique.

Nous avons présenté au tableau 3.1 les signaux de l'interface entre un module matériel et son adaptateur. La taille de chacun de ces signaux a été choisie en tenant compte de ce qui suit :

- Respecter les concepts de Space à haut niveau, en l'occurrence les paramètres des fonctions de communications décrites au paragraphe 2.2.3. En effet, nous avons vu que l'identificateur numérique d'un module est de type « unsigned char » équivalent à 8 bits au niveau RTL.
- Respecter certaines limitations dues aux librairies de Xilinx, spécialement le taille de transfert codée sur 5 bits, ce qui permet de transférer un maximum de 64 octets de manière atomique en utilisant le mode rafale.
- Le signal Mod2Adapt_xferType qui défini le type de requête est codé sur 3 bits. Cette taille permet d'avoir jusqu'à 8 types de requêtes. La version actuelle de l'adaptateur supporte 5 types de requêtes à savoir l'écriture module à module, la lecture module à module, l'écriture module à périphérique, la lecture module à périphérique et le placement des données dans la FIFO d'envoi avant d'initier une requête d'écriture.

L'interface d'un module présentée au tableau 3.1 ne permet pas d'effectuer une lecture simultanément avec une écriture, il serait possible de séparer les deux opérations en ajoutant d'autres signaux à cette interface ou en procédant à une gestion en interne dans le code VHDL d'un module. Ci-dessous nous décrivons plus en détail certains signaux de l'interface d'un module.

- **Mod2Adapt_xferType** Ce signal permet de choisir le type de requête à initier par le module. Il est assigné avec l'une des valeurs constantes du tableau 3.2.

Tableau 3. 2 Types de requêtes d'un module matériel au niveau RTL

Valeur	Description
SP _NO_XFER	Aucune requête.
SP _MOD_WRITE	Écriture des données à un autre module
SP _MOD_READ	Lecture des données envoyées par un autre module, cette lecture est locale au niveau l'adaptateur et ne nécessite pas l'accès au bus.
SP _DEV_WRITE	Écriture des données à un périphérique.
SP _DEV_READ	Lecture des données à partir d'un périphérique.
SP _INIT_WR	La donnée devra être écrite dans une FIFO d'envoi au niveau de l'adaptateur avant d'initier la requête d'écriture.

- **Adapt2Mod_wrStatus ou Adapt2Mod_rdStatus** Ces signaux permettent de vérifier le statut de la requête courante. Ils sont assignés au niveau de l'adaptateur du module avec l'une des valeurs constantes du tableau 3.3.
- **Mod2Adapt_timeOut** Ce signal est utilisé uniquement pour les communications entre modules. Selon sa valeur, la communication est bloquante (indéfiniment ou avec un certain délai) ou non bloquante.

Tableau 3. 3 Statut d'une requête d'un module matériel au niveau RTL

Valeur	Description
SP _RST	L'adaptateur est encours d'initialisation
SP _IDLE	L'adaptateur est libre. Le module peut initier la requête.
SP_XFER_IN_PROG	Le transfert des données est encours.
SP _XFER_OK	Le transfert des données est achevé.
SP_XFER_TIMEOUT	Le délai spécifié pour une communication bloquante est écoulé.
SP _XFER_ERR	Erreur lors du transfert. Cette valeur est retournée quand le bus OPB retourne une erreur.
SP _RD_NO_DATA	Aucune donnée n'est disponible dans la FIFO de réception.
SP _SIZE_ERR	La taille de transfert dépasse la taille maximale permise.
SP _WAIT	L'adaptateur est occupé à traiter une autre requête.

3.3.2 Fonctionnement d'un module matériel au niveau RTL

Nous avons décrit au paragraphe précédent l'interface RTL entre un module matériel et son adaptateur. Un module conçu à haut niveau avec la méthodologie de Space sera raffiné au niveau RTL en retranscrivant manuellement son code en VHDL.

Afin de minimiser les signaux de contrôle au niveau RTL, nous suggérons d'implémenter le module avec une machine à états finis. Au maximum deux états serviront à la communication alors que les autres états serviront au calcul.

- **Écriture** Dans l'exemple de la figure 3.5, le module a besoin d'écrire en premier 5 données de 4 octets chacune dans la FIFO d'envoi de son adaptateur (état INIT_DATA) et ensuite d'initier l'écriture (état WRITE_DATA). Dans ce dernier état, il vérifie si l'écriture a été achevée sinon il reste dans le même état. À la figure 3.5, d'autres tests de contrôle pourraient aussi être effectués (e.g. erreur ou délai d'expiration), mais ceux-ci ne sont pas été inclus dans le code de la figure 3.5 afin de simplifier la compréhension du lecteur.

```
.... États avant l'écriture
when ST_1 =>
        ... Traitement

-- Ecriture de 5 données du module de ID = 1 au module de ID= 2
when INIT_DATA =>
        if (count = 5 ) then
                count                   <= 0;
                iMod2Adapt_DataWr  <= (others=>'0');
                nstate                  <= WRITE_DATA;
                iMod2Adapt_xferType <= SP_NO_XFER;
        else
                iMod2Adapt_xferType <= SP_INIT_WR_BUF;
                iMod2Adapt_DataWr  <= iMod2Adapt_DataWr + 1;
                count                   <= count +1;
                nstate                  <= INIT_DATA;
        end if;
when WRITE_DATA =>
        if (Adapt2Mod_wrStatus = SP_XFER_OK )then
                iMod2Adapt_xferType         <= SP_NO_XFER;
                iMod2Adapt_DestID           <= (others=>'0');
                iMod2Adapt_OrgID            <= (others=>'0');
                iMod2Adapt_Timeout <= 0;
                nstate                      <= ST_2;
                iMod2Adapt_xferSize         <= 0;
        else
                iMod2Adapt_xferType         <= SP_MOD_WRITE;
                iMod2Adapt_DestID           <= X"02";
                iMod2Adapt_OrgID            <= X"01";
                iMod2Adapt_Timeout <= 0;
                nstate                      <= WRITE_DATA;
                iMod2Adapt_xferSize         <= 5;
        end if;
.... États après l'écriture
when ST_2 =>
        .... Reste du traitement
```

Figure 3. 5 Écriture d'un module matériel à un autre module au niveau RTL

- **Lecture** Le module initie la lecture des données qui lui ont été envoyées par un autre module (lecture locale dans la FIFO de réception au niveau de l'adaptateur) ou la lecture des données d'un périphérique (l'adaptateur initie une requête de lecture au niveau du bus OPB). Dans ce cas, le statut de la requête (signal Adapt2Mod_rdStatus) est testé et la valeur **SP _XFER_IN_PROG** indique que

le transfert est encours et que la donnée reçue par le signal **Adapt2Mod_DataRd** est valide. La fin de la lecture est signalée par la valeur **SP_XFER_OK**.

3.3.3 Adaptateur d'un module matériel

Nous avons développé cette composante en VHDL en utilisant l'IPIF maître/esclave de Xilinx. Tel qu'illustré à la figure 3.6, l'adaptateur d'un module matériel fournit une interface pour se connecter au bus OPB ainsi qu'une interface pour se connecter au module matériel. De plus, il est composé de plusieurs entités où chaque entité gère une partie de la fonctionnalité.

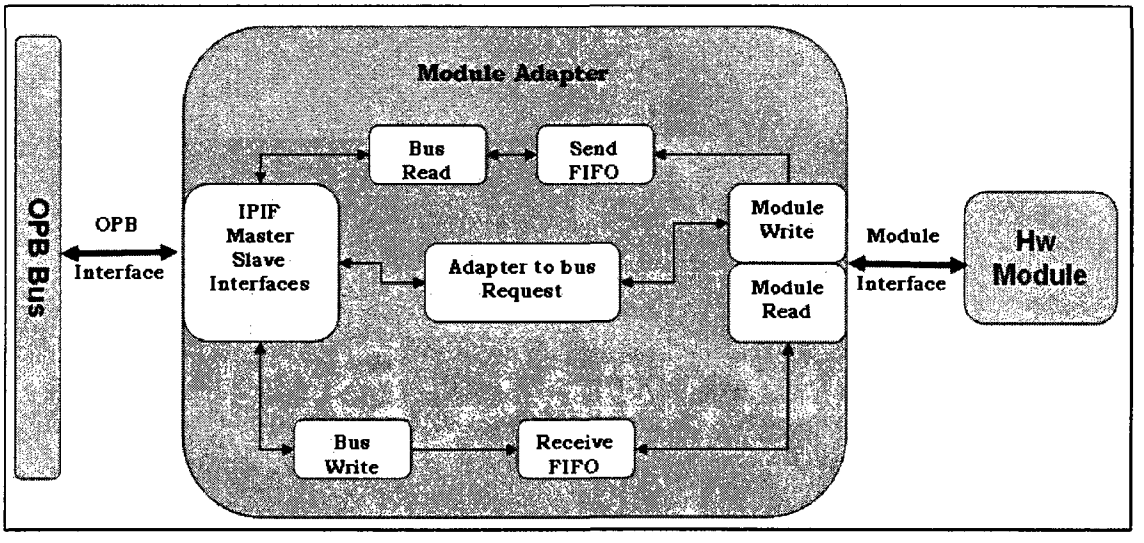

Figure 3. 6 Schéma de principe d'un adaptateur pour un module matériel

Description

Nous décrivons ci-dessous chacune des entités qui composent l'adaptateur d'un module.

- **Interface maître/esclave de l'IPIF** Nous décrivons cette entité à l'annexe C.

- **Entité Module Write** Avant d'initier l'écriture de données à un module ou à un périphérique, le module commence par l'écriture des données dans une FIFO d'envoi et initie ensuite l'écriture telle que nous l'avons décrit à la section 3.3.2.

La plupart du temps, le module a besoin d'utiliser le mode rafale pour envoyer plus qu'une donnée à la fois. Ce mode minimise la latence totale puisque le module demande l'accès au bus une seule fois pour plusieurs données. Autrement dit, un seul en-tête est envoyé dans un message composé de plusieurs données.

- **Entité Send FIFO** Déclare et implémente une FIFO d'une profondeur de 16 données. Ce paramètre suffisant pour notre expérimentation peut être modifié au besoin.

- **Entité Adapter to bus Request** C'est une machine à états finis qui gère les requêtes d'écriture vers un module ou vers un périphérique, ainsi que les requêtes de lecture vers un périphérique. Dans les deux cas, elle propage la requête en question à l'interface maître de l'IPIF. L'entité gère aussi les tâches suivantes :
  1. Gestion des écritures bloquantes d'un module à un autre module.
  2. Composition de l'en-tête du message à partir des identificateurs numériques de la source et la destination et selon le type d'écriture (e.g. bloquant ou non bloquant).
  3. Gestion du statut de la requête du module selon les réponses du bus OPB.
  4. Envoi d'un acquittement quand un message reçu provient d'une écriture bloquante.

La figure 3.7 présente le digramme d'états allégé de cette entité, nous y indiquons les états uniquement et nous expliquons textuellement les conditions de transitions ainsi que le statut de la requête **Adapt2Mod_wrStaus ou Adapt2Mod_rdStatus**.

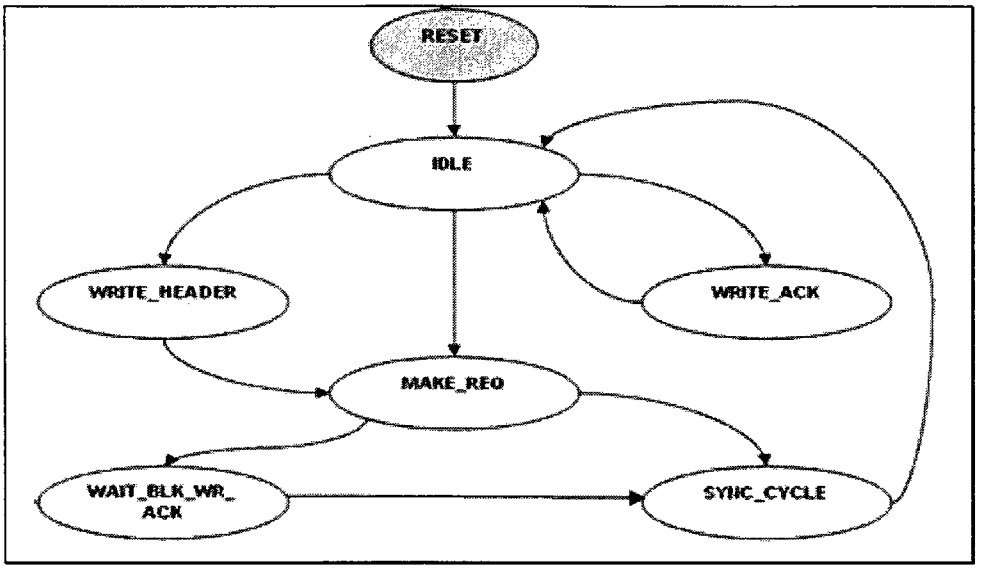

Figure 3. 7 Diagramme d'états simplifié de l'entité « Adapter to bus Request »

Après un état de **Reset**, l'entité passe à l'état **IDLE** et attend les requêtes du module.

- **Écriture à un module** Quand une écriture à un autre module est demandée, l'entité passe à l'état **WRITE _HEADER**. Cet état effectue une demande d'écriture au bus en mode rafale (i.e. un message composé d'un en-tête et d'au moins une donnée). À la réception de l'acquittement du bus, l'état suivant est **MAKE_REQ** dans lequel les données (provenant de l'entité « Send FIFO ») sont envoyées. La requête prend fin à la réception du dernier acquittement (le signal **Bus2IP_MstLastAck**). Si l'écriture est non bloquante le statut **Adapt2Mod_wrStatus** indiquera au module que les données ont été transférées en prenant la valeur **SP_XFER_OK**. L'entité passe alors à l'état **SYNC_CYCLE** et puis à l'état **IDLE.** Si l'écriture est bloquante, l'entité passe à l'état **WAIT _BLK_WR_ACK** dans lequel elle attendra le message d'acquittement de la part du module destinataire de l'écriture. Ce dernier envoie l'acquittement quand il consommera le message.

Dans l'état **WAIT_BLK_WR_ACK**, Si la valeur du signal **Mod2Adapt_TimeOut** est 255, l'attente dans cet état sera indéfinie jusqu'à la réception d'un message d'acquittement de la part du module destinataire de l'écriture.

Si la valeur du signal **Mod2Adapt_TimeOut** est comprise entre 0 et 255, le nombre de cycles d'attente de l'acquittement correspondra à cette valeur. Si le temps d'attente dépasse la valeur du signal **Mod2Adapt_TimeOut**, le module qui a effectué l'écriture bloquante se débloque avec un statut **SP_XFER_TIMEOUT**.

- **Écriture ou lecture d'un périphérique** L'entité passe de l'état **IDLE** à l'état **MAKE_REQ** dans lequel elle initie la lecture ou l'écriture via le bus OPB. Lorsqu'il s'agit de lire ou d'écrire un message contenant plusieurs données. Le transfert est effectué en mode rafale et le bus envoi alors plusieurs acquittements (un acquittement pour chaque donnée transférée). Le statut de la requête **Adapt2Mod_wrStatus ou Adapt2Mod_rdStatus** prend la valeur **SP _XFER_IN_PROG** à chaque acquittement reçu. À la réception du dernier (signal Bus2IP_LastAck), le statut de la requête prend la valeur **SP_XFER_OK** et l'entité passe alors à l'état **SYNC_CYCLE** puis à l'état **IDLE**.

  Dans les deux cas précédents, l'état **SYNC_CYCLE** insère un cycle mort entre deux requêtes afin de synchroniser le module et l'adaptateur.

  Le bus retourne un acquittement pour chaque donnée transférée, dans le cas d'une erreur ou d'un délai d'expiration. L'interface maître de l'IPIF propage l'information jusqu'à l'entité « Adapter to bus Request » via le signal **Bus2IP_MstError** ou **Bus2IP_MstTimeOut**. Dans ces cas la requête prend fin et le statut **Adapt2Mod_wrStatus ou Adapt2Mod_rdStatus** prend la valeur **SP _XFER_ERR** ou **SP_XFER_TIMEOUT**.

- **Entité Bus Read** Quand le module initie une requête d'écriture via l'adaptateur, l'entité « Adapter to Bus Request » initie la même requête au niveau de l'interface maître de l'IPIF qui a été conçue par Xilinx de façon à effectuer en premier une lecture locale des données à envoyer (les données devront être au préalable stockées dans un espace mémoire local à l'adaptateur). Ensuite l'interface maître de l'IPIF initie l'écriture à travers le bus OPB. Ce fonctionnement est illustré par la trace de simulation de la figure 4.3.
  La tâche de l'entité « Bus Read » est de présenter les données à l'interface maître de l'IPIF lors de la lecture locale faite par celle-ci. Selon le cas, il peut s'agir de l'en-tête d'un message, d'une donnée (stockée préalablement dans la FIFO d'envoi) ou d'un message d'acquittement (contenant uniquement un en-tête de 4 octets). Le signal **IP2IP_Addr** sert à préciser l'adresse locale de lecture. L'IPIF convertit la valeur de ce signal en une valeur correspondante pour le signal **Bus2IP_RdCe** encodé à l'interne dans l'IPIF selon le tableau 3.4.

Tableau 3. 4 Encodage de l'adresse de lecture locale de l'IPIF

Valeur de IP2IP_Addr (4 octets)	Valeur de Bus2IP_RdCe
0x0	0x80000000
0x4	0x40000000
0x8	0x20000000
0Xc	0x10000000

L'exemple d'encodage du tableau 3.4 est une partie d'un encodage pour un espace mémoire local composé de 32 adresses.

- **Entité Bus Write** Elle gère la réception des données acheminées par le bus à l'adaptateur lors d'une écriture par un autre module ou d'une lecture d'un périphérique. Dans le cas de données reçues de la part d'un module, elles seront stockées dans une FIFO de réception dédiée à ce module source, alors que dans

le cas de données reçues en réponse à une requête de lecture d'un périphérique, elles seront directement acheminées au module.

Notez qu'étant donné que dans un système, un module peut recevoir des messages de plusieurs modules et que l'ordre de lecture de ces messages n'est pas nécessairement l'ordre dans lequel les messages arrivent, il est possible d'inclure plusieurs FIFO de réception dans un adaptateur au lieu d'une seule FIFO. Le nombre de FIFO doit être connu à l'avance et correspondre au nombre de modules qui écrivent des messages au module en question.

Quand un message arrive dans un adaptateur, il devra être stocké dans une des FIFO de réception. L'identification de ces FIFO de réception est basée sur l'identificateur numérique du module source. Un tableau permet de mémoriser ces identificateurs numériques et le choix de la FIFO est effectué selon le pseudo-code présenté à la figure 3.8.

- Récupérer l'identificateur source à partir de l'entête du message.
- Si l'identificateur n'est pas inscrit dans le tableau des identificateurs des modules sources
    - Allouer une nouvelle FIFO pour le module source en question.
    - Stocker les données dans cette nouvelle FIFO.
- Sinon stocker les données dans la FIFO identifiée par l'identificateur source.

Figure 3. 8 Pseudo-code pour allouer une FIFO

Cette entité est connectée à l'interface esclave de l'IPIF qui lui passe les données acheminées par le bus, ainsi qu'à l'entité « Receive FIFO » où les messages seront stockés.

- **Entité Receive FIFO** Elle instancie les FIFO de réception des messages qui seront envoyés par les autres modules. Le nombre de FIFO est un paramètre générique de l'adaptateur et devra être connu à l'avance. L'écriture dans ces FIFO est effectuée par l'entité « Bus Write », alors que leur lecture est effectuée par l'entité « Module Read ».

- **Entité Module Read** La figure 3.9 présente un diagramme d'états simplifié de cette entité dans lequel nous présentons uniquement les états.

  Un module lit les messages qui lui ont été envoyés par les autres modules en initiant une requête de lecture via le signal **Mod2Adapt_xferType**. Cette lecture est locale entre le module et son adaptateur et ne requiert par l'accès au bus OPB. Le signal **Mod2Adapt_DestID** porte la valeur de l'identificateur numérique du module qui est censé avoir envoyé le message. La FIFO de réception est retrouvée (état FIND_FIFO_INDEX) grâce à cette valeur, si elle contient des données. L'en-tête est analysé pour déterminer si le message provient d'une écriture bloquante. Les données sont lues immédiatement (état READ_DATA) si l'écriture n'est pas bloquante. Dans le cas contraire, l'envoi d'un acquittement est demandé (état WAIT_BLK_WR_ACK_DONE) à l'entité «**Adapter to bus Request** » du même adaptateur et les données sont ensuite lues.

  Dans le cas où la FIFO ne contient pas de données, le statut de retour et l'achèvement de la requête dépend du type de lecture (bloquante ou non) défini selon la valeur du signal **Mod2Adapt_timeOut**. Trois cas de figure sont possibles:

  - Si cette valeur est nulle, la requête s'achève immédiatement, le statut de la lecture sera **SP_NO_DATA** et l'entité passe de l'état FIND_FIFO_INDEX à l'état IDLE.
  - Si cette valeur est 255, il y aura une attente jusqu'à ce que des données soient envoyées par le module source. L'entité reste dans l'état FIND_FIFO_INDEX.

- Si cette valeur est comprise entre 0 et 255, il y aura une attente sur les données ou sur un nombre de cycles égal à la valeur spécifiée par le signal **Mod2Adapt_timeOut**. Le statut de la requête prend la valeur **SP_ _XFER_TIMEOUT** si le temps d'attente est écoulé sans que les données aient été reçues et l'entité passe de l'état FIND_FIFO_INXED à l'état IDLE

Quand l'entité est dans l'état READ_DATA, un nombre de lectures fixé par le signal Mod2Adapt_xferSize est effectué. À chaque cycle une donnée est lue à partir de la FIFO de réception et renvoyé au module via le signal **Adapt2Mod_dataRd**. Le statut de la requête prend la valeur **SP_XFER_IN_PROG** ou **SP _XFER_OK** quand la dernière donnée est lue.

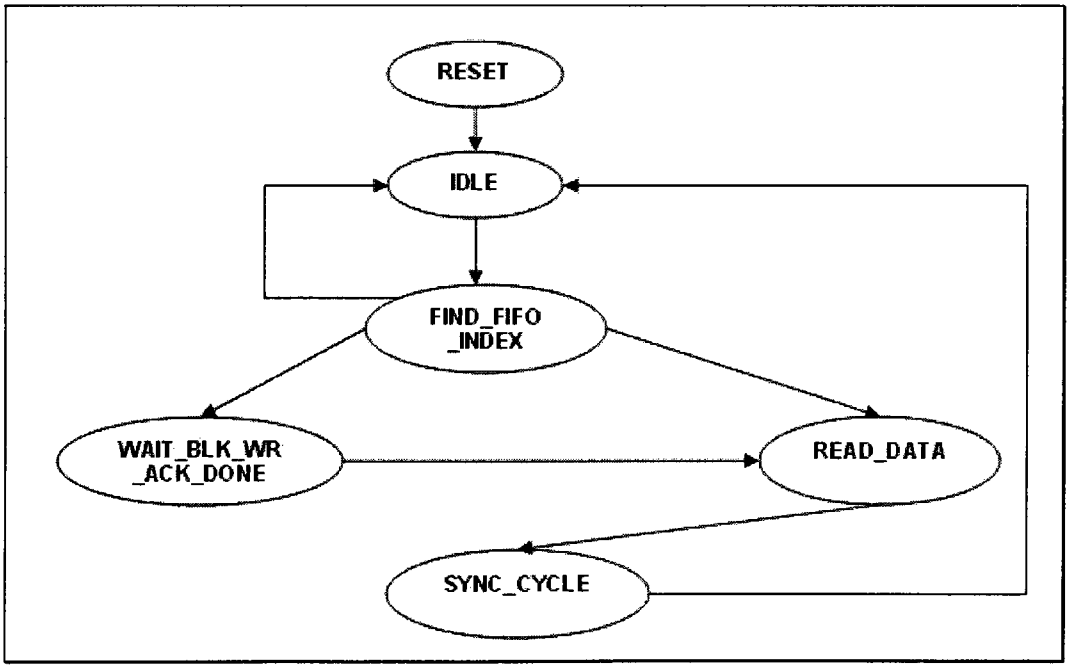

Figure 3. 9 Diagramme d'états simplifié de l'entité « Module Read »

3.3.4 Adaptateur des communications entre le matériel et le logiciel

Un message écrit par un module matériel vers un module logiciel sera acheminé par le bus OPB vers l'adaptateur des communications entre le matériel et le logiciel (HwSwCom). Cette composante connectée au bus en tant qu'esclave, est basée sur l'IPIF de Xilinx dans sa version esclave uniquement [40]. La figure 3.10 présente le schéma bloc de la composante HwSwCom.

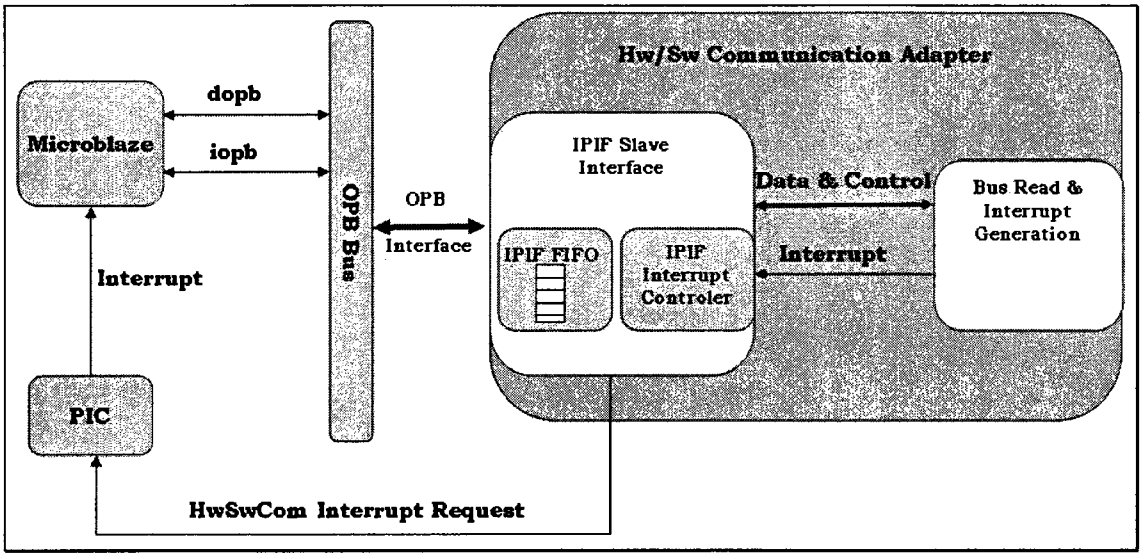

Figure 3. 10 Schéma bloc de l'adaptateur des communications matériel/logiciel

Un message destiné à un module logiciel est écrit par le bus OPB dans la FIFO de l'IPIF qui fait partie de l'interface esclave de l'IPIF. Cette interface dispose aussi d'un contrôleur d'interruptions dont les registres [40] sont configurés lors de l'initialisation de la partie logicielle du système.

L'entité « Bus Read & Interrupt Generation » que nous avons implémentée en VHDL a deux rôles :

1. Générer une interruption quand des messages envoyés par des modules matériels sont stockés dans la FIFO de l'IPIF. Le signal d'interruption de cette entité est connecté au contrôleur d'interruptions de l'IPIF, qui à son tour génère un signal d'interruption connecté au gestionnaire d'interruptions

du système (PIC sur la figure 3.10). Le PIC lève une interruption au niveau du processeur Microblaze et le traitement de l'interruption est lancé au niveau logiciel. Plus précisément, la partie logicielle du système démarre une tâche logicielle gérée au niveau du RTOS qui prend en charge la lecture des données destinées à un module logiciel, stockées dans la FIFO de la composante HwSwCom. La lecture entre le Microblaze et le bus OPB est effectuée par le biais de l'interface de données (**dopb** sur la figure 3.10).
L'en-tête du message est lu en premier, il est ensuite analysé pour déterminer l'identificateur du module destinataire, les options de la communication (bloquante, non bloquante, ou message d'acquittement) et le nombre de données du message. Ensuite, les données sont lues et sont acheminées dans une FIFO (en logiciel) de réception dédiée au module logiciel destinataire.

2. Répondre aux requêtes de lecture initiées par le processeur Microblaze suite à une interruption qui signale que des données ont été reçues du matériel. Le mécanisme implémenté est le même que celui expliqué précédemment (annexe C, Interface esclave de l'IPIF) et qui utilise l'interface esclave de l'IPIF en l'occurrence les signaux **Bus2IP_RdCe, IP2Bus_Data**, et **IP2Bus_RdAck**

3.4 Comparaison avec les autres travaux de recherche

Nous avons présenté au chapitre 1 (section 1.5) une revue des travaux de raffinement des communications d'un système conçu à haut niveau. Dans les sections précédentes nous avons exposé notre approche pour implémenter au niveau RTL le protocole de communication utilisé dans Space.
Notre approche est similaire à celle décrite dans [3] sur le principe général d'utilisation des adaptateurs pour connecter des modules matériels sur une architecture donnée. C'est aussi le cas pour les communications entre le matériel et le logiciel.
Cependant certaines différences existent entre les deux approches.

- Le flot de conception proposé dans [3] intègre un raffinement au niveau des signaux en utilisant SystemC RTL et le protocole OCP et ceci avant d'arriver au modèle de l'implémentation. Dans notre cas, nous passons du niveau BCA au modèle de l'implémentation.
- Dans [3], l'adaptateur d'un module matériel (appelé Accessor) utilise l'interface OCP du côté du module matériel. Alors que dans notre cas, l'interface entre un module et son adaptateur est le résultat du raffinement du protocole de communication utilisé à haut niveau dans Space.
- Dans [3], les communications du matériel vers le logiciel et vice-versa passent par l'adaptateur des communications entre le matériel et le logiciel. Ce dernier est connecté au processeur par un lien point à point. Dans notre cas, seules les communications du matériel vers le logiciel passent par l'adaptateur des communications entre le matériel et le logiciel. Alors que les communications du logiciel vers le matériel passent directement par le bus de l'architecture.

Chapitre 4 Résultats, discussions et améliorations

Nous présentons dans ce chapitre les résultats liés au fonctionnement des adaptateurs conçus en VHDL et utilisés pour implémenter le protocole de communication de Space sur un FPGA Virtex2 de Xilinx.

Nous définissons les latences de communications du matériel au matériel, du matériel au logiciel, du logiciel au matériel, et du logiciel au logiciel.

Nous proposons deux versions de l'adaptateur d'un module matériel. La première version utilise la composante IPIF de Xilinx version maître/esclave. La deuxième version utilise une version esclave de l'IPIF et une interface maître que nous avons conçue. Cette dernière version nous permet de réduire la latence d'une écriture d'environs 40 % par rapport à la première version.

Nous analysons les latences générées par le logiciel lors d'une communication matériel-logiciel qui utilise la composante HwSwCom décrite au paragraphe 3.3.4. Pour réduire ces latences, nous proposons une nouvelle version de la composante HwSwCom qui communique avec le processeur Microblaze par le lien FSL [25], et qui n'utilise pas le mécanisme d'interruptions contrairement à la version originale de cette composante.

4.1 Quelques restrictions de fonctionnement de l'IPIF

La conception de l'adaptateur d'un module matériel se base sur la composante IPIF version 2.00.h de Xilinx. Nous soulevons deux restrictions de fonctionnement dans cette version de l'IPIF.

- **Première restriction** L'interface maître de l'IPIF a été conçue par Xilinx pour effectuer par défaut le transfert de 8 données en mode rafale. Or cette valeur par défaut ne correspond pas toujours aux besoins des applications. Pour y remédier nous avons ajouté un signal **Mst_Num** qui permet de choisir le nombre de données à transférer. La figure 4.1 présente le code source original (4.1a) et le code source modifié (4.1b).
- **Deuxième restriction** La documentation de Xilinx conseille l'utilisation des signaux Bus2IP_WrCe et Bus2IP_RdCe (annexe C) pour détecter le respectivement le début d'une écriture et d'une lecture. Or, ces signaux gardent la même valeur durant les deux premiers cycles quand il s'agit d'un transfert en mode rafale. Ceci cause l'envoi de la première donnée deux fois, doublant ainsi sa présence dans la FIFO de réception de l'adaptateur destinataire. Pour y remédier, nous avons utilisé les signaux Bus2IP_WrReq ou Bus2IP_RdReq et Bus2IP_Addr pour détecter une demande d'écriture ou de lecture.

a/ Code source original

```
Set_Value_of_MA2SA_Num_PROCESS: process (DMA_sel_IP_sel_not,
IP2Bus_MstBurst, DMA2Bus_MstNum)
begin
if(DMA_sel_IP_sel_not = '0') then
MA2SA_Num_i <= (others => '0');
MA2SA_Num_i(MA2SA_Num'right-3) <=     IP2Bus_MstBurst;
MA2SA_Num_i(MA2SA_Num'right  ) <= not IP2Bus_MstBurst;
else
MA2SA_Num_i <= DMA2Bus_MstNum;
end if;
end process Set_Value_of_MA2SA_Num_PROCESS;
MA2SA_Num <= MA2SA_Num_i;
```

b/ Code source modifié

```
Set_Value_of_MA2SA_Num_PROCESS: process (DMA_sel_IP_sel_not,
IP2Bus_MstBurst, DMA2Bus_MstNum,Mst_Num )
begin
if(DMA_sel_IP_sel_not = '0') then
MA2SA_Num_i <= Mst_Num ; --(others => '0');
-- MA2SA_Num_i(MA2SA_Num'right-3) <=     IP2Bus_MstBurst;
-- MA2SA_Num_i(MA2SA_Num'right  ) <= not IP2Bus_MstBurst;
else
MA2SA_Num_i <= DMA2Bus_MstNum;
end if;
end process Set_Value_of_MA2SA_Num_PROCESS;
MA2SA_Num <= MA2SA_Num_i;
```

Figure 4. 1 Modification de l'IPIF de Xilinx (cas du transfert en rafale)

4.2 Latences des communications matériel-matériel

Dans ce qui suit, nous présentons une application simple de communication matériel-matériel. Deux versions d'adaptateurs sont ensuite comparées à l'aide de cette application. Cette comparaison est réalisée à l'aide d'un calcul précis sur la latence de communication.

4.2.1 Présentation de l'application

Pour vérifier le fonctionnement de l'adaptateur d'un module matériel et déterminer les valeurs des différentes latences définies dans la section suivante, nous avons implémenté

une application simple dans laquelle nous avons 4 modules en matériel. Chaque module est connecté au bus OPB à travers un adaptateur. La figure 4.2 présente l'architecture de cette application.

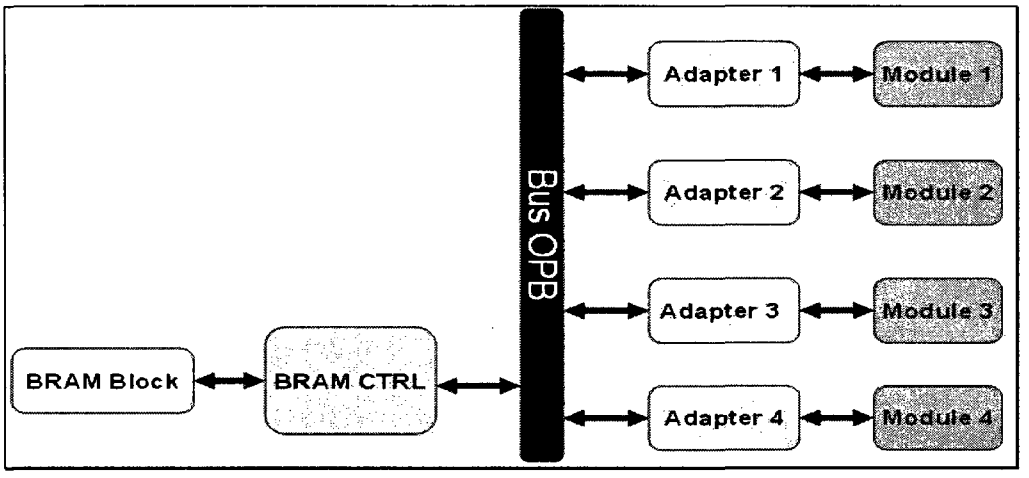

Figure 4. 2 Application d'illustration des latences de communication matérielle

Dans cette application le module 1 écrit en continu au module 2. Plus précisément, il écrit des messages de 5 données chacun en mode rafale. Le module 2 effectue donc la lecture des données qui lui ont été envoyées par le module 1. Finalement, le module 3 écrit 5 données en mode rafale dans la mémoire **BRAM Block**, alors que le module 4 lit les 5 données à partir de cette mémoire.

Cette application simple nous permet de valider rapidement le fonctionnement de l'adaptateur selon le protocole utilisé dans Space. Un projet a été crée dans l'environnement de développement XPS de Xilinx (Xilinx Platform Studio) et la vérification du fonctionnement a été faite par la simulation comportementale et structurelle sur l'outil ModelSim, ainsi qu'au niveau de la carte de prototype (carte multimédia de Xilinx).

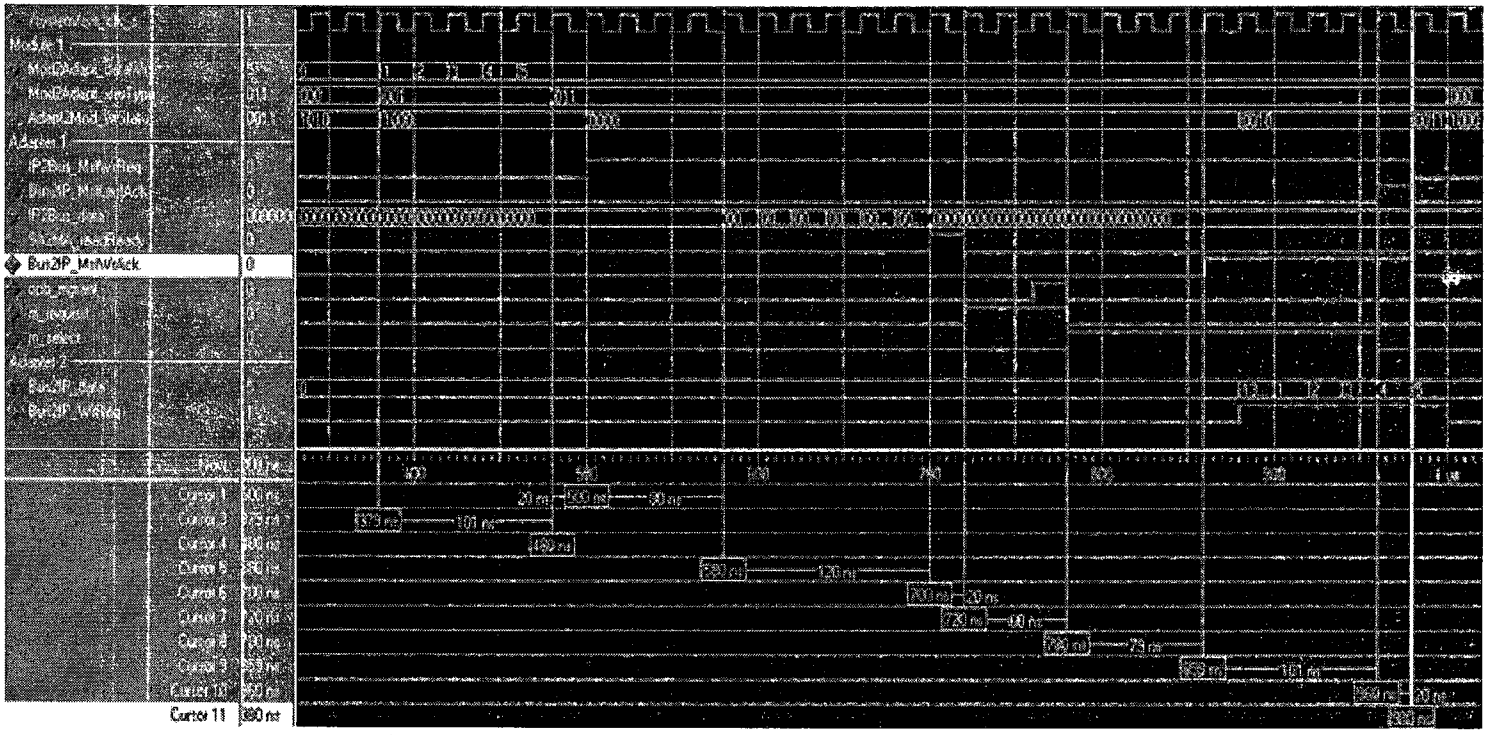

Figure 4. 3 Trace de simulation de l'écriture non bloquante

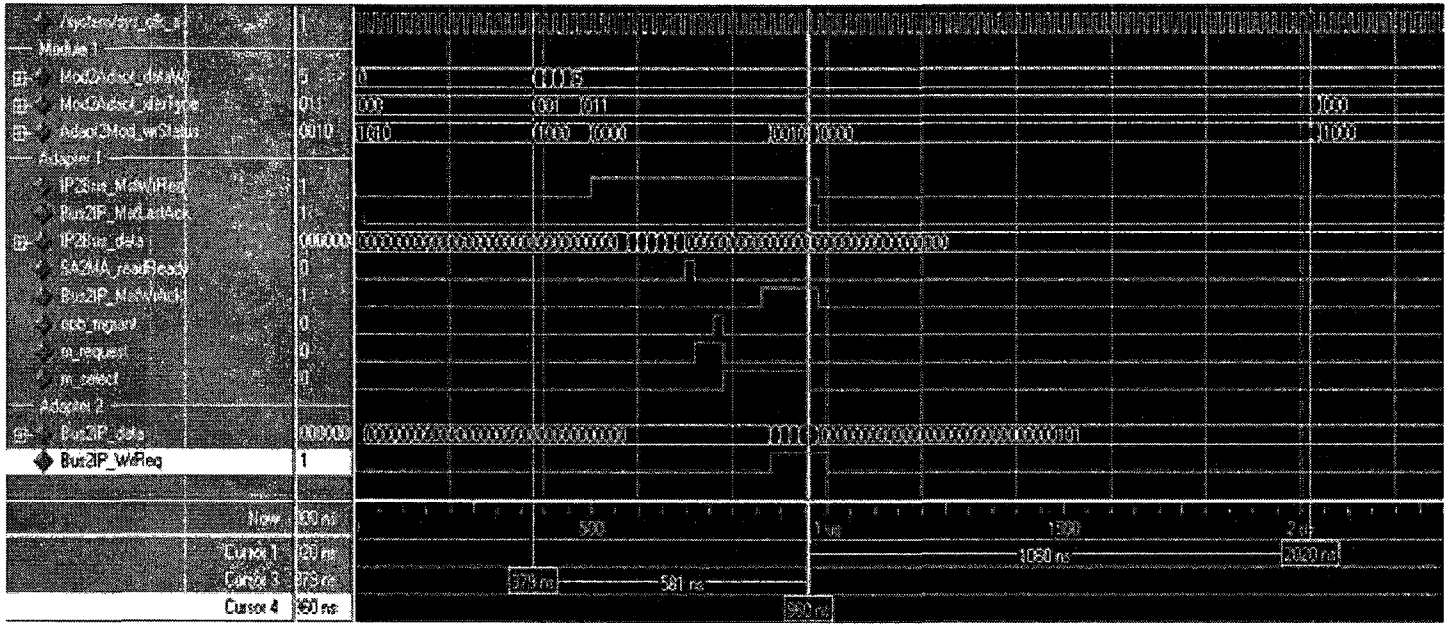

Figure 4. 4 Trace de simulation de l'écriture bloquante

4.2.2 Latences des communications de l'adaptateur version no 1

La version no 1 de l'adaptateur d'un module matériel a été conçue autour de l'IPIF de Xilinx.

Afin d'illustrer les différentes latences qui ont lieu lors d'une transaction, nous commençons par les définir et nous établissons la valeur de chacune de ces latences à l'aide des figure 4.3 et 4.4 représentant le fonctionnement de l'adaptateur pour une écriture bloquante et non bloquante, respectivement.

- **Latence d'écriture de la donnée sur la FIFO d'envoi (L0)** Nous avons expliqué à la section 3.3.2 que l'initiation de l'écriture est précédée par l'écriture des données par le module dans la FIFO d'envoi au niveau de l'adaptateur. Cette opération est effectuée quand la valeur du signal **Mod2Adapt_xferType est « 001 »**.

  Sur le diagramme temporel de la figure 4.3 cette latence est représentée par l'intervalle entre les curseurs 3 et 4.

- **Latence entre le module et son adaptateur (L1)** Elle représente la latence entre le moment où le module initie une requête (signal **Mod2Adapt_xferType**) et le moment où l'adaptateur prend en compte la requête en question. Cette requête peut être locale entre le module et l'adaptateur tel que la lecture des données reçues d'un autre module ou à travers le bus tel que l'envoi de données à un autre module. Également, cette latence peut correspondre à l'écriture et la lecture de données à partir d'un périphérique.

  À la figure 4.3, cette latence est représentée par l'intervalle entre les curseurs 4 et 1.

- **Latence avant l'opération de lecture interne (L2)** L'interface maître de l'IPIF (annexe C) effectue une lecture interne des données à envoyer. Cette lecture est effectuée à travers l'interface esclave de l'IPIF sur un espace mémoire interne à

l'adaptateur. L2 représente la latence entre le moment où l'interface de L'IPIF a pris en compte la demande de l'écriture et le déclenchement de la lecture interne des données.

À la figure 4.3, cette latence est représentée par l'intervalle entre les curseurs 1 et 5.

- **Latence de la lecture interne (L3)** C'est le nombre de cycles nécessaires à l'interface maître de l'IPIF pour lire localement une donnée avant de l'envoyer.

  À la figure 4.3, cette latence est représentée par l'intervalle entre les curseurs 5 et 6.

- **Latence entre la fin de la lecture interne et la demande du bus (L4)** C'est le nombre de cycles entre la fin de la lecture interne et la demande d'accès au bus par l'interface maître de l'IPIF (activation du signal **m_request**).

  À la figure 4.3, cette latence est représentée par l'intervalle entre les curseurs 6 et 7.

- **Latence entre la demande et l'obtention d'accès au bus (L5)** C'est le nombre de cycles entre l'activation du signal **m_request** et l'activation du signal **m_select**.

  À la figure 4.3, cette latence est représentée par l'intervalle entre les curseurs 7 et 8.

- **Latence entre l'obtention du bus le premier acquittement (L6)** C'est le nombre de cycles entre l'activation du signal **m_select** et la réception du premier acquittement **(Bus2ip_mstRdAck ou Bus2IP_mstWrAck)**.

  À la figure 4.3, cette latence est représentée par l'intervalle entre les curseurs 8 et 9.

- **Latence de transfert de la donnée par le bus OPB (L7)** C'est le nombre de cycles nécessaire au bus OPB pour transférer une donnée.
  À la figure 4.3, cette latence est représentée par l'intervalle entre les curseurs 9 et 10.

- **Latence d'acquittement de la requête entre l'adaptateur et le module (L8)** C'est le nombre de cycles nécessaires pour acquitter la requête du module matériel par son adaptateur après que le dernier transfert a eu lieu. Cet acquittement entre l'adaptateur et le module est effectué à travers le signal **Adapt2Mod_wrStatus** (pour une écriture) ou le signal **Adapt2Mod_rdStatus** (pour une lecture). Il doit prendre la valeur « 0011», équivalente à SP_XFER_OK (paragraphe 3.3.1).
  À la figure 4.3, cette latence est représentée par l'intervalle entre les curseurs 10 et 11.

- **Latence d'attente d'acquittement pour une écriture bloquante (L9)** Elle correspond au nombre de cycles qu'un module devra attendre pour recevoir l'acquittement suite à une écriture bloquante qu'il a effectuée vers un autre module. Cette latence est variable et dépend du rythme de lecture du module destinataire (qui devra envoyer un acquittement sur l'écriture bloquante) et de l'arbitration (i.e. la priorité d'accès au bus OPB).
  Sur le diagramme temporel de la figure 4.4, cette latence est représentée par l'intervalle entre les curseurs 4 et 1.

- **Latence d'attente pour accéder au bus (L10)** Nous rappelons que le bus OPB tel que conçu par Xilinx supporte deux modes de priorités. Les termes utilisés par Xilinx pour désigner ces modes sont la priorité dynamique (de l'anglais Round Robin) et la priorité fixe. La latence L10 représente le nombre de cycles d'attente pendant que d'autres maîtres plus prioritaires accèdent au bus OPB.

Nous définissons ensuite les latences liées à l'opération de lecture d'un message par un module matériel. La lecture en question est locale au niveau de l'adaptateur du module, le message à lire est stocké dans une FIFO de réception.

- **Latence de recherche de la FIFO de réception (L11)** Un module matériel peut recevoir des messages de plusieurs modules matériels ou logiciels. Nous avons vu à la section 3.3.3 (description de l'entité **Bus Write**) que ces messages sont stockés dans une FIFO de réception au niveau de l'adaptateur. Rappelons-nous que plusieurs FIFO de réception peuvent être intégrées à l'adaptateur, chacune étant dédiée pour stocker les données d'un module en particulier. Quand un module initie l'opération de lecture vers un autre module, cette requête est traitée localement par l'adaptateur et ne requiert pas l'accès au bus. La recherche de l'index de la FIFO de réception correspond à la latence **L11**. Cet index est retrouvé à partir des identificateurs du module source et destination de la requête de lecture.

- **Latence de lecture des données à partir de la FIFO de réception (L12)** C'est le nombre de cycles nécessaires pour lire une donnée à partir de la FIFO de réception après avoir trouvé l'index lors de l'étape correspondante à L11.

- **Latence d'attente d'envoi d'acquittement lorsque le message lue est bloquant (L13)** Nous avons vu au à la section 3.3.3 (description de l'entité **Module Read**) qu'un module peut effectuer une écriture bloquante d'un message vers un autre module. Le module destinataire analyse l'en-tête du message reçu lorsqu'il effectue la lecture. Quand l'en-tête du message indique que ce dernier a été envoyé par une écriture bloquante, le module qui effectue la lecture devra alors envoyer un acquittement à travers son adaptateur et ce avant de lire les données.

La latence liée à l'envoi de l'acquittement n'est pas constante et dépend de la disponibilité de l'adaptateur et du bus.

4.2.3 Illustration du calcul de la latence

Le tableau 4.1 présente le sommaire des latences de communications pour les différentes requêtes supportées par l'adaptateur d'un module matériel. Nous avons testés les cas suivants :

- Écriture non bloquante par un module matériel vers un module matériel (R1)
- Écriture bloquante par un module matériel vers un autre module matériel (R2)
- Écriture par un module matériel vers un périphérique (R3)
- Lecture par un module matériel vers un périphérique (R4)
- Lecture par un module matériel vers un autre module (R5) lors d'une écriture non bloquante

Légende du tableau 4.1

Les latences sont données en cycles.

*	En cycles/ donnée
(N+1)	Nombre de données du message plus un en-tête
NA	Ne s'applique pas
D	Nombre de cycles d'attente pendant que le bus traite les requêtes des maîtres les plus prioritaires.
W	Nombre de cycles d'attente pour recevoir l'acquittement suite à une écriture bloquante.
ξ	Nombre de cycles d'attente quand l'interface esclave de l'IPIF est occupée à traiter une requête de l'arbitre du bus OPB.
X:	Nombre de cycles d'attente d'envoi de l'acquittement quand le message lu provient d'une écriture bloquante

Tableau 4. 1 Sommaire des latences de communication de l'adaptateur version 1

	R1	R2	R3	R4	R5
L0	1*	1*	1*	NA	NA
L1	1	1	1	1	1
L2	(4+ξ)	(4+ξ)	(4+ξ)	0	NA
L3	1*	1*	1*	0	NA
L4	1	1	1	1	NA
L5	3	3	3	3	NA
L6	4	4	3	4	NA
L7	1*	1*	1*	1*	NA
L8	1	1	1	1	1
L9	NA	**W**	NA	NA	NA
L10	D	D	D	D	NA
L11	NA	NA	NA	NA	2
L12	NA	NA	NA	NA	2+1*
L13	NA	NA	NA	NA	**X**
Latence totale pour un message de N données	**N + 14 + (N+ 1) + (N+1) + ξ + D = 3N +16 + ξ+D**	**N + 14 + (N+ 1) + (N+1) + ξ + W +D = 3N +16 + W + ξ + D**	**N + 13 + N + N + ξ +D = 3N +13 + ξ + D**	**= N + 10 + D**	**= N + 6 + X**

4.2.4 Latences de communication de l'adaptateur version no 2

Nous avons vu dans la section précédente les différentes latences pour des communications entre deux modules matériels ainsi que pour des communications entre un module matériel et un périphérique. On remarque que les latences (L2 + L3) dues au fonctionnement de l'IPIF (version maître/esclave) de Xilinx représentent environ la moitié de la latence totale pour chaque type de communication nécessitant l'accès au bus OPB.

Afin de réduire la latence totale, nous avons donc conçu une deuxième version de l'adaptateur d'un module matériel. Dans cette nouvelle version, nous avons utilisé l'IPIF de Xilinx dans sa version esclave uniquement et nous avons implémenté une entité maître du bus OPB. Le reste de la logique de l'adaptateur n'a pas été changé.

L'entité maître du bus OPB que nous avons conçue implémente deux choses : 1) le protocole du bus OPB tel que décrit à l'annexe A (**Requête d'un maître**) et 2) un minimum de logique afin d'assurer le bon fonctionnement du protocole de communication de Space, en l'occurrence le transfert de données en mode rafale

À l'aide de notre exemple de la section 4.1.1, nous évaluons les nouvelles latences de communications. Les résultats sont présentés au tableau 4.2.

En comparant les résultats du tableau 4.1 et 4.2, nous constatons une différence importante pour les opérations nécessitant l'accès au bus OPB et utilisant les services de l'IPIF. À titre d'exemple prenons le cas d'une écriture non bloquante de 5 données par un module vers un autre module. Les latences suivantes sont observables:

- **Adaptateur version 1** : latence = $3N + 16 + \xi + D = 31 + \xi + D$ cycles.
- **Adaptateur version 2** : latence = $2N + 9 + D = 19 + D$ cycles.

En considérant le cas parfait ($\xi = 0$ et $D = 0$), nous avons 12 cycles de moins entre la version no 2 et la version no 1.

Tableau 4. 2 Sommaire des latences de communication de l'adaptateur version no 2

	R1	R2	R3	R4	R5
L0	1*	1*	1*	NA	NA
L1	1	1	1	1	1
L2	0	0	0	0	NA
L3	0	0	0	0	NA
L4	1	1	1	1	NA
L5	3	3	3	3	NA
L6	2	2	2	2	NA
L7	1*	1*	1*	1*	NA
L8	1	1	1	1	1
L9	NA	**W**	NA	NA	NA
L10	D	D	D	D	NA
L11	NA	NA	NA	NA	2
L12	NA	NA	NA	NA	2+1*
L13	NA	NA	NA	NA	**X**
Latence totale pour un message de N données	**N + 8 + (N+1) + D = 2N +9 + D**	**N + 8 + (N+1) + D +W = 2N +9 + D +W**	**N + 8 + N +D = 2N + 8 + D**	**= N + 8 + D**	**= N + 6 + X**

Remarque :

Les latences pour les cas R4 et R5 peuvent varier au niveau de la valeur de L6. En effet, les périphériques présents dans un système ont des latences différentes, en l'occurrence le temps nécessaire pour répondre à une requête de lecture ou d'écriture.

4.2.5 Latence de l'adaptateur par rapport à l'utilisation directe de l'IPIF

Les adaptateurs conçus dans le cadre de notre travail ajoutent une latence de deux cycles par rapport à l'utilisation directe de l'IPIF de Xilinx pour faire communiquer deux composantes d'une architecture utilisant le bus OPB.

La synchronisation, sur les fronts de l'horloge, des signaux entre un module et son adaptateur est la raison qui explique les deux cycles de latence ajoutés par l'adaptateur. Nous avons jugés plus sur de synchroniser les signaux pour assurer leur stabilité. Le premier cycle est ajouté lors d'une demande de requête par le module (signal Mod2Adapt_xferType), et le deuxième cycle est ajouté lors de l'acquittement de la requête (signal Adapt2Mod_wrStatus ou Adapt2Mod_rdStatus).

4.3 Latence des communications entre le matériel et le logiciel

Au chapitre 3 (section 3.3.4), nous avons vu que pour les communications allant du matériel au logiciel sont effectuées via des requêtes de lecture ou d'écriture de données à l'aide d'un module matériel vers un module logiciel. Ce type de communication fait intervenir la composante HwSwCom. À l'opposé, pour les communications allant du logiciel au matériel, il s'agit de requêtes de lecture ou d'écriture de données par un module logiciel vers un module matériel ou vers un périphérique.

Dans ce qui suit, la section 4.3.1 discute du compromis sur le choix du type de mémoires utilisé pour les différents tests, alors que la section 4.3.2 présente une technique utilisée pour calculer le temps d'exécution au niveau logiciel. Les notions présentées dans ces 2 sections seront utiles pour la compréhension des sections 4.3.3 à 4.3.5 qui traitent du calcul de latence pour les différents types de communication. Finalement la section 4.3.6

jette les bases d'une nouvelle architecture qui améliore les performances obtenues aux sections 4.3.3 et 4.3.5.

4.3.1 Considération sur la taille des mémoires

La taille totale de la mémoire BRAM interne disponible varie selon le type de FPGA utilisé : 126 koctets dans le cas d'un Virtex 2 XC2V2000 et 306 koctets dans le cas d'un Virtex 2p VP30. Lorsque le logiciel est chargé dans une mémoire BRAM connectée directement au processeur Microblaze à travers le bus LMB, les latences dues aux accès mémoire, pour les instructions et les données, sont minimales. En effet, le bus LMB supporte un seul maître, et offre une connexion très rapide à la mémoire BRAM.

Toutefois, le code des librairies logicielles de Space, des pilotes des périphériques, du RTOS et des modules logiciels génère un exécutable dont la taille typique est d'environ 250 koctets. Il est donc souvent difficile d'avoir un seul bloc de mémoire interne directement sur le FPGA. Notez que la mémoire BRAM, disponible sous forme de blocs dont la taille maximale de chacun est de 64 koctets, peut être utilisée par allocation de plusieurs blocs contigus de BRAM. Par contre, une trop grande quantité de mémoire allouée au logiciel risque de causer une pénurie de mémoire pour les besoins en matériel.

Lorsque la mémoire BRAM ne peut être utilisée, l'exécutable généré pour la partition logicielle est chargé et exécuté à partir d'une mémoire externe au FPGA.

4.3.2 Méthode de mesure du temps d'exécution en logiciel

Afin de déterminer les latences de communication du matériel vers le logiciel et vis-versa, nous avons utilisé les résultats de la simulation sur ModelSim quand les données sont échangées entre deux composantes matérielles et nous avons utilisé une composante *minuterie* en matériel dont le fonctionnement permet de compter le nombre de cycles d'exécution d'une section choisie du code logiciel. En effet, un compteur interne à la minuterie est initialisé au début à une certaine valeur, et est décrémenté à chaque coup de l'horloge du système.

À partir du code logiciel, il est possible de récupérer la valeur du compteur de la minuterie (figure 4.5). Par conséquent, pour déterminer le temps d'exécution d'une section de code logiciel, nous récupérons cette valeur avant et après l'exécution de la section en question. Afin de minimiser l'incertitude sur le résultat obtenu, il faut éviter toute interruption du processeur pendant qu'il exécute la section de code en question.
Le résultat obtenu par cette méthode ne tient pas compte du temps nécessaire à la lecture de la valeur du compteur. Cette lecture requiert l'accès au bus OPB pour accéder aux registres de la composante minuterie. Nous avons déterminé en simulation sur ModelSim que cette lecture prend 6 cycles.

```
        instruction
        instruction
        instruction
// début de la section de code pour laquelle on détermine
// le temps d'exécution
count1 = XTmrCtr_mGetTimerCounterReg( TIMER_ADDR )
        instruction
        instruction
        instruction
// fin de la section de code pour laquelle on détermine
// le temps d'exécution

count2 = XTmrCtr_mGetTimerCounterReg( TIMER_ADDR )
```

Figure 4. 5 Mesure du temps d'exécution d'une section de code logiciel

4.3.3 Communication du matériel vers le logiciel

La lecture par un module matériel de données envoyées par un module logiciel est locale et s'effectue au niveau l'adaptateur du module matériel. La latence de la lecture est donc équivalente à la valeur indiquée au tableau 4.1 ou 4.2 (cas R5) selon la version de l'adaptateur (version 1 ou version 2).

Pour ce qui est de l'écriture de données par un module matériel vers un module logiciel, elle comporte deux phases :

- **Phase** 1 L'envoi des données du module matériel vers l'adaptateur, plus précisément la composante HwSwCom. Cette phase a la même latence que le cas R1 indiqué aux tableaux 4.1 et 4.2 selon la version de l'adaptateur.
- **Phase 2** La composante HwSwCom signale au processeur, par un mécanisme d'interruption, que des données ont été reçues. Le processeur traite l'interruption et récupère les données pour les stocker dans une FIFO (géré par logiciel) dédiée au module logiciel destinataire.

Pour déterminer la latence totale de cette écriture, nous décomposons la latence de la manière suivante:

- L14 : latence de transfert du message du module matériel vers la composante HwSwCom.
- L15 : latence entre le moment de la réception du message par la composante HwSwCom et le moment où l'interruption est signalée au processeur Microblaze.
- L16 (latence due au changement de contexte pour traiter l'interruption) : latence entre le moment où le processeur prend en considération la demande d'interruption (lorsque l'exécution de la routine de traitement des interruptions commence) et le moment où le processeur commence la lecture des données reçues au niveau de la composante HwSwCom.
- L17 : latence de lecture des données par le processeur à travers le bus OPB.
- L18 : latence de transfert des données lues vers la FIFO dédiée au module logiciel destinataire.

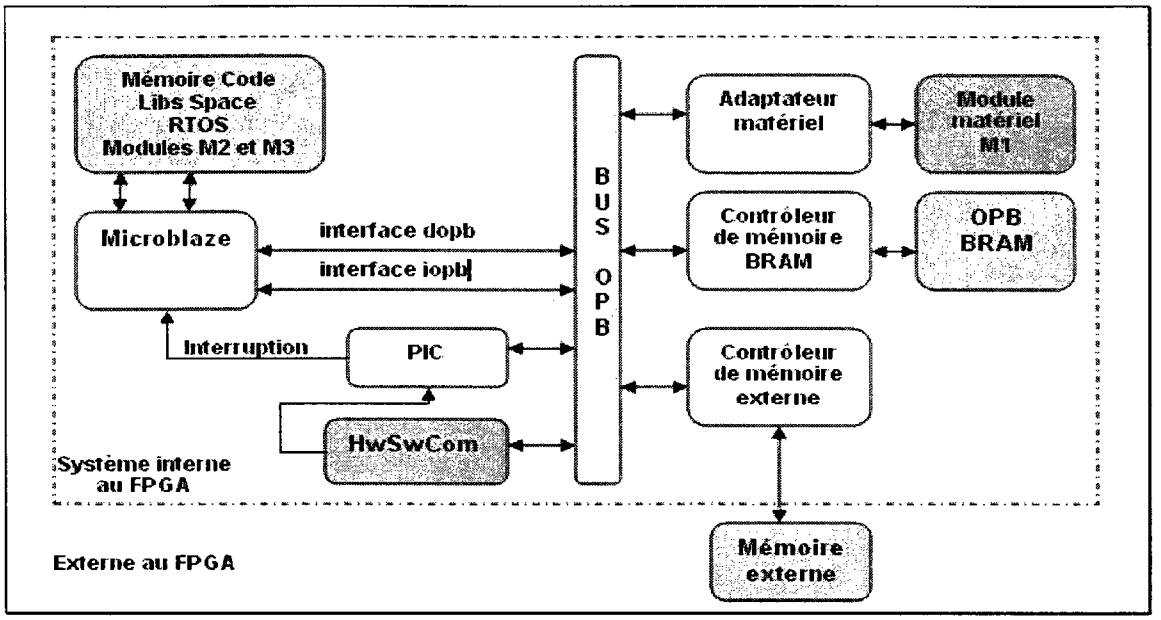

Figure 4. 6 Système utilisé pour déterminer les latences de communication matériel-logiciel

La figure 4.6 présente le système utilisé afin de déterminer ces latences. Le module matériel M1 envoie une donnée à une certaine fréquence au module logiciel M2. Le module M2 consomme la donnée et l'affiche sur HyperTerminal à travers le lien série RS232.

Nous avons déterminé les valeurs L14 et L15 par simulation sur ModelSim, alors que nous avons utilisé la méthode décrite à la section 4.3.2 pour déterminer les valeurs de L16, L17 et L18.

Le tableau 4.3 présente les différentes latences de l'opération d'écriture d'un message contenant une donnée de 4 octets et un en-tête de 4 octets par un module matériel vers un module logiciel selon plusieurs cas de configuration (e.g. mémoire interne, externe, utilisation d'une mémoire cache, etc.). Pour chacun des cas, la lecture des données par le processeur se fait à travers le bus OPB et la composante HwSwCom (version 1) utilise le mécanisme d'interruption pour signaler au processeur que des données ont été reçues. Dans ce qui suit voici le détail de chacun de ces cas :

1. **Cas 1** Le code logiciel est chargé et exécuté à partir d'une mémoire externe de type ZBT SRAM que l'on trouve sur la carte multimédia de Xilinx.
2. **Cas 2** Le code logiciel est chargé et exécuté à partir d'une mémoire externe de type DDR SDRAM que l'on trouve sur la carte de prototype XUP de Xilinx.
3. **Cas 3** Le code logiciel est chargé et exécuté à partir d'une mémoire BRAM interne au FPGA (plusieurs blocks contigus de BRAM afin de contenir tout l'exécutable logiciel).
4. **Cas 4** Le code logiciel est chargé et exécuté à partir d'une mémoire externe DDR SDRAM, avec utilisation d'une cache de 8 koctets pour les instructions et une cache de 8 koctets pour les données
5. **Cas 5** L'exécutable produit par le compilateur croisé **mb-gcc** pour le processeur Microblaze est composé de plusieurs de sections [26] telles que la section de code (.text), la section de données (.data), la section de la pile et du tas (bss_stack). Il est possible de charger chacune des sections dans une mémoire différente. Pour ce cas de test, nous avons déterminé les latences en ayant chargé la section de code (.text) dans une mémoire BRAM et le reste des sections dans une mémoire externe DDR SDRAM.

Tableau 4. 3 Latences de communication entre le matériel et le logiciel, composante HwSwCom version 1, adaptateur matériel version 1

Cas	L14 cycles	L15 cycles	L16 cycles	L17 cycles	L18 cycles	Latence totale en cycles
Cas 1	19	2	12770	2970	29774	45514
Cas 2	19	2	18130	4290	42883	65324
Cas 3	19	2	1260	307	2927	4515
Cas 4	19	2	11581	2191	20386	34158
Cas 5	19	2	1466	1030	13400	15896

Les résultats présentés au tableau 4.3 montrent que la latence L18 est la plus importante. L18 correspond au transfert des données lues par le processeur à partir de la composante HwSwCom, vers la FIFO de réception du module logiciel destinataire.

On remarque que l'exécution du code logiciel à partir d'une mémoire externe implique des latences environs 10 à 15 fois plus grandes que celles calculées lorsque tout le logiciel est exécuté sur la mémoire interne BRAM. Ceci s'explique par la latence de lecture des instructions à partir de la mémoire externe.

La comparaison entre Cas 1 et Cas 2 montre que le type de mémoire externe utilisée a aussi une influence importante. En effet, la différence entre les deux cas est de l'ordre de 30%.

Dans le cas 4, le logiciel est chargé dans une mémoire externe et l'utilisation des caches pour les données et pour les instructions apporte une baisse d'environs 25% de la latence totale comparativement au cas 1.

Dans le cas 5, seule la section du code a été chargée dans une mémoire BRAM. Le reste des sections de l'exécutable est chargé dans une mémoire externe de type DDR SDRAM. En comparant la latence totale du cas 5 avec celle du cas 2, nous remarquons une baisse d'environs 76%.

Nous avons ensuite conçu une deuxième version de la composante HwSwCom. Dans cette version (version 2), la composante HwSwCom communique avec le processeur Microblaze à travers une connexion point à point assurée par le bus FSL [26]. Nous avons déterminé les latences L16, L17 et L18 pour les cas suivants.

- **Cas 6** La composante HwSwCom utilise le mécanisme d'interruptions pour signaler au processeur que des données envoyées par un module matériel et destinées à un module logiciel, ont été reçues. La lecture des données par le processeur se fait à travers une connexion point à point FSL [25] et non pas à travers le bus OPB comme c'était le cas pour les latences présentées au tableau 4.3.

- **Cas 7** La composante HwSwCom n'utilise pas le mécanisme d'interruptions. Le processeur lit les données reçues par la composante HwSwCom destinées à un module logiciel, à travers la connexion FSL entre le processeur et la composante HwSwCom. Une tâche logicielle vérifie par scrutation (de l'anglais *polling*) si la FIFO de la connexion FSL contient des données.

Pour les cas 6 et 7, le logiciel est chargé et exécuté à partir d'une mémoire BRAM interne au FPGA et connectée au processeur Microblaze à travers le bus LMB.

Tableau 4. 4 Latences de communication entre le matériel et le logiciel, composante HwSwCom version 2, adaptateur matériel version 1

Cas	L14 cycles	L15 cycles	L16 cycles	L17 cycles	L18 cycles	Latence totale en cycles
C6	19	2	1260	292	2927	4500
C7	19	2	0	292	2927	3240

Les résultats présentés au tableau 4.4 nous permettent de conclure que le mécanisme d'interruptions (latence L16) implique une latence d'environs 1260 cycles (valeur de L16 pour le cas 3 et 6) quand le code logiciel est chargé dans une mémoire BRAM. Cette latence est due principalement au changement de contexte quand le processeur est interrompu.

Nous remarquons que la lecture des données à partir de la composante HwSwCom, implique une latence de 307 cycles pour la version 1 (lecture des données à travers le bus OPB) et une latence de 292 cycles pour la version 2 (lecture des données par le lien FSL). La différence de 15 cycles entre les deux latences (environs 5%) est relativement négligeable par rapports aux latences L16 et L18. Certes, il est important de réduire la latence totale et par conséquent il est préférable, même si le gain n'est pas énorme, d'utiliser une connexion FSL pour récupérer les données.

Afin d'établir le surplus de latence ajouté par l'utilisation des librairies de Space, nous avons déterminé dans les mêmes conditions de mesure les latences L16 et L17 sans utiliser les librairies de Space et d'un RTOS. Nous trouvons 38 cycles pour L16 et 44 cycles pour L17. La comparaison des ces valeurs avec celles présentées aux tableaux 4.3 et 4.4 montre un écart très important dû à l'utilisation des librairies de Space et au mécanisme d'interruptions au sein d'un système multitâches. En particulier, ce dernier implique un changement de contexte et une sauvegarde de l'état du système logiciel avant de traiter l'interruption.

4.3.4 Communication du logiciel vers le matériel

Ce type de communication nécessite l'accès au bus OPB par le processeur Microblaze dans le cas d'écriture d'un message par un module logiciel vers un module matériel ou encore par l'écriture ou la lecture de données vers un périphérique.

Par contre la lecture de données reçues se fait localement à partir de la FIFO logicielle dédiée à la réception des messages pour le module logiciel en question.

- **Lecture de données reçues** Nous avons expliqué au paragraphe précédent que le message envoyé par un module matériel vers un module logiciel est d'abord reçu par la composante HwSwCom, puis récupéré par une tâche logicielle suite à l'interruption signalée par la composante HwSwCom. Le message est ensuite stocké par cette tâche logicielle dans une FIFO dédiée au module logiciel destinataire.

  Quand un module logiciel effectue une requête de lecture d'un message vers un autre module (logiciel ou matériel), cette lecture est effectuée à partir de sa FIFO de réception. Si la FIFO contient le message à lire, les données sont consommées, et le traitement continu. Si le message à lire n'est pas disponible dans la FIFO de réception, le module bloque en attendant la réception des données quand il s'agit de lecture bloquante ou retourne immédiatement quand la lecture est non bloquante.

- **Écriture de données vers un module matériel** Une requête d'écriture est effectuée directement par le processeur Microblaze à travers le bus OPB.
- **Écriture ou lecture de données vers un périphérique** Identique au cas précédent. Il faudra noter que le temps de réponse du périphérique en question varie selon son type.

Pour déterminer les latences de ces opérations, nous avons utilisé la méthode de mesure décrite à la section 4.3.2 appliquée à l'architecture présentée à la figure 4.6. Quatre autres cas s'ajoutent donc :

1. **Cas 8** Écriture d'un message par le module logiciel M2 vers le module matériel M1
2. **Cas 9** Lecture d'un message par le module logiciel M2. Le message est envoyé par le module matériel M1
3. **Cas 10** Écriture d'une donnée de 4 octets par le module logiciel M2 vers la mémoire OPB BRAM
4. **Cas 11** Lecture d'une donnée de 4 octets par le module logiciel M2 à partir de la mémoire OPB BRAM

Les tableaux 4.5 et 4.6 présentent les latences (en cycles) obtenues pour ces cas de tests quand le code logiciel est chargé dans une mémoire interne BRAM connectée au processeur Microblaze à travers le bus LMB et quand le code logiciel est chargé dans une mémoire externe de type DDR SDRAM.
Les résultats du tableau 4.5 sont obtenus pour des cas de tests avec utilisation des librairies de Space, alors que ceux du tableau 4.6 sont obtenus par l'exécution d'un code logiciel en C qui n'utilise pas Space.

Tableau 4. 5 Latences des communications logiciel- matériel avec utilisation de Space

Type de mémoire code / cas de test	Cas 8	Cas 9	Cas 10	Cas 11
LMB BRAM	535	2051	321	326
Mémoire externe DDR SDRAM	7480	30010	3400	3465

Tableau 4. 6 Latences des communications logiciel- matériel sans utilisation de Space

Type de mémoire code / cas de test	Cas 8	Cas 9	Cas 10	Cas 11
LMB BRAM	24	NA	12	14
Mémoire externe DDR SDRAM	108	NA	68	84

L'analyse des résultats présentés aux tableaux 4.5 et 4.6 nous permettent de conclure que l'exécution du code logiciel à partir d'une mémoire externe dégradent largement les latences (un facteur allant de 10 à 14 pour les résultats du tableau 4.5) par rapport à l'exécution du code à partir d'une mémoire BRAM interne au FPGA et connectée au processeur à travers le bus LMB.

La comparaison des résultats des deux tableaux confirme aussi le surplus de latence ajouté par l'utilisation des librairies de Space.

4.3.5 Communication du logiciel vers le logiciel

L'opération d'écriture d'un message par un module logiciel vers un autre module logiciel se matérialise par l'écriture du message en question dans une FIFO de réception dédiée au module destinataire.

La lecture d'un message par un module logiciel est effectuée, comme pour le cas d'une communication logiciel-matériel, c'est-à-dire à partir de la FIFO de réception du module logiciel initiateur de la lecture.

Pour déterminer les latences de ces opérations, nous avons utilisé la méthode de mesure décrite à la section 4.3.2 appliquée à l'architecture présentée à la figure 4.6. Deux autres cas s'ajoutent donc :

1. **Cas 12** Le module logiciel M2 écrit un message d'un en-tête de 4 octets et d'une donnée de 4 octets au module logiciel M3
2. **Cas 13** Le module M3 consomme les messages qui lui ont été envoyé par le module M2

Afin de ne pas fausser la valeur de la latence de lecture, nous avons ordonnancé l'exécution des modules M2 et M3 de manière à ce que le message destiné au module M3 soit disponible à la consommation quand la lecture non bloquante est effectuée par M3.

Le tableau 4.7 présente les valeurs des latences (en cycles) d'écriture et de lecture (cas C12 et C13) entre deux modules logiciels. Nous présentons ces valeurs dans le cas où le logiciel est exécuté respectivement à partir d'une mémoire BRAM connectée au processeur par le bus LMB et à partir d'une mémoire DDR SDRAM externe au FPGA.

Tableau 4. 7 Latences des communications logiciel-logiciel

Type de mémoire code / cas de test	Cas 12 Première écriture	Cas 12 Autres écritures	Cas 13
LMB BRAM	3764	2948	2051
Mémoire externe DDR SDRAM	55050	43150	30003

Nous rappelons que la lecture d'un message par un module logiciel s'effectue au niveau de sa FIFO de réception. Ainsi, nous constatons que la lecture d'un message provenant d'un module logiciel ou d'un module matériel a la même latence (cas C13 du tableau 4.7 et cas C9 du tableau 4.5).

Nous trouvons une latence très élevée pour l'opération d'écriture entre deux modules faisant partie de la même partition logicielle.

Nous avons prélevé la latence pour la première écriture et celle pour les écritures subséquentes. L'écart de 21% entre les deux latences est imputable à l'opération d'allocation d'une FIFO de réception pour le module logiciel qui reçoit des messages. En effet, quand un message est reçu, il devra être stocké dans une FIFO. Or, l'allocation d'une FIFO de réception, pour un module logiciel est effectuée dynamiquement (fonction **ModuleWrite** de la classe **SWBus**) parmi un ensemble de FIFO libres créées lors la compilation du code source. La FIFO est libérée quand tous les messages qu'elle contient ont été consommés.

4.3.6 Proposition d'un nouveau concept pour les communications matériel-logiciel

Tel que mentionné à plusieurs reprises, l'objectif principal de notre travail consiste à reproduire à bas niveau (implantation sur une puce de type FPGA) les concepts de Space à haut niveau. Nous avons vu que l'envoi des messages par un module matériel vers un module logiciel fait intervenir la composante HwSwCom qui utilise le mécanisme d'interruptions pour signaler au processeur que des données ont été reçues. Ces données sont ensuite lues par le processeur et transférées vers la FIFO de réception dédiée au module logiciel destinataire.

Quand le module logiciel a besoin de consommer un message, il le récupère à partir de sa FIFO de réception (implantée en logiciel).

Nous avons vu aux paragraphes précédents que ces opérations génèrent des latences très importantes. D'une part la latence est due au mécanisme d'interruptions (les changements de contexte et les sauvegardes impliquées) qui dépasse 10000 cycles pour un code logiciel exécuté à partir d'une mémoire externe et avoisine 1000 cycles pour un code exécuté à partir de la mémoire BRAM. D'autre part, la latence de l'opération de transfert des données de la composante HwSwCom vers la FIFO du module logiciel destinataire engendre une latence de plus que 30000 cycles et d'environ 3000 cycles quand le code logiciel est exécuté à partir d'une mémoire externe ou d'une mémoire BRAM interne, respectivement.

Afin d'améliorer les latences de communication entre le matériel et le logiciel, en particulier du côté du logiciel, nous proposons un nouveau concept pour la composante HwSwCom et une nouvelle manière d'effectuer la lecture par un module logiciel des messages envoyés par un module matériel.

Le concept consiste à éliminer l'interruption entre le processeur et la composante HwSwCom, à connecter cette composante au processeur par deux liens FSL [26] et à effectuer la lecture des messages directement par le module logiciel à partir de cette composante au lieu de transférer les messages vers une FIFO logicielle.

Cette nouvelle version de la composante HwSwCom diffère des versions proposées pour le cas 6 et le cas 7 pour lesquelles le stockage des données, destinées à un module logiciel, se fait dans une FIFO logicielle. Ce qui nécessite une étape de lecture de ces données à partir de la composante HwSwCom (latence L17) et une étape de transfert des données lues vers la FIFO de réception (latence L18).

La figure 4.8 présente un schéma de principe de la connexion entre la composante HwSwCom et le processeur Microblaze.

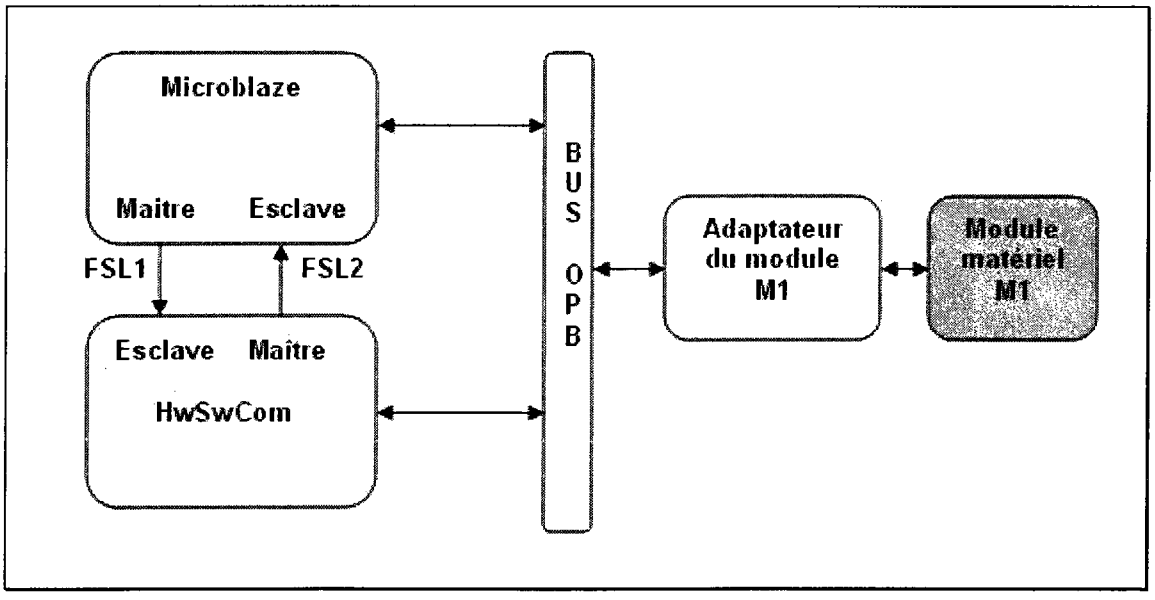

Figure 4. 7 Adaptateur des communications matériel-logiciel, version FSL

Le fonctionnement de cette nouvelle version de la composante HwSwCom ressemble en partie au principe que nous avons utilisé pour concevoir l'adaptateur d'un module matériel (voir section 3.3.3). La ressemblance réside dans le fait d'utiliser une ou plusieurs FIFO de réception selon le nombre de modules qui sont censés envoyer des messages à un module en particulier.

Le fonctionnement détaillé de la nouvelle version de la composante HwSwCom est comme suit :

- Les messages envoyés par le module matériel M1 vers un module logiciel M2 sont acheminés par le bus OPB vers la composante HwSwCom
- Un message reçu par la composante HwSwCom est identifié selon son en-tête et stocké dans une FIFO matérielle dédiée.
- Quand un module logiciel effectue une requête de lecture pour récupérer un message qui lui a été envoyé par un module matériel, une commande de lecture est alors envoyée par le processeur à la composante HwSwCom à travers la connexion FSL1 (figure 4.8). Cette commande de lecture d'une grandeur de 4 octets est formée à partir de l'identificateur du module source de la requête, de l'identificateur du module destinataire de la requête, de la taille du message à lire et de l'option de lecture bloquante ou non.
- Le module logiciel bloque sur la lecture du statut de traitement de sa requête. Ce statut lui sera renvoyé par la composante HwSwCom à travers le lien FSL2.
- Une machine à états finis de la composante matérielle HwSwCom récupère les commandes de lecture reçues par le lien FLS1 et les traitent.
- Après traitement de la requête de lecture, la composante HwSwCom renvoie au module logiciel un statut (valeur de 4 octets représentant le résultat du traitement) et les données (quand elles sont disponibles) à travers le lien FSL2.
- Le module logiciel récupère le statut de la requête et les données si sa requête a été traitée avec succès par la composante HwSwCom.

Nous avons implémenté une version préliminaire de la composante HwSwCom selon la description ci-haut afin de vérifier son fonctionnement et le gain apporté en termes de réduction des latences.

Notez que le raffinement de cette proposition d'améliorations fera l'objet de travaux futurs et elle dépasse l'objet du présent travail. Toutefois, nous avons procédé à quelques modifications de l'architecture existante pour obtenir une version préliminaire. En particulier nous avons procédé à des modifications du code source de certaines classes des librairies de Space afin de porter les adaptateurs à ce concept. Les modifications consistent à éliminer le traitement de l'interruption de la composante HwSwCom, à éliminer le transfert intermédiaire des messages entre la composante HwSwCom et la FIFO logicielle du module destinataire, et à effectuer la lecture des messages directement à partir de la composante matérielle HwSwCom.

Pour déterminer le gain total, en nombre de cycles, apporté par la nouvelle version de la composante HwSwCom, nous présentons au tableau 4.8 le sommaire des latences impliquées par le processus de transfert des données de la composante HwSwCom (version 1 avec mécanisme d'interruptions et lecture des données par le logiciel à travers le bus OPB) et au tableau 4.9 les latences d'une opération de lecture des données par un module logiciel à partir de la composante HwSwCom selon ce nouveau concept.

Tableau 4. 8 Sommaire des latences de transfert des données entre la composante HwSwCom et le logiciel

	BRAM	DDR SDRAM
Mécanisme d'interruptions	1260	18130
Lecture des données à partir de la composante HwSwCom	307	4290
Transfert des données vers une FIFO de réception du module logiciel	2927	42883
Latence totale des 3 opérations précédentes	4494	65303
Lecture d'un message par le module logiciel	2051	30010
Latence totale dans le pire des cas (quand un module logiciel demande une lecture, et que les données ne sont pas disponibles dans sa FIFO de réception. Il devra attendre au minimum que les trois premières opérations soient effectuées)	**6545**	**95313**

Tableau 4. 9 Latences de lecture d'un message à partir de la composante HwSwCom, version améliorée

	BRAM	DDR SDRAM
Lecture des données par le module logiciel, à travers le lien FSL, à partir de la composante HwSwCom	**185**	**2576**

La nouvelle version de la composante HwSwCom améliore largement les performances du logiciel en réduisant la latence de la communication matériel-logiciel d'environs 97%. En plus, cette solution n'utilise pas le bus OPB pour effectuer la lecture des données, ce qui a l'avantage de libérer le bus pour que d'autres maîtres peuvent l'utiliser.

Cependant, cette solution présente l'inconvénient d'utiliser plusieurs FIFO de réception au niveau de la composante HwSwCom. Les FIFO sont crées en matériel à partir de la mémoire BRAM, ce qui du même coup peut être une limitation pour les adaptateurs de modules matériels.

Un compromis entre la taille des FIFO utilisées et la fréquence de fonctionnement de l'application peut être déterminé assez aisément. Ce compromis dépend en grande partie de la vitesse à laquelle les messages reçus au niveau de la composante HwSwCom, sont consommés par le module logiciel destinataire. Nous décrivons à la section 4.4 une manière de limiter la profondeur des FIFO de réception.

Il faut noter également que cette nouvelle version de la composante HwSwCom réduit le nombre de FIFO utilisées en logiciel pour stocker les messages puisque les messages provenant des modules matériels seront stockés dans des FIFO en matériel au niveau de la composante HwSwCom.

Nous pouvons également étendre cette solution pour la communication logiciel-logiciel. Prenons l'exemple d'un module logiciel M1 qui envoie des messages au module logiciel M2 :

- Les messages envoyés par M1 sont acheminés par le lien FSL1 (figure 4.8) vers la composante HwSwCom et seront stockés dans une FIFO matérielle.
- Le module M2 consomme les messages de M1 à partir de la composante HwSwCom à travers le lien FSL2 (figure 4.8).

Cette solution présente aussi un inconvénient dans le cas d'une lecture bloquante et que les données ne sont pas disponibles au niveau de la composante HwSwCom. Dans un tel cas, le module logiciel devra effectuer une scrutation. Nous pensons qu'une scrutation périodique (période de la minuterie du RTOS) peut être une bonne solution. En utilisant la fonction **wait( )**, il est possible paramétrer la scrutation avec le nombre de cycles d'attente voulu.

4.4 La profondeur des FIFO d'envoi et de réception

Nous avons vu au chapitre 3 que les données à envoyer par un module à un autre module ou à un périphérique sont d'abord stockées dans une FIFO d'envoi, alors que les données reçues de la part des autres modules sont aussi stockées dans une ou plusieurs FIFO de réception (une FIFO de réception est dédiée à chaque module source).

Par défaut, la FIFO d'envoi a une profondeur limitée à 16 données. Cette limitation de la profondeur provient de l'IPIF de Xilinx pour lequel le nombre de données à envoyer en mode rafale ne doit pas dépasser 16.

Les FIFO de réception utilisent des blocs de mémoire BRAM interne au FPGA. La seule limitation au niveau de la profondeur de ces FIFO est la quantité de BRAM disponible. Dans certains cas le module producteur de données est plus rapide que le module consommateur. Deux possibilités sont alors possibles afin d'éviter de perdre des données,

- **Prévoir une profondeur de FIFO de réception suffisamment grande** Cette solution présente l'inconvénient d'utiliser plus de mémoire BRAM.
- **Utiliser le mécanisme d'écriture bloquante de l'IPIF** Par l'intermédiaire du paramètre générique **C_WAIT_SLV_ACK** au niveau de l'adaptateur, l'utilisateur choisira le mode de fonctionnement dans lequel l'acquittement d'une requête d'écriture sera effectué uniquement si l'adaptateur qui reçoit les données répond par un acquittement. Par contre, dans le cas où la FIFO de réception est pleine, l'adaptateur destinataire pourra utiliser le signal **IP2Bus_retry** pour signaler au contrôleur du bus qu'il est n'est pas prêt à recevoir les données. Le contrôleur du bus maintient sa requête jusqu'à ce que l'adaptateur destinataire soit prêt à recevoir la donnée.

  Cette deuxième solution réduit notamment la fréquence de fonctionnement de l'adaptateur et de tout le système, puisque d'autres maîtres peuvent être dans l'état d'attente d'accès au bus. Mais elle permet d'utiliser des FIFO de réception de profondeur raisonnable en évitant la perte de données.

4.5 Utilisation des ressources et fréquence maximale

4.5.1 Cas de l'adaptateur d'un module matériel

L'adaptateur d'un module matériel utilise une FIFO d'envoi de 16 éléments de 4 octets chacun. Il peut ou non utiliser une ou plusieurs FIFO de réception selon le nombre de modules qui lui envoient des messages. Le nombre de FIFO de réception et leur profondeur sont des paramètres génériques qui doivent être définis par l'utilisateur.

Le tableau 4.10 présente le sommaire des ressources utilisées par l'adaptateur (version 1 et version 2) d'un module matériel et sa fréquence maximale de fonctionnement. Ces résultats sont obtenus par la synthèse du code VHDL de l'adaptateur avec l'outil de Xilinx ISE version 8.1 pour un FPGA Virtex2p VP30.
Sachant que l'adaptateur d'un module matériel (version no 1) est conçu autour de la composante IPIF de Xilinx, nous présentons aussi au tableau 4.10 les ressources utilisées par l'IPIF afin d'évaluer le surplus de ressources utilisées par la couche de l'adaptateur que nous avons conçu autour de l'IPIF de Xilinx.

Au tableau 4.10, nous présentons également les ressources utilisées par une composante maître/esclave générée automatiquement par l'outil EDK de Xilinx (menu Create or Import Peripheral). La composante générée est constituée d'un IPIF et d'une entité nommée « UserLogic » qui implémente la logique nécessaire afin que cette composante soit adressable et puisse effectuer des requêtes de lecture ou d'écriture à travers le bus.

Tableau 4. 10 Utilisation des ressources et fréquence maximale pour l'adaptateur matériel (Virtex 2P VP30)

	Slices	LUTs	Bascules (Flip flop)	Fréquence maximale MHz
Adaptateur version 1	435 / 13696 = 3 %	749 / 27392 = 2 %	616 / 27392 = 2 %	163
Adaptateur version 2	353 / 13696 = 3 %	300 / 27392 = 2 %	600 / 27392 = 2 %	122
IPIF version opb_ipif_v2_00_h	247 / 13696 = 1 %	391 / 27392 = 1 %	289 / 27392 = 1 %	222
IP généré par EDK	1027 / 13696 = 7 %	1240 / 27392 = 4 %	1581 / 27392 = 5 %	130

Les résultats du tableau 4.10 sont présentés dans le cas d'un adaptateur ne contenant aucune FIFO de réception. Par conséquent, la quantité de mémoire BRAM utilisée est nulle.

Nous remarquons que l'adaptateur (versions 1 et 2) utilise moins de ressources que la composante générée automatiquement pour l'outil EDK de Xilinx. Nous remarquons également que l'IPIF utilise environs 50% des ressources utilisées par l'adaptateur (version 1).

Les tableaux 4.11 et 4.12 présentent les résultats de l'utilisation des ressources par l'adaptateur (version 1) pour différentes valeurs des paramètres génériques **C_FIFO_SIZE** (profondeur de la FIFO de réception) et **C_NUM_FIFO** (nombre des FIFO de réception utilisées par l'adaptateur).

Pour les résultats du tableau 4.11, nous avons fixé C_NUM_FIFO à 1 et nous avons varié la valeur de C_FIFO_SIZE. Alors que pour les résultats du tableau 4.12, nous avons fixé la valeur de C_FIFO_SIZE à 512 éléments et nous avons varié la valeur de C_NUM_FIFO.

Tableau 4. 11 Ressources utilisées par l'adaptateur matériel en fonction de la profondeur de la FIFO de réception (Virtex 2P VP30)

Profondeur de la FIFO de réception (valeur de C_FIFO_SIZE)	Slices	LUTs	Bascules (Flip flop)	Fréquence maximale MHz
16	668 / 13696 = 4 %	842 / 27392 = 3 %	1160 / 27392 = 4 %	158
32	671 / 13696 = 4 %	850 / 27392 = 3 %	1172 / 27392 = 4 %	158
128	678 / 13696 = 4 %	865 / 27392 = 3 %	1180 / 27392 = 4 %	158
512	689 / 13696 = 5 %	877 / 27392 = 3 %	1195 / 27392 = 4 %	158

Tableau 4. 12 Ressources utilisées par l'adaptateur matériel en fonction du nombre de FIFO de réception (Virtex 2P VP30)

Nombre de FIFO de réception (valeur de C_NUM_SIZE)	Slices	LUTs	Bascules (Flip flop)	Fréquence maximale MHz
1	681 / 13696 = 4 %	871 / 27392 = 3 %	1187 / 27392 = 4 %	158
2	727 / 13696 = 5 %	956 / 27392 = 3 %	1259 / 27392 = 4 %	158
3	843 / 13696 = 6 %	1069 / 27392 = 3 %	1450 / 27392 = 5 %	158
4	923 / 13696 = 6 %	1169 / 27392 = 4 %	1582 / 27392 = 5 %	157
5	1042 / 13696 = 7 %	1266 / 27392 = 4 %	1796 / 27392 = 6 %	157

Les résultats du tableau 4.11 montrent que la profondeur de la FIFO de réception a une légère influence sur le nombre de ressources utilisées par l'adaptateur d'un module matériel. L'augmentation du nombre de ressources utilisées est imputable au contrôleur de la FIFO qui utilise des blocks de mémoire BRAM, des bascules pour mémoriser le pointeur sur l'élément courant, des bascules pour mémoriser le nombre d'éléments contenus dans la FIFO, etc.

Les résultats du tableau 4.12 montrent que le nombre de ressources utilisées par l'adaptateur augmente de façon significative quand le paramètre C_NUM_FIFO augmente. Cette augmentation est due à la logique utilisée par l'adaptateur pour emmagasiner dans les FIFO de réception les messages selon l'identificateur du module source et l'identificateur du module destinataire.

En effet, ce concept de FIFO dédiée nécessite l'identification du message à sa réception afin de l'insérer dans la FIFO correspondante (figure 3.14). Il nécessite aussi de la

logique de contrôle lors de la lecture du message. Cette lecture est précédé par une phase de recherche de la FIFO à partir de laquelle les données seront lues (figure 3.15).

4.6 Remarques générales

Nous émettons dans ce paragraphe quatre remarques générales concernant les concepts et le fonctionnement de la plate-forme Space.

1. **Le mécanisme d'interruptions entre la composante HwSwCom et le processeur Microblaze** Nous avons vu dans les sections précédentes que ce mécanisme ainsi que les opérations d'écriture ou de lecture du côté de la partition logicielle génèrent des latences très importantes. Ceci limite la fréquence à laquelle le logiciel est capable recevoir des messages provenant d'un module ou de plusieurs modules matériels.

   Quand le code logiciel est exécuté à partir de la mémoire externe, il faut environs 65000 cycles pour traiter l'interruption provenant de la composante HwSwCom et transférer les données vers la FIFO du module logiciel destinataire. Cette valeur risque d'augmenter si le processeur reçoit d'autres interruptions de la composante Timer (nécessaire pour le fonctionnement du RTOS) pendant qu'il est en train de transférer les données.

   Il existe donc des compromis importants à faire. D'une part l'utilisation d'un RTOS facilite la programmation multitâches et favorise la réutilisation du logiciel, mais d'autre part cela peut dégrader rapidement la vitesse d'exécution. Également, un compromis existe au niveau du choix d'un mode de fonctionnement par interruptions ou par scrutation.

2. **Les latences générées par les opérations d'un module logiciel** Le résultat du cas C9 est intéressant. En effet, il s'agit d'une opération de lecture d'un message par un module logiciel. Le message en question a été envoyé par un module

matériel et il est déjà stocké dans une mémoire FIFO (en logiciel) dédiée au module logiciel.

Pour comprendre la raison de cette valeur pour le cas C9, nous avons analysé sommairement le code source de la fonction **ModuleRead ( )** de la classe **SWBus**. Nous avons constaté que cette fonction effectue plusieurs opérations de contrôle avec des tests **if else**. Nous pensons que les branchements lors de l'exécution du code influence les performances du pipeline du processeur. Cette fonction effectue aussi plusieurs appels à d'autres fonctions qui prennent en moyenne 2 paramètres chacune. Par conséquent elle utilise assez souvent la pile pour y stocker les paramètres des fonctions, les adresses de retour et possiblement des variables locales créées par les fonctions appelées.

3. **L'initialisation de la partition logicielle** Le code logiciel, basé sur les librairies de Space a besoin d'effectuer plusieurs configurations et initialisations avant de créer et démarrer les modules logiciels. Cette phase d'initialisation dure environs **400 cycles et 5500 cycles** respectivement quand le code logiciel est exécuté à partir de la mémoire BRAM connecté au processeur par le bus LMB et à partir d'une mémoire externe de type DDR SDRAM.

   Nous avons remarqué que si une interruption provient de la composante HwSwCom juste avant la création des modules logiciels, l'exécution de l'application gèle complètement. Ceci s'explique par le fait que le traitement de l'interruption comporte une phase de transfert des données vers une FIFO dédié au module logiciel destinataire. Or, il se trouve que le module logiciel n'a pas encore été crée. Et nous avons établi la durée de création d'un module logiciel à environs **8750 cycles et 129470 cycles** quand le code logiciel est exécuté respectivement à partir de la mémoire BRAM (connecté au processeur par le bus LMB) et à partir d'une mémoire externe de type DDR SDRAM.

Pour y remédier, nous avons modifié la logique de la composante HwSwCom pour ne pas déclencher les interruptions avant la création des modules logiciels. Ceci est réalisé par l'écriture d'un mot du logiciel à la composante HwSwCom pour lui signaler que le logiciel est prêt à recevoir les interruptions.

4. **L'utilisation d'un délai d'attente lors des communications bloquantes** Quand l'écriture d'un message par un module vers un autre module est bloquante, le module source de la requête bloquera en attente d'un acquittement qui lui sera envoyé par le module destinataire de l'écriture lorsque ce dernier consomme le message. Lorsque l'écriture est bloquante avec un certain délai d'attente (*timeout*), une incohérence de fonctionnement au niveau du concept peut se produire. Ce cas d'incohérence survient quand le délai d'attente s'écoule du côté du module source et que le module destinataire consomme et renvoi l'acquittement après. Le module source se débloque avec un statut de « TimeOut » et ne sait pas si le module destinataire a réellement consommé le message ou non.

Conclusion et travaux futurs

Nous avons vu au chapitre 1 l'évolution des méthodologies de conception des systèmes sur puce. La tendance actuelle de telles méthodologies est d'intégrer la partie logicielle et la partie matérielle dans le même flot de conception et de procéder par raffinement progressif de la spécification jusqu'à l'obtention d'une configuration satisfaisante en termes de performances. L'introduction de librairies bâties sur des langages de programmation de haut niveau, tel que le langage C++, pour la validation des systèmes sur puce apporte une accélération importante du temps de simulation, et par conséquent réduit le temps nécessaire à la validation à haut niveau de tels systèmes.

Nous avons décrit l'approche de la plate-forme Space basée sur SystemC pour la conception et la validation à haut niveau d'un système composé d'une partition matérielle et / ou logicielle. Cette plate-forme utilise un protocole de communication, basé sur l'échange de message (de l'anglais message passing) entre les différentes composantes à travers une architecture de bus.

La majorité des travaux de recherche pour la conception et la validation des systèmes sur puce à haut niveau d'abstraction ne proposent pas de solutions pour étendre leurs méthodologies jusqu'à l'implémentation du système sur une puce. Nous assistons à l'émergence d'outils commerciaux capables de synthétiser une spécification de haut niveau (basée sur un langage tel que C, C++ ou SystemC) en vue de son implémentation physique, citons Forte Synthetizer de Forte Design et CoDeveloper de ImpulseC.

Notre travail a consisté à implémenter le protocole de communication de Space sur une puce reprogrammable de type FPGA. Par conséquent, nous avons contribué à ajouter un autre niveau de raffinement (niveau RTL) dans la méthodologie de Space. Désormais, un système conçu avec Space à haut niveau pourra être implémenté de manière semi automatique sur une puce.

Pour atteindre cet objectif nous avons conçu en VHDL, l'entité adaptateur d'un module matériel (basée sur la composante IPIF de Xilinx) qui permet à ce dernier d'être connecté en tant que maître/esclave sur un bus OPB. Nous avons aussi conçu un adaptateur pour les communications entre les partitions matérielle et logicielle du système.
Nous avons établi avec précision les latences pour les communications matériel-matériel, matériel-logiciel, logiciel-logiciel et logiciel-matériel. Les résultats obtenus nous ont amenés à proposer une nouvelle version de l'adaptateur d'un module matériel. Cette version permet de réduire la latence d'une écriture, faite par un module matériel, de 40%.

Nous avons aussi proposé un nouveau concept pour la lecture des données par un module logiciel lorsque ses données lui ont été envoyées par un module matériel. Ce nouveau concept concerne la composante HwSwCom et consiste à stocker les données destinées aux modules logiciels dans des FIFO au niveau de la composante HwSwCom, à éliminer définitivement le mécanisme d'interruptions entre cette composante et le processeur Microblaze, et à effectuer la lecture des données, par un module logiciel, à partir de la composante HwSwCom.
Dans cette nouvelle version, la composante HwSwCom est connectée au processeur Microblaze par deux liens FSL à travers lesquelles la lecture des données est effectuée par les modules logiciels. Notons que la lecture pourra être faite à travers le bus de l'architecture, dans notre cas le bus OPB. Par conséquent, notre proposition reste valide si un autre processeur, ne disposant pas de liens FSL, est utilisé.
Nous rappelons que ce nouveau concept permet de réduire la latence de la communication matériel-logiciel de 97%.

Les travaux futurs qui peuvent découler de notre projet sont nombreux. Nous pensons qu'il est nécessaire de compléter la méthodologie de raffinement, d'un système conçu avec Space, pour intégrer le passage direct de la spécification haut niveau d'un module matériel vers une spécification RTL synthétisable. Ceci permet d'éliminer l'étape de la retranscription manuelle du code SystemC TLM vers un langage HDL synthétisable.

Les fonctionnalités de l'adaptateur d'un module matériel peuvent être étendues afin qu'il puisse supporter plusieurs modules, et supporter la communication directe entre les modules connectés sur le même adaptateur sans passer par le bus OPB. Cette extension de l'adaptateur nécessite l'ajout d'un arbitre pour gérer l'accès de plusieurs modules aux services du même adaptateur.

Il est aussi essentiel de mener des réflexions afin de mieux optimiser les librairies logicielles ou trouver de nouveaux concepts afin de réduire les latences de communications du côté du logiciel. En effet, nous avons vu au chapitre 4 que ces latences sont de l'ordre de plusieurs milliers de cycles, et il devient insignifiant de discuter des latences du côté matériel par rapport à celles-ci.

Nous pensons que le même principe des adaptateurs pourra être facilement implémenté pour d'autres bus, en particulier le bus PLB qui fait partie du standard CoreConnect. En effet, nous avons vu que la version no 1 de l'adaptateur d'un module matériel utilise l'IPIF de Xilinx pour connecter le module matériel au bus OPB. Un travail futur serait de concevoir un adaptateur pour connecter un module matériel au bus PLB en utilisant l'IPIF de Xilinx pour le bus PLB.

Références

[1] GAUTHIER François : Pas de flot complet pour l'ESL, mais standards et outils sont déjà là. Avril 2006 n168- Tendance.

[2] Spécifications de SystemC. [En ligne]. http://www.systemc.org. (Page consultée 15 juin 2006).

[3] KLINGAUF Wolfgang, GUNZEL Robert. Rapid Prototyping with SystemC and Transaction Level Modeling. Technical Univertsity of Braunschweig, Dept E.I.S. Germany.

[4] YOO Sungjoo, NICOLESCU Gabriela, LYONNARD Damien, BAGHDADI Amer, JERRAYA A. Ahmed. A Generic Wrapper Architecture for Multi-Processor SoC Cosimulation and Design. Hardware/Software Codesign, 2001. CODES 2001. Proceedings of the Ninth International Symposium on

[5] SINGH Amarjeet , CHHABRA Amit, GANGWAR Anup, DWIVEDI K. Basant , BALAKRISHMAN M., KUMAR Anshul. SoC Synthesis with Automatic Hardware Software Interface Generation. VLSI Design, 2003. Proceedings. 16th International Conference on.

[6] CLASSER Mark. [En ligne]. Applying Transaction-Level Models for Design and Testbenches. Contributor: Mentor Graphics Corp. http://www.soccentral.com/results.asp?CategoryID=488&EntryID=19220. (Page consultée le 10 janvier 2007).

[7] MOUSSA Imed, ROUDIER Thierry. IP Modeling and Reuse for SoC Design using Standard Bus. DESIGN and REUSE, article 4837.

[8] JERRAYA A. Amine. Conception haut niveau des systèmes monopuces.EGEM. ISBN 2-7462-0433-9

[9] ROWEN Chris. Engineering the Complex SOC. Prentice Hall Modern Semiconductor Design Series. ISBN 0-13-145537-0.

[10] CoreConnect™ bus architecture (White Paper), 1999. [En ligne]. http://www-306.ibm.com/chips/products/coreconnect/. (Page consultée le 22 février 2006).

[11] ARM AMBA Specification. [En ligne]. http://www.arm.com/products/solutions/AMBA_Spec.html. (Page consultée le 30 janvier 2007).

[12] Avalon Memory-Mapped Interface Specification. [En ligne]. http://www.altera.com/literature/manual/mnl_avalon_spec.pdf. (Page consultée le 20 janvier 2007).

[13] SHIN Dongwan, GERSTLAUER Andreas, PENG Junyu, DOMER Rainer, GAJSKI D. Daniel. Automatic Generation of Transaction-Level Models for Rapid Design Space Exploration. 4th international conference on Hardware/software codesign and system synthesis

[14] BOUCHHIMA Aimen, CHEN Xi, PETROT Frédéric, CESARIO O. Wander , JERRAYA A. Ahmed. A Unified HW/SW Interface Model to Remove Discontinuities between HW and SW Design. Septembre 2005, 5th ACM international conference on Embedded software EMSOFT '05.

[15] POLETTI Francesco, POGGIALI Antonio, MARCHARL Paul. Flexible hardware/software support for message passing on a distributed shared memory architecture. March 2005, DATE'05.

[16] ZERGAINOH Nacer-Eddine, BAGHDADI Amer, JERRAYA A. Ahmed. Hardware/software codesign of on-chip communication architecture for application-specific multiprocessor system-on-chip. International Journal of Embedded Systems 2005- Vol. 1, No. ½ pp. 112-124.

[17] KLINGAUF Wolfgang, GADKE Hagen, GUNZEL Robert. TRAIN: A Virtual Transaction Layer Architecture for TLM-based HW/SW Codesign of Synthesizable MPSoC. DATE '2006.

[18] NACUL C. André, LAJOLO Marcello, GIVARGIS Tony. Interface-Centric Abstraction Level for Rapid Hardware/Software Integration. Forum on Specification and Design Languages 2005.

[19] KU David, DEMICHELI Giovanni. HardwareC -- A Language for Hardware Design (Version 2.0). Stanford University, 1990.

[20] Celoxica, Handel-C- An effective method for designing FPGAs and ASICs. [En ligne]. http://www.celoxica.com. (Page consultée le 5 mars 2007).

[21] SpecC Language Reference Manual Version 2.0. [En ligne]. http://www.cecs.uci.edu/~specc/reference/. (Page consultée 27 février 2007).

[22] Stefan Fischer, Jacek Wytrebowicz, Stanislaw Budkowski. Hardware/Software Co-Design of Communication Protocol. 22nd EUROMICRO Conference

[23] CLOUTÉ François, CONTENSOU Jean-Noêl, ESTEVE Daniel, PAMPAGNIN Pascal, PONS Philippe, FAVARD Yves. Hardware/Software Co-Design of an Avionics Communication Protocol Interface System: an industrial Case Study. Hardware/software codesign CODES '99.

[24] KANGAS Tero. Methods and Implementations for Automated System On Chip Architecture Exploration. Tampere University of Technology. Publication 616. 29th of September 2006.

[25] Fast Simplex Link (FSL) Bus (v2.00a). [En ligne]. http://www.xilinx.com/bvdocs/ipcenter/data_sheet/FSL_V20.pdf. (Page consultée 11 avril 2006).

[26] Microblaze Processor Reference guide. [En ligne]. http://www.xilinx.com/ise/embedded/mb_ref_guide.pdf. (Page consultée le 10 janvier 2006).

[27] WIEFERINK Andreas, LEUPERS Rainer, ASCHEID Gerd, MEYR Heinrich, MICHIELS Tom. Retargetable Generation of TLM Bus Interfaces for MP-SoC Plateforms. 3rd IEEE/ACM/IFIP international conference CODES+ISSS '2005.

[28] MURAOKA Michiaki, NISHI Hiroaki, MORIZAWA K. Rafael, YOKOTA Hideaki, HAMADA Hideyuki. Design Methodology for SoC Architectures based on Reusable Virtual Cores. 2004 conference on Asia South Pacific design automation: electronic design and solution fair ASP-DAC '04.

[29] OPB IPIF (v2.00h). [En ligne]. http://www.xilinx.com. (Page consultée le 5 janvier 2006).

[30] MOYA M. José, RINCON Fernando, MOYA Francisco, LOPEZ Juan Carlos. Improving Embedded System Design by means of HW-SW Compilation on Reconfigurable Coprocessors. System Synthesis, 2002, 15th International Symposium on.

[31] YUYAME Yoichi, ARAMOTO Masao, KOBAYASHI Kazutoshi, ONODERA Hidetoshi. An SoC Architecture and its Design Methodlogy using Unifunctional Heterogeneous Processor Array.

[32] PASRICHA Sudeep. Transaction level modeling of SoC with SystemC 2.0.Design Flow and Reuse/CR&D. STMicroelectronics Ltd.

[33] Open Core Protocol Specification, Release 2.1. [En ligne]. http://www.ocpip.org. (Page consultée le 20 février 2007).

[34] Data-Side OCM BRAM Interface Controller (v3.00a). [En ligne]. http://www.xilinx.com/bvdocs/ipcenter/data_sheet/ (Page consultée le 20 septembre 2006).

[35] Introduction à SystemC. [En ligne]. http://www.comelec.enst.fr/hdl/sc_intro.html. (Page consultée le 19 février 2007).

[36] GROETKER Thorsten, LIAO Stan, GRANT Grant, SWAN Stuart. System Design with SystemC. ISBN 1-4020-7072-1.

[37] Functional Specification for SystemC 2.0. [En ligne] http://www.systemC.org. (Page 24, consultée le 17 février 2007).

[38] On-Chip Peripheral Bus V2.0 with OPB Arbiter (v1.10c), DataSheet401. [En ligne]. http://www.xilinx.com. (Pages consultées 10 mars 2006).

[39] On-Chip Peripheral Bus. Architecture Specifications, Version 2.1. [En ligne]. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/ (Page consultée 10 avril 2006).

[40] OPB IPIF (V3.01c), Product Specification. [En ligne]. http://www.xilinx.com/bvdocs/ipcenter/data_sheet/opb_ipif.pdf (Page consultée le 5 février 2006)

[41] Virtex-II Platform FPGAs: Complete Data Sheet, Product Specification. [En ligne]. http://direct.xilinx.com/bvdocs/publications/ds031.pdf. (Page consultée le 13 mai 2006).

[42] CAI Lukai, GAJSKI Daniel. Transaction Level Modeling: An Overview. Center for Embedded Computer Systems. University of California, Irvine.

ANNEXE A

Description de l'interface du bus OPB

L'architecture CoreConnect [10] développée par IBM fournit trois types de bus pour interconnecter entre elles les composantes d'un SoC.

- **Le bus PLB (Processor Local Bus)** C'est un bus synchrone avec un mécanisme d'arbitrage pour l'accès au bus par les maîtres. Il supporte plusieurs maîtres (un maximum de 16) et un nombre non limité d'esclaves dépendamment des ressources. Il offre de faibles latences de communication et il est utilisé pour interconnecter des composantes qui ont besoin d'une grande largeur de bande pour le transfert des données typiquement des contrôleurs de DMA, des mémoires externes et des processeurs tels que le PowerPC 405.
- **Le bus DCR (Device Control Register Bus)** C'est un bus relativement simple qui ne supporte qu'un seul maître. Il est généralement utilisé pour récupérer des statuts et passer des informations de configuration entre le processeur et d'autres périphériques du système. Il est aussi possible de l'utiliser pour des cas de tests (en angalis *design for testability*).
- **Le bus OPB (On Chip Peripheral Bus)** C'est un bus optimisé pour connecter des périphériques relativement lents (PIC, UART, etc.). Il est synchrone, supporte un maximum de 16 maîtres (avec arbitrage de l'accès au bus), et un nombre non limité d'esclaves dépendamment des ressources. Il est généralement utilisé pour connecter les composantes internes d'un SoC avec le processeur Microblaze.

Notre projet utilise une architecture basée sur le bus OPB et le processeur Microblaze. Pour cette architecture, Xilinx fournit des interfaces standards pour connecter les différentes composantes au bus OPB en maître et/ou en esclave. Un arbitrage est ajouté pour gérer les accès des maîtres au bus OPB quand leur nombre est au moins deux.

L'implémentation de Xilinx supporte deux modes d'arbitrage, le mode fixe donne accès au maître de plus haute priorité quand ce dernier le demande, alors que le mode dynamique donne accès égal aux maîtres (Round Robin). Il est possible pour un maître d'utiliser le mode rafale afin d'envoyer plusieurs données sans redemander à chaque fois l'accès au bus.

Les signaux de l'interface du bus OPB sont groupés en signaux de données, d'adresses et de contrôles. On distingue les signaux de l'interface maître, ceux de l'interface esclave et ceux du bus. Plus de détails sur l'ensemble des signaux de l'interface du bus OPB telle qu'implémentée par Xilinx sont disponibles dans le document [38].

Nous décrivons ci-dessous les signaux les plus importants et les opérations qui en dépendent [39].

- **Requête d'un maître** Un maître demande l'accès au bus OPB pour la lecture ou l'écriture (selon la valeur du signal **m_rnw**) en activant le signal **m_request**. Le bus accorde l'accès en activant le signal **opb_mgrant.** Le temps nécessaire entre les deux dépend du nombre de maîtres et de la politique d'arbitrage utilisée. Quand il s'agit d'un seul maître, il faut un ou deux cycles entre les deux dépendamment si le signal **opb_mgrant** est émis par une logique combinatoire ou séquentielle. Une fois que l'accès est donné au maître, celui-ci confirme qu'il est prêt à effectuer sa requête en activant le signal **m_select**. Ce dernier devra rester actif tant que le maître n'a pas reçu les acquittements pour le nombre de transferts demandés. L'adresse et la donnée devront être assignées respectivement aux signaux **m_abus** et **m_dbus** en même temps que l'activation du signal **m_select**. Si le signal m_select n'a pas été activé 16 cycles après le signal **opb_mgrant,** le bus active le signal **opb_timeOut** et donne l'accès à un autre maître.

  Si le maître a plusieurs données à transférer, il peut activer le mode rafale (burst transfer) en utilisant le signal **m_busLock**, ce qui évite l'attente liée à la demande du bus pour transférer plusieurs données. Le bus signale chaque transfert réussi

par un acquittement en activant le signal **opb_xferAck**. En cas d'erreur, le bus active le signal **opb_ErrAck.**

La figure A.1 illustre le protocole de demande, d'accès au bus et d'acquittement de la requête par le bus OPB dans le cas de deux maîtres.

- **Réponse d'un esclave** La requête d'un maître vers un périphérique esclave passe par le bus OPB qui sélectionne l'esclave en question selon l'adresse activée par le maître. Le signal **opb_select** est activé par l'arbitre du bus bus tant que l'esclave n'a pas répondu par un acquittement en activant le signal **Sl_xferAck** ou en activant le signal **Sl_retry** si l'esclave est lent ou occupé. Jusqu'à 16 cycles peuvent s'écouler sans reception d'un acquittement d'une requête, à partir du moment où le maître accède au bus. Dans ce cas et pour ne pas annuler la transaction, l'esclave devra activer le signal **Sl_timeOutSup** afin que le bus ne tienne pas compte de la lenteur du périphérique à répondre à la transaction. Le signal **opb_abus** est utilisé pour l'adresse du périphérique, les signaux **opb_dbus** et **Sl_dbus** sont utilisés pour la donnée respectivement lors de l'écriture ou la lecture du périphérique par le bus selon la valeur du signal **opb_rnw.**

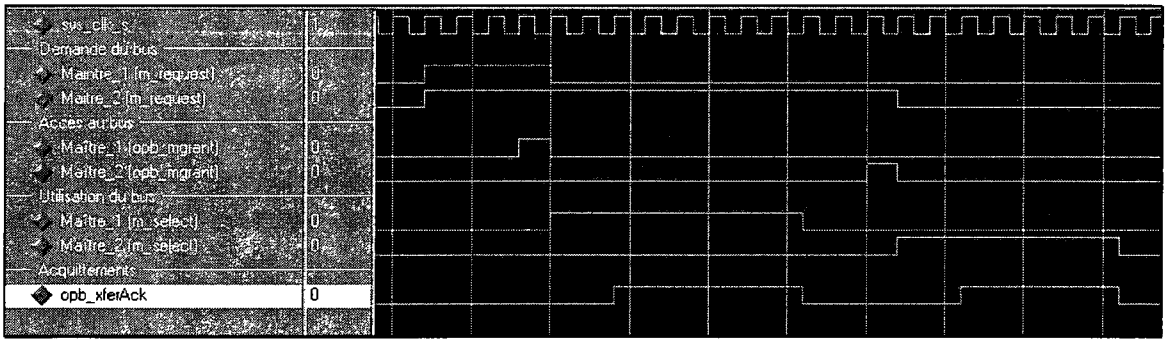

Figure A. 1 Illustration de la demande d'accès au bus OPB par deux maîtres

La largeur du signal de données est configurable et peut prendre l'une des valeurs de 8, 16, 32, ou 64 bits. Dans le cas d'une donnée dont la taille est inférieure à la largeur du signal de donnée sur l'interface du bus OPB, le signal **opb_be** (opb byte enable) permet de sélectionner les octets valides.

ANNEXE B

Description du processeur Microblaze

Le Microblaze est un processeur propriété de la société Xilinx conçu pour une utilisation optimale sur un FPGA [26]. C'est un processeur de type ***soft***, c'est-à-dire crée à partir de la logique configurable sur un FPGA. Il est basé sur une architecture RISC (Reduced Instruction Set Computer). Le Microblaze dispose d'un pipeline de 3 étages (Fetch, Decode, et Execute) et la majeure partie des instructions sont complétées en un cycle. Il peut atteindre une fréquence maximale de 150 MHz sur un FPGA Virtex4.

Le Microblaze dispose d'interfaces pour être connecté sur le bus OPB afin d'accéder aux autres périphériques du système. Il peut aussi être connecté à une mémoire interne au FPGA de type BRAM via un bus rapide LMB (Local Memory Bus).

Plusieurs autres paramètres du Microblaze sont configurables au moment de la synthèse grâce aux possibilités de configurations offertes par un FPGA. Ainsi, le processeur dispose d'un support matériel pour certaines opérations telles que la multiplication, la division et l'arithmétique des points flottants. Ce support matériel peut être activé ou non selon le besoin de l'application.

Le Microblaze peut aussi être configuré avec des interfaces pour des liaisons rapides point à point appelées FSL (Fast Simplex Link) pour y connecter des accélérateurs matériels en coprocesseur. Les interfaces FSL disponibles sont au nombre de 8 en entrée et 8 en sortie. La figure B.1 montre un exemple d'extension de l'unité d'exécution du Microblaze avec un accélérateur matériel connecté au processeur via deux liaisons FSL [25].

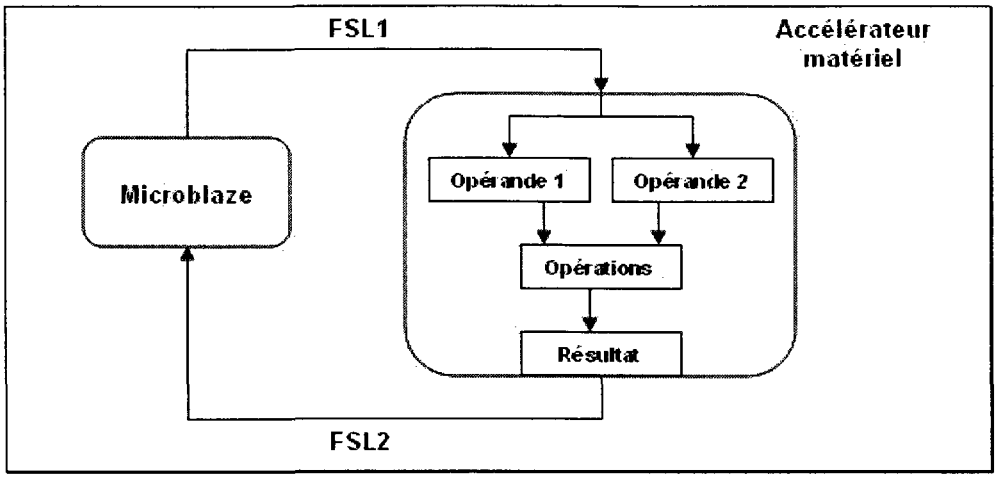

Figure B. 1 Exemple d'accélérateur matériel connecté au Microblaze par FSL

ANNEXE C

Description de l'IPIF maître/esclave

Pour standardiser et faciliter la connexion d'une composante quelconque au bus OPB, Xilinx a développé le module IPIF qui permet de réduire la complexité du protocole du bus OPB en gérant les signaux de l'interface maître et esclave au niveau du bus. Il gère aussi le décodage d'adresses locales de la composante connectée au bus. L'IPIF inclut optionnellement un gestionnaire d'interruptions, des FIFO d'envoi et de réception, et un DMA. Il peut être connecté au bus en tant que maître/esclave ou simplement esclave. Notre projet utilise le premier type et plus exactement la version **opb_ipif_v2_00_h** [29]. Nous présentons à la figure C.1 le schéma de principe de cette composante (consulter [29] pour plus de détails). Nous décrivons brièvement dans ce qui suit chacune des composantes de l'IPIF.

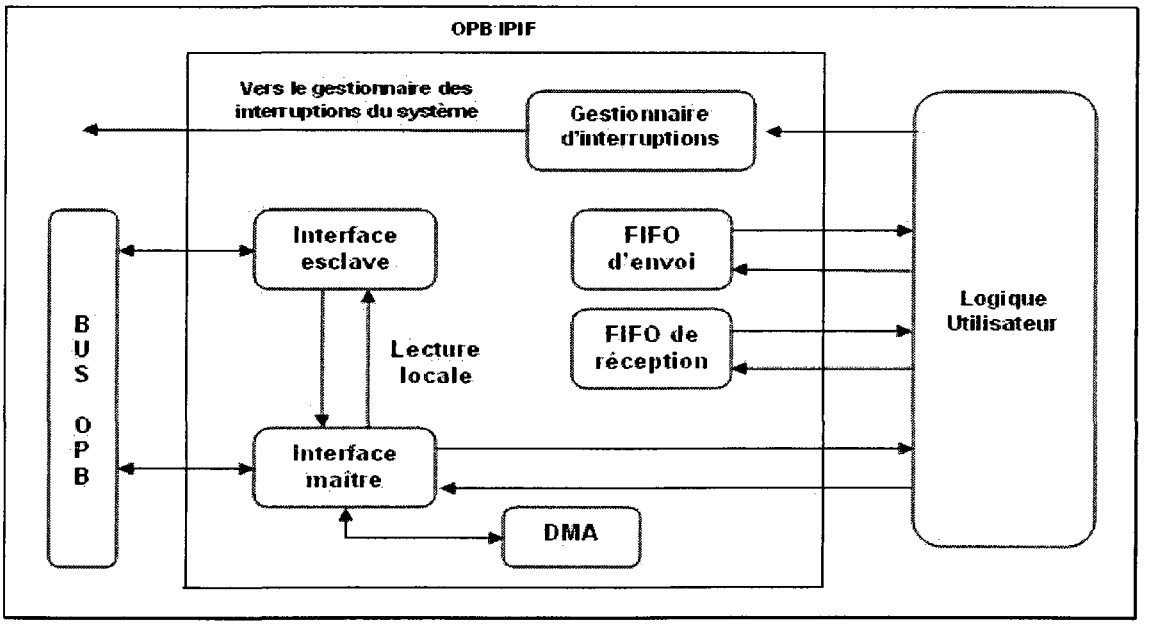

Figure C. 1 Schéma bloc de l'IPIF maître/esclave

L'interface esclave

La composante IPIF est constituée au minimum de l'interface esclave. C'est le cas d'un bloc utilisateur agissant simplement comme esclave du bus OPB et répondant aux requêtes de lecture ou d'écriture de son espace mémoire interne composé généralement d'un ensemble de registres. Cette interface gère les signaux de l'interface esclave du bus OPB et les transforme en un ensemble de signaux connectés à la logique de l'utilisateur (figure C.1). Les valeurs de ces signaux dépendent du type de la requête courante. Elle transforme aussi les signaux de sortie de la logique utilisateur en signaux de réponses compatibles avec ceux de l'interface esclave du bus. La figure C.2 représente l'interface esclave de l'IPIF avec la correspondance entre quelques signaux du bus et ceux du périphérique (logique utilisateur).

Quand l'interface esclave est adressée par le bus OPB en lecture ou en écriture, sa logique interne transforme l'adresse **opb_abus** en une adresse interne dépendamment de la largeur de son espace mémoire. Cette transformation (dont un exemple est présenté au tableau C.1) est relatée par les signaux **Bus2IP_WrCe** et **Bus2IP_RdCe.** La logique utilisateur répond à la requête en assignant correctement les signaux **IP2Bus_WrAck**, **IP2Bus_RdAck** ou autres signaux de contrôle (erreur, timeout, etc.). En cas de lecture la donnée est assignée au signal **IP2Bus_Data** quand le signal **IP2Bus_RdAck** a été activé.

Tableau C. 1 Encodage de l'adresse de destination par l'IPIF

Valeur de opb_abus (4 octets)	Valeur de Bus2IP_RdCe ou de Bus2IP_WrCe
Adresse de base	0x80000000
Adresse de base + 0x4	0x40000000
Adresse de base + 0x8	0x20000000
Adresse de base + 0Xc	0x10000000

L'exemple d'encodage du tableau C.1 représente une partie d'un encodage pour un espace mémoire local composé de 32 adresses.

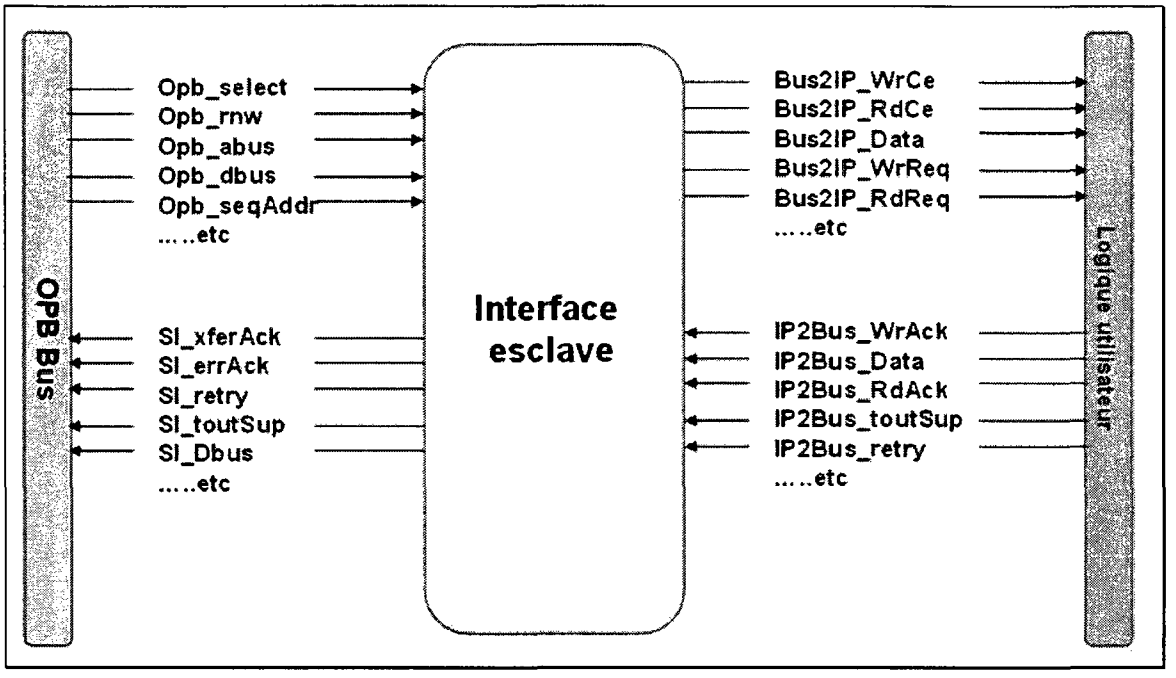

Figure C. 2 Interface esclave de l'IPIF

L'interface maître

L'IPIF fournit à l'entité logique utilisateur (figure C.1) une interface maître au bus OPB pour permettre d'initier des requêtes de lecture ou d'écriture. Cette interface maître effectue la correspondance entre les signaux de l'IP et ceux de l'interface maître du bus OPB de la même manière que l'interface esclave décrite au paragraphe précédent. La figure C.3 présente l'interface maître de l'IPIF ainsi que la correspondance entre les signaux en entrée et en sortie de l'entité logique utilisateur (figure C.1) et ceux de l'interface maître du bus OPB.

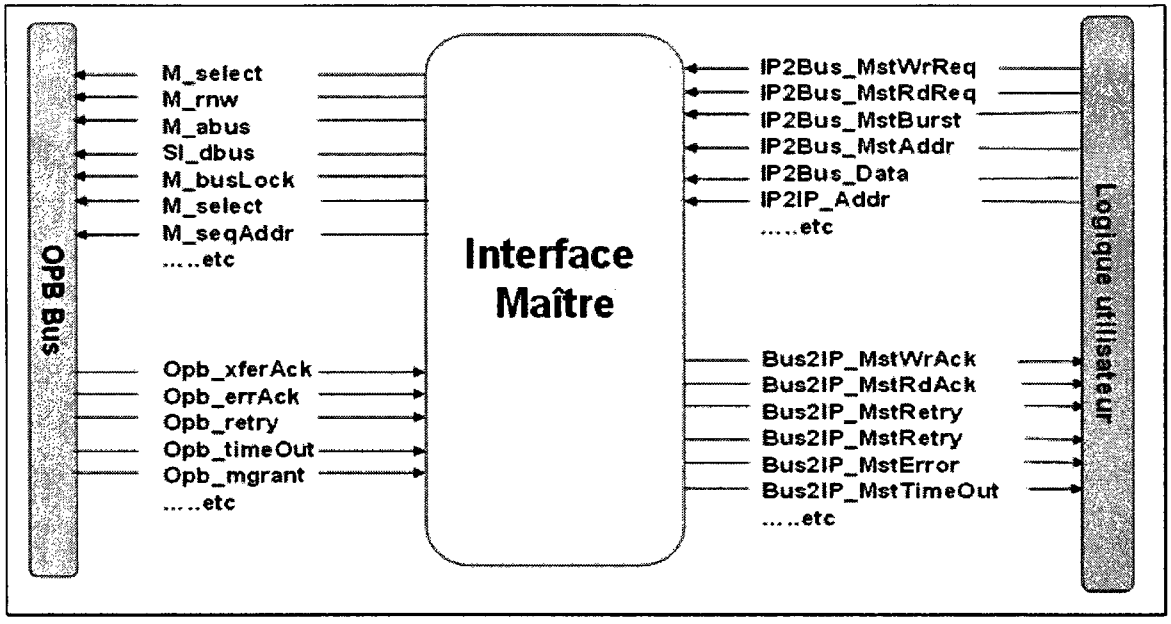

Figure C. 3 Interface maître de l'IPIF

L'entité logique utilisateur initie une requête en activant le signal correspondant **IP2Bus_MstWrReq** pour l'écriture ou **IP2Bus_MstRdReq** pour la lecture. L'assignation de l'un de ces signaux est effectuée simultanément avec l'assignation d'autres signaux pour compléter correctement la requête. À titre d'exemples **IP2Bus_MstBurst** signifie un transfert en mode rafale (burst transaction) et **IP2Bus_Addr** précise l'adresse de destination.

La connexion entre l'interface maître et l'entité logique utilisateur comporte un signal **IP2IP_Addr** (en plus de l'adresse de la destination **IP2Bus_Addr**) qui sert à préciser une adresse locale quand une requête est initiée. En effet, pour écrire des données à une autre composante du système l'entité logique utilisateur devra en premier placer ces données dans des éléments de mémoire. Par la suite l'interface maître de l'IPIF effectuera une lecture locale au à travers de l'interface esclave, placera les données dans une FIFO et démarrera ensuite l'écriture.

Pour une requête de lecture, l'interface maître initie la lecture au niveau du bus OPB à l'adresse de destination IP2Bus_Addr. Les données lues sont écrites par le bus à travers l'interface esclave à l'adresse locale **IP2IP_Addr.**

FIFO d'envoi et FIFO de réception

Il est possible d'inclure dans l'IPIF une FIFO d'envoi et/ou une FIFO de réception. La FIFO d'envoi est écrite par l'entité logique utilisateur et lue par le bus, puis inversement pour la FIFO de réception.

La profondeur, le type de mémoire, la largeur de la donnée des FIFO sont configurables. La caractéristique principale de ces FIFO est la capacité de retour en arrière quant aux opérations effectuées. Les données sont lues ou écrites de manière provisoire, et commutées ultérieurement. L'opération « *marquer*» permet de maintenir une certaine position dans la FIFO. Après la lecture la donnée et son utilisation dans une opération quelconque, la position retenue par la commande « *marquer* » est relâchée si l'application n'a plus besoin de relire la même donnée. Par contre, si l'application a besoin de relire la donnée marquée, alors la commande « *restaurer* » est utilisée afin de remettre le pointeur de la FIFO à la position marquée initialement. L'utilisation de ces fonctionnalités, par le concepteur de l'application, est optionnelle.

Gestionnaire d'interruptions

En général le mécanisme d'interruptions est utilisé dans un système pour demander au processeur un traitement spécifique. Le système dispose alors d'une composante pour gérer les demandes d'interruptions des différents périphériques du système. L'IPIF fournit un contrôleur d'interruptions locales (telles que les événements liés au fonctionnement du DMA), ou des interruptions que le concepteur de l'application peut ajouter au niveau de la logique utilisateur (figure C.1). Il est possible d'avoir 32 entrées pour les demandes d'interruptions au niveau de l'IPIF. Le contrôleur d'interruptions de l'IPIF dispose d'un ensemble de registres de contrôle et de statut. De plus, à sa sortie un signal d'interruption est connecté au gestionnaire d'interruptions du système.

DMA

L'IPIF fournit un service de DMA (direct memory access) utilisé généralement pour transférer une grande quantité de données d'une mémoire à une autre. Nous ne décrivons pas ce service qui n'est pas utilisé dans notre projet. Référez à [29] pour plus de détails là-dessus.