Titre: Title:	Conception, réalisation et étude d'une plate-forme générique basée sur le protocole AMBA AHB
Auteur: Author:	Marc Bertola
Date:	2003
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Bertola, M. (2003). Conception, réalisation et étude d'une plate-forme générique basée sur le protocole AMBA AHB [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7106/

UNIVERSITÉ DE MONTRÉAL

CONCEPTION, RÉALISATION ET ÉTUDE D'UNE PLATE-FORME GÉNÉRIQUE BASÉE SUR LE PROTOCOLE AMBA AHB

MARC BERTOLA
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

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

AVRIL 2003

Your file Votre référence
ISBN: 0-612-86381-6
Our file Notre référence
ISBN: 0-612-86381-6

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

CONCEPTION, RÉALISATION ET ÉTUDE D'UNE PLATE-FORME GÉNÉRIQUE BASÉE SUR LE PROTOCOLE AMBA AHB

présenté par : BERTOLA Marc

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

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

M. SAVARIA Yvon, Ph.D., président

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

M. AUDET Yves, Ph.D., membre

Remerciements

Ce travail a été accompli grâce aux contributions des personnes suivantes:

Guy Bois – D'abord et avant tout, merci à mon directeur de recherche qui m'a toujours soutenu, même lorsque l'inspiration n'était pas à son meilleur. Son support financer substantiel ainsi que les diverses conférences qu'il m'a payées m'ont mis en contact avec le domaine, me permettant de me développer rapidement. Merci!

Hélène et Mario Bertola – Mes parents, qui m'ont donné tant... je ne sais comment décrire leur générosité. Je suis éternellement reconnaissant, jusqu'au plus profond de moi, et j'espère pouvoir faire comme ils ont fait pour les miens plus tard... Merci!

Luc Filion – Mon compagnon en armes et soutien moral. Son amitié, sa bonté et sa générosité m'ont encouragé dans les moments creux où je ne voyais pas le bout du tunnel. Il était là pour faire évacuer la vapeur lorsque le cerveau surchauffait, mais aussi pour me rappeler à l'ordre lorsque mon cerveau devenait de la bouillie. Bref, un bon ami et parfois je regrette de ne pas avoir été suffisamment là pour lui. Merci!

Yvon Savaria – Pour les conseils techniques qui lui semblaient sans doute petits mais qui, pour moi, tenaient la clef pour déjouer mes problèmes. Merci!

Geneviève Desaulniers – La motivation pour en venir à bout... à notre beau futur, ma belle! Merciwheeeeeeeeeeeeeeeee!

Eric Yorke – Cet homme exceptionnel a tenu les rênes de certaines tâches administratives de notre compilation musicale VARIA, me laissant la chance de finir mon mémoire. Merci!

Résumé

Devant la croissance fulgurante de la densité des transistors, les concepteurs se voient obligés de modifier leurs méthodes de travail pour profiter adéquatement de ces nouvelles ressources. Certains se tournent vers la technique du raffinement progressif d'un modèle initialement très abstrait; d'autres tentent de recycler des modules d'un système à un autre, évitant ainsi d'avoir à re-concevoir un système à partir de zéro. Ce projet vient supporter la seconde approche: il s'agit de la conception d'une plate-forme modulaire et extensible, composée de divers modules qui permettent l'assemblage facile et rapide d'un système sur puce comportant un nombre arbitraire de maîtres de bus. Un second objectif du projet est de développer un bassin de connaissances sur le protocole AMBA AHB en l'utilisant comme standard de communication pour la plate-forme.

Avant de commencer la réalisation de la plate-forme, nous avons d'abord assimilé les diverses réalités de la conception par réutilisation de PI (propriété intellectuelle): La division des communications et du traitement, l'uniformisation des communications grâce aux protocoles de bus et la standardisation des interfaces (les recommandations de VCI). Nous avons ensuite étudié le protocole AMBA AHB pour nous assurer de bien comprendre tous ses aspects. Grâce au protocole, il a été possible de classer chacun des modules dans un de cinq rôles: maître, esclave, pont, arbitre et décodeur. Ce travail expose donc les résultats de cette étude sur AMBA AHB.

Plusieurs besoins ont influencé le développement de la plate-forme. Il fallait une architecture modulaire et configurable qui pouvait être générée automatiquement à l'aide d'outils informatiques. De plus, la plate-forme produite devait supporter un RTOS (système d'exploitation temps-réel) et fournir une bande passante maximale au microprocesseur qu'elle contient. Ainsi, les modules ont été regroupés ensemble dans un *bloc-processeur* qui comporte un seul bus: un microprocesseur ARM7TDMI, de la mémoire, un gestionnaire d'interruptions et une minuterie. Tous ces modules permettent le support et le bon fonctionnement d'un RTOS, tout en offrant un bus dédié au seul

maître (ARM7TDMI). Pour adapter les modules au protocole AHB, ces derniers ont été reliés au bus par des ponts (*wrappers*) qui effectuent la conversion. Afin d'étendre les fonctions du système, un pont a été ajouté afin de relier le bloc-processeur à un bus externe sur lequel se trouveraient les modules spécifiques à l'application (coprocesseurs, périphériques).

L'effort de conception a mené à l'émergence de certaines méthodologies de conception. Un outil a été créé pour générer un réseau d'interconnexion pour relier les divers modules. Cet ouvrage présente aussi les étapes pour créer des ponts pour des maîtres et des esclaves. On y trouve aussi des exemples d'arbitres.

Un système a été monté à l'aide de la plate forme. Il s'agit d'un détecteur de patrons. Le résultat du design a été très favorable: la structure modulaire permet beaucoup de flexibilité tout en offrant une performance égale aux designs *ad hoc* qui avaient été créés précédemment. Une analyse de performance présente l'effet du nombre de maîtres ou d'esclaves sur la surface et l'énergie dissipée par le réseau d'interconnexion.

Les travaux futurs incluent: l'amélioration du pont pour le ARM7TDMI, la conception d'un moyen d'initialisation réaliste pour le système et le développement de ponts reliant deux bus ayant des vitesses de fonctionnement différentes.

Abstract

Faced with the incredible increases in transistor density, designers are forced to modify their methods in order to take advantage of these new resources efficiently. Some turn towards progressive refinement, the gradual detailing of an initially abstract model. Others attempt to transplant modules from existing systems, avoiding having to rebuild a system from scratch. This project supports the second approach: it consists of the development of a modular and expandable platform, composed of several modules that allow the rapid and easy assembly of an on-chip system containing an arbitrary number of bus masters. A secondary objective of the project is to develop know-how in the use of the AHB protocol, by using it as the standard for inter-module communication.

Before building the platform, it was necessary to assimilate the various concepts related to IP (intellectual property) reuse: the separation of communication and computation, the standardisation of communications through the use of bus protocols, and the adoption of standard interfaces (such as the VCI). It was then necessary to pick apart the AMBA AHB protocol in order to ensure that it was well understood. Thanks to the protocol, it was possible to categorize the various modules into five groups: master, slave, bridge (or wrapper), arbiter and decoder. This document presents the results of this interpretation of the specification.

Many needs influenced the development of the platform. It was necessary to create a modular and configurable architecture that could ideally be generated with the aid of software. The platform also had to support an RTOS (Real-Time Operating System) and supply a lot of bandwidth to the microprocessor that it contains. The modules were grouped into a larger module, a processor-block, containing a single bus: an ARM7TDMI processor, memory, an interrupt register and a timer. All of these modules allow the support and the proper functioning of the RTOS, while offering a dedicated bus to the only master (ARM7TDMI). The interfaces of the various modules are converted to AHB through the use of wrappers. In order to allow the system to expand, the processor-block

also contains a bridge that allows it to connect itself as the master of another AHB-compliant bus to which the custom coprocessors and peripherals are also connected.

The efforts that were invested in the design brought certain methodologies into being. A tool was created to automate the generation of the interconnection network for the bus. This document also presents a set of steps to aid the design of wrappers for masters and slaves. It also offers a few examples of arbiters.

A complete system was built, using the platform: a pattern-matching algorithm. The design results were quite encouraging: the modular structure offers a lot more flexibility while offering the same performance of older, custom, *ad hoc* designs. A performance analysis presents the effect of the number of masters and slaves on both the power consumption and area of the interconnection network.

Future work includes: improvements to the AR7TDMI-to-AHB wrapper, the implementation of a realistic boot-up mechanism and the development of bridges that connect bridges with different clock frequencies.

Table des matières

Liste des tableaux

Liste des figures

Liste des acronymes

AMBA	**Advanced Microcontroller Bus Architecture**
AMBA AHB	**Advanced High-speed Bus**
AMBA APB	**Advanced Peripheral Bus**
AMBA ASB	**Advanced System Bus**
BCA	**Bus Cycle Accurate**
BP	**Bloc-Processeur**
CVE	**Co-Verification Environment**
DMA	**Direct Memory Access**
DPRAM	**Dual-Port Random Access Memory**
DRAM	**Dynamic Random-Access Memory**
DSP	**Digital Signal Processor**
FTP	**File Transfer Protocol**
MPEG	**Motion Pictures Expert Group**
PCI	**Peripheral Component Interconnect**
PI	**Propriété Intellectuelle**
RTL	**Register Transfer Level**
RTOS	**Real-Time Operating System**

SoC	**System on Chip**
SMC	**Société Canadienne de la Microélectronique**
SRAM	**Static Random-Access Memory**
SSRAM	**Synchronous Static Random-Access Memory**
TF	**Timed Functional**
UTF	**UnTimed Functional**
VCI	**Virtual Component Interface**
VHDL	**VLSI Hardware Description Language**
VSIA	**Virtual Socket Interface Alliance**

Lexique

Accès *Voir Transaction*

Arbitre Ce module cède l'accès au bus à un maître lorsque plusieurs le demandent.

AMBA **Advanced Microcontroller Bus Architecture:** Spécification comportant trois protocoles de bus servant à des systèmes SoC.

AMBA AHB **Advanced High-speed Bus:** Protocole de la spécification AMBA servant pour les bus à haute performance. Ce protocole sert généralement pour relier des modules comme des microprocesseurs au reste du système.

AMBA APB **Advanced Peripheral Bus:** Protocole de la spécification AMBA servant pour des bus à basse performance. Les possibilités d'interconnexion sont moindres, mais la complexité de la conception est plus simple.

AMBA ASB **Advanced System Bus:** Protocole de la spécification AMBA servant pour des bus à haute performance. Maintenant remplacé par AHB.

Architecture Agencement particulier de modules dans un système. Certains protocoles proposent des architectures afin de normaliser la structure des systèmes qui les respectent.

Bande Passante Débit maximal de transfert de données en une unité de temps.

BCA	**Bus Cycle Accurate:** Niveau d'abstraction d'un système où seules les transactions sur le bus marquent le passage du temps (leurs durées réelles sont ignorées)
BP	**Bloc-Processeur:** Module conçu dans ce projet de maîtrise. Il contient un microprocesseur, de la mémoire, un gestionnaire d'interruptions, une minuterie et un pont permettant de relier le bus local à un autre bus. Vu de l'extérieur, le bloc processeur ressemble à un maître qui peut de relier à un bus.
Bus	Ensemble de signaux reliant plusieurs modules ensemble, généralement selon un protocole donné.
Circuit imprimé	Circuit dont les fils sont imprimés sur une carte.
CIRCUS	Division du Groupe de recherche en microélectronique de l'École polytechnique de montréal qui se spécialise dans le domaine des systèmes sur puce.
CVE	**Co-Verification Environment:** Environnement de co-vérification. Outil permettant la simulation simultanée de matériel et de logiciel.
Décodeur	Module qui observe l'adresse émise par un maître et qui active l'esclave correspondant.
DMA	**Direct Memory Access:** Un module DMA permet un transfert de données entre deux esclaves sans passer par le microprocesseur.
DPRAM	**Dual-Port Random Access Memory:** Mémoire vive caractérisée par le fait qu'elle possède deux ports de communication. Deux accès peuvent être effectués sur elle simultanément.

DRAM — **Dynamic Random-Access Memory:** Mémoire vive dynamique. Ne requiert qu'un seul transistor, mais perd sa valeur à cause des courants de fuite si elle n'est pas rafraîchie souvent.

DSP — **Digital Signal Processor:** Un microprocesseur spécialisé dans le calcul et un haut débit de données.

Esclave — Module qui répond aux transactions initiées par un ou plusieurs *maîtres*.

Exception — Situation anormale dans un système. Requiert généralement un traitement spécial.

FTP — **File Transfer Protocol:** Protocole permettant la transmission de fichiers sur un réseau (comme l'Internet).

Interface — Ensemble des signaux d'un module qui sont présentés au monde extérieur. Lorsqu'un protocole impose une interface, il exige que certains signaux soient présents.

Interruption — Signal provoquant une réaction d'un microprocesseur. Ce dernier traite une interruption en activant une fonction de gestion, qui sauvegarde généralement le contenu des registres, effectue un traitement spécial, puis revient à la tâche interrompue.

Java — Langage de programmation interprété, conçu par Sun Microsystems, qui est indépendant du système d'exploitation sur lequel il est exécuté.

Maître — Module qui initie des transactions sur un bus (ex: microprocesseur)

Module — Ensemble de fonctions regroupées de façon logique.

MPEG	**Motion Pictures Expert Group:** Groupe d'experts en imagerie vidéo et la compression de celle-ci. MPEG désigne aussi les divers formats de compression que le groupe a créés.
Niveau d'abstraction	Ensemble de caractéristiques d'un modèle donné: plus le niveau d'abstraction est élevé, moins on considère de détails d'implantation.
Noyau	Bloc de matériel qui effectue un traitement.
PCI	**Peripheral Component Interconnect:** Protocole de communication surtout utilisé pour l'ajour de périphériques dans les ordinateurs personnels.
Périphérique	Module autre qu'un processeur, coprocesseur ou mémoire. Les périphériques sont généralement des esclaves.
PI	**Propriété Intellectuelle:** Dans le contexte de la réutilisation des modules, la propriété intellectuelle représente des modules de matériel pouvant être échangés, vendus ou transplantés dans un système.
Pipeline	Structure matérielle divisant une opération en sous-opérations dont les résultats intermédiaires sont stockés dans des registres. Un pipeline peut généralement fonctionner à une cadence d'horloge plus élevée qu'un circuit combinatoire complexe.
Plate-forme	Ensemble de blocs de propriété intellectuelle (PI) conçus pour fonctionner ensemble, servant généralement de base à laquelle un concepteur peut greffer ses propres blocs.

Pont	Module reliant deux bus distincts. Un pont reliant un bus contenant qu'un seul module à un autre bus s'appelle parfois *adaptateur* ou *wrapper*.
Port	Signal d'un module qui communique avec le monde extérieuré
Protocole	Ensemble de règles régissant la communication entre divers modules. Tous les modules qui respectent un protocole donné peuvent donc communiquer ensemble sans modification.
RTL	**Register Transfer Level:** Niveau d'abstraction bas, représentant les détails des données transmises d'un registre à l'autre. Précède le niveau *portes*, où le fonctionnement est décrit comme une suite de portes logiques.
RTOS	**Real-Time Operating System:** Système d'exploitation temps-réel. Permet la gestion de plusieurs processus s'exécutant en parallèle sur un microprocesseur.
Signal	Information transmise sur un canal. Dans le contexte de ce travail, tension sur un fil.
SoC	**System on Chip:** Système informatique dont tous les modules ou presque sont contenus sur une seule puce.
SMC	**Société Canadienne de la Microélectronique**
SRAM	**Static Random-Access Memory:** Mémoire vive statique. Ce type de mémoire demande un grand nombre de transistors, mais garde sa valeur tant qu'elle est alimentée.

SSRAM	**Synchronous Static Random-Access Memory:** Mémoire vive statique dont les transferts de données s'effectuent sur des coups d'horloge.
Système sur Puce	*Voir SoC*
TF	**Timed Functional:** Se dit d'un niveau d'abstraction élevé où seuls l'algorithmique et la séquence des événements sont considérées.
Transaction	Le fait de transmettre ou de demander une donnée à un esclave via un bus.
UTF	**UnTimed Functional:** Se dit d'un niveau d'abstraction très élevé où seule la dimension algorithmique d'un système est considérée.
VCI	**Virtual Component Interface:** Ensemble d'interfaces normalisées proposées par la VSIA pour le design de modules de matériel. Facilite grandement la réutilisation de modules.
VHDL	**VLSI Hardware Description Language**. Langage très populaire servant à décrire des modules matériels en utilisant du code.
VSIA	**Virtual Socket Interface Alliance:** Alliance d'entreprises et de chercheurs dont la mission est de faciliter la réutilisation de propriété intellectuelle (PI)

INTRODUCTION

Ce projet consiste à développer un ensemble de modules VHDL pouvant servir au prototypage rapide d'un système. Dans les sections qui suivent, le travail est situé dans le contexte des autres projets du groupe de recherche, puis les objectifs du projet sont énoncés. Les diverses sections du mémoire seront ensuite présentées brièvement.

Les projets actuels du CIRCUS

Ce projet vient répondre aux besoins de plusieurs projets au sein du CIRCUS [CIRC01], le laboratoire de recherche sur le co-design logiciel/matériel de l'École Polytechnique de Montréal. Cette section fait état des travaux qui étaient en cours au début du projet et qui ont influencé son développement.

Picasso

Au début de ce projet, le travail sur l'outil Picasso battait son plein. Le problème de la gestion de l'écart de productivité (discuté dans la section 1.1.2) a attiré beaucoup d'attention vers le domaine de la conception de systèmes sur puce (SoC, pour system on chip) qui servent au domaine des systèmes embarqués.

Picasso est un outil de co-design permettant à un concepteur d'automatiser une partie du développement d'un système embarqué. Le concepteur dessine des blocs, qui représentent divers modules soit matériels, soit logiciels et les relie ensemble comme s'ils étaient de la même nature. Les modules ont des ports qui sont reliés par des signaux. Pour chacun des blocs, le concepteur écrit soit une fonction en C, soit une description en VHDL. Pour simuler le système, Picasso prend chacun des liens, selon sa nature (matériel-matériel, matériel-logiciel, etc.), et génère automatiquement du code VHDL ou C pour réaliser des mécanismes de communication (bus direct pour les communications M-M, registres accessibles via le système de mémoire pour les communications L-M, etc.). Ainsi, le concepteur n'a pas à se soucier des détails de communication : il peut se concentrer sur les aspects fonctionnels de son système. De plus, il n'a pas à fournir de modèle matériel

pour le microprocesseur : le concepteur spécifie le nombre et le modèle des microprocesseurs, puis assigne les divers modules logiciels aux processeurs ainsi choisis. Le tout est ensuite simulé à l'aide de Seamless CVE [MeGr97], l'outil de co-vérification de Mentor Graphics.

Un problème dans la méthodologie de la première version de Picasso était qu'il ne fonctionnait que sous le système d'exploitation Windows, tandis que Seamless ne fonctionne que sous UNIX. Il fallait donc transmettre les fichiers générés par FTP au serveur UNIX, puis effectuer la simulation. Il était donc nécessaire de passer d'une machine à l'autre pour chaque modification du système, ce qui était fastidieux.

La deuxième version de Picasso a été entamée. Afin d'unifier toutes les étapes du processus de design sous un environnement de travail, l'outil a été codé en Java. Ce langage de programmation est disponible pour toutes les plates-formes principales et offre des primitives de programmation d'interfaces graphiques, simplifiant la tâche de développer une interface usager. Picasso pouvait donc fonctionner sous UNIX.

Également, on voulait ajouter un mécanisme pour effectuer l'exploration du partitionnement matériel-logiciel. En effet, le fait de coder des modules en VHDL ou en C impose immédiatement un partitionnement dès le début. L'idée d'un langage unifié pour décrire les deux a donc été proposée. Une partie du groupe de recherche s'est alors tournée vers Syslib, une méthodologie de conception système. Finalement, une architecture cible a été développée, puis éprouvée dans le cadre de la conception d'un détecteur de patrons.

Syslib

Pour permettre l'exploration du partitionnement du système, il faut être capable de décrire un comportement tout en faisant abstraction de son implantation finale. La méthodologie habituelle pour cette approche est de commencer à un niveau d'abstraction très élevé : la description du comportement se fait au niveau purement algorithmique : on ne parle donc pas de fonctions, de tâches, de jeux d'instructions, d'architectures matérielles ou de

modèle de microprocesseur. Le travail se limite à décrire seulement le comportement désiré, sans imposer de contraintes de performance. Le concepteur peut ajouter des détails par la suite en raffinant le modèle algorithmique.

Plusieurs langages permettant cette méthodologie existent maintenant, comme SystemC et SpecC, mais à l'époque, ils n'étaient pas encore suffisamment développés. Des membres du CIRCUS se sont alors tournés vers le développement de Syslib, une méthodologie qui se démarquerait par l'utilisation d'un ensemble de fonctions implantées à divers niveaux de raffinement. Ainsi, on peut développer un système à un niveau très abstrait en utilisant les primitives de communication de Syslib, puis passer à des niveaux plus raffinés en modifiant très peu le code et en associant les primitives de communication à d'autre code plus raffiné et précisé.

Au niveau le plus bas de la méthodologie, on trouve la réalisation matérielle : le système est alors appliqué à une architecture modèle, une plate-forme, qui contient tout ce qu'il faut pour utiliser un système d'exploitation. Cependant, la plate-forme qui avait été suggérée initialement pour la deuxième version de Picasso avait des lacunes. Celles-ci sont mises en relief dans la prochaine section.

Le détecteur de patrons

Pour mettre la plate-forme à l'épreuve, un système détectant un patron dans une image a été développé. Un microprocesseur ARM7TDMI génère une image et un patron, puis active un coprocesseur qui cherche le patron dans l'image. Une fois que le patron a été détecté, le coprocesseur retourne la réponse au ARM7TDMI. On assiste donc à des communications de tous les types : logiciel-matériel (écriture en mémoire des données, activation du coprocesseur), matériel-matériel (coprocesseur-mémoire), matériel-logiciel (coprocesseur-processeur) et enfin logiciel-logiciel (interaction entre divers processus sur le ARM7TDMI).

Le fonctionnement du système a été validé: les primitives de communication ont fait leur travail. Cependant, des lacunes plus sournoises ont fait surface : la difficulté de modifier

le système, notamment au niveau des mécanismes de communication. L'architecture de la plate-forme n'était tout simplement pas assez robuste aux modifications. Le changement du type de processeur aurait exigé de grosses modifications au niveau de l'architecture. L'ajout ou le retrait de modules comme des mémoires aurait le même effet.

Objectif du projet

L'objectif du travail est de développer une plate-forme matérielle configurable et hautement modulaire qui sert de cible pour Picasso et Syslib. Le modèle est codé en VHDL et a été co-simulé avec Seamless CVE.

La plate-forme doit aussi servir de laboratoire pour éprouver les principes de la réutilisation de la propriété intellectuelle, une approche envisagée pour tirer profit de la fulgurante avance des technologies. Ainsi, elle doit avoir une architecture régulière où les modules sont reliés ensemble sur un bus. Un protocole spécifique de communication a été adopté: AMBA AHB, une spécification qui permet de normaliser les interfaces des divers modules afin qu'ils puissent s'intégrer facilement à un nouveau design. Afin de convertir les interfaces non-standard des divers modules vers AHB, on aura recours à des adaptateurs, ou *ponts*. Ces modules se placent entre le module et le bus, assurant une compatibilité.

L'étude de cas du détecteur de patrons sera ensuite effectuée sur la plate-forme afin de l'étudier et de vérifier de manière informelle son comportement. L'expérience servira à souligner l'importance de certaines considérations de design. Cette expérience de conception permettra aussi de développer certaines méthodes de travail. Le texte de ce mémoire est donc aussi important que le code VHDL produit au cours du projet: C'est une chose de faire du code, mais la compréhension et l'interprétation des spécifications constitue une partie importante du travail. On peut même dire que la plate-forme a servi de prétexte pour apporter du savoir-faire dans le domaine des architectures SoC au groupe de recherche.

Contributions de ce mémoire

Ce projet de maîtrise a apporté les contributions suivantes:

- Un savoir-faire concernant les architectures de bus en général et le protocole AHB en particulier.
- Un modèle VHDL d'une plate-forme matérielle pouvant servir à la conception de prototypes de systèmes embarqués. La plate-forme peut aussi être utilisée dans un contexte didactique.
- Une méthodologie pour la conception des *wrappers* ou ponts pour les maîtres de bus AMBA.

Plan du mémoire

Le mémoire est divisé en six chapitres. Le chapitre 1, la revue de littérature, situe le projet dans le contexte actuel de la conception des systèmes sur puce. La problématique de l'écart de productivité est présentée. Une discussion suit sur les divers moyens de contrer cet écart. Enfin, on choisit une des techniques, puis on montre comment elle est appliquée au projet.

Le chapitre 2 porte sur le protocole AMBA. Plus qu'une simple transcription des spécifications, ce chapitre tente de défricher le protocole pour souligner les liens entre les signaux et d'éclaircir les intentions de certaines règles qui sembleraient un peu obscures au premier abord. Le chapitre 3 présente le contenu de la plate-forme. On cite d'abord les besoins à combler à l'aide de la plate-forme, puis on présente les divers modules qui en font partie.

Le chapitre 4 rapporte des expériences de design développées lors de la conception. On voit comment générer automatiquement un réseau d'interconnexion AHB. Des méthodologies de conception, ainsi que des exemples de machines à états complètenent le chapitre.

Le chapitre 5 décrit une expérience de conception assistée de la plate-forme, soit l'implantation du détecteur de patrons. Une comparaison est établie entre l'ancienne plate-forme et la nouvelle.

Le chapitre 6 présente certains résultats de synthèse et commente qualitativement le design d'un module critique, soit le pont AHB-AHB.

CHAPITRE 1: Revue de littérature

Tout travail de recherche est provoqué par le contexte scientifique de son époque. Avant d'entrer dans le vif du sujet, il est important de situer le travail par rapport aux problématiques actuelles afin de bien cerner en quoi il vient faire une contribution.

Le système développé dans le cadre de ce projet est la réalisation d'un modèle en VHDL d'une plate-forme de base pour le développement de systèmes qui seront implantés sur une seule puce. Cette approche sert particulièrement dans le domaine des systèmes embarqués, où l'espace occupé par les composantes est un facteur critique. En effet, ces appareils sont généralement portatifs et doivent donc être petits, légers et peu énergivores (afin de réduire la taille de la source d'alimentation, comme la pile).

Le présent chapitre présente donc les enjeux actuels de ce domaine. La section 1.1 présente le problème du fossé de la productivité. La section 1.2 montre comment réduire l'effort de conception en réutilisant des morceaux judicieusement codés de systèmes précédents.

1.1. L'écart de productivité

L'enjeu actuel du domaine de la conception de systèmes sur puce est que la technologie avance trop vite pour les concepteurs qui ne peuvent utiliser efficacement toutes les ressources qui sont à leur disposition. Cette section décrit ce problème, puis montre quelques approches de conception qui servent à le résoudre.

1.1.1. La "loi" de Moore

Gordon Moore, un des co-fondateurs de Fairchild et de Intel, énonça ce principe selon lequel la densité des transistors sur une puce double à tous les 18 mois. Ce nombre croissant de transistors permet l'intégration d'un nombre croissant de fonctions sur une même puce. L'allure de cette augmentation est schématisée dans la figure 1.1.

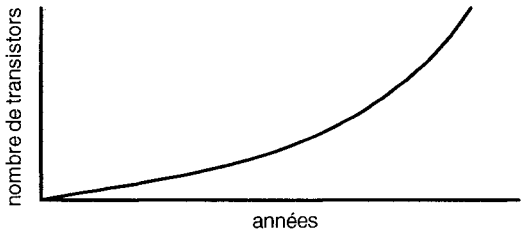

Figure 1.1 : L'allure de la loi de Moore

Cette croissance exponentielle de la densité permet de rapprocher les transistors et d'accélérer les interactions entre eux: on atteint donc des vitesses plus grandes, mais aussi une augmentation de la puissance dissipée.

1.1.2. L'écart de productivité

Pour pouvoir profiter de ces avances technologiques, il faut avoir des méthodologies de travail qui permettent de gérer un nombre aussi grand de transistors. Or, un écart existe actuellement entre le nombre de transistors disponibles et le nombre de transistors utilisables efficacement par les concepteurs. La figure 1.2 illustre ce phénomène nommé *écart de productivité*.

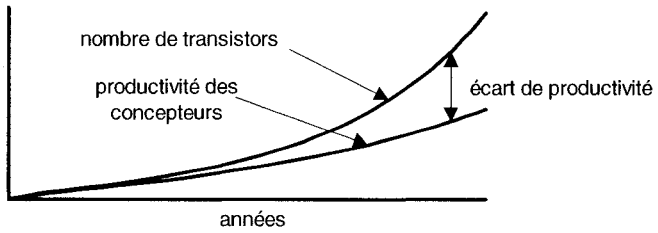

Figure 1.2 : L'écart de productivité

Beaucoup de facteurs expliquent cet écart. Dans [Nepp98], on remarque que les outils de conception assistée par ordinateur ne sont tout simplement pas assez puissants pour traiter des systèmes comportant des dizaines de millions de transistors. On note aussi qu'il n'existe pas de moyen efficace pour faire la vérification d'un système aussi complexe.

Ajouter du personnel n'aide plus après un certain seuil: trop de main-d'œuvre peut ralentir la progression d'un projet par le simple fait d'avoir à maintenir un réseau de

communication entre les membres de l'équipe de conception: c'est en quelque sorte la loi d'Amdahl appliquée à la gestion de personnel [Pres97], [Broo95].

Les méthodes de conception *ad hoc* ne tiennent plus la route. La complexité des systèmes est maintenant tellement grande que les concepteurs doivent se plier à des nouvelles méthodes. Ces techniques ne sont pas toujours les plus faciles à respecter, mais une discipline s'impose si on veut surmonter les difficultés liées aux systèmes modernes. Il est certain que la conception *ad hoc*, lorsqu'elle est appliquée à la création d'un petit système, peut donner des résultats quasi optimaux. Cependant, comme dans le domaine du placement-routage, les systèmes atteignent un niveau de complexité qui oblige le concepteur à utiliser des méthodes possiblement moins efficaces, mais plus structurées afin de réussir [GZDG00].

1.1.3. Le raffinement progressif

Deux techniques attirent actuellement l'attention des chercheurs [Fili02]. La première, le *raffinement progressif*, prône le développement sur plusieurs niveaux successifs d'abstraction. On fait abstraction des spécificités de l'implantation pour se concentrer sur un seul problème à la fois. Au niveau dit *fonctionnel* (*untimed functional* ou UTF, en anglais) aucune notion de temps n'est considérée, ce qui permet au concepteur de se concentrer uniquement sur la validité de l'algorithme à implanter. On passe ensuite au niveau *comportemental* (*timed functional*, ou TF), où une certaine notion de temporalité est introduite: l'ordre des opérations devient important et les modules doivent se synchroniser de façon réaliste.

On passe ensuite au niveau *transactionnel* (*bus cycle accurate*, BCA) où la granularité du modèle est raffiné au point de décrire abstraitement les transactions entre les modules. Enfin on trouve le niveau *horloge*, qui exprime le niveau le plus élevé de détail avant la synthèse proprement dite. C'est à ce niveau qu'il existe suffisamment de détails pour voir l'effet de l'horloge. La figure 1.3 récapitule les diverses étapes.

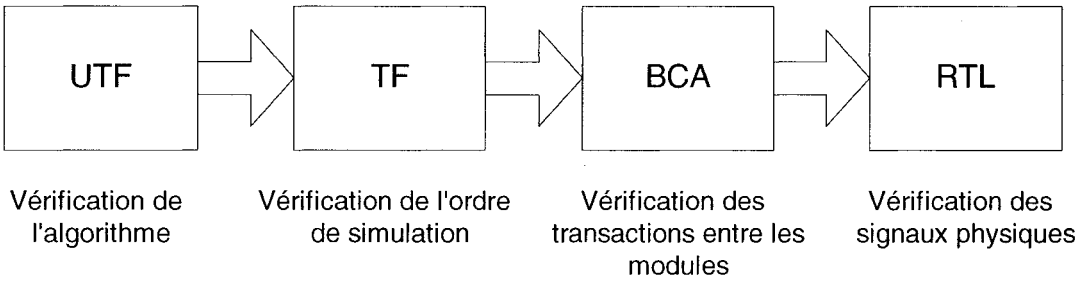

Figure 1.3 : Le raffinement progressif

Plusieurs langages de description spécialisés existent pour supporter cette technique de design. Les deux plus connus sont SystemC [OSCI02] et SpecC [GZDG00], mais il est aussi possible d'utiliser des langages comme C++ ou Java pour les niveaux d'abstraction plus élevés.

1.1.4. Réutilisation de la propriété intellectuelle

La seconde technique de conception, soit la *réutilisation de propriété intellectuelle*, suggère plutôt l'assemblage d'un système à partir d'une bibliothèque de modules matériels existants. Ces modules peuvent provenir du concepteur ou d'un fournisseur de modules de propriété intellectuelle (PI), qui vend des blocs de matériel accomplissant une tâche précise. La tâche de conception est donc réduite à celle de l'intégration qui, à ce niveau de complexité, demeure une tâche formidable, notamment au plan de la vérification. C'est cette approche qui est explorée dans ce travail.

Ces deux techniques ne sont pas mutuellement exclusives, et peuvent s'imbriquer: un fournisseur de modules de propriété intellectuelle peut offrir des modèles à tous les niveaux d'abstraction.

1.2. Enjeux de la réutilisation de la propriété intellectuelle

La réutilisation de modules de manière *ad hoc*, c'est-à-dire sans utilisation d'une norme, n'est pas en soi une solution. On veut effectuer une intégration qui requiert le moins de modifications possibles aux modules qui composent le système. Des modules mal conçus ainsi qu'une architecture mal choisie mènent à des pertes de temps causées par la ré-

ingénierie de certaines parties du système. Cette section présente les philosophies de design qui permettent de faciliter le processus d'intégration.

1.2.1. Gérer la complexité en modularisant

À la base, pour permettre la réutilisation, il faut savoir diviser le système en divers modules. De nos jours, la conception de matériel se fait à l'aide de langages de description. La logique est donc représentée par du code qui peut par la suite être simulé. Il est un fait connu dans le domaine du génie logiciel que plus on modifie du code, plus on risque d'insérer des défauts [Pres97].

La stratégie à adopter est donc de découper le système en morceaux qui ont une fonction propre, qui forment "un tout", comme les registres et la logique de contrôle d'une mémoire, par exemple. Ainsi, lorsqu'on apporte une modification au module, on ne risque pas d'introduire des erreurs dans d'autres parties du système qui n'ont aucun lien avec lui.

C'est la première étape. Généralement, les langages de description obligent le concepteur à diviser son système ainsi. C'est le cas des *entity* en VHDL.

1.2.2. Résister aux modifications en encapsulant

Un module matériel a toujours une *fonction* bien précise. En mathématique, une *fonction* est une opération effectuée sur des opérandes et qui produit un résultat. Ainsi, un module reçoit des paramètres en entrée, les traite, et donne une réponse. Les entrées proviennent d'ailleurs et la réponse est utilisée par un autre module (ou est fournie ultimement à l'utilisateur). On remarque donc qu'un module a deux aspects: un aspect *traitement* (l'opération sur les données) et l'aspect *communication* (le moyen avec lequel le module reçoit et transmet des données).

Il est intéressant de noter que ces deux aspects sont relativement indépendants: quel que soit le moyen de communication choisi, l'opération effectuée sur les données en entrée ne change pas. Puisqu'il existe une infinité de moyens pour faire communiquer deux modules, l'adaptation d'un module vers un autre système risque d'exiger certaines

modifications à l'aspect *communication*. Puisqu'on ne veut pas insérer accidentellement des défauts dans la partie *traitement*, il est intéressant de diviser les deux aspects en des modules séparés. On obtient donc le *noyau*, qui effectue le traitement et le *pont* (*wrapper* en anglais) qui s'occupe de communiquer avec le reste du système.

Ces principes ont été fortement encouragés dans la littérature, notamment dans le développement de SpecC [GZDG00] et de VCI [VSIA00].

1.2.3. Ignorer la diversité en normalisant

Bien que la division en noyau et pont soit une technique bien utile, les concepteurs doivent néanmoins produire un nouveau pont pour chaque système, qui n'est pas toujours une petite tâche en soi. Il devient alors intéressant d'établir des *protocoles de communication*, soit des règles qui normalisent les communications inter-modules. Ainsi, un module ayant un pont qui respecte un protocole donné peut être transplanté facilement vers un autre système qui respecte le même protocole.

Un fournisseur de propriété intellectuelle peut ainsi garantir l'utilité de ses noyaux en fournissant une batterie de ponts respectant les divers protocoles standards pour chacun d'entre eux. L'utilisation d'un protocole peut aussi aider dans la vérification d'un module: il existe des outils de validation pour la plupart des protocoles [NiGo01].

On remarque donc une nette amélioration par rapport à la structure initiale. La figure 1.4 récapitule l'évolution de la philosophie de conception. En (a), on voit que le noyau au complet doit être modifié pour l'adapter à un nouveau système hôte. En (b), on ajoute un pont (en blanc) qui est le seul module à être modifié lors du passage au nouveau système. Finalement en (c), on observe que l'utilisation d'un protocole de bus permet d'éviter complètement d'apporter des modifications aux modules.

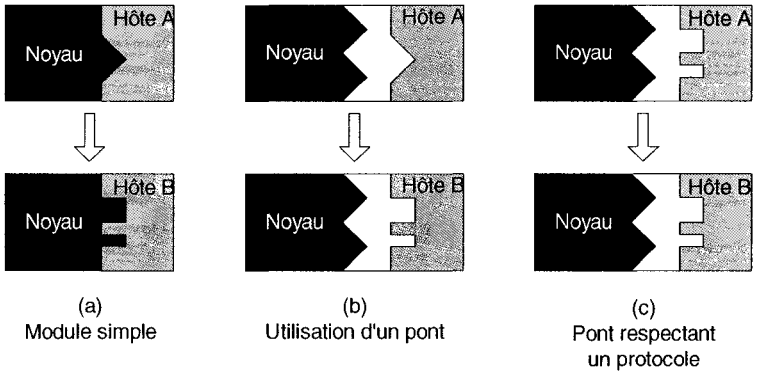

Figure 1.4 : Les raffinements facilitant la réutilisation

1.3. Les protocoles de bus

Dans la section précédente, on a soulevé l'importance des protocoles de communication. Il est donc tout naturel d'en choisir un pour ce projet de maîtrise. Cette section contient une revue des protocoles les plus populaires. Ce sont des protocoles *de bus*, c'est-à-dire des protocoles qui utilisent un bus pour transmettre des données.

Bien que les divers rôles des modules soient présentés à la section 2.1, il est important ici de résumer rapidement les concepts de base des protocoles de bus. Il existe des modules maîtres, qui initient des transferts via le bus, et des esclaves qui réagissent aux requêtes des maîtres. Pour communiquer avec un esclave précis, le maître indique l'identité du destinataire à l'aide du signal d'adresses. Généralement, il existe un module nommé décodeur qui interprète cette adresse et qui active l'esclave correspondant. Ce dernier réagit ensuite à la requête, soit en saisissant ou soit en émettant une donnée.

1.3.1. Bus sur puce vs. bus sur carte

On pourrait se demander pourquoi il existe des protocoles spécifiques au domaine des bus sur puce. En effet, des protocoles de bus comme PCI existent depuis bien longtemps pour les circuits imprimés, pourquoi ne pas les utiliser?

L'idée ici est que les bus pour les SoC ne sont pas tenus de respecter les mêmes contraintes que les bus pour les cartes (circuits imprimés). Sur les cartes, les bus relient plusieurs puces dont le coût est fortement influencé par le nombre d'entrées et de sorties

(de "pattes") qu'elles possèdent. On tente donc de minimiser le nombre de signaux et de pattes requis, permettant une plus grande densité de composantes sur la carte, et donc une carte plus petite [Cord99]. C'est pour cela que dans ces conditions, plusieurs modules voulant accéder à un signal y sont reliés directement, se mettant en haute impédance lorsqu'ils n'y ont pas accès. Un exemple classique est celui du bus d'adresses: seul le module qui a le droit de se servir du bus à un moment donné peut transmettre une valeur sur le bus. Tous les autres se mettent en haute impédance pour ne pas nuire au signal.

Les bus SoC modernes n'ont pas les mêmes contraintes que les bus sur cartes: la contrainte principale est celle des tests lors de la production. Puisque toutes les composantes sont intégrées sur une même puce, il devient difficile de faire des tests: on doit utiliser des méthodes automatiques de test. Or, ces méthodes ne sont pas compatibles avec l'utilisation de la haute impédance: les modèles de fautes se servent des valeurs "0" ou "1", pas "Z". De plus, les bus SoC n'ont pas les mêmes contraintes quant au coût des composants: le nombre de ports sur un module n'est pas un facteur aussi important au niveau coût que pour les bus sur circuit imprimé.

Les protocoles suivants sont donc mieux adaptés aux réalités des SoC.

1.3.2. AMBA

Depuis 1999, ARM propose la spécification AMBA [ARM00] en tant que norme pour les systèmes sur puce. Cette spécification offre trois protocoles qui peuvent être utilisés selon les besoins: Le premier, le *Advanced High-speed Bus* (AHB) est spécialement adapté pour les communications à haute vitesse requises par des modules performants. Le deuxième, le *Advanced System Bus* (ASB), est un protocole désuet qui avait été inspiré par les protocoles de bus sur carte et est en cours de disparition. Il a été supplanté par AHB. Enfin, il existe le *Advanced Peripheral Bus* (APB), qui sert pour les modules périphériques qui n'ont pas besoin de la performance du AHB. La simplicité du APB permet d'avoir des interfaces moins complexes et plus faciles à créer et à tester.

Le protocole AHB permet la gestion de plusieurs maîtres sur un même bus. Un algorithme d'arbitrage permet de choisir un maître à la fois afin d'éviter les collisions sur le bus. Il a été retenu pour ce projet car il est bien documenté et suffisamment flexible sans toutefois être trop compliqué. Il est aussi un des protocoles les plus populaires, étant supporté par la vaste gamme de processeurs de la société ARM. Une description détaillée se trouve au chapitre 2 du présent ouvrage.

1.3.3. CoreConnect

CoreConnect [IBM00] est un protocole conçu par IBM. Il ressemble beaucoup à AMBA. En effet, il possède lui aussi deux niveaux de complexité: le PLB (*Processor Local Bus*, l'homologue de AHB) et le OPB (*On-chip Peripheral Bus*, l'équivalent de APB). Il permet aussi les mêmes sortes de transactions. Il se différencie par certaines fonctions qui lui sont propres. Il est, notamment, possible d'effectuer une lecture et une écriture simultanément en utilisant les deux bus de données.

Le désavantage avec CoreConnect est que la structure proposée est trop lourde pour la plupart des applications actuelles. Souvent, les fonctions spécifiées par le protocole n'ont pas suffisamment d'utilité pour justifier l'effort de les implanter. De plus, la documentation est lourde, difficile à déchiffrer et aride. Il est donc difficile de l'apprendre. La société IBM, consciente de ce problème, offre la possibilité d'accéder à des "boîtes à outils" logicielles qui permettent la génération automatique du bus et la vérification de la conformité des modules qui y sont reliés.

1.3.4. Wishbone

OpenCores.org, un site Web d'où l'on peut télécharger des noyaux de code VHDL gratuitement, a choisi le protocole Wishbone comme norme. Cette spécification, publiée par la société Silicore [Sili01], propose un ensemble minimal de signaux pour permettre des accès via un bus.

Pour qu'un module soit certifié compatible avec Wishbone, la spécification exige que de la documentation soit produite pour le module (un "Wishbone Datasheet"). La

documentation nécessaire doit respecter un certain format et contenir certaines informations précises. Cette approche est intéressante: elle impose une discipline au concepteur, le forçant à documenter son travail. La réutilisation du module en est simplifiée parce que l'intégrateur n'a pas à deviner son fonctionnement.

Une autre particularité du protocole est qu'il énonce sous formes de règles précises les spécifications à respecter. En parcourant la liste des règles, on peut cerner rapidement si un module respecte la spécification ou non. Cette liste claire des règles permet au concepteur d'apprendre le protocole rapidement.

Un désavantage de cette spécification est qu'elle permet au concepteur de changer le nom des signaux qu'elle définit, dans la mesure où ces changements sont bien documentés dans la fiche technique. De plus, ce protocole ne possède pas de décodeur: il laisse aux esclaves la tâche de vérifier l'adresse afin de savoir s'ils sont sélectionnés. Si jamais les correspondances esclave-adresses doivent être changées, il faut alors modifier tous les esclaves.

1.3.5. VCI

VCI, la *Virtual Component Interface* produite par l'Alliance VSI, n'est pas exactement un protocole de bus. Il s'agit plutôt d'un protocole de communication point à point pour la communication entre un noyau et son pont. Il définit donc une interface standard pour les maîtres (*Initiators*, dans la nomenclature VCI) et les esclaves (*Targets*). Ainsi, au lieu de produire une batterie de ponts pour chaque noyau, on crée un pont par protocole de bus qui sert à tous les noyaux qui veulent s'y relier. Pour les noyaux qui précèdent VCI, la spécification propose de fournir un autre pont pour adapter leur interface à celle exigée par VCI. La figure 1.5 présente l'allure des modules qui utilisent VCI. On voit en (a) deux noyaux qui respectent le protocole VCI. Ils utilisent le même pont pour accéder au bus. En (b), on voit comment adapter des noyaux non-conformes à VCI grâce à des ponts noyau-VCI (A-VCI et B-VCI).

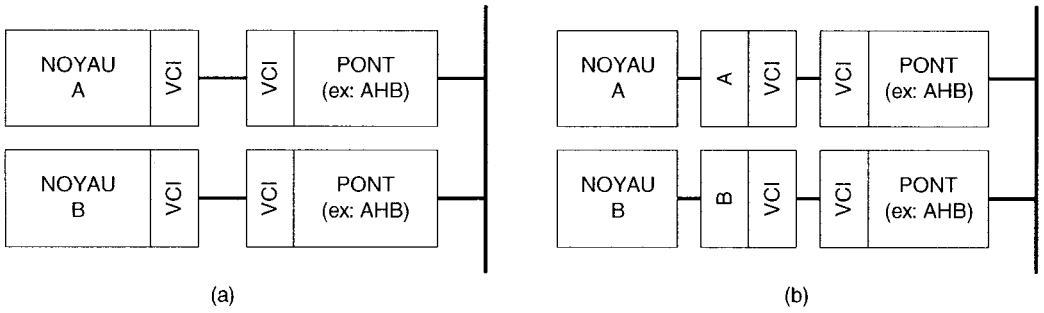

Figure 1.5 : Les interfaces VCI

La communication point à point s'effectue en envoyant des paquets. L'initiateur envoie un paquet de requête (*request*) et attend une réponse (*response*) de la part de la cible. Selon un des trois protocoles de spécification, des signaux de contrôle sont utilisés pour synchroniser les transferts.

En effet, dire "le protocole VCI" est une grossière simplification: il existe plusieurs protocoles dans la spécification, chacun ayant un degré de complexité différent. Le protocole PVCI (*Peripheral VCI*) est le plus simple, permettant des accès composés d'une requête et d'une réponse. BVCI (*Basic VCI*) permet de faire des accès éclatés, c'est à dire de transmettre plusieurs requêtes sans attendre de réponse. Enfin, le AVCI (*Advanced VCI*) apporte un nombre de fonctionnalités qui permettent d'exploiter les optimisations qu'offrent certains modèles de processeurs et de DSP: les réponses aux accès peuvent revenir en désordre et les paquets reçus peuvent avoir une taille différente de ceux qui ont été envoyés.

1.3.5. OCP

OCP est une extension du protocole VCI. Il contient encore les trois niveaux de complexité mentionnés au paragraphe précédent, mais l'effort semble être une concrétisation des idées proposées par VSIA: Il existe une suite d'outils pour vérifier la conformité de modules à OCP.

OCP a obtenu la bénédiction du VSIA et jouit actuellement d'une popularité considérable.

1.4. L'approche plate-forme

L'approche plate-forme pousse la réutilisation un cran plus loin en offrant au concepteur un squelette de système auquel il peut greffer ses propres modules. Ce squelette est composé d'un ensemble de blocs qui fonctionnent déjà ensemble selon un protocole de bus défini.

La Société canadienne de microélectronique, dans le cadre de son initiative pour la promotion de la recherche des systèmes sur puce au Canada [CMC01], s'est procuré trois plate-formes. La première est spécialisée pour la haute performance: elle contient plusieurs microprocesseurs ARM7TDMI en parallèle et des DSP pour effectuer des tâches demandant beaucoup de calcul. La deuxième plate-forme sert plutôt pour des applications de systèmes embarqués portatifs. Elle fournit une implantation de Bluetooth et consomme peu de puissance. Enfin, la SMC s'est aussi procuré une troisième plate-forme qui sert au prototypage rapide, qui contient un processeur ARM7TDMI relié à un FPGA re-programmable via un bus AMBA AHB.

Sachant que ces plates-formes seront bientôt disponibles aux universités canadiennes, il était intéressant de se préparer pour leur arrivée en développant du savoir-faire lors de la conception de la plate-forme de ce projet. Ainsi, il a été choisi que le protocole de bus utilisé dans ce travail soit le protocole AMBA AHB et que le processeur à utiliser soit le ARM7TDMI.

Dans le prochain chapitre, le protocole AMBA AHB est expliqué en détail.

CHAPITRE 2 : Le protocole AMBA AHB

Afin de garantir l'évolution de la plate-forme, il est important de bien respecter les principes de la réutilisation et de la séparation des communications et des traitements. De plus, nous souhaitons utiliser des méthodologies modernes et acceptées de tous, ce qui n'a pas été le cas des premières architectures ciblées de Picasso et Syslib. L'utilisation d'un protocole de communication reconnu est donc tout indiqué : en plus de proposer une architecture qui permet de structurer le système selon les règles de l'art, le protocole en question doit permettre au système de se relier facilement à des modules de propriété intellectuelle existants. Le choix s'est donc arrêté sur le protocole AMBA AHB parce qu'il est moderne, bien documenté et populaire.

Ce chapitre décrit donc ce protocole, en tentant d'éviter la simple traduction de la spécification. L'idée ici est de synthétiser les informations pour mettre en relief les relations entre les acteurs et les signaux qu'ils utilisent, de bien décortiquer les spécifications pour montrer l'intention qui les motive, ce qui n'est pas toujours apparent à la première lecture. D'abord, les divers rôles des modules seront présentés. On verra ensuite les différences entre les multiples protocoles de la norme AMBA. Une présentation des signaux du protocole AHB suivra, puis viendront les différentes opérations dans lesquelles ces signaux interviennent. Le chapitre se terminera avec une courte discussion des divers moyens d'augmenter la bande passante. Il est à noter que bien que ce chapitre consiste en une introduction à la spécification AMBA AHB, une lecture assidue de cette dernière est fortement recommandée à tout concepteur voulant utiliser le protocole.

2.1. Les acteurs dans les protocoles de bus

Avant de plonger dans les spécificités du protocole AMBA, il est important d'identifier les différentes catégories d'acteurs qu'il régit. Cette section décrira les cinq types d'acteurs : maître, esclave, arbitre, pont et décodeur. Il est bien important de noter qu'*acteur* n'est pas un synonyme de *module* : on ne parle pas ici de blocs de matériel

mais bien de *rôles* dans les transactions sur le bus: un module peut être à la fois un esclave et un maître, par exemple, mais il doit avoir des interfaces séparées pour chaque rôle. Selon le point de vue du bus, il y a deux entités à sa place : un maître et un esclave. La figure 2.1 montre un exemple d'un tel module.

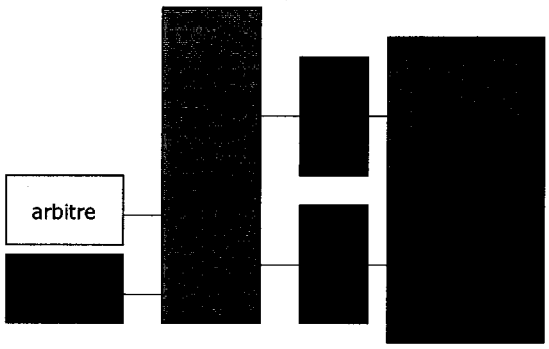

Figure 2.1 : Un module qui est à la fois maître et esclave

2.1.1. Maître

Un maître est une entité qui initie des transactions sur le bus : il pilote le bus d'adresses, émet des données lors des écritures et il reçoit des données lors des lectures. Des exemples de maîtres incluent : les microprocesseurs, les DMA et la partie des caches qui s'occupe de lire dans les mémoires distantes.

Le maître a le devoir d'attendre l'autorisation de l'arbitre avant de procéder à un accès sur le bus. Afin de tirer le maximum du bus AMBA AHB, il est attendu que le maître fasse des rafales d'accès, suivies de périodes d'attente. De cette façon, on évite la latence initiale que certains esclaves imposent pour le premier accès (comme dans le cas des contrôleurs de DRAM [Kozi01]).

2.1.2. Esclave

Un esclave est la contrepartie du maître. Lui seul peut répondre aux requêtes provoquées par les maîtres. Des exemples typiques d'esclaves incluent : les mémoires, les ports de communication, la partie des caches qui est reliée au processeur, les périphériques et certains coprocesseurs.

2.1.3. Arbitre

L'arbitre est utilisé dans des situations où plusieurs maîtres tentent d'effectuer des accès en même temps sur un bus qu'ils partagent. Il décide du maître qui a accès au bus à un moment donné. Le protocole AHB ne définit aucune politique d'arbitrage : le concepteur est libre de choisir le comportement de l'arbitre, dans la mesure ou les signaux respectent la spécification.

2.1.4. Décodeur

La tâche du décodeur est d'observer le bus d'adresses et de sélectionner l'esclave correspondant à l'adresse courante. Le décodeur doit être conçu de façon simple afin de représenter le plus petit délai possible, puisqu'il se trouve sur le chemin critique du bus.

2.1.5. Pont

Au sens strict, un pont fait le lien entre deux bus. Souvent, les deux bus utilisent des protocoles différents. Lorsqu'un maître sur un bus veut accéder à un esclave d'un autre bus, le pont traduit la requête d'un protocole à un autre. Du côté du maître, le pont joue le rôle d'un "super-esclave" qui fait abstraction des esclaves sur le bus distant. De l'autre côté, le pont est un maître qui effectue l'accès à la place du maître qui a initié la transaction.

La littérature [GZDG00] donne le nom spécial de *wrapper* aux ponts qui effectuent une traduction pour un seul module (le bus en question devient alors un bus très simple contenant deux acteurs : le pont et l'autre module). Ce type de pont est très utile dans le domaine de la réutilisation : le pont s'occupe des spécificités de la communication, tandis que le noyau se limite aux opérations. Ainsi, si on veut réutiliser le noyau dans un système ayant un protocole différent, il suffira de changer le pont, sans avoir à toucher le noyau, ce qui réduit le temps d'intégration du module dans un autre système, tout en minimisant les erreurs.

2.2. Les familles de protocoles AMBA

La spécification de AMBA définit trois protocoles sous la norme AMBA. Il s'agit des protocoles AHB (Advanced High-speed Bus), ASB (Advanced System Bus) et APB (Advanced Peripheral Bus). Les deux premiers protocoles, AHB et ASB, servent à relier des modules qui requièrent un maximum de performance. Le dernier, APB, sacrifie de la flexibilité pour faciliter la conception de modules simples pour lesquels l'effort et la surface requis pour implanter un des protocoles plus complexes viendraient alourdir inutilement le noyau.

2.2.1. AMBA ASB

Le protocole ASB fut le premier bus système à être proposé sous la norme AMBA. Ce protocole offre une grande flexibilité pour des modules à haute performance : la possibilité d'avoir plusieurs maîtres sur un même bus grâce à un arbitre, des accès en deux phases (pipelinés) et des transferts de données de tailles variables (octet, demi-mot, mot). La figure 2.2 montre l'architecture proposée par la spécification :

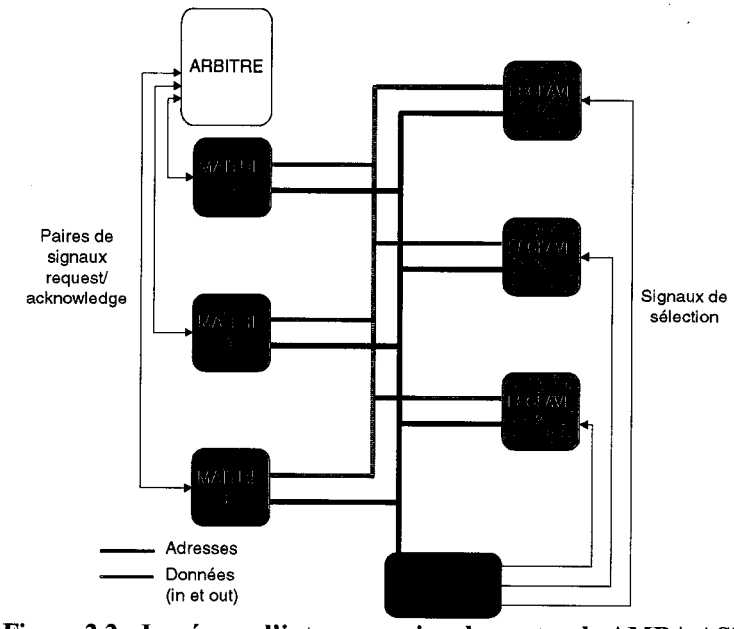

Figure 2.2 : Le réseau d'interconnexion du protocole AMBA ASB

Le bus d'adresses et le bus de données de tous les modules sont tous reliés ensemble. Seul le maître qui a actuellement accès au bus ainsi que l'esclave actuellement sélectionné par le maître ont le droit de piloter les signaux. Tous les autres modules doivent mettre leurs sorties en haute impédance.

La spécification du protocole ASB fut inspirée des protocoles utilisés pour des bus sur des cartes [Cord99]. Ces derniers sont conçus de façon à minimiser les coûts de la carte : le nombre de signaux est limité de façon à réduire le nombre de pattes requises sur les composantes (les pattes représentent un coût élevée dans la fabrication des composantes). Dans la même optique, ces protocoles de carte utilisent généralement un seul bus de données pour les lectures et les écritures.

2.2.2. AMBA AHB

AHB est une version améliorée qui vient remplacer ASB. Le réseau d'interconnexion a été repensé de façon à exploiter le fait que les limitations imposées aux bus sur cartes n'existent pas pour les bus sur puce. On utilise des multiplexeurs pour éliminer les états de haute impédance sur le bus, ce qui permet de produire des systèmes plus faciles à tester et à analyser de façon statique. En effet, les outils de test automatisés ont plus de facilité à tester le fonctionnement d'un multiplexeur que celui d'un module en haute impédance [LSIL98]. La figure 2.3 montre l'allure du nouveau réseau d'interconnexion.

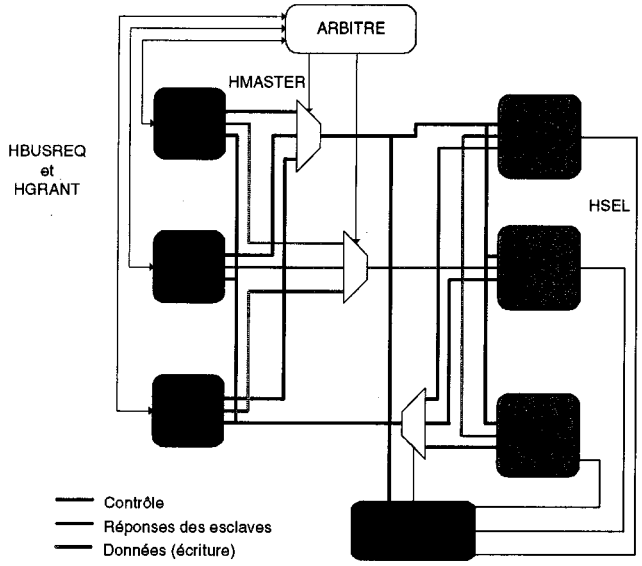

Figure 2.3 : Le réseau d'interconnexion de AMBA AHB

On remarque d'abord comment les bus provenant des différents modules viennent se connecter à des multiplexeurs : ils ne sont plus connectés directement ensemble. Tous les modules pilotent des valeurs sur leurs bus, mais les multiplexeurs ne transmettent qu'une seule des entrées selon le module qui a accès au bus. Les modules n'ont donc plus à se déconnecter à l'aide d'une sortie en haute impédance : le multiplexeur s'occupe de les isoler les uns des autres. L'arbitre du bus s'occupe de gérer le multiplexeur du bus de contrôle et celui du bus de données d'écriture. Le décodeur contrôle le multiplexeur des réponses des esclaves. Tous les esclaves reçoivent les informations provenant du bus d'adresses et de contrôle, mais seul l'esclave sélectionné par le décodeur peut s'en servir (les autres demeurent inertes).

Le protocole AHB vient aussi ajouter la possibilité aux esclaves de répondre à des requêtes et transactions de manières différentes. Un esclave peut indiquer une erreur, demander au maître d'essayer un accès à nouveau s'il n'est pas prêt ou bien suggérer à l'arbitre de céder le bus à un autre maître en attendant la complétion d'un accès de longue durée.

2.2.3. AMBA APB

Le protocole AMBA APB existe pour des périphériques qui sont tellement simples ou lents que l'implantation de AHB ou ASB viendrait augmenter grandement leur complexité et leur surface pour rien. Un bus APB viendrait se connecter via un pont à un bus AHB. Il n'y a aucune provision pour plusieurs maîtres, alors on élimine l'arbitre sur ce bus.

Le pont AHB / APB fait le lien entre le bus périphérique et le bus système. Du côté système, le pont a une interface d'esclave. Il fait donc abstraction du bus derrière lui, devenant une sorte de gros module esclave qui gère tous les accès vers le bus périphérique. On utilise encore une structure à plusieurs multiplexeurs.

2.3 Les signaux AHB

Avant de pouvoir comprendre les diverses règles d'interaction définies dans le protocole AHB, on doit d'abord comprendre l'utilité de chacun des signaux qu'il spécifie. Cette section décrit ces derniers, en les regroupant de façon à bien comprendre leurs interdépendances. Cette section du mémoire est fortement liée à la prochaine : il est difficile d'expliquer l'utilité d'un signal sans faire référence aux transactions dans lesquelles il est impliqué, et il faut naturellement connaître les signaux pour comprendre les transactions. Cette section fait donc appel aux notions de cycle de bus et d'accès pipelinés. Il suffit de savoir pour l'instant qu'un cycle de bus est une période de temps de durée arbitraire au cours de laquelle une transaction est effectuée sur le bus. Ces transactions s'effectuent en deux phases (les données sont transmises au cours du cycle de bus suivant la transmission des signaux de contrôle, tels que le signal d'adresse). On dit donc que les accès s'effectuent de façon pipelinée. Des détails plus précis concernant ces notions sont disponibles à la section 2.4.

2.3.1. Signaux communs

Les signaux communs servent à la synchronisation et à l'initialisation d'un système. On trouve HCLK, l'horloge globale, et HRESETn (le "n" indique que le signal est actif bas), le signal de mise à zéro du système.

2.3.2. Signaux de contrôle

Tableau 2.1 : Les signaux de contrôle

SIGNAL	FONCTION
HADDR(31:0)	Identifie l'esclave avec lequel le maître veut communiquer. Sert aussi à accéder à une sous-section d'un esclave (des mots en mémoire, par exemple)
HWRITE	Indique la direction de l'accès : 1 = Écriture, 0 = Lecture
HSIZE(2:0)	Indique la taille d'un accès. Il est parfois intéressant de faire des accès qui affectent un nombre de bits inférieur à la largeur du bus (modifier un octet d'une chaîne de caractères, par exemple).
HTRANS(1:0)	Indique l'intention du maître 00 (IDLE) : Le maître n'accède pas au bus au cours de ce cycle 01 (BUSY) : Le maître est actuellement occupé et continuera l'opération en cours sous peu 10 (NON-SEQUENTIAL) : Le maître commence une rafale 11 (SEQUENTIAL) : Le maître poursuit une rafale.
HBURST(2:0)	Indique la nature de la rafale d'accès qui est en cours : la durée et le comportement sur des frontières d'adresses.
HPROT(3:0)	Indique des informations au sujet de l'accès qui aident notamment les caches et les tampons d'écriture (voir la section 2.5.4)

Le **maître** génère les signaux d'adresse et de contrôle pour exprimer ses intentions. Tous les maîtres génèrent ces signaux en même temps, mais seul les signaux du maître sélectionné par l'arbitre passeront le multiplexeur de contrôle.

L'**esclave** utilise généralement les bits les moins significatifs de HADDR(31:0) pour sélectionner ses registres internes. Il utilise HWRITE pour déterminer l'action à prendre par rapport à l'accès (saisir ou émettre une donnée). Il utilise une combinaison de HSIZE(2:0) et les bits les moins significatifs de HADDR(31:0) pour déterminer quels octets activer pour un accès donné ([ARM01], page). L'esclave doit aussi prendre note du signal HTRANS(1:0) : Si le maître fait un accès IDLE ou BUSY, il doit ignorer l'accès, car le maître n'utilise pas le bus au cours de ce cycle-là. HPROT(3:0) donne aussi des informations au système de mémoire et permet certaines optimisations.

HPROT(0) (l'accès est pour chercher une instruction ou non) peut servir pour sélectionner une cache d'instructions plutôt qu'une cache de données. HPROT(1) (mode usager versus mode superviseur) peut limiter l'accès à des zones sensibles de mémoire pour des raisons de sécurité. HPROT (3) (peut être mis en cache) et HPROT(2) (peut être mis en tampon) peuvent servir pour contrôler une cache. Ces deux derniers bits de HPROT(3:0) peuvent ainsi servir pour indiquer qu'une lecture sur un port de communication (dont la valeur peut changer indépendamment des actions du microprocesseur) ne peut être mise en cache, par exemple.

L'**arbitre** utilise facultativement ces signaux. HBURST(2:0) lui donne une indication du nombre d'accès faisant partie de la rafale en cours. Cette information lui permet de choisir un moment judicieux (soit la fin de la rafale) pour céder le bus à un autre maître qui serait en attente du bus. Les autres signaux peuvent servir selon la politique d'arbitrage choisie.

Le **décodeur** observe les bits plus significatifs pour activer le signal HSEL (Tableau 2.3) de l'esclave correspondant à la plage d'adresses spécifiée. Il ignore les autres signaux de contrôle.

Le **pont** transmet et traduit ces signaux pour accéder au bus distant.

2.3.3. Signaux de données

Tableau 2.2 : Les signaux de données

SIGNAL	FONCTION
HRDATA(31:0)	Bus de données pour les lectures (esclave vers maître)
HWDATA(31:0)	Bus de données pour les écritures (maître vers esclave)

Tous les **maîtres** émettent leurs données d'écriture sur HWDATA(31:0) en même temps. Seul le signal du maître sélectionné par l'arbitre passera le multiplexeur des données d'écriture.

Tous les **esclaves** émettent leurs données de lecture sur HRDATA(31:0) en même temps. Seul le signal de l'esclave sélectionné par le décodeur passera le multiplexeur des réponses des esclaves.

2.3.4. Signaux de sélection

Ces signaux servent à contrôler les multiplexeurs de l'architecture AHB.

Tableau 2.3 : Les signaux de sélection

SIGNAL	FONCTION
HSELx	Chaque esclave (identifié par un nombre au lieu du "x") possède un signal HSEL qui l'active s'il est sélectionné. Il y a autant de signaux HSEL que d'esclaves. Une copie de ces signaux, retardée d'un cycle de bus, sert à contrôler le multiplexeur des réponses des esclaves.
HBUSREQx	Chaque maître (identifié par un nombre au lieu du "x") a un signal HBUSREQ avec lequel il indique son intention d'effectuer un accès (généralement une rafale d'accès) sur le bus.
HLOCK	Signal activé en même temps que HBUSREQ indiquant que la série d'accès qui suit doit se faire de façon ininterrompue (atomique). L'arbitre n'a donc pas le droit de céder le bus à un autre maître tant que HLOCK est actif.
HGRANTx	Signal activé par l'arbitre pour indiquer au maître (identifié par un nombre au lieu du "x") que le bus lui sera cédé à la fin du prochain cycle de bus. Le signal sera désactivé pour indiquer au maître qu'il perdra le bus au prochain cycle de bus (voir la section 2.3.5 pour une définition des cycles de bus).
HMASTER(3:0)	Signal indiquant l'identité du maître qui a actuellement le bus. Il sert de signal de sélection pour le multiplexeur de contrôle (voir section 2.2.2). Une version retardée d'un cycle de bus sert à contrôler le multiplexeur d'écriture.
HMASTLOCK	Indique si le maître actuel est en cours d'exécution d'une rafale atomique d'accès.

Pour avoir accès au bus, un **maître** active le signal HBUSREQ qui lui correspond. Ce signal restera activé jusqu'à ce qu'il n'ait plus besoin du bus.

Tous les **esclaves** reçoivent continuellement les informations de contrôle, mais seul l'esclave dont le HSEL = 1 devra réagir. Tous les autres esclaves doivent ignorer l'accès. Si HMASTLOCK est actif, un esclave ne peut pas émettre la réponse HSPLIT (voir section 2.3.5), car le maître ne doit pas perdre le bus. HMASTLOCK peut aussi servir à des esclaves ayant plusieurs ports. Il faut s'assurer que des maîtres sur les autres ports n'essaient pas d'accéder aux zones affectées par l'accès atomique. Si un maître effectue une suite d'accès dans une mémoire multiport, par exemple, la mémoire doit protéger les

adresses impliquées par l'accès atomique des maîtres accédant a ses autres ports. Un esclave se sert de HMASTER(3:0) lorsqu'il donne une réponse HSPLIT(15:0) à un accès donné, car il doit noter le numéro du maître qu'il devra réveiller plus tard.

Le **décodeur** génère les divers HSELx (HSEL0, HSEL1, HSEL2...) à partir de HADDR(31:0). Le concepteur du système décide de la correspondance adresse/esclave, puis génère le décodeur en fonction de cette correspondance. Il en résulte que le décodeur est spécifique à chaque système conçu.

Le fonctionnement de l'**arbitre** est fondé sur cet ensemble de signaux. À chaque cycle de bus, l'arbitre sélectionne, parmi les maîtres qui ont activé leur signal HBUSREQ, celui qui aura accès au bus. Ce choix repose sur l'algorithme de sélection qui est spécifique à chaque système. Lorsque l'arbitre a élu le possesseur du bus pour le prochain cycle de bus, il avertit ce dernier en activant le signal HGRANT. Au prochain cycle de bus, le maître sait qu'il a maintenant accès au bus (grâce à HGRANT). L'arbitre laisse passer les signaux de contrôle de ce dernier à travers le multiplexeur de contrôle grâce au signal HMASTER(3:0). L'arbitre doit ensuite observer le signal HLOCK du maître choisi. Tant que le signal demeure actif, le bus ne doit pas être cédé à un autre maître. HMASTLOCK est une copie du signal HLOCK du maître sélectionné.

Les **ponts** se servent de ces signaux selon qu'ils sont des maîtres ou des esclaves du bus AHB. Si les deux bus ont des correspondances esclave-adresse différentes, le pont a la responsabilité de faire la traduction des adresses (changer le signal HADDR(31:0) pour que l'adresse vienne sélectionner le bon esclave sur le bus distant).

2.3.5. Signaux de réponse

Ces signaux proviennent de l'esclave et servent à exprimer le résultat d'un accès et à étirer la durée d'un cycle de bus. HRESP(1:0) et HREADYOUT passent par le multiplexeur de réponse des esclaves qui est contrôlé par HSEL.

Tableau 2.4 : Les signaux de réponse

SIGNAL	FONCTION
HRESP(1:0)	Sert à indiquer le résultat d'une transaction : 00 (OK) : La transaction a été réussie sans problème 01 (ERROR) : Il y a eu une erreur lors de la transaction. 10 (RETRY) : L'esclave n'est pas disponible pour compléter l'accès. Le maître doit recommencer. 11 (SPLIT) : L'esclave a besoin de plusieurs cycles avant de pouvoir traiter la transaction. Le bus est cédé à un autre maître en attendant que l'esclave ait fini de traiter la demande.
HSPLIT(15:0)	Ce signal permet à un esclave, lorsqu'il a terminé un accès auquel il avait répondu HRESP = SPLIT, de signaler à l'arbitre de redonner le bus au maître qui l'avait perdu. La largeur maximale de ce signal est de 16 bits, mais la largeur réelle est égale au nombre de maîtres dans le système.
HREADY	Tous les modules observent HREADY afin de connaître la fin du cycle de bus (indiqué par HREADY = 1).
HREADYOUT	La spécification de AMBA indique que HREADY est à la fois une entrée et une sortie des esclaves. La foire aux questions de AMBA [ARM01d], un document explicatif fournissant des précisions sur le protocole, suggère plutôt l'utilisation du signal HREADYOUT comme le signal de sortie, afin que HREADY soit un signal d'entrée seulement. Ce signal, relié à HREADY, permet à un esclave de contrôler la durée des cycles de bus.

Parfois, si un **esclave** a besoin de plusieurs cycles d'horloge pour compléter un accès, il peut étirer le cycle de bus en tenant HREADYOUT = 0. HREADY prend la valeur du HREADYOUT émis par l'esclave sélectionné. Le début d'un cycle de bus est donc défini par un coup d'horloge où HREADY = 1.

Les **esclaves** utilisent HRESP(1:0) pour indiquer le résultat d'une transaction. Si tout va bien, un esclave doit émettre la réponse 00 (OK). Dans le cas où le maître aurait tenté une opération illégale (adresse invalide, séquence d'accès incorrecte), l'esclave peut indiquer une erreur avec la réponse 01 (ERROR). La valeur 10 (RETRY) sert à indiquer au maître que l'esclave n'est pas prêt à traiter l'accès immédiatement : le maître est invité à réessayer immédiatement. La réponse 11 (SPLIT) est semblable à RETRY : l'esclave peut s'en servir pour indiquer qu'il n'est pas disponible pour l'instant ou qu'il aura besoin d'un grand nombre de cycles pour compléter l'accès. La nuance ici est que la réponse SPLIT vient avec la promesse que l'esclave avertira le maître lorsqu'il sera disponible. L'arbitre donne donc le bus à un autre maître en attendant la réponse de l'esclave, permettant une utilisation plus efficace de la bande passante. L'esclave avertit l'arbitre

qu'il est redevenu disponible grâce au signal HSPLIT(15:0). Chacun des 16 bits de ce signal est assigné à un maître différent. L'esclave active le bit correspondant au maître à réveiller (l'esclave sait quel bit activer grâce au signal HMASTER(3:0) qu'il avait échantillonné au moment où il avait émis la réponse SPLIT). L'arbitre cède le bus au maître correspondant. HSPLIT(15:0) a donc une priorité sur les requêtes "normales" de bus effectuées par HBUSREQ. Si plusieurs bits de HSPLIT(15:0) sont activés en même temps, l'arbitre cède le bus à un des deux maîtres selon un algorithme défini par le concepteur.

Les **maîtres** doivent observer HREADY afin de suivre la progression des cycles de bus. Si HREADY = 0 au début d'un cycle d'horloge, le maître doit tenir les valeurs qu'il pilotait au cycle précédent. Dès qu'un maître détecte une réponse RETRY ou SPLIT à la fin d'un cycle d'horloge, il doit émettre HTRANS(1:0) = IDLE dès le cycle d'horloge suivant, puis réessayer le transfert qui avait provoqué la réponse exceptionnelle dès le prochain cycle de bus (ce qui correspond au prochain cycle d'horloge, en général). Si l'accès est SPLIT, le maître perd le bus avant d'avoir la chance de réessayer : il doit donc attendre que l'arbitre lui redonne le bus.

L'**arbitre** doit aussi observer HRESP(1:0). Dès que HRESP(1:0) = SPLIT au début d'un cycle d'horloge, il doit tout de suite donner le bus à un autre maître dès le prochain cycle de bus.

Les réponses HSPLIT(15:0) sont difficiles à gérer lors de la conception de **ponts** qui ont un bus AHB sur leur côté esclave. Une description plus approfondie de cette problématique est donnée au chapitre 5.

2.4. Les interactions dans le protocole AMBA AHB

Les spécifications de protocoles sont parfois bien arides et il est souvent difficile d'en tirer une vue d'ensemble sans mettre la main à la pâte. Cette section décrit les différentes interactions entre les modules, particulièrement leurs moyens de synchronisation et leurs méthodes de communication. On verra d'abord les concepts de base : les accès pipelinés

et les processus d'arbitrage. Viendront ensuite les notions plus avancées : la gestion des exceptions et les accès atomiques.

2.4.1. Les accès pipelinés

Le processus de pipelinage consiste à diviser une tâche qui s'effectuerait en un long cycle d'horloge en plusieurs sous-tâches qui prennent chacune un court cycle d'horloge. L'avantage est double : on permet à plusieurs étapes d'un même traitement de s'effectuer en parallèle et moins de traitement est effectué par cycle, permettant une cadence d'horloge plus élevée. Cette amélioration de la performance vient à un prix : le premier traitement n'est disponible qu'après un certain nombre de coups d'horloge. Cependant, au cours du premier passage, les traitements subséquents progressent eux aussi dans les autres étapes du pipeline. Donc après la latence initiale, on obtient un nouveau résultat à chaque cycle d'horloge.

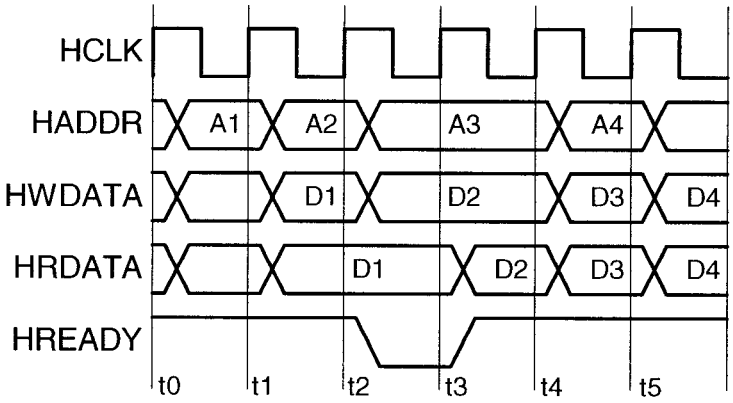

Figure 2.4 : Accès pipelinés

La figure 2.4 montre comment les accès du protocole AMBA AHB sont divisés en deux afin de profiter des avantages du pipelinage. On remarque un ensemble d'accès numérotés de 1 à 4. Une transaction comporte une phase d'adresses (A) et une phase de donnée(D), où chaque phase a une durée d'un cycle de bus (les phases peuvent donc avoir des durées variables selon la durée des cycles de bus). Au cours de la première phase d'adresses (A1), le maître pilote les signaux de contrôle. On ne voit ici que HADDR(31:0), mais les autres signaux de contrôle (HWRITE, HSIZE(2:0), etc.)

changent en même temps. Au cours de cette phase, le décodeur active le signal HSEL correspondant à l'esclave indiqué par l'adresse. Au prochain cycle de bus, c'est à dire lorsqu'il y a un coup d'horloge où HREADY = 1, la phase de données D1 commence : les données sont transmises. L'esclave sélectionné doit saisir les informations de contrôle de la phase d'adresses 1, car au cours de la phase de données de l'accès 1, le maître lance la phase d'adresses de l'accès 2 en parallèle.

Il est à noter que l'esclave a le devoir d'échantillonner les données de contrôle à la fin de la phase d'adresses, au cas où il aurait besoin de ces signaux au cours de la phase de données. De plus, un esclave doit ignorer un accès si HTRANS(1:0) = IDLE (le maître ne cherche pas à faire un accès sur le bus) ou si HSEL = 0 (l'esclave n'est pas visé par l'adresse).

L'accès 2 donne un exemple de la manière avec laquelle un cycle de bus peut durer plusieurs cycles d'horloge : on voit qu'au coup d'horloge t3, HREADY = 0, indiquant que l'esclave désire étirer le cycle de bus (généralement pour avoir plus de temps pour compléter l'accès). La phase de données est donc plus longue : un maître effectuant une écriture tient donc HWDATA(31:0) pour un cycle d'horloge de plus, et un esclave répondant à une lecture n'est tenu de fournir des données valides qu'à la fin du cycle de bus (HREADY = 1).

2.4.2. L'arbitrage et la possession du bus

Le protocole AMBA permet à plusieurs maître de venir se connecter au bus. La figure 2.5 donne un exemple du passage de la possession du bus d'un maître à un autre.

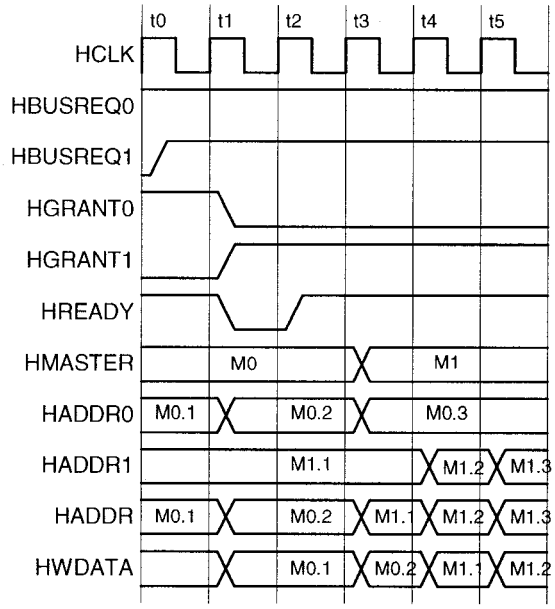

Figure 2.5 : Exemple d'arbitrage

HADDR0 et HADDR1 représentent les valeurs de HADDR émises par le maître 0 (M0) et le maître 1 (M1). Ces valeurs entrent dans le multiplexeur de contrôle. Une des deux valeurs, celle du maître choisi par l'arbitre, sera transférée à HADDR, le signal d'adresse qui sera transmis aux esclaves. Voici les étapes de la progression de la cession du bus de M0 à M1:

- **t0** : Le bus appartient à M0. M1 demande le bus avec HBUSREQ
- **t1** : L'arbitre constate que M1 désire le bus et décide de lui céder au prochain cycle de bus. Il signale cette intention en changeant les valeurs de HGRANT des deux maîtres. Par coïncidence, l'esclave sélectionné par l'adresse (M0.1) indique qu'il veut étirer le cycle de bus en mettant HREADY = 0.
- **t2** : Le bus ne passe pas tout de suite à M1, car le cycle de bus n'est pas terminé. HREADY monte à 1 : le cycle de bus se terminera donc au prochain coup d'horloge.
- **t3** : Le bus passe à M1. HMASTER change à 1, faisant passer les signaux de contrôle de M1 à travers du multiplexeur de contrôle (HADDR prend les valeurs de HADDR1 et non plus celles de HADDR0). On remarque que HWDATA(31:0)

prend encore les valeurs de M0 : étant donné la nature pipelinée du protocole, on doit laisser les bus de données à M0 pour qu'il puisse compléter l'accès commencé au cycle de bus précédent.

- **t4** : Les bus de données passent enfin à M1, qui continuera à utiliser le bus jusqu'à ce que l'arbitre décide de lui enlever le bus. M0 attend en tenant ses signaux de contrôle.

2.4.3. Les accès atomiques

Parfois, il est nécessaire d'effectuer des accès de façon ininterrompue. Un exemple est celui de l'instruction XCHG du ARM7TDMI [ARM01]. Cette instruction échange le contenu de deux registres, mais en utilisant la mémoire. Le contenu du premier registre est écrit en mémoire, puis le contenu du second registre est copié au premier. Finalement, le second registre lit la valeur écrite par le premier registre. Il est important que le processeur garde le contrôle du bus tout au long de cette opération : si par malchance, le maître devait perdre le bus avant que le second registre ne vienne lire l'ancienne valeur du premier registre, il y a la possibilité qu'un autre maître vienne modifier la valeur en mémoire. Le second registre viendrait donc lire une mauvaise valeur, menant à une corruption du système.

HLOCK sert à signaler une suite d'opérations atomiques. Il doit être activé avant que la première phase d'adresses de la suite ne soit commencée. Un exemple est donné à la figure 2.6 :

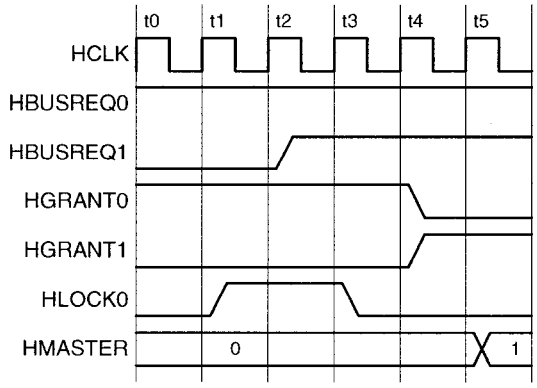

Figure 2.6 : Exemple d'accès atomique

L'exemple présenté ci-haut montre une suite d'accès dont deux sont inséparables.

- **t0** : Un accès normal.
- **t1** : Un accès normal. Le maître signale qu'il a l'intention d'effectuer une suite d'accès atomiques dès le prochain cycle en mettant HLOCK = 1.
- **t2** : Le premier accès de la suite atomique a lieu. Le maître M1 demande le bus. L'arbitre ne lui cédera le bus que lorsque le maître M0 aura signalé la fin de la suite.
- **t3** : Le deuxième accès de la suite a lieu. La suite est terminée, alors M0 met HLOCK à 0.
- **t4** : L'arbitre détecte que la suite est terminée (HLOCK=0). Il annonce aux maîtres que le bus sera cédé à M1 au prochain cycle grâce à HGRANT. M0 en profite pour faire un dernier accès (normal).
- **t5** : Le bus passe à M1. M0 complète la phase de données de son dernier accès.

On remarque donc que selon la spécification, un maître peut toujours faire un accès supplémentaire après une suite atomique.

2.4.4. La gestion d'exceptions

Les exceptions ont lieu lorsque l'esclave sélectionné est incapable de répondre immédiatement à un accès. L'esclave s'exprime en utilisant HRESP(1:0) pour indiquer qu'il y a eu une erreur ou qu'il n'est pas prêt à traiter la requête du maître. La spécification AMBA exige que les valeurs de HRESP(1:0) autres que OK soient tenues pendant un cycle de bus qui dure deux cycles d'horloge. Ceci donne au maître la chance de « changer d'idée » et de vider le pipeline en arrêtant la phase d'adresses en cours (en mettant HTRANS(1:0) = IDLE au cours du second cycle d'horloge). La figure 2.7 illustre la situation :

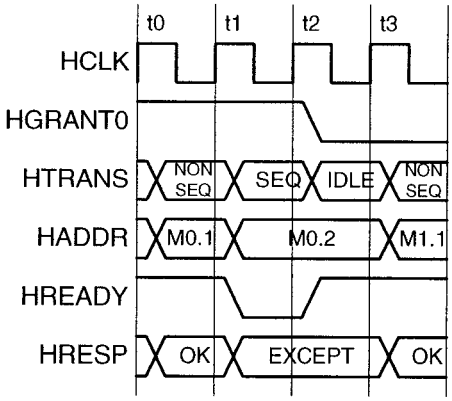

Figure 2.7 : Gestion d'exceptions de HRESP(1:0)

- t0, le maître s'engage dans la phase d'adresses qui va causer l'exception.

- t1 : L'esclave constate que l'adresse cause une exception. Il signale la situation sur HRESP. De plus, il étire le cycle de bus pour que le maître ait la chance de détecter l'exception avant de compléter sa phase de données.

- t2 : Le maître constate l'exception. Il annule la phase d'adresses en cours en transmettant HTRANS(1:0) = IDLE. Ainsi, l'esclave ignorera l'accès en cours et le maître ignorera le résultat de la phase de données. Si la réponse de l'esclave est SPLIT, l'arbitre informe le maître qu'il va perdre le bus au prochain cycle de bus.

- t3 : Selon le type d'exception, le maître prend des mesures pour corriger la situation : dans le cas de ERROR, le comportement dépend du concepteur, mais pour RETRY et SPLIT, le maître doit réessayer l'accès qui a causé l'exception (le comportement du maître est identique pour les deux). De plus, si l'exception était SPLIT, le maître doit attendre que le bus lui soit redonné.

La section 2.3.5. décrit l'utilité de chacun des types d'exceptions.

2.4.5. Maître factice et esclave par défaut

Le protocole AMBA propose l'utilisation d'un maître factice et d'un esclave par défaut. Le maître factice est un module qui n'effectue que des accès où HTRANS(1:0) = IDLE. Il sert dans des situations où une exception SPLIT aurait fait perdre le bus à un maître,

mais où aucun autre maître n'est disponible pour prendre le bus. Puisque le maître n'effectue que des transactions IDLE, il n'a aucun effet sur les esclaves.

L'esclave par défaut est un module qui est sélectionné par défaut si aucun des esclaves n'est choisi par HADDR(31:0). Il émet des réponses ERROR si on tente autre chose que des accès IDLE sur lui. Ainsi on détecte tout de suite un accès dans une région non-assignée de l'espace adresse.

2.5. Considérations sur la bande passante

Un problème classique des protocoles de bus à plusieurs maîtres est le partage de la bande passante. En effet, le temps de service du bus est partagé : si les maîtres sont trop voraces pour des données, comme des microprocesseurs cherchant fréquemment des instructions, la vitesse maximale d'exécution des maîtres sera limitée à la vitesse avec laquelle le bus leur fournira des données.

2.5.1. Augmentation de la fréquence de l'horloge sur le bus

Une première solution envisageable est celle d'augmenter la fréquence d'horloge du bus pour qu'il soit un multiple de la fréquence d'exécution du maître. Les ponts s'occuperaient de séparer les domaines temporels, gardant en tampon les informations de contrôle jusqu'à ce que le maître, plus lent, soit prêt à effectuer sa phase de données.

Avec cette méthode, l'architecture initiale demeure intacte, mais la complexité des ponts doit croître immensément. Il faut aussi garder en tête qu'il existe une limite à la fréquence maximale à laquelle le bus peut fonctionner : il n'est pas facilement envisageable, surtout dans le domaine des systèmes embarqués, de fournir un bus qui fonctionne à 1.6 GHz pour relier huit processeurs de 200MHz.

2.5.2. Division en sous-bus

Une autre solution consiste à partager seulement ce qui est nécessaire. Chaque maître possède son propre bus, avec les esclaves avec lesquels il est le seul à communiquer. Les esclaves partagés sont accessibles sur un autre bus via un pont. Ainsi, les maîtres peuvent

fonctionner en parallèle, risquant peu de se faire déranger [ARM01b]. On peut voir un exemple à la figure 2.8 : les bus privés sont à gauche et le bus partagé à droite.

Cette méthode demande une certaine réorganisation de l'architecture, mais est efficace et demande peu de surface additionnelle. La complexité des ponts reliant les bus privés au bus partagé peut être élevée, particulièrement à cause de la possibilité d'avoir des réponses SPLIT provenant d'esclaves distants. Des fréquences d'horloge différentes pour les divers bus viendraient ajouter un autre degré de difficulté. Le problème de la conception des ponts pour de telles situations est très riche et pourrait servir à lui seul de sujet de recherche très intéressant.

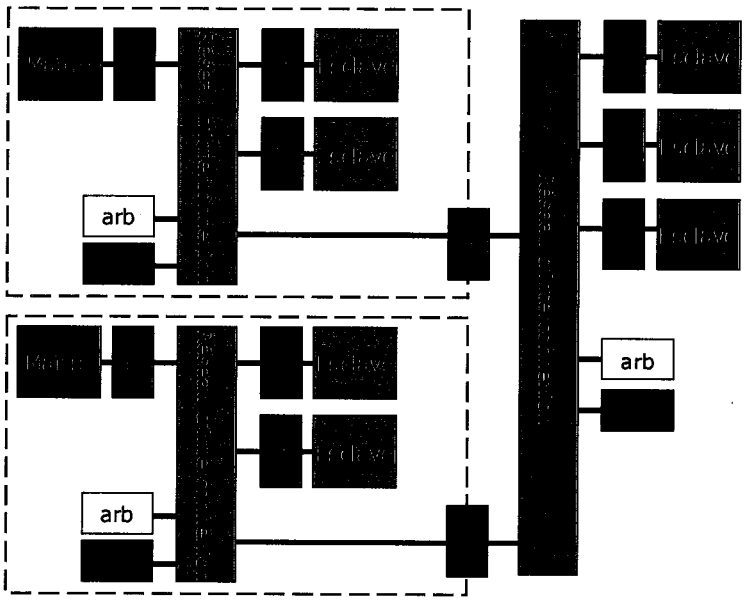

Figure 2.8 : La division en sous-bus

2.5.3. Multiplication des canaux

Une approche plus classique, proposée par ARM dans leur document sur le Multi-AHB, est de multiplier les canaux de communication. Chaque maître a un canal vers chacun des esclaves. Cette méthode est souvent connue comme la solution du "crossbar". Si le nombre de canaux est plus restreint, on parle parfois de "butterfly switch".

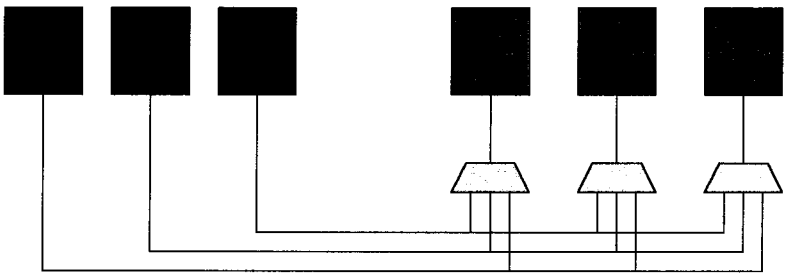

Figure 2.9 : La multiplication des canaux

Comme l'illustre la figure 2.9, au lieu d'avoir un seul multiplexeur pour transmettre les informations de contrôle à tous les esclaves, chaque esclave a son propre multiplexeur, ce qui permet à chacun d'avoir une connexion individuelle avec n'importe quel maître. Cette solution est intéressante, offrant une bande passante presque maximale (sauf dans les situations où plusieurs maîtres viendraient accéder au même esclave au même moment), mais elle demande une surface gigantesque.

2.5.4. Utilisation d'une cache et d'un tampon d'écriture

L'utilisation d'une cache [HePa96] permet à des modules qui sont voraces en données de garder des copies locales de données auxquelles ils accèdent souvent. La cache fait un seul accès à la place du maître lors d'un échec (*cache miss*), mais par la suite, elle garde les données obtenues par l'accès et allège le fardeau sur le bus.

Un tampon d'écriture effectue les écritures à la place du maître. Le tampon saisit les données à la place de l'esclave visé et indique au maître que l'écriture a été faite. Le maître se remet donc au travail pendant que le tampon attend d'avoir le bus pour effectuer ses accès.

On retrouve ces mécanismes souvent dans des noyaux de micro-contrôleurs. Le ARM 940T en est un exemple typique. Le défi, avec ces deux techniques, est d'assurer la cohérence du système dans un contexte multi-maître. Comment s'assurer que les valeurs contenues dans les caches sont cohérentes avec le contenu actuel d'une mémoire? Comment s'assurer qu'une valeur fraîchement écrite en mémoire ne sera pas écrasée par une valeur plus ancienne provenant d'un tampon d'écriture qui a tardé à obtenir le bus?

Plusieurs chercheurs se sont penchés sur cette question qui dépasse la portée de ce travail [CDK00].

Il est intéressant de noter que ces techniques sont, en quelque sorte, des cas particuliers de la division en sous-bus.

2.5.5. AHB lite et Multi-Layer AHB

Des documents publiés par ARM suggèrent deux variantes au protocole AHB: AHB Lite [ARM01c] et Multi-Layer AHB [ARM01b]. AHB Lite propose une simplification des modules impliqués dans un système: on élimine le concept d'arbitrage. Un seul maître est présent dans le système et les réponses SPLIT provenant des esclaves sont prohibées. Puisqu'il y a un seul maître, les multiplexeurs de contrôle et d'écriture ne sont pas nécessaires. Les maîtres conçus avec ce standard n'ont pas de signaux d'arbitrage (HBUSREQ ou HGRANT) et requièrent un pont spécial pour être reliés à un système AHB standard. Parallèlement, un esclave AHB standard a besoin d'un pont spécial pour être inséré à un système AHB Lite.

Le document sur le "Multi-Layer AHB" préconise l'utilisation des techniques de multiplication des canaux et de division en sous-bus pour augmenter la bande passante disponible au maître.

- - -

Après ce chapitre, on peut constater que le protocole AMBA AHB offre de nombreuses possibilités. Tout d'abord, les notions des différents rôles des modules ont permis de structurer les diverses fonctions du système en modules ayant des comportements définis et prévisibles. La spécification et ses documents connexes ont proposé de nombreuses idées quant à la structure hiérarchique d'une éventuelle plate-forme pour la conception. La prochaine section décrit le contenu de la plate-forme en question.

CHAPITRE 3 : Description de l'architecture

Cette section détaille les différents modules qui composent la plate-forme. Elle explique aussi pourquoi chacun d'entre eux a été inclus dans le système.

3.1. Besoins de la plate-forme

Avant de se lancer dans un effort de conception, il est important de bien définir les besoins. La plate-forme conçue au cours de ce projet de maîtrise vient s'inscrire dans un contexte bien particulier qui est venu influencer les choix de conception : le logiciel dans lequel le système doit s'intégrer, le besoin d'une architecture extensible et configurable, l'utilisation éventuelle d'un système d'exploitation temps-réel (un RTOS, ou Real-Time Operating System), l'utilisation de modèles synthétisables et finalement la nécessité de fournir une bande passante maximale au microprocesseur. Cette section détaille chacun des besoins, puis présente les choix architecturaux qui sont venus leur répondre.

3.1.1. Plate-forme cible pour Picasso

La motivation initiale de ce projet était de remplacer l'architecture (plate-forme matérielle) existante dans l'outil logiciel de co-design Picasso[Hene01]. L'outil, à partir d'un ensemble de modules matériels et logiciels codés par l'utilisateur, génère un système complet qui peut être simulé par la suite avec un outil de co-vérification. Notamment, il génère automatiquement des instances du microprocesseur ARM7 et un « contrôleur de mémoire » qui tient le rôle du décodeur et des ponts pour les mémoires. La plate-forme matérielle initiale est peu flexible : la notion de maître et d'esclave est peu définie, et la possibilité d'ajouter des maîtres supplémentaires est inexistante. De plus, elle utilise des modules de mémoire qui ne correspondent pas bien à ce qui est utilisé actuellement dans le domaine des systèmes sur puce. N'importe quel ajout ou retrait de modules de mémoire exigerait une modification du « contrôleur », nuisant ainsi à la modularité du système. Il fallait donc moderniser la partie matérielle tout en gardant en tête la nécessité d'avoir un système qui peut être généré automatiquement par le logiciel Picasso.

3.1.2. Support d'un RTOS et de mécanismes de raffinement

Au cours de ce projet, Picasso a évolué pour faire place à Syslib [Fili02]. Syslib est une librairie C++ pour la description de systèmes au niveau fonctionnel. Pour la réalisation de la partie logicielle à bas niveau, le but est d'éviter de changer le code en réutilisant les mêmes noms de fonctions qu'à haut niveau, mais en faisant correspondre ces noms à du code qui est exécuté sur le processeur. Ce noyau de code fait partie d'un RTOS fourni avec le système. La plate-forme doit donc contenir un minimum de matériel pour supporter un système d'exploitation.

Notez ici qu'avec la popularité actuelle de SystemC, l'utilisation de Syslib est quelque peu remise en question au sein du groupe de recherche. Toutefois, l'utilisation de fonctions RTOS au cours du raffinement d'une spécification SystemC afin de favoriser la réutilisation du logiciel (algorithmes d'ordonnancement, pilotes, etc), est toujours très présente [BCRB03].

Du coté raffinement matériel, puisque la plate-forme se trouve à la toute fin d'une méthodologie de conception (basée sur Syslib ou SystemC), il est impératif de pouvoir produire une puce à partir de ce modèle. Tout le code pour la portion matérielle doit donc être élaboré au niveau RTL, et les modules plus complexes, comme le microprocesseur et les mémoires, doivent être disponibles en tant que modules IP réutilisables.

3.1.3. Utilisation d'un protocole connu

La tendance actuelle dans le domaine du SoC est de réutiliser le plus possible les modules d'un design à l'autre. La plate-forme antérieure n'était basée sur aucun protocole de communication particulier : la communication entre les modules s'effectuait selon un réseau d'interconnexion « maison ». Avec un tel système, l'ajout d'un nouveau module demandait des modifications personnalisées au contrôleur de mémoire, ce qui va contre le principe de la réutilisation de IP : on veut limiter l'effort d'intégration à celui d'assemblage, en évitant au maximum d'avoir à modifier le système auquel on vient se connecter.

L'adoption d'un protocole de bus vient simplifier l'intégration en offrant des interfaces prédéfinies et normalisées pour les différents rôles des modules. Ainsi, l'ajout d'un esclave respectant le protocole choisi peut se faire de façon systématique en ajoutant une interface d'esclave au réseau d'interconnexion. Le chapitre 4 contient une section qui explique comment on peut générer un réseau d'interconnexion AMBA AHB automatiquement, quel que soit le nombre de modules qui s'y rattachent.

3.1.4. Maximisation de l'utilisation du processeur

Comme il a été vu dans la section 2.5, plusieurs maîtres sur un même bus peuvent se ralentir mutuellement si la bande passante (nombre d'accès par coup d'horloge) de celui-ci est inférieure aux besoins des maîtres. Or, un microprocesseur (particulièrement s'il est basé sur une architecture Von Neumann [HePa96]) a continuellement besoin de faire des accès en mémoire pour chercher des instructions et des valeurs intermédiaires de calculs. La plate-forme doit donc fournir un bus presque dédié au microprocesseur afin de s'assurer qu'il ne soit jamais affamé de données.

3.1.5. Choix architecturaux pour répondre aux besoins

L'utilisation d'un système d'exploitation temps-réel implique la présence de certains modules. Tout d'abord, il faut une minuterie, codée au niveau RTL de VHDL, pouvant causer des interruptions périodiques pour signaler les changements de contexte. Puis, il faut choisir un microprocesseur qui est simple à utiliser, relativement performant et pour lequel un modèle synthétisable est disponible. Puisqu'aucun modèle synthétisable n'était disponible au début du projet, le choix est tombé sur le ARM7DMI, processeur très populaire et qui sera bientôt disponible aux universités canadiennes sous forme *hard core* (noyau de matériel pré-défini) grâce à l'initiative SoC de la Société canadienne de la microélectronique [CMC01]. Il faut naturellement de la mémoire pour permettre au processeur de fonctionner correctement. Puisque la minuterie vient prendre une des deux précieuses pattes d'interruption de l'ARM7TDMI, un gestionnaire d'interruptions a été ajouté à l'architecture afin de permettre à un plus grand nombre de modules de

communiquer via des interruptions avec le ARM. Le modèle du gestionnaire est aussi codé en RTL pour assurer la synthèse.

Le choix d'un processeur de la compagnie ARM mène tout naturellement à choisir le protocole de bus proposé par cette même compagnie : le protocole AMBA. Il s'agit d'ailleurs d'un protocole bien documenté et très populaire. De plus, le Groupe de recherche en microélectronique (GRM) ayant déjà acheté une plate-forme basée sur ce protocole, l'apprentissage de AMBA AHB vient fournir un savoir-faire qui pourra servir pour d'autres projets. Un des aspects intéressants du protocole AHB est que les modules sont reliés par un réseau d'interconnexion dont la structure est très régulière : il est possible d'automatiser la génération de ce réseau de façon logicielle, selon le nombre de maîtres et d'esclaves. Le chapitre 4 expliquera comment faire. Une fois que le réseau d'interconnexion est élaboré, il faut ensuite ajouter des ponts à tous les modules afin de les rendre compatibles avec AMBA AHB. La régularisation des interfaces des modules ainsi que la génération automatique du réseau d'interconnexion facilite donc l'utilisation du système en tant que cible bas-niveau dans Picasso.

Finalement, il reste à attaquer le problème de la bande passante. La solution retenue est celle de la division en sous-bus. Tous les esclaves décrits ci-haut sont privés au processeur : aucun autre maître n'a besoin d'y accéder. On peut donc mettre tous les modules sur le même bus qui communique avec un bus externe grâce à un pont. Afin de garder une cohérence dans le système, le bus externe utilise aussi le protocole AMBA AHB. Le pont qui relie les deux ponts est donc un pont AHB-AHB, c'est-à-dire qu'il n'effectue pas de conversion de protocole. On verra que la conception de ce type de pont n'en est pas simplifiée pour autant : il s'agit ici d'un module très complexe qui nécessite plusieurs choix de design. La figure 3.1 montre l'ensemble du bus interne auquel on a donné le nom de bloc-processeur (BP) :

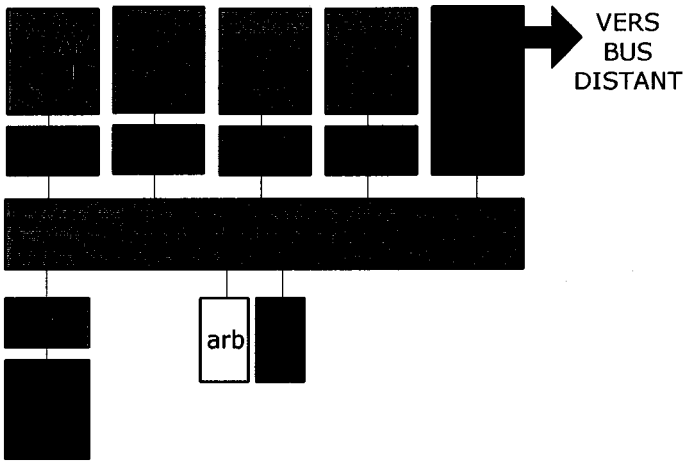

Figure 3.1 : Le bus local du bloc-processeur

3.2. La description des modules

Dans la section précédente, les modules faisant partie de la plate-forme ont été décrits. Cette section reprend chacun des modules et décrit son comportement plus en détail.

3.2.1. Le réseau d'interconnexion

Le réseau d'interconnexion établit tous les liens entre les modules et contient donc les multiplexeurs qui acheminent les informations des maîtres vers les esclaves et vice-versa. Vu de l'extérieur, le réseau ressemble à un bloc hérissé d'interfaces pour accommoder plusieurs maîtres et esclaves, un arbitre et un décodeur. Chaque maître et chaque esclave a un numéro qui est ajouté en suffixe aux ports de son interface. Les modules qui s'y rattachent doivent avoir des interfaces complètes. Un esclave qui ne donne jamais la réponse SPLIT doit quand même avoir les ports HSPLIT, HMASTLOCK et HMASTER, même s'il ne les utilise pas. Dans ce module, on retrouve aussi un maître factice et un esclave par défaut. Ces deux modules sont respectivement le maître 0 et l'esclave 0. On retrouve aussi les multiplexeurs. Le chapitre 4 détaille l'implantation du réseau d'interconnexion.

3.2.2. Le processeur ARM7TDMI

À part le maître factice, le ARM7TDMI est le seul maître du bloc-processeur. Il initie donc toutes les transactions sur le bus. Les spécifications complètes sont disponibles sur le site Web de ARM [ARM01].

Le modèle utilisé dans le projet est un simulateur d'instructions qui vient avec l'outil de co-vérification Seamless CVE. Le modèle VHDL ne contient donc qu'un court fragment de code qui active le simulateur. Le modèle fourni par Seamless est spécialement conçu pour permettre l'activation des optimisations de simulation : le processeur et la mémoire ont la possibilité de communiquer directement via l'outil, permettant ainsi d'accélérer la simulation en évitant de simuler certaines opérations comme les lectures d'instructions [MeGr97]. Étant donné que le modèle communique nécessairement avec Seamless, il est impossible d'effectuer une simulation matérielle seulement avec Modelsim.

Le ARM7TDMI seul n'est pas compatible avec le protocole AMBA AHB, parce qu'il avait été conçu pour fonctionner avec ASB. Il est donc nécessaire de convertir ses signaux avec un pont, et d'utiliser le ARM en mode pipeliné (signal APE = 1). .

La première fonction du pont est de fournir une horloge MCLK au processeur. Puisque l'ARM7TDMI commence ses cycles de bus sur les fronts descendants, le pont doit inverser HCLK, l'horloge du bus. Puisque le ARM n'a pas d'équivalent pour HGRANT, le pont doit arrêter l'horloge du processeur en tenant sa valeur à '1' lorsque le bus ne lui appartient pas. Le ARM7TDMI est un design complètement statique : le contenu des registres reste stable indéfiniment, même si l'horloge ne commute pas. Un comportement identique est utilisé pour étirer des cycles de bus : si HREADY = '0', l'horloge est arrêtée au prochain cycle. La figure 3.2 montre un exemple du comportement du pont et du ARM :

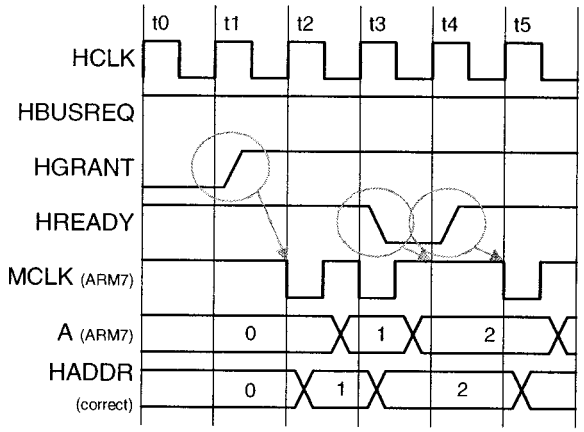

Figure 3.2 : Le comportement du pont du ARM7TDMI

On voit comment l'horloge MCLK est modulée par HGRANT et HREADY. On remarque aussi que l'adresse A fournie par le ARM arrive un peu plus tard que ce qui est demandé par le protocole AMBA AHB : en effet, le ARM7TDMI ne change d'adresse que dans la seconde moitié du cycle d'horloge. Cette déviation aux règles du protocole ne dérange pas à des fréquences d'horloge relativement petites, mais à des fréquences plus élevées, on perd une partie significative du cycle d'horloge, limitant sévèrement la fréquence maximale d'horloge utilisable. Une solution est présentée dans la conclusion de ce mémoire.

Certaines personnes connaissant bien le ARM7TDMI se demanderont peut-être pourquoi le pont n'utilise pas le signal nWAIT pour contrôler le processeur : nWAIT ne peut être changé que lors de la phase basse de MCLK, ce qui inhibe alors le processeur, et du même coup empêche la préparation de la prochaine phase d'adresses.

Le pont traduit aussi les signaux provenant du ARM7TDMI en signaux AHB : MAS sert pour les deux bits les moins significatifs de HSIZE, alors que SEQ et nMREQ servent pour former HTRANS. HPROT prend pour le moment une valeur fixe, mais il est possible de générer des informations plus justes à partir des signaux nOPC et nTRANS. Les bus de données DIN et DOUT sont reliés à HRDATA et HWDATA respectivement.

Finalement, le pont gère aussi les réponses SPLIT en bloquant le processeur et en réessayant l'accès à sa place. Une fois l'accès réussi, le pont relâche le processeur.

3.2.3. Les mémoires SSRAM

Le système utilise deux modules de mémoire statique synchrone (SSRAM). Ce type de mémoire s'adapte bien à des protocoles synchrones comme AMBA AHB. Un des modules contient le code du processeur tandis que l'autre sert d'espace d'entreposage pour la pile, le monceau et les variables. Comme le microprocesseur, ces modèles sont spécifiques à l'outil de co-design, permettant ainsi les optimisations de simulation.

Le pont pour les SSRAM est relativement simple. L'interface des modules de mémoire est légèrement différente de celle d'un esclave AHB : ils ont un bus d'adresse et deux bus de données. Le pont leur transmet directement HADDR, HRDATA et HWDATA. La mémoire possède aussi un signal de sélection SEL qui active le module. Ici, il n'est pas suffisant de transmettre HSEL directement, car l'esclave doit aussi prendre compte de HTRANS. SEL est donc activé, si HSEL est activé et si HTRANS signale un accès (valeur autre que IDLE).

Finalement, on doit fournir les signaux OE_N et BWE_N. OE_N est un signal qui indique qu'on veut effectuer une lecture. Il suffit ici d'inverser HWRITE. BWE_N, est un signal un peu plus complexe : il permet la sélection individuelle des octets dans la mémoire. Cette propriété est utile dans le cas où, par exemple, on désirerait modifier un seul octet en mémoire (changer une lettre dans une chaîne de caractères, par exemple). Le bus étant de 32 bits, une simple écriture viendrait écraser les octets autour de celui qu'on voulait changer. La figure 3.3 illustre ce problème : on veut écrire 0x00 dans l'octet qui est actuellement 0xCC. Sans sélection individuelle (a), tous les octets sont sélectionnés (en gris). Ils sont donc tous remplacés par le contenu du bus. En se servant de HSIZE et

HADDR, il devient possible de sélectionner les octets individuellement[1]. On voit en (b) que seul 0xCC est modifié.

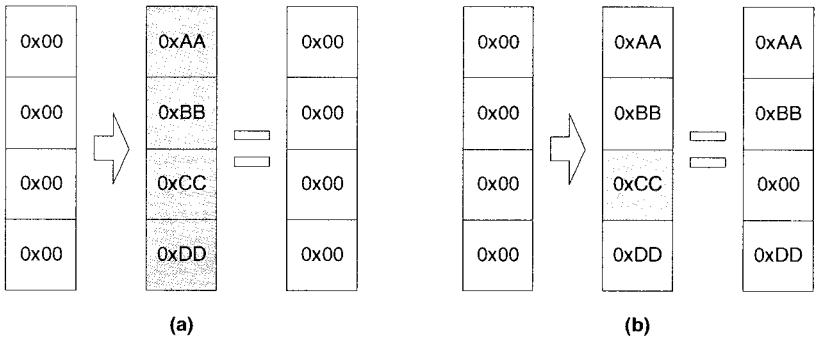

Figure 3.3 : L'utilité de BWE_N

3.2.4. La minuterie programmable

Tout système d'exploitation qui permet des tâches parallèles a besoin d'une minuterie pour lui indiquer le moment où il doit passer d'un processus à l'autre. La minuterie conçue pour ce système a quatre registres de contrôle qui peuvent être lus ou écrits à n'importe quel moment. Pour sélectionner les différents registres lors des accès, on utilise REGSEL (qui sert de bus d'adresses simple).

Tableau 3.1 : Les registres de la minuterie

REGISTRE	REGSEL	FONCTION
RCOUNT	00	Valeur actuelle du décompte
RINIT	01	Valeur chargée lorsque le compte recommence lorsque la minuterie est en mode périodique.
RRUN	10	La minuterie fonctionne seulement si RRUN = 1
RLOOP	11	Indique si la minuterie doit recommencer à compter lorsque RCOUNT = 0

Le programmeur qui désire utiliser la minuterie doit d'abord initialiser le contenu des registres en écrivant à leurs adresses respectives.

---

[1]Référer au chapitre 4 du manuel de référence du ARM7TDMI [ARM01] pour une explication détaillée du processus

Lorsque RRUN = 1, la minuterie décrémente RCOUNT à tous les cycles. Lorsque RCOUNT = 0, une interruption est provoquée : IRQ_N tombe à 0 pendant 5 cycles. On peut arrêter la minuterie en tout temps en fixant RRUN à 0.

RLOOP indique le mode périodique. Si RLOOP = 1 après les cinq cycles d'interruption, RCOUNT charge la valeur de RINIT et se remet à compter vers le bas. On provoque ainsi des interruptions périodiquement, sans avoir à recharger la valeur de RCOUNT manuellement.

Puisque ce module a été conçu pour fonctionner avec un bus pipeliné, le pont n'effectue que des tâches de routage simple : les bits 2 et 3 de HADDR(31 :0) sont transmis à REGSEL, alors que READ et WRITE proviennent de HWRITE, les bus de données sont reliés directement et enfin, IRQ_N est relié au port 0 du gestionnaire d'interruptions.

3.2.5. Le gestionnaire d'interruptions

Le ARM7TDMI n'offre qu'une entrée pour les interruptions matérielles. Afin de pouvoir servir un nombre plus grand de modules, on utilise un gestionnaire d'interruptions. Ce module présente une interface qui comporte huit canaux d'interruption en entrée et une seule en sortie, qui est reliée à la ligne d'entrée du processeur. Si un front descendant survient sur une des lignes d'entrée, le gestionnaire avertit le processeur en activant l'interruption. Le processeur doit alors communiquer avec le gestionnaire via le bus afin de connaître l'identité du module qui a provoqué l'événement. Il active par la suite une fonction de traitement d'interruptions appropriée.

Un module peut provoquer une interruption de deux façons : il peut donner une impulsion sur le signal ou il peut le tenir jusqu'à ce que la situation ait été traitée par le processeur. Il faut donc avoir un moyen de détecter et de se souvenir de fronts descendants sur les canaux d'entrée. Dans le gestionnaire actuel, chaque ligne d'interruption est reliée à une bascule qui prend la valeur '1' chaque fois qu'une interruption a lieu sur le canal correspondant. Les sorties de toutes les bascules sont ensuite groupées dans une porte NOR qui produit l'interruption de sortie : dès que le contenu d'une des bascules devient

1, la sortie tombe à 0 (l'interruption est active basse). L'ensemble de ces bascules constitue le registre de statut. La figure 3.4 montre un exemple de la structure interne du gestionnaire :

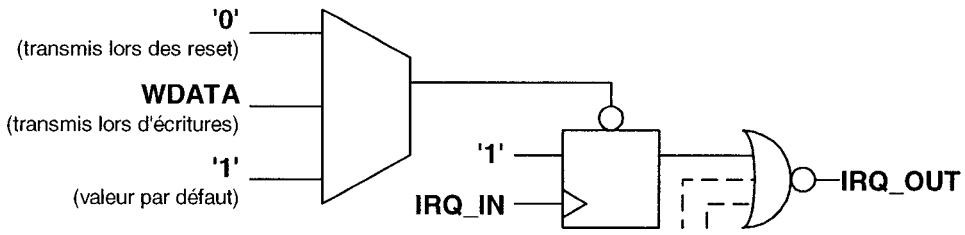

Figure 3.4 : Structure d'un canal d'interruption

Lors du traitement de l'interruption, le processeur doit aller lire le contenu du registre de statut afin de savoir l'identité du module qui l'a sollicité. Il peut ensuite prendre les mesures nécessaires pour gérer l'interruption. Pour indiquer au registre que l'interruption a été traitée, le processeur envoie un vecteur qui ne contient que des '1', sauf pour les bits correspondant aux modules traités (qui sont à 0). Les bits écrits servent de signal de remise à zéro des bascules : les bits pour lesquels on a écrit 0 retombent donc à 0, et les autres demeurent inchangés. Lorsque aucune écriture n'a lieu, tous les signaux RESET sont à 1.

Il est à noter que ce module n'est pas capable de stocker les événements: si une interruption a lieu sur un canal qui n'a pas encore été traité, l'interruption est perdue. Une version subsequente de ce module pourrait se servir de fronts descendants sur ses entrées pour incrémenter des compteurs et ainsi suivre le nombre d'interruptions qui ont eu lieu sur un canal donné.

Le pont pour ce module est très simple : il relie la sortie du registre de statut sur HRDATA et ne garde que les huit bits moins significatifs de HWDATA. Il génère le signal d'écriture à partir de HWRITE et HSEL.

Les canaux 1 à 7 sont reliés avec des ports qui communiquent avec le bus extérieur. Le canal 0 est réservé pour la minuterie.

3.2.6. Le pont AHB-AHB

Le pont AHB-AHB fait le lien entre le bus local et le bus distant. C'est à travers lui que le ARM7TDMI va communiquer avec les esclaves partagés du système. Il y a une infinité de moyens pour concevoir un tel pont. Concrètement, le pont est une machine à états qui contrôle le progrès du maître qui effectue l'accès selon l'état du bus distant. La machine à états est décrite en détail dans le chapitre 4. Ici on ne présente que le comportement général du module.

Le rôle de ce module est de faire abstraction du bus distant : du point de vue du bus local, le bus distant est un seul esclave à la place du pont. De façon semblable, le bus local semble être un maître relié au bus distant. C'est d'ailleurs avec cette perception que la plate-forme est utilisée dans les designs : le bloc-processeur, contenant les modules décrits dans cette section, sert de maître sur le bus distant. L'interface « maître » provient du pont AHB-AHB qui est relié au bus local par son interface esclave. Afin de distinguer les ports de l'interface maître de ceux de l'interface esclave, les noms des signaux ont le sufixe « _m » (pour *master*, maître) ou « _s » (pour *slave*, esclave).

Normalement, le pont est dans l'état IDLE : aucun maître local ne le sollicite et il ne demande pas le bus distant. Si jamais l'arbitre distant lui accorde le bus malgré lui, il effectue des accès IDLE pour passer le temps sans affecter les esclaves distants.

Si un maître le sollicite, il passe en mode RUN : il demande le bus distant avec HBUSREQ_m et fait attendre le maître avec HREADYOUT_s jusqu'à ce que HGRANT_m lui signale que le bus distant lui appartient. Une fois le bus distant accordé, le pont agit de façon presque transparente : il laisse passer les signaux de contrôle et les bus de données presque directement. Si jamais le bus distant lui est enlevé, il change d'état et bloque le maître immédiatement avec HREADYOUT_s (qui pilote HREADY sur le bus local lorsque le pont est sélectionné). Ceci pose un léger problème si le maître est en train d'effectuer une lecture. Comme stipulé dans la section 2.4.2, le maître complète sa dernière phase de données pendant la première phase d'adresse du maître qui vient de gagner le bus. Or, si le maître est bloqué par HREADY, il manquera la réponse

de l'esclave distant qui ne sait pas que le maître vient d'être bloqué par le pont qui s'interpose entre eux. Soit RUN_LAST l'état juste avant le blocage du maître et soit RUN_FIRST l'état de ce même maître immédiatement après sa remise en marche (c'est-à-dire lorsque HREADY remonte à 1 au moment où le bus distant lui est redonné). Le pont a donc la responsabilité, à la fin de l'état RUN_LAST, de saisir les données du bus HRDATA et de les fournir au maître dans l'état RUN_FIRST. . Entre RUN_FIRST et RUN_LAST, le pont est dans l'état HOLD, qui tient HREADYOUT_s = 0.

À des fins expérimentales, le pont a été conçu de façon à gérer des réponses SPLIT provenant d'esclaves distants. Lors d'un accès dont la réponse est SPLIT, l'arbitre distant enlève le bus au pont. Le pont passe à l'état EX_FLUSH, qui vide le pipeline distant en émettant HTRANS_m = IDLE et transmet la réponse à l'arbitre local qui enlève le bus au ARM7TDMI (pour le donner au maître factice). En attendant le bus, le pont reste dans l'état EX_HOLD. Lorsque le bus distant est redonné au pont, ce dernier passe à l'état EX_WAKE et avertit l'arbitre local avec HSPLIT_s. Si jamais un autre maître sur le bus local venait solliciter le pont pendant l'attente, ce dernier émettrait une réponse RETRY.

3.2.7. L'arbitre

Puisque le bloc processeur n'a qu'un seul véritable maître, l'arbitre a bien peu à faire. Par défaut, il cède le bus au ARM7TDMI en tout temps. C'est seulement lorsque le bus distant provoque une réponse SPLIT que le bus passe au maître factice. La figure 3.5 montre le fonctionnement simple de la machine à états de l'arbitre :

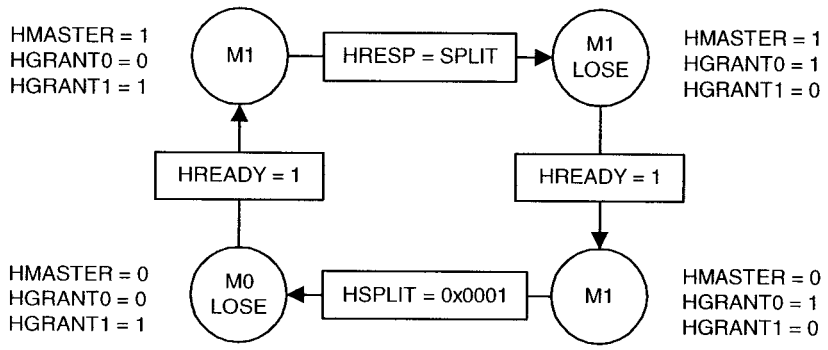

Figure 3.5 : Le fonctionnement de l'arbitre

L'arbitre commence dans l'état M1. Le bus appartient au ARM7TDMI (le maître 1). Dès que l'arbitre détecte une réponse SPLIT, il passe dans l'état M1_LOSE, où il informe le ARM7TDMI qu'il est sur le point de perdre le bus. Au prochain cycle de bus, le pont passe à l'état M1 et le maître factice prend contrôle jusqu'à ce que le pont informe l'arbitre qu'il est prêt à continuer l'accès avec HSPLIT. L'arbitre passe par la suite à l'état M1_LOSE et avertit le ARM7TDMI qu'il va regagner le bus avant de lui donner au prochain cycle de bus (M1).

3.2.8. Le décodeur

Le décodeur est responsable de sélectionner les esclaves dans le bloc processeur. Il se fie sur HADDR pour décider de l'esclave qu'il doit activer. La table 3.2 montre l'association des adresses aux divers esclaves du système. L'esclave par défaut local n'est jamais sélectionné.

Tableau 3.2 : Associations adresse-esclave

ESCLAVE	PLAGE D'ADRESSES
SSRAM (Code)	0x00000000 – 0x00FFFFFF
SSRAM (Data)	0x01000000 – 0x01FFFFFF
Minuterie	0x02000000 – 0x02FFFFFF
Gestionnaire d'interruptions	0x03000000 – 0x03FFFFFF
Pont & bus distant	0x04000000 – 0xFFFFFFFF

On remarque que le décodeur n'observe que l'octet le plus significatif.

CHAPITRE 4 : Méthodologies de conception pour AHB

L'apprentissage de la conception dans le cadre d'un protocole se fait graduellement : la spécification ne donne pas toujours d'indications claires sur la réalisation matérielle des règles imposées, ni d'explications sur les raisons pour lesquelles ces règles existent. En se mettant la main à la pâte, certains patrons de conception émergent. Ce chapitre témoigne donc de l'expérience acquise lors de la conception de la plate-forme et tente d'en dégager une méthodologie de conception.

Trois aspects se sont particulièrement distingués : la structure régulière du réseau d'interconnexion, l'allure générale des maîtres de bus et de leurs ponts vers le bus AHB et finalement le comportement de l'arbitre.

Dans les modèles qui suivent, les notions de délais de propagation et de temps de maintien et de préparation ne sont pas considérées: le but du projet était de déterminer ce qu'il fallait faire pour implanter les règles du protocole et ce, indépendamment de la technologie choisie. Au moment de la synthèse, il faut donc rester attentif aux temps d'arrivée des signaux afin d'ajouter des lignes de délai si nécessaire.

4.1. Génération automatique d'un réseau d'interconnexion

Souvent, lors de l'étape de design initiale, le concepteur dessine un système de bus de la façon suivante :

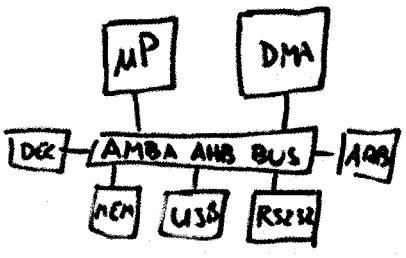

Figure 4.1 : Croquis de concept

Dans la figure 4.1, le bloc « AMBA AHB BUS » cache les détails des interconnexions entre les divers modules, les multiplexeurs et aussi le détail des interfaces des modules.

En effet, ces éléments sont généralement semblables d'un design à l'autre, et le concepteur préfère en faire abstraction afin de se concentrer sur l'architecture et l'effet qu'elle a sur l'interaction entre les modules. Il est intéressant d'offrir au concepteur la possibilité de travailler de façon semblable au moment de la réalisation matérielle. La solution développée dans le cadre de ce projet est de contenir tous les détails du réseau d'interconnexion dans un module auquel tous les autres viendront s'attacher.

La structure du réseau d'interconnexion dépend du nombre de maîtres et d'esclaves dans le système. L'ajout ou le retrait d'un module vient perturber le code VHDL du réseau en plusieurs endroits. Un programme en Java a donc été conçu pour générer automatiquement le fichier VHDL, afin d'éviter de faire les modifications manuellement. Le programme réduit le temps de conception et le risque d'insérer des erreurs en effectuant des modifications. Cette section détaille les différents éléments contenus dans ce module, et comment ils sont générés automatiquement.

Le réseau d'interconnexion contient les multiplexeurs, les interfaces (ports) pour les divers maîtres et esclaves, ainsi que le maître et l'esclave par défaut.

4.1.1. Génération des ports

La première partie du code VHDL est la description de l'entité, où les ports sont déclarés. Au moment de la génération, il faut spécifier le nombre de maîtres et d'esclaves. Pour chaque maître et esclave, on déclare une interface qui contient tous les signaux spécifiés dans la norme AMBA AHB. Puisque le réseau d'interconnexion ne peut pas avoir plusieurs ports avec le même nom, le nom de chaque port porte un suffixe qui identifie le module auquel il doit être connecté : « _m0 » identifie le premier maître, « _s2 » identifie le troisième esclave, etc.

Il est aussi nécessaire de produire les signaux pour l'arbitre et le décodeur. Les signaux pour l'arbitre portent le suffixe « _a ». L'arbitre possède un triplet HBUSREQ – HGRANT – HLOCK pour chaque maître. Le décodeur, identifié par le suffixe « _d » possède une sortie HSEL pour chaque esclave dans le système.

Finalement, les ports HCLK et HRESET_n servent à faire entrer l'horloge et le signal de réinitialisation globaux pour le système au complet.

La table 4.1 résume les noms des ports pour chaque type de module :

Tableau 4.1 : Ports générés par rôle

RÔLE	SIGNAUX
Maître # *i*	HADDR_M*i*(31:0), HWRITE_M*i*, HSIZE_M*i*(2:0), HTRANS_M*i*(1:0), HBURST_M*i*(2:0), HPROT_M*i*(3:0), HBUSREQ_M*i*, HLOCK_M*i*, HGRANT_M*i*, HREADY_M*i*, HRESP_M*i*(1:0), HRDATA_M*i*(31:0), HWDATA_M*i*(31:0), HRESET_M*i*, HCLK_M*i*
Esclave # *i*	HSEL_S*i*, HADDR_Si(31:0), HWRITE_S*i*, HSIZE_S*i*(2:0), HTRANS_S*i*(1:0), HBURST_S*i*(2:0), HREADY_S*i*, HREADYOUT_S*i*, HRESP_S*i*(1:0), HMASTER_S*i*(3:0), HMASTLOCK_S*i*, HSPLIT_S*i*, HRDATA_S*i*(31:0), HWDATA_S*i*(31:0), HRESET_S*i*, HCLK_S*i*
Arbitre	HADDR_A(31:0), HTRANS_A(1:0), HBURST_A(2:0), HRESP_A(1:0), HREADY_A, HSPLIT_A(15:0), HMASTER_A(3:0), HMASTLOCK_A, Pour chaque maître i = (0 à nombre_de_maîtres) HBUSREQ_Ax, HLOCK_Ax, HGRANT_Ax
Décodeur	HADDR_D, Pour chaque esclave i = (0 à monbre_d'esclaves) HSEL_Di (où n = 0 .. nombre d'esclaves)

4.1.2. Maître factice et esclave par défaut

Le concepteur fait généralement abstraction du maître factice et de l'esclave par défaut, car ces modules sont très simples et ne changent jamais. Il est donc pratique de les inclure directement dans le module contenant le réseau d'interconnexion : ils n'ont pas besoin de ports pour se relier au bus. Les ports « _m0 » et « _s0 » ne sont donc pas générés : on crée à leur place un ensemble de signaux internes portant les mêmes noms et qui seront pilotés par des *process*.

Le maître factice ne fait qu'émettre des transactions sur le bus où HTRANS = IDLE. Puisque les accès n'ont aucun effet sur des esclaves correctement conçus, les valeurs

émises sur les autres signaux ont peu d'importance. Néanmoins, pour réduire le risque de modifier un esclave mal conçu, on a :

- HWRITE = 0 (puisque les lectures sont rarement destructives),
- HBURST(3:0) = 0011 (ce qui signifie une transaction ne pouvant être mise en cache ou dans le tampon d'écriture) et
- HSIZE(2:0) = 010 (des accès de 32 bits, soit la largeur du bus).

L'esclave par défaut est un peu plus complexe : il doit émettre une réponse HRESP = ERROR pendant deux cycles d'horloge (comme stipulé dans la section 2.4.4), si jamais un accès non-IDLE est tenté sur lui. Il s'agit donc d'une sorte de machine à trois états : les deux cycles de réponse et l'état normal. On trouve son code dans le processus *defaultslave*.

4.1.3. Multiplexeurs

Comme il a été vu à la section 2.2.2, le réseau d'interconnexion réussit à faire communiquer les modules grâce à l'utilisation de multiplexeurs.

Le multiplexeur de contrôle reçoit les signaux de contrôle provenant de tous les maîtres (HADDR_M*i*(31:0), HWRITE_M*i*, HSIZE_M*i*(2:0), HTRANS_M*i*(1:0) et HBURST_M*i*(2:0)) et choisit un seul ensemble de ces signaux, selon la valeur de HMASTER, pour les router vers des signaux centraux, portant le même nom, mais sans suffixe. Ces derniers sont ensuite routés aux autres modules qui les utilisent (les esclaves, l'arbitre et le décodeur).

Puisque le protocole est pipeliné, on doit utiliser une version retardée d'un cycle de bus de HMASTER pour contrôler le multiplexeur de HWDATA, car la phase de données a lieu dans le deuxième étage du pipeline. Son fonctionnement est semblable à celui du multiplexeur de contrôle : il reçoit les HWDATA_M*i* provenant de tous les maîtres et laisse passer seulement celui du maître qui avait la possession du bus au cycle de bus précédent. Le processus *delay_hmaster* s'occupe de générer le retard.

Finalement, le multiplexeur de réponse s'occupe de router les réponses provenant des esclaves. Il est contrôlé par une copie retardée d'un cycle de bus de l'ensemble des signaux HSEL_S*i*. On doit utiliser une version retardée parce que le signal HSEL de l'esclave est activé au cours de la phase d'adresses mais il est utilisé pour router les réponses seulement au cours de la phase de données (au cours de la phase de données, une autre phase d'adresses est commencée, changeant possiblement HSEL). Ce multiplexeur route les signaux HRDATA_S*i*(31:0), HRESP_S*i*(1:0) et HREADYOUT_S*i*. HREADYOUT est routé vers HREADY, qui est envoyé à tous les maîtres et tous les esclaves.

Un grand nombre de processus concurrents au début de l'architecture VHDL transmettent les sorties de multiplexeurs vers les ports de sortie, et les ports d'entrée vers les entrées des multiplexeurs.

4.1.4. Signaux HSPLIT

La FAQ de AMBA suggère de relier tous les HSPLIT_S*i*(15:0) ensemble via une porte de type OU. Si plusieurs esclaves tentent de réveiller des maîtres différents, l'arbitre doit décider de celui qui aura la possession du bus.

4.2. Structure d'un pont AHB pour un maître

Un protocole s'exprime souvent comme un ensemble de règles à respecter mais propose rarement des structures matérielles réelles à utiliser. Cette section propose une méthodologie générique pour la conception de ponts AHB pour des maîtres, obtenue en observant progressivement l'ensemble des règles et requis qu'ils doivent respecter. Cette méthode est basée sur l'utilisation de noyaux dont les accès mémoire se font déjà de façon pipelinée.

4.2.1. Responsabilités d'un maître AHB

La première étape lors de la conception d'un maître est de bien parcourir la spécification AMBA à la recherche des diverses règles à respecter. On trouve ainsi l'ensemble des comportements qu'on doit implanter. Les responsabilités d'un maître sont :

Tableau 4.2 : Les responsabilités d'un maître

RÉGLE	EXPLICATION
Pipelinage	Les accès doivent être effectués sur deux phases : adresse et données.
Extensibilité des cycles de bus	Le maître doit garder ses signaux stables si le cycle de bus est étiré (sauf dans les cas d'exceptions)
Arbitrage	Le maître doit attendre que le bus lui soit cédé (HGRANT = 1 et HREADY = 1) avant d'effectuer un accès. Si il perd le bus (HGRANT = 0 et HREADY = 1), il doit suspendre la prochaine phase d'adresses, mais doit finir la phase de données en cours.
Accès atomiques	Un maître voulant effectuer un accès atomique doit activer HLOCK = 1 au cours du cycle précédant la rafale d'accès indivisibles. Au cours du dernier accès de la rafale, HLOCK tombe à 0.
Gestion d'exceptions	Si au début d'un cycle, HRESP est différent de OK, le maître doit émettre HTRANS = IDLE. Si la réponse était RETRY ou SPLIT, il doit réessayer l'accès qui a causé l'exception au prochain cycle de bus. Pour les réponses ERROR, le comportement est indéfini

À partir de ces cinq groupes de responsabilités, il sera possible de développer une machine à états qui fera la conversion.

4.2.2. Établir des correspondances

La prochaine étape consiste à définir un état « normal » de fonctionnement. Pour ce faire, il faut observer soigneusement l'interface du maître pour voir s'il existe une correspondance entre ses ports et ceux qui sont spécifiés par le protocole. Ces correspondances vont venir simplifier l'interface en réduisant les fonctions qu'il aura à accomplir.

On soulève d'abord les correspondances *directes*, soit les comportements du maître qui sont identiques aux comportements demandés dans le protocole. Pour adapter les ports impliqués par ces correspondances, il suffit souvent de changer leur nom, leur ajouter ou retirer des bits, les concaténer ou les inverser.

Les correspondances *indirectes* impliquent un certain traitement : il faut changer la phase d'une sortie du maître, il faut insérer des états d'attente, il faut traduire certains signaux pour éviter des valeurs invalides, etc. Il faudra compenser pour ces incompatibilités par l'ajout d'états dans la machine d'états.

Pour les comportements qui n'ont aucune correspondance chez le maître, il faudra que le pont vienne faire la tâche pour le maître. Dans ces situations, il est souvent nécessaire d'avoir un noyau qui a un signal d'entrée qui peut l'arrêter ou une horloge qui peut être arrêtée indéfiniment. En arrêtant le maître, le pont peut faire l'opération à sa place

Cette étape est probablement la plus difficile, car elle demande une connaissance en profondeur du noyau ainsi que du protocole. Il faut souvent utiliser des astuces pour tirer toutes les informations possibles du noyau afin d'anticiper des signaux ou de les retarder. Il est possible que parfois, un maître ne soit tout simplement pas compatible avec un protocole, mais heureusement, ils sont souvent conçus avec un protocole particulier en tête. Étant donné que la plupart des protocoles populaires se ressemblent, les chances d'obtenir une incompatibilité totale sont rares.

4.2.3. Raffinement progressif de la machine à états

La prochaine étape consiste à ajouter progressivement des états selon les besoins établis à l'étape précédente. La figure 4.2 montre l'état de base du système :

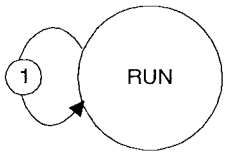

Figure 4.2 : L'état de base

Dans cet état, on effectue les accès pipelinés de base : les bus d'adresses et de données sont transmis et reçus et les signaux de contrôle sont générés. On suppose ici que seule la responsabilité d'effectuer des accès pipelinés est sollicitée. On remarque la transition (1) : elle représente une action qui peut être effectuée sur le front montant de l'horloge. Dans

le protocole AHB, tout est synchronisé sur les fronts montants, et les signaux changent généralement sous son effet.

La prochaine règle à respecter est celle de l'extensibilité des cycles de bus. Dans ce qui suit, nous nommerons *noyau* (de l'anglais core) un maître non compatible AMBA AHB. Si ce noyau ne possède pas de signal analogue à HREADY, il faut alors arrêter son horloge pendant les cycles d'attente. On ajoute donc un état d'attente, comme illustré à la figure 4.3 :

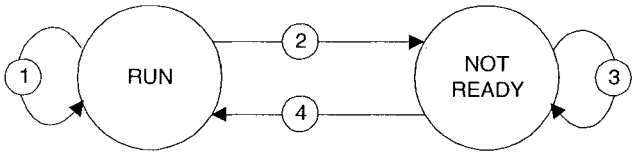

Figure 4.3 : L'ajout d'un état d'attente

Dans l'état NOT_READY, l'horloge du noyau est arrêtée : aucun coup d'horloge n'a lieu et donc le maître ne tente pas de compléter sa phase de données. Selon le point de vue du noyau, il effectue un accès par cycle, mais les cycles n'ont pas des durées égales. La transition (2) est prise lorsque HREADY = 0 à la fin d'un cycle. Le pont remarque que le cycle de bus va être étiré et il met le noyau en suspension. La transition (4) est prise lorsque HREADY = 1. À part l'inhibition de l'horloge, l'état NOT_READY transmet les signaux de façon identique à l'état RUN.

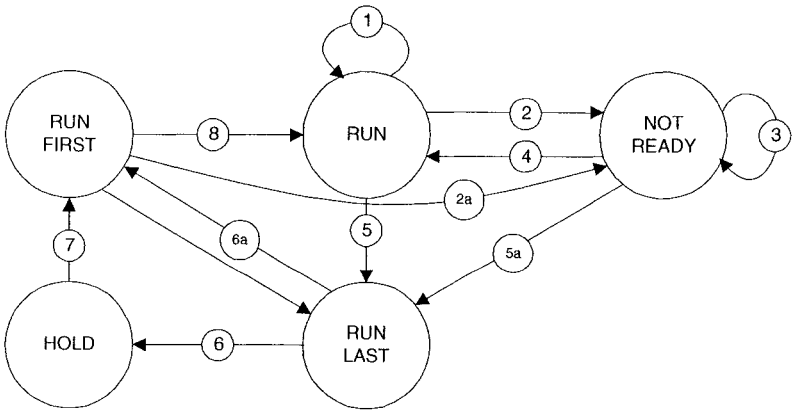

Figure 4.4 : La gestion de l'arbitrage

La prochaine règle à observer est celle de l'arbitrage. Il faut arrêter le maître s'il n'a pas accès au bus. On suppose ici qu'on arrête encore une fois le noyau en inhibant son horloge. La figure 4.4 montre les trois nouveaux états qui ont été ajoutés à la machine précédente : RUN_LAST, RUN_FIRST et HOLD. L'état RUN_LAST s'occupe du dernier cycle de données que le maître doit faire avant de perdre définitivement le bus. L'état HOLD garde le noyau endormi pendant que le bus appartient à un autre. La première phase d'adresses est traitée par RUN_FIRST avant de repasser à RUN. La figure 4.5 montre le chronogramme de la perte et de la saisie subséquente du bus. Les transitions de la figure précédente sont indiquées par leur numéro dans l'image :

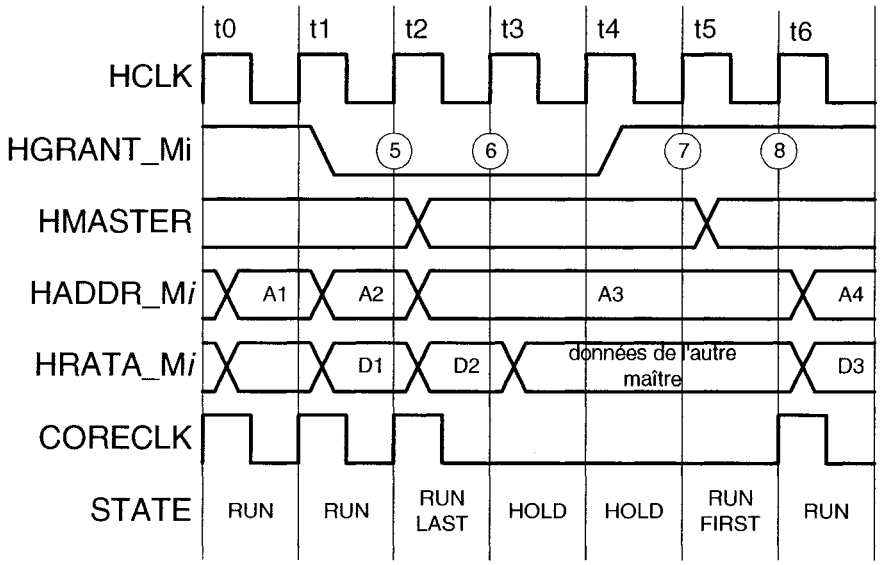

Figure 4.5 : Le chronogramme de l'arbitrage

La transition (5) montre la perte du bus par le maître : il a détecté HGRANT = 0 au début du cycle de bus. Il doit donc accomplir sa dernière phase de données (D2) au cours de RUN_LAST puis se mettre au repos au prochain cycle. L'horloge du noyau (CORECLOCK) n'est pas inhibée au cours de ce cycle : on permet au noyau de saisir la donnée du cycle de données précédent (D1) et de préparer le cycle d'adresses pour l'accès qu'il fera lorsque le bus lui sera remis (A3). Au cycle de bus suivant, la machine à états passe à l'état HOLD (transition 6), qui bloque l'horloge. En effet, il est désirable de l'arrêter pour empêcher le maître de passer à la prochaine phase d'adresses (A4) : la

phase (A3) n'a pas été effectuée puisque le bus ne lui appartenait plus. Cependant, si A3 était une lecture, le blocage de l'horloge empêche le noyau de saisir la donnée (D2) sur HRDATA. *Il faut donc que le pont le fasse à la place du noyau.* On établit donc la règle suivante : toute transition sortant de RUN_LAST (donc (6) et (6a)) doit saisir la valeur de HRDATA et la transmettre au noyau jusqu'à ce que l'horloge soit redémarrée.

Lorsque le bus est enfin redonné au maître (7), on doit garder l'horloge désactivée pour un cycle de bus de plus : ceci permet au pont de transmettre la valeur de HRDATA conservée lors de la sortie de RUN_LAST. Si l'horloge était active au cours de ce cycle, l'adresse du noyau aurait changé, faisant manquer ici l'accès A3.

La prochaine règle à respecter est celle des accès atomiques. Selon la spécification AMBA AHB, le maître doit déclarer son intention d'effectuer une suite d'accès atomiques au cycle de bus précédant la première phase d'adresses de la rafale. Le noyau doit donc avoir un signal qui lui permet d'indiquer qu'il est sur le point d'effectuer un accès atomique (il sera appelé CORELOCK dans le texte qui suit). S'il n'en a pas, il est fort probable que le noyau n'ai jamais besoin de faire des accès atomiques. Si le CORELOCK respecte les exigences de AMBA (il est activé au cours du cycle de bus qui précède la première phase d'adresses de la séquence atomique), la conversion est simple : on n'a qu'à relier CORELOCK à HLOCK et le tour est joué. S'il est activé plusieurs cycles avant la première phase d'adresses, on peut le retarder.

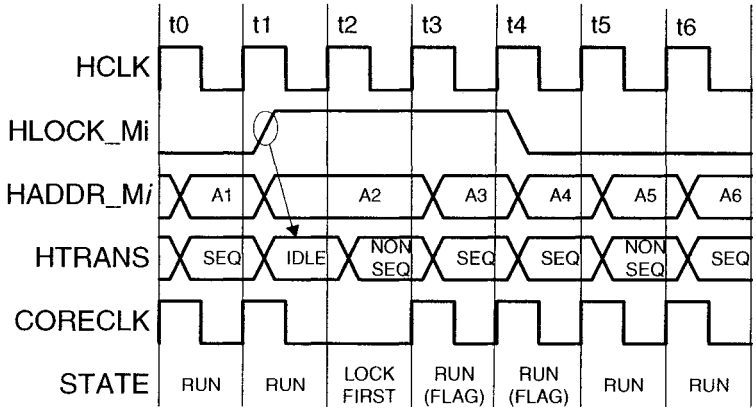

Figure 4.6 : Solution pour le problème des accès atomiques

La figure 4.6 propose une approche pour le cas problématique où le CORELOCK serait activé en même temps que la première phase d'adresses de la suite d'accès atomiques. Une partie de la solution consiste à inhiber l'horloge pour un cycle de bus au cours du premier accès, donnant ainsi le temps à l'arbitre de constater l'activation de HLOCK. On doit donc ajouter un nouvel état, LOCK_FIRST pour provoquer ce blocage. Ceci ne suffit pas, cependant: l'esclave accédé par ce maître, n'étant pas conscient de l'arrêt du maître, verra deux accès identiques à la même adresse, ce qui peut être problématique. Il faut donc inhiber un des accès sur le bus. La façon la plus facile d'accomplir ceci est d'émettre HTRANS = IDLE pour un des "deux" accès. Puisque l'arbitre n'a pas encore établi l'atomicité de l'accès, il est préférable de faire ceci au cours du premier des deux cycles. Il faut donc ajouter une condition dans les états RUN et RUN_FIRST : si CORELOCK = 1, alors forcer HTRANS = IDLE. Puisqu'on revient à RUN après LOCK_FIRST (la transition 12 dans la figure 4.7, ci-dessous), il faut établir un mécanisme pour que cet accès bidon ne soit inséré qu'une seule fois: lors de la transition de RUN vers LOCK_FIRST (11), on active un indicateur (LOCK_FLAG) qui permet au pont de savoir qu'il effectue une séquence d'accès atomiques. L'indicateur est désactivé si HLOCK = 0 sur la transition 1.

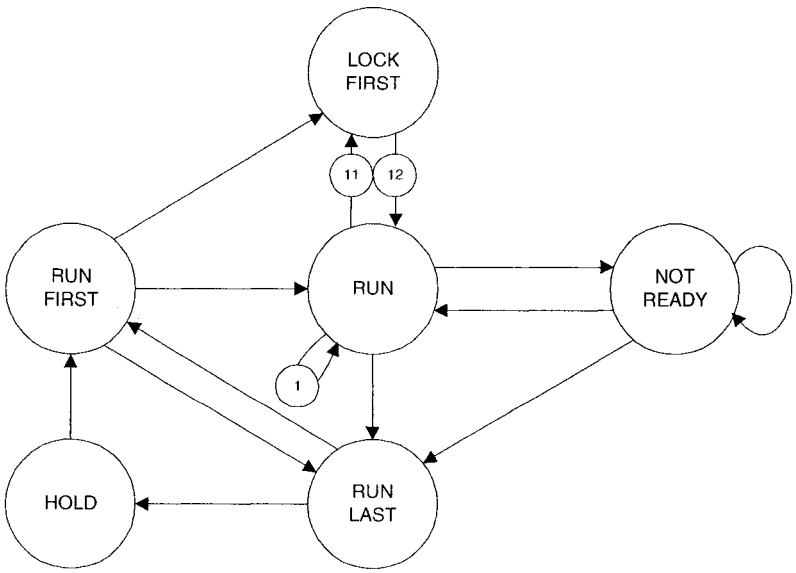

Figure 4.7 : Intégration à la fig. 4.4 de la règle des accès atomiques

Finalement, il reste la règle de la gestion des exceptions. Cette notion est assez particulière au protocole utilisé: la notion de réponse SPLIT, par exemple, n'est pas souvent présente dans d'autres protocoles. C'est donc souvent la responsabilité du pont que de s'occuper des exceptions. Il faut donc encore une fois arrêter le noyau pour faire le travail à sa place. Le noyau ainsi stoppé ne voit pas les détails de la gestion de l'exception, ne voyant passer qu'un seul cycle d'horloge. La figure 4.8 montre un fragment de la machine à états finale, qui incorpore la gestion de ces situations:

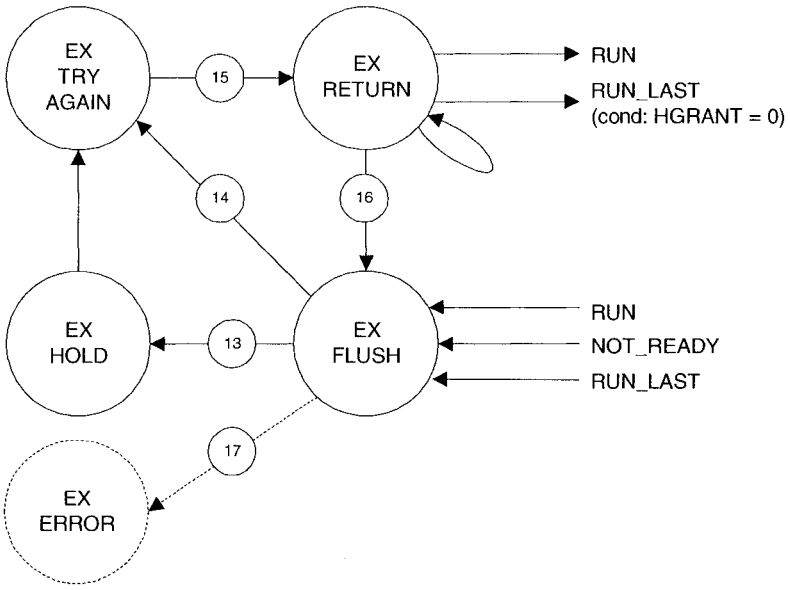

Figure 4.8: Fragment qui s'occupe des exceptions

Un comportement commun à toutes les réponses anormales des esclaves est que ces dernières durent deux cycles d'horloge, tel qu'expliqué à la section 2.4.4. Le maître doit émettre HTRANS = IDLE au cours du deuxième de ces cycles. Ceci correspond à l'état EX_FLUSH dans le diagramme ci-haut. L'horloge est arrêtée et le noyau se met à attendre pour compléter la phase de données qui a causé l'exception. On note ici que le noyau n'a aucune conscience de la réponse de l'esclave: le pont a détecté la situation et bloque le noyau jusqu'à ce que la situation ait été rectifiée.

Aucun comportement spécifique n'est proposé pour les réponses ERROR (17): le concepteur est libre de développer seul cette partie de la machine à états.

Pour les réponses RETRY ou SPLIT, le maître doit essayer l'accès de nouveau dès le cycle suivant. On passe soit dans l'état EX_HOLD (13) (si l'arbitre a retiré le bus au maître, c'est-à-dire que HRESP = 0) ou EX_TRY_AGAIN (14). Le premier état continue à bloquer le noyau pendant l'attente pour le contrôle du bus. Dans le second état, le pont inhibe encore l'horloge du noyau et tente de nouveau l'accès en pilotant lui-même les signaux de contrôle. On note qu'il est donc nécessaire de garder en mémoire les informations de la phase d'adresses qui avait causé l'exception: à chaque transition vers RUN (incluant de RUN à RUN) ces valeurs doivent être stockées. Ensuite, on passe immédiatement à la phase de données qui est l'état EX_RETURN (15) (les données d'écriture sont déjà présentes sur le bus de sortie du noyau: il faut simplement les transmettre au cours de cette phase). Une réponse autre que OK recommencera le processus: la machine revient dans l'état EX_FLUSH (16). Sinon, l'accès est complété en revenant à RUN à la fin du cycle de bus, ce qui libère enfin le noyau pour qu'il puisse saisir la donnée qui attend sur le bus de lecture, si nécessaire. La figure 4.9 montre la machine à états complète:

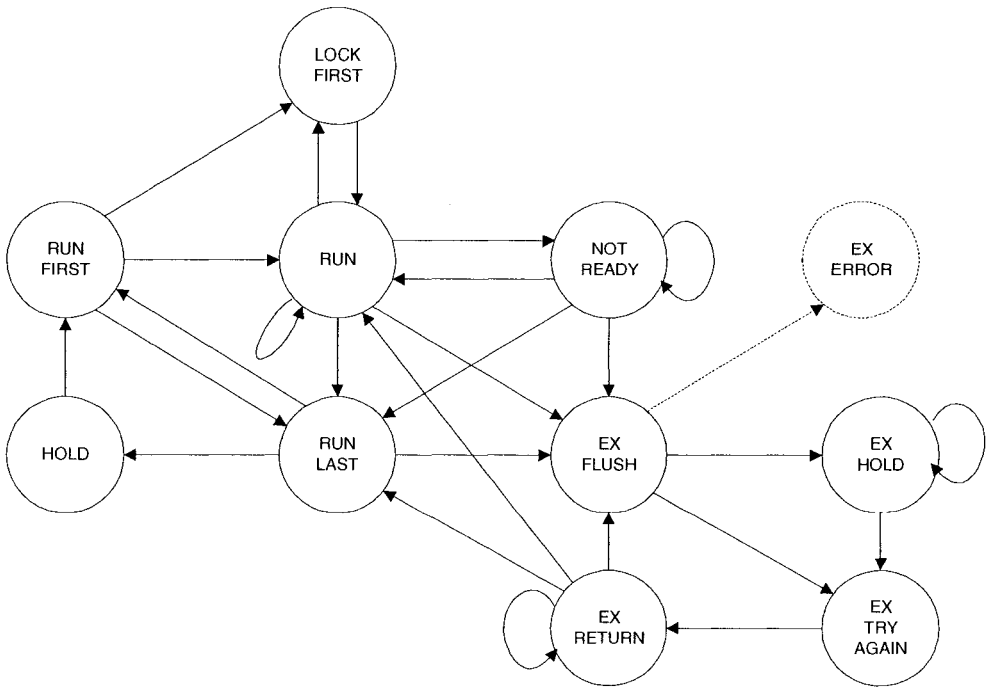

Figure 4.9: La machine à états complète

Ainsi, la complexité de la conception de la machine à états a été contrôlée grâce à une application progressive des règles du protocole. Naturellement, cette structure sert de base: chaque noyau aura sa propre façon d'être adapté et d'autres états sont possiblement nécessaires.

4.3. Structure d'un pont AHB pour un esclave

Une méthodologie semblable à celle de la section précédente peut s'appliquer pour le design des ponts pour les esclaves. Cette section détaille la conception d'une machine à états basée sur les responsabilités d'un esclave AHB.

4.3.1. Les responsabilités des esclaves AHB

Quatre responsabilités ont été identifiées à partir de la spécification AHB:

1. D'abord et avant tout, l'esclave doit échantillonner les signaux d'adresse et de contrôle à tous les cycles de bus, et doit ignorer tout accès où HSEL = 0, HTRANS = IDLE ou HTRANS = BUSY. C'est la règle de *l'échantillonnage*.

2. L'esclave a la responsabilité d'étirer le cycle de bus s'il a besoin de plusieurs coups d'horloge pour répondre adéquatement à un accès. C'est la règle de *l'étirement*.

3. Les réponses autres que "OK" doivent être émises pendant deux cycles d'horloge. Au cours du premier cycle d'horloge, HREADYOUT doit être 0. C'est la règle des *deux cycles*.

4. Les esclaves capables d'émettre des réponses SPLIT doivent pouvoir avertir l'arbitre que l'accès peut être complété. Pour ce faire, l'esclave doit avoir saisi la valeur de HMASTER à la fin de la phase d'adresses qui a causé la réponse SPLIT. La valeur stockée est décodée. Ceci permet alors de savoir quel bit de HSPLIT il est nécessaire d'activer pour avertir l'arbitre. C'est la règle de la *saisie de HMASTER*.

4.3.2. Raffinement progressif de la machine à états

Comme pour le pont du maître, on commence avec un seul état qui boucle sur lui-même (Figure 4.2). Lors de la transition (1), l'esclave vérifie l'état de HSEL et de HTRANS: si ces deux valeurs sont valides, il stocke les signaux de contrôle, et dans le cas d'une écriture, la donnée sur HWDATA. Si les signaux enregistrés décrivent une lecture, il se met à émettre la donnée demandée sur HRDATA.

Pour intégrer la règle de *l'étirement*, on ajoute un ou plusieurs états, selon le nombre de cycles que l'on veut ajouter. La figure 4.10a et 4.10b montrent deux exemples ayant un et deux cycles d'attente respectivement.

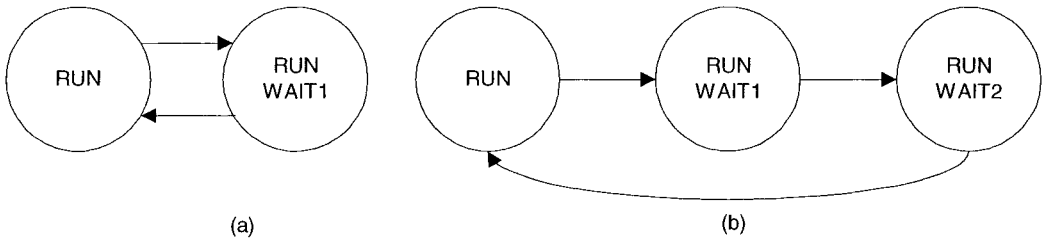

Figure 4.10 : Deux exemples d'états d'attente

Au cours des états d'attente, l'esclave n'échantillonne rien et émet HREADY = 0. Si les états d'attente sont visités à chaque fois que l'esclave est sollicité, il n'y a plus d'utilité pour la transition 1: Les signaux d'adresse et de contrôle sont saisis lorsque le pont entre dans l'état RUN.

Si l'esclave a la possibilité d'émettre une réponse autre que OK, il doit respecter la règle des *deux cycles* énoncée dans la section 4.3.1. Deux états sont ajoutés au pont, au cours desquels HRESP émet une valeur autre que "OK". HREADYOUT est égal à 0 pendant EX_NOT_READY et égal à 1 au cours de EX_READY, tel que spécifié. Il faut une paire d'états par type d'exception. La figure 4.11 montre les modifications:

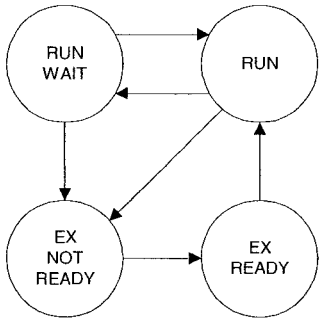

Figure 4.11 : Ajout de la gestion d'exceptions à l'esclave

Enfin, si un esclave utilise des accès SPLIT, il doit être capable d'avertir l'arbitre que l'accès est complété à l'aide de HSPLIT. La figure 4.12 indique les modifications à apporter pour implanter ce comportement. Lors d'une transition dans l'état EX_NOT_READY, le pont saisit la valeur de HMASTER pour noter l'identité du maître qu'il devra réveiller plus tard. Une fois que le pont a terminé sa réponse HSPLIT, il passe à l'état HSPLIT_WAIT où il attend que le noyau esclave ait fini de traiter l'accès. Il passe ensuite à l'état RUN_WARN, au cours duquel le bit de HSPLIT correspondant au maître indiqué précédemment par HMASTER, est activé. Le pont attend qu'il soit sélectionné par le maître qu'il a réveillé avant de passer à l'état RUN.

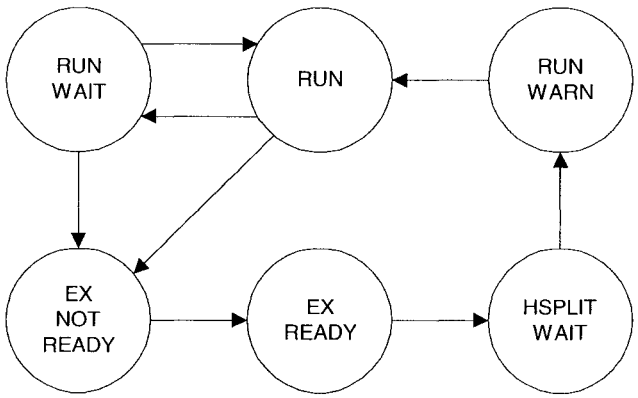

Figure 4.12 : Ajout à la fig. 4.11 de la fonction permettant des accès SPLIT

4.4. Structure d'un arbitre

L'arbitre a un comportement de base très régulier, malgré son enracinement profond à tous les niveaux du système. Il a la responsabilité de gérer l'accès au bus. Pour transmettre la possession du bus d'un maître à un autre, l'arbitre a deux opérations à faire:

d'abord, il doit activer le signal HGRANT du maître qui va obtenir le bus, puis désactiver celui du maître qui va le perdre. Au prochain cycle de bus, il doit émettre le numéro du nouveau possesseur du bus sur HMASTER.

Le comportement d'un arbitre est intimement lié aux caractéristiques du système. Ainsi, il est difficile de présenter une approche unifiée pour sa conception. Cette section ne présente donc que quelques exemples d'arbitres, dans l'espoir qu'ils puissent servir d'inspiration aux concepteurs éventuels.

4.4.1. Arbitre pour un maître et le maître factice

Le protocole AHB stipule que l'arbitre doit retirer le bus d'un maître qui a reçu la réponse SPLIT de la part d'un esclave. Or, dans le bloc-processeur présenté au chapitre précédent, le bus n'a qu'un seul maître réel, le ARM7TDMI. Si jamais un des esclaves donne une réponse SPLIT, l'arbitre doit donner le bus au maître factice (qui est le seul autre maître disponible). Dès que l'esclave signale que l'accès est prêt à être terminé, l'arbitre redonne le bus au maître. La figure 4.13 montre une machine à états qui implante le comportement nécessaire:

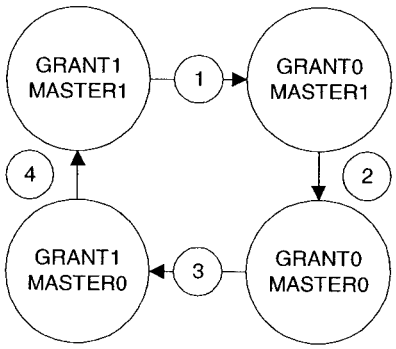

Figure 4.13 : Arbitre pour un seul maître

L'arbitre commence dans l'état GRANT1_MASTER1. Dès que l'arbitre détecte une réponse SPLIT, il sait qu'il doit donner le bus à un autre. Il passe à l'état GRANT0_MASTER1 (1) où il informe les maîtres à l'aide de HGRANT que le maître 0 (le maître factice) est sur le point de recevoir la possession du bus. Au prochain coup d'horloge, l'arbitre change d'état de nouveau (2): HMASTER passe à 0 et le bus n'est plus

au processeur. L'arbitre attend dans cet état jusqu'à ce qu'il reçoive un signal sur HSPLIT. Il passe alors à l'état GRANT1_MASTER0 (3) où il informe le maître 1 (le processeur) que le bus lui revient au prochain cycle de bus. Il revient ensuite à l'état initial (4) au prochain cycle de bus, émettant 1 sur HMASTER, donnant ainsi le bus au processeur.

4.4.2. Arbitre utilisant une priorité statique

Le schème de priorité statique, où chaque maître possède une priorité fixe, offre un exemple intéressant au plan didactique même s'il présente quelques problèmes tels que l'inversion de priorité [BuWe96]. Un arbitre utilisant ce schème n'a qu'un seul état qui émet des signaux de contrôle selon un processus combinatoire qui implante le système de priorités. La figure 4.14 montre cet état, accompagné des processus séquentiel (gauche) et combinatoire (droite). Le processus combinatoire observe les différentes requêtes provenant des maîtres et fixe les sorties HGRANT selon celui qui demande le bus. Ici, on voit que le maître 2 a la plus haute priorité, suivi du maître 1 et enfin le maître 0. Le processus séquentiel est exécuté au début de chaque cycle de bus et observe les valeurs des HGRANT du cycle précédent afin d'affecter la valeur correspondante sur HMASTER.

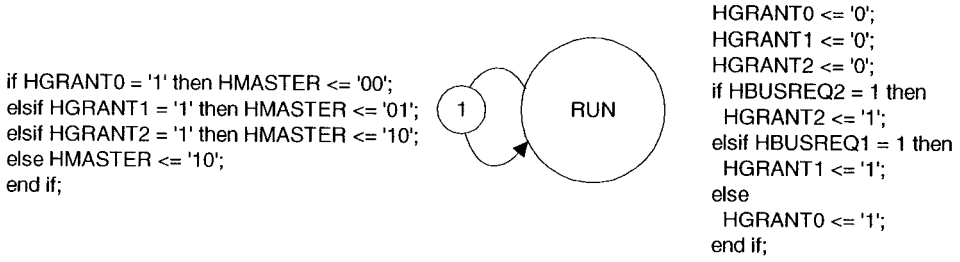

Figure 4.14 : Arbitre avec priorité statique

4.4.3. Arbitre avec respect des rafales

Le protocole AHB fonctionne à son meilleur lorsque les maîtres utilisent le bus par rafales de plusieurs accès. Le comportement prévu de l'arbitre est de permettre aux rafales de se compléter avant de céder le bus à un autre maître. La durée des rafales peut être obtenue en observant HBURST. Au début d'une nouvelle rafale, l'arbitre note la durée de

la rafale et compte les accès pendant qu'ils passent. Il peut donc choisir un moment judicieux pour transférer la possession du bus.

On a de nouveau une machine séquentielle comme dans l'exemple précédent. Au premier cycle, le compte est à 0. HMASTER change et un nouveau maître effectue sa phase d'adresses sur le bus. À la fin du cycle, la valeur de HBURST est stable et donc il est possible de charger le nombre d'accès à faire dans un compteur qui est décrémenté à chaque cycle. Lorsque le compte atteint 1, il est temps de choisir le prochain maître et de l'avertir à l'aide de HGRANT. Lorsque le compte atteint 0, le bus passe à un autre maître.

CHAPITRE 5 : Expérimentation avec la plate-forme

Un des buts du présent travail est de fournir une collection de modules et de macro-modules réutilisables. Afin de témoigner de l'efficacité de la technique de la réutilisation de la propriété intellectuelle, il est intéressant de développer un système en réutilisant les modules de la plate-forme et du bloc-processeur.

Ce chapitre fait état de l'implantation d'une application sur notre plate-forme. La section 5.1 décrit la problématique et le matériel qu'il faut pour cette réalisation. La section 5.2 explique le schème de communication, c'est-à-dire les moyens avec lesquels le processeur ARM7TDMI viendra communiquer avec le matériel spécialisé. La section 5.3 compare le système avec ses versions précédentes, qui avaient été créées avant la venue de la plate-forme. Finalement, la section 5.4 propose un schème de communications alternatif et montre à quel point l'intégration de cette modification est facilitée par la structure régulière du nouveau système.

5.1. Le détecteur de patrons

Pour notre prototype, il fallait une application qui utilisait à la fois un microprocesseur et un coprocesseur (maître) spécialisé permettant un fonctionnement plus efficace. Un exemple souvent utilisé en co-design est un détecteur de patrons [BeFi01]. Il s'agit d'un système qui cherche un patron de 8x8 pixels dans une image de 16x16 pixels. Ce module ferait partie d'un système qui effectue l'encodage MPEG [Kuhn99].

Pour ce faire, une section 8x8 de l'image est choisie, et chaque pixel est comparé avec son homologue dans le patron. Cette sorte d'opération est très coûteuse en temps de traitement pour un microprocesseur: chaque pixel est analysé, un à la fois. Or, les diverses comparaisons n'ont aucune dépendance entre elles. Elles sont donc hautement parallélisables. La solution au problème est donc d'utiliser un module qui porte le nom très original de "détecteur de patrons", un coprocesseur matériel spécialisé.

Le détecteur de patrons charge les pixels à analyser à partir d'une adresse spécifiée par le processeur. Il effectue par la suite les 64 (8x8) comparaisons en parallèle, en un coup d'horloge. La structure de ce coprocesseur est scindée en deux unités ayant des tâches bien précises: le contrôleur et le chemin de données.

Le chemin de données, reçoit les données demandées par le contrôleur et émet un signal si les pixels correspondent bien. Puisqu'il a la capacité de faire une comparaison par cycle, il est nécessaire de trouver un moyen de l'alimenter rapidement en images. Le chargement de pixels sur le bus se fait à raison d'un pixel par cycle de bus, ce qui permet au mieux de faire une comparaison par 64 cycles. L'astuce est donc de conserver tous les pixels déjà lus jusqu'au moment où ils ne sont plus nécessaires. Un moyen simple et efficace qui a été retenu pour cet exemple est de charger les 120 premiers pixels (la latence), comme indiqué dans la figure 5.1 (a). Ainsi, si le patron n'est pas immédiatement détecté, il ne faut charger qu'un seul pixel pour évaluer la prochaine position (b).

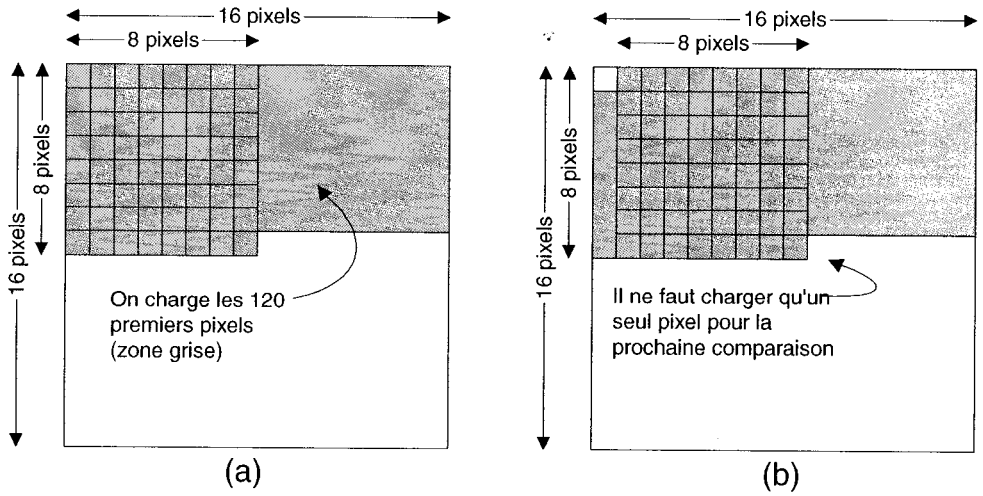

Figure 5.1 : Pré-chargement de pixels

Le contrôleur s'occupe de mémoriser les paramètres de configuration envoyés par le programmeur (les adresses de l'image et du patron) et d'effectuer les accès mémoire pour aller chercher les pixels à analyser. Comme on peut voir dans la figure 5.1, il s'agit d'une machine à états qui a la forme d'un dinosaure:

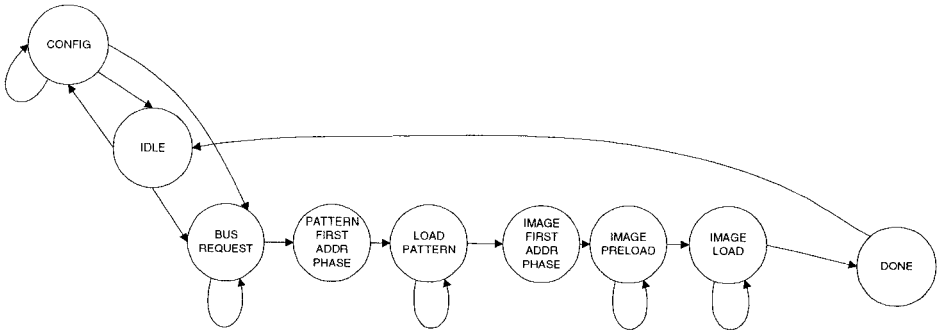

Figure 5.2 : La machine à états du contrôleur

Le "statusaurus" commence dans l'état IDLE, puis passe à l'état CONFIG si on tente de fixer un de ses paramètres. Si on indique au contrôleur de démarrer la comparaison, il suit la séquence du bas: il charge le patron, puis il pré-charge l'image avant de passer à l'analyse proprement dite. Une fois la recherche terminée, il passe à l'état DONE (la "queue" du diplodocus) où un signal est activé pour indiquer que la réponse est disponible.

Si on observe le coprocesseur pour déterminer son identité par rapport au bus on trouve que le détecteur de patrons est à la fois un esclave (pour recevoir les informations du programmeur) et un maître (pour lire les pixels). Il fallait donc créer deux interfaces pour le module. La figure 5.3 montre un schéma-bloc simplifié du détecteur de patrons:

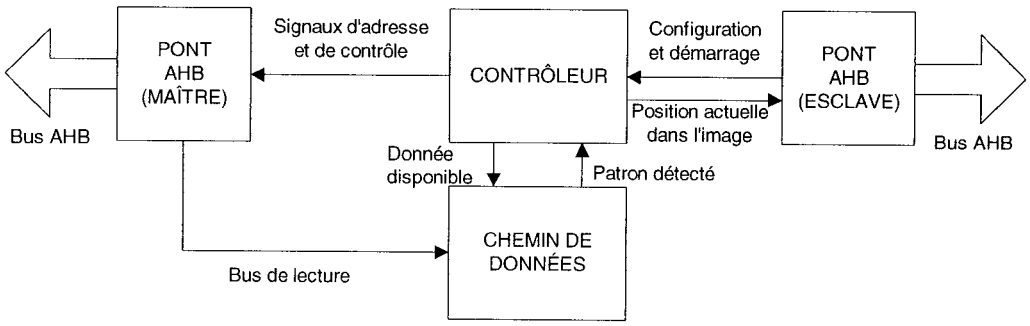

Figure 5.3 : Diagramme-bloc du détecteur de patrons

On voit donc que c'est à travers le pont AHB esclave que le processeur envoie des données de configuration. C'est aussi à travers ce chemin qu'il viendra lire la réponse de la recherche de la position du patron dans l'image. Lorsque le pont esclave met le contrôleur en marche, ce dernier commence à lire des données de la mémoire. Le pont

maître s'occupe de traduire les signaux et de faire attendre le contrôleur si le bus n'est pas disponible. Lorsqu'une donnée est lue, elle est transmise à l'entrée du chemin de données qui la capte si le contrôleur lui demande. Lors de la recherche du patron, le contrôleur observe le signal d'avertissement du chemin de données. S'il est actif, le contrôleur s'arrête et informe le pont que la recherche est terminée. Si aucun patron n'est détecté, le contrôleur s'arrête tout seul et la position prend une valeur spéciale indiquant qu'aucun patron n'a été détecté.

5.2. Communication processeur-coprocesseur

Il existe une infinité de moyens pour établir la communication entre le processeur et le coprocesseur. Pour développer un schème de communication adéquat, il faut d'abord comprendre les interactions des deux modules, puis prendre une décision selon les priorités du design.

5.2.1. Interactions du processeur et du coprocesseur

Dans une application réelle d'encodage MPEG, le processeur joue le rôle de coordonnateur, activant et désactivant des modules selon la progression de l'algorithme. Au début de l'exécution, le processeur configure les modules en leur transmettant des valeurs pour leurs différents paramètres. Dans le cas du détecteur de patrons, ces paramètres sont les adresses du patron et de l'image à analyser. Au cours de l'exécution normale, un périphérique de capture viendrait écrire une adresse en mémoire à l'adresse de l'image (dans le cas de cet exemple, le processeur simule ce comportement en écrivant lui-même l'image). Lorsque le processeur désire le résultat de la recherche du patron, il sollicite le détecteur qui se met en marche. Ce dernier charge le patron et pré-charge une partie de l'adresse, puis se met à analyser l'image, chargeant un pixel par cycle de bus. Une fois qu'il a terminé sa recherche, il informe le processeur qui saisit la position du patron.

On identifie donc trois interactions principales:

1. La configuration des paramètres
2. La mise en marche du détecteur de patrons
3. La transmission de la réponse du traitement

Dans le contexte de ce travail, il était intéressant de choisir une implantation qui allait solliciter le plus de fonctions possibles afin de démontrer le bon fonctionnement d'un maximum des aspects du bus. Un aspect particulièrement tortueux est celui des accès où HRESP = SPLIT, car cette réponse demande beaucoup "d'intelligence" de la part des ponts.

5.2.2. Architecture choisie

Devant ce désir de tester les diverses capacités du bus, il a été décidé d'implanter la mise en marche du détecteur et la transmission de la réponse en une seule transaction utilisant la réponse SPLIT. Une fois qu'il a configuré le coprocesseur en utilisant des écritures normales, le processeur effectue une lecture pour aller chercher immédiatement la réponse. Cette lecture active le détecteur de patrons. Puisque la réponse n'est pas encore disponible, le pont esclave du coprocesseur envoie une réponse SPLIT pour bloquer le maître en attendant. Le pont maître du coprocesseur demande l'accès au bus auquel est reliée la mémoire contenant l'image et le patron, puis commence à lire les données en question. Finalement, lorsque la recherche est terminée et que la réponse est trouvée, le pont esclave réveille (via l'arbitre) le processeur avec le signal HSPLIT. La figure 5.4 montre un diagramme-bloc du système:

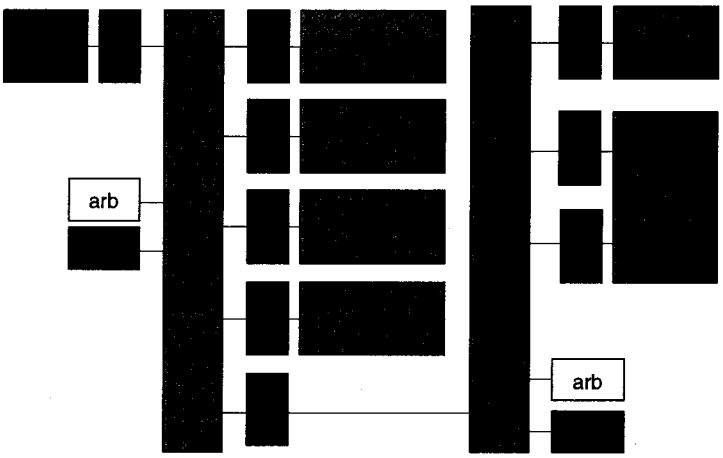

Figure 5.4 : Le système complet

À gauche, on trouve le bus du bloc-processeur. On y retrouve les modules présentés dans la section 3.2. Le bus de droite est hôte pour la mémoire partagée (soit la mémoire qui contient l'image et le patron), le détecteur de patrons et le bloc-processeur (qui se relie au bus externe via le pont AHB-AHB, le "P" au bas de la figure).

L'assemblage du système est simple, car il était tout simplement question de générer un réseau d'interconnexions ayant trois maîtres (le bloc-processeur, le détecteur de patrons et le maître factice) et trois esclaves (la mémoire partagée, le détecteur de patrons et l'esclave par défaut). Relier les modules au bus est également très simple.: le bloc-processeur et la mémoire partagée étaient déjà existants dans la bibliothèque. Le défi est donc de limité au développement des ponts pour le détecteur de patrons.

5.3. Comparaison avec la version antérieure

Devant le nouveau système assemblé, il devient intéressant d'observer en quoi la nouvelle version est meilleure que la version qui la précédait.

5.3.1. Modularité de la structure

La figure 5.5 montre la structure du système précédent:

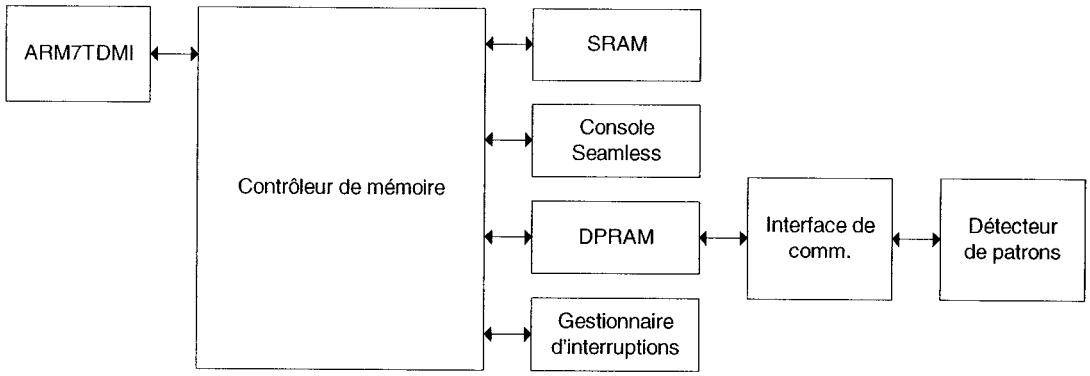

Figure 5.5 : Diagramme-bloc de la version précédente du système

Ce système contient bien sûr un ARM7TDMI et un détecteur de patrons. Ces deux modules communiquent via une mémoire à double port (DPRAM) asynchrone. On retrouve le gestionnaire d'interruptions et la mémoire pour le logiciel (SRAM). Enfin, la console Seamless sert à afficher des messages lors de simulations. On remarque aussi l'absence quasi-totale de ponts, à part l'interface du détecteur de patrons. Le contrôleur de mémoire joue ce rôle en interprétant les accès provenant du ARM7TDMI pour chacun des esclaves. On a donc un super-bloc qui effectue toutes les traductions.

Les modifications du système ne sont pas faciles avec cette structure. Le taux moyen de défectuosités d'un module codé augmente graduellement avec le temps sous l'effet des modifications qu'on lui applique [Press97, p. 12]. Ceci s'explique par le fait que les modifications venant régler un problème dans un module introduisent parfois des effets secondaires sur d'autres parties du code. Ces effets secondaires s'accumulent à chaque modification, rendant le module inutilisable après un certain temps (il faut donc le re-coder). Il est donc préférable de limiter les modifications à un système.

Il est généralement conseillé de séparer les diverses fonctions du système en modules cohérents. Ainsi, les modifications apportées à une fonction ne viendront pas affecter les autres. Dans le cas du design de la figure 5.5, on a un seul bloc qui s'occupe de plusieurs fonctions, ce qui viole ce principe. L'ajout ou le retrait d'un module exige des modifications au contrôleur qui pourraient accidentellement affecter le fonctionnement des autres esclaves.

Le nouveau système développé dans le cadre de ce projet divise les diverses fonctions en utilisant un pont pour chaque esclave, séparant les diverses traductions sur différents modules. Ainsi, l'ajout de nouveaux esclaves se fait facilement (il suffit d'ajouter le noyau et son pont, puis de les relier au réseau d'interconnexion). Le retrait se fait encore plus facilement: on enlève tout simplement les modules et on met à jour le réseau d'interconnexion. Une modification apportée à un pont ne vient pas interférer avec le fonctionnement des autres ponts.

5.3.2. L'utilisation d'un protocole de bus

Comme il a été vu dans la section précédente, l'ancien système utilisait des moyens de communication *ad hoc*: aucun protocole de bus n'était utilisé et donc il fallait s'adapter à chacune des interfaces présentées par les modules. Le contrôleur de mémoire est spécifique au ARM7TDMI. Changer de processeur impliquerait donc de grosses modifications au système: il faudrait concevoir un nouveau contrôleur et effectuer le travail d'analyse nécessaire pour traduire le nouvel ensemble de signaux aux interfaces des esclaves.

Un protocole de bus permet de standardiser les signaux de communication. Changer de processeur n'a presque aucun effet sur les autres modules. En effet, le protocole sert de "zone tampon" entre les maîtres et les esclaves: puisque le maître doit émettre des signaux compatibles au protocole à la sortie de son pont, les ponts des esclaves n'ont qu'à faire une seule traduction pour tous les systèmes possibles. Il devient donc très facile d'expérimenter avec divers processeurs.

L'ancien système ne permettait pas l'arbitrage: la seule façon de relier plusieurs maîtres à un esclave était d'avoir des esclaves multiport, comme la DPRAM de la figure 5.5. Cette technique est efficace au point de vue de la bande passante disponible, puisque chaque maître a un canal dédié pour ces accès. Elle est cependant très coûteuse en termes de surface consommée: c'est un fait connu que des mémoires multiport consomment une quantité immense de transistors. En utilisant le protocole AHB, on peut implanter un

système d'arbitrage afin que tous les maîtres puissent accéder à un esclave, un à la fois, via la même interface.

Enfin, l'ancien système utilisait des mémoires asynchrones, c'est-à-dire des mémoires dont les accès se font à l'aide de signaux asynchrones comme Chip Enable (CE), Output Enable (OE), Write Enable (WE). Ces mémoires, activées grâce à CE, enregistrent ou émettent des données à l'aide de pulsations sur WE et OE respectivement. L'utilisation des cette sorte d'interface est une pratique désuète provenant de l'époque des composantes de mémoire TTL.

Le nouveau système utilise des mémoires statiques synchrones, mieux adaptées à des systèmes sur puce. Le comportement en deux phases du bus AHB a été conçu pour avec des modules de mémoire synchrones en tête.

5.4. Implantation d'un nouveau schème de communication

Il est intéressant de voir à présent à quel point il est facile de modifier le schème de communication. Cette section présente une idée alternative, utilisant cette fois-ci des interruptions. On note ensuite les quelques modifications qu'il faut apporter au système actuel pour implanter cette idée, puis on soulève les difficultés qui auraient été rencontrées dans le système précédent.

5.4.1. L'utilisation d'interruptions dans le système

Il est possible de considérer d'autres schèmes de communication plus efficaces que celui mentionné ci-haut. En effet, lors de la recherche du patron, le processeur attend, perdant de précieux cycles au cours desquels il pourrait effectuer un quelconque traitement. Un moyen efficace de fonctionner serait d'activer le détecteur de patrons, puis d'être averti via une interruption:

1. La configuration des paramètres se fait encore via des écritures

2. La mise en marche du détecteur de patrons se fait grâce à une écriture à une adresse autre que celle des paramètres. Le processeur se met alors à attendre l'interruption avant d'aller chercher le résultat.
3. Le processeur, averti par l'interruption, vient lire la réponse.

5.4.2. Modifications à apporter au système

Grâce à la structure très modulaire du système, le schème de communication au complet peut être changé en modifiant le pont esclave du coprocesseur. En effet, pour la mise en marche du module, il suffit d'observer les adresses à l'entrée lors d'écritures: si l'adresse correspond à l'adresse de mise en marche, le coprocesseur commence sa recherche. Pour avertir le processeur de la fin de la recherche, le pont utilise le signal de fin de recherche et l'achemine à un port d'interruptions du bloc-processeur (bref, la ligne d'interruption vient se connecter au gestionnaire d'interruptions.

5.4.3. Avantages par rapport au système précédent

Le système précédent n'aurait pas été aussi facile à modifier: puisque le processeur et le coprocesseur ne sont pas reliés ensemble sur un même bus, on configure le coprocesseur en écrivant les valeurs des paramètres dans des adresses fixées d'avance de la mémoire à double port. Cette pratique nuit au processus de conception car il faut changer le pont du détecteur de patrons si jamais l'adresse de la mémoire change. Pour lancer le coprocesseur, il faut ajouter un module esclave sur le "bus" du processeur, entraînant une modification du contrôleur de mémoire.

On constate donc que l'utilisation d'un protocole de bus et d'une structure modulaire vient aider l'exploration architecturale en offrant un contexte résistant aux modifications.

- - -

Grâce au système développé, il fut possible d'évaluer la performance de certains modules de la plate-forme. L'analyse est présentée au chapitre suivant.

CHAPITRE 6: Analyse de performance et discussions

Le système présenté au chapitre 5 a permis la réalisation de certaines analyses et a soulevé quelques points sensibles de l'architecture sur lesquels il serait sage de se pencher. Dans un système comme celui-là, il est important de s'assurer que les voies de communications permettent aux modules de fonctionner à une fréquence maximale. Dans la section 6.1, une analyse du réseau de communication sera effectuée afin de le caractériser. La section 6.2 discutera plutôt de l'effet d'insérer des ponts entre des bus différents et exposera les faiblesses potentielles de l'implantation actuelle de certains modules.

6.1. Analyse du réseau d'interconnexion

Généralement, les systèmes embarqués sont caractérisés par la dissipation de puissance et la surface requise pour la logique. Dans cette section, les trois critères seront évalués pour le réseau d'interconnexion. Puisque les caractéristiques changent selon le nombre de maîtres et d'esclaves, les analyses sont effectuées sur diverses configurations. Des mesures ont été prises d'abord avec deux maîtres (un maître normal et le maître factice) et un nombre grandissant d'esclaves. Ensuite, les mêmes mesures ont été prises avec 3 esclaves et un nombre grandissant de maîtres. Les analyses ont été effectuées sur la technologie 0.35 microns.

6.1.1. Surface

La première analyse effectuée était celle de la surface occupée. Comme l'indiquent les graphiques aux figures 6.1 et 6.2, la surface occupée est en fonction du nombre de modules que le réseau accommode. On établit la relation suivante:

$$A = Mx + Ey + C$$

x est le nombre de maîtres, y est le nombre d'esclaves, A est la surface, et M, E et C sont des coefficients. À l'aide d'une analyse de régression sur les données recueillies (annexe C), on a établi la relation suivante:

$$A = 10342x + 6279y - 13044$$

La réponse est exprimée en micromètres carrés. On remarque que la taille croît linéairement avec le nombre de maîtres. On remarque aussi que le nombre de maîtres n'a aucun effet sur la "surface séquentielle", soit la surface représentant les registres (la "surface combinatoire" représente la surface occupée par le reste de la logique).

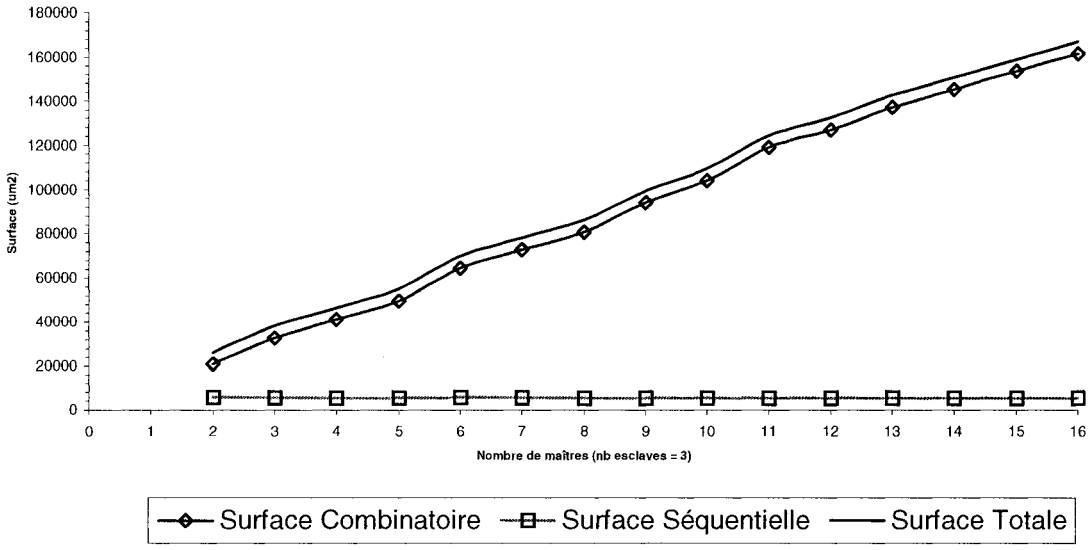

Figure 6.1 : Surface occupée en fonction du nombre de maîtres

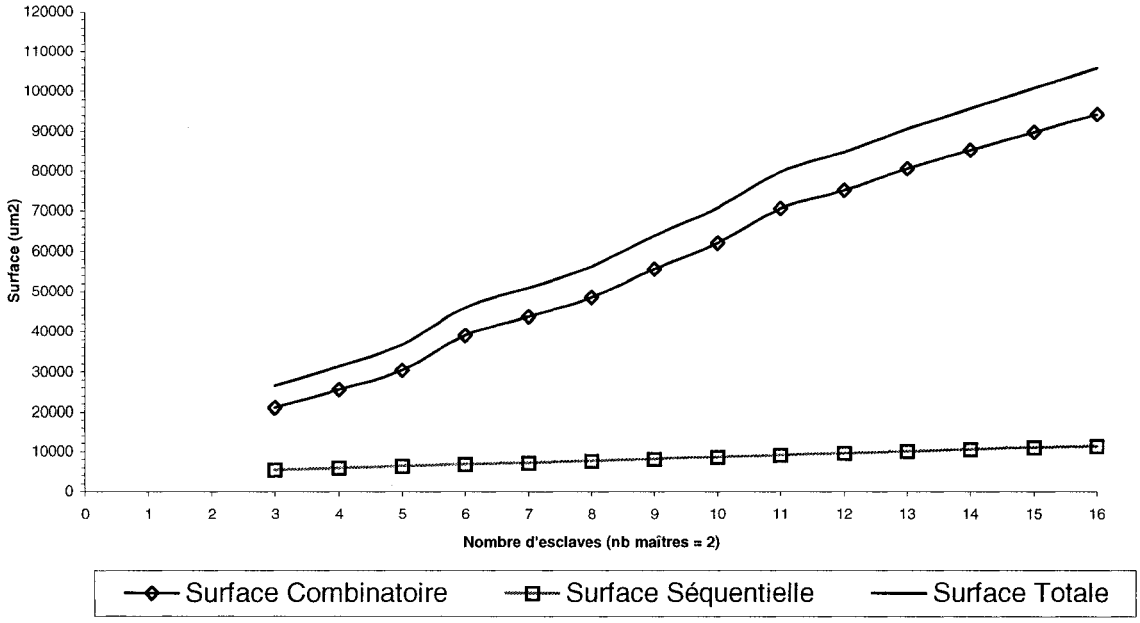

Figure 6.2 : Surface occupée en fonction du nombre d'esclaves

6.1.2. Consommation de puissance

La consommation de puissance est fonction de la tension d'opération (alimentation), de la capacité de divers éléments du circuit et de la fréquence d'opération. Le nombre de maîtres et d'esclaves affecte la consommation, car la taille du circuit augmente: il y a plus de signaux qui peuvent basculer. La figure 6.3 présente les résultats de l'analyse avec une fréquence de 40MHz, et une tension de 3.135 V:

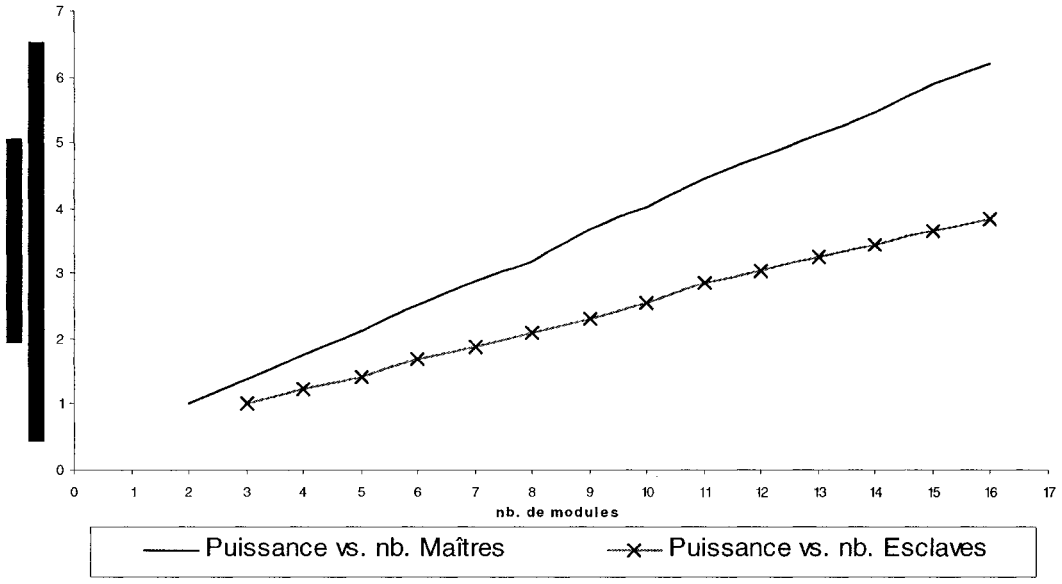

Figure 6.3 : Analyse de puissance du réseau d'interconnexion

On trouve donc une relation semblable à celle de la surface. Après une étude de régression on obtient la relation suivante:

$$0.35x + 0.22y - 0.46.$$

6.2. Analyse du pont AHB-AHB

Deux approches existaient au moment de la conception du pont AHB-AHB. La première était de transmettre les signaux correspondants de façon combinatoire. L'autre était de les enregistrer et de les retransmettre de l'autre côté au prochain coup d'horloge. La première méthode fait en sorte qu'on épargne un ou plusieurs cycles d'horloge, mais la propagation d'un signal dure plus longtemps, limitant ainsi la fréquence maximale du système. La figure 6.4 montre la différence entre les deux. On voit en (a) que le HADDR distant est disponible dans le même cycle que le HADDR local, mais qu'il accumule un délai à cause du passage à travers le pont et le réseau d'interconnexion. En (b), le HADDR distant a un cycle de retard sur le HADDR local, mais il est disponible plus tôt dans le cycle, permettant une cadence d'horloge plus élevée.

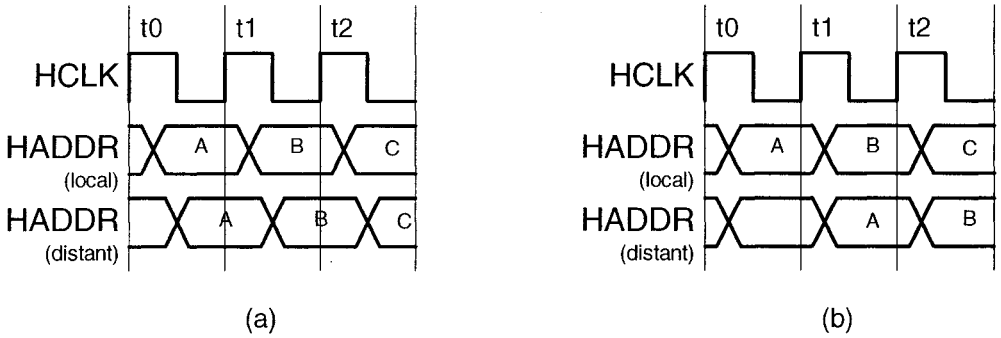

Figure 6.4 : Les deux options pour le pont AHB-AHB

La première approche a été retenue. Les délais de propagation ne pouvant être évalués avec le modèle de simulation utilisé (délais de durée delta), il avait été jugé important, au début du projet, de réduire le nombre de cycles insérés lors d'accès. Cette approche n'est cependant pas réaliste si on considère un accès traversant plusieurs niveaux de bus. Chaque passage à travers un pont ou un réseau d'interconnexion retarde les signaux de plus en plus. En rétrospective, cette décision peut venir limiter la fréquence maximale du système.

6.2.1. La structure du pont AHB-AHB

La structure actuelle du pont s'inspire donc de la première approche (Figure 6.4a). Il s'agit d'une machine à états qui contrôle l'acheminement et la traduction des signaux locaux vers les signaux distants (et vice versa). Ces sorties combinatoires insèrent un délai, et la discussion qui suit utilise comme exemple la propagation de HADDR vers le bus distant. Dans la section 4.2.3, on a vu que pour gérer correctement les exceptions, il faut parfois forcer une valeur sur le HADDR distant. HADDR passe donc par un multiplexeur. Le réseau d'interconnexion vient aussi ajouter des délais : HADDR doit traverser le multiplexeur de contrôle qui lui-même dépend de la valeur de HMASTER provenant de l'arbitre. Non seulement la transmission de HADDR est-il dépendant de plusieurs autres signaux, mais en plus, il est transmis à tous les esclaves, augmentant ainsi sa charge capacitive.

Il est théoriquement possible de relier plusieurs bus ensemble avec des ponts. La figure 6.5 montre l'effet accumulé de tous les délais. On constate que plusieurs niveaux de ponts et de bus peuvent faire en sorte qu'il devienne nécessaire de baisser la fréquence de l'horloge pour permettre à HADDR d'arriver à l'esclave à temps.

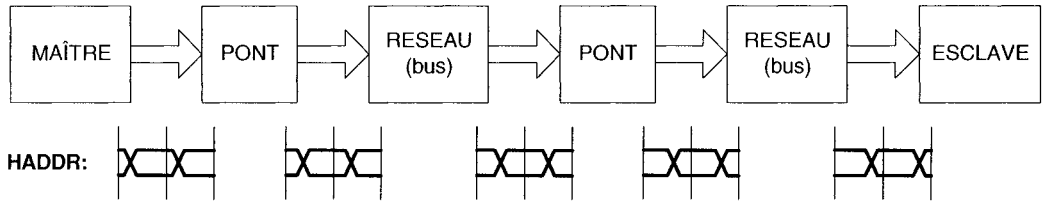

Figure 6.5 : Accumulation des délais sur HADDR

Le pipelinage est la solution classique à ce problème. En divisant le chemin combinatoire en plusieurs segments séparés par des bascules, les signaux ont moins de distance à parcourir en un cycle d'horloge. Cette solution qui constitue la seconde approche de la section précédente (Figure 6.4b), effectue du pipelinage et doit donc être considérée dans les cas où on veut augmenter la vitesse de l'horloge.

Il faut cependant noter qu'il n'est pas toujours possible d'augmenter la fréquence d'horloge. Parfois il y a des objectifs de surface ou de puissance à respecter. Certains modules limitent peut-être déjà la vitesse du bus. Dans ce cas, il est intéressant de retenir la première option. En effet, si la fréquence d'horloge ne peut pas être accélérée, il faut tenter de limiter le nombre de coups d'horloge perdus par l'attente.

6.2.2 Autres considérations concernant le pont AHB-AHB

Le protocole AMBA spécifie que le nombre de cycles d'horloges où HREADY est bas devrait se limiter à 16. Ainsi, un maître ne possède jamais le bus trop longtemps, permettant aux autres d'y accéder dans un délai raisonnable.

Or, le pont utilise HREADY pour inhiber le maître pendant qu'il attend que le bus distant se libère. Le nombre de cycles est donc indéfini, et peut être long si plusieurs maîtres se partagent le bus distant. Dans cette situation, il est acceptable de violer la règle, car le bus

local ne contient qu'un seul vrai maître (le maître factice ne demande jamais le bus). Il serait cependant intéressant de trouver un moyen de respecter cette règle.

6.3 Amélioration du pont ARM7TDMI-AHB

La méthodologie présentée au chapitre 4 a été appliquée lors de la conception du pont du processeur ARM7TDMI. Malheureusement, le pont est *compatible* avec AHB mais non pas *conforme*. En effet, dans la spécification AMBA AHB, les signaux d'adresse et de contrôle changent dans la première phase du cycle d'horloge. Dans le cas du ARM7TDMI, les signaux changent dans la seconde phase, d'où la non-conformité. Puisque le protocole AHB ne fonctionne que sur les fronts montants de l'horloge, la différence ne semble pas avoir d'effet sur le du système. C'est pour cela qu'il est *compatible.*

Les concepteurs de AHB ont cependant suggéré que les signaux soient changés en début de cycle pour une raison précise: puisque le décodeur utilise HADDR pour générer les signaux HSEL, il est préférable que le signal d'adresse soit stable le plus rapidement possible. Les signaux d'adresse et de contrôle peuvent aussi servir à influencer une décision de l'arbitre (et donc la valeur de HGRANT à la fin du cycle). On peut donc dire que HSEL et HGRANT sont dépendants de façon combinatoire à HADDR et les autres signaux de contrôle. Il existe un délai de propagation entre le moment où HADDR change et le moment où HSEL prend la nouvelle valeur. La figure 6.6 montre comment ce délai vient influencer la fréquence maximale du système.

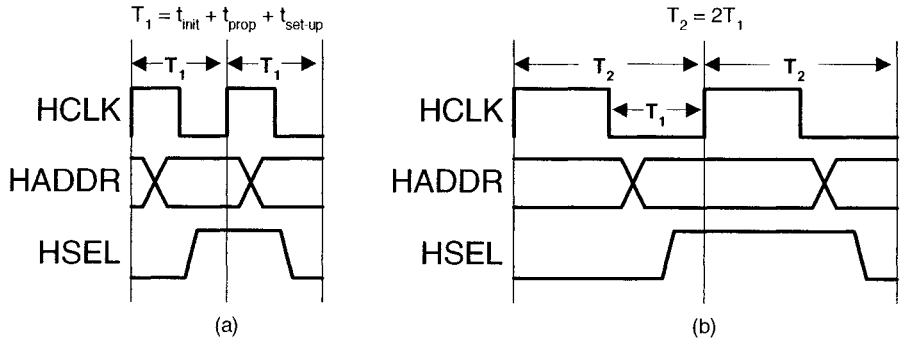

Figure 6.6: Différences de période

Si on prend l'exemple de HSEL, on note qu'il doit être valide au plus tard à t_{set-up} avant le coup d'horloge afin d'être lu correctement par les esclaves. Le temps de propagation t_{prop} est le temps que prend le décodeur à produire HSEL à partir de HADDR. Enfin, on a aussi t_{init}, la quantité de temps après le coup d'horloge où HADDR est valide. La période minimale dans un système AHB est présenté à la figure 6.6a), si on ne considère que ces signaux, est donc $t_{init} + t_{prop} + t_{set-up}$. Or, dans le pont non-conforme produit dans ce travail (figure 6.6b), la période minimale est 2 x ($t_{init} + t_{prop} + t_{set-up}$), car HADDR change après le milieu du cycle. Ainsi, on risque de limiter sévèrement la fréquence maximale du système si on garde le pont tel qu'il est. Deux options s'offrent:

1. Ignorer le problème. Le ARM7TDMI ne fonctionne pas généralement à une fréquence très élevée. Il n'est peut-être pas nécessaire d'accélérer davantage l'horloge.

2. Le ARM7TDMI a des ports qui permettent d'indiquer au système de mémoire si l'adresse du prochain accès a un lien avec l'adresse actuelle. Si SEQ = 1, on peut deviner que l'adresse du prochain accès sera 2 ou 4 de plus que l'adresse de l'accès actuel. Ainsi, on peut anticiper occasionnellement l'adresse du prochain accès. L'approche consiste à insérer un cycle d'attente (où HTRANS = IDLE et où le processeur est bloqué) lorsque le prochain accès n'est pas séquentiel, puis d'anticiper l'adresse à partir des cycles suivants. La figure 6.2 décrit le comportement désiré:

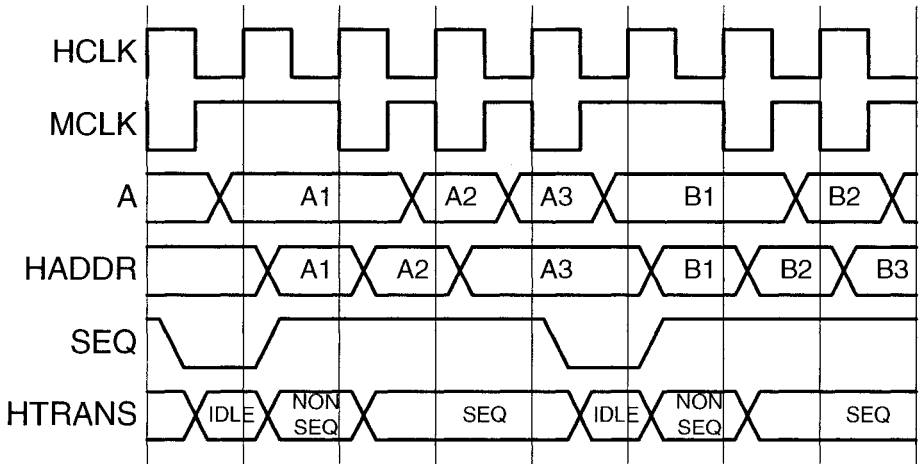

Figure 6.7 : Comportement désiré du pont ARM7-AHB

Dès que SEQ = 0 dans la deuxième phase du cycle d'horloge, on annule l'accès en cours en forçant HTRANS = IDLE. On bloque ensuite le maître pendant un cycle et HADDR prend la valeur de A. Ceci donne le temps au pont d'anticiper les adresses qui suivront. On suppose que SEQ = 1 et que la prochaine adresse sera donc 2 ou 4 (selon MAS) de plus que l'adresse actuelle. À l'aide d'un additionneur ou d'un compteur, on peut incrémenter l'adresse actuelle pour la fournir au début du cycle suivant.

Bien que la deuxième solution ajoute des cycles d'attentes, elle permet en théorie de doubler la cadence de l'horloge. Selon une source de ARM, le nombre de cycles serait augmenté de 33%. On a donc un gain en fréquence de 50,4 %, dans la mesure où le processeur permet cette cadence.

Conclusion

Ce chapitre conclut le travail en résumant les notions présentées tout au long du mémoire, puis propose des améliorations à apporter au projet pour encourager des travaux futurs.

Résumé du travail accompli

Ce travail a contribué une expérience de conception pratique des SoC, basée sur les recommandations théoriques de l'état actuel de la recherche. Il y avait une lacune à cet égard, car les spécifications et les méthodologies offrent rarement des suggestions sur la réalisation matérielle des concepts proposés. La plate-forme produite au cours de ce projet pourra désormais servir d'exemple pour les débutants voulant apprendre, entre autres, le fonctionnement du protocole AMBA AHB et l'utilisation du processeur ARM7DMI.

Cette connaissance sera très utile dans un avenir rapproché, puisque la Société canadienne de microélectronique est actuellement en train d'obtenir un modèle synthétisable du ARM7TDMI pour les universités canadiennes, ainsi que plusieurs plate-formes utilisant le protocole AHB. Ce travail vient défricher les spécifications plutôt arides et prépare le terrain à l'utilisation de ces outils formidables.

C'est dans cette optique que le chapitre 2 du présent ouvrage a été créé. Plutôt que de répéter les règles et les descriptions, le chapitre a tenté plutôt de mettre l'accent sur les interactions entre les divers acteurs (maître, esclave, arbitre, décodeur, pont) afin d'éclaircir le lecteur sur l'utilité de chacun des signaux et de lui montrer dans quelles situations ils sont utilisés. Les trois protocoles AMBA ont été présentés: le protocole AHB vient remplacer ASB, qui est mal ciblé pour les SoC et qui est donc désuet. Ces deux protocoles servent pour la haute performance, comme pour les interactions processeur-mémoire. Il existe aussi APB, qui est un protocole allégé pour les périphériques lents. Le chapitre se termine avec une discussion sur divers moyens d'augmenter la bande passante.

Le chapitre 3 décrit le système conçu. On a vu que le système repose principalement sur le bloc-processeur, un module un processeur ARM7TDMI, de la mémoire, un gestionnaire d'interruptions et une minuterie (pour la préemption de tâches sur le processeur). Ce module contient un bus AHB dédié au processeur qui lui offre une bande passante maximale. Pour accéder à un esclave partagé sur un bus externe, un pont est fourni. Le bloc-processeur devient donc un maître sur le bus partagé. Grâce à la structure modulaire du système, il est possible d'utiliser plusieurs blocs-processeurs sur un même bus partagé. On peut donc construire un système multi-processeur facilement à l'aide de la réutilisation de blocs de propriété intellectuelle.

Dans la revue de littérature (chapitre 1), on a souligné l'importance de séparer des modules en deux parties: le noyau (qui s'occupe du calcul et du traitement) et le pont (qui s'occupe des communications). Ainsi, si on veut changer les moyens de communication, il est seulement nécessaire de modifier le pont. Le chapitre 4 présente une méthodologie pour la conception des ponts. La première étape consiste à identifier dans le protocole implanté toutes les responsabilités que le module doit assumer. La deuxième étape consiste à dresser une liste des similitudes entre le comportement du noyau et le comportement exigé par la spécification. Au cours de la dernière étape, une machine à états est créée. Selon les différences ou les lacunes du noyau face à ses responsabilités, des états sont ajoutés au pont pour compenser. On trouve aussi au chapitre 4 une méthode pour la génération automatique du réseau d'interconnexion pour relier les divers modules sur un bus AHB. Ce chapitre fournit aussi quelques exemples concrets pour la conception d'arbitres.

Le chapitre 5 décrit un exemple monté à l'aide de la plate-forme. Il s'agit d'un système qui effectue la détection d'un patron dans une image à l'aise d'un module matériel spécialisé. Il s'agit d'un coprocesseur qui peut évaluer en un coup d'horloge si un patron se trouve à une certaine position dans une image. Le système complet est composé d'un bus partagé entre le détecteur de patrons et un bloc-processeur. L'image et le patron sont stockés dans une mémoire commune sur ce bus. Le but de l'exemple étant de mettre à l'épreuve le plus

de fonctions possibles, la synchronisation entre le processeur et le coprocesseur s'est fait à l'aide de l'exception SPLIT du protocole AHB.

Une comparaison avec une version précédente a été établie, montrant qu'effectivement, le fait d'avoir respecté les principes de la réutilisation simplifie la modification du système. Le changement des schèmes de communication et des moyens de synchronisation sont des problèmes qui peuvent devenir très épineux. Néanmoins, l'intégration de telles modifications peut se faire avec peu de difficultés dans le nouveau système grâce à sa structure très régulière et modulaire.

Dans ce chapitre on peut aussi trouver une évaluation des latences de fonctionnement du pont AHB-AHB qui sert à relier le bloc-processeur…

Travaux futurs et améliorations

La durée d'un projet de maîtrise étant toujours limitée, il demeure parfois des parties du travail qui ne sont pas raffinées comme on le souhaiterait: on se satisfait d'une version fonctionnelle mais non optimale des modules. Parfois de nouveaux problèmes surviennent lors de la réalisation. Les section 6.2 et 6.3 ont fait état des améliorations possibles concernant les pont AHB-AHB et ARM7-TDMI-AHB, respectivement. Cette section fait état du travail qui reste à faire et des améliorations à apporter pour d'autres aspects de la plate-forme.

Initialisation du système

La plate-forme actuelle n'a pas de mécanisme pour son initialisation: les SSRAM sont remplies par un artifice de simulation provenant de Seamless. Des versions futures de la plate-forme devront comporter un moyen réel de remplir les SSRAM.

La première solution envisagée est de remplacer la SSRAM contenant les instructions par une ROM. Cette solution serait très pratique, mais on reproche souvent aux ROMs leur faible fréquence maximale de fonctionnement. Cette approche n'est donc pas

envisageable pour d'éventuels blocs-processeurs contenant des microprocesseurs relativement performants.

Une autre possibilité est d'ajouter un module DMA qui saisit le contrôle du bus du bloc-processeur dès la mise à zéro. Il copierait le code d'une ROM (sur le bus local ou sur un bus partagé) vers la SSRAM de code. Pour cela, il faudrait modifier le bus local pour accommoder un maître de plus et changer l'arbitre pour qu'il donne un accès prioritaire au module DMA. Cette solution est peu intéressante, parce qu'il faut programmer les adresses des modules de mémoire directement dans le code du DMA, ce qui nuit à l'encapsulation.

Enfin, une dernière solution, déjà utilisée pour certains noyaux commerciaux comme le ARM940T, est de remplacer les mémoires par des caches. Ces caches se relient directement au bus externe. Cette solution est élégante parce qu'elle élimine le besoin d'avoir un pont AHB-AHB. Lors des échecs, les caches effectuent des accès sur une mémoire externe partagée qui contient toutes les instructions.

Ponts ayant des domaines temporels différents

Tous les ponts conçus dans ce projet fonctionnent avec la même vitesse d'horloge sur leurs deux interfaces. Le ARM7TDMI fonctionne à 40MHz, mais que faire pour les systèmes qui ont besoin d'une fréquence plus élevée sur le bus distant? Comme il a été vu dans le chapitre 6, la conception de ponts peut devenir très complexe. C'est un domaine encore peu exploré mais très important.

Structures alternatives de bus

Il existe de la recherche qui a été effectuée sur des structures de systèmes autres que le bus. On note par exemple les divers *network-on-chip* qui ont une structure semblable à des réseaux Éthernet traditionnels et l'*Octagon* [KNDR01]. Ces structures offrent un défi de conception très intéressant, notamment au niveau de la sérialisation et le tamponnage des accès.

Considérations finales

Le marché des composantes, outils et méthodes de conception micro-électronique est un domaine où la compétition est féroce dans lequel chacun garde ses secrets pour lui. Il est frustrant pour un débutant de se heurter à des secrets professionnels lorsqu'il veut apprendre.

Le but de ce projet était de créer une plate-forme qui pourrait à la longue servir d'outil didactique, un exemple pour les néophytes qui cherchent des exemples de la conception avec AMBA. Il est donc important de s'assurer de la diffusion et du support de ce travail.

RÉFÉRENCES

[ARM00] ARM. 2000. *AMBA – Advanced Microcontroller Bus Architecture Specification.* [En ligne]. Cambridge, Angleterre: Advanced RISC Machines. ARM IHI 0011A. http://www.arm.com/arm/ documentation?OpenDocument. (Page consultée le 18 mars 2003)

[ARM01] ARM. 2001. *ARM7TDMI Technical Reference Manual, Revision 4.* [En ligne]. Cambridge, Angleterre: Advanced RISC Machines. ARM IHI 0011A. http://www.arm.com/arm/ documentation?OpenDocument. (Page consultée le 18 mars 2003)

[ARM01b] ARM. 2001. *Multi-Layer AHB Overview.* [En ligne]. Cambridge, Angleterre: Advanced RISC Machines. ARM IHI 0011A. http://www.arm.com/arm/ documentation?OpenDocument. (Page consultée le 18 mars 2003)

[ARM01c] ARM. 2001. *AHB-Lite Overview.* [En ligne]. Cambridge, Angleterre: Advanced RISC Machines. ARM IHI 0011A. http://www.arm.com/arm/ documentation?OpenDocument. (Page consultée le 18 mars 2003)

[ARM01d] ARM. 2001. *AHB FAQ.* [En ligne]. Cambridge, Angleterre: Advanced RISC Machines. ARM IHI 0011A. http://www.arm.com/support.nsf/html/amba?OpenDocument&style=Technical_Support. (Page consultée le 18 mars 2003)

[BCRB03] BENNY O., CHEVALIER J., RONDONNEAU M., BOIS G., BOYER F., ABOULHAMID M., *SPACE: A Hardware/Software SystemC modeling platform including an RTOS*, soumis à Forum on Design Language (FDL) 2003, Allemagne

[BeFi01] BERTOLA Marc, FILION Luc, BOIS Guy. 2001. *Block Matcher Seamless CVE Tutorial.* [En ligne] 38p. http://www.grm.polymtl.ca /circus/commun/publications/ (Page consultée le 18 mars 2003)

[Broo95] BROOKS Frederick P., 1995, *The mythical man-month*, 2e édition, Addison-Wesley Professional, 322 p.

[BuWe96] BURNS Alan, WELLINGS Andy, 1996, *Real-Time Systems and Programming Languages*, Addison-Wesley, 611 p.

[CDK00] COULOURIS George, DOLLIMORE Jean, KINDBERG Tim, 2000, *Distributed Systems: Concepts and Design*, 3e édition, Addison-Wesley

[CIRC01] CIRCUS, École polytechnique de Montréal, Site Web, 2001, www.grm.polymtl.ca/circus

[Cord99] CORDAN Bill, 1999, "An Efficient Bus Architecture for System-on-Chip Design", *IEEE 1999 Custom Integrated Circuits Conference Proceedings*, IEEE, p 623-626

[CMC01] POLLITT-SMITH Hugh, 2001, Discours donné à l'événement TEXPO 2001, Ottawa, document interne

[Fili00] FILION Luc, 2002, *Analyse, implantation et intégration d'une bibliothèque pour la spécification des systèmes embarqués dans une méthodologie de codesign*, Mémoire de maîtrise, Département de génie électrique, École Polytechnique de Montréal.

[GZDG00] GAJSKI Daniel D., ZHU Jianwen, DÖMER Rainer, GERSTLAUER Andreas, et al. 2000. *SpecC: Specification Language and Methodology.* Norwell, MA : Kluwer Academic Publishers. 1-5; 14-22; 55-68.

[Hene01] HÉNEAULT Yannick. 2001. *BasicARM - Technical document for HW/SW codesign and simulation on the BasicARM Platform.* [En ligne]. Groupe de recherche sur le codesign (CIRCUS). 31p. http://www.grm.polymtl.ca /circus/interne

[HePa96] HENNESSY John L., PATTERSON Davis A. 1996. *Architecture des ordinateurs – Une approche quantitative 2e édition.* Paris : Morgan Kaufmann Publishers. 3-5, 527-544.

[IBM00] IBM, 2000, *Processor Local Bus Architecture Specifications*, 1ère édtion, disponible sur demande seulement.

[Kozi01] KOZIEROK Charles M, 2001, "Memory speed, access and timing", document en ligne, http://www.pcguide.com/ref/ram/timingAccess-c.html (Page consultée le 18 mars 2003)

[KNDR01] KARIN Faraydon, NGUYEN Anh, DEY Sujit, RAO Ramesh, 2001, "On-Chip Communication Architecture for OC-768 Network Processors", *DAC 2001 Proceedings*, ACM SIGDA, Las Vegas

[Kuhn99] KUHN Peter M., 1999, *Algorithms, Complexity Analysis and VLSI Architectures for MPEG-4 Motion Estimation*, Boston : Kluwer Academic Publishers, 248 p.

[LSIL98] LSI Logic, 1998, *Green AMBA Peripherals*, PDF anciennement disponible sur le site Web. N'est plus disponible.

[MeGr97] MENTOR GRAPHICS, 1997, Site Web de Seamless: http://www.mentor.com/seamless (Page visitée le 18 mars 2003)

[Nepp98] NEPPL Franz, 1998, "Intellectual Property will cure design woes", *IEEE Spectrum*, Janvier 1998

[NiGo01] NIGHTINGALE Andy, GOODENOUGH John, 2001, "Testing for AMBA Compliance", *14th Annual IEEE International ASIC/SOC Conference Proceedings*, IEEE, p 301-305

[OSCI02] OSCI. 2002. *SystemC Version 2.0.1 User's Guide*. [En ligne]. http://www.systemc.org (Page consultée le 18 mars 2003)

[Pres97] PRESSMAN RogerS., 1997, *Software Engineering: A Practitioner's Approach,* 4e Édition, McGraw Hill, New York, 852 p.

[Sili01] Silicore, 2001, *Wishbone System-On-Chip (SoC) Interconnection Architecture for Portable IP Cores*, révision B.2, disponible en-ligne à www.silicore.net/wishbone.htm (Page consultée le 18 mars 2003)

[VSIA00] VSIA On-Chip Bus Development Working Group 2000. *Virtual Component Interface Specification (OCB 2 2.0).* Disponible aux membres de VSIA.

ANNEXES

Annexe A : Génération du réseau d'interconnexion

La génération du code VHDL du réseau d'interconnexion s'effectue à l'aide d'une fonction en Java. Cette fonction construit graduellement le contenu du fichier VHDL en accumulant ses diverses parties dans la chaîne de caractères "vhdlData".

La fonction reçoit quatre paramètres. *numMasters* et *numSlaves* indiquent le nombre de maîtres et d'esclaves qui seront reliés au bus. Si dummyMaster est vrai, les ports pour le maître 0 sont remplacés par le maître factice. De façon semblable, si defaultSlave est vrai, le code pour l'esclave par défaut vient remplacer les port pour l'esclave 0.

```
import java.io.*;
import java.lang.*;

public class InterconnectAMBA extends Object
  {
  public static String makeAHBMatrix(int numMasters, int numSlaves,
                                      boolean dummyMaster, boolean defaultSlave)
    {
    int i;
    String vhdlData;
    String allZeros;
    String currentModule;

    allZeros = new String("0000");

    // Generate header and entity declarations
    vhdlData =
      "library IEEE;\n" +
      "use IEEE.std_logic_1164.all;\n" +
      "\n" +
      "entity ahb_matrix_"+
      Integer.toString(numMasters) +
      "x" +
      Integer.toString(numSlaves) +
      " is\n"+
      "  port (\n";
```

```
    // Generate ports for masters
    for (i=0;i<numMasters;i++)
      {
      if (!((i==0) && (dummyMaster==true)))
       {
       vhdlData +=
       "       -- Master " + Integer.toString(i) + " ports --\n" +
       "       hgrant_m" + Integer.toString(i) + "     : out std_logic;\n" +
       "       hready_m" + Integer.toString(i) + "     : out std_logic;\n" +
       "       hresp_m" + Integer.toString(i) + "      : out std_logic_vector(1 downto
0);\n" +
       "       hrdata_m" + Integer.toString(i) + "     : out std_logic_vector(31 downto
0);\n" +
       "       hclk_m" + Integer.toString(i) + "       : out std_logic;\n" +
       "       hreset_m" + Integer.toString(i) + "_n   : out std_logic;\n" +
       "       haddr_m" + Integer.toString(i) + "      : in std_logic_vector(31 downto
0);\n" +
       "       hwrite_m" + Integer.toString(i) + "     : in std_logic;\n" +
       "       hsize_m" + Integer.toString(i) + "      : in std_logic_vector(2 downto
0);\n" +
       "       htrans_m" + Integer.toString(i) + "     : in std_logic_vector(1 downto
0);\n" +
       "       hburst_m" + Integer.toString(i) + "     : in std_logic_vector(2 downto
0);\n" +
       "       hprot_m" + Integer.toString(i) + "      : in std_logic_vector(3 downto
0);\n" +
       "       hwdata_m" + Integer.toString(i) + "     : in std_logic_vector(31 downto
0);\n" +
       "       hbusreq_m" + Integer.toString(i) + "    : in std_logic;\n" +
       "       hlock_m" + Integer.toString(i) + "      : in std_logic;\n" +
       "       \n";
       }
      }

    // Generate ports for slaves
    for (i=0;i<numSlaves;i++)
      {
      if (!((i==0) && (defaultSlave==true)))
        {
        vhdlData +=
        "       -- Slave " + Integer.toString(i) + " ports --\n"+
        "       hsel_s" + Integer.toString(i) + "       : out std_logic;\n" +
        "       haddr_s" + Integer.toString(i) + "      : out std_logic_vector(31 downto
0);\n" +
        "       hwrite_s" + Integer.toString(i) + "     : out std_logic;\n" +
        "       hsize_s" + Integer.toString(i) + "      : out std_logic_vector(2 downto
0);\n" +
        "       htrans_s" + Integer.toString(i) + "     : out std_logic_vector(1 downto
0);\n" +
        "       hburst_s" + Integer.toString(i) + "     : out std_logic_vector(2 downto
0);\n" +
        "       hmaster_s" + Integer.toString(i) + "    : out std_logic_vector(3 downto
0);\n" +
        "       hmastlock_s" + Integer.toString(i) + " : out std_logic;\n" +
        "       hready_s" + Integer.toString(i) + "     : out std_logic;\n" +
        "       hwdata_s" + Integer.toString(i) + "     : out std_logic_vector(31 downto
0);\n" +
        "       hclk_s" + Integer.toString(i) + "       : out std_logic;\n" +
        "       hreset_s" + Integer.toString(i) + "_n   : out std_logic;\n" +
        "       hreadyout_s" + Integer.toString(i) + " : in std_logic;\n" +
        "       hresp_s" + Integer.toString(i) + "      : in std_logic_vector(1 downto
0);\n" +
        "       hsplit_s" + Integer.toString(i) + "     : in std_logic_vector(15 downto
0);\n" +
        "       hrdata_s" + Integer.toString(i) + "     : in std_logic_vector(31 downto
0);\n" +
        "       \n";
```

```
    }
  }

// Generate ports for arbiter
vhdlData +=
    "       -- Arbiter ports --\n";

for (i=0;i<numMasters;i++)
  {
  vhdlData +=
    "       hbusreq_a" + Integer.toString(i) + "    : out std_logic;\n";
  }

for (i=0;i<numMasters;i++)
  {
  vhdlData +=
    "       hlock_a" + Integer.toString(i) + "      : out std_logic;\n";
  }

vhdlData +=
    "       haddr_a       : out std_logic_vector(31 downto 0);\n" +
    "       htrans_a      : out std_logic_vector(1 downto 0);\n" +
    "       hburst_a      : out std_logic_vector(2 downto 0);\n" +
    "       hsplit_a      : out std_logic_vector(15 downto 0);\n" +
    "       hresp_a       : out std_logic_vector(1 downto 0);\n" +
    "       hready_a      : out std_logic;\n" +
    "       hreset_a_n    : out std_logic;\n" +
    "       hclk_a        : out std_logic;\n";
for (i=0;i<numMasters;i++)
  {
  vhdlData +=
    "       hgrant_a" + Integer.toString(i) + "     : in std_logic;\n";
  }
vhdlData +=
    "       hmaster_a     : in std_logic_vector(3 downto 0);\n" +
    "       hmastlock_a   : in std_logic;\n" +
    "\n";

// Generate ports for decoder
vhdlData +=
    "       haddr_d       : out std_logic_vector(31 downto 0);\n";
for (i=0;i<numSlaves;i++)
  {
  vhdlData +=
    "       hsel_d" + Integer.toString(i) + "       : in std_logic;\n";
  }
  vhdlData +=
    "\n";


// Finish entity declaration
vhdlData +=
    "       hreset_n   : in std_logic;\n" +
    "       hclk       : in std_logic\n" +
    "       );\n" +
    "end ahb_matrix_"+
    Integer.toString(numMasters)+
    "x"+
    Integer.toString(numSlaves) +";\n\n\n";
```

```
// Begin architecture implementation
vhdlData +=
    "architecture rtl of ahb_matrix_" +
    Integer.toString(numMasters) +
    "x" +
    Integer.toString(numSlaves) +
    " is\n";

if (dummyMaster==true)
    {
    vhdlData +=
    "  -- Signals for dummy master\n"+
    "  signal haddr_m0     : std_logic_vector(31 downto 0);\n" +
    "  signal hwrite_m0    : std_logic;\n" +
    "  signal hsize_m0     : std_logic_vector(2 downto 0);\n" +
    "  signal htrans_m0    : std_logic_vector(1 downto 0);\n" +
    "  signal hburst_m0    : std_logic_vector(2 downto 0);\n" +
    "  signal hprot_m0     : std_logic_vector(3 downto 0);\n" +
    "  signal hwdata_m0    : std_logic_vector(31 downto 0);\n" +
    "  signal hbusreq_m0   : std_logic;\n" +
    "  signal hlock_m0     : std_logic;\n" +
    "  -- These signals are basically sinks and are ignored\n"+
    "  signal hgrant_m0    : std_logic;\n" +
    "  signal hready_m0    : std_logic;\n" +
    "  signal hresp_m0     : std_logic_vector(1 downto 0);\n" +
    "  signal hrdata_m0    : std_logic_vector(31 downto 0);\n" +
    "  signal hclk_m0      : std_logic;\n" +
    "  signal hreset_m0_n  : std_logic;\n" +
    "\n";
    }

if (defaultSlave==true)
    {
    vhdlData +=
    "  --Signals for default slave\n"+
    "  signal hreadyout_s0 : std_logic;\n" +
    "  signal hresp_s0     : std_logic_vector(1 downto 0);\n" +
    "  signal hsplit_s0    : std_logic_vector(15 downto 0);\n" +
    "  signal hrdata_s0    : std_logic_vector(31 downto 0);\n" +
    "  -- These signals are basically sinks and are ignored\n"+
    "  signal hsel_s0      : std_logic;\n" +
    "  signal haddr_s0     : std_logic_vector(31 downto 0);\n" +
    "  signal hwrite_s0    : std_logic;\n" +
    "  signal hsize_s0     : std_logic_vector(2 downto 0);\n" +
    "  signal htrans_s0    : std_logic_vector(1 downto 0);\n" +
    "  signal hburst_s0    : std_logic_vector(2 downto 0);\n" +
    "  signal hmaster_s0   : std_logic_vector(3 downto 0);\n" +
    "  signal hmastlock_s0 : std_logic;\n" +
    "  signal hready_s0    : std_logic;\n" +
    "  signal hwdata_s0    : std_logic_vector(31 downto 0);\n" +
    "  signal hclk_s0      : std_logic;\n" +
    "  signal hreset_s0_n  : std_logic;\n" +
    "  signal ds_errwait   : std_logic;\n"+
    "  \n";
    }
```

```
// AHB Signal definition
vhdlData +=
    "  signal haddr     : std_logic_vector(31 downto 0);\n" +
    "  signal hwrite    : std_logic;\n" +
    "  signal hsize     : std_logic_vector(2 downto 0);\n" +
    "  signal hburst    : std_logic_vector(2 downto 0);\n" +
    "  signal htrans    : std_logic_vector(1 downto 0);\n" +
    "  signal hwdata    : std_logic_vector(31 downto 0);\n" +
    "  signal hrdata    : std_logic_vector(31 downto 0);\n" +
    "  signal hresp     : std_logic_vector(1 downto 0);\n" +
    "  signal hready    : std_logic;\n"+
    "  signal hsplit    : std_logic_vector(15 downto 0);\n"+
    "  signal hmaster   : std_logic_vector(3 downto 0);\n"+
    "  signal hmastlock : std_logic;\n";

// Other signals
vhdlData +=
    "  signal hmaster_del : std_logic_vector(3 downto 0);\n";

for (i=0;i<numSlaves;i++)
  {
  vhdlData +=
    "  signal hsel_del_d" + Integer.toString(i) + " : std_logic;\n";
  }


vhdlData +=
    "begin\n";

// Dummy master signals
if (dummyMaster == true)
  {
  vhdlData +=
    "  -- Generate signals from dummy master\n"+
    "  haddr_m0   <= (others=>'0');\n" +
    "  hwrite_m0  <= '0';\n" +
    "  hsize_m0   <= \"010\";\n" +
    "  htrans_m0  <= \"00\";\n" +
    "  hburst_m0  <= \"000\";\n" +
    "  hprot_m0   <= \"0011\";\n" +
    "  hwdata_m0  <= (others =>'0');\n" +
    "  hbusreq_m0 <= '0';\n" +
    "  hlock_m0   <= '0';\n" +
    "\n";
  }
```

```
// Default slave state machine & signals
if (defaultSlave == true)
  {
  vhdlData +=
    "  -- Generate signals from default slave\n"+
    "  defaultslave:process(hclk_s0, hreset_s0_n)\n"+
    "  begin\n"+
    "    if hreset_s0_n = '0' then\n"+
    "      ds_errwait   <= '0';\n"+
    "      hreadyout_s0 <= '1';\n"+
    "      hresp_s0     <= \"00\"; -- OK\n"+
    "    elsif hclk_s0'event and hclk_s0 = '1' then\n"+
    "      if ds_errwait = '1' then\n"+
    "        ds_errwait   <= '0';\n"+
    "        hreadyout_s0 <= '1';\n"+
    "        hresp_s0     <= \"01\"; -- ERROR\n"+
    "      elsif hsel_s0 = '1' and htrans_s0(1) = '1' and\n"+
    "            hready_s0 = '1' then\n"+
    "        ds_errwait   <= '1';\n"+
    "        hreadyout_s0 <= '0';\n"+
    "        hresp_s0     <= \"01\"; -- ERROR\n"+
    "      else\n"+
    "        ds_errwait   <= '0';\n"+
    "        hreadyout_s0 <= '1';\n"+
    "        hresp_s0     <= \"00\"; -- OK\n"+
    "      end if;\n"+
    "    end if;\n"+
    "  end process;\n"+
    "\n"+
    "  hsplit_s0 <= (others=>'0');\n" +
    "  hrdata_s0 <= (others=>'0');\n" +
    "\n";
  }
```

```
    // Basic signal routing
    for (i=0;i<numMasters;i++)
      {
      vhdlData +=
        "  -- Route signals to Master " + Integer.toString(i) + " --\n" +
        "  hgrant_m" + Integer.toString(i) + "    <= hgrant_a" + Integer.toString(i) +
";\n" +
        "  hready_m" + Integer.toString(i) + "    <= hready;\n" +
        "  hresp_m" + Integer.toString(i) + "     <= hresp;\n" +
        "  hrdata_m" + Integer.toString(i) + "    <= hrdata;\n" +
        "  hreset_m" + Integer.toString(i) + "_n  <= hreset_n;\n" +
        "  hclk_m" + Integer.toString(i) + "      <= hclk;\n" +
        "  hbusreq_a" + Integer.toString(i) + "   <= hbusreq_m" + Integer.toString(i) +
";\n" +
        "  hlock_a" + Integer.toString(i) + "     <= hlock_m" + Integer.toString(i) +
";\n" +
        "  -- (the other signals are routed by the master mux) --\n" +
        "\n";
      }

    for (i=0;i<numSlaves;i++)
      {
      vhdlData +=
        "  -- Route signals to Slave " + Integer.toString(i) + " --\n" +
        "  hsel_s" + Integer.toString(i) + "      <= hsel_d" + Integer.toString(i) +
";\n" +
        "  haddr_s" + Integer.toString(i) + "     <= haddr;\n" +
        "  hwrite_s" + Integer.toString(i) + "    <= hwrite;\n" +
        "  htrans_s" + Integer.toString(i) + "    <= htrans;\n" +
        "  hsize_s" + Integer.toString(i) + "     <= hsize;\n" +
        "  hburst_s" + Integer.toString(i) + "    <= hburst;\n" +
        "  hwdata_s" + Integer.toString(i) + "    <= hwdata;\n" +
        "  hmaster_s" + Integer.toString(i) + "   <= hmaster;\n" +
        "  hmastlock_s" + Integer.toString(i) + " <= hmastlock;\n" +
        "  hready_s" + Integer.toString(i) + "    <= hready;\n" +
        "  hreset_s" + Integer.toString(i) + "_n  <= hreset_n;\n" +
        "  hclk_s" + Integer.toString(i) + "      <= hclk;\n" +
        "  -- (the other signals are routed by the slave mux) --\n" +
        "\n";
      }

    vhdlData +=
        "  -- Route remaining Arbiter and Decoder signals --\n" +
        "  haddr_d    <= haddr;\n" +
        "  haddr_a    <= haddr;\n" +
        "  htrans_a   <= htrans;\n" +
        "  hburst_a   <= hburst;\n" +
        "  hresp_a    <= hresp;\n" +
        "  hready_a   <= hready;\n" +
        "  hreset_a_n <= hreset_n;\n" +
        "  hsplit_a   <= hsplit;\n" +
        "  hclk_a     <= hclk;\n" +
        "  hmaster    <= hmaster_a;\n" +
        "  hmastlock  <= hmastlock_a;\n" +
        "\n";

    // OR all of the HSPLIT signals together (see AMBA 2.0 doc, p. 3-35)
    vhdlData +=
        "  -- Combine HSPLIT signals --\n" +
        "  hsplit <= \"0000000000000000\"";

    for (i=0;i<numSlaves;i++)
      {
      vhdlData +=
        " or hsplit_s" + Integer.toString(i);
      }
    vhdlData +=         ";\n\n";
```

```
// Generate delayed HMASTER
vhdlData +=
    "  -- Generate delayed HMASTER\n" +
    "  delay_hmaster: process(hclk, hreset_n)\n" +
    "  begin\n" +
    "    if hreset_n = '0' then\n" +
    "      hmaster_del <= \"0000\";\n" +
    "    elsif hclk'event and hclk = '1' then\n" +
    "      hmaster_del <= hmaster;\n" +
    "    end if;\n" +
    "  end process;\n\n";

// Generate delayed HSEL
vhdlData +=
    "  -- Generate delayed HSEL\n" +
    "  delay_hsel: process(hclk, hreset_n)\n"+
    "  begin\n" +
    "    if hreset_n = '0' then\n"+
    "      hsel_del_d0 <= '1';\n";

// Reset all delayed signals
for (i=1;i<numSlaves;i++)
  {
  vhdlData +=
    "      hsel_del_d" + Integer.toString(i) + " <= '0';\n";
  }

// Delay hsel
vhdlData +=
    "    elsif hclk'event and hclk = '1' then\n"+
    "      case hready is\n"+
    "        when '1' =>\n";


for (i=0;i<numSlaves;i++)
  {
  vhdlData +=
    "          hsel_del_d"+Integer.toString(i)+
          " <= hsel_d"+ Integer.toString(i) +";\n";
  }

vhdlData +=
    "        when others=>\n"+
    "          null;\n";

vhdlData +=
    "      end case;\n" +
    "    end if;\n" +
    "  end process;\n\n";
```

```
// Generate Master Control MUX
vhdlData +=
    "  -- Select proper inputs for master mux --\n" +
    "  mastermux:process(hmaster, hmaster_del";
for (i=0;i<numMasters;i++)
  {
  vhdlData +=
    ",\n                    haddr_m" + Integer.toString(i) +
    ", hwrite_m" + Integer.toString(i) +
    ", hsize_m" + Integer.toString(i) +
    ", hburst_m" + Integer.toString(i) +
    ", htrans_m" + Integer.toString(i) +
    ", hwdata_m" + Integer.toString(i);
  }

vhdlData +=
    "  )\n" +
    "  begin\n" +
    "    -- set default values\n" +
    "    haddr  <= (others => '0');\n" +
    "    hwrite <= '0';\n" +
    "    hsize  <= \"010\";\n" +
    "    hburst <= \"000\";\n" +
    "    htrans <= \"00\";\n" +
    "    hwdata <= (others => '0');\n" +
    "\n";

// (route countrol signals with HMASTER)
vhdlData +=
    "    case hmaster is\n";

for (i=0;i<numMasters;i++)
  {
  currentModule = Integer.toString(i, 2);
  vhdlData +=
    "      when \"" +
    allZeros.substring(0, 4-currentModule.length()) +
    currentModule + "\" =>\n" +
    "        haddr  <= haddr_m" + Integer.toString(i) + ";\n" +
    "        hwrite <= hwrite_m" + Integer.toString(i) + ";\n" +
    "        hsize  <= hsize_m" + Integer.toString(i) + ";\n" +
    "        hburst <= hburst_m" + Integer.toString(i) + ";\n" +
    "        htrans <= htrans_m" + Integer.toString(i) + ";\n" +
    "\n";
  }

vhdlData +=
    "      when others =>\n" +
    "        haddr  <= haddr_m0;\n" +
    "        hwrite <= hwrite_m0;\n" +
    "        hsize  <= hsize_m0;\n" +
    "        hburst <= hburst_m0;\n" +
    "        htrans <= htrans_m0;\n" +
    "    end case;\n";
```

```
// (route countrol signals with HMASTER_DEL1
vhdlData +=
    "    case hmaster_del is\n";

for (i=0;i<numMasters;i++)
  {
  currentModule = Integer.toString(i, 2);
  vhdlData +=
    "      when \"" +
    allZeros.substring(0, 4-currentModule.length()) +
    currentModule + "\" =>\n" +
    "        hwdata <= hwdata_m" + Integer.toString(i) + ";\n" +
    "\n";
  }

vhdlData +=
    "      when others =>\n" +
    "        null;\n" +
    "      end case;\n" +
    "  end process;\n";

// Generate Slave MUX
vhdlData +=
    "\n" +
    "  -- Select proper inputs for slave mux --\n" +
    "  slavemux:process(\n                    ";
// (build senitivity list -- crazy, huh?)
for (i=0;i<numSlaves;i++)
  {
  vhdlData +=
    "hsel_del_d" + Integer.toString(i) +
    ", hreadyout_s" + Integer.toString(i) +
    ", hresp_s" + Integer.toString(i) +
    ", hrdata_s" + Integer.toString(i);

  if (i != (numSlaves-1))
    {
    vhdlData +=
      ",\n                    ";
    }
  else
    {
    vhdlData +=
      ")\n";
    }
  }
```

```
  // (whew! on to the default values...)
  vhdlData +=
      "  begin\n" ;

  // (if a slave is selected, route its signals)
  for (i=0;i<numSlaves;i++)
    {
    if (i != 0)
      {
      vhdlData +="    els";
      }
    else
      {
      vhdlData +="    ";
      }

    vhdlData +=
      "if hsel_del_d" + Integer.toString(i) + " = '1' then\n"+
      "      hready <= hreadyout_s" + Integer.toString(i) + ";\n" +
      "      hresp  <= hresp_s" + Integer.toString(i) + ";\n" +
      "      hrdata <= hrdata_s" + Integer.toString(i) + ";\n" +
      "\n";
    }


   vhdlData +=
      "    else\n" +
      "      hready <= hreadyout_s0;\n"+
      "      hresp <= hresp_s0;\n"+
      "      hrdata <= hrdata_s0;\n"+
      "    end if;\n" +
      "  end process;\n";

  vhdlData +=
    "end rtl;\n";



  return vhdlData;
  }

}
```

Annexe B :
Exemple de code pour le réseau d'interconnexion

Cette annexe contient un exemple de réseau d'interconnexion généré par le programme de l'annexe A. Il contient deux maîtres, dont un factice, et six esclaves, dont un par défaut. C'est le réseau utilisé dans le bloc-processeur (voir section.3.2.1).

```
library IEEE;
use IEEE.std_logic_1164.all;

entity ahb_matrix_2x6 is
  port (
        -- Master 1 ports --
        hgrant_m1     : out std_logic;
        hready_m1     : out std_logic;
        hresp_m1      : out std_logic_vector(1 downto 0);
        hrdata_m1     : out std_logic_vector(31 downto 0);
        hclk_m1       : out std_logic;
        hreset_m1_n   : out std_logic;
        haddr_m1      : in std_logic_vector(31 downto 0);
        hwrite_m1     : in std_logic;
        hsize_m1      : in std_logic_vector(2 downto 0);
        htrans_m1     : in std_logic_vector(1 downto 0);
        hburst_m1     : in std_logic_vector(2 downto 0);
        hprot_m1      : in std_logic_vector(3 downto 0);
        hwdata_m1     : in std_logic_vector(31 downto 0);
        hbusreq_m1    : in std_logic;
        hlock_m1      : in std_logic;

        -- Slave 1 ports --
        hsel_s1       : out std_logic;
        haddr_s1      : out std_logic_vector(31 downto 0);
        hwrite_s1     : out std_logic;
        hsize_s1      : out std_logic_vector(2 downto 0);
        htrans_s1     : out std_logic_vector(1 downto 0);
        hburst_s1     : out std_logic_vector(2 downto 0);
        hmaster_s1    : out std_logic_vector(3 downto 0);
        hmastlock_s1  : out std_logic;
        hready_s1     : out std_logic;
        hwdata_s1     : out std_logic_vector(31 downto 0);
        hclk_s1       : out std_logic;
        hreset_s1_n   : out std_logic;
        hreadyout_s1  : in std_logic;
        hresp_s1      : in std_logic_vector(1 downto 0);
        hsplit_s1     : in std_logic_vector(15 downto 0);
        hrdata_s1     : in std_logic_vector(31 downto 0);
```

```
-- Slave 2 ports --
hsel_s2       : out std_logic;
haddr_s2      : out std_logic_vector(31 downto 0);
hwrite_s2     : out std_logic;
hsize_s2      : out std_logic_vector(2 downto 0);
htrans_s2     : out std_logic_vector(1 downto 0);
hburst_s2     : out std_logic_vector(2 downto 0);
hmaster_s2    : out std_logic_vector(3 downto 0);
hmastlock_s2  : out std_logic;
hready_s2     : out std_logic;
hwdata_s2     : out std_logic_vector(31 downto 0);
hclk_s2       : out std_logic;
hreset_s2_n   : out std_logic;
hreadyout_s2  : in std_logic;
hresp_s2      : in std_logic_vector(1 downto 0);
hsplit_s2     : in std_logic_vector(15 downto 0);
hrdata_s2     : in std_logic_vector(31 downto 0);

-- Slave 3 ports --
hsel_s3       : out std_logic;
haddr_s3      : out std_logic_vector(31 downto 0);
hwrite_s3     : out std_logic;
hsize_s3      : out std_logic_vector(2 downto 0);
htrans_s3     : out std_logic_vector(1 downto 0);
hburst_s3     : out std_logic_vector(2 downto 0);
hmaster_s3    : out std_logic_vector(3 downto 0);
hmastlock_s3  : out std_logic;
hready_s3     : out std_logic;
hwdata_s3     : out std_logic_vector(31 downto 0);
hclk_s3       : out std_logic;
hreset_s3_n   : out std_logic;
hreadyout_s3  : in std_logic;
hresp_s3      : in std_logic_vector(1 downto 0);
hsplit_s3     : in std_logic_vector(15 downto 0);
hrdata_s3     : in std_logic_vector(31 downto 0);

-- Slave 4 ports --
hsel_s4       : out std_logic;
haddr_s4      : out std_logic_vector(31 downto 0);
hwrite_s4     : out std_logic;
hsize_s4      : out std_logic_vector(2 downto 0);
htrans_s4     : out std_logic_vector(1 downto 0);
hburst_s4     : out std_logic_vector(2 downto 0);
hmaster_s4    : out std_logic_vector(3 downto 0);
hmastlock_s4  : out std_logic;
hready_s4     : out std_logic;
hwdata_s4     : out std_logic_vector(31 downto 0);
hclk_s4       : out std_logic;
hreset_s4_n   : out std_logic;
hreadyout_s4  : in std_logic;
hresp_s4      : in std_logic_vector(1 downto 0);
hsplit_s4     : in std_logic_vector(15 downto 0);
hrdata_s4     : in std_logic_vector(31 downto 0);
```

```
        -- Slave 5 ports --
        hsel_s5       : out std_logic;
        haddr_s5      : out std_logic_vector(31 downto 0);
        hwrite_s5     : out std_logic;
        hsize_s5      : out std_logic_vector(2 downto 0);
        htrans_s5     : out std_logic_vector(1 downto 0);
        hburst_s5     : out std_logic_vector(2 downto 0);
        hmaster_s5    : out std_logic_vector(3 downto 0);
        hmastlock_s5  : out std_logic;
        hready_s5     : out std_logic;
        hwdata_s5     : out std_logic_vector(31 downto 0);
        hclk_s5       : out std_logic;
        hreset_s5_n   : out std_logic;
        hreadyout_s5  : in std_logic;
        hresp_s5      : in std_logic_vector(1 downto 0);
        hsplit_s5     : in std_logic_vector(15 downto 0);
        hrdata_s5     : in std_logic_vector(31 downto 0);

        -- Arbiter ports --
        hbusreq_a0    : out std_logic;
        hbusreq_a1    : out std_logic;
        hlock_a0      : out std_logic;
        hlock_a1      : out std_logic;
        haddr_a       : out std_logic_vector(31 downto 0);
        htrans_a      : out std_logic_vector(1 downto 0);
        hburst_a      : out std_logic_vector(2 downto 0);
        hsplit_a      : out std_logic_vector(15 downto 0);
        hresp_a       : out std_logic_vector(1 downto 0);
        hready_a      : out std_logic;
        hreset_a_n    : out std_logic;
        hclk_a        : out std_logic;
        hgrant_a0     : in std_logic;
        hgrant_a1     : in std_logic;
        hmaster_a     : in std_logic_vector(3 downto 0);
        hmastlock_a   : in std_logic;

        haddr_d       : out std_logic_vector(31 downto 0);
        hsel_d0       : in std_logic;
        hsel_d1       : in std_logic;
        hsel_d2       : in std_logic;
        hsel_d3       : in std_logic;
        hsel_d4       : in std_logic;
        hsel_d5       : in std_logic;

        hreset_n   : in std_logic;
        hclk       : in std_logic
        );
end ahb_matrix_2x6;
```

```
architecture rtl of ahb_matrix_2x6 is
  -- Signals for dummy master
  signal haddr_m0     : std_logic_vector(31 downto 0);
  signal hwrite_m0    : std_logic;
  signal hsize_m0     : std_logic_vector(2 downto 0);
  signal htrans_m0    : std_logic_vector(1 downto 0);
  signal hburst_m0    : std_logic_vector(2 downto 0);
  signal hprot_m0     : std_logic_vector(3 downto 0);
  signal hwdata_m0    : std_logic_vector(31 downto 0);
  signal hbusreq_m0   : std_logic;
  signal hlock_m0     : std_logic;
  -- These signals are basically sinks and are ignored
  signal hgrant_m0    : std_logic;
  signal hready_m0    : std_logic;
  signal hresp_m0     : std_logic_vector(1 downto 0);
  signal hrdata_m0    : std_logic_vector(31 downto 0);
  signal hclk_m0      : std_logic;
  signal hreset_m0_n  : std_logic;

  --Signals for default slave
  signal hreadyout_s0 : std_logic;
  signal hresp_s0     : std_logic_vector(1 downto 0);
  signal hsplit_s0    : std_logic_vector(15 downto 0);
  signal hrdata_s0    : std_logic_vector(31 downto 0);
  -- These signals are basically sinks and are ignored
  signal hsel_s0      : std_logic;
  signal haddr_s0     : std_logic_vector(31 downto 0);
  signal hwrite_s0    : std_logic;
  signal hsize_s0     : std_logic_vector(2 downto 0);
  signal htrans_s0    : std_logic_vector(1 downto 0);
  signal hburst_s0    : std_logic_vector(2 downto 0);
  signal hmaster_s0   : std_logic_vector(3 downto 0);
  signal hmastlock_s0 : std_logic;
  signal hready_s0    : std_logic;
  signal hwdata_s0    : std_logic_vector(31 downto 0);
  signal hclk_s0      : std_logic;
  signal hreset_s0_n  : std_logic;
  signal ds_errwait   : std_logic;

  signal haddr     : std_logic_vector(31 downto 0);
  signal hwrite    : std_logic;
  signal hsize     : std_logic_vector(2 downto 0);
  signal hburst    : std_logic_vector(2 downto 0);
  signal htrans    : std_logic_vector(1 downto 0);
  signal hwdata    : std_logic_vector(31 downto 0);
  signal hrdata    : std_logic_vector(31 downto 0);
  signal hresp     : std_logic_vector(1 downto 0);
  signal hready    : std_logic;
  signal hsplit    : std_logic_vector(15 downto 0);
  signal hmaster   : std_logic_vector(3 downto 0);
  signal hmastlock : std_logic;
  signal hmaster_del : std_logic_vector(3 downto 0);
  signal hsel_del_d0 : std_logic;
  signal hsel_del_d1 : std_logic;
```

```
  signal hsel_del_d2 : std_logic;
  signal hsel_del_d3 : std_logic;
  signal hsel_del_d4 : std_logic;
  signal hsel_del_d5 : std_logic;

begin
  -- Generate signals from dummy master
  haddr_m0   <= (others=>'0');
  hwrite_m0  <= '0';
  hsize_m0   <= "010";
  htrans_m0  <= "00";
  hburst_m0  <= "000";
  hprot_m0   <= "0011";
  hwdata_m0  <= (others =>'0');
  hbusreq_m0 <= '0';
  hlock_m0   <= '0';

  -- Generate signals from default slave
  defaultslave:process(hclk_s0, hreset_s0_n)
  begin
    if hreset_s0_n = '0' then
      ds_errwait   <= '0';
      hreadyout_s0 <= '1';
      hresp_s0     <= "00"; -- OK
    elsif hclk_s0'event and hclk_s0 = '1' then
      if ds_errwait = '1' then
        ds_errwait   <= '0';
        hreadyout_s0 <= '1';
        hresp_s0     <= "01"; -- ERROR
      elsif hsel_s0 = '1' and htrans_s0(1) = '1' and
            hready_s0 = '1' then
        ds_errwait   <= '1';
        hreadyout_s0 <= '0';
        hresp_s0     <= "01"; -- ERROR
      else
        ds_errwait   <= '0';
        hreadyout_s0 <= '1';
        hresp_s0     <= "00"; -- OK
      end if;
    end if;
  end process;

  hsplit_s0 <= (others=>'0');
  hrdata_s0 <= (others=>'0');
```

```
-- Route signals to Master 0 --
hgrant_m0     <= hgrant_a0;
hready_m0     <= hready;
hresp_m0      <= hresp;
hrdata_m0     <= hrdata;
hreset_m0_n   <= hreset_n;
hclk_m0       <= hclk;
hbusreq_a0    <= hbusreq_m0;
hlock_a0      <= hlock_m0;
-- (the other signals are routed by the master mux) --

-- Route signals to Master 1 --
hgrant_m1     <= hgrant_a1;
hready_m1     <= hready;
hresp_m1      <= hresp;
hrdata_m1     <= hrdata;
hreset_m1_n   <= hreset_n;
hclk_m1       <= hclk;
hbusreq_a1    <= hbusreq_m1;
hlock_a1      <= hlock_m1;
-- (the other signals are routed by the master mux) --

-- Route signals to Slave 0 --
hsel_s0       <= hsel_d0;
haddr_s0      <= haddr;
hwrite_s0     <= hwrite;
htrans_s0     <= htrans;
hsize_s0      <= hsize;
hburst_s0     <= hburst;
hwdata_s0     <= hwdata;
hmaster_s0    <= hmaster;
hmastlock_s0  <= hmastlock;
hready_s0     <= hready;
hreset_s0_n   <= hreset_n;
hclk_s0       <= hclk;
-- (the other signals are routed by the slave mux) --

-- Route signals to Slave 1 --
hsel_s1       <= hsel_d1;
haddr_s1      <= haddr;
hwrite_s1     <= hwrite;
htrans_s1     <= htrans;
hsize_s1      <= hsize;
hburst_s1     <= hburst;
hwdata_s1     <= hwdata;
hmaster_s1    <= hmaster;
hmastlock_s1  <= hmastlock;
hready_s1     <= hready;
hreset_s1_n   <= hreset_n;
hclk_s1       <= hclk;
-- (the other signals are routed by the slave mux) --
```

```
-- Route signals to Slave 2 --
hsel_s2      <= hsel_d2;
haddr_s2     <= haddr;
hwrite_s2    <= hwrite;
htrans_s2    <= htrans;
hsize_s2     <= hsize;
hburst_s2    <= hburst;
hwdata_s2    <= hwdata;
hmaster_s2   <= hmaster;
hmastlock_s2 <= hmastlock;
hready_s2    <= hready;
hreset_s2_n  <= hreset_n;
hclk_s2      <= hclk;
-- (the other signals are routed by the slave mux) --

-- Route signals to Slave 3 --
hsel_s3      <= hsel_d3;
haddr_s3     <= haddr;
hwrite_s3    <= hwrite;
htrans_s3    <= htrans;
hsize_s3     <= hsize;
hburst_s3    <= hburst;
hwdata_s3    <= hwdata;
hmaster_s3   <= hmaster;
hmastlock_s3 <= hmastlock;
hready_s3    <= hready;
hreset_s3_n  <= hreset_n;
hclk_s3      <= hclk;
-- (the other signals are routed by the slave mux) --

-- Route signals to Slave 4 --
hsel_s4      <= hsel_d4;
haddr_s4     <= haddr;
hwrite_s4    <= hwrite;
htrans_s4    <= htrans;
hsize_s4     <= hsize;
hburst_s4    <= hburst;
hwdata_s4    <= hwdata;
hmaster_s4   <= hmaster;
hmastlock_s4 <= hmastlock;
hready_s4    <= hready;
hreset_s4_n  <= hreset_n;
hclk_s4      <= hclk;
-- (the other signals are routed by the slave mux) --
```

```
  -- Route signals to Slave 5 --
  hsel_s5      <= hsel_d5;
  haddr_s5     <= haddr;
  hwrite_s5    <= hwrite;
  htrans_s5    <= htrans;
  hsize_s5     <= hsize;
  hburst_s5    <= hburst;
  hwdata_s5    <= hwdata;
  hmaster_s5   <= hmaster;
  hmastlock_s5 <= hmastlock;
  hready_s5    <= hready;
  hreset_s5_n  <= hreset_n;
  hclk_s5      <= hclk;
  -- (the other signals are routed by the slave mux) --

  -- Route remaining Arbiter and Decoder signals --
  haddr_d    <= haddr;
  haddr_a    <= haddr;
  htrans_a   <= htrans;
  hburst_a   <= hburst;
  hresp_a    <= hresp;
  hready_a   <= hready;
  hreset_a_n <= hreset_n;
  hsplit_a   <= hsplit;
  hclk_a     <= hclk;
  hmaster    <= hmaster_a;
  hmastlock  <= hmastlock_a;

  -- Combine HSPLIT signals --
  hsplit <= "0000000000000000" or hsplit_s0 or hsplit_s1 or hsplit_s2
or hsplit_s3 or hsplit_s4 or hsplit_s5;

  -- Generate delayed HMASTER
  delay_hmaster: process(hclk, hreset_n)
  begin
    if hreset_n = '0' then
      hmaster_del <= "0000";
    elsif hclk'event and hclk = '1' then
      hmaster_del <= hmaster;
    end if;
  end process;
```

```
-- Generate delayed HSEL
delay_hsel: process(hclk, hreset_n)
begin
  if hreset_n = '0' then
    hsel_del_d0 <= '1';
    hsel_del_d1 <= '0';
    hsel_del_d2 <= '0';
    hsel_del_d3 <= '0';
    hsel_del_d4 <= '0';
    hsel_del_d5 <= '0';
  elsif hclk'event and hclk = '1' then
    case hready is
      when '1' =>
        hsel_del_d0 <= hsel_d0;
        hsel_del_d1 <= hsel_d1;
        hsel_del_d2 <= hsel_d2;
        hsel_del_d3 <= hsel_d3;
        hsel_del_d4 <= hsel_d4;
        hsel_del_d5 <= hsel_d5;
      when others=>
        null;
    end case;
  end if;
end process;
```

```
-- Select proper inputs for master mux --
mastermux:process(hmaster, hmaster_del,
                  haddr_m0, hwrite_m0, hsize_m0, hburst_m0,
                  htrans_m0, hwdata_m0,
                  haddr_m1, hwrite_m1, hsize_m1, hburst_m1,
                  htrans_m1, hwdata_m1  )
begin
  -- set default values
  haddr  <= (others => '0');
  hwrite <= '0';
  hsize  <= "010";
  hburst <= "000";
  htrans <= "00";
  hwdata <= (others => '0');

  case hmaster is
    when "0000" =>
      haddr  <= haddr_m0;
      hwrite <= hwrite_m0;
      hsize  <= hsize_m0;
      hburst <= hburst_m0;
      htrans <= htrans_m0;

    when "0001" =>
      haddr  <= haddr_m1;
      hwrite <= hwrite_m1;
      hsize  <= hsize_m1;
      hburst <= hburst_m1;
      htrans <= htrans_m1;

    when others =>
      haddr  <= haddr_m0;
      hwrite <= hwrite_m0;
      hsize  <= hsize_m0;
      hburst <= hburst_m0;
      htrans <= htrans_m0;
  end case;
  case hmaster_del is
    when "0000" =>
      hwdata <= hwdata_m0;

    when "0001" =>
      hwdata <= hwdata_m1;

    when others =>
      null;
    end case;
end process;
```

```
  -- Select proper inputs for slave mux --
  slavemux:process(
                   hsel_del_d0, hreadyout_s0, hresp_s0, hrdata_s0,
                   hsel_del_d1, hreadyout_s1, hresp_s1, hrdata_s1,
                   hsel_del_d2, hreadyout_s2, hresp_s2, hrdata_s2,
                   hsel_del_d3, hreadyout_s3, hresp_s3, hrdata_s3,
                   hsel_del_d4, hreadyout_s4, hresp_s4, hrdata_s4,
                   hsel_del_d5, hreadyout_s5, hresp_s5, hrdata_s5)
  begin
    if hsel_del_d0 = '1' then
      hready <= hreadyout_s0;
      hresp  <= hresp_s0;
      hrdata <= hrdata_s0;

    elsif hsel_del_d1 = '1' then
      hready <= hreadyout_s1;
      hresp  <= hresp_s1;
      hrdata <= hrdata_s1;

    elsif hsel_del_d2 = '1' then
      hready <= hreadyout_s2;
      hresp  <= hresp_s2;
      hrdata <= hrdata_s2;

    elsif hsel_del_d3 = '1' then
      hready <= hreadyout_s3;
      hresp  <= hresp_s3;
      hrdata <= hrdata_s3;

    elsif hsel_del_d4 = '1' then
      hready <= hreadyout_s4;
      hresp  <= hresp_s4;
      hrdata <= hrdata_s4;

    elsif hsel_del_d5 = '1' then
      hready <= hreadyout_s5;
      hresp  <= hresp_s5;
      hrdata <= hrdata_s5;

    else
      hready <= hreadyout_s0;
      hresp <= hresp_s0;
      hrdata <= hrdata_s0;
    end if;
  end process;
end rtl;
```

Annexe C : Code du pont AHB-AHB

Cette section contient le code du pont AHB-AHB. Comme on peut le constater, ce code est très complexe et il est mieux d'observer le diagramme d'état à l'aide de l'outil de conception HDL Designer Pro qui a servi à le produire.

```
-- renoir header_start


--
-- VHDL Entity interconnect.bridge_ahb_ahb.symbol
--
-- Created:
--          by - bertola.etudiant (snowdon)
--          at - 10:04:30 02/11/03
--
-- Generated by Mentor Graphics' Renoir(TM) 2000.4 (Build 5)
--
-- renoir header_end
LIBRARY ieee ;
USE ieee.std_logic_1164.all;

ENTITY bridge_ahb_ahb IS
   PORT(
      haddr_s      : IN     std_logic_vector (31 DOWNTO 0) ;
      hburst_s     : IN     std_logic_vector (2 DOWNTO 0) ;
      hclk_m       : IN     std_logic  ;
      hclk_s       : IN     std_logic  ;
      hgrant_m     : IN     std_logic  ;
      hmaster_s    : IN     std_logic_vector (3 DOWNTO 0) ;
      hmastlock_s  : IN     std_logic  ;
      hrdata_m     : IN     std_logic_vector (31 DOWNTO 0) ;
      hready_m     : IN     std_logic  ;
      hready_s     : IN     std_logic  ;
      hreset_m     : IN     std_logic  ;
      hreset_s     : IN     std_logic  ;
      hresp_m      : IN     std_logic_vector (1 DOWNTO 0) ;
      hsel_s       : IN     std_logic  ;
      hsize_s      : IN     std_logic_vector (2 DOWNTO 0) ;
      htrans_s     : IN     std_logic_vector (1 DOWNTO 0) ;
      hwdata_s     : IN     std_logic_vector (31 DOWNTO 0) ;
      hwrite_s     : IN     std_logic  ;
      haddr_m      : OUT    std_logic_vector (31 DOWNTO 0) ;
      hburst_m     : OUT    std_logic_vector (2 DOWNTO 0) ;
      hbusreq_m    : OUT    std_logic  ;
      hlock_m      : OUT    std_logic  ;
      hprot_m      : OUT    std_logic_vector (3 DOWNTO 0) ;
      hrdata_s     : OUT    std_logic_vector (31 DOWNTO 0) ;
      hreadyout_s  : OUT    std_logic  ;
      hresp_s      : OUT    std_logic_vector (1 DOWNTO 0) ;
      hsize_m      : OUT    std_logic_vector (2 DOWNTO 0) ;
      hsplit_s     : OUT    std_logic_vector (15 DOWNTO 0) ;
      htrans_m     : OUT    std_logic_vector (1 DOWNTO 0) ;
      hwdata_m     : OUT    std_logic_vector (31 DOWNTO 0) ;
      hwrite_m     : OUT    std_logic
   );
```

```
-- Declarations

END bridge_ahb_ahb ;

-- renoir interface_end
--
-- VHDL Architecture interconnect.bridge_ahb_ahb.fsm
--
-- Created:
--          by - bertola.etudiant (snowdon)
--          at - 10:04:31 02/11/03
--
-- Generated by Mentor Graphics' Renoir(TM) 2000.4 (Build 5)
--
LIBRARY ieee ;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
USE ieee.std_logic_arith.ALL;

ARCHITECTURE fsm OF bridge_ahb_ahb IS

   -- Architecture Declarations
   signal hsplit_int : std_logic_vector(15 downto 0);
   signal hmaster_int : std_logic_vector(3 downto 0);
   signal hrdata_int : std_logic_vector(31 downto 0);

   TYPE STATE_TYPE IS (
      idle,
      hold,
      run,
      run_last,
      run_first,
      split_hold,
      split_flush,
      split_wake,
      split_retry1,
      split_retry2
   );

   -- State vector declaration
   ATTRIBUTE state_vector : string;
   ATTRIBUTE state_vector OF fsm : ARCHITECTURE IS "current_state" ;


   -- Declare current and next state signals
   SIGNAL current_state : STATE_TYPE ;
   SIGNAL next_state : STATE_TYPE ;

BEGIN
```

```
-------------------------------------------------------------------
clocked : PROCESS(
      hclk_s,
      hreset_s
   )
-------------------------------------------------------------------
BEGIN
      IF (hreset_s = '0') THEN
         current_state <= idle;
         -- Reset Values
         hmaster_int <= (others => '0');
         hrdata_int <= (others => '0');
      ELSIF (hclk_s'EVENT AND hclk_s = '1') THEN
         current_state <= next_state;
         -- Default Assignment To Internals

         -- Transition Actions for internal signals only
         CASE current_state IS
         WHEN run =>
            IF (hresp_m ="11") THEN
               hmaster_int <= hmaster_s;
            ELSIF (hgrant_m = '0'
            and hready_m = '1') THEN
            END IF;
         WHEN run_last =>
            IF (hgrant_m = '1'
            and hready_m = '1') THEN
            ELSE
               hrdata_int <= hrdata_m;
            END IF;
         WHEN OTHERS =>
            NULL;
         END CASE;

      END IF;

   END PROCESS clocked;
```

```
-------------------------------------------------------------------------
nextstate : PROCESS (
      current_state,
      hgrant_m,
      hmaster_int,
      hmaster_s,
      hready_m,
      hready_s,
      hresp_m,
      hsel_s
   )
-------------------------------------------------------------------------
BEGIN
      CASE current_state IS
      WHEN idle =>
         IF (hsel_s = '1'
         and hready_m = '1'
         and hgrant_m = '1') THEN
            next_state <= run;
         ELSE
            next_state <= idle;
         END IF;
      WHEN hold =>
         IF (hgrant_m  = '1'
         and hready_m = '1') THEN
            next_state <= run_first;
         ELSE
            next_state <= hold;
         END IF;
      WHEN run =>
         IF (hresp_m ="11") THEN
            next_state <= split_flush;
         ELSIF (hgrant_m = '0'
         and hready_m = '1') THEN
            next_state <= run_last;
         ELSIF (hsel_s = '0'
         and hready_s = '1') THEN
            next_state <= idle;
         ELSE
            next_state <= run;
         END IF;
      WHEN run_last =>
         IF (hgrant_m = '1'
         and hready_m = '1') THEN
            next_state <= run_first;
         ELSE
            next_state <= hold;
         END IF;
      WHEN run_first =>
         IF (hgrant_m = '0'
         and hready_m = '1') THEN
            next_state <= hold;
         ELSE
            next_state <= run;
         END IF;
      WHEN split_hold =>
         IF (hsel_s = '1' and
         not hmaster_s = hmaster_int) THEN
            next_state <= split_retry1;
         ELSIF (hgrant_m = '1'
```

```
      and hready_m = '1') THEN
         next_state <= split_wake;
      ELSE
         next_state <= split_hold;
      END IF;
   WHEN split_flush =>
      IF (hgrant_m = '1'
      and hready_m = '0') THEN
         next_state <= run;
      ELSE
         next_state <= split_hold;
      END IF;
   WHEN split_wake =>
      IF (hmaster_s = hmaster_int) THEN
         next_state <= run;
      ELSE
         next_state <= split_wake;
      END IF;
   WHEN split_retry1 =>
         next_state <= split_retry2;
   WHEN split_retry2 =>
      IF (hgrant_m = '1'
      and hready_m = '1') THEN
         next_state <= split_wake;
      ELSIF (hsel_s = '1' and
      not hmaster_s = hmaster_int) THEN
         next_state <= split_retry1;
      ELSE
         next_state <= split_hold;
      END IF;
   WHEN OTHERS =>
      next_state <= idle;
   END CASE;

END PROCESS nextstate;
```

```
-------------------------------------------------------------------------
output : PROCESS (
      current_state, haddr_s, hburst_s, hmastlock_s, hrdata_int,
      hrdata_m, hready_m, hresp_m, hsel_s, hsize_s, hsplit_int,
      htrans_s, hwdata_s, hwrite_s
   )
 -------------------------------------------------------------------------
   BEGIN
      -- Default Assignment
      haddr_m <= haddr_s;
      hburst_m <= hburst_s;
      hbusreq_m <= hsel_s;
      hlock_m <= hmastlock_s;
      hprot_m <= "0011";
      hrdata_s <= hrdata_m;
      hreadyout_s <= hready_m;
      hresp_s <= hresp_m;
      hsize_m <= hsize_s;
      hsplit_s <= (others=>'0');
      htrans_m <= (htrans_s(1) and hsel_s) & (htrans_s(0) and hsel_s);
      hwdata_m <= hwdata_s;
      hwrite_m <= hwrite_s;
      -- Default Assignment To Internals

      -- State Actions
      CASE current_state IS
      WHEN idle =>
         hbusreq_m <= '0';
      WHEN hold =>
         hreadyout_s <= '0';
      WHEN run_last =>
         hreadyout_s <= '0';
      WHEN run_first =>
         hrdata_s <= hrdata_int;
         hreadyout_s <= '0';
      WHEN split_flush =>
         htrans_m <= "00";
      WHEN split_wake =>
         hreadyout_s <= '0';
         -- BUSY
         htrans_m <= "01";
         hsplit_s <= hsplit_int;
      WHEN split_retry1 =>
         -- Retry
         hresp_s <= "10";
         -- Bust
         htrans_m <= "01";
      WHEN split_retry2 =>
         -- Retry
         hresp_s <= "01";
         -- Busy
         htrans_m <= "01";
      WHEN OTHERS =>
         NULL;
      END CASE;

   END PROCESS output;
```

```
-- Concurrent Statements
decode: process(hmaster_int)
begin
  case hmaster_int is
    when "0000" => hsplit_int <= "0000000000000001";
    when "0001" => hsplit_int <= "0000000000000010";
    when "0010" => hsplit_int <= "0000000000000100";
    when "0011" => hsplit_int <= "0000000000001000";
    when "0100" => hsplit_int <= "0000000000010000";
    when "0101" => hsplit_int <= "0000000000100000";
    when "0110" => hsplit_int <= "0000000001000000";
    when "0111" => hsplit_int <= "0000000010000000";
    when "1000" => hsplit_int <= "0000000100000000";
    when "1001" => hsplit_int <= "0000001000000000";
    when "1010" => hsplit_int <= "0000010000000000";
    when "1011" => hsplit_int <= "0000100000000000";
    when "1100" => hsplit_int <= "0001000000000000";
    when "1101" => hsplit_int <= "0010000000000000";
    when "1110" => hsplit_int <= "0100000000000000";
    when "1111" => hsplit_int <= "1000000000000000";
    when others  => hsplit_int <= "0000000000000000";
  end case;
end process;

END fsm;
```

Annexe D : Code du pont ARM7TDMI-AHB

Cette section contient le code du pont ARM7TDMI-AHB. Comme on peut le constater, ce code est très complexe et il est mieux d'observer le diagramme d'état à l'aide de l'outil de conception HDL Designer Pro qui a servi à le produire.

```
-- renoir header_start
--
-- VHDL Entity arm7tdmir1.wrap_ahb_arm7tdmi.symbol
--
-- Created:
--          by - bertola.etudiant (snowdon)
--          at - 22:16:01 05/19/03
--
-- Generated by Mentor Graphics' Renoir(TM) 2000.4 (Build 5)
--
-- renoir header_end
LIBRARY ieee ;
USE ieee.std_logic_1164.all;

ENTITY wrap_ahb_arm7tdmi IS
   PORT(
      arm_a       : IN     std_logic_vector (31 DOWNTO 0) ;
      arm_dout    : IN     std_logic_vector (31 DOWNTO 0) ;
      arm_lock    : IN     std_logic  ;
      arm_mas     : IN     std_logic_vector (1 DOWNTO 0) ;
      arm_nmreq   : IN     std_logic  ;
      arm_nrw     : IN     std_logic  ;
      arm_seq     : IN     std_logic  ;
      hclk        : IN     std_logic  ;
      hgrant      : IN     std_logic  ;
      hrdata      : IN     std_logic_vector (31 DOWNTO 0) ;
      hready      : IN     std_logic  ;
      hreset_n    : IN     std_logic  ;
      hresp       : IN     std_logic_vector (1 DOWNTO 0) ;
      arm_abort   : OUT    std_logic  ;
      arm_din     : OUT    std_logic_vector (31 DOWNTO 0) ;
      arm_mclk    : OUT    std_logic  ;
      arm_nreset  : OUT    std_logic  ;
      haddr       : OUT    std_logic_vector (31 DOWNTO 0) ;
      hburst      : OUT    std_logic_vector (2 DOWNTO 0) ;
      hbusreq     : OUT    std_logic  ;
      hlock       : OUT    std_logic  ;
      hprot       : OUT    std_logic_vector (3 DOWNTO 0) ;
      hsize       : OUT    std_logic_vector (2 DOWNTO 0) ;
      htrans      : OUT    std_logic_vector (1 DOWNTO 0) ;
      hwdata      : OUT    std_logic_vector (31 DOWNTO 0) ;
      hwrite      : OUT    std_logic
   );

-- Declarations

END wrap_ahb_arm7tdmi ;
```

```
-- renoir interface_end
--
-- VHDL Architecture arm7tdmir1.wrap_ahb_arm7tdmi.fsm
--
-- Created:
--          by - bertola.etudiant (snowdon)
--          at - 22:16:02 05/19/03
--
-- Generated by Mentor Graphics' Renoir(TM) 2000.4 (Build 5)
--
LIBRARY ieee ;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ARCHITECTURE fsm OF wrap_ahb_arm7tdmi IS

   -- Architecture Declarations
   signal haddr_int: std_logic_vector(31 downto 0);
   signal hwrite_int: std_logic;
   signal hsize_int: std_logic_vector(2 downto 0);
   signal hrdata_int: std_logic_vector(31 downto 0);
   signal mclken_n: std_logic;
   signal lock_flag: std_logic;

   TYPE STATE_TYPE IS (
      run,
      run_first,
      hold,
      run_last,
      lock_first,
      not_ready,
      ex_flush,
      ex_return,
      ex_try_again,
      ex_hold
   );

   -- State vector declaration
   ATTRIBUTE state_vector : string;
   ATTRIBUTE state_vector OF fsm : ARCHITECTURE IS "current_state" ;


   -- Declare current and next state signals
   SIGNAL current_state : STATE_TYPE ;
   SIGNAL next_state : STATE_TYPE ;

BEGIN

-----------------------------------------------------------------------
clocked : PROCESS(
      hclk,
      hreset_n
   )
-----------------------------------------------------------------------
BEGIN
      IF (hreset_n = '0') THEN
         current_state <= run;
         -- Reset Values
         haddr_int <= (others=> '0');
```

```
   hrdata_int <= (others=>'0');
   hsize_int <= (others=>'0');
   hwrite_int <= '0';
   lock_flag <= '0';
ELSIF (hclk'EVENT AND hclk = '1') THEN
   current_state <= next_state;
   -- Default Assignment To Internals

   -- Transition Actions for internal signals only
   CASE current_state IS
   WHEN run =>
      IF (hready = '1' and
      arm_lock = '1' and
      lock_flag = '0') THEN
         lock_flag <= '1';
      ELSIF (hready = '0' and
      (hresp = "10" or
       hresp = "11")) THEN
      ELSIF (hready = '1' and
      hgrant = '0') THEN
         haddr_int <= arm_a;
         hwrite_int <= arm_nrw;
         hsize_int <= '0' & arm_mas;
      ELSIF (hready = '0') THEN
      ELSIF (hready = '1') THEN
         haddr_int <= arm_a;
         hwrite_int <= arm_nrw;
         hsize_int <= '0' & arm_mas;
         if (arm_lock = '0') then
           lock_flag <= '0';
         end if;
      END IF;
   WHEN run_first =>
      IF (hready = '1' and
      arm_lock = '1') THEN
      ELSIF (hready = '1') THEN
         haddr_int <= arm_a;
         hwrite_int <= arm_nrw;
      ELSIF (hready = '1' and
      hgrant = '0') THEN
         haddr_int <= arm_a;
         hwrite_int <= arm_nrw;
         hsize_int <= '0' & arm_mas;
      END IF;
   WHEN run_last =>
      IF (hready = '0' and
      (hresp = "10" or
       hresp = "11")) THEN
      ELSIF (hready = '1' and
      hgrant = '1') THEN
         hrdata_int <= hrdata;
      ELSIF (hready = '1' and
      hgrant = '0') THEN
         hrdata_int <= hrdata;
      END IF;
   WHEN not_ready =>
      IF (hready = '0' and
      (hresp = "10" or
       hresp = "11")) THEN
      ELSIF (hready = '1' and
```

```
            hgrant = '0') THEN
               haddr_int <= arm_a;
               hwrite_int <= arm_nrw;
            ELSIF (hready = '1') THEN
               haddr_int <= arm_a;
               hwrite_int <= arm_nrw;
               hsize_int <= '0' & arm_mas;
               if (arm_lock = '1') then
                 lock_flag <= '0';
               end if;
            END IF;
         WHEN ex_return =>
            IF (hready = '0' and
            (hresp = "10" or
             hresp = "11")) THEN
            ELSIF (hready = '1' and
            hgrant = '0') THEN
            ELSIF (hgrant = '1') THEN
               haddr_int <= arm_a;
               hwrite_int <= arm_nrw;
               hsize_int <= '0' & arm_mas;
               if (arm_lock = '0') then
                 lock_flag <= '0';
               end if;
            END IF;
         WHEN OTHERS =>
            NULL;
         END CASE;

      END IF;

   END PROCESS clocked;

-------------------------------------------------------------------------
nextstate : PROCESS (
      arm_lock,
      current_state,
      hgrant,
      hready,
      hresp,
      lock_flag
   )
-------------------------------------------------------------------------
BEGIN
      CASE current_state IS
      WHEN run =>
         IF (hready = '1' and
         arm_lock = '1' and
         lock_flag = '0') THEN
            next_state <= lock_first;
         ELSIF (hready = '0' and
         (hresp = "10" or
          hresp = "11")) THEN
            next_state <= ex_flush;
         ELSIF (hready = '1' and
         hgrant = '0') THEN
            next_state <= run_last;
         ELSIF (hready = '0') THEN
            next_state <= not_ready;
         ELSIF (hready = '1') THEN
```

```
      next_state <= run;
   ELSE
      next_state <= run;
   END IF;
WHEN run_first =>
   IF (hready = '1' and
   arm_lock = '1') THEN
      next_state <= lock_first;
   ELSIF (hready = '1') THEN
      next_state <= run;
   ELSIF (hready = '1' and
   hgrant = '0') THEN
      next_state <= run_last;
   ELSE
      next_state <= run_first;
   END IF;
WHEN hold =>
   IF (hready = '1' and
   hgrant = '1') THEN
      next_state <= run_first;
   ELSE
      next_state <= hold;
   END IF;
WHEN run_last =>
   IF (hready = '0' and
   (hresp = "10" or
    hresp = "11")) THEN
      next_state <= ex_flush;
   ELSIF (hready = '1' and
   hgrant = '1') THEN
      next_state <= run_first;
   ELSIF (hready = '1' and
   hgrant = '0') THEN
      next_state <= hold;
   ELSE
      next_state <= run_last;
   END IF;
WHEN lock_first =>
   IF (hready = '1') THEN
      next_state <= run;
   ELSE
      next_state <= lock_first;
   END IF;
WHEN not_ready =>
   IF (hready = '0' and
   (hresp = "10" or
    hresp = "11")) THEN
      next_state <= ex_flush;
   ELSIF (hready = '1' and
   hgrant = '0') THEN
      next_state <= run_last;
   ELSIF (hready = '1') THEN
      next_state <= run;
   ELSE
      next_state <= not_ready;
   END IF;
WHEN ex_flush =>
   IF (hready = '1' and
   hgrant = '1') THEN
      next_state <= ex_try_again;
```

```
            ELSIF (hready = '1' and
            hgrant = '0') THEN
               next_state <= ex_hold;
            ELSE
               next_state <= ex_flush;
            END IF;
         WHEN ex_return =>
            IF (hready = '0' and
            (hresp = "10" or
             hresp = "11")) THEN
               next_state <= ex_flush;
            ELSIF (hready = '1' and
            hgrant = '0') THEN
               next_state <= run_last;
            ELSIF (hgrant = '1') THEN
               next_state <= run;
            ELSE
               next_state <= ex_return;
            END IF;
         WHEN ex_try_again =>
            IF (hready = '1' and
            hgrant = '1') THEN
               next_state <= ex_return;
            ELSE
               next_state <= ex_try_again;
            END IF;
         WHEN ex_hold =>
            IF (hready = '1' and
            hgrant = '1') THEN
               next_state <= ex_try_again;
            ELSE
               next_state <= ex_hold;
            END IF;
         WHEN OTHERS =>
            next_state <= run;
         END CASE;

      END PROCESS nextstate;

-------------------------------------------------------------------------
output : PROCESS (
         arm_a,
         arm_dout,
         arm_lock,
         arm_mas,
         arm_nmreq,
         arm_nrw,
         arm_seq,
         current_state,
         haddr_int,
         hgrant,
         hrdata,
         hrdata_int,
         hready,
         hreset_n,
         hresp,
         hsize_int,
         hwrite_int,
         lock_flag
      )
```

```
-------------------------------------------------------------------
BEGIN
      -- Default Assignment
      arm_abort <= hresp(0) and not hresp(1) and hgrant;
      arm_din <= hrdata;
      arm_nreset <= hreset_n;
      haddr <= arm_a;
      hburst <= "001";
      hbusreq <= arm_nmreq;
      hlock <= arm_lock;
      hprot <= "0011";
      hsize <= '0' & arm_mas;
      htrans <= (not arm_nmreq) & ((not arm_nmreq) and arm_seq);
      hwdata <= arm_dout;
      hwrite <= arm_nrw;
      -- Default Assignment To Internals
      mclken_n <= '1';

      -- State Actions
      CASE current_state IS
      WHEN run =>
         if (lock_flag = '0' and arm_lock = '1') then
           htrans <= "00";
         end if;
      WHEN run_first =>
         arm_din <= hrdata_int;
      WHEN ex_try_again =>
         haddr <= haddr_int;
         hwrite <= hwrite_int;
         hsize <= hsize_int;
      WHEN OTHERS =>
         NULL;
      END CASE;

      -- Transition Actions
      CASE current_state IS
      WHEN run_first =>
         IF (hready = '1' and
         arm_lock = '1') THEN
         ELSIF (hready = '1') THEN
            hsize <= '0' & arm_mas;
         END IF;
      WHEN not_ready =>
         IF (hready = '0' and
         (hresp = "10" or
          hresp = "11")) THEN
         ELSIF (hready = '1' and
         hgrant = '0') THEN
            hsize <= '0' & arm_mas;
         END IF;
      WHEN OTHERS =>
         NULL;
      END CASE;

   END PROCESS output;
```

```
-- Concurrent Statements
-- Génère le signal d'horloge pour le ARM7... très important!
make_mclk: process(hclk, hreset_n)
begin
  if hreset_n = '0' then
    arm_mclk <= not hclk;
  elsif hclk'event and hclk = '1' then
    if next_state = run or next_state = run_last then
      arm_mclk <= '0';
    else
      arm_mclk <= '1';
    end if;
  elsif hclk = '0' then
    arm_mclk <= '1';
  end if;
end process;

END fsm;
```

Annexe E : Résultats de la synthèse

Cette section contiendra les tableaux de données compilées de la synthèse du réseau d'interconnexion.

numMaster	numSlave	Surface Combinatoire	Surface Séquentielle	Surface TOTALE	Puissance (mW)
2	3	21100	5449	26549	1.0027
3	3	32889	5449	38338	1.3935
4	3	41135	5449	46584	1.7457
5	3	49746	5449	55195	2.1234
6	3	64408	5449	69857	2.5178
7	3	72776	5449	78225	2.8868
8	3	80821	5449	86270	3.1997
9	3	93967	5449	99416	3.6978
10	3	103989	5449	109438	4.0357
11	3	118952	5449	124401	4.4509
12	3	126888	5449	132337	4.7781
13	3	137081	5449	142530	5.1395
14	3	145235	5449	150684	5.472
15	3	153494	5449	158943	5.8804
16	3	161560	5449	167009	6.212
2	4	25532	5914	31446	1.2145
2	5	30446	6379	36825	1.4201
2	6	39144	6844	45988	1.6862
2	7	43706	7309	51015	1.8869
2	8	48515	7774	56289	2.0898
2	9	55617	8239	63856	2.3027
2	10	62035	8704	70739	2.5576
2	11	70595	9169	79764	2.8594
2	12	75186	9634	84820	3.0507
2	13	80527	10099	90626	3.2502
2	14	85202	10564	95766	3.4509
2	15	89785	11029	100814	3.6531
2	16	94280	11494	105774	3.8529

Tableau E.1: Résultats de la synthèse