POLYPUBLIE
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Interface configurable pour un processeur ARM basée sur le protocole VCI
Auteur: Author:	Geneviève Cyr
Date:	2001
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Cyr, G. (2001). Interface configurable pour un processeur ARM basée sur le protocole VCI [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/6946/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/6946/
Directeurs de recherche: Advisors:	Guy Bois, & El Mostapha Aboulhamid
Programme: Program:	Non spécifié

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

UNIVERSITÉ DE MONTRÉAL

INTERFACE CONFIGURABLE POUR UN PROCESSEUR ARM BASÉE SUR LE PROTOCOLE VCI.

GENEVIÈVE CYR
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE ET INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION DU DIPLÔME
DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
FÉVRIER 2001

© Geneviève Cyr, 2001

National Library of Canada
Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services
Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-65560-1

Canada

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

INTERFACE CONFIGURABLE POUR UN PROCESSEUR ARM BASÉE SUR LE PROTOCOLE VCI.

présenté par : CYR Geneviève

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. ABOULHAMID Mostapha, Ph.D., membre et codirecteur de recherche

M. SAWAN Mohamad, Ph.D., membre

REMERCIEMENTS

Je tiens particulièrement à remercier mon directeur de recherche Guy Bois, pour son soutien, sa compréhension, sa confiance et sa flexibilité. Il m'a permis de m'épanouir et de travailler à un projet que j'avais envie de réaliser, tout en supportant mes initiatives et mes idées. Je voudrais également remercier mon codirecteur pour ses judicieux conseils et sa disponibilité. De plus, je voudrais remercier Jacques Baillargé dont le soutien technique et les idées ont été très appréciés.

Je dois aussi remercier les organismes qui m'ont subventionnée, c'est-à-dire le GRIAO et le CRSNG. De plus, je dois souligner la compagnie Mentor Graphics qui a subventionné le projet et dont les membres ont fourni un support technique intéressant.

En outre, deux stagiaires ont collaboré à ce projet : Marc Bertola et Grégory Royer. Ils m'ont aidée à implanter les algorithmes du tri par segmentation et de l'encodeur/décodeur *Reed Solomon* utiles pour l'analyse des résultats de l'interface développée. De plus, je ne peux passer sous silence mes compagnons de travail. Yannick Héneault et Olivier Hebert m'ont orientée en début de projet et Jean-Marc Tremblay m'a soutenue tout au long du projet.

Enfin, je dois remercier mon entourage pour son appui, notamment au début du projet. En particulier, le soutien de ma mère et celui de mon copain Nicolas ont été essentiels.

RÉSUMÉ

Le rythme des progrès de la technologie entraîne l'augmentation rapide de la taille des circuits intégrés. Le développement de systèmes sur une puce est une des seules solutions viables à ce problème de croissance des circuits. La réutilisation de modules est essentielle pour éviter d'augmenter le délai de conception tout en permettant d'augmenter la complexité des circuits. Cependant, pour que la réutilisation soit efficace, l'intégration des modules et le développement des communications entre ceux-ci doivent se faire rapidement. L'utilisation d'un protocole standard pour établir les communications entre les modules est une solution intéressante.

L'objectif du projet est de développer une partie de méthodologie de synthèse des communications pour des systèmes sur une puce en allégeant la tâche des concepteurs de systèmes utilisant un ARM7 (Advanced Risc Machine). Pour ce faire, une interface matérielle VHDL (Very High Speed Integrated Circuit Description Language) configurable pour ce processeur est développée. Tous les paramètres de l'interface sont regroupés dans un fichier VHDL. La modification de ces paramètres permet d'obtenir une interface spécifique à un système selon les mécanismes de communication et de synchronisation choisis. L'interface permet au ARM7 de communiquer au moyen du protocole VCI (Virtual Component Interface) de VSIA (Virtual Socket Interface Alliance). Ainsi, la performance de ce protocole, encore en développement, peut être évaluée dans un contexte de communication point-à-point.

L'interface VHDL est développée à l'aide de l'outil Renoir de Mentor Graphics. Les mécanismes de communication implantés sont la mémoire partagée et le passage de messages. Le premier mécanisme sera implanté à l'aide d'une mémoire duale et le deuxième à l'aide d'une FIFO (First In First Out). En plus, plusieurs mécanismes de synchronisation seront possibles. La génération automatique se fera avec un programme Perl à partir des fichiers VHDL contenant des clauses "generate" et d'un

"package" VHDL contenant des constantes. De plus, la synthèse de l'interface sera effectuée avec l'outil Design Analyser de Synopsys. Pour tester l'interface matérielle VHDL, des bancs de tests seront simulés à l'aide de l'outil Modelsim. De plus, les modèles VHDL des mémoires et du processeur seront utilisés pour simuler le système matériel/logiciel à l'aide de l'outil de cosimulation Seamless de Mentor Graphics.

Trois applications différentes seront utilisées pour tester l'interface et le protocole : un algorithme de filtrage, de tri par segmentation et d'encodage/décodage Reed Solomon. L'interface esclave est évaluée avec l'implantation matérielle du filtre et de l'encodeur. La performance de l'interface maître est mesurée avec l'implantation logicielle du filtre. Enfin, l'implantation multiprocesseur de l'algorithme du tri par segmentation permet de tester les performances globales de l'interface développée.

En somme, l'utilisation d'un protocole standard peut diminuer la tâche de synthèse et d'intégration des systèmes. Pour la conception de systèmes sur une puce, l'adoption d'un protocole standard au niveau de l'industrie est essentielle. Les normes de VSIA, particulièrement le protocole VCI, constituent une bonne initiative. Cependant, l'utilisation de VCI peut, dans certains cas, augmenter le temps des communications. Par exemple, une interface VCI augmentera le temps d'une opération de lecture du processeur de 67 % et celui d'une écriture de 50 %. Pour les applications axées sur le transfert de données, les performances peuvent être diminuées si des dispositifs particuliers ne sont pas introduits. Cependant, les performances des applications où le contrôle est prédominant ne sont pratiquement pas affectées.

ABSTRACT

Technology advances have a tremendous effect on the complexity of integrated circuits. While systems complexity is growing, the time-to-market window is decreasing. The current methodology for chip design cannot adequately handle this growth. That is why System-On-Chip (SoC) methodologies are becoming a viable way to deal with this complexity. Synthesis of system-on-chip is not straightforward and IP (Intellectual Property) reuse is necessary to shorten design time. Also, communication synthesis enables rapid integration of intermodule communication.

Developing an interface synthesis tool is a complex task and we have decided to focus only on certain aspects. We chose to generate IP interfaces for an ARM (Advanced Risc Machine) processor to allow IP-to-IP communication using the VCI (Virtual Component Interface) standards from VSIA (Virtual Socket Interface Alliance). Different design architectures and communication synthesis methodologies are explored. Also, the performance of this standard under development is tested. To achieve these goals, a generic interface in VHDL (Very High Speed Integrated Circuit Description Language) for an ARM processor has been developed. This interface integrates many communication and synchronization mechanisms to fill different application needs. Parameters of the interface are grouped in a VHDL files and modification of this file allow the creation of a specific interface for an application.

All the design has been done with Renoir, the Mentor Graphics HDL (Hardware Description Language) graphical design environment. The interface has been written in VHDL using user-defined types and constants as well as VHDL generate clauses. From the VHDL files, a Perl script can be run with user parameters to obtain the desired specific interface. The synthesis has been done using Design Analyser of Synopsys. The overall hardware-software design is then co-simulated using Seamless-

CVE from Mentor Graphics. The VHDL files of the design include Seamless models for the processor and memories.

Three different algorithms have been implemented to test VCI protocol and generic interface performances: a pattern matcher, a Reed Solomon encoder/decoder and a quicksort. The slave interface is tested with a hardware implementation of the pattern matcher and of the encoder. The master interface performance is evaluated with a software implementation of the pattern matcher. Finally, a software implementation of the quicksort on a multiprocessor is used to test the global interface.

In summary, we believe that the use of a standard protocol can symplify the interface synthesis task. However, the use of VCI can increase communication time for certain applications. In fact, a single load operation time is increased by 67% and a single store operation by 50%. Thus, for data-driven applications, performances could decrease if special architectural features are not introduced. Nevertheless, performances are not affected for control-driven applications or applications where requests are independent of previous responses. VSIA is a good standardization initiative. However, different modifications could improve it. For example, adding control signals could allow full use of the address space for storing data. Also, allowing other protocols than handshake could reduce the performance loss of load and store operations.

TABLE DES MATIERES

REMERCIEMENTS IV

RÉSUMÉ V

ABSTRACT VII

TABLE DES MATIÈRES IX

LISTE DES TABLEAUX XIII

LISTE DES FIGURES XV

LISTE DES ABRÉVIATIONS XVIII

LEXIQUE XXI

LISTE DES ANNEXES XXV

INTRODUCTION 1

CHAPITRE 1 : REVUE DES MÉTHODOLOGIES DE CONCEPTION DE SOC 6

1.1. Synthèse et raffinement d'une spécification de haut niveau 7
1.1.1. Chinook 7
1.1.2. SpecC 8
1.1.3. COOL 10
1.1.4. COSMOS 10
1.1.5. VULCAN 11
1.1.6. Autres méthodologies 12
1.2. Architecture et normes de SoC 16
1.2.1. Bus et plate-forme 17
1.2.2. Protocoles 21
1.2.3. VSIA 23

1.3. Possibilités de réutilisation des IP 27
1.3.1. Principes de réutilisation 27
1.3.2. Synopsys 31
1.3.3. Cadence 32
1.3.4. Coware 34
CHAPITRE 2 : MÉTHODOLOGIE PROPOSÉE ET DÉFINITION DU PROJET 36
2.1. Méthodologie de conception de SoC 36
2.1.1. Méthodologie globale 36
2.1.2. Représentation du canal de communication 40
2.1.3. Aspects de la méthodologie traités 42
2.2. Description des caractéristiques de l'interface 43
2.2.1. Type d'IP : ARM7TDMI 43
2.2.2. Moyens de communication et de synchronisation 47
2.2.3. Protocole de communication 50
2.3. Utilisation de l'interface 53
2.3.1. Configuration de l'interface 54
2.3.2. Contexte d'utilisation 57
CHAPITRE 3 : ANALYSE DU DESIGN DE L'INTERFACE 59
3.1. Choix d'implémentation 59
3.1.1. Limitations du protocole VCI 59
3.1.2. Synchronisation 61
3.1.3. Structure des modules 63
3.1.4. Description du niveau hiérarchique supérieur 64
3.1.5. Composants mémoires 65
3.2. Registres d'interruptions 67
3.2.1. Utilité des registres 67
3.2.2. Description des registres 68
3.2.3. Structure des registres 69

3.3. Commutateur 71

3.3.1. Description du commutateur 71

3.3.2. Contraintes d'utilisation du commutateur 74

3.4. Interface esclave 75

3.4.1. Description des requêtes et des réponses 75

3.4.2. Description globale de l'interface esclave 78

3.4.3. Chemin de données de la requête 81

3.4.4. Chemin de données de la réponse 84

3.5. Interface maître 85

3.5.1. Utilisation de l'interface 85

3.5.2. Structure générale 87

3.5.3. Synchronisation 88

3.5.4. Description du contrôleur 90

CHAPITRE 4 : RÉSULTATS 93

4.1. Résultats généraux 93

4.1.1. Covérification de l'interface 93

4.1.2. Synthèse de l'interface 94

4.1.3. Contexte d'évaluation des performances 98

4.2. Algorithme de filtrage 99

4.2.1. Description de l'implantation de l'algorithme de filtrage 100

4.2.2. Partitionnement logiciel/matériel 101

4.2.3. Partitionnement logiciel/logiciel 103

4.2.4. Analyse des résultats de l'algorithme de filtrage 105

4.3. Algorithme d'encodage et de décodage Reed Solomon 107

4.3.1. Description de l'implantation de l'encodeur/décodeur ReedSlomon 107

4.3.2. Analyse des résultats de l'encodeur/décodeur Reed Solomon 109

4.4. Algorithme du tri par segmentation 111

4.4.1. Description de l'implantation de l'algorithme de tri 111

4.4.2. Analyse des résultats de l'algorithme de tri 113

4.5. Discussion 114
4.5.1. Évaluation de l'interface 115
4.5.2. Évaluation du protocole BVCI 118
4.5.3. Évaluation de la méthodologie proposée 121
CONCLUSION 123
RÉFÉRENCES 127
ANNEXES 134

LISTE DES TABLEAUX

Tableau 2.1. Mécanismes de communication de l'interface....50
Tableau 2.2. Paramètres des composants de communication....56
Tableau 3.1. Description des différents registres....69
Tableau 3.2. Espace d'adressage des registres d'interruptions....69
Tableau 3.3. Association des signaux du processeur aux signaux mémoires....74
Tableau 3.4. Restriction des délais des mémoires....74
Tableau 3.5. Description des requêtes traitées par l'interface esclave....76
Tableau 3.6. Paramètres VCI de l'interface esclave....81
Tableau 3.7. Association des signaux du VCI aux signaux des mémoires....83
Tableau 3.8. Association des signaux du processeur aux signaux VCI....88
Tableau 3.9. Temps des opérations de lecture et écriture....91
Tableau 4.1. Espace occupé (unités de base de Synopsys) par chaque module....96
Tableau 4.2. Période minimale de l'horloge VCI pour chaque module....97
Tableau 4.3. Délais du filtrage pour une implantation matérielle....103
Tableau 4.4. Délais d'exécution des opérations mémoire du processeur....104
Tableau 4.5. Délais du filtrage pour une implantation logicielle....105
Tableau 4.6. Performances des implémentations de l'algorithme de filtrage....107
Tableau 4.7. Délais de l'algorithme d'encodage....111
Tableau 4.8. Comparaison des différentes implémentations du tri....113
Tableau B.1. Code pour la résolution de conflit d'accès....156
Tableau B.2. Opérations atomiques et leurs opérations duales....156
Tableau B.3. Étapes de génération d'une interface....157
Tableau G.1. Description des signaux importants du ARM....200
Tableau G.2. Délais pour les transactions de lectures et écritures....203
Tableau G.3. Description des signaux VCI....206
Tableau H.1. Paramètres BVCI....208
Tableau I.1. Description des ports d'une DPRAM....209

Tableau I.2. Paramètres de la DPRAM....210
Tableau I.3. Mode de fonctionnement d'une DPRAM....211
Tableau I.4. Description des ports d'une SRAM....211
Tableau I.5. Modes de fonctionnement de la SRAM....212
Tableau I.6. Paramètres de la SRAM....212
Tableau I.7. Description des ports de la FIFO....212
Tableau J.1. Structure du tableau du le banc de test....217

LISTE DES FIGURES

Figure 1.1. Génération d'enveloppes d'IP proposé dans [DoGa00] 9
Figure 1.2. Canal de communication selon Madsen [KnMa98] 12
Figure 1.3. Architecture du module d'interface utilisé dans [SuBr92] 13
Figure 1.4. Architecture type du système proposé dans [BPM98] 14
Figure 1.5. Interface à plusieurs niveaux d'abstractions 15
Figure 1.6. Architecture "CoreConnect" d'IBM 18
Figure 1.7. Architecture "CoreFrame" de PalmChip 19
Figure 1.8. Architecture AMBA de ARM 20
Figure 1.9. Utilisation d'un pont pour connexion au OCB 22
Figure 1.10. Raffinement hiérarchique proposé par VSIA 24
Figure 1.11. Évolution de la réutilisation 28
Figure 1.12. Étapes de conception de l'outil VCC de Cadence 33
Figure 1.13. Étapes de design de Coware 35
Figure 2.1. Étapes de conception proposées dans le présent projet 37
Figure 2.2. Étapes de l'établissement des communications entre deux blocs 40
Figure 2.3. Représentation du système 41
Figure 2.4. Modèles de communication 48
Figure 2.5. Synchronisation par mémoire partagée et interruptions 49
Figure 2.6. Synchronisation par passage de messages 49
Figure 2.7. Transfert de données entre l'initiateur et la cible (BVCI) 52
Figure 2.8. Mécanisme de poignée de main (BVCI) 52
Figure 2.9. Topologies proposées par VSIA 58
Figure 2.10. Topologie des systèmes réalisés dans le cadre du projet 58
Figure 3.1. Spécifications des délais du protocole BVCI 61
Figure 3.2. Horloges et signal de synchronisation de l'interface 62
Figure 3.3. Structure typique des modules de l'interface 63
Figure 3.4. Diagramme bloc structuré de l'interface 64

Figure 3.5. Diagramme bloc fonctionnel de l'interface 65
Figure 3.6. Construction d'une mémoire 32 bits à partir d'une 8 bits 66
Figure 3.7. Représentation du double FIFO 67
Figure 3.8. Accès non alignés à la FIFO 67
Figure 3.9. Schéma bloc des registres d'interruptions 70
Figure 3.10. Schéma logique et diagramme temporel du bloc de lecture 71
Figure 3.11. Schéma logique et diagramme temporel du bloc d'écriture 71
Figure 3.12. Schéma bloc du commutateur 73
Figure 3.13. Diagramme temporel du commutateur 73
Figure 3.14. Accès aux boîtes aux lettres et génération d'interruption 76
Figure 3.15. Types d'interruptions directes 77
Figure 3.16. Diagramme bloc de l'interface esclave 80
Figure 3.17. Diagramme temporel de l'interface esclave 80
Figure 3.18. Structure de la FIFO de l'interface esclave 82
Figure 3.19. Schéma bloc de l'interface maître 87
Figure 3.20. Diagramme temporel de l'interface maître 89
Figure 3.21. Schéma bloc simplifié du contrôleur de l'interface maître 91
Figure 3.22. Machine à états du contrôleur : la lecture 92
Figure 3.23. Machine à états du contrôleur : l'écriture 92
Figure 4.1. Schéma bloc des deux types d'implantation de système 99
Figure 4.2. Représentation des 256 pixels d'une image 100
Figure 4.3. Modèles de communication avec l'interface VCI 104
Figure 4.4. Utilisation d'une procédure pour établir les communications 109
Figure 4.5. Schéma de l'implantation de l'algorithme de tri 113
Figure A.1. Le système de co-synthèse Chinook 136
Figure A.2. Équations de détermination du délai de communication 141
Figure A.3. Communication multi-hop 141
Figure A.4. Association des ports dans Chinook 146
Figure B.1. Modèles d'implémentation 158
Figure C.1. Architecture utilisée dans l'outil COOL 163

Figure D.1. Représentation d'un système en SOLAR 170
Figure D.2. Allocation des canaux dans la synthèse des communications 173
Figure E.1. Architecture cible de l'outil VULCAN 176
Figure E.2. Architecture d'interface utilisée pour la simulation 180
Figure E.3. Machine à états du contrôleur de la FIFO 180
Figure E.4. Logique combinatoire pour relier des interfaces incompatibles 181
Figure F.1. Illustration du wait transfer mode dans COWARE 190
Figure F.2. Adapteur pour des processus d'horloges reliées 191
Figure F.3. Communications entre un processeur et un module externe 194
Figure F.4. Module d'interface synthétisée avec Integral 196
Figure G.1. Diagramme fonctionnel du ARM7TDMI 199
Figure G.2. Diagramme temporel pour un adressage en mode pipeline 201
Figure G.3. Diagramme temporel pour un adressage en mode normal 201
Figure G.4. Diagramme temporel général 202
Figure G.5. Diagramme temporel d'une écriture 202
Figure G.6. Diagramme temporel d'une lecture 202
Figure G.7. Architecture de mémoire typique 203
Figure G.8. Définition d'un espace d'adressage typique 204
Figure G.9. Schéma logique d'un activateur de banque mémoire 204
Figure G.10. Schéma logique de base d'un registre d'interruption 204
Figure H.1. Signaux du transfert d'une requête BVCI 205
Figure H.2. Signaux du transfert d'une réponse BVCI 205
Figure H.3. Diagramme temporel d'une opération de lecture 207
Figure H.4. Diagramme temporel d'une opération d'écriture 207

LISTE DES ABRÉVIATIONS

AHB	***Advanced High-performance Bus*** : bus haute performance de l'architecture AMBA
AMBA	***Advanced Microcontroller Bus Architecture*** : architecture de SoC
APB	***Advanced Peripheral Bus*** : bus de périphérique de l'architecture AMBA
ARM	***Advanced Risc Machine***
ASB	***Advanced System Bus*** : bus de système de l'architecture AMBA
ASIC	***Application Specific Integrated Circuit*** ou circuit intégré à une application
ASIP	***Application Specific Instruction Set Processor*** ou processeur à instructions dédiées à une application spécifique
AVCI	***Advanced Virtual Component Interface***
BVCI	***Basic Virtual Component Interface***
CFG	***Control Flow Graph*** ou diagramme de contrôle
CMOS	***Complementary Metal Oxide Semiconductor***
CPU	***Control Processing Unit***
CU	***Communication Unit*** (*channel unit*) ou unité de communication (canal)
DCR	***Device Control Register bus*** bus de contrôle de registre (utilisé par IBM)
DMA	***Direct Memory Access***
DPRAM	***Dual Port Random Access Memory*** ou mémoire à double ports
DSP	***Digital Signal Processing unit*** ou processeur numérique de traitement de signal
DU	***Design Unit*** ou unité fonctionnelle
FIFO	***First In First Out*** ou Premier entré, premier sorti
FPGA	***Field Programmable Gate Array*** ou circuit intégré programmable
FSM	***Finite State Machine*** ou machine à états finis et étendus
GBI	***Generic Bus Interface*** : protocole de communication pour IP
HW	Hardware *ou matériel*

I/O	***In/Out*** **ou entrée/sortie**
IP	**Intellectual Property module** ***ou module de propriété intellectuelle (*****voir** ***core)***
ISA	***Industry Standard Architecture*** **: norme de bus**
MAC	***Memory Access Controler*** **ou controleur d'accès mémoire**
MB	***MailBox*** **ou Boîte aux lettres**
MIPS	**Million d'Instructions Par Seconde**
MU	***Memory Unit*** **ou unité de mémoire**
N2C	**Napkin-to-Chip : outil de design SoC (Coware)**
OCB	***On Chip Bus*** **ou bus sur une puce**
OCP	***OpenCore Protocol*** **: protocole de communication pour IP**
PCI	***Peripheral Component Interconnect*** **: bus synchrone utilisé entre autres dans les ordinateurs personnels**
PLB	***Processor Local Bus*** **ou Bus local pour le processeur (utilisé par IBM)**
PU (PE)	***Processing Unit (element)*** **ou unité fonctionnelle ou éléments de traitements (voir DU)**
PVCI	***Peripheral Virtual Component Interface***
RAM	***Random Access Memory***
RISC	**Reduced Instruction Set Computers**
ROM	***Read Only Memory***
RPC	***Remote Procedure Call*** **ou technique utilisant le modèle Client-Serveur : protocole gérant les interactions entre le client et le serveur.**
RTL	***Register Transfer Level*** **ou description au niveau des registres**
SDL	***System Description Language*** **ou langage de description système**
SEQ	***Sequence*** **ou séquence d'opérations à exécuter pour accéder à un périphérique**
SLIF	***System-Level Interface Standard*** **: norme de VSIA**
SOC	**Systeme On Chip ou Système sur une puce**
SPR	***Static Pin Rules*** **ou information au niveau matériel sur les ports des composants**
SRAM	***Static Random Access Memory***
ST	***State Table*** **ou table d'états**

STG	*State Transition Graph* ou graphe de transition d'états, on peut aussi lui donner la signification *Signal Transition Graph* ou graphe de transition de signaux
SW	*Software* ou logiciel
TDT	*Timed Decision Table* ou table de décision basée sur les délais
VC	*Virtual Component* (voir IP)
VCC	*Virtual Component Co-Design* : outil SoC de Cadence
VCI	*Virtual Component Interface* : protocole de communication pour IP proposé par VSIA
VHDL	*Very High Speed Integrated Circuit Description Language* ou langage de description de circuit intégré de très haute vitesse
VME	*VerseModule Eurocard* : bus asynchrone souvent utilisé dans des systèmes d'instrumentation
VSIA	*Virtual Socket Interface Alliance*

LEXIQUE

À trois états	Composant logique pouvant générer une sortie à haute impédance **Ang :** Tristate
Accusé de réception	**Ang :** Acknowledge
Activateur	**Ang :** enable
Activateur d'octets	**Ang :** byte enable
Aléas	Voir Impulsion transitoire
Analyseur syntaxique	**Ang :** Parser
Approche ascendante	Méthode de conception dans laquelle le système est construit de bas en haut, donc différents blocs sont assemblés pour former le système. **Ang :** Bottom-up
Approche descendante	Méthode de conception dans laquelle une description est rafinée de haut en bas, donc la description globale est sectionnée en sous-blocs. **Ang :** Top-down
Asynchrone	Dans le domaine de la conception et de la synthèse, signifie sans horloge. Dans le domaine des communications, signifie communication non-bloquante.
Biais de synchronisation	Différence de temps entre les montées de l'horloge à deux endroits différents de la puce **Ang :** Skew
Boîte aux lettres	Espace mémoire réservé à la génération d'interruption. **Ang :** Mailbox
Chaîne de balayage	**Ang :** Scan
Chemin de données	**Ang :** Datapath
Cible	Voir Esclave
Communication de diffusion	Diffusion pour des envois de masse **Ang :** Broadcast

Commutateur	**Ang :** Shifter
Couche	Couche ou niveau d'abstraction selon VSIA **Ang :** Layer
Déviation	**Ang :** ByPass
Division	Processus permettant la diminution de la taille des paquets **Ang :** Splitting
Emmagasinage	**Ang :** Store
Envoi/réception	**Ang :** Send/receive
Esclave	Composant au service d'un composant maître (exemple une mémoire) **Ang :** Slave, Target ou Client
Fichier d'interconnexions	Fichier de description d'un circuit au niveau portes **Ang :** Netlist
Filtrage	Algorithme permettant la détection de la position d'un patron dans une image. **Ang :** Pattern matching
Gestionnaire	Partie fonctionnelle d'un design utilisée pour synchroniser les communications **Ang :** Handler
Graphe de flux de données	**Ang :** Flowgraph
Gros-boutiste	Description d'une façon de stocker les nombres dans plusieurs octets où l'octet de poids fort est stocké avant l'octet de poids faible **Ang :** Big-endian
Impasse	Blocage du système, par exemple à cause d'une boucle infinie ou de deux processus qui s'attendent mutuellement. **Ang :** Deadlock
Impulsion transitoire	Saut non désiré sur un signal **Ang :** Glitch
Initiateur	Voir Maître

Liaison à plusieurs bonds	Qualificatif d'une communication passant par un composant intermédiaire
Maître	Composant ayant un pouvoir d'exécution **Ang :** Master, Initiator ou Server
Mélange et appariement	**Ang :** Mix and Match
Multi-thread	Capacité d'effectuer plusieurs tâches légères, petits programmes ou routines
Opération mémoire bloquante	**Ang :** Locked operation
Paquet	Communication assurant une transmission continue des informations entre deux éléments et permettant une réception sans état d'attente **Ang :** Burst
Paquetage	Processus permettant l'augmentation de la taille des paquets **Ang :** Packing
Petit-Boutiste	Description d'une façon de stocker les nombres dans plusieurs octets où l'octet de poids faible est stocké avant l'octet de poids fort. **Ang :** Little endian
Plan de masse	**Ang :** Floorplan
Poignée de main	Mécanisme de synchronisation permettant d'établir la communication entre deux modules **Ang :** Handshake
Processus d'association	Phase de la synthèse destinée à établir les liens entre le nom des fonctions codées et les bibliothèques de composants physiques **Ang :** Linking ou Mapping
Récupération	**Ang :** Load
Reed Solomon	Nom donné à un algorithme d'encodage et de décodage de données permettant la détection et la correction d'erreurs.
Reliure	Phase de la synthèse destinée au choix des composants physiques du système **Ang :** Binding

Requête	Code indiquant une demande de transmission ou d'utilisation **Ang :** Request
Restaurer	Remise à zéro **Ang :** Reset
Sémaphore	Méchanisme de synchronisation permettant de coordonner l'accès aux ressources partagées
Socketization	Étape du design qui permet rendre un module réutilisable
Sortance	Charge sur un fil de sortie d'un composant physique **Ang :** Fanout
Synchrone	En conception et synthèse : avec une horloge En communication : communication bloquante
Tâche de fond	Dernière phase de la conception d'une puce (plan de masse, placement-routage, génération de vecteurs de test...) **Ang :** Back-end
Temps d'accès au marché	**Ang :** Time-to-market
Temps de maintien	**Ang :** Hold time
Temps de préconditionnement	**Ang :** Setup time
Thread	Tâche Légère : processus correspondant à l'exécution d'un petit programme, ou d'une routine d'un programme plus gros
Topologie	Positionnement des transistors sur un dé et construction physique par couches de métal **Ang :** Layout
Tourniquet	Algorithme d'ordonnancement où chaque processus dispose d'un quantum de temps pour s'exécuter, puis c'est au tour du suivant **Ang :** Round-robin
Tri par segmentation	**Ang :** QuickSort

LISTE DES ANNEXES

ANNEXE A : SYSTÈME CHINOOK 135

A.1. Description générale du système 135

A.2. Synthèse des communications 139

A.3. Synthèse de l'interface 142

ANNEXE B : SYSTÈME SPECSYN 150

B.1. Méthodologie de design de systèmes embarqués 150

B.2. Raffinement des variables 152

B.3. Génération de bus 152

B.4. Raffinement des communications 154

B.5. Résolution des conflits d'accès 155

B.6. Génération d'interface 155

B.7. Raffinement d'interfaces logiciel/matériel 157

ANNEXE C : SYSTÈME COOL 161

C.1. Algorithme général de développement d'interface de communication 161

C.2. Synthèse de communications dans COOL : architecture type 162

C.3. Raffinement de la logique de contrôle : conception du contrôleur système 165

C.4. Raffinement du chemin de données et synthèse de l'interface 167

ANNEXE D : SYSTÈME COSMOS 169

D.1. Description du langage SOLAR 169

D.2. Synthèse des communications dans COSMOS 171

ANNEXE E : SYSTÈME VULCAN 176

E.1. Description générale de la méthodologie de synthèse 176

E.2. Synchronisation dans un système *multi-thread* 178

E.3. Synthèse des communications pour interfaces incompatibles 180

ANNEXE F : SYSTÈME COWARE **185**

F.1. Modèle des données dans COWARE 185
F.2. Description générale du processus de synthèse 188
F.3. Sélection du mécanisme de communication 189
F.4. Implémentation des communications 192
F.5. Synthèse des interfaces 195

ANNEXE G : PROCESSEUR ARM7DTMI **199**

G.1. Signaux du ARM7TDMI 199
G.2. Diagrammes temporels du ARM7TDMI 201
G.3. Éléments de conception intéressants 203

ANNEXE H : PROTOCOLE BVCI **205**

H.1. Diagramme fonctionnel du protocole BVCI 205
H.2. Description des signaux BVCI 206
H.3. Diagramme temporel d'opérations BVCI 207
H.4. Paramètres BVCI 208

ANNEXE I : MODÈLES SEAMLESS **209**

I.1. DPRAM 209
I.2. SRAM 211
I.3. FIFO 212

ANNEXE J : COMPLÉMENTS DIVERS **214**

J.1. "Package" de base 214
J.2. Exemple de code assembleur pout les routines d'interruptions 216
J.3. Banc de test générateur de requêtes 217

INTRODUCTION

Progrès de la technologie de fabrication

La vitesse des progrès de la technologie entraîne l'augmentation rapide de la complexité des circuits intégrés. Par exemple, le passage d'une technologie de 0,5 microns à une technologie de 0,18 microns libère 88 % de l'espace sur une puce [Charl00]. La différence de densité entre les puces fabriquées aujourd'hui (technologie à 0,18 microns comportant jusqu'à 6 millions de portes), et celles fabriquées en 2002 (technologie à 0,13 microns comportant jusqu'à 25 millions de portes), est remarquable. En plus, le temps de développement d'une puce devrait passer de 18 mois à 10 mois. Le temps de conception normalisé pour une complexité fixe diminue donc de façon drastique avec les années.

Les concepteurs de circuits intégrés font face à de nombreux problèmes très souvent contradictoires. Le temps d'accès au marché, les performances (par exemple, la vitesse de fonctionnement), les coûts, la consommation de puissance, l'espace occupé et la flexibilité illustrent la divergence de ces contraintes. Les concepteurs doivent faire des compromis pour respecter ces contraintes. En conséquence, la technologie et la méthodologie de conception doivent s'adapter aux progrès de la technologie de fabrication des puces. En effet, garder la méthodologie de conception actuelle provoquerait une augmentation des coûts d'un facteur de 64 en dix ans [Charl00].

Évolution des méthodologies de conception

L'évolution de la méthodologie de conception suit un cycle. En effet, depuis les 25 dernières années, elle a changé tous les 10 ans. À chacune des étapes, il y a de nouveaux défis. Initialement, la description et la simulation du système se faisaient au niveau des portes. Cette méthodologie se nomme *Saisie et Simulation* (en anglais "Capture and Simulate"). Ensuite, les langages de haut niveau (Very High Speed Integrated Circuit Description Language (VHDL) et Verilog) ainsi que les outils de

synthèse ont vu le jour. Cela a permis d'augmenter le niveau d'abstraction de la description des circuits pour concevoir des circuits plus complexes en moins de temps. L'industrie utilise actuellement une méthodologie de *Description et Synthèse* (en anglais "Describe and Synthesize"). Cependant, les pressions du marché et l'intervalle toujours croissant entre la complexité disponible et la nécessité de réduire les temps de développement poussent l'industrie à modifier de nouveau sa méthodologie.

Pour combler l'écart entre la productivité actuelle et la productivité désirée, l'industrie doit se tourner vers le développement de systèmes sur une seule puce ("System-On-Chip" ou SoC). Un SoC constitue un circuit complexe qui intègre tous les éléments fonctionnels d'un produit sur une même puce. Les modules logiciels (DSP ou Digital Signal Processing), les mémoires, les périphériques, les coprocesseurs matériels et les modules analogues ou optoélectroniques peuvent tous être groupés sur une même puce. Ainsi, la taille des puces augmente et le nombre de composants sur la carte diminue.

Défis de la conception de SoC

La conception de SoC est encore au stade embryonnaire dans l'industrie actuelle. En effet, une nouvelle technologie de design doit toujours s'accompagner d'une nouvelle méthodologie de conception pour assurer son efficacité. Les besoins engendrés diffèrent d'un type de design à l'autre. Par exemple, l'augmentation de la proportion d'une application ou d'un produit directement placée sur la puce complexifie beaucoup la tâche des concepteurs. De plus, le nombre d'interconnexions sur une puce augmente et nécessite l'utilisation d'un bus sur la puce. Cependant, les protocoles des bus utilisés sur les cartes ne conviennent pas aux puces.

Les problèmes introduits par la nouvelle technologie de design sont nombreux. D'abord, la complexité de conception de l'architecture du système force l'intégrateur à

avoir des connaissances poussées dans plusieurs domaines de compétences. C'est-à-dire qu'il doit comprendre le fonctionnement, les paramètres et les particularités de tous les composants du système. De plus, le faible niveau d'automatisation de la tâche d'intégration augmente la possibilité d'erreurs. Il devient alors essentiel d'adapter les techniques actuelles, permettant d'effectuer la vérification du système ainsi que les tâches de fonds, à la conception de SoC. Un autre problème à résoudre est l'intégration de logiciel et de matériel.

Pourquoi la réutilisation?

Le problème actuel consiste à établir une méthodologie et à concevoir des outils qui permettront de concevoir des circuits d'une plus grande complexité en peu de temps tout en atteignant des performances et une dissipation de puissance acceptables. Toute solution viable passe par la réutilisation de modules, aussi appelés "core" ou IP (module de propriété intellectuelle). La réutilisation est essentielle au leadership d'une entreprise à court terme et à sa survie à moyen terme [CCH+99].

Néanmoins, son efficacité implique certaines conditions, tant pour la conception que pour l'intégration. D'abord, la conception et la documentation doivent se faire en fonction de la réutilisation. Ensuite, la protection et la diffusion adéquate des IP (ou modules) permettent une réutilisation au sens plus large. Le développement de méthodes simples d'assemblage et de test facilitent l'intégration des composants d'un système.

Actuellement, plusieurs chercheurs résolvent le problème ou une partie du problème avec différentes approches tout en développant leurs outils. Cependant, la réutilisation à grande échelle n'est pas une réalité. Le passage de la réutilisation de puces sur différentes cartes à la réutilisation de IP sur différentes puces est conditionnel à l'établissement de normes, ainsi qu'à l'adoption d'outils et de méthodes au niveau de l'industrie.

Définition des limites du projet

Le présent projet aborde le problème du développement d'une méthodologie de conception pour des systèmes sur une puce. Cependant, étant donné la complexité de la tâche de développement d'une méthodologie, le projet se concentre sur certains aspects particuliers. D'abord, la méthodologie générale de conception se base sur la génération de systèmes par un assemblage de différents IP sur une plate-forme variable. Ce projet ne considère qu'un seul module, un ARM7TDMI (Advanced Risc Machine). De plus, il permet de tester l'utilisation des normes de VSIA (Virtual Socket Interface Alliance) pour la conception de IP.

Alléger la tâche des concepteurs de systèmes utilisant un ARM7TDMI constitue l'objet principal du projet. Pour ce, une interface matérielle VHDL configurable pour ce processeur ainsi qu'un script sont conçus. Ainsi, à partir du code configurable, le script génère automatiquement une interface spécifique aux mécanismes de communication et de synchronisation choisis. L'utilisation du protocole VCI (Virtual Component Interface) de VSIA facilite l'établissement des communications externes du processeur. La vérification du fonctionnement de l'interface s'effectue à l'aide de plusieurs applications (multiprocesseur ou processeur/ coprocesseur). De plus, l'implantation de ces applications permet d'évaluer les pertes de performance associées à l'utilisation du protocole VCI, encore en développement, dans un contexte de communication point-à-point.

Plan du mémoire

Les chapitres deux et quatre résument l'essentiel du présent projet. La lecture des chapitres un et trois donne une compréhension plus approfondie du problème ainsi que des solutions proposées. Le premier chapitre donne un aperçu général des approches et outils développés par les différents groupes de recherche. Le deuxième chapitre présente la méthodologie de conception proposée et ses aspects. Ensuite, il détaille les caractéristiques générales de l'interface développée et précise les

particularités d'utilisation de cette interface. Le troisième chapitre détaille les techniques de conception de l'interface et justifie les choix de design pour chacune des parties du circuit. Le dernier chapitre fournit les résultats de synthèse de l'interface ainsi que les résultats de simulation obtenus pour chacune des applications testées. Les tests des performances de l'interface ainsi que celles du protocole se basent sur les algorithmes de filtrage, de tri par segmentation et de l'encodeur/décodeur *Reed Solomon*. L'évaluation des performances du protocole VCI s'effectue dans un contexte de communication point-à-point.

Pour compléter le mémoire, plusieurs annexes sont disponibles. Les annexes A à F présentent la revue de littérature initiale. En effet, cette revue ne relate qu'une partie des recherches sur le développement de SoC. Étant donné sa pertinence au présent projet, le chapitre un n'en fournit qu'un résumé et les annexes la détaillent. Le reste des annexes constitue un complément d'informations (par exemple sur le processeur ou le protocole) à consulter au besoin.

Chapitre 1 : REVUE DES MÉTHODOLOGIES DE CONCEPTION DE SoC

La progression de la technologie de fabrication de puce force l'industrie à sans cesse développer de nouvelles techniques et méthodologies de conception ainsi que de nouveaux outils. Après les approches "*Capture and Simulate*" et "*Describe and Synthesize*", le marché se dirige vers la conception de SoC et la réutilisation d'IP. Cependant, aucune méthodologie, aucune norme et aucun outil n'ont été acceptés par la plupart des compagnies.

Selon [GDG00], il existe trois grandes approches, chacune faisant un compromis entre le coût, la flexibilité, la qualité des résultats, la complexité et le degré d'automatisation. La première consiste à faire la synthèse et le raffinement d'une spécification de haut niveau ("*Specify, Explore and Refine*"). Le haut degré d'automatisation de cette approche entraîne souvent le sacrifice d'un des critères essentiels, comme la qualité des résultats obtenus. Les compilateurs comportementaux l'illustrent bien. La deuxième consiste à configurer une plate-forme de base en fonction d'une application ("*Configure and execute*"). Son faible coût est associé à un manque de flexibilité. L'appariement de différents IP pour construire une application ("*Mix and match*") constituent l'essence de la troisième approche. Malgré sa flexibilité accrue, son faible degré d'automatisation peut la rendre moins attrayante.

Pour obtenir le meilleur compromis, les concepteurs utilisent souvent une combinaison d'approches. Le classement des solutions des différents groupes de recherche ne suit donc pas strictement ces catégories. La première section du présent chapitre fait une revue des méthodologies de synthèse et de raffinement des systèmes logiciels/matériels. Le reste du chapitre décrit les différents aspects de méthodologies portant strictement sur la conception de systèmes sur une puce. D'abord, la section deux explique les architectures et les normes des SoC. Puis, la dernière section énonce les possibilités de configuration et de réutilisation d'IP.

1.1. Synthèse et raffinement d'une spécification de haut niveau

L'augmentation du niveau d'abstraction de la description du système permet aux concepteurs d'améliorer leur productivité. Il faut utiliser un langage de haut niveau pour décrire le comportement des différents modules ainsi que leurs communications. Puis, l'implantation physique s'obtient après la division, le raffinement et la synthèse du système. L'avantage de cette approche réside dans l'automatisation des différentes étapes. La nature même du projet amène une analyse concentrée plus particulièrement sur la synthèse des communications.

Les méthodologies de synthèse d'interface logicielle/matérielle présentées ne s'appliquent pas directement à la conception de SoC. Cependant, plusieurs parties de ces méthodologies sont pertinentes à la conception de SoC. D'ailleurs, plusieurs groupes de recherche sur la conception se tournent aussi vers le design des SoC. Cette section résume d'abord les outils Chinook, SpecC, Cool, Cosmos et Vulcan, puis présente d'autres méthodologies.

1.1.1. Chinook

Chinook est un outil axé sur la synthèse d'interface de systèmes dominés par les signaux de contrôle [BCO96] [OrBo97] [OrBo98] [COB92] [COB95a] [COB95b] [COH+99] [ChBo94],[WaBo94]. L'accent est plus particulièrement mis sur deux aspects du problème de synthèse d'interface. Le premier permet l'association des ports d'un processeur avec les ports de ses périphériques. Cela permet donc la génération de l'association des ports, des routines logicielles permettant de contrôler les périphériques ainsi que des modules matériels (séquenceurs) lorsque la communication ne peut se faire directement. L'algorithme proposé utilise d'abord les ports réservés aux entrées et sorties du processeur, puis les ports d'accès mémoire.

Le deuxième aspect traité est celui des communications entre plusieurs processeurs. La communication entre les processeurs peut se faire de façon directe ou par le biais

d'autres composants (liaison à multiples bonds). Dans ce cas, des modules logiciels (gestionnaires) permettent le transfert des données entre les processeurs.

Il n'y a qu'un seul environnement de simulation et un algorithme d'ordonnancement logiciel est disponible. Aucun partitionnement n'est effectué (il doit être effectué au préalable) et des bibliothèques doivent être montées par le concepteur avant tout processus de synthèse. En plus des différentes bibliothèques, (processeurs, périphériques, composants de communications), une description fonctionnelle du système sous forme de CFG (Control Flow Graph ou diagrammes de contrôle) est nécessaire. Ces graphes devront contenir l'information sur le partitionnement du système. Pour plus d'informations, voir l'annexe A.

1.1.2. SpecC

La méthodologie de conception SpecC [GZGH00] [DoGa00] [GDG00] [KlGa98] est une extension de SpecSyn [NaGa94] [DGZ98] [GGB97] [GaRa94] [NVG91] [GVNG94] [NaGa95] [GaVa95] pour des applications SoC. Un résumé de SpecSyn est donné à l'annexe B, les principes de base, se retrouvant aussi dans SpecC, y sont décrits. La méthodologie SpecC est beaucoup plus raffinée que celle de SpecSyn, de plus les concepts d'IP et de réutilisation y sont introduits. Une spécification abstraite du système, décrite avec le langage SpecC, est raffinée jusqu'à l'obtention d'une implantation physique. Les étapes sont les suivantes : spécification, analyse et estimation, exploration architecturale et synthèse des communications.

Les deux dernières étapes méritent d'être détaillées. La phase d'exploration de l'architecture est divisée en trois étapes : l'allocation, le partitionnement et l'ordonnancement. D'abord le nombre et le type d'éléments de traitement (PE), de mémoires (MU) et de bus (CU) sont déterminés. Cette phase d'allocation peut être faite à partir d'une bibliothèque d'IP (PE, MU ou CU).

Ensuite, le partitionnement du système sur les différents PE est effectué. À cette étape, certains éléments (PE ou CU) peuvent être ajoutés pour assurer l'intégrité de la synchronisation et la communication entre les blocs. L'étape suivante est le partitionnement et l'association des variables aux différents éléments de mémoire. Le partitionnement des canaux de communications consiste ensuite à construire le réseau complet de communication. Une fois le partitionnement terminé, l'ordonnancement statique ou dynamique est possible.

L'étape suivante est la détermination de l'architecture et la synthèse des communications. Cela correspond à l'insertion de protocole et au raffinement des communications. Le comportement et l'interface doivent être séparés. Il est possible de générer une enveloppe pour chaque module qui fait la conversion de protocole et permet un haut niveau de flexibilité. Ainsi, tout module synthétisable (Figure 1.1.a) ou canal virtuel (Figure 1.1.b) peut être remplacé par un IP. L'outil permet de générer et d'utiliser des IP décomposés en une partie publique et secrète assurant ainsi leur protection.

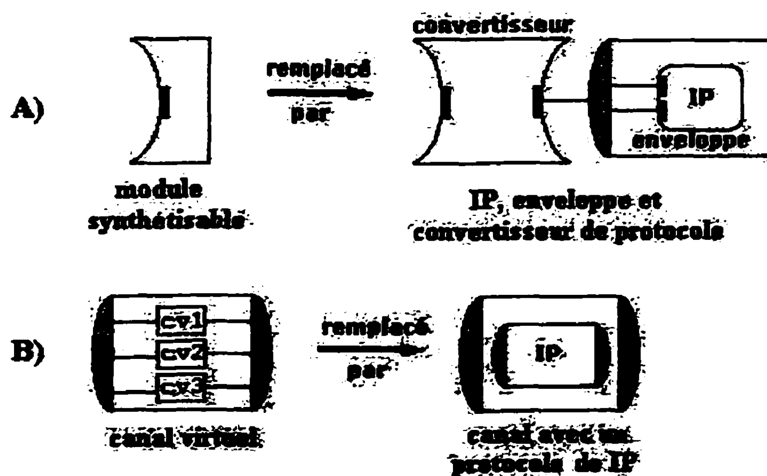


Figure 1.1. Génération d'enveloppes d'IP proposé dans [DoGa00]

1.1.3. COOL

COOL [NiMa98] permet la synthèse de système utilisant une architecture bien définie comprenant un contrôleur général du système, un contrôleur pour chacun des modules communicants ainsi qu'un contrôleur pour les entrées/sorties. La méthodologie est très flexible. Elle permet tous les types de synchronisation et de communication. L'architecture s'applique davantage aux systèmes dominés par les signaux de contrôle puisque la communication directe entre les modules n'est pas implicite. Les principales étapes de synthèse sont semblables à celles de l'approche de SpecSyn.

L'ordonnancement des communications est effectuée à l'aide d'un modèle de lecture/écriture. Les communications peuvent être bloquantes ou non bloquantes selon le schéma choisi. Les modules sont synchronisés par des machines à états qui leur indiquent quand réagir. Les états sont calculés par le contrôleur général ou par les contrôleurs de modules. Un algorithme de génération de machine à états pour la synchronisation est donné. Pour plus d'informations, voir l'annexe C.

1.1.4. COSMOS

COSMOS [ObJe92] [DIMJ97] [DMVJ97] [VCV+95] [VIJK94] [HCM+99] utilise une représentation intermédiaire (SOLAR) pour faire le raffinement d'interfaces. Cette représentation permet l'utilisation de plusieurs modèles de communication. La communication est par défaut bloquante, mais des queues peuvent être ajoutées pour la rendre non bloquante. Trois sortes d'éléments composent un système : unité de communication, unité fonctionnelle et machines à états. La description fonctionnelle du système se fait donc à partir de machines à états.

La communication, la synchronisation et le traitement des données se font par les unités de communications. Les RPC (Remote Procedure Call) ainsi que des fonctions spéciales sont utilisées pour incorporer les détails des protocoles. La synthèse est en fait un problème d'allocation nécessitant des bibliothèques (unités de communications

et implémentations de ces unités) et une fonction de coût. Le grand désavantage de cette approche est que les détails d'implémentation sont compris dans les bibliothèques qui sont construites par le concepteur. Pour plus d'informations, voir l'annexe D.

1.1.5. VULCAN

L'architecture utilisée par VULCAN [GCM92] [CLG96] [GuMi97] [MiGu93] [LiGu98] est composée d'un processeur, d'une mémoire et de plusieurs ASIC (Applicattion. Le processeur est le maître et il contrôle le bus et la mémoire. Le modèle permet l'implantation de programmes à plusieurs "thread" (tâche légère). Les transferts des données entre les composants se font toujours à travers le bus mais une queue est souvent insérée pour mémoriser les données en attendant la disponibilité du processeur. La communication non bloquante est favorisée. La synchronisation se fait par passage de messages et par interruptions. En effet, le maître envoie des signaux de contrôle aux autres composants et ceux-ci peuvent l'interrompre avec les interruptions.

Deux aspects de la synthèse de communications sont abordés. En premier lieu, la génération de mécanismes de synchronisation pour un système à plusieurs "thread" permet un ordonnancement dynamique sans préemption. Une FIFO permet de mémoriser l'ordre d'exécution des "thread". En deuxième lieu un algorithme de conversion de protocole permet d'identifier les ports maîtres qui pourront générer les signaux voulus sur les ports esclaves. Cependant, aucune logique séquentielle ne doit être nécessaire, les données doivent donc être de même grandeur et le taux de transfert, le même des deux côtés. Pour plus d'informations, voir l'annexe E.

1.1.6. Autres méthodologies

1.1.6.1. LYCOS

Plusieurs autres groupes de recherche se sont penchés sur le problème de la synthèse des communications. LYCOS [KnMa98] [KnMa99] [MaHa95] est un outil axé sur le partitionnement logiciel/matériel. Les estimateurs de communication sont bien développés et la sélection du protocole de communication se fait à l'étape du partitionnement. Les données circulent à travers trois étages : le pilote de transmission, le canal puis le pilote de réception (Figure 1.2). L'estimation du délai est basée sur un pipeline construit en étages. Les transmissions par paquet avec ou sans division ou regroupement sont considérées. Des estimateurs d'espace de pilotes (en matériel et en grosseur de code) et de canaux sont développés.

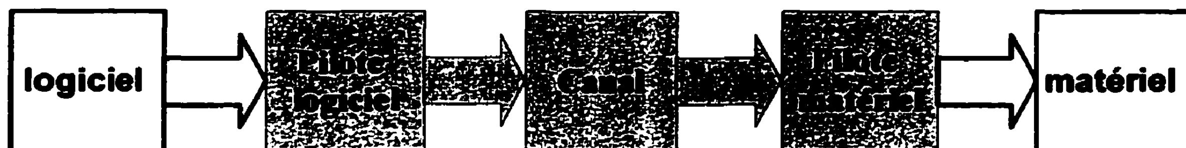


Figure 1.2. Canal de communication selon Madsen [KnMa98]

1.1.6.2. Polis

L'algorithme de synthèse d'interface entre modules ayant des protocoles incompatibles, utilisé dans l'outil Polis, est décrit dans [PRS98]. Le module d'interface est divisé en deux parties : une machine à états, permettant de gérer les signaux de contrôle, et un module contenant le chemin de données. Le module de chemin de données est très simple car les données sont de même grandeur et tout le système utilise la même horloge. La conversion de protocole se fait à partir de la description des protocoles avec un langage particulier et la méthode des graphes est utilisée pour générer la machine à états.

1.1.6.3. Sierra

La méthodologie, utilisé dans l'outil SIERRA, est expliquée dans [SuBr92]. Le rôle de l'interface est d'établir un chemin physique pour les données, de contrôler les séquences de transfert et de synchroniser les transferts. Un graphe de flux est généré à partir des informations sur les protocoles contenues dans des bibliothèques. Puis, un ordonnancement et une allocation des ressources permettent un raffinement de ce graphe pour obtenir une description RTL (Register Transfer Level) du module de communication. La plate-forme, utilisée pour construire le convertisseur de protocole, est composée de trois modules : deux contrôleurs et un chemin de données (Figure 1.3).

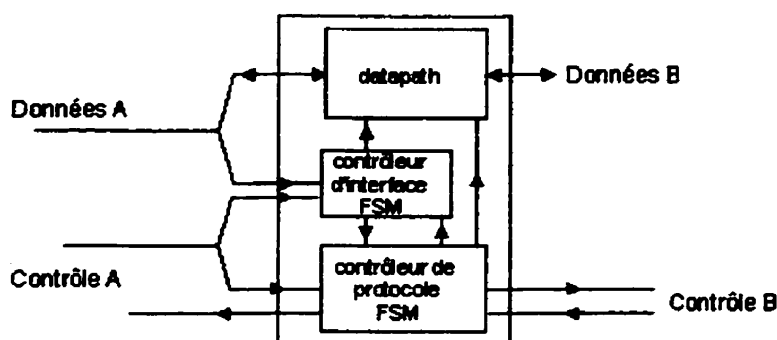


Figure 1.3. Architecture du module d'interface utilisé dans [SuBr92]

1.1.6.4. Synthèse d'interface par analyse des séquences de communication

Une méthodologie plus complète de synthèse d'interface pour des systèmes hétérogènes est donnée dans [BPM98]. Le système est d'abord divisé, puis les modules logiciels sont compilés pour un processeur cible et la synthèse des bus est effectuée. Il est alors possible d'extraire l'information sur les séquences de communication et les protocoles du processeur. Cette information est utilisée pour générer les unités de communication. La dernière étape consiste à effectuer la synthèse du matériel comprenant les unités de traitement et les mémoires. L'architecture type est constituée d'un processeur et d'un ASIC (Figure 1.4).

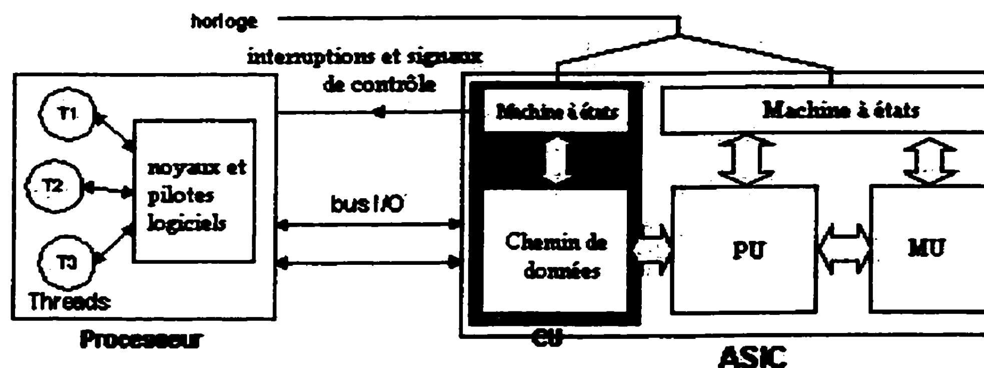


Figure 1.4. Architecture type du système proposé dans [BPM98]

1.1.6.5. Synthèse d'interface basée sur la minimisation des délais et de l'espace

Une méthodologie de design, basée sur la minimisation des délais et de l'espace des communications, est proposée dans [GABP98]. À partir d'un graphe des communications entre les processus, une priorité est assignée à chaque événement et les transferts qui ne pourront pas être exécutés de façon synchrone sont identifiés. Ces transferts nécessitent une FIFO qui ralentit le temps de communications et augmente l'espace occupé. Ensuite, l'architecture de base est déterminée en identifiant le type de transfert, les ressources de communications (FIFO et bus) et leur protocole. Les opérations exclusives dans le temps sont associées aux mêmes ressources et des composants de contrôle sont ajoutés.

1.1.6.6. Génération d'interface pour simulation de systèmes

Dans [GuRo94], la méthodologie de conception de modules d'interface permet la simulation et la validation de la fonctionnalité et des délais du système durant les premières phases de design. Le système est décrit en VHDL et les communications sont établies par l'intermédiaire de modules d'interface. Chaque module d'interface permet l'utilisation d'un groupe de ports ayant une fonctionnalité commune (comme les ports d'adresses et de données mémoire).

Le module d'interface est divisé en deux parties (Figure 1.5). La première partie (synchrone et synthétisable) permet d'effectuer les transformations et les transmissions de données et supporte le protocole interne de communication. La deuxième partie est constituée d'un ensemble de procédures permettant d'exprimer les contraintes du protocole externe avec des délais très fins ou même des comportements complètement asynchrones. Les appels à ces procédures sont synchronisés avec la partie interne pour provoquer une communication externe lorsque désiré.

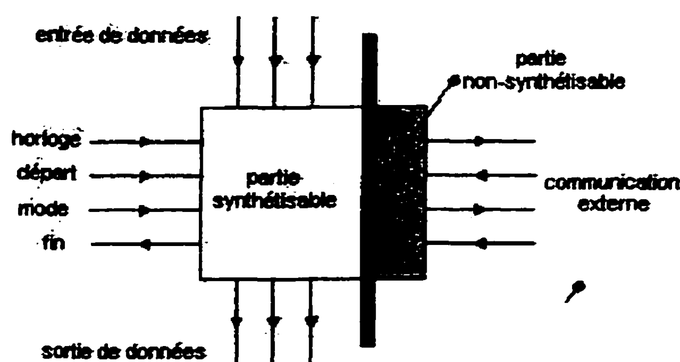


Figure 1.5. Interface à plusieurs niveaux d'abstractions

1.1.6.7. Synthèse de contrôleur asynchrone

Jusqu'à présent, les systèmes traités étaient, pour la plupart, synchrones. Cependant, les modules d'un SoC peuvent avoir des horloges différentes. Dans ce cas, il n'est pas évident de développer des interfaces de communication synchrones. Une communication synchrone est d'ailleurs souvent plus lente [BNY99]. Dans [KCK+99], l'utilisation d'un contrôleur asynchrone permet de diminuer la taille de celui-ci tout en augmentant ces performances. De plus, une solution semi-synchrone peut permettre de régler les problèmes de biais dans de gros circuits.

Cependant, le problème des circuits asynchrones est l'interblocage. En effet, une horloge globale est très utile à la filtration des effets de hasards. De plus, la conception de circuits synchrones est beaucoup moins complexe. Il n'en reste pas

moins que lorsque les contraintes sont très strictes, les circuits asynchrones peuvent être la seule solution.

Une méthodologie de synthèse de contrôleur d'interface asynchrone est présentée, dans [KCK98], à l'aide d'un exemple où un périphérique transfère des données sur le bus asynchrone. À partir de l'information contenue dans les diagrammes de temps, un réseau de pétri, puis un graphe d'états sont construits. Chaque nœud représente un état et chaque flèche, une transition d'état. Une méthode de synthèse des graphes permet de dériver une fonction déterminant l'état suivant.

1.2. Architecture et normes de SoC

La plupart des méthodologies de conception SoC utilisent une combinaison des approches de configuration de plate-forme et d'assemblage de IP. Le concepteur emploie une approche plate-forme [TaKn00] s'il débute avec une architecture de base dont il modifie les paramètres pour l'adapter à une application. Cependant, avoir une seule plate-forme pour tout type d'applications diminue les performances des systèmes. Ceci nécessite de définir une plate-forme pour chaque type d'applications.

D'un autre côté, un concepteur qui combine et apparie différents IP pour construire un système utilise une approche d'assemblage d'IP. Cependant, cette approche nécessite l'utilisation d'une architecture de base. De plus, un IP configurable offre une plus grande flexibilité. Les solutions proposées associent souvent ces deux approches complémentaires. D'ailleurs, les termes "IP-based design" et "Platform-based design" désignent un même type de méthodologie.

Un aperçu des architectures et des normes ainsi qu'une description des principes de réutilisation résument l'essence de ce type de méthodologie. Cette section décrit les architectures et les normes en présentant les types de bus, de plates-formes et de

protocoles ainsi que leur méthodologie de conception de SoC. En outre, on accorde une attention particulière à l'alliance VSI.

1.2.1. Bus et plate-forme

Un SoC est constitué d'IP qui sont assemblés suivant une architecture particulière avec un ou plusieurs bus appelés OCB (*On-chip Bus*). Sur une puce, contrairement aux cartes où un seul bus à fonctions multiples (par exemple, le bus PCI ou Peripheral Component Interconnect) est utilisé, plusieurs types de bus à fonction unique sont employés. De plus, les OCB proposés ont seulement deux états permettant une communication point-à-point multiplexée, plutôt que d'avoir un bus à trois états. Dans la majorité des architectures, il y a un bus à haute performance, un bus plus lent pour les périphériques et quelques fois un bus local au processeur.

Plusieurs organisations proposent des plates-formes ainsi que leurs OCB. Dans certains cas, l'architecture proposée vient avec une bibliothèque d'IP et même un outil d'intégration. Cette diversité permet le choix du bus et de l'architecture qui convient le mieux au système à concevoir. Trois de ces architectures sont présentées.

1.2.1.1. CoreConnect

La compagnie IBM [BeLe00] propose une architecture de système sur une puce ainsi qu'un OCB. L'architecture (ou bus) s'appelle "CoreConnect" (Figure 1.6). En plus d'une architecture de base, IBM détient une bibliothèque d'IP appelée "Blue Logic Core Library". À partir de l'architecture de base ainsi que la bibliothèque d'IP, l'outil "Coral" permet l'automatisation de la tâche d'intégration d'IP.

D'abord, l'architecture utilisée mérite des explications supplémentaires. Elle est composée de trois éléments de base. Le premier bus (PLB ou Processor Local Bus) est local au processeur et lui permet de communiquer avec ses mémoires. Ce bus est particulièrement adapté au processeur PowerPC. De plus, ce bus à grande bande

passante (256 bits à 133MHz) contenant un arbitre complexe peut être utilisé pour des applications à hautes performances.

Le second composant est le bus de périphérique (OPB). Sa bande passante est beaucoup moins élevée (32 bits à 66MHz) et son arbitre moins complexe. Le dernier composant est un pont ainsi qu'un bus de contrôle de registre (DCR ou Device Control Register) : le pont permet de lier les deux bus précédents et le DRC sert à transférer des informations sur la configuration et les états des composants du système sans affecter ses performances.

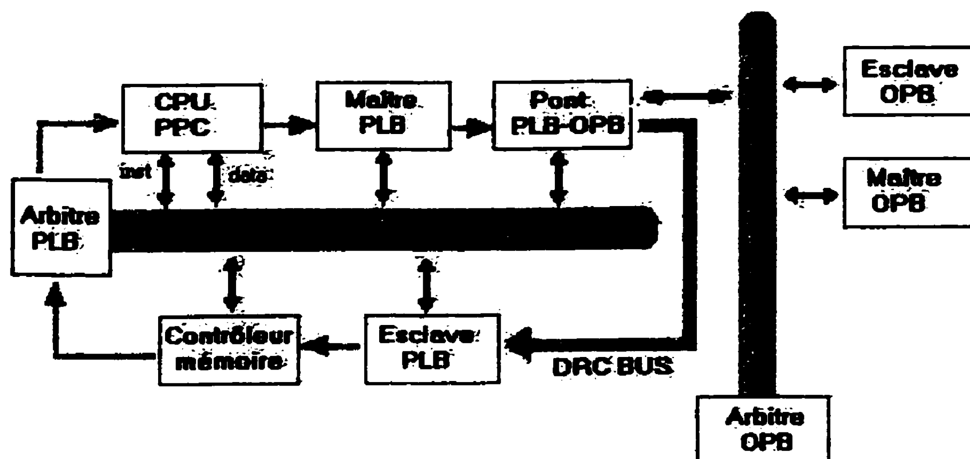


Figure 1.6. Architecture "CoreConnect" d'IBM

À partir de cette architecture de base, "Coral" permet d'intégrer tous les éléments du système automatiquement. En élevant le niveau d'abstraction, il est possible de diminuer le niveau de détails de description des IP et de leurs ports. L'idée est de regrouper les ports d'IP, appelés composants virtuels (VC), ayant une fonction commune pour diminuer le nombre de connexions. Ainsi, une interface virtuelle, conçue pour chaque VC, simplifie la compréhension du concepteur.

Certains paramètres et propriétés sont associés à chacune des interfaces et donc, les IP peuvent être configurés. Un raffinement de la description virtuelle du circuit permet la génération du système réel, ayant une architecture du type "CoreConnect". Pour utiliser des IP qui ne sont pas dans la bibliothèque, une enveloppe doit être crée autour de chaque IP et des paramètres doivent lui être associés.

1.2.1.2. CoreFrame

PalmChip [Ditt00] propose une seconde architecture appelée "CoreFrame". L'avantage de cette architecture est que le processeur a un bus local qui ne peut être accédé par d'autres maîtres (Figure 1.7 sous-système du CPU (Control Processong Unit)). Le Mbus est le bus le plus performant (100MHz) et sert à connecter le processeur avec les périphériques maîtres. Sur ce bus, il n'y a qu'un seul esclave : le contrôleur d'accès mémoire (MAC). Plusieurs types de mémoires sont supportés. Les canaux de connexion sont individuels et les adresses et données sont multiplexées. L'arbitre est invisible. Le PalmBus est moins performant, plus simple et permet au processeur de communiquer avec les autres esclaves du système.

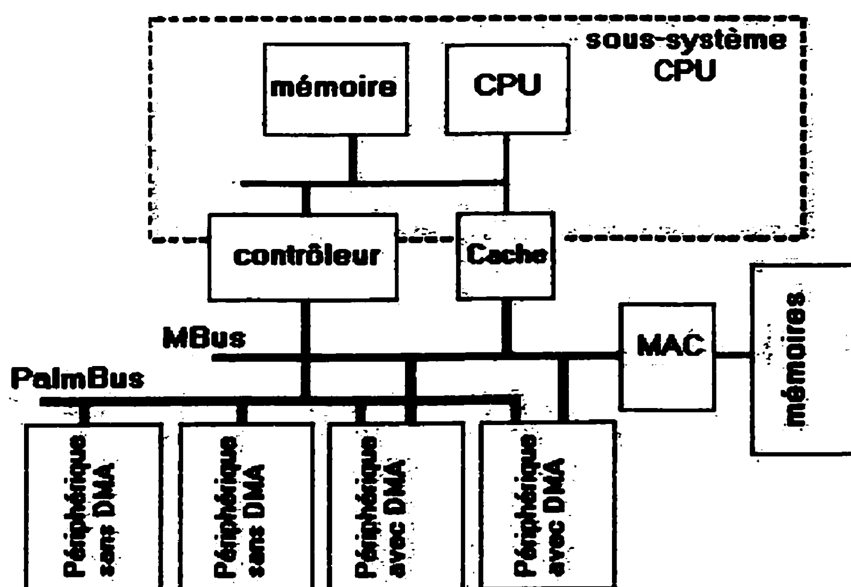


Figure 1.7. Architecture "CoreFrame" de PalmChip

Pour permettre d'intégrer les IP ne communiquant pas avec le protocole du bus, un outil de conversion de protocoles est disponible. Cet outil contient une bibliothèque de protocoles couramment utilisés pour la conception de SoC. Si l'IP utilise un des protocoles de la bibliothèque, la conversion se fait automatiquement en identifiant des paramètres. Cependant, si le protocole est inconnu, l'outil guidera le concepteur pour la construction de l'interface. Toutefois, une grande partie de cette tâche doit être faite manuellement.

1.2.1.3. AMBA

La compagnie ARM [ARM00a] propose aussi son architecture de SoC AMBA (Advanced Microcontroller Bus Architecture) bien adaptée à ces processeurs (Figure 1.8). Elle est composée de trois bus : deux bus à haute performance et un bus pour les périphériques plus lents. En plus, une bibliothèque de périphériques (appelée "PrimeCells") est disponible pour cette architecture. Des périphériques sont disponibles pour les trois types de bus.

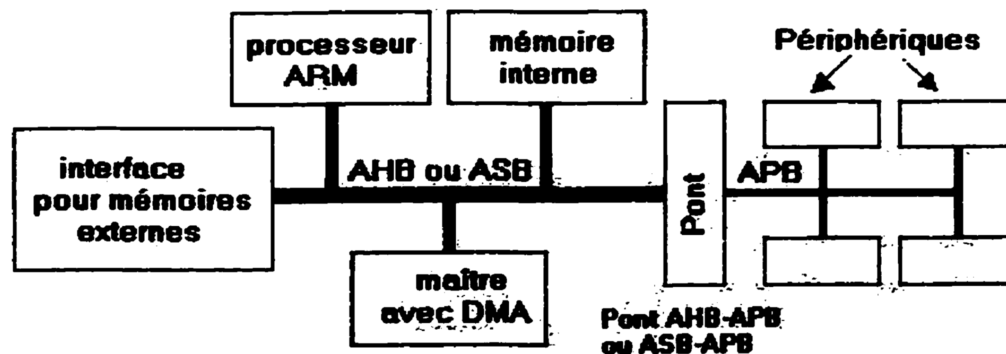


Figure 1.8. Architecture AMBA de ARM

Le premier bus ("Advanced High performance bus" ou AHB) est un bus de très haute performance utilisé pour connecter le processeur avec ses mémoires, ainsi qu'avec une interface pour les mémoires hors puce et les périphériques nécessitant des transferts à haute vitesse. Un arbitre est disponible pour les configurations à plusieurs

maîtres. Le pipeline du bus permet des transferts par paquets ainsi que des transactions divisées.

Le deuxième bus ("Advanced System Bus" ou ASB), lui aussi à haute performance, est très semblable au premier, mais il est moins rapide et possède une fonctionnalité restreinte. En particulier, il ne permet pas les transactions par paquets ou divisées. Le dernier bus ("Advanced Peripheral Bus" ou APB) est celui des périphériques lents. Il permet de réduire la consommation de puissance et la complexité des interfaces lorsque possible. Un pont permettant de lier les bus est disponible.

1.2.2. Protocoles

Comme expliqué dans [Mert00], définir une norme OCB, comme dans le cas du bus PCI sur cartes, n'est pas idéal en ce qui a trait au design SOC. L'utilisation de différents bus permet le choix d'une architecture adaptée au système à concevoir. Cependant, il n'est pas pratique de devoir construire une nouvelle interface pour un IP chaque fois que l'on change l'OCB. En effet, les possibilités d'exploration d'architecture sont très limitées si les IP sont tous conçus pour ne communiquer qu'avec un seul bus. De même, la tâche d'intégration est très laborieuse lorsque l'on construit un système avec des IP qui ne sont pas tous adaptés au même OCB.

Pour solutionner ce problème, certains organismes proposent des protocoles de communication plutôt que des bus. De cette façon, les IP communiquent tous avec un même protocole et selon l'OCB choisi, les connexions (Figure 1.9) sont effectuées à l'aide d'un pont (le même pour tous les IP). Si le fournisseur du bus donne aussi le pont et que le concepteur du IP développe l'interface communiquant avec le protocole standard, la tâche de l'intégrateur est réduite à l'assemblage des différents composants. Cette solution implique que toute l'industrie s'entende sur un seul protocole standard pour les IP. Pour le moment, plusieurs de ces protocoles sont en développement.

Figure 1.9. Utilisation d'un pont pour connexion au OCB

1.2.2.1. General Bus Interface

Le “General Bus Interface” (GBI) est proposé dans [LKSN99]. C'est un protocole avec 32 bits d'adresse et 32 bits de données. Il supporte les transferts par paquets avec des paquets de différentes tailles. Ce protocole est très semblable au IP bus (de OMI) et donc est très semblable à un protocole de bus sur le plan de la complexité. Le protocole a été testé avec quatre OCB : AMBA (de ARM), FISbus (de MentorGraphics), PI bus (de OMI) et PLB (de IBM). Le problème associé à un protocole complexe et peu flexible est que plus le protocole du bus utilisé en diffère, plus le coût du pont (quant aux délais et à l’espace) est important.

1.2.2.2. OpenCore Procotol

Le “OpenCore Protocol” (OCP) est proposé dans [Smit00]. Ce protocole est moins complexe et plus flexible que le précédent. Il est constitué de signaux de base pour une communication simple entre un maître et un esclave (basée sur des transferts de données simples). En outre, une extension aux signaux de base est disponible pour les IP plus complexes. Ces signaux sont typiquement des signaux de contrôle (interruptions, erreur, signaux d'états) et des signaux de tests (scan, JTAG, contrôle d'horloge). Ce protocole a été testé dans des systèmes utilisant les bus AMBA et “CoreConnect”.

1.2.2.3. Autres protocoles

Un autre protocole de ce type est utilisé par l'outil de Coware, c'est le "Virtual Bus". Ce protocole permet de définir les communications d'une manière beaucoup plus abstraite et ainsi simplifier la tâche du concepteur. Plusieurs schémas de communication y sont disponibles. L'outil de Coware sera détaillé plus loin.

Le dernier protocole à décrire est le protocole VCI de VSIA. C'est un protocole axé sur le transfert de données et qui ne contient pas implicitement de signaux de contrôle. Ce protocole offre cependant beaucoup de flexibilité au niveau des transferts de données, mais est limité à un seul schéma de communication : la poignée de main. Puisque ce protocole sera utilisé tout au long du projet, la section suivante est entièrement consacrée aux normes proposées par VSIA et la section 2.2.3 fournit une description plus précise du procotole VCI.

1.2.3. VSIA

L'alliance VSI ("Virtual Socket Interface") a été formée en septembre 1996. L'objectif est de promouvoir une vision unifiée du design de SoC à travers l'industrie et d'établir des normes techniques facilitant l'intégration des IP sur une puce. L'objectif visé n'est pas de régler tous les problèmes reliés au design SoC, mais plutôt de fournir les normes nécessaires à l'industrie pour qu'elle puisse le faire.

L'alliance compte 8 groupes bénévoles de développement dont les membres viennent de plus de 100 compagnies différentes (fournisseurs d'outils, concepteurs d'IP et concepteurs de systèmes). Les groupes de développement sont les suivants : "Analog/Mixed Signals", "Implementation Verification", "Functional Verification", "System-Level Design", "On-Chip Bus", "Manufacturing Test", "Virtual Component Transfer" et "IP Protection". Dans [Mert00] et [BiSs99], une vue d'ensemble des normes et groupes de VSIA est fournie. La mission des deux groupes pertinents au présent projet est décrite aux paragraphes suivants.

1.2.3.1. Groupe de conception au niveau système

Le rôle du groupe de conception au niveau système [VSIA00b] est d'identifier une norme pour faciliter la spécification, l'évaluation et l'intégration d'un système durant la phase d'exploration et de raffinement de l'architecture. Il est donc essentiel, dans un premier temps, de définir un vocabulaire commun. C'est pourquoi il existe un document appelé "Taxonomy" qui définit et classifie les objets et principes ainsi que leurs attributs. De plus, une norme de type de données pour C/C++ ("Data Type Standard") est disponible, comme le module IEEE 1164 en VHDL.

Ensuite, la norme d'interface au niveau système (SLIF ou System Level Interface Standard) définit une structure de niveaux d'abstraction utile pour le raffinement du système (particulièrement des communications) ainsi que pour la documentation de l'architecture à chacun des niveaux. Il est donc possible de faire un raffinement hiérarchique en partant d'un niveau d'abstraction élevé (*layer 1.0*) jusqu'à l'implantation physique (*layer 0.0*). La Figure 1.10 donne un exemple de raffinement partant du niveau de l'application jusqu'à l'implantation physique. Le protocole utilisé au niveau physique est VCI.

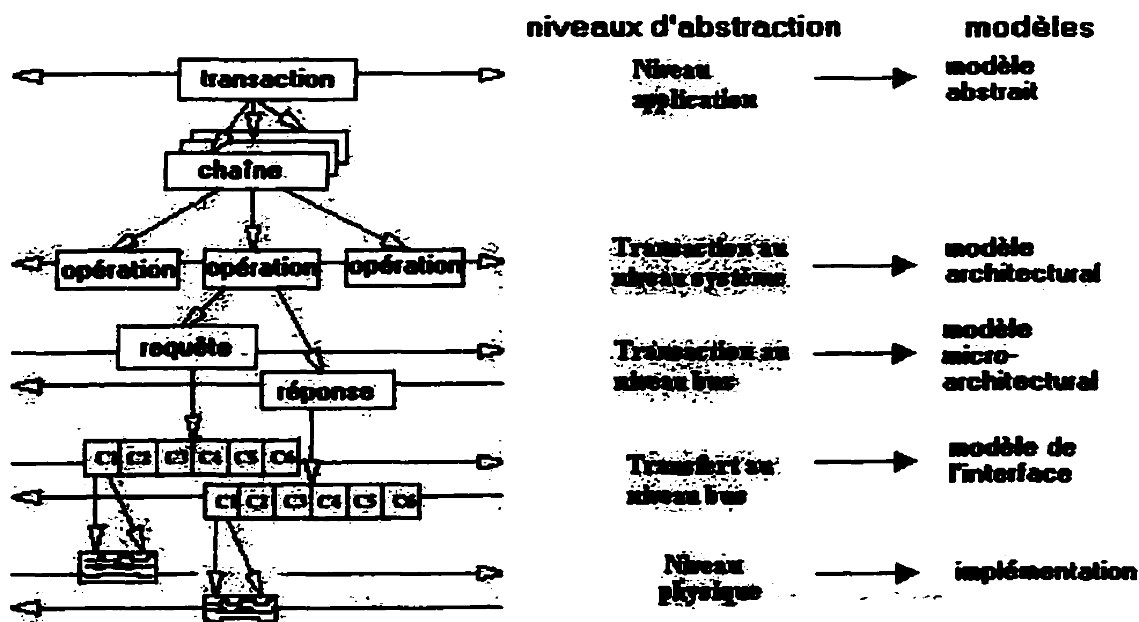


Figure 1.10. Raffinement hiérarchique proposé par VSIA

La structure de chacun des objets (blocs) du système doit être conçue de sorte que le comportement et les communications soient séparées pour ainsi augmenter la flexibilité. Des attributs sont associés à chacun des objets selon le niveau d'abstraction dans lequel ils sont représentés. Des transactions ainsi que leurs attributs et représentations sont définis pour chacun des niveaux d'abstraction.

1.2.3.2. Groupe du bus sur une puce

L'objectif du groupe de OCB [VSIA00a] est de définir une norme à un niveau d'abstraction très bas facilitant l'intégration d'IP sur une puce. Puisqu'il n'est pas réaliste de définir un bus standard [Mert00], un protocole de communication pour les IP indépendants du OCB a été défini. Tout comme ceux énoncés à la section 1.2.2, le protocole VCI permettra de construire plus facilement tout un système à partir d'IP et de ponts (Figure 1.9).

Ce protocole permet une communication unidirectionnelle point-à-point entre un initiateur et une cible (maître et esclave). La requête (envoyée par l'initiateur) et la réponse (envoyée par la cible) sont contrôlées par un protocole de poignée de main. Pour communiquer avec un bus, une communication point-à-point doit être établie entre le IP et un pont. Les paramètres de l'interface VCI peuvent être modifiés. De plus, on offre trois niveaux de norme : "Peripheral VCI", "Basic VCI" et "Advanced VCI". Ces trois niveaux sont comparables aux niveaux de bus proposés par ARM. Le protocole, faisant partie de la méthodologie proposée, est détaillé au chapitre deux.

1.2.3.3. Analyses antérieures du protocole VCI

Le protocole étant utilisé et analysé à l'intérieur du présent projet, il convient de revoir les analyses antérieures de ce protocole. Puisque la norme est encore en développement, peu de résultats ont été publiés.

La première étude a été menée en collaboration avec Coware et Nokia [LSJ+00]. Un module utile dans des systèmes de communication mobile a été conçu en effectuant un raffinement comme proposé par SLIF et en utilisant le protocole VCI. En parallèle, le même système a été raffiné en utilisant l'outil Coware (section 1.3.4). Les performances du protocole n'ont pas été étudiées. Cependant, il a été montré que les normes de VSIA facilitent l'intégration de VC (Virtual Component) et le raffinement des systèmes.

Dans la seconde étude [CCS99], l'analyse les performances du protocole (*Basic VCI*) est faite en deux étapes. Dans un premier temps, un pont permettant de lier un esclave VCI au bus ASB d'ARM (Figure 1.8) est construit. Le pont compte de 500 à 1000 portes logiques selon la complexité de l'interface et aucune précision sur la latence n'est donnée. Dans un second temps, une interface VCI pour un contrôleur d'interruptions est générée. Dans ce cas, le temps de communication est doublé par l'utilisation du double protocole de poignée de main de VCI.

Une analyse des performances du protocole "Peripheral VCI" est donnée dans [LVG00]. Six implantations du même système sont utilisées (trois avec le bus ISA (Industry Standard Architecture) et trois avec un autre bus standard). Dans la première implantation, le protocole est directement intégré aux blocs. Dans la deuxième, une enveloppe VCI est utilisée pour lier les blocs au bus. Dans la dernière, une enveloppe générique est utilisée. Les résultats obtenus avec une enveloppe générique et une enveloppe VCI sont très semblables. L'utilisation d'une enveloppe entraîne une légère augmentation de la puissance, causée par le double transfert de chacune des données, et une légère augmentation du nombre de portes (environ 1500 portes). La latence n'est pas augmentée dans le cas du bus ISA (bus très lent) et est légèrement augmentée dans le cas du bus standard.

1.3. Possibilités de réutilisation des IP

Le design de SoC nécessite la réutilisation des IP. Quels sont les critères de réutilisation et comment concrètement peut-on bâtir un système à partir d'IP ? Pour répondre à ces questions, les principes de réutilisation sont d'abord présentés. Puis, les approches de design SoC de trois compagnies spécialisées en développement d'outils sont expliquées. Les outils sont présentés en ordre décroissant de flexibilité et l'augmentation de la flexibilité amène une diminution de l'automatisation.

Il faut noter que, même si notre partenaire industriel, Mentor Graphics, est spécialisé en développement d'outils, aucun outil de design SoC à proprement dit n'existe pour le moment. Néanmoins, une bibliothèque d'IP est disponible et s'appelle "Inventra Portefolio". De plus, l'outil de covérification Seamless est très populaire. En outre, le projet consiste à fournir des pistes à notre partenaire en proposant une méthodologie de conception SoC basée sur leurs outils actuels.

1.3.1. Principes de réutilisation

Pour bien cibler les principes associés à la réutilisation, l'histoire de la réutilisation sera présentée. Puis, les types d'IP et de paramètres seront expliqués. Enfin, les critères d'une réutilisation efficace seront donnés.

1.3.1.1. Histoire de la réutilisation

La réutilisation a toujours existé [CCH+99], mais sa forme a changé à travers l'évolution de la technologie (Figure 1.11). Initialement, la réutilisation se faisait au niveau personnel et individuel. Chaque concepteur utilisait son expérience et ses connaissances pour faciliter la réalisation de nouveaux projets. Ensuite, le code des designs précédents était utilisé comme squelette pour les nouveaux designs. Ce type de réutilisation est très populaire chez les programmeurs. Jusqu'à ce point, la réutilisation était plus opportuniste que planifiée.

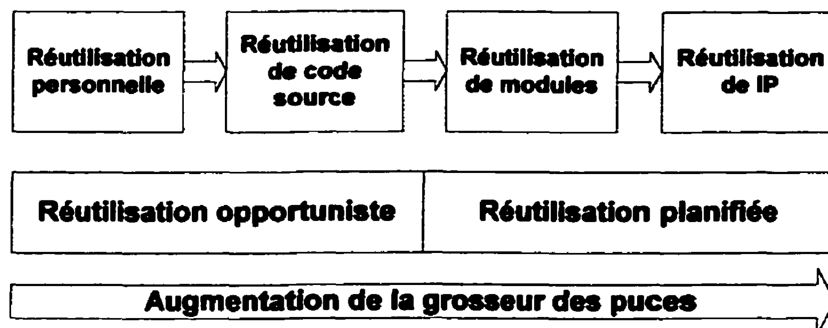


Figure 1.11. Évolution de la réutilisation

Puis, les concepteurs ont commencé à reprendre des modules complets de design et à les réutiliser dans de nouveaux designs. Les modules sont alors un peu mieux documentés dans le but de permettre à plusieurs concepteurs de les utiliser. Beaucoup d'entreprises se situent encore à ce niveau de la réutilisation.

Le pas suivant implique la "socketization" des IP amène à la réutilisation à proprement dit. Cependant, il existe plusieurs niveaux de réutilisation d'IP [Runn00]. D'abord, il est possible de transformer un module pour le rendre réutilisable sans penser au contexte futur d'utilisation. Si ces contextes sont considérés lors de la conception d'un IP (par exemple, au niveau des paramètres), alors l'efficacité est augmentée. Enfin, si les échanges d'IP se font à l'intérieur d'une entreprise, mais aussi au niveau de toute l'industrie, on parle de réutilisation à grande échelle.

1.3.1.2. Types d'IP

Il existe trois types d'IP réutilisable que l'on nomme "soft", "hard" et "firm" et chacun comporte des avantages et inconvénients [FaWa00]. Il pourrait aussi y avoir des IP logiciels, mais ils sont peu considérés dans la documentation. Le IP "hard" est un module complètement implanté physiquement. Il n'y a donc aucune flexibilité, mais toute l'information nécessaire pour une technologie cible est disponible et le travail d'intégration est facile.

Le IP "soft" est tout simplement un bloc de code RTL qui se synthétise. Son grand avantage est sa flexibilité. Cependant, puisque le module n'est pas implanté physiquement, aucune information sur les performances, le coût du silicium, la grandeur n'est disponible. De plus, le travail d'intégration à un système est lourd et la protection du IP est très difficile.

Le IP "firm" est une combinaison des précédents. C'est un IP "soft" qui a été implanté physiquement. Si l'implantation est satisfaisante, il est possible de bénéficier de tous les avantages du IP "hard". Dans le cas contraire, ce sont les désavantages du IP "soft" qui seront subis.

Les IP peuvent être utilisés à plusieurs fins [Lee00]. Dans la majorité des cas, on parle de l'utilisation d'IP à l'intérieur d'un design. Cependant, il est aussi possible d'utiliser un IP pour de la vérification, donc à l'intérieur d'un banc de test. De plus, l'évaluation d'outils, de méthodologies ou de technologie peut se faire à l'aide d'IP.

1.3.1.3. Configurations et paramètres

La flexibilité passe très souvent par la possibilité de configuration. Une configuration est la sélection d'un ensemble de paramètres pour former un tout (par exemple, un IP "soft") [VaGi00]. Un paramètre est un attribut qui peut être modifié et qui, sans changer la fonctionnalité du module, permet de trouver le meilleur compromis (pour un système donné) entre les performances, les tailles, la dissipation de puissance, etc. Il est très important que les différents paramètres soient indépendants les uns des autres pour éviter des configurations finales incohérentes.

Les paramètres peuvent être statiques, s'ils sont modifiés avant l'étape de fabrication, ou dynamique, s'il faut ajouter une structure supplémentaire pour permettre la variation du paramètre en temps réel. Les paramètres statiques peuvent impliquer des changements de grandeurs de signaux ou des changements complets de structure. Les

paramètres dynamiques peuvent être modifiés à l'allumage seulement ou n'importe quand durant le fonctionnement. Les paramètres peuvent être au niveau des circuits, de l'architecture ou de l'application.

1.3.1.4. Critères de réutilisation

Peu importe le type d'IP utilisé, certaines étapes doivent être effectuées pour préparer l'IP à la réutilisation, c'est le concept de "socketization". Cette tâche peut allonger le temps de design d'un IP de 2 à 7 fois selon le cas [Birn00]. Par exemple, un IP "hard" est plus long à préparer qu'un IP "soft". L'important est de s'assurer que le temps perdu sera récupéré lors de la réutilisation.

Pour ce faire, il faut respecter certains critères [Jani00] comme avoir des dispositifs intéressants, être compatible avec les normes de l'industrie, être indépendant de l'environnement de développement, etc. Il est essentiel d'abréger les détails inutiles à l'utilisateur et de rendre accessibles ceux qui lui seront importants. Le type de codage doit aussi respecter certaines règles [KeBr99] comme de synchroniser les entrées et sorties, éviter les circuits asynchrones, etc.

En résumé, un IP devra être documenté et avoir des paramètres bien définis, un modèle à différents niveaux d'abstraction, des scripts de simulation et de synthèse, un plan de test, un banc de test, une liste de contraintes, etc. L'idéal est de donner le plus de flexibilité (paramètres, choix de technologies, dispositifs) tout en simplifiant le plus possible le travail d'intégration. L'intégrateur système doit facilement pouvoir explorer différentes implémentations.

1.3.2. Synopsys

L'outil de synthèse logique le plus utilisé actuellement est le "DesignCompiler" de Synopsys. La méthodologie de conception SOC est basée sur le fait que si un IP peut être configuré et qu'il respecte les contraintes énoncées à la section 1.3.1, le temps d'intégration au niveau système est grandement réduit. Deux façons de générer un IP spécifique à partir d'un IP générique sont proposées [CRK00] [SCC00]. La première consiste à utiliser un fichier de paramètres, un fichier RTL utilisant ces paramètres et un script Perl pour générer un fichier RTL reflétant les besoins de l'usager. La deuxième consiste à utiliser les outils de SoC : "CoreBuilder" et "CoreConsultant".

1.3.2.1. CoreBuilder

L'outil "CoreBuilder" permet au concepteur d'IP d'être guidé à travers les étapes de "socketization" et d'entrer un IP dans une base de données qui pourra être utilisé par l'intégrateur. Le concepteur peut saisir le code VHDL synthétisable ainsi que l'intervalle des paramètres, les contraintes, le banc de test, etc. L'outil guide l'usager et s'assure de la cohérence des données entrées. Ensuite, il génère les scripts, les fiches techniques, les données utiles à la synthèse, etc. Il est donc possible de bâtir une bibliothèque d'IP bien documentés utiles aux intégrateurs.

1.3.2.2. CoreConsultant

L'outil "CoreConsultant" guide l'intégrateur dans ses choix de paramètres et d'implémentations. À partir des fichiers générés par "CoreBuilder" et des contraintes de l'intégrateur, un IP répondant spécifiquement aux besoins du système ainsi que son banc de test sont générés. Tous les scripts et données utiles à l'obtention d'une implantation physique réelle sont aussi fournis. L'outil permet donc de réduire le support et la maintenance des IP en améliorant les résultats obtenus.

1.3.2.3. Intégration du SoC

L'objectif futur est de permettre la génération d'un système complet. L'idée est de permettre à l'usager de créer des plates-formes de base (incluant aussi des paramètres et des attributs). À partir des données sur les plates-formes et sur les IP, il serait alors possible de générer un système complet spécifique à une application. L'étape suivante serait de rendre toutes ces bibliothèques accessibles sur l'Internet pour permettre la réutilisation à grande échelle.

1.3.3. Cadence

L'outil VCC ("Virtual Component Co-Design") de Cadence [CDS00a] [CDS00b] est le résultat de l'initiative Felix qui a débuté en 1997. L'outil a pour buts de fournir un support pour la spécification d'IP, d'améliorer les possibilités d'évaluation, de sélection et d'implantation des IP et de permettre la définition, la capture et l'exploration des architectures. L'outil cible autant les concepteurs d'IP que les intégrateurs de systèmes. En effet, une infrastructure de capture, d'empaquetage, de catalogage, de distribution, et de gestion d'IP a été mise sur pied afin d'accéder éventuellement une banque de données par le biais d'Internet.

1.3.3.1. Capture du système et exploration des architectures

Les étapes de conception de l'outil VCC (Virtual Component Co-Design) sont données à la Figure 1.12. La spécification du système peut être hétérogène et la capture des informations du système (architecture, fonctionnalité, types de données et IP) peut se faire à l'aide d'une interface graphique. Pour créer des fonctions de contrôle ou des convertisseurs de protocole, le concepteur peut utiliser le générateur de machines à états.

Une fois la capture terminée, l'outil permet de lier les fonctions à l'architecture. Un environnement d'analyse permet d'évaluer les performances du système à l'aide de simulations. Il est possible d'associer des contraintes de performance aux fonctions et

de vérifier si elles sont respectées. Ainsi, l'intégrateur peut explorer différentes architectures. Un estimateur logiciel permet d'accélérer le temps de simulation. De plus, la simulation d'un design à plusieurs niveaux d'abstraction est possible.

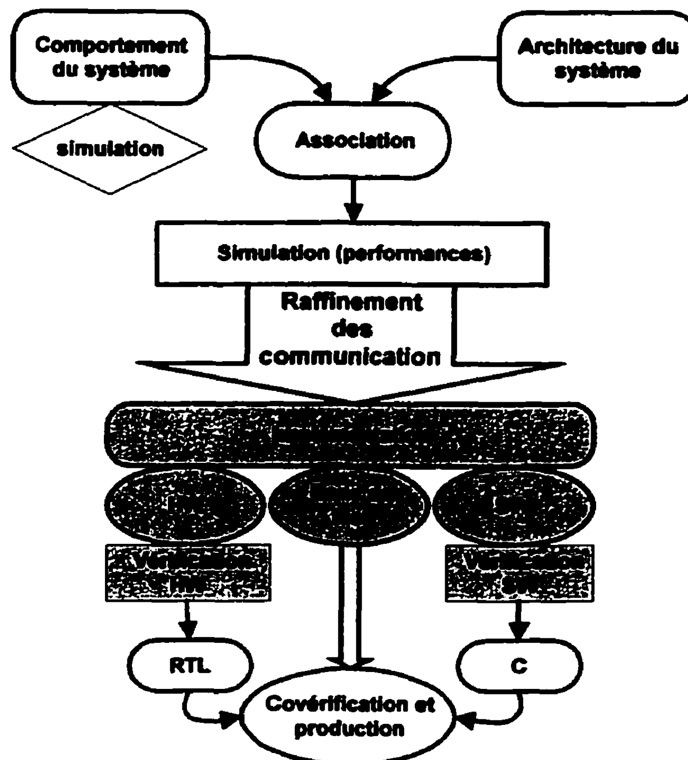


Figure 1.12. Étapes de conception de l'outil VCC de Cadence

1.3.3.2. Synthèse des communications et génération du système final

L'étape suivante est le raffinement et la synthèse des communications. Toutes les données nécessaires à ces étapes sont dans des bibliothèques. Des modèles de base sont disponibles mais l'usager peut aussi en ajouter. Ensuite, à partir des paramètres fournis par l'usager, le raffinement et la synthèse sont effectués. Plus la flexibilité est recherchée, moins le processus est automatisé. Finalement, l'outil génère une description de la partie matérielle du système au niveau structurel prêt à passer aux étapes de synthèse et simulation. En plus, le code logiciel de l'application (pilotes,

système d'exploitation temps réel, ordonnateurs) et les données essentielles au test et à la cosimulation ont aussi générés.

1.3.4. Coware

Initialement, le champ d'intérêt de Coware était la conception conjointe logicielle/matérielle en général [BML+97] [RVBM96] [MBL+96] [LiVe94] [LiVe96] (annexe F). Maintenant, l'intérêt est tourné vers la conception logicielle/matérielle pour SoC et la réutilisation d'IP. La version actuelle de Coware s'appelle N2C ("Napkin-to-Chip") [Romp00] [Cowa00]. L'outil permet au concepteur de capturer une spécification en C/C++ (avec le langage SystemC), d'évaluer les effets de l'architecture sur les performances de l'implémentation finale, de faire de la conception logicielle/matérielle et de réutiliser les IP.

1.3.4.1. Description homogène

L'outil Coware est le plus automatisé des trois présentés dans cette section. Il ressemble aux méthodologies de la section 1.1, puisqu'il permet le raffinement de système à partir d'une description de haut niveau. Un seul langage est utilisé pour décrire le logiciel et le matériel (SystemC) et deux niveaux d'abstraction sont disponibles. Le code des modules matériels peut être automatiquement transformé en une description synthétisable. Cependant, tous les IP sont décrits en C/C++ pour augmenter la flexibilité et permettre des changements de partition. Un prototype virtuel permet de faire de la cosimulation très tôt dans le processus de design.

1.3.4.2. Synthèse des communications

Un aspect important de l'outil est le synthétiseur d'interfaces. La description fonctionnelle est indépendante des communications et celles-ci sont implantées à l'aide d'une enveloppe. Comme on l'explique à la section 1.2.2.3, c'est à partir de la description des communications avec le "Virtual Bus" (le protocole de communication d'IP) que les connexions réelles sont générées. Il est possible

d'utiliser des IP de source externe, mais une enveloppe doit être conçue pour les faire communiquer avec le protocole approprié. L'objectif principal de l'outil de synthèse est de permettre un assemblage facile d'IP et la réutilisation de ceux-ci. La synthèse des communications multiprocesseurs est très efficace.

1.3.4.3. Étapes de design

En résumé, les étapes de design sont les suivantes (Figure 1.13). D'abord le système est décrit au niveau fonctionnel en SystemC. À ce stade, de nouveaux IP peuvent être crées. En parallèle, il est possible de créer des plates-formes et de faire des choix d'implémentation. De plus, le choix des IP se fait en même temps que le choix de l'architecture. Très tôt dans les étapes de design, des tests de performance sur les choix d'implémentation peuvent être effectués. Ensuite, le synthétiseur d'interfaces permet de lier la description fonctionnelle à la description de l'architecture. Enfin, le système complet est généré : modules matériels, contenu de la mémoire (programme et données), décodage d'adresses, routine d'interruptions, pilotes logiciels, etc.

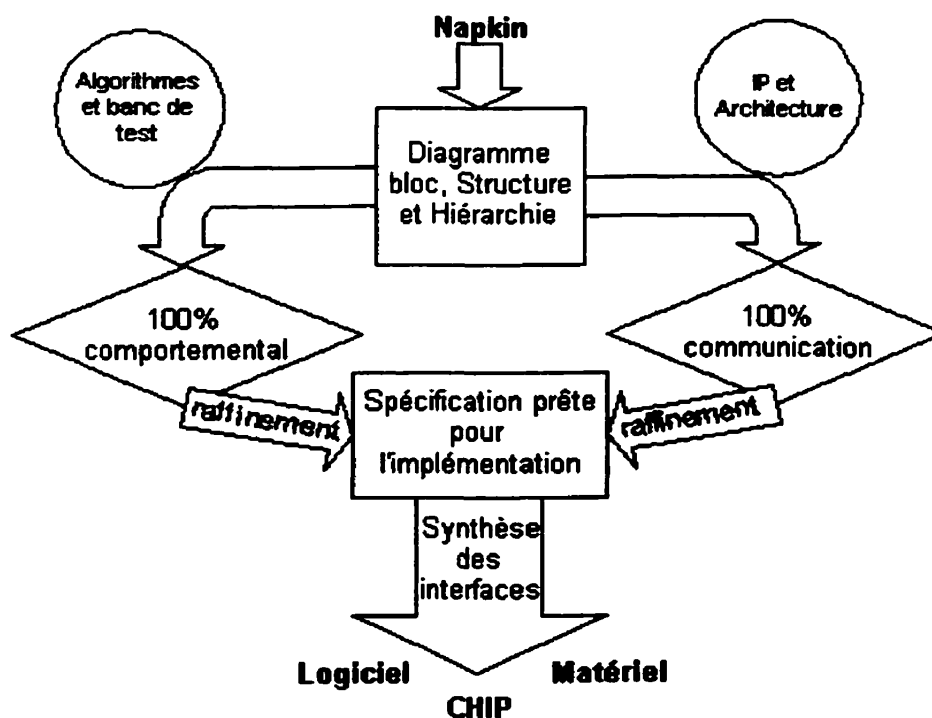


Figure 1.13. Étapes de design de Coware

Chapitre 2 : MÉTHODOLOGIE PROPOSÉE ET DÉFINITION DU PROJET

L'avancement de la technologie de fabrication et le changement de la technologie de conception amènent les chercheurs à définir une nouvelle méthodologie. Définir précisément une méthodologie complète de conception de SoC dépasse le cadre et le délai d'un projet de maîtrise. Ainsi, ce chapitre présente les restrictions et les objectifs fixés pour réaliser un projet pertinent au développement d'une méthodologie.

D'abord, la première section décrit la méthodologie générale dans laquelle le projet pourrait s'inscrire ainsi que les aspects traités. L'objectif général du projet consiste à concevoir une interface de communication pour un IP. La deuxième section précise les caractéristiques de cette interface et la dernière section présente ses possibilités de configuration et ses particularités d'utilisation.

2.1. Méthodologie de conception de SoC

Avant de décrire la problématique précise du projet, il convient de présenter la méthodologie générale proposée ainsi que les représentations du système et du canal de communication.

2.1.1. Méthodologie globale

La méthodologie proposée considère les méthodologies décrites au chapitre précédent ainsi qu'une combinaison des approches de plate-forme et d'assemblage IP (Figure 2.1). En effet, l'outil génère un système à partir d'une bibliothèque d'IP et de plates-formes configurables. Les composants de la bibliothèque peuvent être prédéfinis ou décrits par le concepteur. De cette façon, il est possible d'obtenir une grande flexibilité sans pour autant allonger le temps de conception. La méthodologie utilise donc les principes d'allocation et de configuration.

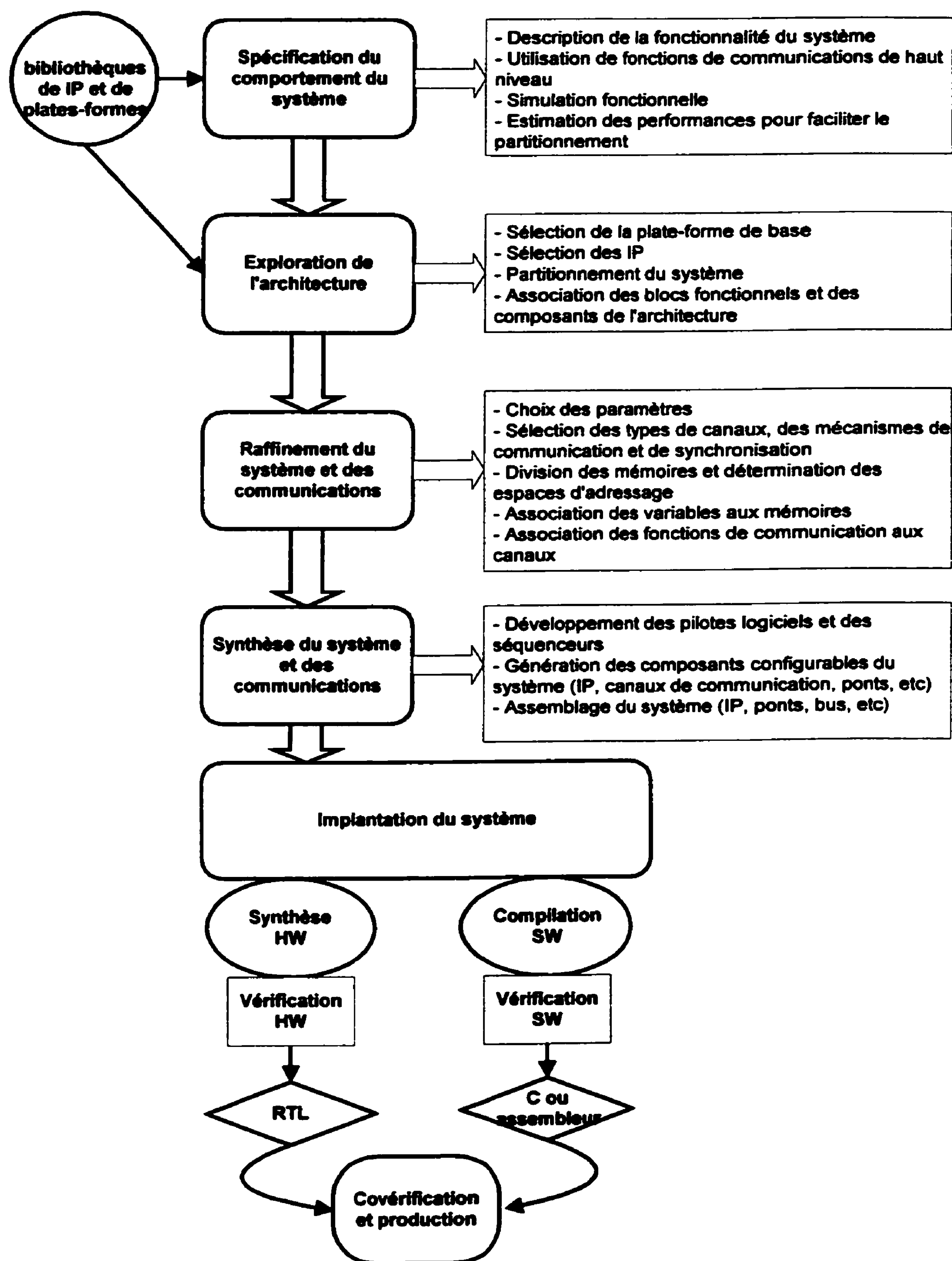


Figure 2.1. Étapes de conception proposées dans le présent projet

2.1.1.1. Extension des outils de Mentor Graphics

Comme expliqué dans le chapitre précédent, la compagnie Mentor Graphics n'a pas encore d'outil spécifique de conception SoC. La méthodologie proposée permet d'utiliser les outils disponibles de Mentor Graphics pour faire de la conception SoC. D'abord, une extension de l'outil Renoir permet de représenter et décrire le système à différents niveaux d'abstraction. De plus, il est possible d'effectuer des raffinements hiérarchiques, comme proposé par VSIA (quatre premières étapes de la méthodologie). La simulation du système logiciel/matériel s'effectue avec Seamless.

L'extension de Renoir doit aussi permettre la création et l'utilisation de bibliothèques d'IP et de plates-formes. Un outil semblable à celui proposé par Synopsys [CRK00] peut faciliter la gestion des bibliothèques. Les IP doivent être représentés à différents niveaux d'abstraction et pouvoir être simulés et testés à travers les étapes de raffinement. Des scripts de configuration permettent la génération d'IP spécifiques aux besoins d'une application. Les critères de réutilisation expliqués à la section 1.3.1.4 s'appliquent.

2.1.1.2. Description des étapes de conception

La première étape est la spécification fonctionnelle du système. Les modules peuvent être décrits en HDL ou en C (ou les deux). Des blocs prédéfinis (IP) peuvent aussi être sélectionnés à cette étape pour éviter de concevoir deux fois un même bloc. Les communications sont représentées à un niveau d'abstraction élevé en utilisant les fonctions SLIF de VSIA. La simulation du système à cette étape permet de vérifier sa fonctionnalité et de développer des estimateurs pour déterminer ses partitions.

Une fois la fonctionnalité du système déterminée, il est possible d'explorer les différentes architectures. À partir des bibliothèques de plates-formes et d'IP, l'architecture du système (structure, composants) est déterminée. Ensuite, il faut diviser le système et associer les blocs fonctionnels aux composants de l'architecture.

Différentes configurations peuvent être implantées jusqu'à l'obtention des performances voulues. À cette étape, les communications sont décrites à un niveau d'abstraction élevé et les paramètres du système ne sont pas fixés.

Le raffinement du système consiste à fixer les paramètres de la plate-forme et des IP en fonction des besoins de l'application. Ensuite, la sélection des types de canaux de communication et des mécanismes de communication et synchronisation constitue la phase de raffinement des communications. De plus, les mémoires sont divisées, les espaces d'adressage sont fixés et les variables sont associées aux mémoires. De cette façon, les fonctions de communication peuvent être affectées aux canaux. Le niveau d'abstraction est donc abaissé et le système est prêt à passer à l'étape de synthèse.

L'étape suivante est celle de la synthèse du système où les choix effectués à l'étape précédente sont implantés de façon détaillée (bas niveau d'abstraction). Pour gérer l'ordre des accès aux ressources par le logiciel, des pilotes et des séquenceurs sont ajoutés au code. De plus, des scripts sont utilisés pour générer la bonne configuration de la plate-forme et des IP. Des composants de communication de bas niveau (ponts) sont ajoutés pour permettre l'assemblage du système. Ainsi, le système est prêt pour la phase finale d'implémentation.

Lors de la dernière étape de l'implémentation du système, le code matériel est synthétisé et vérifié à l'aide d'outils commerciaux (par exemple Design Compiler de Synopsys). D'un autre côté, le code logiciel est compilé et vérifié. Ensuite, à partir du code généré, il est possible de passer à la phase de covérification finale (avec l'outil Seamless) puis aux tâches de fond.

2.1.2. Représentation du canal de communication

La partie la plus importante et la plus complexe de la conception de SoC est l'implantation des communications entre les modules. Pour mieux comprendre les étapes de raffinement et de synthèse des communications, il est essentiel d'expliquer les composants de la communication et de les situer dans un système global.

2.1.2.1. Description des étapes de communication

Les étapes de la communication, données à la Figure 2.2, viennent de la représentation des communications entre des modules logiciels et matériels présentée à la Figure 1.2, de la représentation des canaux virtuels donnée à la Figure 1.1 et du principe d'enveloppe proposé dans [BML+97]. Le canal à proprement dit correspond seulement à la partie centrale (grise sur la Figure 2.2). Cependant, pour que les modules logiciels et matériels puissent transmettre des messages sur ce canal, ils doivent utiliser un protocole commun (par exemple VCI de VSIA). Ainsi, pour raffiner les communications entre deux blocs, plusieurs étapes doivent être exécutées.

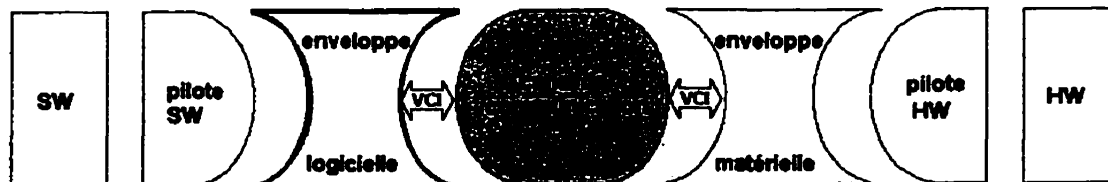


Figure 2.2. Étapes de l'établissement des communications entre deux blocs

Du côté logiciel, le bloc *SW* (logiciel) correspond au programme ainsi qu'au processeur sur lequel il est exécuté. Les pilotes représentent les routines logicielles qui permettront au programme d'accéder les ports du processeur et de communiquer avec l'extérieur. L'enveloppe logicielle est constituée de tous les périphériques permettant au processeur de fonctionner et de communiquer. De plus, la génération des signaux (signaux VCI) nécessaires aux communications externes est faite par l'enveloppe.

Du côté matériel, la partie *HW* (matérielle) correspond à la description fonctionnelle d'un bloc. En effet, la partie fonctionnelle et les communications sont bien séparées pour faciliter l'intégration. Le pilote matériel correspond aux fonctions de communication implantées dans le bloc. Si ces fonctions n'utilisent pas le bon protocole de communication, une enveloppe est nécessaire pour faire la conversion de protocole.

2.1.2.2. Représentation d'un système communicant

Maintenant que le chemin emprunté par les messages transmis entre deux blocs a été décrit, il est intéressant d'intégrer les éléments de la communication dans un système réel. La Figure 2.3 représente un système quelconque constitué d'IP et d'un OCB. Comme mentionné à la section 1.2.2, lorsque le protocole de communication des IP choisis n'est pas le même que le protocole du bus, un pont est nécessaire. Le canal de communication entre les deux IP peut alors être vu comme étant le bus et les deux ponts. On voit donc qu'une communication point-à-point peut facilement être transformée en une communication à travers un bus (Figure 2.2 et Figure 2.3).

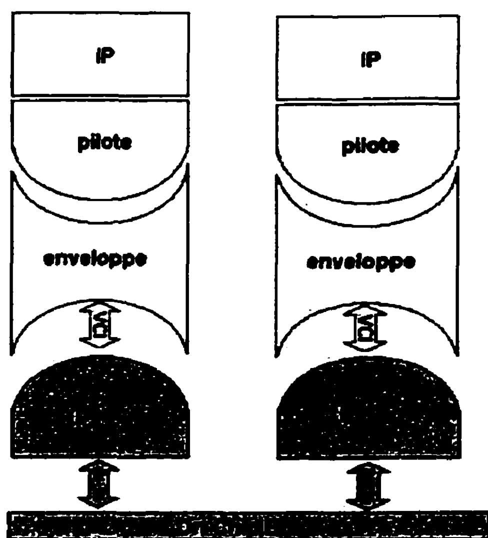


Figure 2.3. Représentation du système

Dans la méthodologie proposée, deux bibliothèques de composants configurables sont utilisées. La bibliothèque d'IP comprend les pilotes et les enveloppes associés à chacun des IP et la bibliothèque de plate-forme comprend les ponts associés au OCB utilisé. Ainsi, tous les composants nécessaires à l'établissement de communication entre les blocs sont compris dans les bibliothèques. En choisissant les paramètres adéquats, il est possible de configurer chacun des éléments du système et ainsi être prêt pour la phase finale d'implémentation.

2.1.3. Aspects de la méthodologie traités

Dans le type de méthodologie proposée, le degré d'automatisation est sacrifié au profit de la flexibilité. Néanmoins, la méthodologie est d'autant plus automatisée que les bibliothèques sont élaborées. Une fois les bibliothèques construites, l'extension de Renoir, permettant l'utilisation de composants configurables et le raffinement du système, peut être développée. Cependant, l'objectif du présent projet est d'abord de vérifier la validité de la méthodologie et ces principes de base en effectuant les différentes étapes manuellement. Pour ce faire, les outils actuels de Mentor Graphics (Renoir et Seamless) seront utilisés, car ils constituent la base de la méthodologie.

Dans le cadre du projet, une interface de communication configurable pour un IP est développée et cet IP est implanté dans différents systèmes. De cette façon, sans développer une bibliothèque complète, il est possible d'évaluer l'utilisation des principes de configuration et d'allocation pour la construction de systèmes. Ainsi, les étapes de la méthodologie qui méritent d'être automatisées et les éléments des bibliothèques qui devraient être développés peuvent être identifiés. En outre, les applications sont implantées avec et sans le protocole VCI. Par conséquent, l'analyse des performances des différentes implantations permet d'évaluer la pertinence de l'utilisation du protocole VCI.

En résumé, il faut concevoir, avec Renoir, une enveloppe pour un processeur ARM7DTMI et la simuler avec Seamless. L'enveloppe permet d'implanter différents moyens de communication et de synchronisation et d'établir les communications avec le protocole VCI. La sélection des paramètres permet de générer une enveloppe au processeur en fonction des besoins de l'application.

Les pilotes logiciels sont construits manuellement. De plus, l'association des variables aux composants mémoires et les problèmes de consistance mémoire ne sont pas traités. La Figure 2.2 résume bien les limites du projet en identifiant la partie du canal traitée : l'enveloppe de communication du processeur.

2.2. Description des caractéristiques de l'interface

L'interface de communication développée est une enveloppe de communication pour le processeur ARM7TDMI basée sur le protocole VCI et intégrant plusieurs moyens de communication et de synchronisation. Avant de décrire l'interface en détail, il convient de justifier les choix effectués en précisant les caractéristiques du processeur, les moyens de communication et de synchronisation et les particularités du protocole.

2.2.1. Type d'IP : ARM7TDMI

Dans le cadre du projet, un seul IP est étudié. Cet IP est le processeur ARM7TDMI. Les caractéristiques générales sont d'abord présentées. Puis, le choix de cet IP est justifié. Ensuite, un bref aperçu du modèle de programmation ainsi que de l'interface mémoire est donné. Pour plus d'information, voir l'annexe G et les documents [ARM00b] [ARM00c] [ARM00d] et [ARM00e].

2.2.1.1. Description générale du processeur

L'ARM7TDMI est membre de la famille de microprocesseurs à usage général de 32 bits d'ARM. L'architecture ARM est basée sur les principes RISC (Reduced Instruction Set Computers). Elle possède un débit d'instructions très élevé et un excellent temps de réponse pour les interruptions en temps réel. Le processeur peut exécuter 54 MIPS (Million d'Instructions Per Second), 690 MIPS par watt et sa fréquence maximale est de 60MHz avec une technologie de CMOS 0,35 microns.

Le processeur ARM7TDMI utilise une stratégie architecturale particulière. Essentiellement, le processeur ARM7TDMI a deux ensembles d'instructions : l'ensemble standard ARM de 32 bits et l'ensemble THUMB de 16 bits. L'ensemble THUMB de 16 bits permet d'obtenir une densité de code deux fois plus performante en conservant presque tous les avantages de performance d'un processeur de 32 bits.

Le processeur ARM7TDMI possède des modules utiles pour le test et la vérification : "Debug interface module" et "ICEBreaker module". Il possède aussi une interface pour un coprocesseur. Cependant, le modèle de cette interface n'est pas disponible dans Seamless. De plus, l'architecture AMBA est spécialement conçue pour des systèmes contenant un processeur ARM (voir la section 1.2.1.3).

2.2.1.2. Justification du choix du IP

Le but est d'utiliser un IP pour implanter plusieurs systèmes. Étant donné que la plupart des SoC contiennent au moins un processeur et que ces composants sont programmables, il convient de choisir un processeur comme type d'IP. De cette façon, plusieurs applications peuvent être implantées facilement sur différentes plates-formes (multiprocesseur, processeur/coprocesseur).

L'ARM est un des processeurs embarqués les plus utilisés de nos jours. En effet, l'architecture ARM a atteint le troisième rang mondial au niveau des ventes en 1997

[Moli00]. Le taux de croissance a été de 133 % les années précédentes. Plus de 30 compagnies avaient des licences d'ARM en décembre 1999. Ce processeur est utilisé dans des systèmes embarqués comme les modems, les disques durs, la technologie sans fil, etc. Cette architecture permet d'avoir la meilleure combinaison entre les performances, la dissipation de puissance, le coût et la facilité d'utilisation.

Étant donné son utilisation répandue, l'architecture AMBA (et l'ARM) font souvent partie des bancs de test dans la littérature [CCS99] [Smit00] [LKSN99] [Lee00] [Sett00]. En plus, le modèle du processeur est disponible sur Seamless et ceci est essentiel si les outils de Mentor Graphics doivent être utilisés. Pour toutes ces raisons, le processeur ARM7DTMI sera l'IP utilisé dans le cadre du présent projet.

Il faut cependant noter qu'il est totalement inutile d'utiliser un protocole d'IP pour lier un processeur et un bus lorsque, à la base, les deux utilisent le même protocole [LVG00]. Dans le cas d'un système utilisant l'architecture AMBA et le processeur ARM, l'interface proposée peut être considérée inutile. Néanmoins, pour des systèmes multiprocesseurs (PowerPC ou DSP), une autre architecture (CoreConnect) pourrait s'avérer être plus appropriée. De plus, la fréquence d'utilisation de l'ARM dans la littérature augmente l'intérêt des résultats obtenus.

2.2.1.3. Modèle de programmation

D'abord le processeur possède deux modes de fonctionnement, le mode normal et le mode THUMB. Dans le cadre du projet, seul le mode normal sera utilisé. De plus, il possède plusieurs modes d'opération. Entre autres, il possède deux modes d'interruption, le mode général (IRQ) et le mode de transfert de données (FIQ). Dans ce projet, seul le mode général (IRQ) sera utilisé.

Les instructions (section G.3) permettent d'effectuer des opérations sur les registres (section G.2). Les instructions peuvent contenir jusqu'à trois opérandes de 32 bits

chacune. Le processeur contient 37 registres, dont 31 registres généraux et six registres d'états. Seulement 16 registres généraux et deux registres d'états peuvent être visibles à la fois.

Les pipelines sont employés de façon à ce que chaque partie du système de traitement et de mémoire puisse fonctionner en même temps. En général, pendant qu'une instruction est traitée, l'opération subséquente est décodée et une troisième instruction est récupérée de la mémoire. Les instructions d'accès mémoire d'emmagasinage et de récupération prennent respectivement 3 et 2 cycles.

2.2.1.4. Interface mémoire

Le processeur ARM ne possède pas d'entrées/sorties spécifiques et le modèle de l'interface du coprocesseur n'est pas disponible sur Seamless. En conséquence, les communications externes doivent passer par l'interface mémoire du processeur. Elle est décrite brièvement ici. Cependant, la liste des signaux est disponible à la section G.1. De plus, certains diagrammes temporels sont disponibles à la section G.4. À partir de la documentation sur la conception d'interface pour l'ARM, certains éléments de conception utiles sont donnés à la section G.5.

L'ARM offre deux façons de générer les adresses (avec ou sans pipeline). Les deux possibilités d'adressage de la mémoire (gros-boutiste et petit-boutiste) sont disponibles. Il est possible d'insérer des cycles d'attente lorsque le périphérique est lent. Les données peuvent être multiplexées (un seul bus pour la lecture et l'écriture) ou non (un bus pour l'écriture et un autre pour la lecture). La grandeur des données est flexible. Les transferts par paquets sont disponibles par l'intermédiaire de certaines instructions. Il existe aussi des opérations mémoire bloquantes.

2.2.2. Moyens de communication et de synchronisation

Un schéma de communication est toujours défini par des mécanismes de communication et de synchronisation. Après la description de ces mécanismes, ceux qui sont spécifiquement utilisés dans ce projet sont donnés.

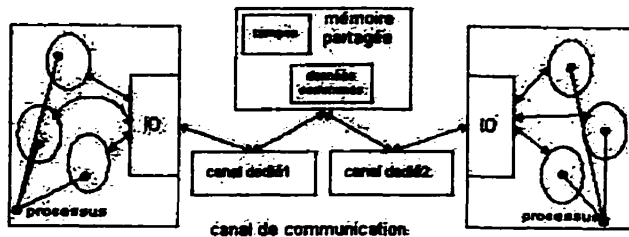
2.2.2.1. Types de communication

On définit deux types de communication : l'un par mémoire partagée et l'autre par passage de messages. Dans le cas de mémoire partagée, le processus émettant les données doit écrire dans un espace qui est également accessible au processus recevant le message (Figure 2.4). Dans ce cas, la synchronisation n'est pas implicite. Ce type de communication permet l'utilisation du mécanisme de diffusion (l'envoi à plusieurs processus en même temps). Un médium est persistant si la valeur inscrite est retenue jusqu'à la réécriture de cet espace, ceci même si l'alimentation est interrompue.

Le deuxième type de communication est le passage de messages. Le canal choisi peut être une ligne réservée pour une connexion point-à-point ou un bus pour permettre des messages diffusés (Figure 2.4). Ce canal peut comprendre une ou plusieurs FIFO pour permettre une communication non bloquante éliminant les pertes de cycles pour la synchronisation. La communication peut donc être bidirectionnelle ou unidirectionnelle, point-à-point ou à directions multiples, bloquante (ou synchrone) ou non bloquante (asynchrone) selon l'implémentation choisie.

Lorsque la communication bloquante est utilisée, la synchronisation est implicite et aucun composant de mémorisation n'est nécessaire. Cependant, la performance des processus est diminuée. Pour une communication non-bloquante, les processus n'ont pas à se synchroniser et la performance n'est pas réduite. Cependant, des composants de mémorisation (registre ou FIFO) sont ajoutés pour éviter les pertes de données.

A) Mémoire partagée



B) Passage de messages

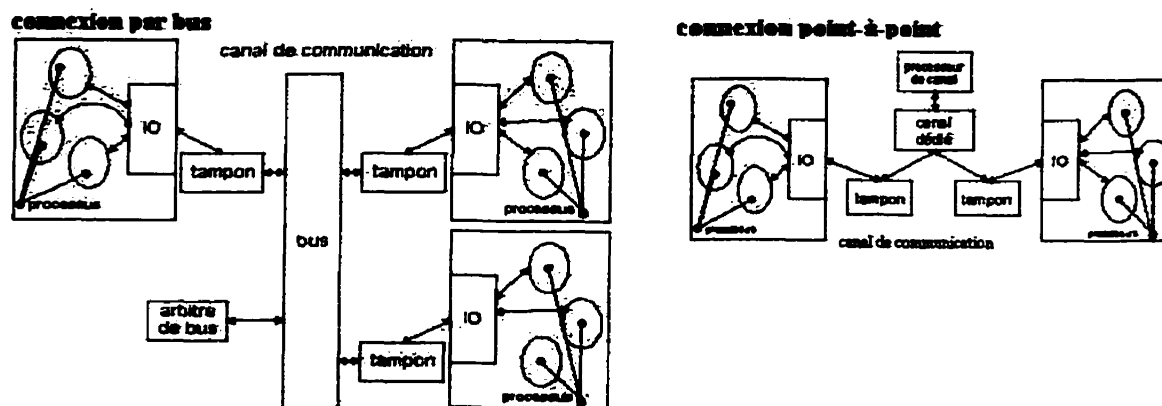


Figure 2.4. Modèles de communication

2.2.2.2. Types de synchronisation

La synchronisation des communications peut se faire de trois façons : par mémoire partagée, par passage de message ou par interruptions logicielles. Lorsque la synchronisation est directement liée au mécanisme de communication, elle est dite implicite. Un bon exemple est une communication bloquante par passage de messages où un protocole de poignée de main est utilisé pour gérer les transferts.

La mémoire partagée est un moyen d'inscrire des données pour indiquer l'état de la communication. Les sémaphores, les boîtes aux lettres et les espaces mémoire réservés sont trois exemples de synchronisation par mémoire partagée. La synchronisation par passage de messages est plus directe. Dans ce cas, des signaux réservés à la communication sont utilisés ou des données sont écrites sur un canal de

communication précis. L'utilisation d'interruptions logicielles est très efficace, mais est seulement applicable aux communications avec un processeur.

Dans [NiMa98], des schémas de synchronisation utilisant simultanément plusieurs moyens sont proposés. Le premier schéma utilise les interruptions pour indiquer un changement d'états au processeur et la mémoire partagée pour noter l'état (Figure 2.5). Le deuxième schéma utilise, pour indiquer l'état du processeur, des lignes réservées au lieu de la mémoire partagée. Lorsque le système ne contient pas de processeur, les interruptions sont transformées en signaux réservés (Figure 2.6).

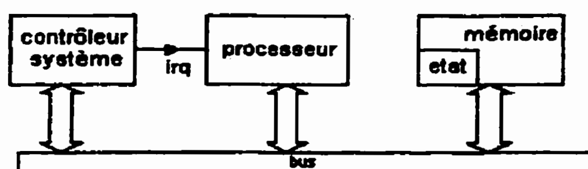


Figure 2.5. Synchronisation par mémoire partagée et interruptions

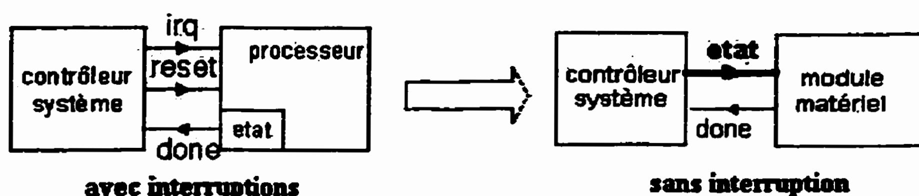


Figure 2.6. Synchronisation par passage de messages

2.2.2.3. Mécanismes disponibles dans l'interface

Le Tableau 2.1 donne un résumé des mécanismes de communication qui pourrait être disponibles dans l'interface. Dans chacun des cas, le type et le médium de communication ainsi que les mécanismes de synchronisation associés sont précisés. En plus, les communications bloquantes et non bloquantes sont identifiées. Notre travail s'est limité à des communications non bloquantes. Néanmoins, ce choix permet d'utiliser la plupart des mécanismes de synchronisation qui ne sont pas

implicites. De plus, les applications développées n'utiliseront que les communications point-à-point (section 2.3.2).

Tableau 2.1. Mécanismes de communication de l'interface

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Mémoire partagée	Mémoire à simple port	Bloquante	"Memory Handshake"	Tous les mécanismes sauf les sémaphores et boîtes aux lettres
	Mémoire à double ports	Non bloquante	"Enable Handshake"	Tous les mécanismes
Passage de messages	Connexion directe	Bloquante	"Full Handshake"	Interruption logicielle ou signaux réservés
	FIFO	Non bloquante	"Memory without handshake"	Interruption logicielle, signaux et données réservés

2.2.3. Protocole de communication

VCI de VSIA a été choisi comme protocole de communication pour IP à cause de sa simplicité, sa flexibilité et l'intérêt que l'industrie lui porte. Il convient de le décrire de façon un peu plus détaillée pour faciliter la compréhension de l'interface de communication ainsi que les résultats obtenus. D'abord, les trois normes seront présentées, puis le protocole BVCI sera détaillé.

2.2.3.1. Trois normes différentes

Tout comme dans l'architecture AMBA (section 1.2.1.3), VSIA propose trois normes différentes mais compatibles les unes avec les autres : le VCI périphérique, de base et avancé.

Le PVCI ("peripheral") est un sous-ensemble du protocole de base et pourrait être associé au APB de AMBA. Il est destiné aux IP peu complexes qui n'ont pas besoin de tous les dispositifs du protocole de base. Son interface étant simple, elle est très facile à concevoir. Ce protocole est particulièrement bien adapté aux modules esclaves, mais certains signaux facultatifs permettent à des maîtres de l'utiliser. La

requête et la réponse sont associées, de sorte que la réponse doit être reçue avant de considérer la fin d'une transaction. De plus, seuls les transferts par paquets simples sont permis et aucune génération d'erreur n'est possible.

Le protocole de base ("basic" ou BVCI) est équivalent au ASB de AMBA. Il définit une interface qui est destinée à la majorité des applications. C'est d'ailleurs pourquoi il a été sélectionné pour la réalisation de l'interface. Les requêtes et les réponses sont indépendantes. Ainsi, contrairement au PVCI qui n'a qu'une poignée de main à deux signaux, deux mécanismes de poignée de main sont nécessaires dans le cas du BVCI (requête et réponse). Cependant, l'ordre des requêtes et des réponses doit être respecté. Les transferts par paquets complexes sont permis (chaînage, opération de blocage) et la génération d'erreur est possible. Plusieurs signaux sont facultatifs, ce qui permet d'adapter la complexité de l'interface en fonction des besoins de la communication.

Le dernier protocole ("advanced" ou AVCI) est le plus complexe et est équivalent au AHB de AMBA. Il est utile dans des systèmes à plusieurs processeurs (ou maîtres) nécessitant une grande bande passante. Des dispositifs (signaux) sont ajoutés au protocole de base pour éviter une diminution des performances des IP complexes. D'abord, il est possible d'utiliser un mode d'adressage où un paquet peut contenir une seule requête et plusieurs réponses. En plus, l'ordre des requêtes et des réponses dans une chaîne peut différer, car les maîtres et "thread" peuvent être identifiés individuellement. La génération d'erreur complexe ainsi que de l'arbitrage en arrière-plan est possible.

2.2.3.2. Contrôle des accès par poignée de main

Puisque le protocole BVCI est celui utilisé dans le projet, il mérite un peu plus d'explications. Il faut noter que ce protocole est limité aux transferts simples et unidirectionnels et aux connexions point-à-point. Le transfert de données selon le

protocole BVCI se fait en deux parties. En premier lieu, l'initiateur envoie une requête à la cible. Pour contrôler le transfert de données, un mécanisme de poignée de main est utilisé. Dans un second temps, la cible envoie une réponse à l'initiateur et le transfert est contrôlé par un autre mécanisme de poignée de main. La requête et la réponse sont donc complètement indépendantes (Figure 2.7 et Figure 2.8).

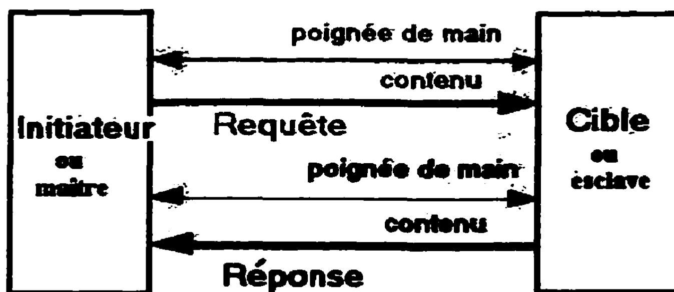


Figure 2.7. Transfert de données entre l'initiateur et la cible (BVCI)

Le mécanisme d'échange est composé de trois éléments : un signal VAL (contenu valide), un signal ACK (accusé de réception) et le contenu de la requête ou de la réponse (Figure 2.8). Pour considérer la fin d'une transaction, le signal VAL et le signal ACK doivent être hauts sur un front montant de l'horloge. Puisque chaque transfert doit se faire sur le front montant de l'horloge, les transferts d'une requête et d'une réponse ne peuvent se produire dans un même cycle. Aucun délai n'est imposé, mais il est recommandé que les sorties soient stables 20% après la montée de l'horloge et que les entrées soit stables 20% avant la montée de l'horloge.

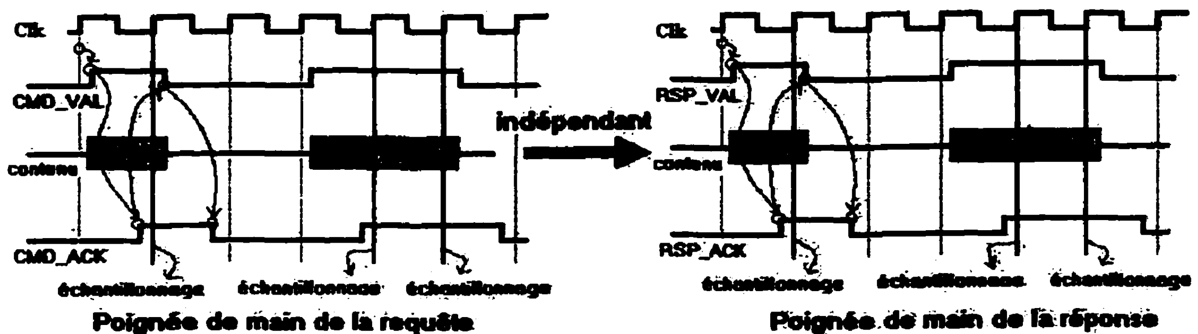


Figure 2.8. Mécanisme de poignée de main (BVCI)

En résumé, en même temps que l'initiateur envoie le contenu d'une requête, il doit mettre le signal CMD_VAL à 1 et attendre la montée de l'horloge où la cible aura mis le signal CMD_ACK à 1. De même, lorsque la cible envoie une réponse, elle doit mettre le signal RSP_VAL à 1 et attendre la montée de l'horloge où l'initiateur aura mis le signal RSP_ACK à 1. Il faut noter qu'il est possible de fixer le signal ACK à 1 par défaut ("default_ack").

2.2.3.3. Contenu des requêtes et des réponses du BVCI

Les différents signaux ainsi que leur description sont donnés à l'annexe H. Il y a cinq catégories de signaux : les signaux du système (horloge et restauration), les signaux de poignée de main (requêtes et réponse) et les signaux de contenu des requêtes et des réponses. Les transferts sont équivalents à des opérations de lecture et d'écriture dans une mémoire. Les transferts par paquets sont possibles. La grandeur des différents signaux est variable en fonction du système et de l'IP et les paramètres doivent être bien documentés. La section H.3 donne une description des paramètres de BVCI ainsi que leur portée.

La requête doit absolument contenir une adresse, un signal d'activation d'octet, une commande, un signal indiquant la fin d'un paquet et une donnée (pour une écriture). La commande peut indiquer une lecture, une lecture en blocage, une écriture ou une opération vide. En plus, des signaux utiles à la gestion des paquets sont facultatifs. Ils permettent entre autres de connaître la séquence des adresses. Ces signaux ne seront pas utilisés dans l'interface développée. La réponse contient une donnée (pour une lecture), un signal d'erreur et un signal indiquant la fin d'un paquet.

2.3. Utilisation de l'interface

Il convient maintenant de décrire comment utiliser l'interface dans un système. Pour répondre à cette question, il faut d'abord expliquer comment générer l'interface convenant à nos besoins. Ensuite, il faut donner le contexte d'utilisation de celle-ci.

2.3.1. Configuration de l'interface

La configuration correspond à la sélection de paramètres pour former un tout (voir section 1.3.1.3). Comment le principe de configuration a-t-il été exploité ? Quels sont les paramètres et les composants disponibles ? Quelles sont les limites de la génération automatique ? La réponse à ces questions permettra de mieux comprendre comment utiliser l'interface proposée.

2.3.1.1. Exploitation du principe de configuration

La génération automatique d'une configuration (code VHDL désiré) est réalisée à l'aide d'un script Perl [ScCh97], comme proposé dans [CRK00]. Pour ce faire, un fichier de configuration doit être complété par l'intégrateur en fonction de l'application à implanter. À partir de ce fichier, du script Perl et d'une interface générique, l'interface spécifique peut être générée. Le fichier générique doit être codé en respectant les critères de réutilisation décrit à la section 1.3.1.4.

En poussant plus loin, il est possible de faire comme Synopsys et remplacer le fichier de configuration par une interface usager pour faciliter l'utilisation. En outre, il est possible de guider l'usager en indiquant ses choix incohérents. Dans le présent projet, le script est très peu poussé, le fichier de configuration très détaillé et le concepteur doit faire des choix cohérents.

Pour simplifier le script de génération, il convient d'utiliser une syntaxe VHDL particulière [Ashe96]. Deux éléments du VHDL sont essentiels : les "package" et les instructions "if generate". L'idéal est de regrouper les définitions de types, de constantes, de fonctions dans un même fichier ("package"). Ainsi, en modifiant ce seul fichier, tout le code peut être modifié facilement. Par exemple, la taille des bus peut être adaptée facilement. Cependant, lorsque des modules doivent être supprimés dans certaines configurations, il faut utiliser l'instruction "if generate" pour indiquer dans quel cas garder le module et dans quel cas le rejeter.

2.3.1.2. Définitions des composants et paramètres

Il convient maintenant de préciser les paramètres et composants permettant d'avoir une interface flexible. Il faut noter que, dans le cadre de ce projet, seulement certaines possibilités ont été explorées, étant donné la limite de temps et de ressources. Pour obtenir une plus grande flexibilité, d'autres possibilités devraient être explorées.

La documentation d'ARM propose (Figure G.10) d'utiliser seulement des mémoires SRAM (Static Random Access Memory) sur la puce et de mettre la mémoire ROM (Read Only Memory) hors puce. À l'allumage, le contenu de la ROM devrait être transféré dans les SRAM pour réduire au minimum les accès mémoire hors puce. Le projet ne couvre pas la réalisation des dispositifs permettant d'accéder les mémoires hors puce. Le transfert de données entre la ROM et les SRAM s'effectue virtuellement au début des simulations pour qu'ensuite le processeur ne fonctionne qu'avec deux mémoires SRAM (programme et données).

En outre, les médias utilisés sont une mémoire à double ports ainsi qu'une FIFO (Section 1.2.2.3). Cependant, d'autres composants, comme une mémoire à ports simples, pourraient être ajoutés pour augmenter la flexibilité. Les modèles VHDL des différents composants sont disponibles dans Seamless. Le logiciel "Memory Modeler of Denali" est utilisé pour générer le composant ayant les caractéristiques voulues (type, manufacturier, délais, dispositifs). L'annexe I résume les caractéristiques des modèles des composants utilisés.

Les mécanismes de synchronisation (interruptions, sémaphores, boîtes aux lettres) sont implémentés à l'aide de signaux de contrôle. Cependant, l'utilisation du protocole VCI, orienté vers le transfert de données, oblige à adopter des stratégies de conception particulières. Une partie de l'espace d'adressage est attribuée aux signaux de contrôle. Par exemple, une interruption est envoyée au processeur par un accès à

une adresse réservée et la logique de décodage de l'interface transmet ensuite l'interruption réelle.

Pour définir précisément l'interface, la valeur de paramètres doit être fixée et les premiers paramètres à spécifier correspondent à la définition de l'espace d'adressage et à la période de l'horloge. Une division de l'espace mémoire typique est proposée à l'annexe G. En plus, les paramètres des différents composants doivent être fixés (Tableau 2.2). De plus, si le protocole VCI est flexible, il faut identifier ses paramètres (annexe H).

Tableau 2.2. Paramètres des composants de communication

Composants de communication	Paramètres
SRAM du programme et des données du processeur	Manufacturier, grosseur de la mémoire, espace d'adressage, délais, grandeur des bus d'adresse et de données
Mémoires hors puce (ROM)	
DPRAM (Dual Port RAM)	Paramètres de la SRAM, dispositifs, nombre de boîtes aux lettres, espace d'adressage des boîtes aux lettres.
FIFO (le médium de communication ainsi que les FIFO à l'intérieur de l'interface)	Profondeur, espace d'adressage, signaux de synchronisation
Sémaphores	Nombre, espace d'adressage
Registre d'interruption (accéder par le processeur pour connaître l'état des interruptions)	Type d'accès, information disponible, configuration, espace d'adressage
Génération d'interruption (pour générer des interruptions au processeur)	Nombre, types, espace d'adressage

2.3.1.3. Limites de la génération automatique

À partir des paramètres proposés au Tableau 2.2, il est possible de générer des systèmes très différents. Cependant, les paramètres qui peuvent réellement être modifiés, à partir du programme, sont très limités (section J.1). Pour le moment, tout ce que le programme permet de changer est l'espace d'adressage des différents composants. Ainsi, le gaspillage de l'espace des mémoires sur la puce peut être minimisé en modifiant les grandeurs des mémoires.

Il serait intéressant de permettre la modification d'autres paramètres (énumérés dans le Tableau 2.2) et ainsi pouvoir optimiser le code en éliminant les parties inutilisées. De plus, simplifier le fichier de configuration pour augmenter l'automatisation et diminuer les responsabilités du concepteur serait essentiel. Enfin, il serait intéressant d'automatiser la génération des modèles de mémoire en fonction des paramètres.

En outre, l'interface est limitée aux composants donnés plus haut. Cependant, il serait intéressant, d'y ajouter d'autres composants pour augmenter la flexibilité. Par exemple, des mémoires FLASH, des mémoires avec des tailles de bus autres que 32 bits, des mémoires synchrones pourraient aussi être utiles.

2.3.2. Contexte d'utilisation

Pour intégrer l'IP dans un système, il faut bien comprendre les possibilités de l'interface de communication de cet IP. Les détails sur l'interface sont donnés au chapitre suivant. Il reste maintenant à préciser la topologie du système dans lequel l'IP devrait être utilisé. VSIA propose trois topologies utilisées pour l'assemblage de l'IP (Figure 2.9), mais précise que le protocole VCI est plus particulièrement destiné aux systèmes utilisant une connexion à travers un bus (Figure 2.9.B).

Ce projet traite seulement de la connexion point-à-point (Figure 2.9.A). Cela permet de réduire la complexité des systèmes implémentés et de compléter les premières analyses du protocole [CCS99] [LVG00] qui ne vérifient pas ses performances pour une connexion point-à-point. Dans [CCS99], un pont entre le protocole VCI et le bus AMBA est construit. Le présent projet permettra donc de compléter cette analyse en construisant une interface entre le protocole du processeur ARM et VCI.

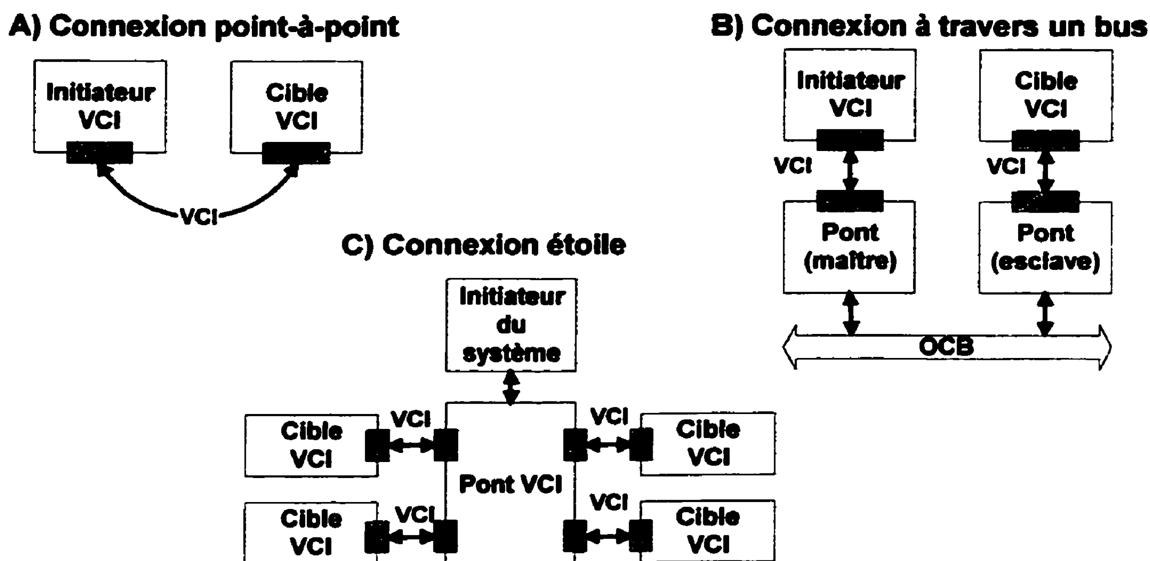


Figure 2.9. Topologies proposées par VSIA

La Figure 2.10 illustre comment transformer notre connexion point-à-point en une connexion avec un bus, en suivant le modèle proposé par Gajski (section B.7.2). Deux interfaces VCI sont utilisés pour que chaque IP puisse être à la fois maître et esclave.

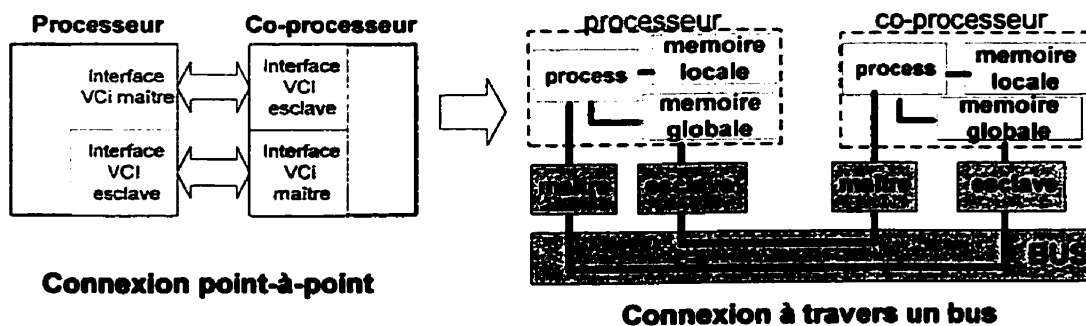


Figure 2.10. Topologie des systèmes réalisés dans le cadre du projet

Chapitre 3 : ANALYSE DU DESIGN DE L'INTERFACE

Ce chapitre est entièrement consacré à la description de l'interface contenant tous les mécanismes implémentés. D'abord, les choix d'implémentation généraux sont expliqués. Puis, une fois que le niveau hiérarchique supérieur de l'interface est défini, chacun des modules sera décrit plus en détail. Un complément d'informations est donné à l'annexe J et le code VHDL complet est disponible sur demande.

3.1. Choix d'implémentation

Dans cette section, une vue générale de l'interface est donnée. D'abord, les techniques utilisées pour combler les limitations du protocole VCI, la méthode de synchronisation de l'interface et la structure des modules sont présentées. Ensuite, après avoir donné une description générale du niveau hiérarchique supérieur de l'interface, la structure des composants mémoires sera fournie.

3.1.1. Limitations du protocole VCI

L'utilisation d'un protocole VCI impose certaines limitations. D'abord, pour permettre des communications bidirectionnelles, deux interfaces VCI sont utilisées, l'interface maître et l'interface esclave. Ces interfaces sont décrites en détail aux sections suivantes. Ensuite, l'absence de signaux spécifiques au contrôle force à adopter une stratégie particulière : la division de l'espace mémoire. Enfin, l'utilisation du mécanisme de poignée de main et l'impossibilité de transférer une requête et une réponse dans un même cycle complexifie la tâche de conception.

3.1.1.1. Espace d'adressage

Pour permettre l'utilisation de signaux spécifiques de contrôle, des ajouts aux signaux VCI auraient pu être faits. Cependant, pour maintenir la compatibilité de l'interface, l'espace d'adressage est divisé et une partie est attribuée aux signaux de contrôle.

Ainsi, chaque composant de communication a un espace mémoire propre et la logique de décodage est utilisée pour générer les signaux de contrôle au besoin.

Pour diviser l'espace mémoire, VSIA propose d'attribuer une partie des bits pour identifier un composant et le reste pour contenir l'adresse. Étant donné l'identification d'un composant effectuée à partir d'un nombre limité de bits, la logique de décodage est simple. Cependant, la division est peu optimale étant donné qu'un espace identique est attribué à chaque composant. Dans ce projet, le nombre de bits d'adresse de chacun des composants est très différent et une autre solution doit être envisagée.

Dans [COB95b], l'algorithme d'Huffman est utilisé lorsque l'espace disponible est limité. Néanmoins, la complexité de l'algorithme a mené à l'utilisation d'une autre méthode. L'espace d'adressage est défini par l'association de deux adresses à chaque composant. Cela permet d'optimiser l'espace d'adressage, mais augmente un peu la complexité de la logique de décodage.

3.1.1.2. Mécanisme de poignée de main

L'utilisation du mécanisme de poignée de main peut grandement diminuer l'efficacité des communications entre les IP. D'abord, étant donné que chaque transfert s'effectue sur une montée d'horloge, une requête et une réponse ne peuvent être transmises dans un même cycle. En outre, si les signaux du mécanisme de poignée de main des IP ne sont pas coordonnés correctement, un transfert peut s'étendre sur plus d'un cycle. Au cours des prochains chapitres, des stratégies de conception spécifiques à ce problème ainsi que les conséquences de ces limitations sont données.

Il importe de préciser les délais considérés. Dans les spécifications, il est suggéré que les sorties soient stables 20 % après la montée de l'horloge et que les entrées soient stables 20 % avant la montée de l'horloge. Dans le cas d'une connexion point-à-

point, les sorties correspondent aux entrées. Si les spécifications sont respectées, un coup d'horloge est perdu à chaque transfert (Figure 3.1.a). Pour éviter cette perte, les entrées sont captées au moment où les sorties sont valides. La génération des sorties respecte les contraintes de VSIA, mais les entrées sont captées plus tôt (Figure 3.1.b). Le non-respect de cette contrainte ne devrait normalement pas affecter le fonctionnement de l'interface, mais seulement augmenter le délai des communications. Cependant, il est possible que l'interface entre dans un état de méta stable et ce problème reste encore à voir.

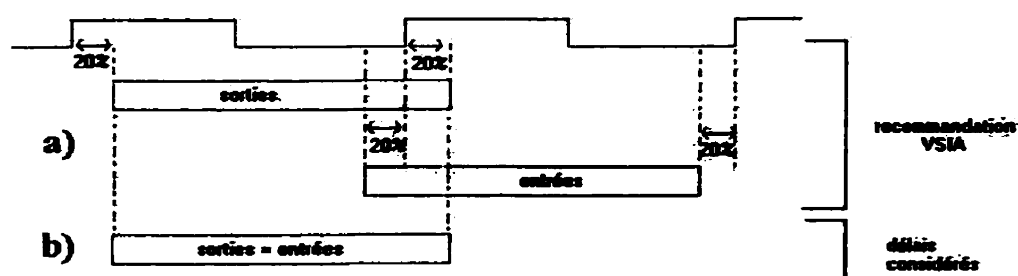


Figure 3.1. Spécifications des délais du protocole BVCI
a) proposées par VSIA, b) utilisées dans le projet

3.1.2. Synchronisation

La grande difficulté de la conception de l'interface est la synchronisation. En effet, le fonctionnement de ces composants repose sur le respect des contraintes de temps. Au départ, étant donné l'impossibilité de respecter toutes les contraintes avec une seule horloge, une approche de conception asynchrone était utilisée. Cependant, comme mentionné à la section 1.1.6.7, la construction et la synthèse de modules asynchrones sont ardues. De plus, les règles de réutilisation proposées dans [KeBr99] bannissent les designs asynchrones. Cette approche fut donc rejetée.

L'idéal est d'avoir un design synchrone, qui respecte les délais de tous les composants et dont toutes les bascules sont actives sur le front montant. Puisque dans notre cas le design avec une seule horloge est impossible, deux horloges sont utilisées. D'abord,

pour simplifier la conception, les horloges de processeur et de celle de l'interface VCI sont identiques.

L'interface est construite sous le modèle d'un pipeline où les informations passent d'un étage à l'autre suivant une certaine fréquence. Les signaux à la sortie de chacun des étages sont générés à des moments différents du cycle VCI. Pour respecter les contraintes des mémoires, du processeur et de VSIA, l'horloge du pipeline est fixée à 10 fois l'horloge VCI (ou du processeur). La fréquence VCI théorique est de 25MHz (fréquence maximale du modèle du processeur sur Seamless) donc la fréquence du pipeline (de l'interface) est de 250MHz. Le problème majeur de cette solution est que la vitesse de fonctionnement est très élevée et il faudra analyser les résultats de synthèse pour vérifier si l'hypothèse est justifiée.

Un petit contrôleur est utilisé pour générer la fréquence de l'interface ainsi que le signal d'états (Figure 3.2). Ce signal permet de synchroniser les différents modules de l'interface et de diminuer la puissance dissipée. En effet, il indique le moment de rafraîchissement des bascules. Ainsi, les signaux du système respectent les contraintes des composants et les changements d'états des bascules sont minimisés.

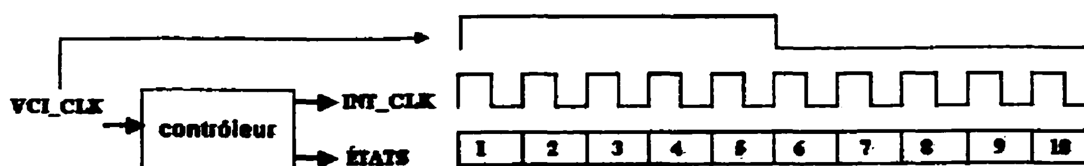


Figure 3.2. Horloges et signal de synchronisation de l'interface

Pour contrôler les bascules, des signaux d'activation sont générés, à l'intérieur de chaque module, à partir du signal global d'états. En modifiant ce bloc de contrôle (gris foncé sur la Figure 3.3), l'interface peut facilement être adaptée aux changements d'architecture. Pour des changements majeurs de composants,

l'interface ne pourra pas être adaptée aussi facilement. Par exemple, l'ajout de mémoires hors puce pourrait impliquer l'ajout de machines à états complexes.

3.1.3. Structure des modules

L'interface est divisée en plusieurs modules qui sont construits suivant une structure typique (Figure 3.3). La description des modules suit la plupart des règles de design données dans [KeBr99]. Toutefois, le code VHDL généré par Renoir ne respecte pas toutes ces règles. En plus, comme expliqué au chapitre précédent, une sémantique particulière a été utilisée pour faciliter la configuration automatique.

Les modules sont conçus de façon à séparer le chemin de données et le contrôleur (machine à états générant les signaux de contrôle), comme proposé dans [BPM98] [SuBr92] [MaHa95]. De cette façon, chaque module contient, en plus du bloc de contrôle décrit à la section précédente, un ou plusieurs chemins de données ainsi que leur contrôleur. Chaque chemin de données peut contenir plusieurs étages (pipeline).

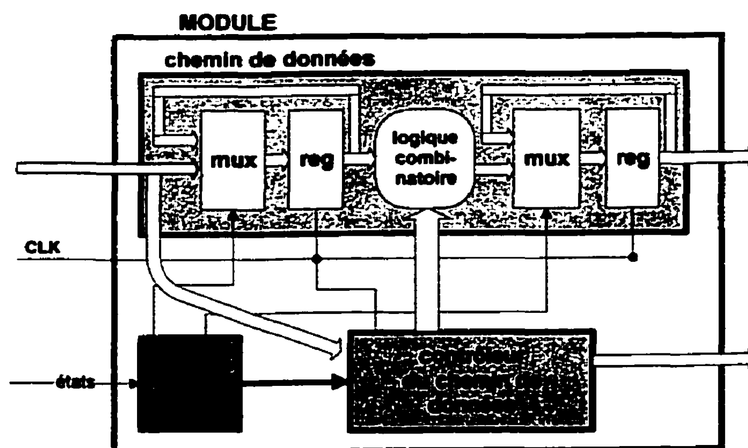


Figure 3.3. Structure typique des modules de l'interface

Deux stratégies peuvent être utilisées pour permettre le contrôle des bascules. D'abord, le bloc de contrôle peut générer des horloges dérivées pour chacune des bascules. Cette approche est déconseillée dans [KeBr99] et [FaWa00] à cause des

problèmes de biais de synchronisation et de testabilité. L'alternative consiste à utiliser une seule horloge et de faire le contrôleur à l'aide d'activateurs de bascule ou de sélecteurs de multiplexeurs. Le premier cas est à rejeter à cause de sa complexité d'insertion de la chaîne de balayage (une bascule avec activateur est plus difficile à transformer en bascule de test). Nous croyons que le second cas est le plus approprié. Même si la quantité de silicium est légèrement augmentée, la puissance dissipée (fonction du changement d'états des sorties des bascules) et la complexité de l'insertion de la chaîne de balayage sont respectables.

3.1.4. Description du niveau hiérarchique supérieur

La Figure 3.4 montre le diagramme bloc de l'interface structurée dans un objectif de synthèse. Toutes les boîtes foncées sont des modules qui ne seront pas synthétisés parce qu'ils correspondent à des composants de Seamless (mémoire ou processeur). Dans la phase d'implémentation finale, ils peuvent être remplacés par des composants physiques disponibles sur le marché. La boîte gris pâle contient tous les modules synthétisables de l'interface, tout ce qui est construit dans le cadre du présent projet.

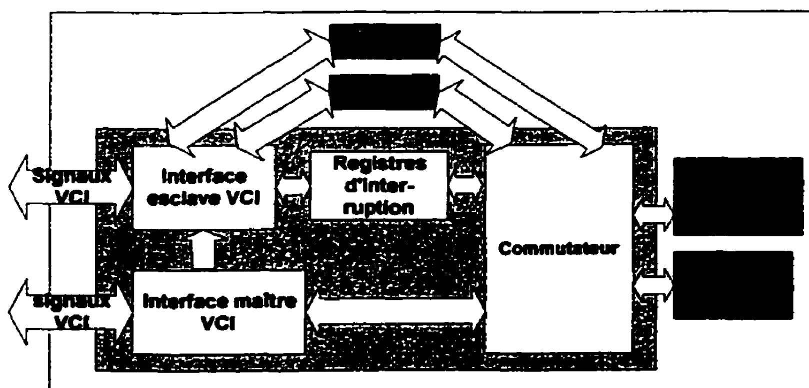


Figure 3.4. Diagramme bloc structuré de l'interface

Pour mieux comprendre le fonctionnement de chacun des principaux modules, une représentation plus fonctionnelle de l'interface est nécessaire (Figure 3.5). D'abord, l'interface spécifique du processeur contient le processeur et ses mémoires ainsi qu'un

commutateur qui gère la transmission des données d'une instance à l'autre. Ensuite, des registres d'interruptions matériels permettent au processeur de recevoir plusieurs types d'interruptions et de connaître les états de celles-ci. Pour communiquer avec l'extérieur, une interface maître et une interface esclave sont utilisées. Le lien entre ces interfaces et le processeur se fait à travers une FIFO ou une DPRAM (Dual Port Randon Access Memory).

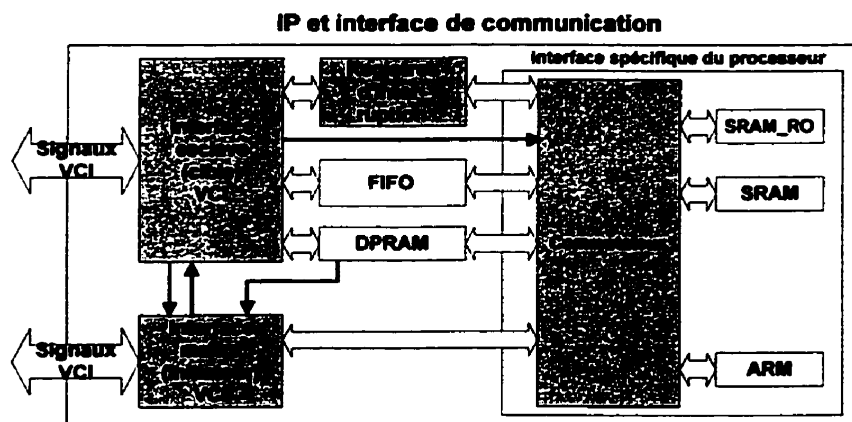


Figure 3.5. Diagramme bloc fonctionnel de l'interface

3.1.5. Composants mémoires

Avant de décrire le fonctionnement et les détails sur le design de chacun des quatre modules principaux, il importe d'expliquer comment les composants de Seamless ont été utilisés. L'annexe I donne des détails sur le fonctionnement des trois composants utilisés : la mémoire SRAM, la mémoire DPRAM et la FIFO.

D'abord deux mémoires SRAM de 32 bits sont utilisées pour le programme et les données du processeur. La première (SRAM_RO) ne permet pas les écritures (à part à l'allumage) et donc le signal d'écriture est toujours inactif. De plus, elle est construite à partir de deux mémoires de 16 bits auxquelles l'accès est toujours simultané (les instructions sont toujours de 32 bits). La deuxième (SRAM) est

construite à partir de quatre mémoires de 8 bits. Ainsi il est possible d'avoir accès aux octets de façon indépendante.

La DPRAM est aussi construite à partir de quatre mémoires de 8 bits (Figure 3.6). À partir des signaux d'activation (ndpram_cs) de la mémoire et des octets (ndpram_be), il est possible d'activer les bons blocs mémoire pour avoir une donnée de la grandeur voulue. La DPRAM a deux ensembles de ports auxquels on peut avoir accès par différents composants simultanément. Cependant, pour simplifier le schéma, seulement un ensemble est représenté à la Figure 3.6.

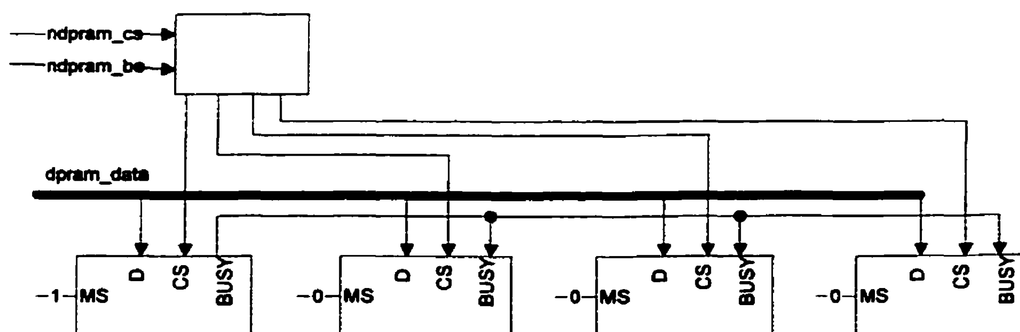


Figure 3.6. Construction d'une mémoire 32 bits à partir d'une 8 bits

Pour faciliter la synchronisation des composants, la DPRAM incorpore des sémaphores, des boîtes aux lettres ainsi qu'un mécanisme d'arbitrage interne. Cependant, la cascade de plusieurs mémoires oblige à adopter une stratégie d'arbitrage particulière. Pour éviter que les arbitres de chacun des blocs prennent des décisions opposées, seulement l'arbitre d'un des blocs est utilisé. Ce maître (MS = 1) génère le signal BUSY des autres composants et décide du port gagnant.

La FIFO est le dernier composant mémoire implanté. Deux FIFO sont utilisées pour permettre une communication bidirectionnelle (Figure 3.7) . Une adresse est associée à chacune des FIFO pour simplifier la configuration dans Seamless. Les indicateurs de la FIFO pleine et vide permettent de générer les erreurs VCI, mais pour le moment,

aucune erreur n'est envoyée au processeur. De plus, les erreurs sont générées lors de la dernière lecture ou écriture valide. En conséquence, si les erreurs sont considérées, il est conseillé de ne jamais remplir ou vider entièrement la FIFO. Enfin, pour construire la FIFO d'une largeur de 32 bits, quatre FIFO de 8 bits sont employées. Cependant, il faut faire très attention lors d'accès de moins de 32 bits car les octets des différents mots peuvent être totalement mélangés (Figure 3.8).

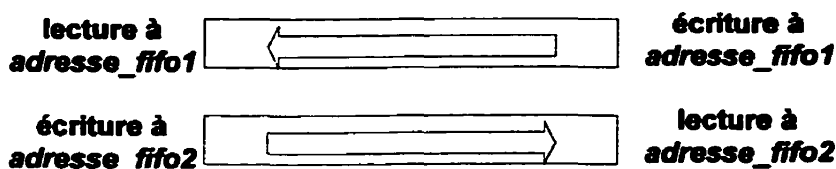


Figure 3.7. Représentation du double FIFO

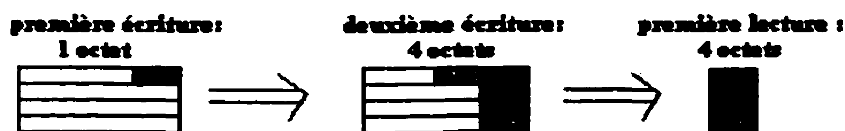


Figure 3.8. Accès non alignés à la FIFO

3.2. Registres d'interruptions

Le registre d'interruptions est un des quatre composants de base de l'interface. D'abord, un aperçu de l'utilité et une description des registres seront donnés. Puis, la structure sera décrite.

3.2.1. Utilité des registres

Le processeur peut recevoir des interruptions directes ou provenant des boîtes aux lettres. Le processeur a un seul port d'interruption générale (IRQ), ainsi l'information sur la routine à exécuter doit donc être inscrite quelque part. Puisque les boîtes aux lettres ont un espace mémoire réservé, le processeur peut y lire l'information recherchée. Dans le cas des interruptions directes, les registres d'interruptions

contiennent le numéro des interruptions activées. Il faut noter que dans un système sans DPRAM (sans boîtes aux lettres), les registres d'interruptions sont essentiels.

Tout ce qui concerne le traitement des interruptions ainsi que les priorités de celles-ci se fait en logiciel. Lorsque le processeur reçoit une interruption, il entre dans une routine spéciale et selon les informations trouvées dans les registres, la bonne fonction est exécutée. L'annexe K.2 donne un exemple de routine en assembleur.

Le module de registres a été conçu en suivant le modèle du contrôleur d'interruptions (Figure G.10) donnée dans [ARM00e]. Il est possible d'avoir jusqu'à 32 sources d'interruptions selon les paramètres fixés dans le "package". Le nombre de bits de chacun des registres est configuré en fonction du nombre de sources et chaque bit est associé à une source. Toutes les interruptions passent par l'interface esclave et l'activation d'un des bits du bus de données indique le numéro de l'interruption envoyée. La conception de systèmes contenant plusieurs composants source d'interruptions implique une gestion particulière de l'attribution des différentes interruptions pour que les bonnes routines soient exécutées.

3.2.2. Description des registres

Le module contient six registres qui peuvent être écrits et lus par l'interface esclave (IE) ou par le processeur (P). Le Tableau 3.1 décrit chacun des registres. Lorsque l'interface esclave reçoit un accès à l'adresse destinée aux interruptions, elle écrit le numéro de l'interruption reçu dans le registre *Status*. Si cette interruption est en traitement, l'écriture n'est pas considérée. De plus, l'interface esclave peut lire les registres *Status* et *Treated* pour connaître les états du traitement des interruptions.

Tableau 3.1. Description des différents registres

Nom	Lecture	Écriture	Description
Status	P + IE	IE + P	Indique les interruptions envoyées sans être traitées
Raw_status	P		Indique les interruptions envoyées et activées sans être traitées (statut réel des interruptions envoyées au processeur)
Enable_set		P	Permet d'activer une ou plusieurs interruptions
Enable_clear		P	Permet de désactiver une ou plusieurs interruptions
Enable	P		Indique les interruptions activées
Treated	IE	P	Indique les interruptions en traitement

Le processeur a accès aux registres par le biais des adresses données au Tableau 3.2. L'activation et la désactivation des interruptions (*Enable_set* et *Enable_Clear*) permettent au processeur d'établir des niveaux de priorité. Le registre *Enable* reflète l'état d'activation des interruptions et le registre *Raw_status,* les interruptions actives envoyées par l'interface esclave.

Tableau 3.2. Espace d'adressage des registres d'interruptions

Adresse	Emplacement de lecture	Emplacement d'écriture
Base + 00	Status	Réservé
Base + 04	Raw_status	Treated
Base + 08	Enable	Enable_set
Base + 0C	Réservé	Enable_clear

L'écriture au registre *Treated* identifie les interruptions en traitement. C'est la responsabilité du concepteur logiciel d'écrire dans le registre *Treated* pour indiquer le début (registre à 1) et la fin du traitement (registre à 0). Écrire 1 dans le registre *Treated* restaure automatiquement le bit correspondant du registre *Status*. Ainsi, aucune écriture au bit du registre *Status* d'une interruption en traitement n'est possible.

3.2.3. Structure des registres

Exceptionnellement, aucun contrôleur n'est utilisé dans ce module. Les sorties des bascules sont rafraîchies à toutes les montées d'horloge. Le module est divisé en quatre blocs (Figure 3.9). D'abord, un bloc (générateur d'interruption) est responsable de maintenir la valeur du registre *Status* en fonction du registre *Treated* et des

interruptions envoyées par l'interface esclave. Puis, un bloc traite la lecture et l'écriture des registres par le processeur. Un autre module (contrôleur du "enable") est responsable de maintenir la valeur du registre *Enable* en fonction des registres *Enable_Set* et *Enable_Clear*. Enfin, en fonction de la valeur du registre *Raw_Status*, une interruption est envoyée au processeur. De plus, la valeur des registres *Treated* et *Status* est envoyée à l'interface esclave.

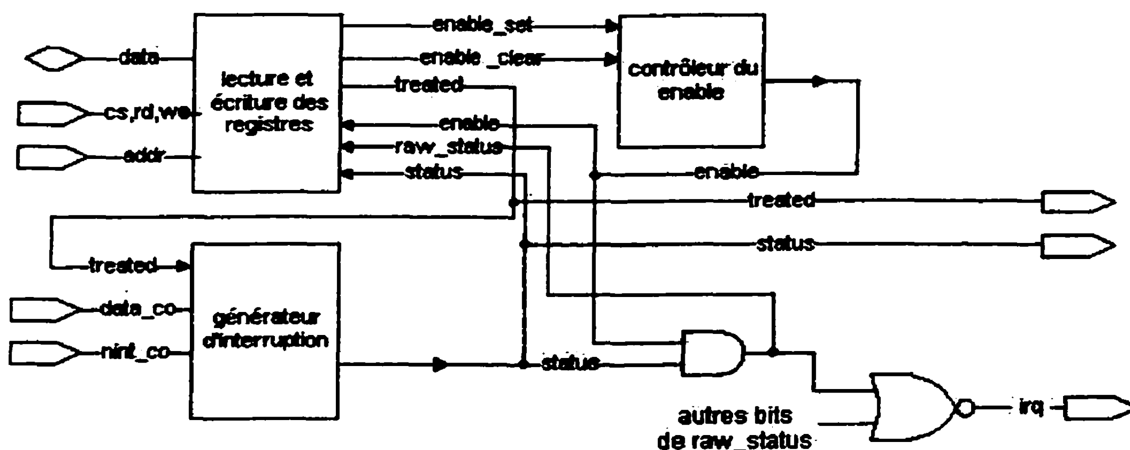


Figure 3.9. Schéma bloc des registres d'interruptions

Les modules permettant de faire la lecture et l'écriture des registres par le processeur doivent être conçus de façon méticuleuse, car les délais du processeur doivent être respectés et l'information transmise doit rester valide. D'abord, les signaux du processeur doivent être valides suffisamment de temps avant la montée de l'horloge pour que les entrées des bascules puissent se stabiliser. De plus, l'adresse et la donnée (pour lecture) doivent être valides pendant toute la période d'activation de l'écriture ou de la lecture et cette période doit être d'au moins deux coups d'horloge. Étant donné que les signaux du processeur sont synchronisés, toutes les conditions sont remplies. Les schémas logiques et diagrammes temporels des blocs de lecture et écriture sont donnés (Figure 3.10 et Figure 3.11).

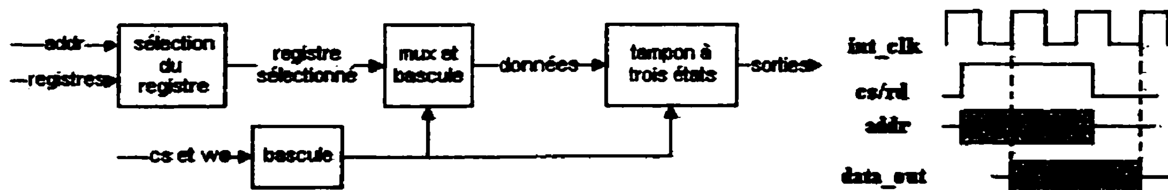


Figure 3.10. Schéma logique et diagramme temporel du bloc de lecture

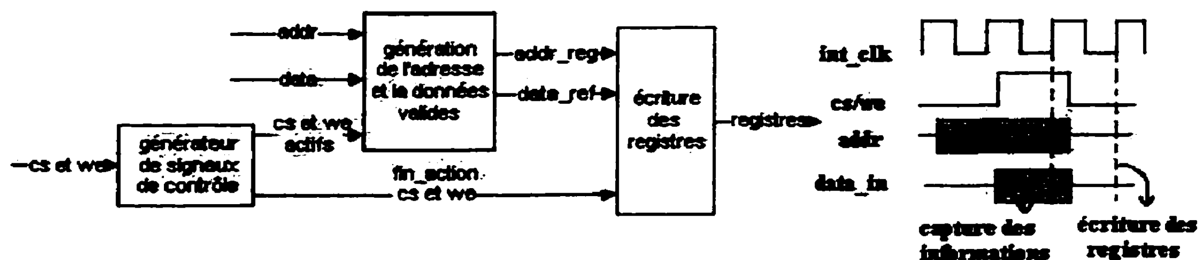


Figure 3.11. Schéma logique et diagramme temporel du bloc d'écriture

3.3. Commutateur

Le fonctionnement du commutateur est décrit par sa structure et ses diagrammes temporels. Ensuite, les contraintes à respecter pour un bon fonctionnement du commutateur sont expliquées.

3.3.1. Description du commutateur

Les rôles du commutateur sont de générer le signal d'interruption du processeur, de transmettre les requêtes du processeur à l'interface maître et d'effectuer les accès mémoire du processeur. D'abord les signaux d'interruption des boîtes aux lettres et des registres d'interruptions sont combinés pour obtenir le signal IRQ du processeur (simple logique combinatoire). Ensuite, les signaux du processeur sont tout simplement synchronisés avant d'être traités par l'interface maître. Enfin, la tâche la plus délicate du commutateur est de transformer les signaux d'accès mémoire du processeur en signaux accessibles aux mémoires (SRAM, DPRAM et FIFO).

3.3.1.1. Synchronisation

Le commutateur (Figure 3.12) est bâti selon le modèle donné à la section 3.1.3 et la synchronisation se fait à partir des signaux du bloc de contrôle. Aucun contrôleur de chemin de données n'est nécessaire. Le diagramme temporel (Figure 3.13) montre comment les signaux de contrôle sont générés pour coordonner la transmission des signaux aux mémoires. La légende de la Figure 3.13 est aussi valable pour la Figure 3.12. Lorsque le processeur effectue un accès mémoire (indiqué par le signal nMreq), les bons signaux mémoire sont générés à partir des signaux du processeur, tout en respectant les contraintes de tous les composants.

Pour faciliter les transferts, les bus unidirectionnels (Din, Dout) du processeur sont utilisés plutôt que le bus bidirectionnel (D). Ainsi, lors d'une écriture, les données sur le bus Dout sont envoyées à la bonne mémoire. Lors d'une écriture, la mémoire sélectionnée envoie les données sur le bus Din. Les signaux Rd et We (lecture et écriture) indiquent aux mémoires le type d'accès. Pour tous les composants, la lecture se fait à l'activation de Rd et l'écriture se fait à la désactivation de We.

3.3.1.2. Détails de conception

À l'entrée du bloc, les signaux du processeur sont mémorisés dans des bascules, avant d'être transformés en signaux mémoire. Un autre étage de bascules permet de synchroniser la génération de ces signaux. Enfin, un bloc de logique combinatoire, à la sortie du dernier étage de bascule, est nécessaire pour la génération des signaux de lecture et d'écriture, étant donné que les mémoires sont asynchrones. La gestion des données envoyées au processeur (Din) n'est pas montrée à la Figure 3.12.

Pour générer les signaux d'accès mémoire à partir des signaux du processeur (Tableau 3.3), quatre blocs sont construits. D'abord, à partir de l'adresse de 32 bits, les bits les plus significatifs sont utilisés pour générer le bon signal d'activation. Puis, l'adresse mémoire est générée à partir des bits restants (pas les deux moins significatifs). La

grosseur de modules précédemment décrits sera affectée par les choix effectués à la section 3.1.1.1. Les deux modules suivants permettent de générer les signaux de lecture et d'écriture ainsi que les signaux d'activation d'octets. Le circuit d'activation d'octets est construit à partir du circuit proposé dans [ARM00b] (Figure G.9).

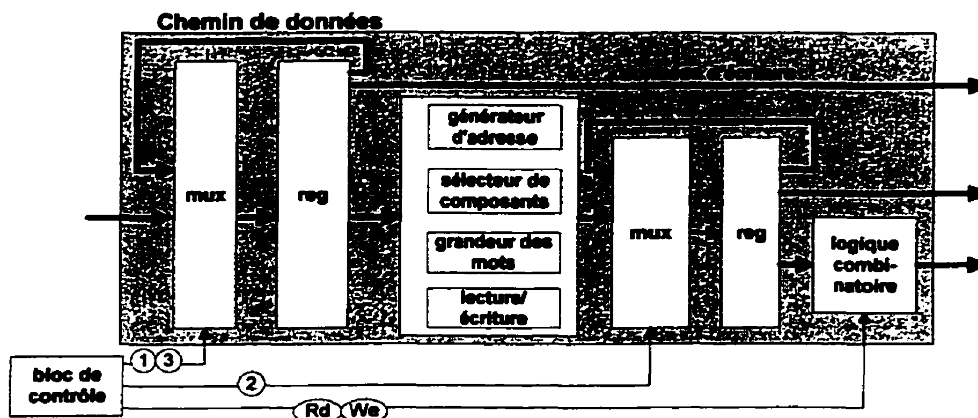


Figure 3.12. Schéma bloc du commutateur

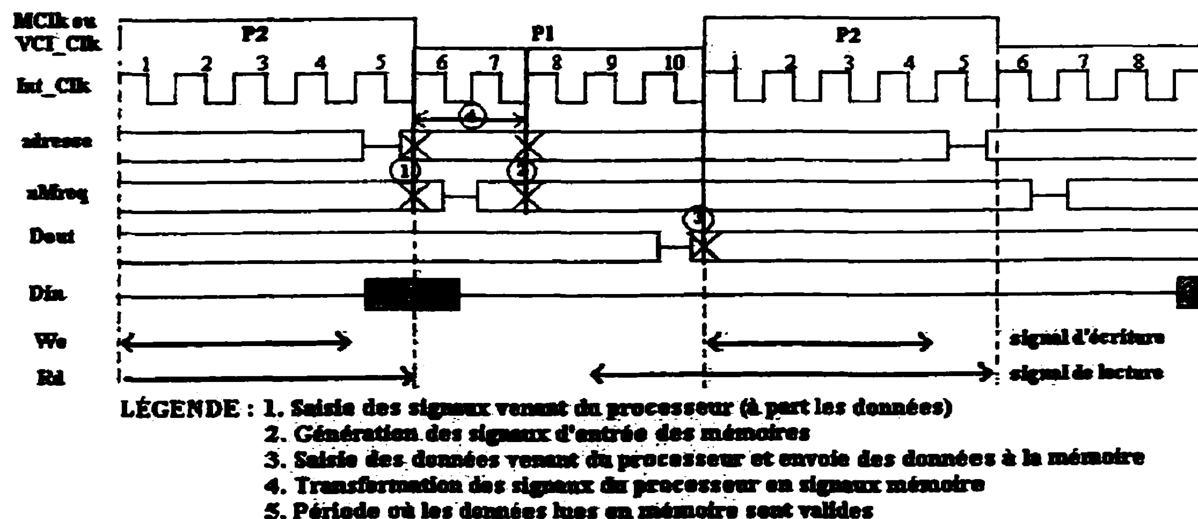


Figure 3.13. Diagramme temporel du commutateur

Tableau 3.3. Association des signaux du processeur aux signaux mémoires

[illegible]	[illegible]	[illegible]
A[31 :2]	Cs, Sem, Int_Reg Add[X :0]	Les bits les plus significatifs permettent d'identifier le composant auquel on a accédé : les mémoires (Cs), les sémaphores (Sem) ou les registres (Int_reg). Les bits restants sont utilisés pour adresser le composant.
Din[31 :0] + Dout[31 :0]	D[31 :0]	Le processeur a 2 bus unidirectionnels et les mémoires, un bus bidirectionnel.
Mas[1 :0] + A[1 :0]	Be[3 :0]	Le signal d'activation d'octets des mémoires est déduit des signaux de la taille de la données (Mas) et des octets adressés (A[1 :0]) du processeur.
NRW	We + Rd	Le processeur a un signal de lecture/écriture et les mémoires en ont deux.

3.3.2. Contraintes d'utilisation du commutateur

L'interface est conçue pour fonctionner à la fréquence maximale acceptée par le modèle du processeur sur Seamless. Ceci correspond à une fréquence VCI (et du processeur) de 25MHz et une fréquence de pipeline de 250MHz. Ainsi, pour augmenter la vitesse du circuit (sans vouloir le simuler avec Seamless), le contrôleur et l'architecture devraient être modifiés. De plus, l'interface a été conçue pour fonctionner à cette vitesse, en supposant que la technologie utilisée le permet. Par exemple, on accorde deux cycles de pipeline (zone 4 à la Figure 3.13) pour effectuer la transformation des signaux du processeur en signaux mémoires. Il faut donc que la technologie utilisée puisse effectuer ces opérations en 8ns pour une fréquence VCI de 25MHz (période de 40ns). Au-delà des contraintes de synthèse, les mémoires doivent aussi respecter certaines contraintes temporelles données au Tableau 3.4.

Tableau 3.4. Restriction des délais des mémoires

[illegible]	[illegible]
LECTURE	
Délai entre le *début* de la période d'activation de lecture (rd) et la donnée valide	< 15 ns
Délai entre l'adresse et le signal de sélection (add et cs) valide et la donnée valide	< 20 ns
Temps de maintien de la donnée valide après la désactivation du signal de lecture (rd)	> 2ns
Temps de maintien de la donnée valide après la désactivation cs et add	> 0ns
ÉCRITURE	
Délai entre l'écriture réelle et la *fin* de la période d'activation de l'écriture (we)	< 20 ns
Le délai entre l'écriture réelle d'un sémaphore et la *fin* de la période d'activation de we est supérieur au délai d'une écriture normale et donc il faut attendre avant de faire la lecture du sémaphore.	
Les données doivent être valides jusqu'à la *fin* de la période d'activation de l'écriture	

3.4. Interface esclave

Avant de présenter globalement l'interface esclave, la description des requêtes et réponses VCI est d'abord exposée. Ensuite, ses deux chemins de données sont décrits plus en détail.

3.4.1. Description des requêtes et des réponses

Le rôle de l'interface est d'abord, de transformer la requête VCI en un accès au bon composant puis, de renvoyer une réponse VCI de la requête et des données retournées par les mémoires. Les requêtes et réponses possibles sont expliquées. Le Tableau 3.5 résume les requêtes externes traitées par l'interface esclave.

Les commandes peuvent être des lectures, des écritures ou des opérations vides. Les lectures bloquantes sont interprétées comme des lectures normales. Les transferts de données se font à la DPRAM ou à la FIFO et ces composants sont décrits à la section 3.1.5. Trois mécanismes de synchronisation sont disponibles : les sémaphores, les boîtes aux lettres ou les interruptions directes. Les accès aux sémaphores sont simples, ils consistent en une écriture ou une lecture dans l'espace d'adressage approprié. Néanmoins, le concepteur est responsable de bien les utiliser.

3.4.1.1. Boîtes aux lettres

Les boîtes aux lettres (MB) peuvent être utilisées par les composants externes pour envoyer des interruptions au processeur (écriture à la MB (MailBox) de droite) ou par le processeur pour envoyer des interruptions aux composants externes (écriture à la MB de gauche). Une interruption est effacée par une lecture à la MB de son côté. Pour recevoir des interruptions de boîtes aux lettres externes, l'adresse de la MB externe est utilisée. Ainsi, l'interface esclave peut détecter ces accès particuliers et envoyer une interruption au processeur. Les différents scénarios sont illustrés à la Figure 3.14.

Tableau 3.5. Description des requêtes traitées par l'interface esclave

Composant	Type d'accès	[illegible]	[illegible]
DPRAM (données)	Écriture	10	Adresses de l'espace mémoire de la DPRAM
	Lecture	01 ou 11	
Sémaphores	Écriture	10	Adresses de l'espace mémoire des sémaphores
	Lecture	01 ou 11	
Boîte aux lettres –> écriture pour générer une interruption	Écriture	10	Adresse de la boîte aux lettres de gauche
Boîte aux lettres –> lecture pour effacer une interruption	Lecture	01 ou 11	Adresse de la boîte aux lettres de droite
Boîte aux lettres –> générer une interruption de source externe	Peu importe	XX	Adresse de la boîte aux lettres externe de droite
FIFO (données)	Écriture	10	Adresse de la FIFO où les données circulent de l'extérieur vers l'intérieur
	Lecture	01 ou 11	Adresse de la FIFO où les données circulent de l'intérieur vers l'extérieur
Interruptions directes - avec attente, sans déviation - sans attente, sans déviation - avec attente, avec déviation - sans attente, avec déviation	Opération vide (NOP)	00	Adresse de base des interruptions + 00 01 10 11

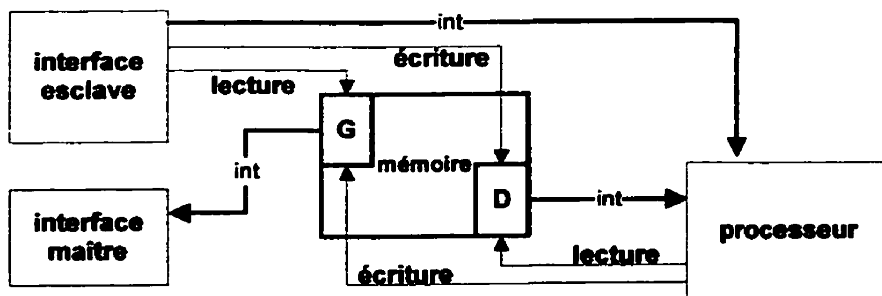


Figure 3.14. Accès aux boîtes aux lettres et génération d'interruption

En somme, l'interface esclave traite trois types de requêtes liées aux MB : l'écriture à la MB de gauche pour générer une interruption au processeur, la lecture de la MB de droite pour retirer une interruption envoyée à un composant externe ou l'écriture à l'adresse d'une MB externe pour générer une interruption au processeur. Dans le dernier cas, le processeur devra aller lire la MB de la mémoire externe pour retirer l'interruption. L'interface maître est responsable de générer l'accès VCI

correspondant à la génération d'une interruption par la MB (droite). De plus, elle doit détecter les lectures VCI à une MB externe, pour envoyer un signal à l'interface esclave et indiquer que l'interruption par MB externe doit être retirée.

3.4.1.2. Interruptions directes

Les adresses réservées aux interruptions permettent l'envoi d'interruptions directes au processeur. L'interface esclave détecte cet accès et ajuste le registre de statut des interruptions. L'interruption peut se faire avec ou sans attente et avec ou sans déviation, donc quatre possibilités découlent de ces propriétés. L'opération de la requête doit être un NOP et l'adresse indique le type d'interruption (Tableau 3.5).

Pour expliquer les différentes options, il faut comprendre que normalement toutes les requêtes de l'interface esclave sont déposées dans une FIFO interne à l'interface esclave. Cela permet d'attendre le traitement des requêtes précédentes. Néanmoins, une interruption peut être envoyée directement au processeur, sans attendre le traitement des requêtes précédentes (avec déviation : Figure 3.15.A), ou en passant par la FIFO (sans déviation : Figure 3.15.B). De plus, l'interruption peut être considérée comme étant traitée soit à l'écriture du registre d'interruptions (sans attente) ou soit au traitement réel de l'interruption (avec attente).

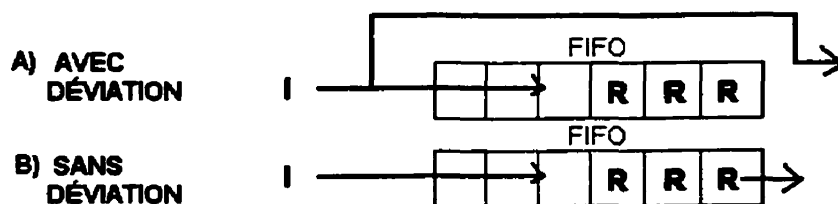


Figure 3.15. Types d'interruptions directes

Lorsque la propriété d'attente est utilisée, aucune nouvelle requête ne sort de la FIFO (aucun "pop") tant que l'interruption n'est pas traitée et les nouvelles requêtes sont empilées dans la FIFO tant qu'il n'est pas pleine. Si l'interruption est déjà en traitement ou en attente de traitement (registres *Treated* et *Status*), l'interface n'est pas bloquée et les requêtes continuent à être traitées. Il faut faire attention de ne pas bloquer la FIFO indéfiniment à cause de l'attente d'interruptions désactivées.

Lorsque la propriété de déviation est utilisée, la requête d'interruption est empilée dans la FIFO pour que l'ordre des requêtes et réponses soit respecté. Durant ce cycle, rien ne peut sortir de la FIFO, car c'est l'interruption qui est générée (Figure 3.15). Au cycle où la requête d'interruption avec déviation sort de la FIFO, l'interface ne fait qu'envoyer la réponse VCI associée à la requête. Ainsi, à moins que la FIFO ne soit vide, deux cycles sont donc utilisés pour une requête d'interruption avec déviation.

3.4.1.3. Génération d'erreurs

La réponse VCI consiste globalement en une donnée (pour une lecture) et un signal d'erreur. Le signal d'erreur est d'un seul bit et est activé dans quatre situations. La première source d'erreur est l'utilisation d'une commande ou d'une adresse non valide. Ensuite, lorsqu'un accès active un des indicateurs de la FIFO (composant mémoire), une erreur est générée. Cependant, il faut faire attention à ce type d'erreur comme expliqué à la section 3.1.5. Enfin, une erreur est envoyée lorsque l'interface esclave tente d'activer une interruption qui est déjà en traitement ou en attente de traitement. Aucune erreur n'est pas envoyée dans le cas d'interruptions avec déviation puisque le traitement de la requête et l'envoi de réponse ne sont pas simultanés.

3.4.2. Description globale de l'interface esclave

La présentation de la structure générale de l'interface est axée sur l'explication de son mode de synchronisation. Puis, les paramètres VCI sont exposés.

3.4.2.1. Synchronisation

Le modèle de l'interface esclave (Figure 3.16) est le même que celui décrit à la section 3.1.3. Tous les registres sont précédés de multiplexeurs synchronisés par le bloc de contrôle. Pour permettre la circulation des données dans les deux sens (requête et réponse), deux chemins de données sont utilisés avec leur contrôleur. Pour assurer la synchronisation et la fonctionnalité de l'interface, les contrôleurs des chemins de données et le bloc de contrôle sont reliés par des signaux réservés.

À partir du moment où une requête est envoyée à l'interface, les étages sont franchis en suivant les délais montrés à la Figure 3.17. Les signaux représentés dans le diagramme temporel sont identifiés sur la Figure 3.16 et la légende des deux figures est la même. Cela permet de voir quand les sorties des bascules de chacun des étages sont rafraîchies ainsi que la période de validité des signaux. Comme expliqué à la section 3.1.1.2, si les requêtes VCI ne sont pas valides 20 % après la montée de l'horloge, le traitement de la requête sera retardé d'un cycle. De plus, si le signal d'accusé de réception de la réponse n'est pas capté 10 % avant la montée de l'horloge, le traitement d'une nouvelle requête sera reporté au cycle suivant. Le non-respect de ces contraintes pourraient aussi engendré, dans des cas extrêmes, une possibilité de méta stabilité.

3.4.2.2. Paramètre VCI

Pour le moment, les paramètres VCI ne sont pas flexibles, mais il serait intéressant d'ajouter cette possibilité à l'interface. Néanmoins, pour faciliter l'intégration de l'IP dans un système, il est primordial de documenter tous les paramètres. La description détaillée de chacun des paramètres est donnée à la section B.4 et la valeur de chacun des paramètres dans l'interface est présentée au Tableau 3.6.

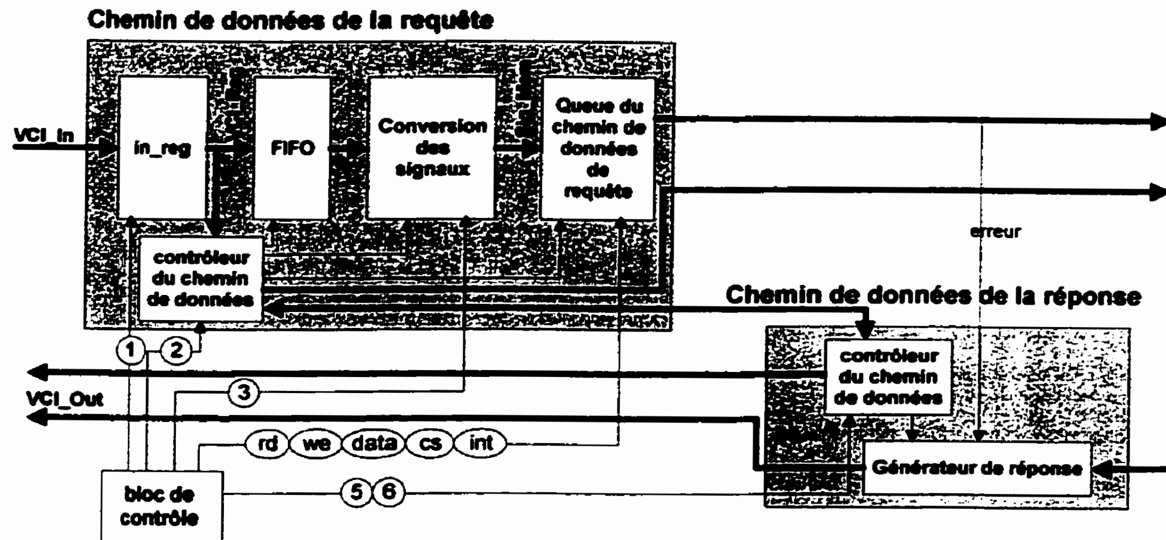


Figure 3.16. Diagramme bloc de l'interface esclave

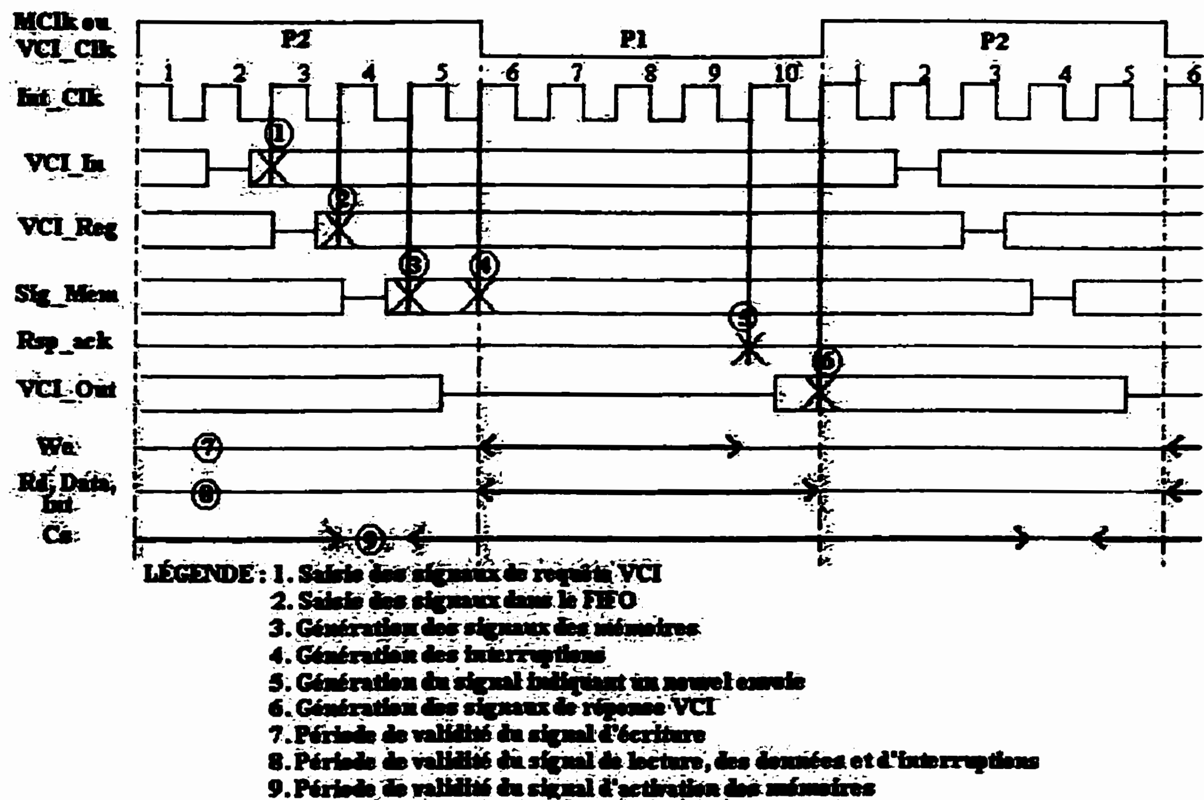


Figure 3.17. Diagramme temporel de l'interface esclave

Tableau 3.6. Paramètres VCI de l'interface esclave

Paramètres généraux		Paramètres [illegible]		Valeurs par défaut de BVCI	
Paramètre	Valeur	Paramètre	Valeur	Paramètre	Valeur
INITIATOR	0	CHAINING	0	DCFIXED	0
ADDRSIZE(N)	32	PLENSIZE(K)	0	DCONST	0
CELLSIZE(B)	4	CLENSIZE(Q)	0	DCONTIG	0
BIGENDIAN	0	DefCMDACK	1	DWRAP	0
NOENDIAN	0	DefRSPACK	1		
RESETLEN	8				
ERRLEN(E)	0				

3.4.3. Chemin de données de la requête

Il convient maintenant de donner des détails concernant la conception des chemins de données. Le chemin de la requête transforme les signaux VCI en signaux d'accès mémoire. En plus, il permet d'emmagasiner, dans une FIFO, les requêtes dans le cas où le flux des requêtes serait plus élevé que celui des réponses. Les spécifications VCI suggèrent de réduire au minimum l'espace tampon. Dans l'interface, la grandeur de la FIFO est fixée par l'intégrateur.

En bref, la requête VCI est saisie à tous les cycles, par un premier niveau de bascule (int_reg), 20 % après la montée de l'horloge VCI. Ensuite, les informations sont transmises à la FIFO et au contrôleur. C'est le contrôleur qui guidera les données à travers les étages de l'interface. Les données sorties de la FIFO sont converties en signaux mémoire et un bloc de logique combinatoire est utilisé pour la génération des signaux asynchrones. Ces trois derniers blocs sont décrits plus en détail.

3.4.3.1. FIFO

VSIA suggère de minimiser la quantité de registres dans l'interface. Cependant, pour accepter une nouvelle requête avant de recevoir l'accusé de réception de la réponse précédente, un espace tampon minimal est nécessaire. Cet espace peut être placé à l'entrée, à la sortie ou aux deux localisations. Dans le cas où le temps de traitement des requêtes serait plus lent que leur rythme d'entrée, il serait plus efficace de placer

la FIFO à l'entrée, même si la quantité de données à mémoriser est plus élevée. Ainsi, il y a une FIFO à l'entrée et un banc de registre à la sortie. Il est donc possible de recevoir des requêtes, de traiter une requête et d'attendre l'accusé de réception d'une réponse simultanément. De plus, avec une profondeur flexible pour la FIFO, il est possible d'obtenir le meilleur compromis entre l'espace utilisé et la performance.

La FIFO est composée de plusieurs cellules semblables dont seule la première cellule diffère (Figure 3.18). Les cellules sont construites de bascules et de multiplexeurs. Une cellule peut maintenir sa donnée (1), prendre la donnée de la cellule précédente (2) ou capter une nouvelle entrée (3). Ainsi, le contrôleur du chemin de données gère la circulation des données en sélectionnant les bonnes entrées des multiplexeurs. Si la FIFO est le moindrement profonde, il est préférable de remplacer la série de registres par une mémoire pour réduire l'espace des registres.

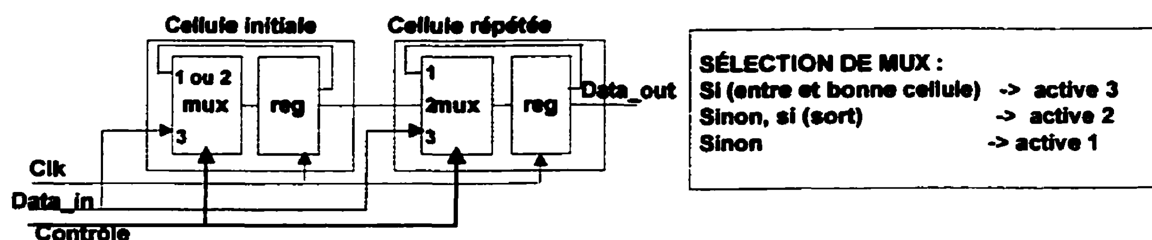


Figure 3.18. Structure de la FIFO de l'interface esclave

3.4.3.2. Contrôleur

Le contrôleur gère la circulation à travers le chemin de données et le bloc de contrôle gère la synchronisation des étapes en ce qui a trait aux délais. Certains signaux du bloc de contrôle sont donc utiles au contrôleur. Son rôle est de déterminer quand de nouvelles requêtes peuvent entrer dans la FIFO, où elles entrent et quand les requêtes peuvent aller vers la mémoire. Pour y arriver, trois signaux sont générés : *in* (entre), *out* (sort) et *no_cell* (pointant sur la bonne cellule de la FIFO). Lorsque la FIFO est vide, les requêtes qui entrent vont directement être envoyées vers les mémoires.

Ces trois signaux sont utilisés pour contrôler les multiplexeurs de la FIFO (Figure 3.18) et pour indiquer au module de conversion et au deuxième chemin de données qu'une nouvelle donnée peut être traitée. Ces signaux sont générés à partir des signaux de poignée de main des requêtes VCI et de l'état de la FIFO (pleine, vide), des interruptions et du deuxième chemin de données (envoi de réponse et accusé de réception).

En outre, le contrôleur s'occupe d'envoyer les interruptions déviées au processeur. Ainsi, lorsqu'une requête d'interruption avec déviation est envoyée, le contrôleur génère les signaux *in*, *out* et *no_cell* en conséquence et envoie une interruption au processeur en fonction des registres *Treated* et *Status*.

3.4.3.3. Conversion des signaux

Le bloc de conversion est très semblable à celui du commutateur, mais ce sont les signaux VCI qui sont transformés en signaux mémoires (Tableau 3.7). Le bloc est donc composé d'une partie combinatoire suivie de multiplexeurs et de registres. En plus de générer les signaux mémoires, le signal d'erreur et le signal de fin de paquet de la réponse VCI sont aussi générés et envoyés au deuxième chemin de données.

<u>Tableau 3.7.</u> Association des signaux du VCI aux signaux des mémoires

Signaux		[illegible]
[illegible]	[illegible]	
A[31 :2]	Cs Add[X :0]	Les bits les plus significatifs permettent d'identifier le composant auquel on a accès. Les bits restants sont utilisés pour adresser le composant.
A[31 :0]	Int	Les interruptions sont générées si la requête est une interruption sans déviation et si les registres *Treated* et *Status* le permettent.
Wdata[31 :0]	D[31 :0]	Les données du bus unidirectionnel de VCI sont envoyées sur le bus bidirectionnel des mémoires.
Be[3 :0]	Be[3 :0]	Le signal d'activation d'octets est directement transmis aux mémoires.
Cmd[1 :0]	We + Rd	La commandes VCI détermine le type d'accès (lecture ou écriture).

3.4.3.4. Queue du chemin de données

Ce module consiste en trois blocs de logique combinatoire utiles pour adresser des composants asynchrones. Son rôle est donc d'ajuster les délais et la période de validité des signaux. Le premier bloc s'occupe des signaux d'activation des composants et des signaux de lecture et d'écriture. Le second bloc génère les interruptions (internes et externes). Le dernier bloc contrôle le bus de données en fonction du type d'accès (lecture et écriture) et du composant adressé.

3.4.4. Chemin de données de la réponse

L'objectif du deuxième chemin de données est de générer les réponses VCI. Seulement deux blocs sont nécessaires, le contrôleur et le générateur de réponse.

3.4.4.1. Contrôleur

Le contrôleur détermine le moment d'envoyer les réponses VCI. Il génère donc le signal VCI *RspAck* lorsque les données de réponse sont prêtes. De plus, à partir des signaux de synchronisation du bloc de contrôle, il met à jour l'étage de bascules de sortie pour faire l'envoi de réponses au moment opportun. Enfin, il envoie un signal au contrôleur du premier chemin de données pour indiquer l'envoi d'une nouvelle réponse. Ainsi, pendant l'attente de l'accusé de réception de la réponse, une nouvelle requête peut être envoyée à la mémoire. Cependant, l'interface est restreinte à attendre l'accusé pour traiter une troisième requête, à cause de l'absence d'une FIFO à sa sortie.

Étant donné la vitesse de réaction des mémoires utilisées, le traitement d'une requête de lecture ou d'écriture s'effectuera toujours en un cycle. Ainsi, dès qu'une telle requête est envoyée aux mémoires, le contrôleur sait que la réponse sera disponible à la fin du cycle actuel. Cette partie du contrôleur devra être ajustée pour permettre l'utilisation de mémoires plus lentes.

Pour faire l'envoi d'une nouvelle réponse, le contrôleur doit vérifier trois éléments. D'abord, il vérifie que l'accusé de réception de la réponse précédente a été reçu. Si l'accusé n'a pas été reçu 10 % avant la montée d'horloge, aucune réponse ne sera envoyée avant le cycle suivant. Puis, il regarde si une nouvelle requête a été envoyée par le premier chemin de données. Enfin, il regarde si la requête était une interruption. Si c'est le cas et qu'il faut attendre le traitement, aucune réponse n'est envoyée tant que le registre *Treated* ne le permet pas.

3.4.4.2. Générateur de réponse

Le générateur de réponse est constitué d'un bloc de logique combinatoire, d'un multiplexeur et d'un banc de registres. Les signaux de fin de paquet et d'erreur venant du premier chemin de données, les données de lecture des mémoires et les signaux d'erreur des mémoires sont d'abord captés. Ensuite, le signal d'erreur (voir section 3.4.1.3), le signal de fin de paquet et les données VCI sont envoyés au moment opportun en fonction des sélecteurs du multiplexeur.

3.5. Interface maître

Puisque les principes de design ont été largement présentés dans les sections précédentes, l'accent est mis sur les particularités de ce module plutôt que sur les détails du design. D'abord, les requêtes disponibles sont décrites. Puis la structure générale, les particularités de la synchronisation et les choix effectués sont expliqués. Enfin, des figures illustrant le comportement du contrôleur sont données.

3.5.1. Utilisation de l'interface

L'interface maître est le miroir de l'interface esclave. D'ailleurs, pour une application multiprocesseur, l'interface maître d'un IP est directement connectée sur l'interface esclave de l'autre. L'objectif est donc de transformer des requêtes du processeur en requêtes VCI et de transmettre les réponses VCI en messages au processeur.

3.5.1.1. Description des requêtes

D'abord, le processeur peut faire plusieurs types de requêtes VCI en effectuant des lectures et écritures dans la banque de mémoire externe. En connaissant les espaces d'adressage externes, il peut générer tous les types de requêtes décrites au Tableau 3.5. Par exemple, si le processeur veut envoyer une interruption directe à un second processeur, il n'a qu'à effectuer une lecture ou une écriture à l'adresse correspondante. Ensuite, le bloc de logique combinatoire à l'entrée du premier chemin de données détectera cet accès et générera la requête VCI correspondante.

Un autre signal doit être considéré lors de la génération de requête VCI : le signal d'interruption généré par la boîte aux lettres de droite de la mémoire (Figure 3.5). Lorsque ce signal est actif, l'interface maître doit faire une requête VCI à l'adresse de cette boîte aux lettres pour signifier l'interruption à l'autre composant. Cependant, il est bien important de ne faire l'envoi qu'une seule fois pendant toute la période d'activité de l'interruption. En contrepartie, le maître doit informer l'esclave lorsque le processeur effectue une lecture à une boîte aux lettres externe pour que le signal d'interruption soit désactivé (section 3.4.1.1).

3.5.1.2. Signaux et paramètres VCI

Pour simplifier la première exploration du protocole, les signaux facultatifs de BVCI ne sont pas considérés. Ces signaux sont principalement utilisés pour effectuer des transmissions par paquets et par chaînes. Pour le moment, l'interface ne permet pas les transmissions par paquets et donc, le signal de fin de paquet sera toujours nul. Les paramètres VCI sont les mêmes que ceux de l'interface esclave (Tableau 3.6) mais l'interface est maître plutôt qu'esclave.

3.5.2. Structure générale

Tout comme dans l'interface esclave, le module contient deux chemins de données, mais ils sont inversés (Figure 3.19). Le premier transforme les requêtes de l 'ARM en des requêtes VCI et le deuxième transforme les réponses VCI en messages à l'ARM. Un seul contrôleur est nécessaire pour les deux chemins de données ainsi, son flux bidirectionnel. De plus, les signaux du bloc de contrôle sont envoyés au contrôleur qui, lui, gère les sélecteurs des multiplexeurs.

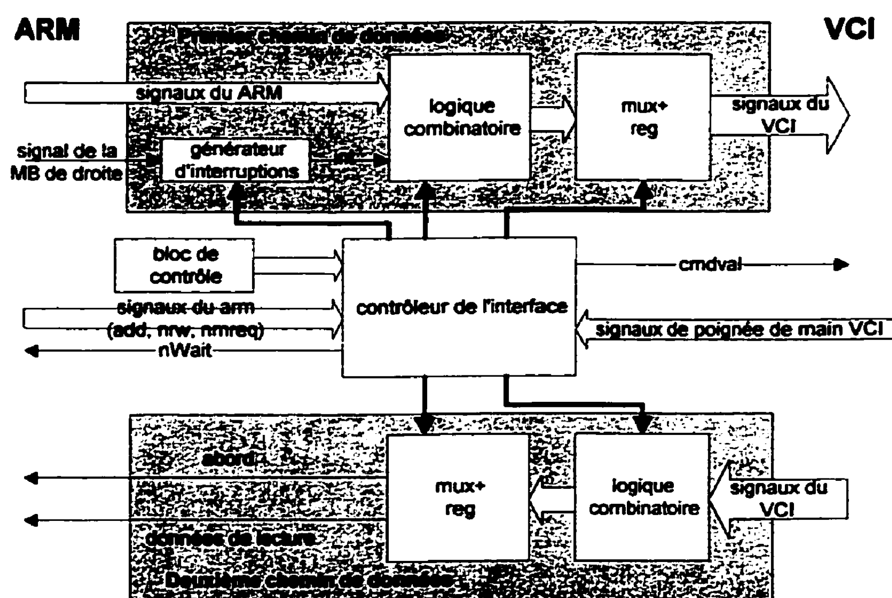


Figure 3.19. Schéma bloc de l'interface maître

La synchronisation de l'interface qui relie directement des signaux VCI à des signaux de l'ARM n'est pas aussi simple que celle qui relie les mémoires. Le contrôleur est donc assez compliqué et est décrit plus en détail dans les deux prochaines sections. Les deux chemins de données sont semblables au cas de l'interface esclave. Le Tableau 3.8 résume comment les signaux des deux protocoles sont associés.

Tableau 3.8. Association des signaux du processeur aux signaux VCI

[illegible]	[illegible]	[illegible]
A[31 :0]	A[31 :0]	Les bus d'adresse sont directement associés.
Mas[1 :0]+ A[1 :0]	Be	Le signal d'activation d'octets des mémoires est déduit des signaux de la taille de la données (Mas) et des octets adressés (A[1 :0]).
Dout[31 :0]	Rdata[31 :0]	Les bus de données dans les deux sens sont directement associés.
Din[31 :0]	Wdata[31 :0]	
Abord	Error	Le processeur est interrompu si une réponse VCI est erronée.
NRW+A[31 :0]	Cmd[1 :0]	La commande VCI est déduite du signal de RW et de l'adresse

3.5.3. Synchronisation

Synchroniser les accès mémoire de l'ARM avec les requêtes VCI n'est pas tâche facile et ces accès sont difficiles à exécuter en un cycle. Ce problème est d'abord présenté. Puis, les solutions seront explorées et la solution adoptée est donnée.

3.5.3.1. Problèmes de synchronisation

Le premier problème se situe au niveau de la lecture. En effet, il est impossible de faire une requête et de recevoir la réponse dans le même cycle. Ainsi, un problème se pose lorsque le processeur s'attend à recevoir la donnée de lecture dans un même cycle. Pour l'écriture, le problème se pose au niveau de la validité des données sur bus. Elles arrivent presque un demi-cycle après l'adresse, ce qui retarde l'envoi de la requête VCI. De plus, avant de faire passer le processeur à l'instruction suivante, il faut attendre l'accusé de réception de la requête, même si la réponse n'est pas nécessaire.

En outre, pour ralentir le processeur, il faut envoyer un signal d'attente (nWait) et ce signal doit être envoyé dans la partie P1 du cycle (Figure 3.20). Il est donc essentiel de savoir rapidement si le processeur doit être mis en attente et donc la capture des signaux de poignées de main VCI se fait tôt dans le cycle.

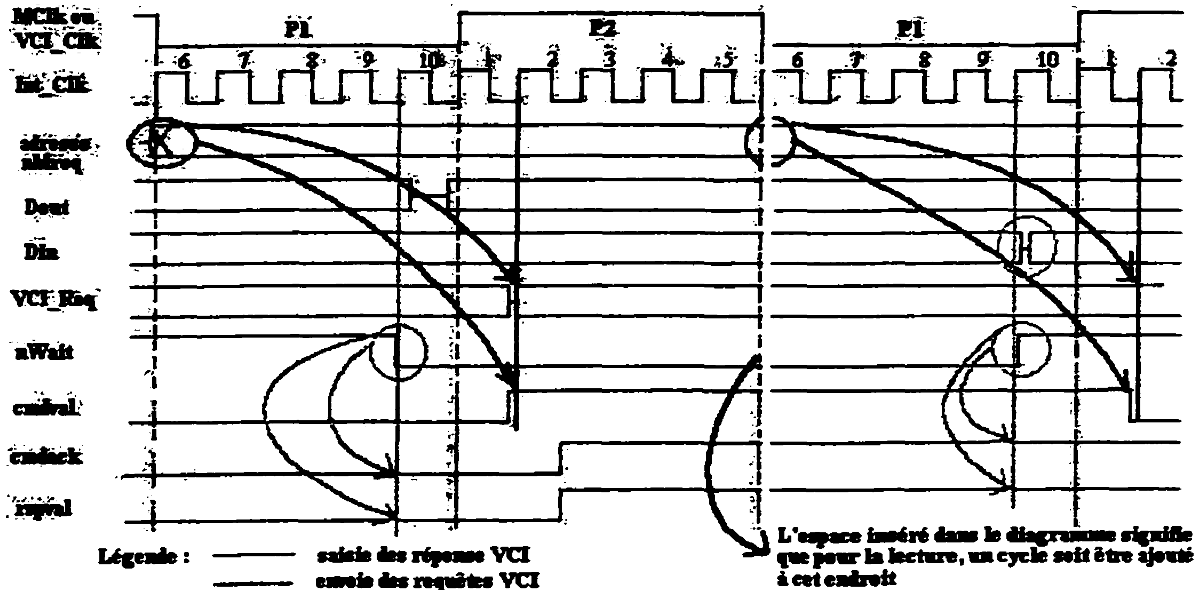


Figure 3.20. Diagramme temporel de l'interface maître

3.5.3.2. Exploration des solutions

D'abord, le déphasage entre les horloges a été modifié, mais les solutions obtenues n'étaient pas valides. Ensuite, la fréquence VCI a été augmentée par rapport à la fréquence du processeur. Cependant, cette augmentation de fréquence cause des problèmes ailleurs. Par exemple, un accès mémoire interne ne se produit plus dans un cycle. De plus, les délais entre les niveaux de bascules sont trop courts.

La réponse VCI d'une écriture ne contient pas de données, mais seulement un signal d'erreur. Il est possible d'économiser un cycle en terminant l'écriture du processeur (avec nWait) avant de recevoir la réponse VCI. Néanmoins, il est essentiel de pouvoir identifier toutes les réponses qui entrent pour éviter le mélange des réponses des lectures et des écritures. Ainsi, on peut libérer le processeur une fois l'écriture effectuée, mais il faut attendre la réponse VCI avant de permettre un autre accès externe (VCI). Le processeur peut immédiatement effectuer des accès mémoires internes (programme) et un cycle est économisé à chaque écriture. Le seul problème est que les erreurs VCI pour les écritures ne sont pas communiquées au processeur.

Pour la lecture, il est bien difficile de régler le problème à cause du protocole VCI. Cependant, une façon d'économiser un cycle est de faire l'envoi VCI sans attendre les données de l'ARM (seulement nécessaire lors de l'écriture). Cela complique l'interface et la synchronisation.

Une solution très intéressante aux problèmes mentionnés plus haut est l'utilisation de paquets où les adresses suivent une séquence. Ainsi, en utilisant les bits VCI facultatifs, l'adresse suivante peut être prédite et le temps d'une opération de lecture est réduit. Pour ce faire, il faut que le processeur puisse faire des accès successifs. Les seules instructions de l'ARM7 qui permettent plusieurs transferts mémoire sont les instructions de copie de registres et leur utilisation est limitée. En contrepartie, l'utilisation d'un DMA (Direct Memory Access) est idéale pour le transfert de grands blocs de données.

Pour implanter la transmission par paquets, il faut d'abord intégrer les signaux VCI facultatifs. En plus, il faut utiliser les signaux SEQ et BL de l'ARM permettant de savoir quand les accès sont séquentiels. Dans le cas d'utilisation de mémoire de 16 ou 8 bits ou d'utilisation de bus VCI de 16 bits, cette implantation est essentielle pour permettre au processeur de faire des accès de 32 bits.

3.5.3.3. Solution adoptée

Pour diminuer la complexité du contrôleur et assurer la validité des envois, les requêtes VCI sont toujours envoyées au même moment, peu importe le type. L'écriture est considérée comme terminée aussitôt l'accusé de réception reçu, mais avant de pouvoir faire un nouvel envoi, la réponse de cette requête doit être saisie.

3.5.4. Description du contrôleur

Le contrôleur est légèrement complexe. Le schéma bloc du contrôleur montre sa structure de base (Figure 3.21). D'abord, un module sert à générer une requête

d'interruption externe lorsque le signal de la boîte aux lettres droite est actif. Ensuite, une machine à états génère des signaux de contrôle. Certains de ses signaux doivent aussi être combinés et registrés.

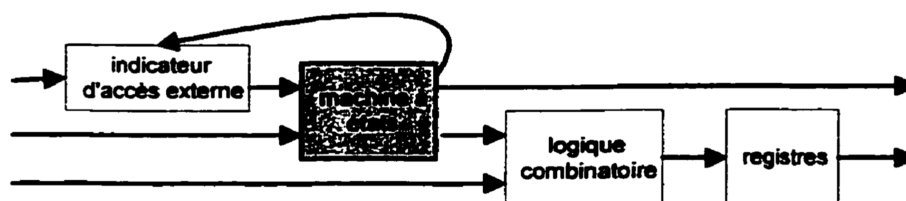


Figure 3.21. Schéma bloc simplifié du contrôleur de l'interface maître

Pour simplifier la description du contrôleur, sa machine à états est séparées en deux : la lecture (Figure 3.22) et l'écriture (Figure 3.23). La partie de cette machine qui gère les interruptions n'est pas représentée. À chaque accès externe, le processeur est mis en attente pendant au moins un cycle (one_wait), à cause du délai minimal d'envoi d'une requête. Ensuite, selon le type d'instruction, le processeur est gardé en attente ou libéré durant l'attente de la réponse.

Le Tableau 3.9 résume les délais résultants pour les lectures et écritures. Les cycles considérés dans le calcul du délai sont représentés sur les machines à états (en gris sur la Figure 3.22 et la Figure 3.23). En somme, la génération d'une demande VCI dépend de la présence d'un accès externe et le temps d'attente du processeur est lié à l'accusé de réception de la requête et à la réception de la réponse (Figure 3.20)

Tableau 3.9. Temps des opérations de lecture et écriture

Instruction	[illegible]	[illegible]	[illegible]
Lecture	2 cycles	3 cycles	5 cycles
Écriture	1 cycle	2 cycles	3 cycles

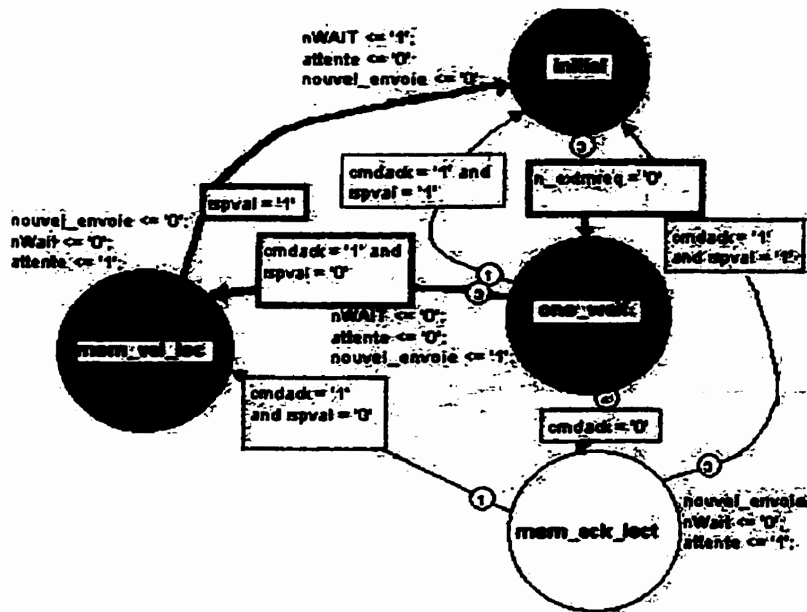


Figure 3.22. Machine à états du contrôleur : la lecture

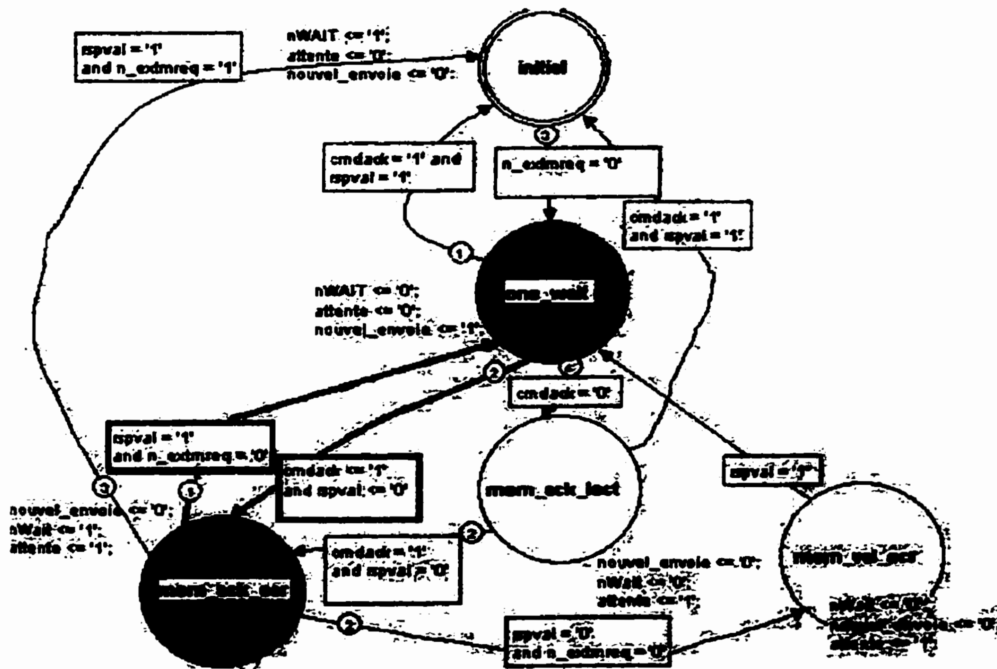


Figure 3.23. Machine à états du contrôleur : l'écriture

Chapitre 4 : RÉSULTATS

L'interface est conçue en vue d'être utilisée dans un contexte de communication point-à-point. Différentes applications sont implémentées à partir de l'interface et du processeur et ce chapitre relate et analyse les résultats obtenus. D'abord les résultats généraux sont présentés. Puis les résultats des trois applications implémentées sont expliqués. Enfin, une brève discussion termine le chapitre. Les résultats présentés ici sont ceux qui concernent le bon fonctionnement de l'interface ainsi que sa synthèse. D'abord, la méthode de vérification utilisée, puis les résultats de synthèse sont expliqués. Enfin la méthode d'analyse des performances est présentée.

4.1. Résultats généraux

4.1.1. Covérification de l'interface

D'abord, il importe de préciser comment le banc de test est construit pour permettre de vérifier toutes les configurations de l'interface. Le système est simulé à l'aide de l'outil Seamless, qui permet de simuler des fichiers VHDL (avec Modelsim) et d'exécuter du code assembleur sur un processeur en parallèle. Ainsi, le banc de test est composé de fichiers VHDL et de fichiers C (ou assembleur) compilés pour ARM. La covérification se fait seulement avant la synthèse. Elle devrait aussi se faire après la synthèse, mais nous ne l'avons pas effectuée à cause des problèmes de compatibilité entre l'outil de cosimulation (Mentor Graphics) et l'outil de synthèse (Synopsys). Néanmoins, la vérification individuelle après synthèse de chaque bloc a été effectuée.

Le banc de test VHDL est divisé en deux, une interface maître, qui envoie des requêtes et reçoit des réponses, et une interface esclave, qui fait l'inverse. Les requêtes et réponses du fichier C (donc du processeur) doivent correspondre avec les interfaces VHDL du banc de test. L'objectif est d'éviter au concepteur de regarder la trace de simulation pour vérifier le bon fonctionnement du système. Ainsi, la

vérification de la validité des signaux est automatique et des messages s'affichent à l'écran pour indiquer les erreurs au concepteur.

Toutes les informations concernant les requêtes envoyées et les réponses attendues ou les réponses associées à chacune des requêtes sont contenues dans des tableaux. Un tableau est utilisé pour l'interface maître, l'autre pour l'interface esclave et un troisième pour le fichier C. La grandeur des tableaux est variable et les données dans les tableaux le sont aussi. Ainsi, il est possible de tester toutes les possibilités de communication en ajustant le contenu des tableaux. Le code VHDL et la structure du tableau du module de génération de requête sont donnés à la section J.3.

Pour le moment, l'intégrateur doit coordonner les valeurs des trois tableaux avec les paramètres de l'interface à tester. Cependant, il serait possible de créer un programme qui génère automatiquement les valeurs des tableaux en fonction des paramètres de l'interface à vérifier. En outre, un processus VHDL pourrait être ajouté au banc de test pour permettre de modifier de façon aléatoire les valeurs des tableaux et ainsi tester vraiment toutes les possibilités. De cette façon, un second script pourrait créer un banc de test correspondant à l'interface générée par le premier script.

4.1.2. Synthèse de l'interface

Puisque l'objectif est d'utiliser l'interface développée dans un système sur une puce, il est essentiel qu'elle soit synthétisable. L'outil Design Compiler de Synopsys est utilisé pour faire la synthèse. La synthèse permet de vérifier le comportement d'un système synthétisé, d'évaluer l'espace occupé ainsi que la fréquence maximale de fonctionnement. Malheureusement aucune évaluation de la dissipation de puissance n'est effectuée.

Puisque les résultats de synthèse dépendent des paramètres de l'interface, une configuration de base a été sélectionnée et elle sera utilisée pour les autres tests

effectués sur l'interface. Le "package" de la configuration est donnée à l'annexe J.1. Même si la cosimulation complète vise seulement les fichiers pré-synthèse, la simulation individuelle post-synthèse des modules principaux a été effectuée.

4.1.2.1. Espace occupé

L'espace occupé est exprimé en fonction des unités de base de Synopsys pour une technologie. Le développement de SoC n'a de sens que si une technologie récente est utilisée. Les performances de l'interface doivent donc être évaluées pour une telle technologie, par exemple la 0,18 micron. Cependant, à titre de comparaison, les tests sont aussi effectués sur une technologie de 0,35 micron. De cette façon la progression de la technologie et les performances de l'interface peuvent être reliées.

Le Tableau 4.1 résume bien tous les résultats obtenus. D'abord, il donne l'espace occupé, pour chaque bloc de base ainsi que pour l'interface globale. De plus, il donne le pourcentage de la logique qui est séquentielle et le pourcentage d'espace occupé par chacun des blocs. La différence entre les résultats obtenus pour chacune des technologies est aussi présentée.

Il est difficile de juger si l'espace obtenu est raisonnable car cela dépend beaucoup de la proportion de l'interface par rapport à l'IP lui-même. Étant donné que l'espace occupé par le processeur n'est pas disponible, cette comparaison ne peut être établie. Néanmoins, la validité du compromis entre la facilité d'intégration et la perte d'espace dépend de la grosseur du IP utilisé. Plus le IP est gros, moins la proportion relative de l'interface est grande et meilleur est le compromis.

Le pourcentage de logique séquentielle est très près de 50 %. Ceci peut s'expliquer par le fait que, dans l'architecture proposée, la logique combinatoire (essentiellement de conversion) est assez simple mais qu'à chaque banc de registres est associé une série de multiplexeurs. Ainsi, l'augmentation de la logique combinatoire suit celle de

la logique séquentielle et la réduction de la quantité de registres double la diminution de l'espace occupé.

Tableau 4.1. Espace occupé (unités de base de Synopsys) par chaque module

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
0,35 microns	nombre d'unités	17 728	89 758	349 808	69 143	526 437
	% de logique séquentielle	64,66 %	56,77 %	42,03 %	55,35 %	47,06 %
	% de l'espace total	3,37 %	17,05 %	66,45 %	13,13 %	100,00 %
0,18 microns	nombre d'unités	4 147	17 852	49 938	14 444	86 381
	% de logique séquentielle	68,44 %	65,14 %	63,66 %	58,57 %	63,34 %
	% de l'espace total	4,80 %	20,67 %	57,81 %	16,72 %	100,00 %
Différence entre les technologies (en %)	nombre d'unités	76,61 %	80,11 %	85,72 %	79,11 %	83,59 %
	% de logique séquentielle	5,52 %	12,84 %	33,97 %	5,49 %	25,71 %
	% de l'espace total	29,85 %	17,50 %	-14,94 %	21,45 %	0,00 %

La dominance de l'interface esclave, en terme d'espace, est flagrante. Le rôle minoritaire des registres d'interruptions concorde avec l'espace occupé. Cependant, pourquoi l'interface esclave est-elle beaucoup plus grande que l'interface maître ou que le commutateur ? La profondeur de son pipeline, donc son nombre d'étages de registres, en est probablement la cause. L'utilité de tous ces registres est à discuter, mais, il est évident que le meilleur moyen de diminuer la grosseur de l'interface est de réduire le nombre d'étages du pipeline de l'interface esclave.

Le passage d'une technologie de 0,35 à 0,18 microns amène une diminution d'espace d'environ 80 %. La proportion d'espace occupé par l'interface diminue dramatiquement d'une technologie à l'autre. De plus, l'espace de logique séquentielle augmente avec une technologie de 0,18 microns. Cette différence est plus marquée pour l'interface esclave, même si sa proportion totale diminue. Ceci s'explique par le fait que le multiplexeur est plus optimisé que les autres cellules. Ainsi, le nombre de bascules (et multiplexeurs) de l'interface esclave lui confère les propriétés données plus haut.

4.1.2.2. Fréquence maximale

L'ensemble des simulations et des tests effectués considérait une période d'horloge VCI de 40ns (fréquence de 25MHz), et une période de pipeline de 4ns (fréquence de 250MHz). Cette période est minimale étant donné les délais des mémoires, les délais de l'ARM et les contraintes de son modèle sur Seamless. Cependant, cette fréquence est-elle acceptable pour ce qui est des délais de la logique combinatoire entre les niveaux de bascules? En effet, une fréquence de 250MHz est assez élevée, et pour une technologie de 0,35 micron (technologie utilisée pour implanter un processeur qui roule à 25MHz) , cela peut causer des problèmes.

Pour faire cette vérification, l'analyseur de délais de l'outil de synthèse est utilisé. Encore une fois, deux technologies sont comparées. L'observation des résultats obtenus (Tableau 4.2), démontre que l'hypothèse selon laquelle le délai entre les niveaux de bascules serait suffisamment petit avec une fréquence VCI de 25MHz n'est pas confirmée. La technologie de 0,35 microns est nettement trop lente et celle de 0,18, un peu trop lente. Cependant, la différence entre les technologies pousse à croire que l'hypothèse serait confirmée avec une technologie de 0,15 ou 0,13 microns.

Tableau 4.2. Période minimale de l'horloge VCI pour chaque module

[illegible]	[illegible]	[illegible]	[illegible]
Registres d'interruptions	48 ns	22 ns	54,17 %
Commutateur	50 ns	31 ns	38,00 %
Interface esclave	50 ns	29 ns	42,00 %
Interface maître	78 ns	50 ns	35,90 %
Période minimale	78 ns	50 ns	35,90 %

Néanmoins, utiliser une technologie de 0,13 micron pour implanter la partie matérielle de l'interface impliquerait d'utiliser la même technologie pour le processeur. Le processeur pourrait alors fonctionner à une fréquence beaucoup plus élevée que 25MHz. Ainsi, en gardant la même structure pour le pipeline ainsi que le même rapport de fréquences, la fréquence de fonctionnement du pipeline serait,

encore une fois, trop élevée pour la technologie. Ainsi, pour obtenir un système réalisable physiquement, il faudrait diminuer le rapport des fréquences (par exemple utiliser 5 au lieu de 10) ou modifier la structure du pipeline.

Notons que le goulot d'étranglement est l'interface esclave. Puisque le nombre d'étages de son pipeline est élevé, la proportion du cycle de l'horloge VCI disponible pour chaque bloc de logique combinatoire est petite et sa fréquence maximale est petite. Pour pouvoir augmenter la fréquence maximale (pour une technologie donnée), il faudrait remanier la structure de l'interface pour diminuer le nombre d'étages du pipeline. Ainsi une plus grande portion d'un cycle VCI serait accordée à chaque bloc de logique combinatoire.

4.1.3. Contexte d'évaluation des performances

Il faut maintenant évaluer les performances de l'interface développée et du protocole utilisé dans des applications diverses. Pour y arriver, différentes situations sont comparées. Les systèmes sont construits avec et sans interface VCI. Dans les systèmes sans VCI, le commutateur communique directement avec les mémoires ou l'autre processeur sans passer par les interfaces VCI maître ou esclave.

Un contexte de communication point-à-point avec seulement deux IP est employé. Les systèmes sont implantés avec deux processeurs ARM ou avec un processeur et un coprocesseur matériel (Figure 4.1). Le médium de communication est toujours une DPRAM et les moyens de synchronisation sont les sémaphores, les interruptions ou les boîtes aux lettres. Deux des architectures proposées dans [Erns97] sont utilisées : le modèle du coprocesseur dépendant et du coprocesseur indépendant. Trois algorithmes ayant des besoins différents sont implantés : un algorithme de filtrage, un encodeur/décodeur Reed Solomon et un algorithme de tri par segmentation.

Les paramètres de l'interface utilisés sont ceux du "package" de l'annexe J et les mémoires utilisées sont celles proposées à l'annexe I. Les simulations sont effectuées avec Seamless sur l'interface avant synthèse. Les temps sont donnés en cycles VCI (ou de l'ARM) et la période utilisée est de 40ns. La nécessité de coordonner les simulateurs logiciels et matériels diminue la précision de Seamless. Il faut donc interpréter les résultats à quelques cycles près.

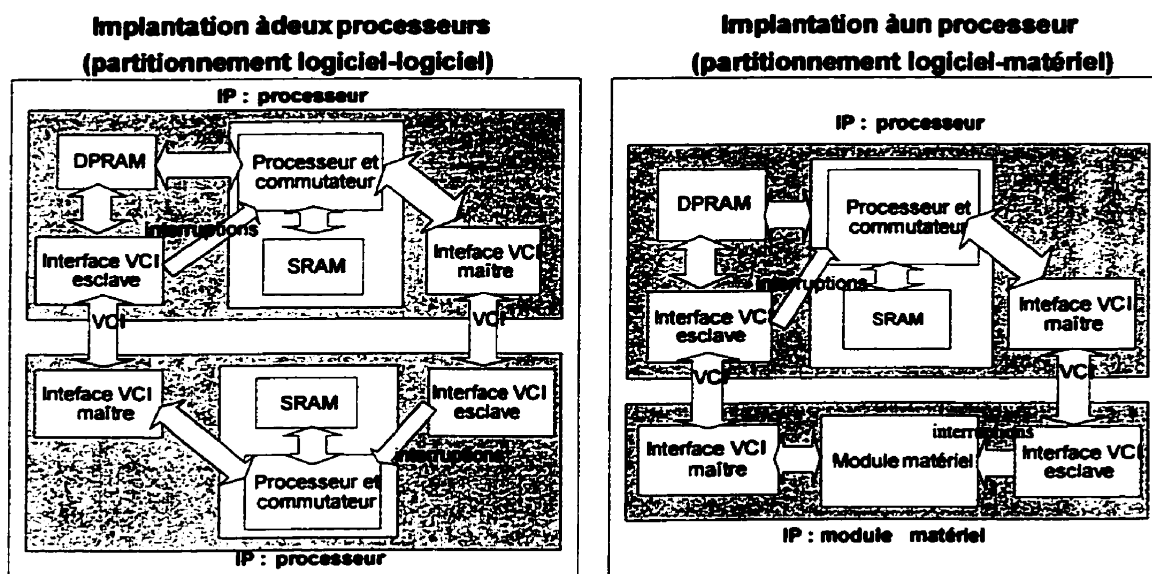


Figure 4.1. Schéma bloc des deux types d'implantation de système

4.2. Algorithme de filtrage

Dans plusieurs applications, un coprocesseur (ou accélérateur) est souvent associé à un processeur (ou initiateur) pour augmenter la vitesse de traitement. L'algorithme de filtrage utilise un modèle de coprocesseur dépendant. Logiquement, le coprocesseur (accélérateur) est implanté en matériel pour travailler en parallèle avec le processeur. Cependant, il est aussi implanté sur un second processeur, pour permettre de mieux évaluer les performances de l'interface maître.

4.2.1. Description de l'implantation de l'algorithme de filtrage

L'algorithme de filtrage, utilisé pour le traitement d'images, consiste à identifier la position d'un patron dans une image. Dans cet algorithme, le processeur effectue essentiellement des écritures et l'accélérateur des lectures. Le processeur inscrit une image de 16 pixels et un patron de 8 pixels dans la DPRAM et chaque pixel est codé sur 32 bits. Ensuite, il indique au coprocesseur que les données sont prêtes en utilisant un sémaphore, une interruption ou une boîte aux lettres.

Puis, le coprocesseur lit l'image et le patron et identifie la position du patron dans l'image. Le coprocesseur inscrit la position du patron dans la DPRAM et indique au processeur que le résultat est prêt. Pour identifier la position du patron, un bloc de la grandeur du patron se déplace dans l'image et ses pixels sont comparés à ceux du patron. Si une association est trouvée, la position est identifiée comme la coordonnée, dans l'image (matrice 16 par 16) du premier pixel de ce bloc. Ainsi, le temps de recherche du patron dépend de la position de celui-ci dans l'image. La Figure 4.2 illustre les 256 pixels de l'image et les positions possibles du patron.

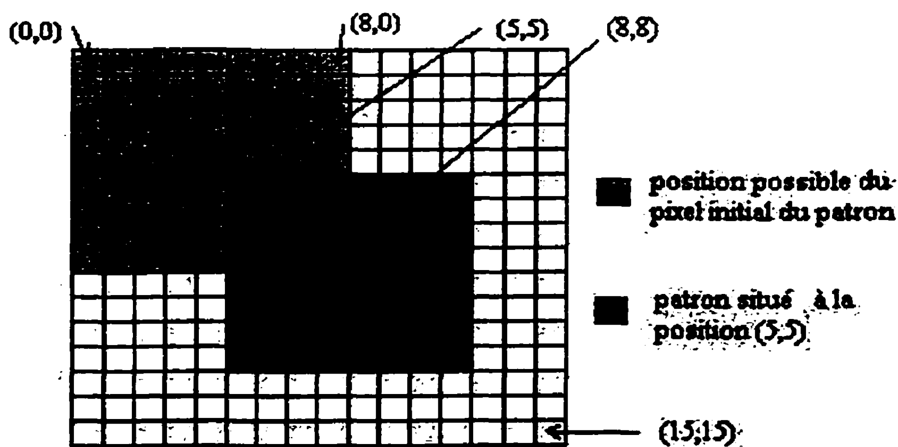


Figure 4.2. Représentation des 256 pixels d'une image

Le coprocesseur a d'abord été conçu en matériel (partitionnement logiciel/matériel) avec trois types de synchronisation différents. Même si le code n'est pas totalement

synthétisable, le fonctionnement réel est bien reproduit et les résultats des simulations sont représentatifs. Ensuite, le coprocesseur est codé sur un second ARM (partitionnement logiciel/logiciel).

Dans les deux cas, le processeur principal peut effectuer d'autres tâches pendant les comparaisons. Cependant, le coprocesseur matériel est beaucoup plus efficace. En effet, il peut effectuer les comparaisons des pixels du patron et d'un bloc de l'image en parallèle. De plus, il contient des registres matériels dédiés à mémoriser des pixels et jouant ainsi le rôle d'une mémoire cache. Ainsi, chaque pixel, mémorisé dans la DPRAM, n'est accédé qu'une seule fois, contrairement au coprocesseur logiciel.

Dans le banc de test, le processeur effectue 384 opérations d'écriture. En effet, il effectue 256 écritures pour mettre des 0 dans toute l'image. Ensuite, il effectue deux fois 64 écritures pour inscrire le patron (dans l'image et le patron). Les comparaisons sont faites jusqu'à ce que la position (8,8) soit atteinte (Figure 4.2). Si le patron est à l'extérieur de l'image, la position (8,9) est retournée. Les résultats des essais sont donnés aux paragraphes qui suivent. Il faut noter que pour une implantation réelle de l'application, une opération de mise à zéro (dans le cas matériel) aurait dû être utilisée au lieu d'effectuer les 256 premières écritures.

4.2.2. Partitionnement logiciel/matériel

La première architecture testée constitue un partitionnement logiciel/matériel basé sur l'architecture présentée dans la partie de droite de la Figure 4.1 et le modèle des communications de la Figure 4.3.A. Trois mécanismes de synchronisation sont utilisés. En premier lieu, le processeur bloque un sémaphore pendant l'écriture des données et le coprocesseur doit attendre qu'il soit débloqué avant de faire son traitement. Le même mécanisme est utilisé du côté du coprocesseur. Le problème est que plusieurs lectures aux sémaphores sont faites pour connaître l'état du système.

Dans le deuxième essai, le sémaphore du coprocesseur est remplacé par une interruption. Ceci évite au processeur d'effectuer plusieurs lectures au sémaphore. Pour le troisième essai, des boîtes aux lettres sont utilisées. Ainsi, même si les MB sont plus lentes que des interruptions directes, aucun sémaphore ne doit être lu et le temps d'exécution est optimisé.

Le coprocesseur matériel accélère la recherche grâce à son pipeline de lecture et au parallélisme de ces comparaisons. Au départ, tout le patron est transféré dans des registres matériels. En plus, les 120 premiers pixels de l'image sont mis dans un registre à décalage. Ensuite, à chaque nouvelle position testée, une nouvelle lecture est effectuée et le registre des pixels de l'image est décalé. En somme, pour un patron à la position 0, 184 (64 + 120) lectures sont effectuées et ensuite, à chaque nouvelle comparaison, une lecture additionnelle est effectuée. Avec cette méthode, le patron est cherché dans tous les pixels de l'image, malgré les positions impossibles.

Le Tableau 4.3 résume le nombre de cycles nécessaires pour chaque étape exécutée. D'abord, comme attendu, le temps de communication est le plus élevé pour le sémaphore et le moins élevé pour l'interruption directe. Puis, l'écriture d'un pixel prend à peu près 25 cycles à cause du temps de lecture des instructions et du temps d'exécution des boucles. Puisque les écritures sont effectuées par l'initiateur, elles ne sont pas transmises via VCI (Figure 4.3.A) et leur délai n'est pas affecté par l'implantation. D'un autre côté, le temps de recherche de l'accélérateur est proportionnel au nombre de lectures à effectuer, c'est-à-dire un cycle par lecture.

Toutes les lectures mémoire sont séquentielles, puisque les données sont transférées par bloc. De plus, la lecture d'un pixel est indépendante de celle du pixel précédent. Ainsi, il est possible de concevoir le module effectuant les accès mémoire de façon à considérer qu'un accès mémoire prenne deux cycles (requête et réponse). La lecture du pixel nécessaire à la prochaine comparaison commence donc un cycle plus tôt pour éviter de prendre du temps à cause du protocole VCI. C'est pour cela que les

différences entre les systèmes avec et sans VCI sont négligeables, considérant la précision des résultats fournis par Seamless.

Tableau 4.3. Délais du filtrage pour une implantation matérielle

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	
				[illegible]	[illegible]
Écriture des données		9 316	384 écritures		
Communication P->C - avec sémaphores - avec interruptions - avec MB	 15 16 15	 15 14 13		 0 2 2	 0,0 % 14,29 % 15,38 %
Recherche : patron à (0,0)	189	188	184 lectures	1	0,53 %
Recherche: patron à (5,5)	273	273	269 lectures	0	0,00 %
Recherche: pas de patron	326	325	321 lectures	1	0,31 %
Communication C->P - avec sémaphores - avec interruptions - avec MB	 27 69 72	 28 68 74		 -1 1 -2	 -3,57 % 1,47 % -2,70 %

4.2.3. Partitionnement logiciel/logiciel

Pour tester complètement l'interface maître du processeur, il faut utiliser une architecture à deux processeurs (partitionnement logiciel/logiciel à la Figure 4.1). En effet, avec une implantation matérielle, l'interface maître du processeur est seulement utilisée pour envoyer des interruptions au coprocesseur. Néanmoins, si deux processeurs sont employés, un des processeurs doit accéder la DPRAM en passant par l'interface VCI. En effet, la mémoire DPRAM peut être à l'intérieur de l'enveloppe d'un ou l'autre des processeurs (initiateur ou accélérateur). Ainsi, deux modèles de communication sont possibles (Tableau 4.3). En utilisant le modèle A de la Figure 4.3, la performance des lectures à travers VCI est évaluée. En utilisant le modèle B, ce sont les écritures qui sont testées.

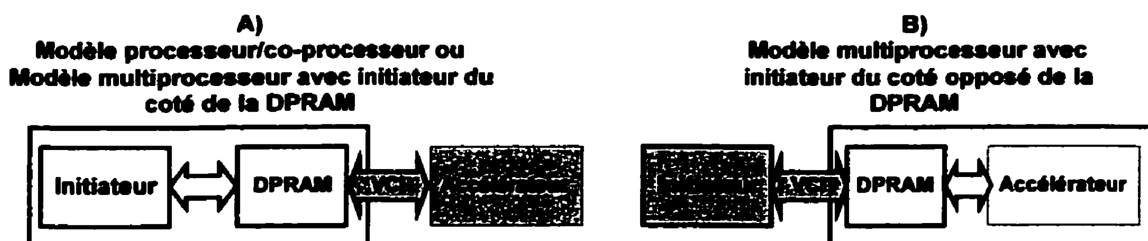


Figure 4.3. Modèles de communication avec l'interface VCI

En effectuant des écritures et des lectures simples d'un processeur à travers l'interface VCI, les résultats du Tableau 4.4 ont été observés sur la trace de simulation. Comme expliqué à la section 3.5.3, les contraintes du processeur et du protocole VCI ont mené à un circuit qui utilise, au minimum, deux cycles pour une écriture et trois cycles pour une lecture. C'est pour cela que le temps d'opération d'une lecture est augmenté de 67 % et celui de l'écriture de 50 %. Il faut maintenant observer l'effet sur une application réelle.

Tableau 4.4. Délais d'exécution des opérations mémoire du processeur

Instruction	[illegible]	[illegible]	Différence entre avec et sans VCI
Lecture	5	3	2
Écriture	3	2	1

Le Tableau 4.5 donne la comparaison entre les résultats obtenus pour l'implantation de l'algorithme de filtrage sur deux processeurs avec et sans VCI. D'abord, pourquoi le nombre de lectures nécessaires à la recherche est-il si élevé ? Contrairement à l'implantation matérielle, il n'y a aucune mémoire cache et les comparaisons sont effectuées au rythme d'un pixel par cycle. Ainsi, pour chaque position à vérifier, il faut faire 64 comparaisons et 128 lectures. Néanmoins, l'algorithme permet d'éviter de faire la recherche de patron aux positions impossibles. Il faut aussi noter qu'étant donné l'efficacité des boîtes aux lettres, seul ce mécanisme est implanté.

Tableau 4.5. Délais du filtrage pour une implantation logicielle

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Écriture des données	384	9316	0	0,00 %	9700	384	4,12%
Communication P->C	1 => MB	64	8	14,29 %	58	2	3,57%
Recherche : patron à (0,0)	128	2901	293	11,23 %	2605	-3	-0,12%
Recherche : patron à (5,5)	5760	140323	13021	10,23 %	127265	-10	-0,01%
Recherche : pas de patron	9216	221190	20737	10,35 %	200378	-16	-0,01%
Communication C->P	1 => MB	86	-1	-1,15 %	92	5	5,75%

Comme expliqué plus haut, pour connaître les différences de temps d'accès entre une implantation avec et sans VCI, les résultats à regarder sont les écritures dans le cas B et le temps de recherche (les lectures) dans le A. D'ailleurs, ces résultats sont les seuls où la différence n'est pas négligeable.

En somme, il faut compter un cycle de plus par écriture et 2,25 par lecture. Le délai de lecture est légèrement plus élevé que ce qu'indique le Tableau 4.4, mais tout de même très semblable. En poussant un peu l'observation, il est possible de noter une augmentation du temps de communication lorsque l'interruption d'une boîte aux lettres passe à travers l'interface VCI. Cette augmentation est tout de même assez négligeable.

4.2.4. Analyse des résultats de l'algorithme de filtrage

Le grand désavantage de VCI est la perte d'un cycle (requête et réponse) au transfert de chaque paquet. Il faut maintenant vérifier si les résultats sont affectés dans une implantation réelle. Les différentes architectures utilisées pour réaliser l'algorithme de filtrage ont permis d'évaluer plusieurs aspects.

D'abord, l'utilisation du protocole VCI par un coprocesseur matériel (partitionnement L/M) pour effectuer des lectures de bloc de données a été évaluée. Dans ce cas, c'est

l'interface esclave du processeur qui est mise à l'épreuve. Les conclusions sont que si la lecture d'une donnée est indépendante de celle de la donnée précédente et que l'émetteur peut coordonner l'envoi des requêtes de façon à recevoir les réponses au bon moment, l'utilisation de VCI n'affecte pas les résultats.

Puis, l'interface maître du processeur est testée avec l'implantation logicielle (partitionnement L/L). Le déplacement de la DPRAM d'une enveloppe à l'autre permet de vérifier l'efficacité des lectures et des écritures du processeur à travers une interface VCI. Les résultats montrent une augmentation du délai de 67 % pour une lecture et de 50 % pour une écriture.

Il est maintenant possible d'évaluer l'efficacité globale de toutes les architectures proposées pour implanter l'algorithme. Le Tableau 4.6 résume le temps total nécessaire pour effectuer trois recherches de patron (l'écriture des données est répétée à chaque recherche). D'abord, l'implantation matérielle est très peu affectée par l'utilisation de VCI et permet une accélération de 90 % par rapport à une implantation à un seul processeur. En plus, malgré la très petite différence entre les trois, le mécanisme de synchronisation le plus efficace est bel et bien les boîtes aux lettres et le moins efficace, les sémaphores.

En ce qui concerne l'implantation logicielle, l'augmentation du temps d'exécution attribuable à l'utilisation de VCI est beaucoup plus marquée lorsque l'initiateur est placé du côté de la DPRAM. Ceci s'explique par le fait que le nombre d'écritures est beaucoup moins élevé que le nombre de lectures. Par conséquent, dans la configuration où les lectures doivent passer à travers l'interface VCI, les délais engendrés sont beaucoup plus importants. En somme, l'effet global de l'utilisation de VCI sur les performances d'une application dépend de la proportion de celle-ci constituée d'accès VCI.

Tableau 4.6. Performances des implémentations de l'algorithme de filtrage

Partitionnement (implantation)	[illegible]	[illegible]	[illegible]	[illegible]	
				[illegible]	[illegible]
Partitionnement L/M					
- avec interruptions	29 722	29 704	0,06 %	91,71 %	91,72 %
- avec sémaphores	33 802	33 798	0,01 %	90,58 %	90,58 %
- avec MB	29 167	29 158	0,03 %	91,87 %	91,87 %
Partitionnement L/L					
- côté de la DPRAM	380 992	347 189	9,74 %	-6,23 %	3,20 %
- côté opposé à la DPRAM	347 866		0,19 %	3,01 %	
Aucun partitionnement	358 656	358 656			

En plus, l'utilisation de deux processeurs ne permet pas de diminuer le temps d'exécution de l'algorithme. En effet, l'utilisation d'un deuxième processeur ne permet pas d'éviter le temps de lecture des instructions, ni d'éviter la lecture multiple d'un même pixel. En outre, le délai de communication entre les processeurs rend l'architecture où l'initiateur est du côté de la mémoire moins efficace que celle avec un seul processeur. Cependant, ces résultats ne tiennent pas compte du fait que, si toutes les données sont déjà en mémoire, l'initiateur peut exécuter d'autres tâches pendant que l'accélérateur effectue les comparaisons.

4.3. Algorithme d'encodage et de décodage Reed Solomon

L'algorithme d'encodage et de décolage Reed Solomon sont utilisés, entre autres, dans le modem de type XDSL. Ce type d'application est un cas de test intéressant car le transfert de données entre les blocs est très important. La description de l'implantation de l'algorithme est donnée, puis les résultats obtenus sont commentés.

4.3.1. Description de l'implantation de l'encodeur/décodeur ReedSlomon

L'algorithme permet d'encoder et de décoder des données de façon à pouvoir faire de la détection et de la correction d'erreurs. Plus précisément, l'encodeur calcule quelques octets à partir des données fournies et le résultat est annexé aux données avant qu'elles ne soient transmises au bloc suivant. Ensuite, le décodeur peut détecter,

localiser et corriger les erreurs qui ont pu s'insérer lors des différents transferts qu'ont subis les mots décodés.

L'objectif est d'évaluer l'effet de l'utilisation de VCI sur un algorithme où les transferts des données sont importants et non d'en concevoir une implantation réelle et efficace. Ainsi, l'encodeur est implanté en matériel et le décodeur en logiciel, même si l'algorithme de décodage est beaucoup plus dense que celui de l'encodage. L'architecture de droite de la Figure 4.1 est employée et les analyses sont orientées sur l'interface esclave du processeur et l'interface maître du coprocesseur.

Pour permettre à l'encodeur et au décodeur d'accéder aux données, celles-ci sont placées dans la DPRAM du système. En outre, une série de constantes est utilisée pour l'encodage tout comme pour le décodage. Puisque l'espace de ces constantes est important, il serait tout à fait inefficace d'utiliser à la fois la mémoire logicielle et les registres matériels pour les emmagasiner. Par conséquent, toutes les constantes sont placées seulement dans la DPRAM. L'encodeur matériel doit accéder à la DPRAM, par l'intermédiaire de l'interface VCI, pour obtenir les données et les constantes voulues.

De plus, puisque les constantes utilisées dépendent des données à encoder, les requêtes ne sont pas indépendantes les unes des autres. Il n'est donc pas possible d'effectuer les lectures de constantes en bloc. Ces lectures se font à mesure que l'algorithme progresse et en fonction des calculs effectués. La technique utilisée pour le coprocesseur de l'algorithme de filtrage ne peut être employée ici, car l'envoi d'une nouvelle requête est dépendant de la réponse de la requête précédente.

En outre, l'algorithme du coprocesseur n'est pas adapté au protocole. Comme dans [GuRo94] [DMVJ97] ou [KlGa98], toutes les contraintes du protocole sont concentrées dans une procédure et l'algorithme appelle la procédure pour établir les communications avec l'extérieur (Figure 4.4). Lorsque le protocole de

communication est modifié, seule la procédure doit être changée. Cela permet donc de tester si l'utilisation d'une procédure pour contenir toutes les contraintes du protocole est une solution acceptable. Ainsi, si c'est le cas, la synthèse automatique de coprocesseur matériel communiquant avec VCI serait très simple à développer.

Il faut faire très attention lors de l'interprétation des résultats de simulation. En effet, l'algorithme est codé au niveau comportemental et sa synthèse n'est pas directe. Ainsi, même si la procédure est synthétisable et contrôlée par une horloge (dix fois plus rapide que l'horloge VCI), l'algorithme lui-même n'a aucun délai et n'est arrêté que par les appels à la procédure. Ainsi, les délais observés lors des simulations pré-synthèse correspondent aux délais imputés aux communications seulement.

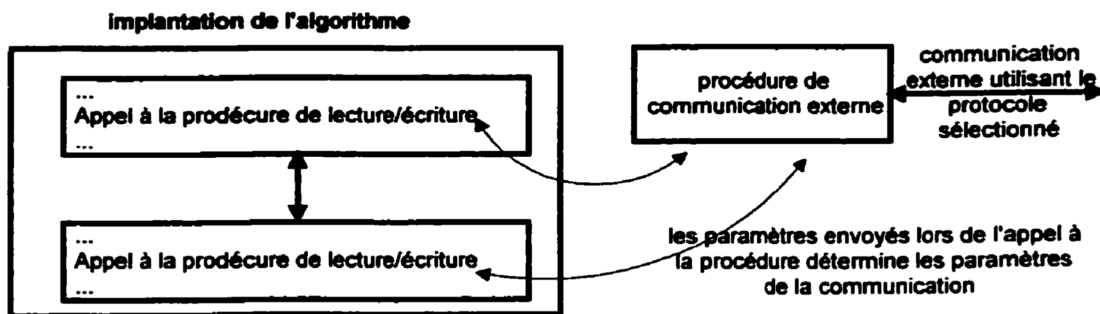


Figure 4.4. Utilisation d'une procédure pour établir les communications

4.3.2. Analyse des résultats de l'encodeur/décodeur Reed Solomon

Pour vérifier l'efficacité de l'utilisation de VCI, une première procédure est conçue pour communiquer directement avec le protocole de la mémoire. Ensuite, une deuxième procédure permet d'accéder à la mémoire à travers des accès VCI. Le Tableau 4.7 donne le temps d'encodage matériel dans les deux cas.

Pour commencer, il faut analyser les délais obtenus pour les lectures et les écritures simples. D'abord, il faut préciser que l'utilisation d'une procédure bloque

complètement l'exécution du programme appelant tant que la procédure n'est pas terminée. La fin de la procédure est atteinte lorsque l'accès mémoire se termine. Ainsi, pour une communication avec VCI, la réponse d'une requête doit être reçue et pour une communication sans VCI, la donnée de lecture doit être arrivée.

Les délais de lecture et d'écriture ne sont pas des nombres entiers de cycles. En effet, les délais sont calculés en fonction des cycles de l'horloge du système à 25MHz. Cependant, la procédure de communication fonctionne à une fréquence de 250MHz. Étant donné les délais de la mémoire utilisée, il est possible d'exécuter les accès mémoire en 0,5 cycle de l'horloge VCI. Dans le cas du processeur, cela n'est pas possible car chacun de ses accès mémoire doit durer un nombre entier de cycles. Le principe est le même pour l'implantation avec VCI, dès que la réponse VCI est reçue, la procédure se termine même si le cycle VCI n'est pas terminé.

En somme, avec un temps d'accès VCI de 1,7 cycles et un temps d'accès sans VCI de 0,5 cycle, la perte de performance causée par l'utilisation de VCI s'élève à 340 % par accès mémoire. Dans le cas de l'encodeur, le pourcentage global est encore plus élevé, étant donné les pertes de cycles causées par la synchronisation.

Ces résultats ne tiennent pas compte du temps d'exécution de l'encodage en tant que tel. Cela correspond donc seulement aux différences de délais imputées aux communications. Ainsi, dans une application réelle, la perte de performance sera toujours pondérée par le pourcentage de l'algorithme constitué d'accès mémoire. Dans le cas de l'encodeur, le nombre d'accès mémoire est très élevé. Par conséquent, même si la perte de performance ne s'élève pas réellement à 377 %, elle est beaucoup trop élevée pour que l'utilisation de VCI soit acceptable.

Tableau 4.7. Délais de l'algorithme d'encodage

[illegible]	[illegible]	[illegible]	[illegible] VCI	
			[illegible]	[illegible]
Temps d'une écriture	1,7	0,5	1,2	340 %
Temps d'une lecture	1,7	0,5	1,2	340 %
Temps total d'encodage	8 935	1 873	7 062	377 %

En conclusion, séparer complètement les détails de communication (dans une procédure) de l'implantation de l'algorithme augmente beaucoup la portabilité, mais peut grandement réduire les performances pour certains protocoles de communication (VCI). De plus, étant donné que le coprocesseur n'est pas contrôlé par une horloge aussi lente que celle de VCI, la perte de performance causée par l'utilisation de VCI se situe autour de 350 % par accès mémoire. Enfin, la perte de performance liée à l'utilisation de VCI est directement proportionnelle à la quantité d'accès aux données d'une application.

4.4. Algorithme du tri par segmentation

L'algorithme de tri par segmentation ("QuickSort") est le dernier qui sera utilisé pour tester l'interface et le protocole. D'abord cet algorithme est très propice à une implantation à plusieurs processeurs. En outre, le partitionnement des tâches entre les processeurs est effectué dynamiquement en fonction des données à trier et du temps que prenne chacun des processeurs pour faire le tri. Enfin, cet algorithme est axé sur le contrôle davantage que sur le transfert de données, mais est fortement dépendant des données. Pour toutes ces raisons, l'évaluation des performances de cette application permet de finaliser l'analyse de l'interface et du protocole.

4.4.1. Description de l'implantation de l'algorithme de tri

L'algorithme de tri par segmentation permet de classer une liste, par exemple de nombres, en divisant la liste initiale en plus petites listes qui peuvent être triées indépendamment. L'initiateur débute avec une liste et la réorganise pour la segmenter

en deux parties qui peuvent être triées séparément. Ainsi, à mesure que l'algorithme avance, plusieurs listes peuvent être triées séparément pour obtenir une liste finale complètement ordonnée.

Cet algorithme peut donc facilement être implanté sur plusieurs processeurs puisque les listes filles peuvent être triées séparément. En outre, selon les données à trier, les listes créées sont différentes. Les tâches des processeurs sont donc déterminées de façon dynamique. Pour implanter l'algorithme, une architecture à deux processeurs indépendants est utilisée (Figure 4.1). Comme dans le cas du filtrage à deux processeurs, la DPRAM est placée du côté de l'initiateur et de l'accélérateur pour vérifier la différence entre les deux implémentations (Figure 4.3).

Un processeur (initiateur) inscrit la liste à trier dans une partie de la DPRAM et amorce le tri. Il ordonne la liste initiale et inscrit la nouvelle liste à la place de la liste initiale et identifie les nouvelles sections à trier. Il inscrit aussi, dans une seconde partie de la mémoire, une liste de tâches à exécuter, correspondant aux sections de la liste à trier. Un pointeur est utilisé pour connaître l'état de la liste des tâches. Ce pointeur est contrôlé par un sémaphore pour éviter que les deux processeurs n'y aient accès en même temps (Figure 4.5).

Ensuite, le deuxième processeur peut effectuer la première tâche non effectuée dans la liste. À son tour, il inscrit des nouvelles tâches dans la liste et modifie le pointeur. Les deux processeurs continuent à trier les sections de la liste initiale jusqu'à ce qu'elle soit ordonnée. Une interruption (par boîte aux lettres) est utilisée par le processeur qui effectue la dernière tâche pour indiquer à l'autre processeur que le tri est terminé. À ce moment, l'initiateur peut commencer le tri d'une nouvelle liste.

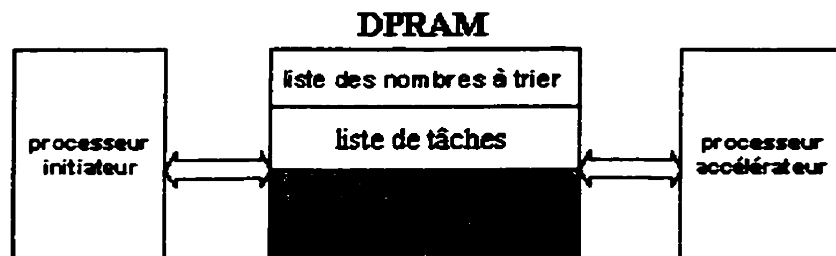


Figure 4.5. Schéma de l'implantation de l'algorithme de tri

4.4.2. Analyse des résultats de l'algorithme de tri

Cinq listes de 15 nombres ont été triées et le temps du tri est reporté au Tableau 4.8. Les implémentations testées sont les suivantes : un processeur, deux processeurs sans VCI, deux processeurs avec VCI (avec la DPRAM placée d'un côté et de l'autre de l'initiateur). D'abord, l'utilisation d'un second processeur accélère le tri de seulement 30 %. Ensuite, l'utilisation de VCI n'influence pratiquement pas les résultats de l'application.

Tableau 4.8. Comparaison des différentes implémentations du tri

[illegible]	[illegible]	[illegible]					
1 processeur	18 240	5 470	30 %	5 408	29,7 %	5 378	29,5 %
2 processeurs sans VCI	1 270			62	0,49 %	92	0,72 %
2 processeurs avec VCI et initiateur côté DPRAM	12 832					30	0,23 %
2 processeurs avec VCI et accélérateur côté DPRAM	12 862						

Le fait d'utiliser un second processeur ne double pas l'efficacité du tri. Ceci s'explique par le fait que les processeurs doivent souvent attendre que le sémaphore soit débloqué pour poursuivre leur travail. Probablement que pour des listes beaucoup plus longues, le temps de tri de chaque section de liste serait plus élevé et la proportion du temps d'attente serait plus faible. Ainsi, l'amélioration causée par un

deuxième processeur serait beaucoup plus marquée. Cet essai n'a pas pu être effectué à cause d'un problème avec l'outil. Le logiciel ne semble pas bien gérer les sémaphores sur de très longues simulations.

VCI n'influence pas beaucoup les résultats d'abord parce que la quantité de données transmises d'un côté à l'autre est très peu élevée. Cet algorithme étant plus axé sur le contrôle, l'effet de VCI est moindre. En outre, le fait que les processeurs passent beaucoup de temps à attendre la libération du sémaphore contribue à atténuer les effets négatifs que VCI pourrait causer. Il reste à voir si, avec de très grandes listes, les résultats sont semblables. En somme, une chose est sûre, pour des applications axées sur le contrôle, l'utilisation de VCI affecte peu les performances.

L'objectif de l'analyse des résultats de synthèse et de simulation des différentes implantations d'algorithmes est de pouvoir tirer des conclusions à propos des performances de l'interface développée et des performances du protocole VCI. Ainsi la méthodologie proposée peut être commentée et des voix de recherches peuvent être proposées. Avant de tirer des conclusions, les résultats obtenus sont résumés. Ensuite, l'interface, le protocole et la méthodologie sont évalués. La conclusion du mémoire présente les voie de recherches proposées.

4.5. Discussion

L'objectif de l'analyse des résultats de synthèse et de simulation des différentes implantations d'algorithmes est de pouvoir tirer des conclusions à propos des performances de l'interface développée et des performances du protocole VCI. Ainsi la méthodologie proposée peut être commentée et des voix de recherches peuvent être proposées. La discussion résume l'évaluation de l'interface, du protocole et de la méthodologie. La conclusion du mémoire présente les voies de recherches proposées.

4.5.1. Évaluation de l'interface

À partir des résultats obtenus, il est possible d'évaluer l'enveloppe du processeur. Des améliorations liées au protocole, aux résultats de synthèse et finalement à la fonctionnalité de l'interface sont données.

4.5.1.1. Améliorations liées au protocole

En utilisant le protocole BVCI à deux poignées de main, une requête et une réponse ne peuvent être envoyée durant le même cycle. Différentes solutions permettant de diminuer les pertes de performances reliées au protocole ont été proposées. Pour simplifier la structure de l'interface, une solution économique pour la lecture a été rejetée. Ainsi, en émettant les requêtes de lecture aussitôt que l'adresse générée par le processeur est valide, un cycle aurait été économisé à chaque lecture. Malgré l'augmentation de complexité de l'interface engendrée par cette solution, il aurait été préférable de ne pas la rejeter.

Les signaux facultatifs permettant de faire des accès par paquet n'ont pas été utilisés. Cependant, lors de tels accès, il est possible de prédire l'adresse avant qu'elle ne soit valide sur le bus. Ainsi, les requêtes peuvent être anticipées et les réponses peuvent être envoyées dans le même cycle que les requêtes sont reçues. Tous les désavantages du protocole de double poignée de main sont éliminés et les effets négatifs de l'utilisation de VCI sont pratiquement enrayés. Il serait donc très intéressant d'ajouter la possibilité d'utiliser ses signaux pour améliorer les performances des applications.

Pour des applications où les accès séquentiels sont limités, l'utilisation du protocole PVCI au lieu du BVCI aurait probablement mené à des résultats beaucoup plus intéressants. En effet, le protocole PVCI est très simplifié et ne contient qu'un mécanisme de poignée de main. Le ARM7TDMI est un processeur qui a des instructions séquentielles limités, des cycles d'accès compressés et un pipeline

restreint. Ainsi, pour des systèmes sans cache contenant ce type de processeur, le PVCI est un choix plus judicieux.

Le ARM9 aurait été un meilleur candidat pour l'utilisation du protocole BVCI. En effet, son pipeline est plus élaboré et ses cycles moins compressés. Ainsi, l'utilisation d'un double mécanisme de poignée de main aurait été beaucoup moins néfaste sur un système comprenant un ARM9.

L'interface a été développée en suivant strictement les recommandations de l'Alliance. Une discussion avec le directeur de la section du bus sur une puce a permis de comprendre que les contraintes strictes ne sont que des propositions et que beaucoup de liberté est laissée au concepteur. Ainsi, le protocole peut être configuré pour supporter d'autres mécanismes que la double poignée de main. Par exemple, en permettant des comportements asynchrones pour les accusés de réception, il est possible d'avoir un mécanisme sans poignée de main. De cette façon, les pénalités entraînées par le mécanisme de double poignée de main sont évitées.

L'utilisation du mécanisme VCI sans poignée de main est déconseillée car il implique l'utilisation de circuits asynchrones. En début de projet, l'utilisation de circuits asynchrones a été rejetée à cause de sa complexité. En effet, comme expliqué à la section 1.1.6.7, ce type de circuit peut conduire à des solutions beaucoup plus performantes, mais sa portabilité s'en trouve grandement réduite.

4.5.1.2. Améliorations liées aux résultats de synthèse

Les résultats de synthèse ont permis d'identifier différents aspects de l'interface qui pourraient être améliorés. D'abord, comme discuté précédemment, il serait essentiel de rendre l'interface physiquement réalisable. En portant des modifications sur la structure de l'interface esclave, il serait possible de diminuer l'espace occupé tout optimisant la fréquence de l'interface.

En premier lieu, la fréquence peut être optimiser en augmentant la proportion du cycle du processeur disponible entre les étages de bascules. Suite au développement d'une première enveloppe complètement asynchrone et très difficile à synthétiser, une deuxième interface synchrone a été développée suivant le modèle donné au chapitre 3. Cependant, les règles de [KeBr00] ont été sur-utilisées et l'interface esclave résultante contient trop de niveaux de bascules. Par conséquent, l'espace occupé est doublement augmenté à cause des multiplexeurs associés aux bascules et la vitesse de fonctionnement diminue. Il serait donc pertinent de revoir la structure (surtout de l'interface esclave) pour tenter de diminuer la profondeur du pipeline et améliorer les performances de l'enveloppe (espace et fréquence).

En outre, étant donné l'espace utilisé par les multiplexeurs, l'utilisation de bascules avec activateur aurait peut-être été une meilleure solution, malgré la complexité d'insertion de la chaîne de balayage. En plus, le délai entre les étages de bascules aurait pu être diminuer en simplifiant les opérations de décodage. En utilisant un nombre fixe de bits dans l'adresse pour identifier le composant accédé, la grosseur et le délai de la logique de décodage seraient diminués. Pour des applications où le maximum de l'espace d'adressage est utilisé, cette solution est à rejeter.

Enfin, le choix d'une horloge de pipeline à une fréquence 10 fois plus élevée que celle de VCI est discutable. Malgré l'augmentation de la complexité de conception, l'utilisation d'une horloge cinq fois plus rapide aurait pu contribuer à faciliter l'obtention d'un circuit réalisable physiquement. En effet, la fréquence nécessaire au bon fonctionnement du pipeline aurait ainsi pu être diminuée. De plus, l'utilisation d'horloge différente pour VCI et pour le processeur aurait pu améliorer les performances des applications.

4.5.1.3. Améliorations liées à la fonctionnalité de l'interface

La structure de l'interface est intéressante, car elle permet de modifier la synchronisation en modifiant seulement le contrôleur de chacun des modules. Cependant, il est très important d'identifier quand l'augmentation de la complexité est justifiée par l'augmentation de flexibilité. D'abord, l'interface développée constitue une première exploration mais beaucoup d'aspects supplémentaires auraient pu être considérés pour améliorer la fonctionnalité et la flexibilité de l'interface. Par exemple, l'utilisation d'autres types de mémoire, l'ajout des signaux facultatifs de VCI, l'utilisation de transferts bloqués ou l'insertion de paramètres permettant de rendre les signaux VCI configurables sont des améliorations de fonctionnalité intéressantes. Ces éléments sont élaborés dans la conclusion du mémoire.

De plus, l'implémentation de la FIFO mérite d'être retravaillée. Aucun mécanisme ne contrôle le flot de données dans la FIFO. La génération d'erreur et l'utilisation des indicateurs est à revoir. La FIFO pourrait être accédée avec la même adresse dans les deux sens. Ainsi, l'implantation n'est pas encore au point. En outre, l'importance accordée à certains aspects et la complexité de l'interface qui en résulte ne vaut pas l'augmentation de flexibilité qui en découle. Entre autres, l'implantation des quatre types d'interruptions directes rend l'utilisation de l'interface plus complexe que flexible. De même, le registre d'interruption pourrait être simplifiée pour en faciliter l'utilisation et diminuer les chances d'obtenir des situations imprévisibles.

4.5.2. Évaluation du protocole BVCI

L'utilisation d'un protocole standard peut réduire le tâche de synthèse des communications. En effet, étant donné que le pont liant l'IP au bus est standard pour tous les IP et est construit par le fournisseur de bus, l'assemblage de tel système est simple. Néanmoins, les signaux traversent deux étages entre le IP et le bus : l'interface VCI et le pont. L'augmentation de l'espace et de la latence provoquée par

ce modèle peut être très néfaste pour de petits IP. Ainsi, l'utilisation de protocole standard est une solution idéale pour des systèmes contenant des gros IP.

Le protocole BVCI (à double poignée de main) peut aggraver cet effet d'architecture à doubles étages. En effet, pour des applications axées sur le transfert de donnée et dont il est impossible d'adapter l'algorithme au protocole, les performances sont affectées. Par exemple, les accès du processeur ARM7 prennent deux fois plus de temps à travers une interface VCI. Pour des applications de contrôle, l'effet est moindre. Pour évaluer le protocole BVCI plus en détail, dans ce qui suit les résultats obtenus sont comparés à ceux des analyses précédentes. Puis, des dispositifs permettant de diminuer les pertes de performances sont proposés. Enfin, des améliorations intéressantes au protocole sont expliquées.

4.5.2.1. Comparaison avec les analyses précédentes de VCI

Dans l'étude menée dans [LSJ+00], l'utilisation d'un protocole standard de communication pour les IP est identifiée comme étant un moyen de simplifier l'assemblage de système. Ceci est vrai, mais comme mentionné précédemment, il faut préciser que pour de petits IP, les pertes d'espace et l'augmentation de la latence peuvent rendre cette solution peu pratique.

Dans [CCS99], un pont entre le protocole BVCI et le bus ASB de ARM est construit. L'espace du pont est évaluée à 1000 portes et aucune information n'est donnée sur la latence. Le nombre de cellule de notre interface est grandement plus élevé que le nombre de portes de leur pont. Cependant, la technologie et le type de cellule utilisé n'est pas précisé et donc la comparaison est difficile. Une interface VCI est aussi construite pour un contrôleur d'interruption. Le temps de communication est évalué comme étant doublé par l'utilisation de VCI. Même si les conditions d'utilisation et d'implantation ne sont pas précisées, ces résultats concordent toutefois avec ceux obtenus dans le cadre de cette étude.

En dernier lieu, dans [LVG00], une implantation à doubles étages avec et sans PVCI est comparée avec une implantation directe (sans pont). Le modèle à doubles étages affecte les performances d'un système, mais pas l'utilisation de VCI plutôt qu'un autre protocole. Ce modèle augmente légèrement la puissance dissipée, l'espace occupé et dans certains cas la latence. Ceci vient corroborer que l'utilisation d'une architecture à double étage est seulement justifiée pour de gros IP. Néanmoins, l'étude traitait du protocole périphérique et non du protocole de base. Ainsi, il semble que le protocole BVCI affecte plus la latence des communications que le PVCI. L'augmentation de la flexibilité du BVCI entraîne une nécessité de prendre des précautions lors de la conception pour éviter la diminution des performances.

4.5.2.2. Dispositifs à utiliser pour éviter des pertes de performances

La première chose à considérer pour diminuer la latence d'utilisation de VCI est les transferts par paquets. La première réponse prendra un cycle avant d'être envoyée, mais les réponses suivantes se suivront successivement. Pour des applications où les transferts par paquet sont limités, l'architecture doit être raffinée pour vraiment améliorer les performances. Par exemple, l'utilisation d'une cache par le processeur contribue à diminuer sa quantité d'accès à travers l'interface VCI et ainsi diminuer les pertes de temps. En outre, l'implantation d'un DMA permet d'augmenter les accès par paquet à travers VCI et peut améliorer les performances d'applications ou des transferts de blocs de données sont nécessaires. L'alternative est d'adopter un autre mécanisme que la double poignée de main et d'avoir des circuits asynchrones.

4.5.2.3. Propositions de modifications au protocole

VSIA est une bonne initiative de standardisation à tous les niveaux grâce aux nombreux groupes de développement et à la participation massive de différentes branches de l'industrie (compagnie de développement d'outils, de développement de IP, d'intégration de systèmes,...). Cependant, le protocole BVCI pourrait être légèrement modifié pour permettre d'enrayer les désavantages qu'il peut entraîner.

Entre autres, l'ajout de signaux de contrôle, comme dans le "Open Core Protocol" [Smit00], permettrait la pleine utilisation de l'espace d'adressage pour le transfert de données. En outre, les limites d'utilisation des différents mécanismes de communication devraient être examinées.

4.5.3. Évaluation de la méthodologie proposée

La méthodologie proposée constitue une extension des outils de Mentor Graphics pour permettre le design de SoC. Les approches de plate-forme et d'assemblage de IP sont utilisées pour construire un système à partir de bibliothèques. Les principes de base sont donc l'allocation et la configuration. L'objectif est de développer des bibliothèques pour faciliter la construction de système. Ensuite, une extension à Renoir doit être développée pour permettre l'intégration des composants et la simulation du système à différents niveaux d'abstraction. Néanmoins, après avoir développé une enveloppe pour un processeur ARM et après l'avoir testée pour différentes applications, des aspects importants sont ressortis.

D'abord, développer un IP réutilisable et configurable est un travail très fastidieux. Il est plus intelligent d'adapter le IP aux applications dans lesquelles il servira. Pour ce faire, il faut cerner les besoins du groupe d'applications et faire les choix d'implémentations et de paramètres en conséquence. Ainsi, la tâche de développement de IP ne revient pas au concepteur d'outils. Les compagnies développant des applications doivent cerner leurs besoins et sous-traiter le développement de IP à d'autres compagnies plus spécialisées (comme la compagnie ARM). Il n'y a donc aucun intérêt à développer plusieurs IP ou plate-forme, il faut plutôt se concentrer sur les moyens d'intégrer des IPs de différentes sources dans une même bibliothèque. Les compagnies Synopsys et Cadence ont d'ailleurs choisi d'utiliser cette voie.

L'objectif doit donc être axé sur l'intégration plus que sur le développement et sur la flexibilité plus que sur l'automatisation. Plusieurs solutions doivent pouvoir être explorées facilement et ce à différent niveaux d'abstraction. Pour faciliter l'intégration, il faut que les IP de la bibliothèque soient enregistrés d'une façon homogène. De plus, la flexibilité peut être améliorée en utilisant le principe de configuration. Ainsi, l'utilisation d'un IP, comme celui conçu dans le projet, est très intéressant, mais ces IP doivent être développés minutieusement par des groupes expérimentés.

Ainsi, les aspects importants (exemple les paramètres) et les compromis pertinents doivent être identifiés. Tous les choix doivent être bien documentés. Des bancs de tests configurables facilitent la vérification et l'utilisation de script facilite la génération. Une structure de capture et de saisie de IP, semblable à celle de Synopsys, doit donc d'abord être mise sur pied. Une bibliothèque peut alors être montée à partir de IP de source différentes et la construction du système, dans Renoir, à partir de cette bibliothèque facilite la création de système. L'étape suivante est de permettre la simulation sur Seamless du système, capturé dans Renoir.

Étant donné les performances du protocole VCI, il n'est pas évident que l'utilisation de ce protocole est la meilleure solution. De plus, le problème de perte de performance pour les petits IP reste présent. Il faudra voir ce que l'industrie décidera d'adopter comme normes, pour savoir identifier le protocole à utiliser. Une solution alternative pourrait être de développer plus qu'une interface pour chaque IP ou d'avoir une bibliothèque de ponts. Ainsi, l'exploration des solutions permettra d'obtenir la meilleure combinaison à implanter pour une application donnée.

CONCLUSION

Récapitulation des objectifs du projet

Après avoir fait une revue des différentes approches de conception SoC proposées dans la littérature, une méthodologie a été proposée. L'objectif du projet n'était pas de développer l'outil complet qui incarnerait la méthodologie proposée, mais plutôt d'en valider les concepts de base. Le projet est réalisé en collaboration avec Mentor Graphics, qui n'a pas d'outil de conception SoC à proprement dit, mais dont les outils existants constituent la base de la méthodologie. La méthodologie proposée utilise les principes d'allocation et de configuration pour intégrer tous les éléments d'un système à partir de bibliothèques de IP et de plate-forme. Pour faciliter l'intégration des parties du système, un protocole de communication de IP en développement, VCI de VSIA, est utilisé.

Pour valider les principes sélectionnés (configuration, protocole,...), une enveloppe configurable est développée pour un processeur ARM avec l'outil Renoir de Mentor Graphics. À partir de l'attribution de valeurs aux paramètres de l'enveloppe, certains mécanismes de communication et détails d'implémentation peuvent être sélectionnés. Ainsi, selon les besoins de l'application à implanter, la bonne enveloppe peut être générée. À partir de ce IP, plusieurs algorithmes sont implantés et simulés, avec Seamless de Mentor Graphics. Ainsi, les performances de l'enveloppe et du protocole ainsi que les principes de base de la méthodologie proposée peuvent être évalués.

Contribution du mémoire

Le développement de l'interface a permis de tirer des conclusions intéressantes en ce qui concerne la conception d'interfaces configurables, de l'utilisation du protocole VCI et des principes intéressants de la méthodologie proposée.

D'abord, l'idée d'utiliser un bloc de contrôle pour synchroniser tous les éléments de l'interface donne à celle-ci une flexibilité intéressante. Chaque module est divisé en un chemin et données et contrôleur. Le chemin de données est sous forme de pipeline dont les étages sont synchronisés par le bloc de contrôle. Il faut cependant faire attention d'éviter d'ajouter trop d'étages au pipeline ou de choisir une fréquence de contrôle trop élevée. De plus, l'utilisation de multiplexeur pour contrôler les bascules peut parfois faire augmenter beaucoup l'espace occupé par l'interface.

En outre, la méthode de division de l'espace d'adressage proposée permet à des composants d'établir toutes sortes de communication sans ajouter des signaux de contrôle. Cette méthode à des inconvénients et il est parfois plus performant d'ajouter directement les signaux de contrôle. Un autre aspect important du développement de IP configurable est le choix des paramètres en fonction des besoins. Il faut effectuer les bons compromis et s'assurer que le nécessaire est inclus et le superflu est retiré.

Ensuite, le protocole a été analysé. D'abord, l'utilisation d'un protocole de IP différent du OCB impose l'utilisation d'un pont. Ainsi, deux étages sont traversés par les données et pour de petits IP cela n'est pas approprié. De plus, le protocole VCI, utilisant un mécanisme de double poignée de main, peut parfois introduire des délais. Si l'application contient peu de transfert de données, les performances sont peu affectées. Cependant, lorsque beaucoup de données sont transférées et qu'il existe une dépendance entre les transferts, des dispositifs particuliers doivent être introduits pour éviter la dégradation des performances. Par exemple, effectuer des transferts par paquet à l'aide d'un DMA est une solution.

Enfin, la méthodologie a été analysée. D'abord, l'objectif ne devrait par être de développer les bibliothèques de IP, mais plutôt de faciliter l'intégration de plusieurs IP dans la bibliothèque. Ensuite, les principes de configurations et d'allocation sont très intéressants, mais doivent être utilisés minutieusement. Les paragraphes suivants présentent les points qui pourraient être abordés dans les prochains projets.

Travaux futurs liés à l'interface développée

D'abord, les améliorations proposées à la section 4.1.2 devraient être considérés. Cela implique de revoir la structure pour diminuer la profondeur du pipeline de l'interface esclave. De plus, la génération de requête de lecture de l'interface maître devrait être corrigée pour éviter la perte de cycle inutile. Ensuite, les signaux facultatifs de VCI devraient pouvoir être utilisés pour permettre le transfert de paquets et éliminer la latence de transfert. De plus, les paramètres des interfaces VCI (maître et esclave) devraient être modifiables. Enfin, il faudrait évaluer la pertinence des quatre sortes d'interruptions directes, plus précisément voir si ces modèles correspondent à un besoin réel dans les applications nécessitant la gestion d'interruption. Finalement, les parties moins développées devraient être améliorées.

De plus, d'autres composants pourraient être ajoutés pour permettre de mieux satisfaire les contraintes d'une application. Par exemple, ajouter une mémoire flash, une mémoire synchrone, une SRAM à simples ports ou des mémoires à 8 ou 16 bits. La conception d'un contrôleur de mémoire hors puce serait aussi intéressante.

En outre, les possibilités de configuration de l'interface devraient être augmentées en ajoutant des constantes au "package" et des clauses "generate". La génération automatique pourrait être améliorée, pour permettre des modifications autres que seulement l'espace d'adressage. Ainsi le code serait optimisé par l'élimination des parties inutilisées. Le choix des paramètres pourrait être guidé et le fichier de configuration pourrait être simplifié pour faciliter le travail de l'intégrateur. De plus, la génération des pilotes logiciels et du banc de test pourraient se faire automatiquement à partir des choix de paramètres.

Travaux futurs liés à l'évaluation du protocole VCI

Pour compléter les résultats de tests de performances sur protocole VCI, un système complet devrait être construit. Ainsi, au lieu de se limiter à la communication point-à-point, il serait possible d'intégrer un OBC ainsi que plusieurs composants. Entre autres, faire communiquer le ARM avec un autre processeur, comme un PowerPC ou même un processeur DSP, serait une bonne alternative. De plus, il serait intéressant de vérifier si la simplicité du protocole PVCI permet effectivement d'éviter la dégradation de performances pour des IP simples. Enfin, il serait très pertinent de vérifier si les pénalités du protocole BVCI pour un processeur ARM9 sont réduites.

Travaux futurs liés au développement d'une méthodologie complète

Comme expliqué plus haut, la première étape à réaliser pour commencer le développement d'outils serait de définir les paramètres de capture des IPs de la bibliothèque. Il faudrait aussi mettre sur pied une base de données pour contenir les informations sur les IP. Une interface usager pourrait assister le concepteur lors de la capture de IP et permettre de remplir adéquatement la base de données. L'extension de Renoir devrait permettre d'accéder la base de données pour instancier les IP désirés dans le circuit. Un lien devrait être construit entre l'outil de capture (extension de Renoir) et l'outil de simulation, Seamless. L'étape suivante pourrait être de permettre la simulation à différents niveaux d'abstractions en utilisant la représentation SLIF de VSIA.

RÉFÉRENCES

[ARM00a] ARM, Ltd (2000). AMBA – Advance Microcontroller Bus Architecture Specification. URL : www.arm.com/Documentation/UserMans.

[ARM00b] ARM, Ltd (2000). ARM7TDMI Data Sheet. URL : www.arm.com/Documentation/UserMans.

[ARM00c] ARM, Ltd (2000). Application Note 25 : Exception handling on the ARM. URL : www.arm.com/Documentation/UserMans.

[ARM00d] ARM, Ltd (2000). Application Note 29 Interfacing a memory system to the ARM7TDMI without using AMBA.
URL : www.arm.com/Documentation/UserMans.

[ARM00e] ARM, Ltd (2000). Reference peripheral specification. URL : www.arm.com/Documentation/UserMans.

[Ashe96] P. J. Ashenden (1996). The designer's guide to VHDL. Morgan Kaufmann Publishers, Inc, USA, 688p.

[BCO96] G. Borriello, P. Chou, R. Ortega (1996). Embedded System Co-design: Towards Portability and Rapid Integration. In G. De Micheli and M. Sami, editeurs, Hardware/Software Co-design, NATO Advanced Study Institute (ASI) series E, vol 310, Kluwer Academic Publishers, pp 243-264.

[BeLe00] R. A. Bergamaschi, W. R. Lee. Designing Systems-on-chip using cores (2000). In Proceedings of the 42th ACM/IEEE Design Automation Conference, Los Angeles, LA.

[Birn00] Mark Birnbaum (2000). Socketization process stocks design-reuse store. EE Times, may 00.
URL : www.eetimes.com/story/OEG20000515S0039.

[BiSa99] M. Birnbaum, H. Sachs (1999). How VSIA answers the SOC dilemma. IEEE Computer, vol 32, no 6, pp. 42-49.

[BML+97] Ivo Bolsens, Hugo De Man, Bill Lin, Karl Van Rompaey, Steven Vercauteren, Diedrick Verkest (1997). Hardware-Software Codesign of Digital Telecommunication Systems. In Proceedings of the IEEE, vol. 85, no. 3, pp. 391-418.

[BNY99] Erik Brunvand, Steven Nowick, Kenneth Yun (1999). Practical advances in asynchronous design and in asynchronous/synchronous. In Proceedings of the 36th ACM/IEEE Design automation conference, New Orleans, LA, pp. 104-109, June 21-25.

[BPM98] Adel Baganne, Jean-Luc Philippe and Eric Martin (1998). A formal technique for hardware interface design. IEEE transactions on circuits & systems, Part 2, vol. 45 no. 5, p. 584.

[CCH+99] Henry Chang, Larry Cooke, Merril Hunt, Grant Martin, Andrew McNelly, Lee Tood (1999). Surviving the SOC Revolution. Kluwer Academic Publishers, Boston, 335 pp.

[CCS99] S. P. Chaudhury, L. Cooke, J. Schultz (1999). Virtual component interface for re-usability. In Proceedings of IP based design 99, Grenoble, France, pp. 119-124.

[CDS00a] Cadence Design System Inc (2000). Cierto Virtual Component Co-Design Link to Implementation URL : www.cadence.com/datasheets/dat_pdf/vcc_implementation.pdf.

[CDS00b] Cadence Design System Inc (2000). Cierto Virtual Component Co-Design Environment.
URL : www.cadence.com/datasheets/vcc_environment.html.

[Charl00] F. Charlot (2000), Comment maîtriser la conception réutilisable. Électronique, janvier 2000, no99, pp70-74.

[ChBo94] P. Chou, G. Borriello (1994). Software Scheduling in the Co-Synthesis of Reactive Real-Time Systems. In Procesings of 31st ACM/IEEE Design Automation Conference, San Diego, CA, pp. 1-4.

[CLG96] K.-S. Chung, C. L. Liu, and R. K. Gupta (1996). An Algorithm for Synthesis of System-Level Interface Circuits. In Proceedings of the IEEE/ ACM IEEE International Conference on Computer Aided Design 96.

[COB92] P. Chou, R. Ortega, G. Borriello (1992). Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems. In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design 92, Santa Clara, CA, pp.488-495.

[COB95a] P. Chou, R. Ortega, G. Borriello (1995). The Chinook Hardware/Software Co-Synthesis System. In Proceedings of the 8th International Symposium on System Synthesis, Cannes, France, pp. 22-27.

[COB95b] P. Chou, R. Ortega, G. Borriello (1995). Interface Co-Synthesis Techniques for Embedded Systems. In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, Santa Clara, CA, pp. 280-287.

[COH+99] Pai Chou, Ross Ortega, Ken Hines, Kurt Partridge, and Gaetano Borriello (1999). IpChinook: An Integrated IP-based Design Framework for Distributed Embedded Systems. In Proceedings of the 36th ACM/IEEE Design automation conference, June 21-25, 1999, New Orleans, LA, pp. 44-49.

[Cowa00] Coware Inc (2000). Coware N2C design System. URL : www.coware.com/pdf/N2C.pdf.

[CRK00] Bill Cordan, Ashwin Rao, Scott Knowlton (2000). Developing Configurable IP for System-on-Chip. In Proceedings of the IP World Forum at Design Con 2000 conference, Santa Clara.

[DGZ98] Rainer Dömer, Daniel D. Gajski, Jianwen Zhu. Specification and Design of Embedded Systems (1998). It+ti magazine, Oldenbourg Verlag, Munich, Germany, No. 3.

[DIMJ97] J-M DAVEAU, T. BEN ISMAIL, G. MARCHIORO, A.A. JERRAYA (1997). Protocol Selection and Interface Generation for HW-SW

Codesign. IEEE Transactions on VLSI Systems, Special Issue on Design Automation of Complex Integrated Systems, March.

[Ditt00] Bill Dittenhofer (2000). Connecting multi source IP to a standard on chip bus architecture. In Proceedings of the System on chip design at Design Con 2000 conference, Santa Clara.

[DMVJ97] J.M. DAVEAU, G. MARCHIORO, C. VALDERRAMA, A. JERRAYA (1997). VHDL Generation from SDL Specification. XIII IFIP Conference on CHDL, Toledo, Espagne, pp 20-25.

[DoGa00] Rainer Dömer and Daniel D. Gajski, Reuse and Protection of Intellectual Property in the SpecC System,. In Proceedings of Asia and South Pacific Design Automation Conference, Yokohama, Japan.

[Erns97] Rolf Ernst (1997). Target architectures. In Jørgen Staunstrup, Wayne Wolf, éditeurs, Hardware/Software Co-Design: Principles and Practice, Kluwer Academic Publishers, Boston, pp. 113-148.

[FaWa00] Brian Faith, Mao T. Wang (2000). Maxinizing Intellectual Property Reuse for PCI. In Proceedings of the IP World Forum at Design Con 2000 conference, Santa Clara.

[GABP98] Guy Gogniat, Michel Auguin, Luc Bianco , Alain Pegatoquet (1998). Communication synthesis and HW/SW integration for embedded system design. In Proceedings of the 6th International workshop on Hardware/Software codesign (CODES/CASHE'98), 15-18 mars, Seattle, WA, p. vi+151, 49-53.

[GaRa94] Daniel D. Gajski, Loganath Ramachandran (1994). Introduction to High-Level Synthesis. Ieee design & test of computers, v 11, n 4, p. 44.

[GaVa95] Daniel D. Gajski, Frank Vahid (1995). Specification and design of embedded hardware/software systems. Ieee design & test of computers, vol. 12, no. 1, p. 53.

[GCM92] R. K. Gupta C. Coelho and G. De Micheli (1992). Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software. In Proceedings of the 29th ACM/IEEE Design automation Conference, June 8-12, Anaheim, pp225-230.

[GDG00] Gajski, Zhu, Domer, Gerstlauer, Zhao (2000). SpecC Design Methodology. Kluwer Academic Publishers, Boston, 313 pp.

[GGB97] Jie Gong, Daniel D.Gajski, Smita Bakshi (1997). Model refinement for hardware-software codesign. ACM Trans. Des. Autom. Electron. Syst. 2, vol. 1, pages 22 – 41.

[GuMi97] R. K.Gupta, G. De Micheli (1997). Specification and Analysis of Timing Constraints for Embedded Systems. Ieee transactions on computer-aided design, vol. 16, no. 3, p. 240.

[GuRo94] P. Gutberlet, W. Rosenstiel (1994). Timing preserving interface transformations for synthesis of behavioural VHDL. In Proceedings of European design automation conference, sept. 19-23, Grenoble, France, pp. 618-623.

[GVNG94] Daniel D. Gajski, Frank Vahid, Sanjiv Nayaran, Jie Gong 1994). Specification and Design of Embedded Systems, Prentice Hall, Englewood Cliffs, NJ, 450 pages.

[GZGH00] A. Gerstlauer, S. Zhao, D. Gajski, A. Horak (2000). SpecC System-Level Design Methodology Applied to the Design of a GSM Vocoder. In Proceedings of the Ninth Workshop on Synthesis and System Integration of Mixed Technologies, Kyoto, Japan.

[HCM+99] F. HESSEL, P. COSTE, Ph. LE MARREC, N.E. ZERGAINOH, J.M. DAVEAU, A.A. JERRAYA (1999). Communication Interface Synthesis for Multilanguage Specifications. RSP'99, Clearwater, USA.

[Jani00] Dhanendra, Jani, Tensilica Inc (2000). Raising the reuse bar with processors. EE Times, may 00.
URL : www.eetimes.com/story/OEG20000515S0031.

[KCK+99] Alex Kondratyev, Jordi Cortadella, Michael Kishinevsky, Luciano Lavagno, Alexander Yakovlev (1999). Automatic synthesis and optimization of partially specified asynchronous systems. In Proceedings of the 36th ACM/IEEE Design automation conference, June 21-25, New Orleans, LA, pp. 110-115.

[KCK98] M. Kishinevsky, Jordi Cortadella, Alex Kondratyev (1998). Asynchronous interface spécification, analysis and synthesis. In Proceedings of the 35th ACM/IEEE Design automation conference, June 15-19, San Francisco, CA, pp. 2-7.

[KeBr99] M. Keating, P. Bricaud (1999). Reuse Methodology Manual. Kluwer Academic Publishers, Massachusetts, 286p.

[KnMa98] Peter Voigt Knudsen and Jan Madsen (1998). Communication estimation for hardware/software codesign. In Proceedings of the 6th International workshop on Hardware/Software codesign (CODES/CASHE'98), 15-18 mars, Seattle, WA, p. vi+151, 55-59.

[KnMa99] Peter Voigt Knudsen, Jan Madsen (1999). Integrating communication protocol selection with hardware/software codesign. IEEE Transaction on computer-aided design of integrating circuits and systems, vol. 18, no. 8, pp. 1077-1095.

[Lee00] James M. Lee. IP reuse fact and fiction (2000). In Proceedings of the IP World Forum at Design Con 2000 conference, Santa Clara.

[LiGu98] J. Li, R. K. Gupta (1998). HDL Code Restructuring Using Timed Decision Tables. In Proceedings of the 6t International Workshop on Hardware/ Software Co-Design (CODES/CASHE '98), Seattle, Washington (USA).

[LiVe94] Bill Lin, Steven Vercauteren (1994). Synthesis of Concurrent System Interface Modules with Automatic Protocol Conversion Generation. In Proceedings of the IEEE International Conference on Computer-Aided Design, p. 101-108.

[LiVe96] Bill Lin, Steven Vercauteren, Hugo De Man (1996). Embedded Architecture Co-Synthesis and System Integration. In Proceedings of the

4th International Workshop on Hardware/Software Codesign (CODES/ CASHE '96), p. 2-9.

[LKSN99] J. Lahteenmaki, M. Kuulusa, I. Saastamoinen, J.Nurmi (1999). General Bus Interface for IP design. In Proceedings of IP based design, Grenoble, France, pp. 133-137.

[LSJ+00] C.K.L. Lennard (Cadence), P. Shaumont (IMEC), G.de Jong (Alcatel), A.Haverinen (Nokia), P. Hardee (Coware) (2000). Standard for System-Level Design : Pratical Reality or Solution in Search of a Question? In Proceeding of Design Automation and Test in Europe Conference and Exhibition, pp 576-583.

[LVG00] Roman L. Lysecky, Frank Vahid, Tony D. Givargis (2000). Experiment with the Peripheral Virtual Component Interface. In Proceedings of the 13th International Symposium on System Synthesis (ISSS'2000), 20-22 sept, Madrid, Espagne, pp 221-224.

[MaHa95] Jan Madsen, Bjarne Hald (1995). An approach to Interface Synthesis. In Proceedings of the 8th International Symposium on System Synthesis, pp. 16-21.

[MBL+96] Hugo De Man, Ivo Bolsens, Bill Lin, Karl Van Rompaey, Steven Vercauteren, Diederik Verkest (1996). Co-Design of DSP Systems. In G. De Micheli and M. Sami, editors, Hardware/Software Co-design, NATO Advanced Study Institute (ASI) series E, vol 310, Kluwer Academic Publishers, pp 75-104.

[Mert00] J.S. Mertoguno. VCI : a stantard for on chip bus interface (2000). In Proceedings of the System on chip design at Design Con 2000 conference, Santa Clara.

[MiGu93] G. De Micheli, R. K. Gupta (1993). Hardware-Software Co-Synthesis of Digital Systems. IEEE Design and Test of Computers, vol.10, no. 3, pp. 29-41.

[Moli00] A. Molina. Advance Risc Machines, SGS-Thomson and Siemens Partnership with OMI and achievements. URL : www.omimo.be/members/book_molina.html.

[NaGa94] Sanjiv Narayan, Daniel D. Gajski (1994). Synthesis of System-Level Bus Interfaces. In Proceedings of European Conference on Design Automation, Paris, France.

[NaGa95] S. Narayan and D. Gajski (1995). Interfacing incompatible protocols using interface process generation. In Proceedings of the 32nd ACM/IEEE Design automation conference, June 12-16, San Francisco, pages 468-473.

[NiMa98] Ralf Niemann, Peter Marwedel (1998). Hardware/Software co-design for data flow dominated embedded systems. Kluwer Academic Publishers, Boston, 244 pp.

[NVG91] S. Nayaran, F. Vahid, D. Gajski (1991). System specification and synthesis with the SpecCharts language. In Proceedings of the

IEEE/ACM International Conference on Computer-Aided Design, Santa Clara, CA.

[ObJe92] Kevin O'Brien, Ahmed Amine Jerraya (1992). SOLAR : An intermediate format for system-level design and specification. . In Proceedings of the 1th International workshop on Hardware/Software codesign (CODES/CASHE'92), Grassau, Germany.

[OrBo97] R. Ortega, G. Borriello (1997). Communication Synthesis for Embedded Systems with Global Considerations. In Proceedings of the 5th International Workshop on Hardware/Software Co-Design (CODES/CASHE'97), Braunschweig, Germany, pp. 69-74.

[OrBo98] Ross B. Ortega and Gaetano Borriello (1998). Communication Synthesis for Distributed Embedded Systems. In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, San Jose, CA.

[PRS98] R. Passerone, J. Rowson, A Sangiovanni-Vincentelli (1998). Automatic synthesis of interface between incompatible protocols. In proceedings of the 35th ACM/IEEE Design Automation Conference.

[Romp00] K. V. Rompaey, CoWare Inc (2000). Smooth interfaces ease IP-to_IP talk. EE Times, may 2000. URL : www.eetimes.com/story/OEG20000515S0029.

[Runn00] J. Scott Tunner, Conexant Systems (2000). Building an infrastructure for IP reuse. EE Times, may 00. URL : www.eetimes.com/story/OEG20000515S0038.

[RVBM96] Karl Van Rompaey, Diederik Verkest, Ivo Bolsens, Hugo De Man. CoWare (1996). A design environment for heterogeneous hardware/software systems. In proceedings of European Design Automation Conference (EURO-DAC).

[SCC00] Warren Savage, John Chilton, Raul Camposano (2000). IP Reuse in the System on a Chip Era. In Proceedings of the 13th International Symposium on System Synthesis (ISSS'2000), 20-22 sept, Madrid, Espagne, pp 2-7.

[ScCh97] R. L. Schwartz, T. Christiansen (1997). Learning Perl. O'Reilly & Associates Inc., Etats-Unis, 269p.

[Sett00] Curtis Settles (2000). Silicon Development Platform Simplifies System design. In Proceedings of the IP World Forum at Design Con 2000 conference, Santa Clara.

[Smit00] Ed Smith, Sonics Inc, Mountain View (2000). Bus protocols limit design reuse of IP. EE Times, may 2000. URL : www.eetimes.com/story/OEG20000515S0026.

[SuBr92] J. Sun and R Brodersen (1992). Design of system interface module. In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, pp. 478-481.

[TaKn00] Danesh Tavana, Steve Knapp (2000). A configurable system on chip device facilitates customization and reuse. In Proceedings of the System on chip design at Design Con 2000 conference, Santa Clara.

[VaGi00] Tony D. Givargis, Frank Vahid (2000). Parametrized system design. In Proceedings of the 8th International workshop on Hardware/Software codesign (CODES/CASHE'00), San Diego, pp 98-102.

[VCV+95] C.A. VALDERRAMA, A. CHANGUEL, P.V. VIJAYA-RAGHAVAN, M. ABID, T. BEN ISMAIL, A.A. JERRAYA (1995). A unified model for co-simulation and co-synthesis of mixed hardware/software systems. In Proceedings of European Design and Test Conference (EDAC-ETC-EUROASIC'95), 6-9 March, Paris, France.

[VIJK94] Markus Voss, Tarek Ben Ismail, Ahmed A. Jerraya, Karl-Heinz Kapp (1994). Towards a Theory for Hardware/Software Codesign. In Proceedings of the 3rd International Workshop on Harware/Software Codesign(CODES/CASHE'94), pp. 173-180.

[VSIA00a] VSIA Alliance, On-Chip Bus Development Working Group (2000). Virtual Component Interface Specification (OCB 2 2.0), available to VSIA members.

[VSIA00b] VSIA Alliance, System-Level Design Development Working Group (2000). System-Level Interface Behavioral Documentation Standard (SLD 1 1.0), available to VSIA members.

[WaBo94] E. Walkup, G. Borriello (1994). Interface Timing Verification with Application to Synthesis. In Proceedings of 31st ACM/IEEE Design Automation Conference, San Diego, CA, pp. 106-112.

ANNEXES

ANNEXE A : SYSTÈME CHINOOK

Pour garder une vue d'ensemble du système et pour permettre la portabilité du design de modules et l'intégration des différents éléments lors d'un design au niveau système, Borriello et collaborateurs proposent un outil appelé Chinook. Cet outil sera d'abord présenté de façon générale. Puis, une description de la méthodologie de la synthèse des communications sera présentée suivi de la méthodologie de synthèse d'interface.

A.1. Description générale du système

A.1.1. Description de l'outil

Cet outil est destiné aux systèmes dominés par les signaux de contrôle et permet de faire une exploration vaste de l'espace des solutions et de faire la synthèse de l'interface logicielle/matérielle. Dans [BCO96], on dit que l'outil, à partir d'une spécification unique, offre un seul environnement de simulation, permet l'ordonnancement logiciel, traite la synthèse d'interfaces et fournit le fichier d'entrée pour la phase finale du design du système.

Dans [COB95a], on décrit le flux de design suivi par Chinook. L'outil comprend un parser (programme d'analyse syntaxique), une bibliothèque de microprocesseurs, de périphériques et d'interfaces, un synthétiseur d'interfaces (interface synthesizer), un synthétiseur de communication (communication synthesizer), un ordonnateur et un simulateur (Figure A.1). Il faut noter que Chinook ne fait aucun partitionnement. On suppose donc que le partitionnement sera toujours effectué par les concepteurs et que la liste des modules à implémenter en logiciel et en matériel sera donnée en entrée à l'outil. Ce dernier permet cependant une formalisation des contraintes de temps et de performance.

Le fichier d'entrée en Verilog contient une description comportementale (indiquant si le module est implémenté en logiciel ou en matériel) et structurelle (liste des processeurs utilisés, des dispositifs périphériques ainsi que les modules de communication). En séparant la description structurelle de la description comportementale, on peut changer l'architecture choisie facilement sans refaire la description comportementale du système. L'architecture de base suppose un ou plusieurs processeurs (maîtres) communiquant avec des périphériques (esclaves). Ce sont les pilotes de périphériques qui permettent la communication entre les processeurs et les périphériques. Cependant, si cette communication directe n'est pas possible, un module matériel est inséré entre les deux composants (ce module matériel devient lui-même un périphérique).

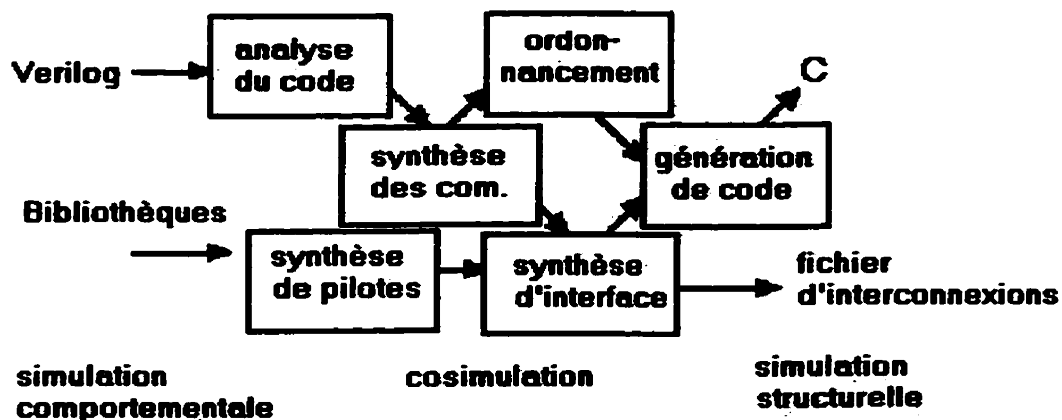


Figure A.1. Le système de co-synthèse Chinook

A.1.2. Bibliothèques

La bibliothèque de périphériques contient la spécification des interfaces des différents périphériques et leur modèle de simulation (en C). Pour générer la bibliothèque, on doit avoir une liste des ports et une routine d'accès pour ces ports ainsi que les informations sur les délais sous forme de chronogrammes et de fichiers en Verilog. Pour chaque port, on enregistre ensuite cette information sous forme de SPRs (static pin rules) représentant l'information matérielle (direction et activation du port) et des

SEQs (sequences) enfermant les routines logicielles. Pour communiquer avec les périphériques, le processeur doit générer une série de signaux et on représente ces séquences de signaux avec les SEQs. On dit que les SEQs sont des représentations textuelles des chronogrammes des périphériques adaptés au processeur utilisé (par l'ajout de paramètres sur les ports). Le pilote de périphériques utilisera ces SEQs pour accéder aux ports des périphériques.

La bibliothèque de microprocesseurs contient une liste des ports ainsi que le type des ports et les chronogrammes des signaux sur ceux-ci. On inscrit aussi une estimation du temps d'exécution du logiciel. Pour générer les pilotes d'interfaces, on doit fournir une liste des instructions de communication et une liste des fonctions spéciales pour la communication ainsi que les ressources d'entrées/sorties disponibles. On a aussi une bibliothèque d'interfaces contenant les descripteurs matériels et logiciels (registres, multiplexeurs,...).

A.1.3. Formalisation des contraintes de temps

Toute l'analyse est basée sur des modes à l'intérieur desquels on définit des contraintes précises. Ces contraintes de haut niveau peuvent être des contraintes de taux de transfert (donnant une référence sur le temps entre les itérations) et de temps de réponse (limitant le temps pour faire une transition de mode). Les transitions de mode et les événements sur les entrées/sorties sont générés par des handlers.

Avant toutes étapes de design, Chinook génère les contraintes de temps du système : c'est la formalisation des contraintes de temps. Dans [WaBo94], on décrit comment spécifier les contraintes de synchronisation des interfaces. On divise d'abord les contraintes de synchronisation en deux catégories : délai de propagation (équivalant aux contraintes maximales) et contraintes de temps ou de performance (équivalant aux contraintes linéaires). On divise aussi les contraintes linéaires en contraintes requises et garanties. On exprime toutes ces contraintes sous une seule contrainte de

façon à construire un graphe et on propose un algorithme déterminant les bornes de temps d'exécution du circuit pour le respect des contraintes.

A.1.4. Ordonnateur

Une fois que toutes les ressources sont allouées, Chinook permet un ordonnancement de bas niveau pour les entrées/sorties et de haut niveau pour les opérations. L'ordonnancement statique est à grain fin et permet de rencontrer les contraintes minimum et maximum en utilisant des heuristiques. On unit les handlers d'un même mode et on fait l'ordonnancement à partir de ce nouveau système en maintenant l'intégrité des états de tous les handlers. On génère donc ici du code C qui pourra être compilé par la suite pour un microprocesseur spécifique. Dans [ChBo94] on explique l'ordonnancement logiciel basé sur les modes. On fait de l'ordonnancement intramode et intermode.

A.1.5. Synthétiseur d'interface

Chinook fournit aussi un synthétiseur d'interfaces. Lorsqu'un module ne peut être implémenté en logiciel, on génère le matériel nécessaire pour son implémentation et ce module devient un périphérique. La synthèse d'interfaces se ramène donc toujours à l'établissement de la communication entre un processeur et un périphérique. On donne trois grandes étapes pour la synthèse d'interfaces : la synthèse des pilotes des périphériques, l'allocation des ports entrées/sorties et l'allocation des ports mémoire. La synthèse de pilote se fait à partir des contraintes de temps contenues dans les chronogrammes. On analyse les différents fichiers et on génère le code nécessaire (en faisant un ordonnancement linéaire) pour permettre l'accès au périphérique en ajoutant de la logique combinatoire en matériel si nécessaire. Ensuite, il faut faire l'allocation des différents ports des processeurs aux différents périphériques. On a deux sortes de ports : les ports d'entrée/sorties dédiés et les ports mémoire. On alloue d'abord les ports dédiés puis les ports mémoire. À mesure que l'on avance

dans l'allocation, on met à jour les routines d'accès en tenant compte de l'association des ports des processeurs aux différents périphériques.

A.1.6. Synthétiseur de communications

Chinook permet de faire la synthèse des communications lorsque le système contient plus d'un microprocesseur. C'est le synthétiseur de communications qui permet de générer le logiciel et le matériel pour le transfert de données entre les processeurs. Ce synthétiseur détermine les données à transmettre, les mécanismes utilisés et les éléments nécessaires à la communication (mémoire, tampon, logique combinatoire,...). Cette synthèse de canaux de communication est faite à partir d'une bibliothèque de composantes de communication (tampon, FIFO, arbitres, ...). On peut utiliser différentes topologies (bus, point-à-point, mémoire partagée,...) ou protocoles (bloquant ou non-bloquant, maître-esclave, ...). Une fois la synthèse de communications terminée (on a identifié les méthodes de communication) on peut passer à la synthèse des interfaces et faire l'association des ports des processeurs.

A.1.7. Simulateur

Puisque l'intérêt est au niveau de la synthèse plutôt qu'au niveau de la simulation, il n'y a pas d'intérêt à décrire le simulateur. Cependant, il faut préciser que Chinook offre la possibilité de simuler tout au long des étapes de la synthèse.

A.2. Synthèse des communications

A.2.1. Modèle de communication

Dans [OrBo97] et dans[OrBo98], on traite de la synthèse des communications en gardant une vue globale du système. On suppose que l'on débute avec une description comportementale du système et une architecture de base prédéfinie. On utilise un modèle où plusieurs processeurs communiquent entre eux de façon non-bloquante pour permettre de séparer le calcul de la communication (on associe une

queue à chaque processus récepteur ou transmetteur). Notons que des communications bloquantes peuvent être implémentées par la génération de messages de reconnaissance. Chaque message contient un nom d'événement et des données (facultatif).

Chaque processus peut spécifier des attributs de réception (grandeur de la queue, temps de réponse,...) et d'envoi. Les handlers sont ceux qui gèrent les communications entre les modules. En effet, selon les messages qu'ils reçoivent, ils modifient l'état des variables, effectuent les changements de modes, envoient des messages et se terminent. Aucun handler d'un même processus ne peuvent fonctionner en même temps. Il existe des handlers qui agissent en fonction du temps et non en fonction des messages d'entrée. L'utilisation de modes permet de définir des comportements mutuellement exclusifs et de faire le filtrage de messages.

A.2.2. Entrées et sorties du processus de synthèse

À partir de l'association entre les processus communiquants et des temps de réponse prescrits, la synthèse de communication, on peut générer un système d'exploitation en temps réel (donc noyau temps réel) pour chacun des processeurs et adapter le code des pilotes de périphériques. Les données nécessaires au processus de synthèse sont une description comportementale du système, une spécification de l'architecture utilisée, une description des processus communiquants, une spécification de la topologie de bus utilisée avec le protocole des bus ainsi qu'une association entre les messages à transmettre et les bus utilisés pour la transmission. On doit donc fournir le chemin emprunté par chacun des messages.

A.2.3 *Synthèse des communications multi-hop*

Les étapes de la synthèse de communication sont très bien décrites dans [COH+99]. D'abord, on a deux types de communications : intraprocesseurs et inter-processeurs. Le premier cas est assez facile à gérer puisque toutes les communications entre

processus d'un même processeur se font à travers la mémoire du processeur en question. Dans le deuxième cas, il faut d'abord savoir si le message passe par des hôtes intermédiaires (Figure A.3). Dans ce cas, il faut diviser le trajet pour n'avoir que des transferts entre des unités adjacentes. On doit ensuite trouver le temps limite pour l'arrivée de chaque message à chaque instance et on fournit un algorithme et des équations pour y arriver (Figure A.2). On obtient alors un problème homogène à régler : la communication entre dispositifs directement connectés avec des contraintes de temps de réponse. Notons que pour chacun des messages passant par un dispositif intermédiaire, il faut générer un processus de routage sur ce processeur.

$$\mathrm{minXmitTime}_i = \frac{\mathrm{messageSize}_i + \mathrm{protocolOverhead}_i}{\mathrm{bw}_i}$$

$$\mathrm{OptDelay} = \max\left(\forall \mathrm{paths}\left(\sum_{path_j} \mathrm{minXmitTime}_i + \mathrm{hopDelay}_j\right)\right)$$

$$\mathrm{ExtraTime} = \mathrm{Deadline} - \mathrm{OptDelay}$$

$$\mathrm{effBw}_i = (1 - \mathrm{utilization}_i)\mathrm{bw}_i$$

$$\mathrm{ETR}_i = \mathrm{ExtraTime} - \sum_{k=1}^{i-1} \mathrm{extraTime}_k$$

$$\mathrm{extraTime}_i = \frac{\frac{1}{\mathrm{effBw}_i}}{\frac{1}{\mathrm{effBw}_i} + \max\left(\forall \mathrm{paths}\left(\sum_{x=i+1}^{\mathrm{endOfPath}} \frac{1}{\mathrm{effBw}_x}\right)\right)} \mathrm{ETR}_i$$

$$\mathrm{deadline}_i = \mathrm{minXmitTime}_i + \mathrm{extraTime}_i$$

Figure A.2. Équations de détermination du délai de communication

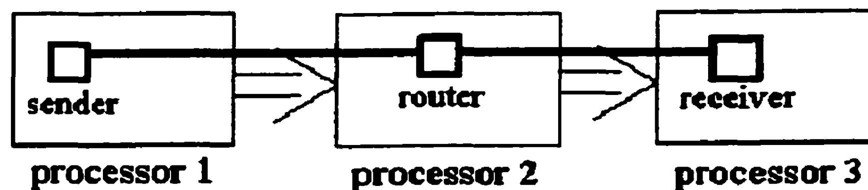


Figure A.3. Communication multi-hop

A.2.3. Synthèse des communications à un seul bus

Pour effectuer la synthèse, on groupe d'abord tous les messages utilisant le même bus. Puis, on assigne des attributs à chacun de ces messages en fonction du schéma d'arbitrage et des protocoles utilisés. On propose plusieurs schémas d'arbitrage (priorité basée sur les messages, sur les processeurs, sur les maîtres-esclaves,...). Lorsqu'on a fixé les attributs des messages et particularité de la communication, la description comportementale peut être modifiée en fonction des décisions prises. Ensuite, on alloue les queues (espace mémoire) en fonction des attributs et de la description comportementale. Puis, on génère les pilotes de périphériques associés aux protocoles des bus à partir de la bibliothèque de protocoles. La dernière étape est de modifier le noyau temps réel des processeurs pour permettre l'envoi et la réception de messages.

Les pilotes des périphériques associés aux protocoles des bus sont constitués de trois handlers. Le premier exécute le protocole d'envoi de messages. Il génère donc les messages à envoyer et reconstitue les messages reçus en communiquant seulement avec la queue de façon non-bloquante. Le deuxième permet de gérer les interruptions servant à indiquer l'arrivée ou l'envoi d'un message. Le troisième fait le lien entre la queue et l'extérieur selon les indications de l'ordonnateur du microprocesseur. On peut faire appel au routeur du processeur pour faire l'envoi des messages venant du troisième handlers vers le bon port.

A.3. Synthèse de l'interface

A.3.1. Définition de la synthèse des communications

Lorsqu'un microprocesseur doit communiquer avec des périphériques, il faut permettre au logiciel qui s'exécute sur le microprocesseur de communiquer avec l'extérieur par l'intermédiaire de pilotes. Il faut aussi assigner les ports du processeur aux périphériques et personnaliser les routines d'accès selon l'assignation des ports.

Le but est de minimiser la grosseur du code et la quantité de matériel pour que l'utilisation d'un microprocesseur soit rentable. Lorsqu'il y a plus de ports de périphériques que de ports de processeur, il faut utiliser des multiplexeurs. De plus, grouper les signaux peut éventuellement réduire la taille du code des pilotes.

Dans [COB95b], on explique la méthodologie utilisée pour générer l'interface de communication entre un processeur et des périphériques en utilisant le moins de matériel possible mais en respectant les contraintes. Les données nécessaires au processus de synthèse sont : une description comportementale du système (que l'on devra CFG) et une description des périphériques et processeurs (qu'il faut implémenter dans une bibliothèque pour appliquer l'algorithme). Les CFGs appellent les pilotes de périphériques pour communiquer avec l'extérieur. Par défaut les CFGs sont en logiciel mais on peut les identifier comme devant être implémentés en matériel. On fournit à la sortie de la synthèse des descriptions Verilog des parties matérielles et le code des pilotes de périphériques.

A.3.2. Type de connexion

On définit deux types de connexions : entrées/sorties directes et entrées/sorties indirectes. Une entrée/sortie directe est connectée directement sans logique combinatoire. On donne trois situations dans lesquelles on a besoin de connexion indirecte : pénurie de ports d'entrées/sorties (on doit alors ajouter de la logique de multiplexage pour partager les ports), nécessité d'avoir un séquenceur d'entrées/sorties (lorsque l'on est restreint aux délais des instructions du processeur et que le périphérique a des contraintes de temps de réaction ou de verrouillage), besoin d'un coprocesseur pour réduire la charge du processeur.

A.3.3. Algorithme général de la synthèse

L'algorithme général de la synthèse débute avec la synthèse des CFGs à implémenter en matériel. Ces CFGs sont implémentés via des séquenceurs d'entrées/sorties. On crée le code correspondant au séquenceur et on modifie les SEQs en fonction de ce

changement (algorithme donné plus bas). On ajoute ensuite les séquenceurs dans la liste des périphériques et on fait l'allocation des ports directs (ceux qui sont directement connectés au processeur). On essaie d'abord de faire l'assignation des ports dédiés d'entrées/sorties du processeur, puis on utilise les ports mémoire (Figure A.4). S'il reste toujours des ports non connectés, on retourne une erreur. Notons que les ports indirects sont déjà implémentés par les séquenceurs. Cependant, lors de l'allocation des ports dédiés du processeur, on ajoute en dernier recours une logique combinatoire pour forcer le partage des ports (ces ports deviennent donc indirects). En dernière étape, on génère le code des pilotes de périphériques du processeur.

Pour faire l'allocation des ports, on classe des ports des périphériques en trois groupes: les gardes (déterminent l'activation des ports gardés), les ports gardés (leur activation est déterminée par les gardes) et les ports non-gardés (sont toujours actifs et ne peuvent pas partager des liens de communication). Tous les ports non-gardés (incluant les gardes) sont associés de façon unique à un port du processeur mais les ports gardés peuvent être groupés avant d'être associés. On suppose que les ports des périphériques ayant plus de ports que le processeur ont été divisés manuellement. Un port d'entrée est divisible lorsque la division ne générera pas de problèmes de consistance de données. Un port de sortie est divisible lorsqu'il est gardé et lorsqu'il reste stable tant que le garde est vrai. Si on doit diviser un port non-divisible, il faut ajouter de la logique pour éviter les problèmes (par exemple ajouter un garde pour un port non-gardé).

A.3.4. Allocation des ports dédiés d'entrées/sorties

L'allocation des ports dédiés d'entrée/sorties fait l'objet d'un article complet [COB92]. On fournit alors un algorithme permettant de passer à travers la liste de ports des périphériques et de générer l'interface nécessaire à la communication (l'association des ports et le code des pilotes de périphériques). Le programme ne doit contenir qu'un seul *thread*. La structure des données utilisées et générées par

l'algorithme est contenue dans 5 listes (netlist, binding-list, free-list, device-list et dport-list). On initialise d'abord toutes les listes puis on effectue les associations nécessaires aux fonctions spéciales de communication. Puis on appelle une fonction récursive utilisant les ports d'entrées/sorties spécifiques du microprocesseur. S'il y a un manque de ports, on termine le développement de l'interface en utilisant les ports mémoire du microprocesseur.

Expliquons maintenant comment fonctionne la fonction récursive. On prend chacun des ports des périphériques et on vérifie sur quel port du processeur il est possible de les connecter. On parcourt la liste des ports du processeur et on identifie le premier port avec lequel il est effectivement possible de l'associer. Pour ce faire, on vérifie qu'il n'y a pas d'accès simultané (lorsque plusieurs ports de périphériques connectés sur la même entrée du processeur sont actifs en même temps) ou de problème de synchronisation à l'intérieur d'une séquence (lorsque deux ports du même périphérique connectés sur le même port du processeur sont actifs dans la même séquence). Si on trouve une entrée du processeur pour connecter le port du périphérique et si ce port est le premier de cette grandeur à avoir une connexion unique, on tente une autre voie (backtrack).

En cas d'échec, on force le partage de port en éliminant les accès simultanés. On y arrive en insérant des registres (port d'entrée), des tristates (ports de sorties) et des verrous bidirectionnels (ports d'entrées/sorties). Forcer le partage lorsque le port est déjà partagé permet d'éviter l'ajout de gardes. Si on se butte encore une fois à un échec, on essaie d'encoder les informations à l'aide de décodeurs (entrées de périphériques qui sont one-hot), de décodeurs registrés (entrées de périphériques qui ne sont pas one-hot) et de multiplexeurs (sorties de périphériques). Une fois qu'un port est bien associé, on peut faire le binding et le linking.

A.3.5. Allocation des ports mémoire

Ensuite, dans [COB95b] on donne un algorithme pour l'allocation des ports mémoire du processeur. On suppose que l'on prend le système où il était lorsque l'on a terminé l'allocation des ports dédiés. On a donc aucun port plus grand que les ports des processeurs. De plus, on suppose que les ports non-gardés sont alloués ou qu'ils ont été transformés en ports gardés. De plus, on suppose que la connexion des ports est directe, car la logique de multiplexage ou les séquenceurs ont déjà été implémentés. Pour activer les ports des périphériques en utilisant les ports mémoire du processeur, on doit générer un appariement d'adresse en tentant de minimiser le matériel. On associe donc un espace d'adressage aux entrées/sorties. L'entrée du processus est l'espace d'adressage, la liste des périphériques à connecter, et la description du comportement du processeur. On fournit ensuite la logique d'appariement d'adresse, les connexions avec le processeur et les nouveaux SEQs ainsi que les instructions pour accéder aux ports mémoire et réaliser la communication.

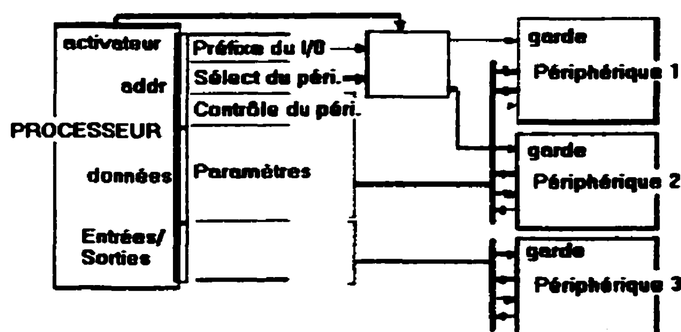


Figure A.4. Association des ports dans Chinook

L'algorithme assigne d'abord les ports gardés (non gardes) à des ports de données ou d'adresse selon le cas. Puis, il assigne un espace d'adressage pour identifier les périphériques et génère la logique d'appariement. Il connecte ensuite la sortie du module d'appariement au garde de chaque périphérique. Pour associer les ports au bon espace (d'adresse ou de données), il utilise une méthode précise. Premièrement, il divise l'espace d'adresse en trois : I/O (Input/Output) prefix (cette valeur passée en

paramètre permet de diviser l'espace d'entrées/sorties et mémoire,), device select (partie de l'adresse utilisée pour activer un des périphériques; ces bits sont des entrées du module d'appariement) et device control (pour connecter les ports gardés des périphériques qui ne peuvent être connectés dans l'espace donné) (Figure A.4). Lorsqu'un port est une sortie, il faut absolument l'associer à l'espace de données. Une fois les sorties associées, il relie les autres ports. Lorsque l'on a une entrée qui fait partie du même SEQ qu'un port de sortie, on doit l'associer à l'espace d'adresse. Tous les autres ports sont d'abord associés aux ports de données et lorsqu'il manque d'espace, on les associe aux ports d'adresse. De cette façon, on peut augmenter la quantité de bits disponibles pour le device select.

Pour générer le module d'appariement, on identifie le nombre de périphériques et le nombre de bits du device select. On utilise l'encodage one-hot lorsque l'on peut associer un bit par périphérique. Le module d'appariement est alors tout simplement une porte AND. Sinon, on vérifie que l'on a associé un code binaire à chaque périphérique et si le nombre de ports du périphérique est suffisant, on utilise un décodeur comme comparateur d'adresse. Sinon, on utilise l'encodage de Huffman.

On peut alors utiliser moins de bits de device select pour les périphériques qui nécessitent beaucoup de bits de device control. On associe donc un petit poids aux périphériques qui ont plus de device select disponibles pour alors leur accorder un code Huffman plus long. On aura donc un nombre de bits variable pour les espaces device select et device control selon le cas. Lorsque toutes les méthodes ont échoué, on doit utiliser plus d'une instruction d'accès mémoire pour exécuter un SEQ. À ce moment, il faut ajouter des registres et autres logiques de synchronisation.

A.3.6. Synthèse des séquenceurs

On fournit aussi un algorithme de synthèse des séquenceurs. L'utilisation de séquenceurs permet au processeur d'initier le transfert en informant le séquenceur de

faire le travail. Le séquenceur est donc un esclave du processeur en attente d'une initiation de transfert. En entrée au processus de synthèse, on fournit les CFGSW et CFGHW et en sortie, on obtient la description matérielle du séquenceur, la connexion entre le séquenceur et le périphérique et les routines logicielles permettant la communication avec le séquenceur (SEQs). On divise le séquenceur en deux parties : la machine à état pour l'interface avec le processeur et celle pour l'interface avec le périphérique. Ici on suppose qu'un périphérique n'est appelé que par un CFG. Sinon, il faudra grouper les CFGs qui utilisent le même périphérique avant de les implémenter en matériel (si un de ces CFG doit être implémenté en matériel).

Lorsqu'un CFG est en matériel, tous les pilotes qu'ils appellent sont en matériel et donc tous les SEQs utilisés par celui-ci sont en matériel. Donc lorsqu'un périphérique a un de ces SEQs en matériel, alors tous ces SEQs le seront aussi. Pour chaque périphérique on forme un ensemble avec le CFGHW, les points d'entrées de ce CFG, les pilotes et le SEQs que l'on appelle cluster. Pour chaque cluster on génère un séquenceur. On convertit le CFG en une première machine à états qui communiquera avec le périphérique. Ensuite, on devra synthétiser le protocole entre le séquenceur et le processeur pour former une seconde machine à états. On connecte alors les deux machines à états et on met à jour le logiciel pour refléter les changements effectués.

La synthèse du protocole permet de générer de nouveaux SEQs pour permettre la communication avec le séquenceur. Ces nouveaux SEQs sont les nouveaux points d'entrées du CFG_{HW}. On associe à chaque SEQ un code (command). Les paramètres nécessaires au séquenceur sont passés par le port bidirectionnel parameters. Après avoir mis la bonne commande sur le port du séquenceur et envoyé le signal start, on peut commencer à communiquer les paramètres (de la façon que l'on choisit en fonction des contraintes de temps). On indique le terme de l'entrée des paramètres à la machine à états du CFG. Tant que celle-ci n'a pas terminé son traitement, on empêche de nouvelles entrées au séquenceur venant du processeur à l'aide du signal ready. Lorsque le signal ready est actif, le processeur peut lire les paramètres de

retour sur le port du séquenceur. Le problème majeur de la synthèse est de déterminer de quelle façon s'effectuera le transfert de paramètres entre le processeur et le séquenceur. On cherche à minimiser le nombre de ports de processeur utilisés en rencontrant les contraintes de temps. La taille du port parameters doit être de :

$W = \max_{e \in cluster} \lceil \#\text{ de données à transmettre}/\#\text{ de cycle d'horloge pour transmission} \rceil$

ANNEXE B : SYSTÈME SPECSYN

SpecSyn a été conçu par Gajski et collaborateurs. On présente la méthodologie de design de systèmes embarqués proposé par les concepteurs de SpecSyn. Puis, passe à travers les étapes de synthèse des communications : le raffinement des variables, la génération de bus, le raffinement des communications, la résolution de conflits d'accès, la génération de l'interface. Puis, on donne les particularités de la génération d'interface logicielle/matérielle.

B.1. Méthodologie de design de systèmes embarqués

Dans [DGZ98], on décrit une méthodologie générale pour le design d'un système à application dédiée (plus particulièrement des systèmes hétérogènes). Il y a d'abord une description de la fonctionnalité du système (la description d'entrée est écrite en SpecChart, [NVG91]). Puis, on passe au design (synthèse) du système comprenant l'allocation, le partitionnement, l'ordonnancement et le raffinement (ou synthèse des communications). Enfin, on peut faire l'implémentation finale comprenant la compilation et la synthèse des différents modules (voir [GaRa94] pour les étapes de synthèse de haut niveau).

Dès que la description et la simulation comportementale du système sont terminées, le design à proprement dit du système peut débuter. On définit trois sortes d'objets fonctionnels : des variables, des blocs fonctionnels et des canaux. Ces objets servent respectivement à conserver, traiter et transmettre les données. Le partitionnement sert à définir les composants utilisés dans le système (mémoire, bus, ASIC, microprocesseur). L'allocation permettra ensuite d'associer à chaque objet fonctionnel un composant du système. L'allocation fournit une architecture cible du système (Figure B.1). Ensuite, l'ordonnancement donne l'ordre dans lequel les objets fonctionnels sont exécutés sur un même composant (exemple le microprocesseur). L'ordonnancement peut être statique ou dynamique.

Le raffinement correspond à l'ajustement des spécifications pour refléter la transformation des objets fonctionnels en composants du système. Cela équivaut à la synthèse de communications, car on ajoute à la spécification des détails par rapport à la mémoire, les interfaces et les arbitres pour permettre la communication entre les composants du système et générer une description générale au niveau système. On passe donc d'une description fonctionnelle à une description d'implémentation. Dans [GaVa95], on décrit le développement de l'interface comme la génération de bus, la génération de protocoles et la génération d'interfaces pour des protocoles différents. Dans certain cas, l'ajout d'arbitre s'avère nécessaire pour permettre la synchronisation des composants.

Une fois le design terminé, il faut passer à l'implémentation finale. On peut alors synthétiser les descriptions qui doivent être implémentées en matériel et compiler le code des descriptions implémentées en logiciel. De la même façon que pour les modules fonctionnels, l'interface est synthétisée à partir des descriptions obtenues lors de la synthèse des communications. On génère les routines des pilotes et les routines d'interruption pour le logiciel et le circuit d'interface pour le matériel.

À l'étape de synthèse des communications ou raffinement expliqué dans [GVNG94] et dans [KlGa98], on considère qu'après l'allocation, on a une architecture cible du système. On connaît donc les connexions des composants fonctionnels du système, les types de communications utilisés et les composants nécessaires à la communication (par exemple, le nombre de bus). On commence par le raffinement des variables et la génération des bus. Puis, on passe au raffinement des communications. Si nécessaire, il faut aussi résoudre les conflits d'accès. Ensuite, il faut générer les modules d'interfaces. On termine en donnant une méthode spécifique du développement d'interfaces de modules matériels/logiciels.

B.2. Raffinement des variables

Dans la description initiale, toutes les communications se font via des variables partagées. Il faut donc, selon le schéma de communication choisi, associer ces variables au bon espace mémoire. Lorsqu'un groupe de variables est associé à un espace mémoire, il faut faire de la complétion de bits si la grandeur des variables n'est pas la même que celle de l'espace mémoire. Selon la situation, on peut par exemple choisir de limiter le nombre de pilotes de bus et d'augmenter le nombre de multiplexeurs en divisant une variable de 12 bits en 8 bits pour le premier octet et 4 bits sur le deuxième au lieu de faire 6-6.

En plus de faire le remplissage de variables, il faut faire des translations d'adresses. Lorsque l'on travaille avec des variables scalaires, la translation d'adresses est assez simple. Cependant, lorsque l'on travaille avec des tableaux, il faut faire attention de bien changer l'indexation de ceux-ci même lorsqu'une variable sert d'index. Notons que l'on peut assigner une variable à la mémoire locale ou à la mémoire globale selon de type de communication choisie. Si on choisit la mémoire partagée, il faut affecter la variable à la mémoire globale, et si on choisit le passage de messages, on affecte la variable à la mémoire locale.

B.3. Génération de bus

Si l'on veut utiliser le passage de messages, il faut associer les canaux aux bus. En supposant que l'association des canaux ait été effectuée, on peut déterminer la taille du bus à utiliser. Dans [NaGa94], on décrit un algorithme pour déterminer la taille de bus à utiliser pour respecter les contraintes fixées et les caractéristiques connues des canaux et du bus. Un canal est caractérisé par quatre paramètres : importance des données, nombre d'accès au canal, taux moyen de transfert du canal et taux maximal de transfert du canal. L'implémentation d'un bus peut être caractérisée par quatre paramètres : buswidth, délai du protocole, taux moyen de transfert du bus et taux maximal de transfert du bus.

Les canaux d'un même bus peuvent relier des processus différents et peuvent être de taille différente. On suppose cependant qu'un seul canal peut utiliser le bus à tout moment. Pour ce faire, il faut que le taux de transfert moyen du bus soit plus grand que la somme des taux moyens de transfert des canaux. De plus, pour avoir une implémentation efficace, on exige que les taux maximaux du bus soient équivalants au taux moyen de celui-ci. On a donc comme critère d'implémentation que le taux maximal du bus soit plus grand que la somme des taux moyens des canaux.

Notons dans cette méthodologie, on présume que tous les processus communiquent via le même protocole, donc utilisent les mêmes lignes de contrôle. Sur un bus, il y a des lignes de données (le nombre de lignes est dicté par l'algorithme donné précédemment), des lignes de contrôle (pour implémenter le protocole sur le bus) et des lignes d'identification (au nombre de $\log_2(n)$ où n est le nombre de canaux pour identifier quel canal utilise le bus). Les lignes d'identification peuvent être incluses dans l'adresse envoyée sur le bus si les processus récepteurs ont un intervalle d'adresses dédiées. Puisque à cette étape, on ne connaît pas le protocole utilisé, on néglige l'espace du bus occupé par les lignes de contrôle et d'assignation. Cependant, on retarde la détermination de la grandeur du bus pour éviter des approximations.

De plus, on peut fixer d'autres contraintes comme une taille maximale ou minimale pour la taille du bus. On peut aussi fixer une limite (inférieure ou supérieure) au taux moyen ou maximal de transfert des canaux. On pourrait aussi exiger que le taux maximal de chacun des canaux soit plus petit que le taux maximal du bus. Toutes ces contraintes serviront à élaborer une fonction de coût dans l'algorithme. On suppose dans l'algorithme que les processus sont synchrones, donc que les délais et temps sont donnés en nombre entier de cycles d'horloge. De plus, on suppose que le processus récepteur est toujours prêt à recevoir un message et que donc qu'il n'y a aucun délai de synchronisation.

Voici les variables utilisées dans l'algorithme :

Bits(c) = nombre de bits à transférer sur le canal à chaque accès
Access(p,c) = nombre d'accès au canal par le processus durant son exécution
Width(b) = grandeur du bus sélectionné
Protdelay = délai de transmission sur le bus durant
Commtime(p) = temps de communication du processus p
= access(p,c) X ($\lceil$ bits(c) /width(b) $\rceil$) X protdelay(b)
Le deuxième terme est nécessaire lorsque la taille du bus est inférieure au nombre de bits à transmettre.

Comptime(p) = temps d'exécution du processus p

L'algorithme fixe d'abord une taille maximale (plus grand nombre de bits transférés sur les canaux) et une taille minimale (de 1). Ensuite, pour chacune des tailles de cet intervalle, on trouve le taux de transfert maximal du bus et la somme des taux moyens de transfert des canaux. Si on satisfait l'inéquation donnée plus haut, on a une solution possible et on calcule son coût, sinon on rejette cette taille de bus. Après avoir parcouru tous les cas, on choisit la solution la moins coûteuse. La fonction de coût est donnée par la somme des écarts aux contraintes au carré multipliée par le poids associé à la valeur pondérée de ces contraintes. Par exemple, si on fixe une taille maximale de bus, la fonction de coût serait équivalente au poids associé à la taille du bus multiplié par le carré de l'écart entre la taille maximale de bus et sa taille réelle (si la taille réelle du bus est plus grande que la contrainte).

B.4. Raffinement des communications

Le raffinement des communications consiste à ajouter à la spécification les détails reliés aux canaux pour permettre une communication bien définie selon le protocole choisi. Une méthodologie est proposée dans [KlGa98]. Cependant, on suppose, encore une fois, que tous les processus utilisent le même protocole de communication. La première étape est de générer (ou sélectionner) le protocole à utiliser. Pour intégrer l'utilisation du protocole aux descriptions fonctionnelles des blocs, on utilise les RPC (remote procedure calls). C'est donc à travers des fonctions send/receive que l'on communique avec les autres processus en respectant le protocole. Ces fonctions doivent être générées à partir des diagrammes temporels annotés de l'interface (ou du

protocole choisi). On identifie les événements (changement d'états sur un signal) et leur lien de causalité. Cela permet ensuite de générer une description comportementale du protocole sous forme de fonctions qui seront appelées par les processus communiquants.

L'étape suivante est l'insertion des appels de fonctions dans les descriptions des blocs. Dans [KlGa98] on propose un algorithme pour exécuter cette tâche qu'on appelle channel insertion. On remplace chaque utilisation des variables globales de la description initiale par l'utilisation de variables locales. Ensuite, on insère les fonctions send/receive pour chaque appel à une variable globale. Puis il faut ajouter les ports nécessaires à la communication et des boucles lorsque les grandeurs des données sont différentes d'un processus à l'autre Lorsqu'un bloc communique avec une mémoire et non un autre composant, on peut ajouter un processus permettant d'accepter le transfert de données pour rendre la description simulable.

B.5. Résolution des conflits d'accès

L'utilisation d'arbitre est parfois nécessaire pour permettre de résoudre les conflits d'accès à un média commun. On utilise les signaux request et grant pour réaliser le handshake entre l'arbitre et les processus. On fournit un algorithme pour l'insertion d'un arbitre dans une spécification. Dans l'algorithme, on suppose que la préemption de l'arrangement n'est pas possible et que la priorité est fixe. On ajoute d'abord les signaux nécessaires au handshake dans la description même des processus. Ensuite, on génère le code pour l'arbitre. Lorsque l'arbitre reçoit une demande (ou plusieurs demandes), il prend les processus qui veulent le bus dans l'ordre de la liste de priorité et il leur lègue à tour de rôle le bus (Tableau B.1).

B.6. Génération d'interface

Si au moins un des composants n'est pas encore synthétisé (exemple : ASICs), on peut inclure la logique nécessaire aux communications dans la description du

composant non-synthétisé. Tel que décrit lors de l'étape du raffinement des communications, on complète la description par l'ajout d'appels aux fonctions send/receive ainsi que de descriptions de ces fonctions. Cependant, lorsque l'on est en présence de deux composants ayant des interfaces fixes et incompatibles, il faut générer un module permettant de joindre ces deux composants. Dans [NaGa95] on fournit une méthode pour faire la génération de ce module permettant de répondre aux signaux de contrôle des deux protocoles et de transférer les données. On fait donc une translation de protocole.

Tableau B.1. Code pour la résolution de conflit d'accès

Code dans un processus [illegible]	Code pour l'arbitre

Précéder tous les accès à R dans Bi par
Req_i <= 1
Wait until (grant_i = 1)
Ajouter la ligne suivant après tous les accès à R dans Bi
Req_i <=0
	if Req_k = 1 **then**
	Grant_k <= 1
	Wait until (Req_k = 0)
	Grant_k <= 0
	end if

On définit un protocole comme un ensemble d'opérations atomiques et on décrit cinq opérations atomiques (Tableau B.2). On décrit les protocoles de trois façons différentes : diagramme de temps, machines à états ou langage de description matérielle. On montre comment générer la description matérielle de l'interface à partir de celle des protocoles des deux blocs à relier. Le Tableau B.3 résume les étapes de l'algorithme permettant de générer l'interface.

Tableau B.2. Opérations atomiques et leurs opérations duales

[illegible]	[illegible] (HD[illegible])	Opération [illegible]
Attente d'un événement sur une entrée	Wait until (ControleDuBus = 1)	ControleDuBus <= 1
Assignation d'une valeur à une sortie (ligne de contrôle)	ControleDuBus <= 1	Wait until (ControleDuBus = 1)
Assignation d'une valeur à une sortie (ligne de données)	DonnéesDuBus <=Variable	TempVariable <=DonnéesDuBus
Lecture d'une valeur sur une entrée de données	Variable <=DonnéesDuBus	DonnéesDuBus <=TempVariable
Attente pendant un intervalle fixe	Wait for 100ns	Wait for 100ns

Tableau B.3. Étapes de génération d'une interface

1	Représenter les deux protocoles comme une suite d'étapes formées par des opérations atomiques
2	Partitionner les étapes en blocs d'étapes et associer les blocs d'étapes des deux protocoles pour former des groupes
3	Ordonnancer les blocs à l'intérieur d'un groupe et ordonnancer les groupes
4	Former un code HDL à partir de toutes les opérations atomiques des blocs et groupes ordonnancés
5	Générer l'interface en trouvant les opérations duales correspondant aux opérations atomiques
6	Optimiser les interconnexions en éliminant le passage à travers l'interface lorsque inutile (lorsque les données sont lues d'un côté et écrites de l'autre sans état d'attente entre les deux ou lorsque deux lignes de contrôle ont simultanément la même valeur)

B.7. Raffinement d'interfaces logiciel/matériel

B.7.1. Modèle d'implémentation

Le raffinement d'interface logiciel/matériel est un cas particulier du raffinement d'interface détaillé plus haut. Les différences dans la méthodologie sont expliquées dans [GGB97] et dans [GVNG94]. Dans l'architecture du système hétérogène, on utilise un ou plusieurs ASICs, un ou plusieurs bus, une ou plusieurs mémoires (locales ou globales), un processeur et quelquefois un arbitre. On propose quatre modèles d'implémentation (Figure B.1):

- Modèle 1 : une mémoire globale à un seul port pour chaque composant (ASICs ou processeur) et un seul bus pour les mémoires globales
- Modèle 2 : une mémoire locale, une mémoire globale à un seul port pour chaque composant, un bus pour relier chaque composant avec sa mémoire locale et un bus pour les mémoires globales
- Modèle 3 : une mémoire locale, une mémoire globale à doubles ports pour chaque composant, un bus pour relier chaque composant avec sa mémoire locale et un bus par port de mémoire globale
- Modèle 4 : une mémoire locale par composant, un module d'interface par composant, un bus par mémoire, un bus par composant et un bus global.

B.7.2. Distribution des variables

Une fois l'allocation et le partitionnement terminés, on doit faire le raffinement des variables et la génération des bus. Dans le cas d'un système hétérogène, il faut distribuer les variables entre le matériel (registre) et le logiciel (mémoire). On décide donc dans quel espace mémoire (mémoire locale ou globale de quel composant) on enregistre chacune des variables. Si on assigne une variable très utilisée par un module matériel à la mémoire du processeur, on augmente la circulation sur le bus global et pour maintenir un bon taux de transfert, il faut augmenter la taille du bus. Cependant, si on assigne une variable souvent utilisée par le logiciel à un registre matériel, on augmente le coût du matériel.

Pour trouver un compromis entre la distribution des variables et la taille du bus utilisé, on propose un algorithme qui permet de satisfaire le taux de transfert maximal du bus en minimisant le coût d'accès aux variables et le coût du bus. L'algorithme présume d'une implémentation de type deux. Cet algorithme évalue toutes les combinaisons de tailles de bus et de distributions de variables possibles en respectant le taux de transfert ainsi que leur coût. On peut alors choisir la combinaison la moins coûteuse.

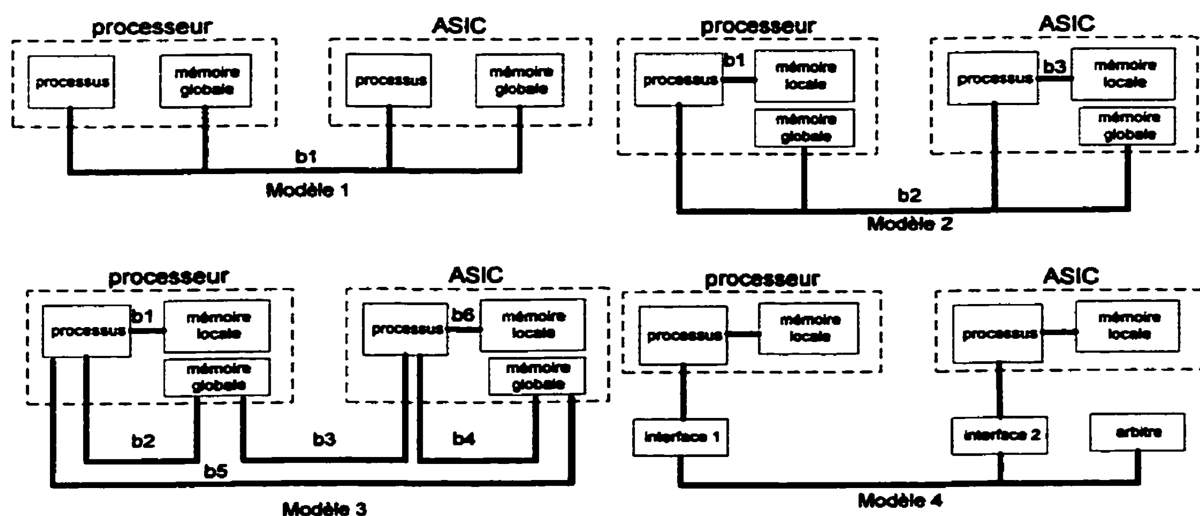


Figure B.1. Modèles d'implémentation

B.7.3. Raffinement de l'accès aux données

Dans le cas logiciel/matériel, on divise le raffinement des communications en trois étapes : le raffinement de l'accès aux données, le raffinement des instructions de contrôle et le raffinement de l'architecture. En effet, une fois que l'on a défini le protocole utilisé, il faut ajuster la spécification en fonction du modèle de communication utilisé.

Expliquons le raffinement de l'accès aux données. On donne quatre types d'accès aux données : accès du matériel et du logiciel à la mémoire, accès du matériel à la banque de registre et accès du logiciel à la banque de registre et aux ports. (L'accès aux ports du matériel ne nécessite pas de raffinement). Pour faire ce raffinement, il faut associer aux variables des adresses dans l'espace global disponibles sur le bus et faire les translations d'adresses. Comme dans le cas général, on propose l'utilisation de RPC pour faciliter l'établissement des fonctions de communication. Dans le cas du logiciel, il faut générer les bonnes routines pour utiliser les ports du processeur et accéder le bus. En effet, l'accès aux ports n'est pas aussi direct que dans le cas matériel ce qui complexifie la tâche du raffinement.

B.7.4. Raffinement des canaux de contrôle

On doit ensuite raffiner les canaux de contrôle. Le but de cette étape est de préserver la séquence d'exécution même si des processus subséquents sont implémentés sur des composants différents. Par exemple, admettons trois processus en séquence (A, B, C) dont le processus B est implémenté en matériel et les deux autres en logiciel. Pour respecter l'ordre d'exécution, on propose d'insérer un nouveau processus NEW_B à l'endroit où le processus B devrait être dans la séquence. Dans ce nouveau processus, on met une instruction pour indiquer au processus B qu'il peut commencer son exécution (signal start) et une instruction pour attendre la fin de l'exécution du processus B (signal done). On propose aussi l'utilisation des interruptions logicielles lorsque l'un des composants est un processeur. En effet, le processeur peut alors

continuer l'exécution des processus sans lien avec B en même temps que l'exécution de B et se faire interrompre seulement lorsque B a terminé.

B.7.5. Raffinement de l'architecture

L'étape du raffinement de l'architecture est définie dans [GGB97] comme la génération d'une interface et la génération d'un arbitre. C'est donc l'équivalent des étapes expliquées aux sections B.4 et B.5 ci-haut. La génération d'une interface est cependant seulement nécessaire lorsque l'implémentation utilisée est de type 4. De plus l'interface développée sert à relier le bus et un bloc et non deux blocs entre eux. On suggère un algorithme ayant les mêmes étapes que dans la section B.6 avec une nuance. L'interface utilise le même protocole que le bus, mais un protocole qui s'adapte au protocole du bloc. Lorsque l'on doit s'adapter, on utilise les opérations duales sinon, on garde les même opérations.

ANNEXE C : SYSTÈME COOL

Niemann et collaborateurs proposent une méthodologie de développement d'interface de communication pour une architecture générale définie. Cette méthodologie est utilisée dans l'outil COOL. On présente donc l'algorithme général de développement d'interface de communication. Puis on donne l'architecture type utilisée pour la synthèse. On explique ensuite le raffinement de la logique de contrôle et du chemin de donné. On termine avec la synthèse de l'interface.

C.1. Algorithme général de développement d'interface de communication

Dans [NiMa98], on divise le développement d'interface de communication en deux étapes : la synthèse des communications (sélection d'un protocole de communication, allocation de la mémoire, choix de la topologie de bus) et le raffinement de l'interface (transformation d'un canal de communication abstrait en une interface physique réelle incluant l'ajout de fonctions permettant les échanges de données).

C.1.1. Raffinement de la logique de contrôle

La tâche du raffinement se divise en trois étapes, le raffinement de la logique de contrôle, le raffinement du chemin de données et la synthèse de l'interface. Le raffinement de la logique de contrôle consiste en l'insertion de mécanismes de synchronisation. Pour la synchronisation on utilise la mémoire partagée, le passage de messages ainsi que les interruptions logicielles.

C.1.2. Raffinement du chemin de données

L'objectif du raffinement du chemin de données est de transformer les accès de données abstraits du système en des accès mémoire utilisant le protocole choisi. On doit d'abord faire une conversion des types des données utilisées par la spécification indépendante de l'implémentation en des types de données compatibles avec

l'architecture choisie. Il faut ensuite remplacer les opérations effectuées sur les données en leurs équivalents dans l'architecture choisie. Puis, on doit associer chacun des bits d'une variable à un mot mémoire. On peut utiliser le little-endian ou le big-endian. En dernier lieu, il faut ajuster les adresses. On transforme alors chacune des références à une variable en un accès mémoire à la bonne adresse.

C.1.3. Synthèse de l'interface

La synthèse de l'interface correspond à la réalisation de la communication. On doit alors utiliser différents modèles de communication (passage de messages ou mémoire partagée), différents canaux (ligne dédiée, bus, FIFO ou mémoire partagée) et différents protocoles. Les tâches de la synthèse de l'interface sont les suivantes :

- Génération du bus : choix de la taille du bus pour satisfaire les contraintes
- Génération du protocole : insertion de fils pour les données, les adresses et les signaux de contrôle et définition des fonctions supplémentaires ajoutées pour le traitement des données selon la taille des bus
- Génération d'un arbitre : gestion de conflits d'accès aux ressources (comme bus) ; on peut l'implémenter avec une priorité fixe ou avec une priorité dynamique
- Raffinement d'interfaces incompatibles : ajout de logique entre deux interfaces pour faire la conversion de protocoles fixes incompatibles.

C.2. Synthèse de communications dans COOL : architecture type

C.2.1. Description du modèle général

Le système COOL permet de faire du développement d'interface de communication. La synthèse des communications (choix de l'architecture, des protocoles, des grandeurs de bus,...) n'est pas une tâche très compliquée puisque l'architecture est définie d'avance. L'architecture type est très générale et peut être utilisée dans plusieurs systèmes. Elle contient un contrôleur système, un contrôleur d'entrées/sorties, un arbitre de bus, les modules matériels exécutant les fonctions

déterminées, des contrôleurs de modules, des pilotes de bus, de la mémoire (locale ou non) et un microprocesseur. Notons que toute la logique de contrôle est exécutée dans les contrôleurs. Les blocs matériels se résument donc à la logique du chemin de données (*datapath*). Sur le schéma de la figure 5.1, les lignes simples représentent les signaux de contrôle bidirectionnels et les flèches représentent les données. Notons que la mémoire et le microprocesseur sont des éléments fixes dans cette architecture.

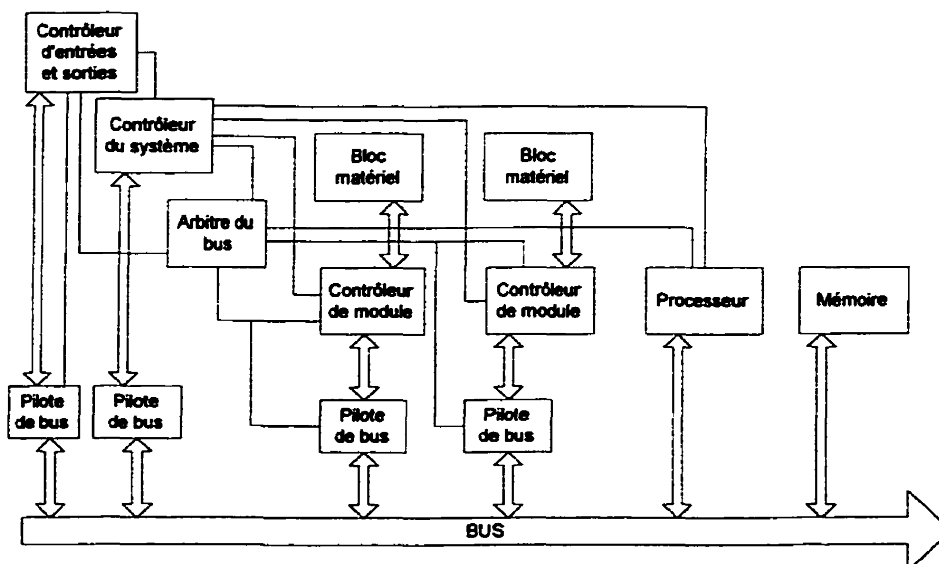


Figure C.1. Architecture utilisée dans l'outil COOL

C.2.2. Contrôleur système

Le contrôleur système correspond en réalité au module permettant la synchronisation entre les différents modules (run-time scheduler). Les moyens que peut utiliser le contrôleur pour faire la synchronisation ont été décrits dans la section 1.2. Lorsque le contrôleur veut communiquer avec un module logiciel on propose deux méthodes : la synchronisation par interruptions et mémoire partagée ou par interruption et passage de messages. Lorsqu'il veut communiquer avec un module matériel, on offre trois méthodes : la synchronisation par passage de messages et mémoire partagée, par passage de messages en utilisant un signal start ou par passage de messages en utilisant un signal état.

C.2.3. Contrôleur d'entrées/sorties

Le contrôleur d'entrée/sorties sert d'interface entre le système et l'extérieur. Il est donc un intermédiaire entre la mémoire et l'extérieur mais est contrôlé par le contrôleur système (qui indique quand lire les données sur le bus de sortie et quand écrire les données sur le bus d'entrée). Il sert donc de handler des entrées et de handler des sorties. Le handler d'entrées garde la valeur d'une nouvelle entrée dans un registre jusqu'à ce que le contrôleur système lui indique qu'il peut l'inscrire en mémoire. On utilise un protocole de handshake pour permettre la communication entre les deux contrôleurs. Le handler de sorties espionne le bus d'adresse de la mémoire jusqu'à ce que l'on effectue une écriture à une adresse de sortie puis il transfère cette valeur sur le bus de sortie.

C.2.4. Arbitre

Étant donné que le contrôleur d'entrées/sorties, le microprocesseur et les autres modules matériel peuvent tous vouloir accéder au bus de la mémoire en même temps, il faut ajouter un arbitre de bus. L'arbitre communique avec les différents composants à l'aide des signaux request et acknowledge.

C.2.5. Contrôleur de module

Le contrôleur de modules est nécessaire pour permettre aux modules implémentés en matériel de communiquer avec les autres composants. Il y a un contrôleur par module qui sert d'interface entre le contrôleur système, l'arbitre de bus et le module matériel exécutant une fonction. On essaie normalement de ne pas avoir de contrôleur module (on évite l'ajout de matériel) pour le microprocesseur. On utilise, en plus du contrôleur de modules, un pilote de bus.

C.2.6. Pilote de bus

Normalement, un pilote logiciel est une routine logicielle permettant l'utilisation des ports d'un processeur pour les communications avec l'extérieur. Ici, on emploie le

terme pilote de bus pour signifier un module permettant la connexion entre les modules matériels et le bus. Puisque le pilote de bus et le contrôleur de modules sont directement liés et sont tous deux en matériel, on pourrait n'en faire qu'un module. Ici on les sépare car l'algorithme de synthèse est plus simple en considérant deux modules séparés qui communiquent.

C.2.7. Détermination de la quantité mémoire

Il faut maintenant déterminer la quantité de mémoire nécessaire aux communications. D'abord, si on utilise la mémoire partagée comme moyen de synchronisation entre le contrôleur et le processeur, il faut calculer l'espace mémoire nécessaire (nombre de blocs mémoire nécessaires = $\log_2$(nombre d'états)/(taille d'un bloc mémoire) en arrondissant vers le haut). Ensuite, il faut mesurer la mémoire nécessaire pour chacune des entrées/sorties du système (nombre de bits de l'entrée ou de la sortie/nombre de bits d'une case mémoire en arrondissant vers le haut). En dernier lieu, il faut déterminer la mémoire nécessaire pour permettre le passage de données entre les modules exécutés sur des ressources différentes. Ici on calcule la quantité de mémoire (de la même façon que pour les entrées/sorties) pour chaque échange de données. Cependant, il est probablement possible de faire une économie de mémoire en regroupant le plus de transferts de données exécutés à des moments différents sur un même espace mémoire.

C.3. Raffinement de la logique de contrôle : conception du contrôleur système

C.3.1. Modèle read/write

Pour permettre les communications entre les modules, on utilise un modèle qui divise l'étape de la communication en deux sous étapes : read et write. Dans le cas de communication bloquante, le write de l'émetteur doit être exécuté en même temps que le read du récepteur. Dans le cas non-bloquant, on n'a aucune restriction sur le temps d'exécution mais seulement sur l'ordre d'exécution (le write de l'émetteur doit être

exécuté avant ou en même temps que le read du récepteur). Ce genre de modèle impose, lors de l'ordonnancement, que l'exécution des étapes de communication ne soit pas interrompue et que le canal de communication ne soit accédé que par un module à la fois.

C.3.2. Génération de la machine à états

Nous savons que la synchronisation se fait en divisant les étapes d'exécution et de communication en différents états. Selon l'état dans lequel ils sont, les modules exécutent des tâches différentes. Les états peuvent être gérés par le contrôleur système ou par les contrôleurs de modules. D'après ce modèle de communication, on propose une méthode pour générer la machine à états nécessaire à l'implémentation du contrôleur de système (et du contrôleur de modules si c'est le module matériel qui gère les changements d'états). Selon le graphique obtenu après le partitionnement, dont chaque nœud représente une opération à être exécutée en logiciel ou en matériel, les liens représentent les dépendances entre les nœuds. Dans ce graphique, on ajoute des nœuds pour les étapes read et write avant et après chaque nœud d'exécution lorsque nécessaire. Le fait d'ajouter ces nœuds permet d'assurer qu'il n'y a pas de conflit de ressources (entre autres pour le bus).

À partir de ce graphique partitionné et du modèle de communication utilisé, on peut faire l'ordonnancement des nœuds pour chacune des ressources et produire un graphe de temps. Ce graphe de temps permet d'assurer que pour toutes les ressources, le canal de communication ne sera jamais utilisé en même temps (assurant qu'aucun read-write ne s'exécutera en même temps). On transforme alors chaque nœud de lecture en nœud d'attente et chaque nœud d'écriture en nœud de terminaison. Et on ajoute des attente-terminaison lorsqu'il n'y avait pas de read-write. On crée ensuite une ligne d'exécution par ressource (modules matériels et processeur) et on y met les nœuds dans l'ordre déterminé par le graphique de temps. On y représente alors les dépendances de ressources (et de données pour des modules sur une même ressource)

en plaçant dans l'ordre les opérations sur une ressource. De plus, on y ajoute des liens entre les lignes d'exécution des ressources pour y représenter les dépendances de données entre les modules de différentes ressources.

Ensuite, il est possible d'optimiser le graphique d'états obtenu. On peut d'abord éliminer les transitions redondantes, par exemple, une dépendance de données entre deux ressources qui est compensée par une dépendance de ressources. Puis, on peut éliminer les états redondants, par exemple l'état done et l'état wait de deux opérations qui se suivent dans une même ressource peuvent être éliminés. Il faut respecter trois règles lors de l'optimisation des états. Un état wait est nécessaire seulement lorsque les données attendues viennent d'une ressource différente. Un état done est nécessaire seulement lorsque les données produites sont utilisées par des modules exécutés sur des ressources différentes. Deux états execute consécutifs sans états done ou wait entre les deux peuvent être regroupés.

Notons que dans le cas où les changements d'états ne se feraient pas par le contrôleur d'états mais plutôt par le contrôleur de chaque module, le graphique obtenu doit être modifié pour obtenir les machines à états des divers contrôleurs.

C.4. Raffinement du chemin de données et synthèse de l'interface

Une fois le raffinement du chemin de données effectué, il faut générer des programmes qui assurent la compatibilité du microprocesseur avec l'interface de communication choisie : ajouter les fonctions read/write, insérer une routine d'initialisation du processeur et des routines d'interruptions, permettre l'accès aux ports du processeur nécessaire à la synchronisation, ... Ensuite, il faut faire le raffinement des modules, par exemple ajouter les signaux nécessaires dans les modules pour la communication avec les contrôleurs et les mémoires locales. Ensuite, on peut faire la génération du contrôleur système et des contrôleurs de modules (on utilise les résultats obtenus lors de la phase du raffinement de la logique

de contrôle pour générer le code VHDL des contrôleurs). Notons que l'implémentation matérielle de l'arbitre et du contrôleur d'entrées/sorties n'est pas préoccupante car ces deux modules sont fixes pour une architecture choisie (exemple : type d'arbitrage).

ANNEXE D : SYSTÈME COSMOS

Jerraya et collaborateurs ont formé un groupe travaillant sur le développement de méthodologies pour la spécification, le design et la synthèse de systèmes hétérogènes. Ils ont développé le logiciel COSMOS permettant de faire du codesign à partir d'une description comportementale en SDL (System Description Language) du système. On présente une description du langage intermédiaire utilisé puis on décrit la méthodologie de synthèse dans COSMOS.

D.1. Description du langage SOLAR

Pour faire la synthèse, ils utilisent une représentation intermédiaire pour faire le raffinement au niveau système. Cette représentation (ou modèle), nommée SOLAR, a aussi été développée et décrite par le groupe [ObJe92]. Chaque étape du raffinement est faite sur le modèle SOLAR du système. L'utilisation d'un format intermédiaire pour la synthèse donne l'avantage de rendre le système indépendant du langage de description utilisée.

SOLAR supporte le concept de communication de haut niveau (canaux abstraits et variables globales) permettant d'utiliser plusieurs modèles de communication (passage de messages, mémoire partagée). SOLAR permet l'utilisation des concepts de machine à états étendus (EFMS) pour la description comportementale et les RPC (remote procedure calls) pour la communication de haut niveau.

SOLAR contient trois éléments de base : les DesignUnits (DUs), les ChannelUnits (CUs) et les StateTable (STs). La représentation au niveau système est constituée de DUs (Figure D.1). Les DUs peuvent contenir d'autres DUs, des STs et des CUs. Les machines à états sont modélisées par les STs et peuvent être exécutées en série ou en parallèle. Les communications entre les processus sont exécutées à travers les CUs. On utilise les RPC pour modéliser les communications à travers les CUs. Cette

architecture permet de séparer les communications des descriptions comportementales.

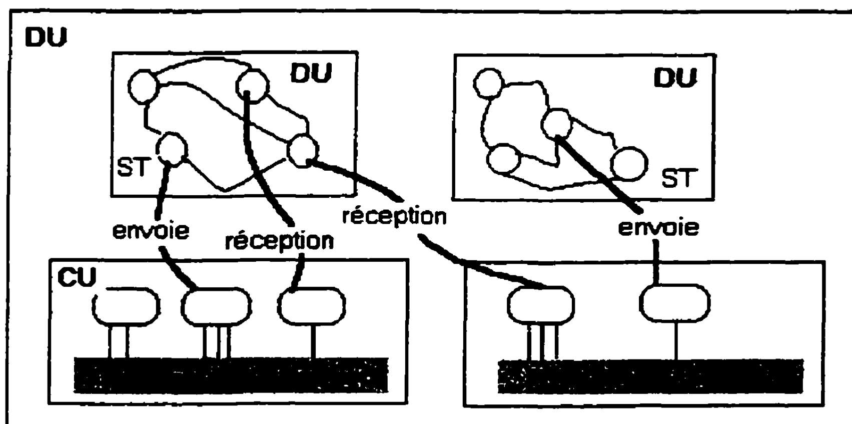


Figure D.1. Représentation d'un système en SOLAR

L'accès aux canaux de communications est contrôlé par un ensemble de procédures appelées Methodes (ou services). Les Methodes constituent la partie visible du canal de communication. Les DUs utilisent des appels aux Methodes pour communiquer selon le type de communication et le protocole utilisé. Les paramètres de la communication sont donc incorporés dans les CUs sans que le concepteur n'ait à s'en préoccuper. Par exemple, si on veut utiliser une communication synchrone par rendez-vous, on conçoit des procédures put et get. Les DUs feront donc un appel RPC à une de ces procédures pour communiquer. Le concepteur n'aura qu'à utiliser ces fonctions pour établir une communication. Si, au lieu d'avoir une communication synchrone, on veut une communication asynchrone, on devra ajouter des queues aux canaux de communication et modifier les procédures en conséquence.

Dans [VCV+95] on parle de CommunicationUnits au lieu de ChannelUnits. La différence entre les unités de communication et les canaux est seulement le niveau d'abstraction ou le niveau d'implémentation. Les canaux sont plus abstraits et, à l'opposé des unités de communication, ils ne renferment aucune information sur

l'implémentation future. Les canaux comprennent seulement les procédures d'accès, tandis que les unités de communication peuvent aussi contenir des composants nécessaires au passage de données (comme les tampons, les multiplexeurs, les FIFO,...) et des contrôleurs de communication. Si on diminue encore le niveau d'abstraction, les unités de communication deviendront l'implémentation finale de la communication. SOLAR permet donc une représentation du système à différents niveaux d'abstraction ce qui est très utile lors du raffinement des communications.

D.2. Synthèse des communications dans COSMOS

D.2.1. Description générale de l'outil

COSMOS débute avec une description du système en SDL et produit un modèle du système sous forme C-VHDL associé à une architecture logicielle/matérielle (déterminée au départ). Dans COSMOS, on décompose la tâche du codesign en quatre étapes : le partitionnement du système en processus, l'allocation des processus aux composants virtuels (qui peuvent être logiciel ou matériel), le raffinement des communications (synthèse des communications), l'implémentation finale (génération de code C et de code VHDL).

La tâche de raffinement se fait en ajoutant progressivement des détails sur l'implémentation et c'est le concepteur qui contrôle cette tâche à travers des primitives de transformations. Dans [DMVJ97] et dans [DIMJ97], on aborde la synthèse des communications comme un problème d'allocation de ressources. L'approche proposée a deux limitations. Premièrement, le concepteur doit fournir une bibliothèque d'unités de communication et une autre d'implémentation de ces unités. Il n'est pas possible d'utiliser une unité qui n'est pas présente dans la bibliothèque. Deuxièmement, le concepteur a besoin d'une fonction de coûts réaliste pour pouvoir estimer les communications.

Dans [VIJK94] on donne trois étapes pour la synthèse des communications : le choix du protocole, le channel binding (ou transformation des canaux) et le channel mapping. Dans [DMVJ97] on parle de sélection de protocole, d'allocation des unités de communication (CommunicationUnit) et de synthèse de l'interface. Nous allons maintenant explorer l'étape du raffinement des communications.

D.2.2. Sélection du protocole et allocation des unités de communication

Tel qu'expliqué ci-haut, tout le raffinement se fait à partir de la représentation SOLAR. Pour diminuer le niveau d'abstraction et arriver à une implémentation physique, on doit d'abord sélectionner un protocole puis transformer les canaux abstraits en unités de communication, c'est le channel binding ou allocation des unités de communication.

Les unités de communication peuvent contenir plusieurs canaux abstraits et donc peuvent exécuter plusieurs services. La partage d'une unité de communication par différents canaux permet un partage des ressources (arbitre de bus, FIFO, bus,...). Un contrôleur (exemple multiplexeur) y est souvent inclus pour permettre la synchronisation des communications. Le choix de l'unité utilisée par un canal abstrait (et ses services) dépend du protocole utilisé, du type de communication (passage de messages ou mémoire partagée) ou des performances requises. De plus, une unité de communication peut être implémentée de différentes façons selon l'architecture générale choisie (voir étape de synthèse de l'interface).

Pour permettre l'allocation des canaux de communications, on doit avoir un système partitionné contenant des DUs communiquant via des canaux (Figure D.2). De plus, il faut avoir une bibliothèque d'unités de communication. Les unités de communication contiennent de l'information sur le type de communication et le protocole utilisé, mais n'enferment aucune information sur l'implémentation en tant quel tel. En effet, on utilisera une seconde bibliothèque lors de la synthèse de

l'interface pour déterminer l'implémentation exacte des unités. Une fonction de coût ainsi que des contraintes sont aussi nécessaires dans l'algorithme d'allocation. En sortie, on aura un système où les DUs communiquent via les CUs et non plus via les canaux abstraits. Il restera ensuite l'étape de la synthèse de l'interface à réaliser avant d'arriver à une implémentation finale.

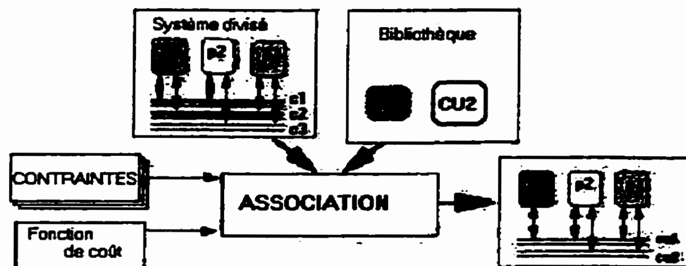


Figure D.2. Allocation des canaux dans la synthèse des communications

On donne un algorithme pour réaliser l'allocation. On construit d'abord un arbre dont les nœuds correspondent aux canaux abstraits du système et les branches correspondent aux unités qu'il est possible d'utiliser pour implémenter le canal du nœud précédant la branche. Les feuilles de l'arbre sont donc vides. Chaque chemin dans l'arbre correspond à une allocation possible pour le système. Pour déterminer si un type d'unité peut être associé à un canal (donc pour connaître les branches de l'arbre), il faut satisfaire trois conditions : le protocole du canal (Protocol(AbstractChannel)) doit être disponible dans l'unité, les services utilisés par le canal (Services(AbstractChannel)) doivent être implémentés dans l'unité et le taux de transmission maximal de l'unité (MaxbusRate(CU)) doit être plus grand que le taux moyen de transmission du canal (AveRate(AbstractChannel)).

Les canaux abstraits sont donc définis par leur protocole, leurs services et leurs taux maximal et moyen de transfert. Les unités sont définies par leur coût (fixé par le concepteur selon les critères de performance et les contraintes à respecter), leurs protocoles, leurs services, leur taux maximal de transfert et le nombre de canaux

abstraits qu'ils peuvent contenir (MaxNumberAC(CU)). On peut vouloir mettre plus qu'un canal dans une unité. Dans ce cas, il faut que le taux maximal de transfert de l'unité soit plus petit que la somme des taux moyens de tous les canaux qu'elle contient. On ajoute, en plus un critère de performance (vitesse de transfert) suivant lequel le taux maximal de transfert de l'unité doit être plus grand que la somme des taux maxima des canaux qu'elle contient. Le coût d'une solution est alors la somme équilibrée des coûts des unités et des coûts associés à la vitesse de transfert.

L'algorithme fait l'allocation des unités de communications. On construit d'abord l'arbre décrit ci-haut. Puis on parcourt tous les chemins de l'arbre pour déterminer la solution la moins coûteuse. Une fonction récursive parcourt l'arbre en allouant les unités en cours de route. Même si l'arbre fournit des informations sur le type d'unités à utiliser, on doit déterminer la quantité d'unités les canaux qui partageront les mêmes unités. Lorsque l'on atteint une feuille, l'allocation pour un des chemins est terminée et il faut vérifier le coût de la solution trouvée.

À chaque branche de l'arbre, on vérifie s'il n'est pas possible d'utiliser une unité qui a déjà été allouée avant d'en allouer une autre (voir conditions données plus haut). On fait une association si c'est possible et sinon on crée une nouvelle instance de l'unité. À chaque allocation, on ajuste la liste des unités allouées, les paramètres de l'unité allouée (exemple MaxNumberAC) et la fonction de coût de la solution explorée. Si une solution n'est pas possible, on lui alloue un coût infini.

D.2.3. Synthèse d'interface

Dès que l'on a une représentation du système en SOLAR avec des unités de communication (CU) et des unités de design (DU), on peut passer à la synthèse de l'interface (channel mapping). Dans [HCM+99] on explique qu'il faut alors déterminer l'implémentation de chaque CU (Communication Unit) en la choisissant dans une seconde bibliothèque. En effet, chaque unité de communication peut être

implémentée de plusieurs façons selon les contraintes de taux de transfert, la capacité des mémoires, le nombre de lignes de contrôle et de données, les protocoles des composants avec des interfaces fixes,... Cela permet de générer une interface pour chacun de DUs du système. Une fois le canal réel sélectionné dans la bibliothèque, on incorpore à la description des DUs les Methodes qu'ils utilisent et on raffine cette description en fonction du canal choisi.

ANNEXE E : SYSTÈME VULCAN

VULCAN est un outil de synthèse de système hétérogène conçu par Gupta et collaborateurs. On présente la description générale de la méthodologie de synthèse, les techniques de synchronisation dans un système *multi-threads* et un algorithme de synthèse des communications entre deux interfaces incompatibles.

E.1. Description générale de la méthodologie de synthèse

E.1.1. Architecture cible

L'architecture typique du système comprend un ou plusieurs composants matériels (ASICs), un processeur ainsi qu'une mémoire globale (Figure E.1). Les composants communiquent entre eux via un bus par mémoire partagée. La mémoire est divisée en trois parties : la mémoire programme, la mémoire donnée et la mémoire d'interface (utilisée pour permettre les communications entre les composants). Notons qu'une des restrictions de l'architecture est que la mémoire du processeur n'est pas relié directement au processeur, mais via le bus global du système. On a donc une seule mémoire et on suppose que le programme est suffisamment petit pour éviter l'utilisation de mémoire virtuelle.

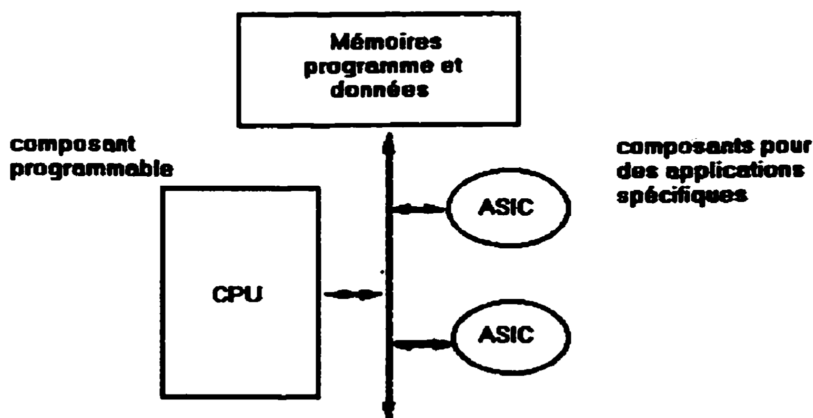


Figure E.1. Architecture cible de l'outil VULCAN

En plus que d'avoir qu'une mémoire externe et qu'un processeur, on exige que le processeur soit le maître du bus. Un avantage de cette approche est que l'on permet une programmation à plusieurs *thread*. Mis à part la mémoire et le bus qui sont contrôlés par le processeur, on suppose qu'aucune synchronisation n'est nécessaire pour le partage des ressources ou de modules. Les composants matériels ont une remise à zéro globale commune. On suppose que le processeur a un nombre d'interruptions et de routines associées. Toutes les communications entre les composants se font à travers le bus (le schéma de la communication est choisi par le concepteur).

E.1.2. Procédure de synthèse de l'outil

Dans [GCM92], on décrit la méthodologie générale de synthèse. L'entrée au processus de synthèse est une description en HardwareC du système. À partir ce cette description, on génère un modèle du système sous forme de graphe à l'aide du programme HERCULE. Le système consiste alors en une série de nœuds représentant les opérations et une série de branches montrant les dépendances de données et de contrôle. On associe à chaque branche correspondant à une entrée ou une sortie de données un taux de transfert maximal. Ensuite, le programme VULCAN effectue le partitionnement.

Dans [GuMi97], on décrit l'analyse de contraintes de délai et de contraintes de taux d'exécution utilisé pour faire le partitionnement. On donne une méthode pour déterminer l'existence d'un ordonnancement des opérations pour rencontrer les contraintes données. On divise le problème en deux selon la présence d'opérations non-déterministes ou non. On fait un ordonnancement statique lorsqu'il n'y a pas de ND. Autrement, on fait l'analyse de graphe et de la propagation de contraintes à travers les niveaux hiérarchiques permettant de déterminer la latence du graphe.

Une fois le partitionnement terminé, on utilise HEBE pour faire la synthèse des parties matérielles. VULCAN génère la description C du programme *multi-thread* à compiler en assembleur pour un processeur dédié. VULCAN effectue aussi la génération d'interface sous certaines contraintes de temps. La dernière étape est la simulation. Le programme POSEIDON est utilisé pour faire la simulation

E.2. Synchronisation dans un système *multi-thread*

Dans [MiGu93] et [GuMi97], on explique l'approche systématique d'automatisation de la synthèse de systèmes hétérogènes et on met l'emphase sur le design d'interface permettant la synchronisation et la communication entre les processus. Le modèle utilisé groupe les opérations sans dépendance de données dans des *threads* et fixe des nœuds d'ancrage lorsqu'une opération est non-déterministe (dépend des données et donc a un délai indéterminé). Ces nœuds correspondent aux points de synchronisation. L'ordonnancement à l'intérieur des *threads* est statique et l'ordonnancement entre les threads est dynamique.

On met l'accent sur la synchronisation nécessaire pour satisfaire les contraintes d'ordonnancement entre le logiciel et le matériel ainsi qu'entre les *threads* du programme logiciel. On un mentionne d'abord que s'il est possible d'écrire le programme en un seul *thread*, on peut tout simplement utiliser un seul canal de communication car toutes les données sont en série. De plus, les seuls points de synchronisations sont ceux qui permettent de transférer le contrôle entre le logiciel et le matériel.

Lorsque les opérations sont dépendantes des données, il n'est pas toujours évident de trouver un ordonnancement statique pour les opérations. On doit alors traiter la synchronisation dans un système *multi-thread*. On explique d'abord comme faire la synchronisation entre les threads. Lorsque le nombre de threads est petit, il et possible d'utiliser une priorité statique pour l'exécution des *threads*. Cependant, dans

la plupart des cas, ce n'est pas le cas et on propose l'utilisation de FIFO contrôlé pour la synchronisation dans ces cas. Cette FIFO contient un pointeur au prochain *thread* à être exécuté dans la séquence et on modifie dynamiquement les pointeurs dans la FIFO selon la séquence voulue. Cependant, puisque avec ce schéma il est impossible d'interrompre un *thread*, la préemption n'est pas permise. Le transfert de données entre les threads peut être effectué par passage de message (en utilisant des coroutines) ou par mémoire partagée (en gardant des pointeurs aux opérations read-write ou en utilisant les registres globaux du processeur).

En plus de la synchronisation entre les *threads*, il est nécessaire de générer un mécanisme de synchronisation entre les modules matériels et logiciels. On propose l'utilisation de queue avec ou sans signaux de contrôle (pour une communication bloquante ou non-bloquante). La grandeur des queues peut être déterminer par un algorithme donné dans [AmBo91]. Les opérations bloquantes étant plus coûteuses que les non-bloquantes, on tente de les limiter. Lorsque le taux de transfert est inconnu ou variant, il faut absolument utiliser une communication bloquante. Autrement, on impose une communication bloquante au processus le plus rapide et une communication non-bloquante au processus le plus lent. Si les taux de transfert sont égaux, on peut utiliser une communication non-bloquante des deux côtés.

Pour la simulation sur POSEIDON, on utilise une architecture précise pour l'interface (Figure E.2). On utilise une FIFO contrôlée en plus d'une queue pour les données. On a autant de queue que l'on a de *threads* et la largeur de ces queues correspond à la grandeur des données à transférer (donc à la largeur du bus). Il n'y a qu'une seule FIFO contrôlée. La FIFO contient la séquence des threads à exécuter. Lorsque des données sont déposées dans la queue, on ajoute à la FIFO le numéro du *thread* associé à la queue dont on vient de déposer une donnée. Lorsque l'on est prêt à communiquer ces données, une demande est envoyée à la FIFO et si le processeur est en mode réception (enable est actif), on envoie un signal à la FIFO pour indiquer qu'elle peut communiquer le numéro du *thread* qu'elle contient à l'ordonnanteur du processeur.

Alors, le transfert entre la queue et le *thread* peut avoir lieu via le bus du système. La machine à état du contrôleur de la FIFO est donnée à la Figure E.3. Il faut noter qu'ici, on montre le cas où il n'y a qu'une seul *thread* donc une seule queue.

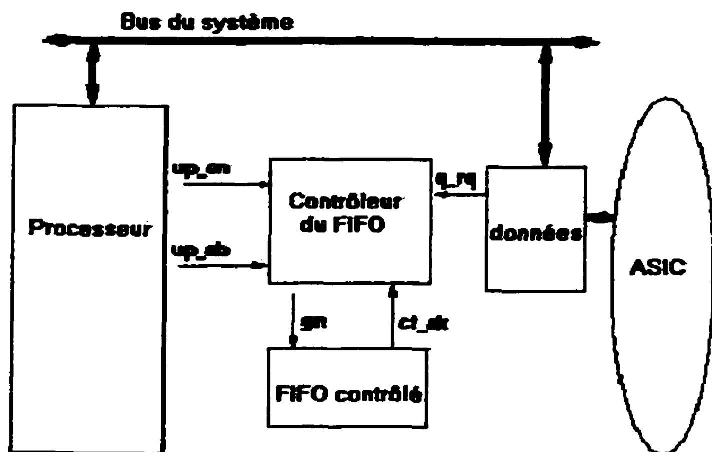


Figure E.2. Architecture d'interface utilisée pour la simulation

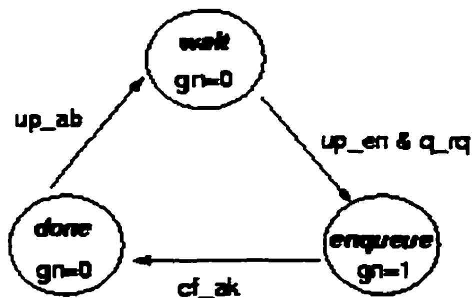


Figure E.3. Machine à états du contrôleur de la FIFO

E.3. Synthèse des communications pour interfaces incompatibles

E.3.1. Définition du problème

Gupta et collaborateurs ont un projet nommé « System interface modeling and synthesis » dont l'objectif est de développer des algorithmes permettant de générer des interfaces de communication observant les contraintes de temps données par les

composants communiquants. Dans [CLG96], donne un algorithme permettant de connecter deux composants ayant des interfaces fixes et incompatibles avec un minimum de logique combinatoire (Figure E.4).

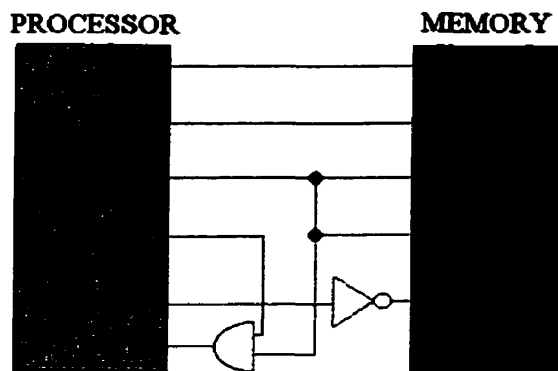


Figure E.4. Logique combinatoire pour relier des interfaces incompatibles

Selon les caractéristiques des interfaces, les ports peuvent être reliés directement, avec de la logique combinatoire ou avec de la logique séquentielle. L'algorithme ne permet pas l'utilisation de logique séquentielle (délai, tampon, bascule, FIFO,...). Cela implique que deux composants n'ayant pas la même largeur de bus de données ou n'ayant pas le même taux de transfert ne peuvent être connectés à l'aide de cet algorithme. L'équipe voudrait développer une méthodologie d'extraction d'interface (combinatoire et séquentielle) à partir de Timed decision tables (définit dans [LiGu98]). Ils veulent aussi développer les techniques d'encodage pour permettre de connecter différents composants sous des contraintes de performances et de coût.

Il y a quatre sortes d'événements sur les ports : des transitions d'une valeur logique haute à une basse ou vice versa ainsi que des transformations des données valides à non valides et vice versa. On définit un protocole comme un ordonnancement partiel des relations d'un ensemble d'événements et une opération de communication comme une séquence d'événements. La synthèse d'interface se fait à partir de la description des événements sur les ports des composants.

E.3.2. Entrées au processus de synthèse

L'entrée au processus de synthèse est la feuille de données sur les entrées/sorties (descriptions des ports et diagrammes temporels) des composants à connecter et le flowchart des opérations entre les composants (représentant le protocole de communication utilisé). À partir du diagramme temporel, on peut savoir le nombre de transitions sur chaque port et le temps minimum et maximum entre deux événements sur le même port. On décrit deux types de contraintes : produites (on garantit que les contraintes seront respectées) et requises (on doit rencontrer ces contraintes). L'entrée à l'algorithme est un graphe de transition de signal (STG). On doit donc transformer le diagramme de temps des ports des composants en STGs.

Les nœuds du STG représentent les événements tels que décrits à la section précédente. Les branches représentent les contraintes de temps entre ces événements. On représente les contraintes maximales avec des lignes pointillées et les contraintes minimales avec des lignes pleines. Les contraintes requises et produites sont représentées par deux STGs différents. Les relations de causalité tirées du flowchart du protocole sont représentées par des contraintes minimales dans les STGs. Pour chaque opération entre les composants, on génère donc deux STGs (requis et produits).

On utilise trois types de connexions pour générer l'interface : connexion aux sources de voltage, connexion directe des ports et connexion nécessitant de la logique combinatoire. Le bue de l'algorithme est de connecter les ports de sortie d'un composant aux ports d'entrée de l'autre en respectant les contraintes de temps tout en minimisant la logique combinatoire. On dit qu'un port de sortie est compatible avec un port d'entrée si les contraintes produites par le port de sortie peuvent satisfaire les contraintes requises du port d'entrée.

Pour minimiser la quantité de logique combinatoire il faut maximiser les connexions de types 1 et 2. De plus, on cherche à minimiser la quantité de fils et la consommation de puissance. Pour minimiser la quantité de fils, on maximise le partage des ports de sorties tout en respectant les contraintes de fanout. En plus, lorsque plusieurs ports de sortie peuvent être connectés à un port d'entrée, on choisit celui qui subit le moins de transition pour réduire la consommation de puissance. L'algorithme tentera alors de trouver quels ports d'entrée et de sortie sont compatibles et choisira la solution qui a le plus petit coût.

E.3.3. Détermination des connexions de type 1 et 2

Comme expliqué plus haut, on tente de maximiser les connexions de types 1 et 2. Les étapes de synthèse sont les suivantes. La première étape de l'algorithme est d'unir les STGs requis avec les STGs produits pour toutes les opérations de façon à ce que chacun des ports d'entrée d'un composant soit associé à un port de sortie de l'autre composant. Si le graphe obtenu après l'union contient des cycles positifs, la solution trouvée n'est pas possible à implanter. On forme autant de graphe qu'il y a de solutions possibles.

La deuxième étape est de construire un graphe d'interface pour chaque opération. Les nœuds de ce graphe représentent les ports d'entrées et de sortie des composants et les branches représentent les associations possibles entre les ports. On obtient les branches du graphe à partir des STGs obtenus à l'étape précédente. On associe un poids à chaque branche pour refléter l'efficacité de l'association. Le poids correspond au nombre de transition du port de sortie (permet une optimisation de la puissance). Ensuite, on fait l'intersection des graphes de toutes les opérations. On garde une branche si elle est présente sur tous les graphes pour avoir une solution qui fonctionne pour toutes les opérations.

La troisième étape est de choisir l'association des ports d'entrées/sorties pour minimiser la quantité de fils et la puissance consommée. On fournit donc une fonction de coût qu'il faut minimiser pour trouver la solution optimale. Il faut cependant toujours s'assurer que chacun des ports d'entrée ne soit connecté qu'à un et un seul port de sortie.

$$\text{COÛT} = \alpha * \Sigma_{VI \in VOUT}(W(VI) * SI) + \beta * \Sigma_{(VI,VJ) \in BRANCHES}(W(VI) * CIJ)$$

α ET β SONT DES FACTEUR DE POIDS
W CORRESPOND AU POIDS ASSOCIÉ AU PORT DE SORTIE
SI EST 1 SI LE PORT DE SORTIE I EST ASSOCIÉ À AU MOINS UN PORT D'ENTRÉE ET 0 AUTREMENT
CIJ EST ÉGAL A 1 SI LES NŒUDS I ET J SONT RELIÉS

E.3.4. Détermination des connexions de type 3

Une fois que l'on a déterminer les connexions de type 1 et 2 en optimisant la fonction de coût, il faut établir des connexions de type 3 pour les ports non connectés. Il faut donc déterminer la logique combinatoire nécessaire. On doit d'abord identifier la plus grande unité de temps qui permet de représenter toutes les transitions. Pour chaque opération, on crée un tableau représentant la valeur sur les ports par (1,0,u,d) à chaque unité de temps. Pour chaque opération, on fait de la composition de signaux sur les ports de sortie en utilisant les opérateurs logiques (AND, OR et NOT) et la composition est valide si elle ne contient aucun u. Par exemple, on prend deux ports et on effectue une opération AND sur la valeur des signaux de ces ports pour chaque unité de temps.

À partir des ports composés, on peut trouver le port composé qui génère des valeurs semblables aux ports d'entrées non connectés. Si l'association entre le port composé et le port d'entrée est valide pour toutes les opérations, on peut générer la logique combinatoire pour connecter ces ports. Notons que d'évaluer toutes les compositions possibles est très long et on donne un algorithme pour aller plus rapidement.

ANNEXE F : SYSTÈME COWARE

COWARE est un outil de synthèse de systèmes hétérogènes. On présente le modèle de données utilisé pour la synthèse. Puis on donne une description générale du processus de synthèse. Certaines étapes de synthèse sont expliquées en détails : la sélection du mécanisme de communication, l'implémentation des communications et la synthèse d'interface.

F.1. Modèle des données dans COWARE

F.1.1. Introduction aux principes de COWARE

Les systèmes de communications ou les systèmes de traitement de signaux sont de plus en plus implantés par des systèmes hétérogènes comportant un ou plusieurs processeurs (DPS ou ASIP (Application Specific Instruction Set Processor) et modules matériels (ASIC). En plus, très souvent, on utilise un type d'architecture mixte. Pour permettre de rencontrer les exigences de ce type de système, l'équipe de COWARE (voir [BML+97]) a développé un environnement de design et synthèse de systèmes hétérogènes basée sur un modèle de données consistant. Ce modèle de données peut être utilisé de la spécification de haut niveau à l'implémentation en passant à travers divers étapes de raffinement.

Le système est décrit par l'intermédiaire de processus qui eux même contiennent différents *threads*. Les processus sont décrits dans un langage défini. Les langages supportés sont le C, le DFL, le VHDL et le Verilog. Les processus communiquent à partir de ports et les ports sont connectés par des canaux. On utilise le principe du RPC pour permettre une description des communications de haut niveau. Le modèle des données comprend donc plusieurs objets : processus, *thread*, ports, canaux et protocoles. Le raffinement de ces objets permet de passer d'une description de haut niveau à une implémentation du système. Les objets primitifs sont les objets les plus

abstraits et donc permettent de décrire le système à un niveau élevé d'abstraction. Les objets contenant les détails d'implémentation sont les objets hiérarchiques.

F.1.2. Objets primitifs et communication de haut niveau

Les objets primitifs permettent de produire une description du système au niveau comportementale sans se préoccuper des signaux, terminaux ou protocole de bas niveaux. Les processus sont des contenants pour les composants du système. Il est alors possible de représenter ces composants avec différents langages et à plusieurs niveaux d'abstraction (selon l'étape de raffinement). Il est parfois intéressant de regrouper plusieurs processus, mais ce groupement peut seulement être fait pour des processus écrit dans un même langage. Pour avoir plus de flexibilité à l'étape du partitionnement, la description des processus avec différents langages est primordiale.

Les *threads* constituent un flow de contrôle unique à l'intérieur d'un processus. On a deux sortes de threads. Les slaves threads sont associés à un port esclave et exécutés lorsque leur port esclave est activé. Les ports esclaves ne peuvent être accédés que par le slave *thread* qui leur est associé. Chaque processus contient un port implicite (construct port) utilisé pour l'initialisation du système et il existe toujours un *thread* spécial associé au construct port. Il y a aussi les autonomous *threads* qui ne sont associés à aucun port esclave et qui sont exécutés dans une boucle infinie dès l'initialisation du circuit. Les ports maîtres peuvent être accédés par les deux sortent de *threads*.

Les ports primitifs permettent la communication entre les *threads* (donc entre les processus). Ils sont caractérisés par un protocole ainsi que des paramètres pour les types de données. Les protocoles définissent la sémantique de communication du port. Au niveau haut d'abstraction, on définit les ports maîtres (activent les RPC) comme inmaster, outmaster ou inoutmaster et les ports esclaves (servent les RPC) comme

inslave, outslave ou inoutslave. On indique donc la direction de données (in, out et inout) ainsi que la direction du contrôle (master ou slave).

Les canaux permettent la connexion point-à-point d'un port maître avec un port esclave. Les canaux primitifs offrent seulement la possibilité de communiquer de façon synchrone sans tampon. Aucun comportement particulier du canal n'est possible avec l'utilisation de port primitif.

Les communications s'effectuent toujours entre les *threads*. Si les deux threads communiquants font partie du même processus, on parle de communication intraprocess. Sinon, on parle de communication interprocess. Les communications intraprocess se font par mémoire partagée par l'utilisation de variables ou signaux communs. Le concepteur est responsable d'éviter l'utilisation simultanée de deux variables. La communication interprocess s'effectue via l'utilisation de RPC. Cette communication est donc implicitement une communication point-à-point. Cependant, il est possible d'implémenter une communication par mémoire partagée de façon indirecte.

F.1.3. Raffinement des communications

Une fois que le comportement du système a été décrit à partir d'objets primitifs, il est possible de raffiner le système en transformant les objets primitifs en objets hiérarchiques. Il faut d'abord raffiner les processus en les regroupant et en les décrivant avec un seul langage. Ensuite, il faut raffiner les canaux en assignant un comportement aux canaux primitifs. Par exemple, on peut permettre une communication parallèle en ajoutant des tampons ou rendre la communication non-bloquante en ajoutant une FIFO. Lors du raffinement, la direction des données est toujours préservée.

Les ports sont ensuite raffinés en leur donnant un comportement particulier. Par exemple, on pourrait faire la vérification des données avant de les communiquer aux autres processus. La direction des données est toujours conservée mais la direction des signaux de contrôle peut être modifiée. On associe des protocoles aux ports. Les protocoles hiérarchiques sont raffinés à partir des protocoles primitifs en ajoutant des informations sur les chronogrammes de la communication ainsi que les terminaux d'entrées/sorties physiques des composants.

F.2. Description générale du processus de synthèse

Dans [MBL+96] et [RVBM96], on explique la méthodologie générale de la synthèse avec COWARE. On doit d'abord fournir une description du système au niveau comportemental à partir des objets primitifs (voir section 7.1.2). À partir de cette description, on fait une optimisation du code pour minimiser les accès mémoire et la dissipation de puissance. Ensuite, il faut choisir les composants du système (allocation).

On peut alors passer à l'étape du partitionnement. Puisque la description initiale est divisée en grain fin, la tâche du partitionnement est de regrouper les processus, unir les spécifications fonctionnelles et les assigner au même processeur (binding). Cette tâche n'est pas automatisée dans COWARE, elle doit être réalisée par le concepteur. Le regroupement de processus et la linéarisation (fait par le Inliner dans COWARE) des descriptions permet d'éliminer des pertes de temps de communication entre les processus. Ensuite, cela diminue le nombre de *threads* et donc diminue le temps de changement de *thread*.

On peut ensuite passer au raffinement. On commence par raffiner les canaux et cette étape est nommée sélection du mécanisme de communication. On détermine le type de communication qui sera utilisée par le canal. Il faut ensuite raffiner les ports et les protocoles, c'est l'implémentation des communications. On transforme alors les ports

et protocoles en une implémentation réelle des communications. On doit générer les processus nécessaires à cette implémentation. L'interface d'un module logiciel est composée de pilote d'interface (processus logiciel) et d'adapteur d'entrées/sorties (processus matériel). Symphony permet de générer les pilotes et de générer une partie de l'interface matérielle. La synthèse du matériel est complétée par le logiciel Integral.

Les interfaces des modules matériels sont implantées à partir des bibliothèques de l'outil Symphony. Une fois que tout le code est généré (matériel et logiciel), on unit les différentes parties du code avant de passer à la phase finale de l'implémentation : le back-end.

F.3. Sélection du mécanisme de communication

La sélection du mécanisme de communication permet de faire l'implémentation des canaux. On choisit le type de communication qui se tiendra sur chaque canal. On peut par exemple choisir une communication complètement bloquante pour s'assurer qu'aucune donnée n'est perdue. Dans ce cas, chacun des processus doit attendre que la communication soit terminée avant de continuer son exécution. On peut aussi choisir une communication bloquante du côté de l'émetteur et non-bloquante du côté du récepteur. Dans ce cas, on a une FIFO entre les deux processus pour permettre la mémorisation des données qui ne sont pas consommées immédiatement par le récepteur.

F.3.1. Processus ayant la même horloge

Dans [LiVe96], on introduit un mode de communication particulier : wait transfert mode. Ce mode peut être utilisé lorsque les processus communiquants utilisent la même horloge et qu'ils sont activés sur le front montant de l'horloge. De plus, on assume que les sorties sont registrées. On a deux signaux de contrôle : recvRdy et sendRdy. L'émetteur met les données sur le bus et initie le transfert en envoyant un

signal sendRdy. Il attend ensuite le signal recvRdy du récepteur avant de passer à un autre état (Figure F.1). Puisque les sorties sont registrées et que les états sont activés sur un front montant d'horloge, on peut présumer que le transfert sera effectué dans le même cycle que le récepteur envoie le signal recvRdy. C'est pourquoi la fin du transfert est implicite et ne nécessite pas de signaux de contrôle supplémentaire. De cette façon des transferts en mode burst peuvent être implémenter très efficacement, c'est-à-dire à raison de un transfert par cycle.

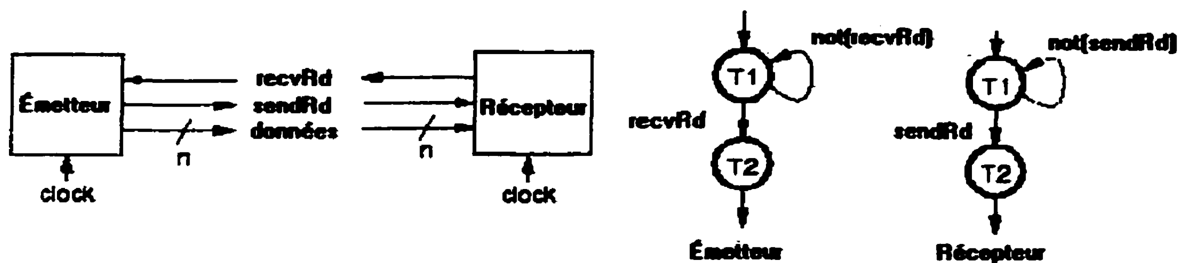


Figure F.1. Illustration du wait transfer mode dans COWARE

F.3.2. Processus ayant des horloges dérivées de la même horloge système

Lorsque les horloges des deux processus ne sont pas les mêmes mais qu'elles sont dérivées de la même horloge, on peut utiliser un adapteur de canal pour permettre la communication (Figure F.2). L'adapteur de canal est en fait une machine à états qui permettra d'insérer les cycles d'attente lorsque l'un des deux côtés est plus rapide que l'autre. Si l'horloge du récepteur est N fois plus rapide que celle de l'émetteur, il faut alors insérer N-1 cycles d'attente du côté du récepteur (avant qu'il effectue la lecture) si celui-ci doit effectuer plusieurs lectures en ligne. Dans ce cas, la machine à états utilise la fréquence du récepteur (la plus rapide des deux) pour insérer ces cycles d'attente. Le principe est exactement le même si l'émetteur est N fois plus rapide que le récepteur sauf que les rôles sont inversés.

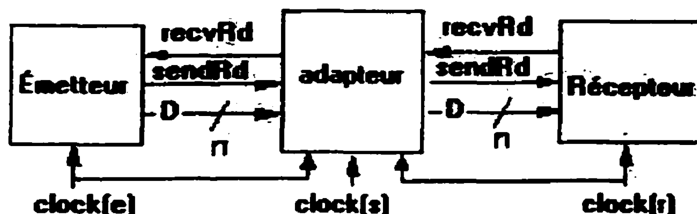


Figure F.2. Adapteur pour des processus d'horloges reliées

Dans le cas où les horloges ne sont pas multiples l'une de l'autre, la machine à états doit utiliser une fréquence équivalente au plus petit commun multiple des horloges des deux processus. On insère alors des cycles d'attente des deux côtés. Si l'horloge de la machine à états est P fois plus rapide que le récepteur et Q fois plus rapide que l'émetteur, on insérera P-1 cycles du côté du récepteur et Q-1 du côté de l'émetteur.

F.3.3. Processus ayant des horloges non reliées ou sans horloge

Lorsque les horloges des processus sont non reliées ou que un des processus fonctionne de manière asynchrone, l'adapteur de canal est un peut plus complexe. Expliquons d'abord le premier cas. Puisqu'il est impossible de relier les deux horloges, il faut utiliser le handshake (à deux ou quatre phases) pour permettre la communication. On insère donc un adapteur de chaque côté fonctionnant à la fréquence du module qu'il connecte. Cet adapteur permet de transformer les signaux du wait transfer mode en des signaux request-acknowledge utilisé dans le protocole de handshake. Il faut aussi ajouter des modules de synchronisation sur les signaux request et acknoledge pour permettre de contrer les différences de fréquence d'horloge entre les deux adapteurs.

Lorsque l'un des modules est complètement asynchrone, le principe est le même. Cependant, au lieu d'ajouter un adapteur, on ajoute un module d'interface pour faire en sorte que la communication du module asynchrone se fasse selon le même protocole du handshake que l'adapteur du module synchrone.

F.4. Implémentation des communications

F.4.1. Communication entre deux modules matériels

Comme expliqué plus haut, l'implémentation des communications se fait avec Symphony. Dans [BML+97], on présente l'implémentation des communications entre deux modules matériels, entre un module matériel et un logiciel et entre deux modules logiciels. Dans le de modules matériels, il faut aussi générer un module d'interface pour permettre la connexion avec le canal. Le module matériel est d'abord synthétisé pour obtenir une description avec seulement des ports maîtres. Ensuite, pour avoir les ports esclaves prévus dans la description, on ajoute une encapsulation. Symphony se sert de ses bibliothèques pour implémenter le module d'interface.

Si on utilise le wait transfer mode décrit dans la section précédente, la connexion au canal se fait facilement. En effet, il faut seulement s'assurer que les processus matériels utilisent les RPC (avec les signaux recvRdy et sendRdy) pour communiquer avec l'extérieur. Si c'est le cas, la connexion est directe. Sinon, il faut ajouter une interface pour permettre la génération des ces signaux. Notons que si un module est asynchrone, il faut que celui-ci utilise le protocole de handshake (signaux request et acknoledge) et non le wait transfer mode pour que la connexion se fasse sans module d'interface. Pour faciliter le design au niveau fonctionnel, il est possible de créer des packages en VHDL pour implanter des fonctions send et receive utilisant les signaux recvRdy et sendRdy. De cette façon, le concepteur ne fait qu'instancier ces fonctions pour établir une communication.

F.4.2. Communication entre un module logiciel et un module matériel

Dans le cas d'une interface logicielle-matérielle, on doit d'abord générer les pilotes d'entrées/sorties pour permettre de lier les processus logiciels implémentés sur le processeur avec l'interface matériel de celui-ci. L'outil Symphony permet de faire la génération des pilotes (processus logiciel). En entrée au processus, il faut fournir un

modèle (logiciel et matériel) du processeur ainsi qu'une bibliothèque de scénarios d'entrées/sorties. Le modèle matériel du processeur comprend une description du comportement de son interface, les protocoles et les chronogrammes de ces ports et parfois une description VHDL du comportement interne du processeur. Le modèle logiciel du processeur comprend la liste de fonctions et des instructions utilisées pour accéder les ports.

Les scénarios correspondent aux façons d'utiliser les ports du processeur pour implanter une communication entre un module logiciel et l'extérieur. Symphony supporte plusieurs scénarios. On peut utiliser les ports mémoire du processeur en associant une zone d'adresse aux entrées/sorties. Ce sont alors les instructions load et store qui permettent d'établir la communication avec l'extérieur. On peut aussi utiliser les ports dédiés du processeur ainsi que les instructions spécifiques à ces ports pour communiquer. Les interruptions peuvent aussi faire partie d'un scénario d'entrées/sorties. Si le nombre de port d'interruption du processeur n'est pas suffisant, on peut utiliser des vecteurs d'interruptions qui indiquent au programme de faire un saut vers une routine spéciale.

La synthèse des pilotes de fait donc en choisissant les ports et les scénarios les plus appropriés pour la communication à implémenter. On propose l'utilisation d'un algorithme basée sur une fonction de coût pour faire les choix. Une fois les modules logiciels générés, il faut développer l'interface matérielle. Symphonie fournit une description de haut niveau de cette interface à partir de ses bibliothèques ainsi que les protocoles des ports utilisés dans la communication mais c'est l'outil Integral qui fait la synthèse réelle du module. Le résultat de l'implémentation des communications entre un module logiciel et matériel est donné à la Figure F.3.

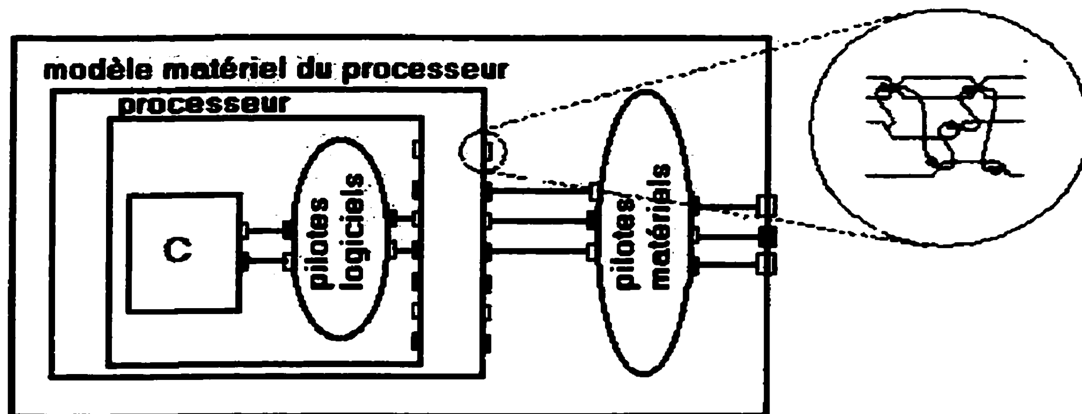


Figure F.3. Communications entre un processeur et un module externe

Une fois les pilotes générés, il faut faire le lien entre les ports physiques du processeur (avec les protocoles associés) et les modules extérieurs (avec les protocoles associés). La génération de ces modules matériels est faite en partie par Symphony et en partie par Integral. Symphony génère une description de haut niveau du module d'encapsulation permettant de traiter les données (vérification des données, packing, splitting). À partir de cette description de haut niveau, Integral génère fait la synthèse du contrôle en respectant les protocoles des deux interfaces.

F.4.3. Communication entre deux modules logiciels

Le problème de communications entre modules logiciels est divisé en deux. Si on a deux processus implémenter sur deux processeurs différents, la communication peut se ramener au cas de la section précédente. Cependant, si les deux processus communiquants sont sur le même processeur, on unit ces processus et la communication devient une communication entre deux *threads* d'un même processus, c'est-à-dire intraprocess communication. Comme expliqué plus haut, cette communication s'effectue via la mémoire partagée. Cependant, c'est la responsabilité du concepteur de vérifier qu'il n'y a pas de conflits de ressources.

Lorsque que la communication implique au moins un slave *thread*, il n'y a pas vraiment de problème puisque ce thread est seulement activé par un événement sur son port esclave. Alors, le processeur utilise des routines d'interruptions pour permettre la communication avec l'extérieur et le résultat est que les deux *threads* ne seront pas en cours d'exécution en même temps. Cependant, un problème se pose lorsque l'on veut synchroniser deux autonomous *threads*. Puisque ces threads sont exécuter en tout temps, il faut un noyau temps réel ou un ordonnanteur pour éviter les problèmes de synchronisme. On propose plusieurs façon de fonctionner pour remédier à ce problème. Cependant, Symphony utilise le principe de rendez-vous pour permettre les communications entre ces threads.

F.5. Synthèse des interfaces

F.5.1. Entrées au processus de synthèse

Dans [LiVe94] on explique comment la synthèse de module d'interface entre des protocoles différents est effectuée avec Integral. En entrée au processus de synthèse, il faut fournir une bibliothèque de protocoles des entrées/sorties des modules à connecter ainsi qu'une description de haut niveau du module de communication (peut venir de Symphony ou du concepteur). Les protocoles sont définis par des chronogrammes annotés de contraintes de temps. Chaque protocole est associé à une série de canaux dont il définit les communications. Il faut convertir les chronogrammes en réseaux de Pétri (ou STG puisque les seules actions possibles des transitions de signaux).

Dans ces STG, on représente une transition montante sur un signal par un plus et une transition descendante par un moins. On représente l'état des données par un signe # lorsque les données sont valides et lorsqu'elles ne le sont pas. Dans le processus de synthèse, on ne tient pas compte des contraintes de temps. On suppose qu'une fois la synthèse terminée, on passe par une étape de vérification pour s'assurer que les contraintes sont respectées.

Le deuxième élément à fournir au processus de synthèse est la description de haut niveau du module de communications. La description utilise le modèle des RCP pour communiquer et chaque RCP est associé à un protocole particulier de la bibliothèque. Dans la description, on peut avoir six types d'opérations (en plus des boucles ou des conditionnelles). Comme pour les descriptions des protocoles, on peut avoir des transitions de signaux de contrôle ou des changements d'état des données. On peut aussi avoir les communications internes ou des communications externes. On peut avoir des opérations arithmétiques et logiques ou des opérations tampons (utilisées pour faire le traitement des données). Cette description doit être transformée en un réseau de Pétri avant de pouvoir faire la synthèse. Si on respecte la syntaxe prescrite pour la description (voir [LiVe94]), la transformation est assez directe.

F.5.2. Raffinement interne de l'interface

Le raffinement de l'interface se fait en deux étapes. On génère d'abord le module principal de l'interface à partir du réseau de Pétri. Ce module représente le comportement interne de l'interface. Les transitions de signaux, les changements d'états des données et les opérations tampons sont déjà raffiner puisqu'ils sont décrits à un niveau d'abstraction bas. Il reste donc à raffiner les communications internes et les opérations arithmétiques. Une fois le raffinement interne terminé, il faut ajouter un second processus au module d'interface (Figure F.4) pour permettent la conversion de protocoles et donc la communication avec l'extérieur.

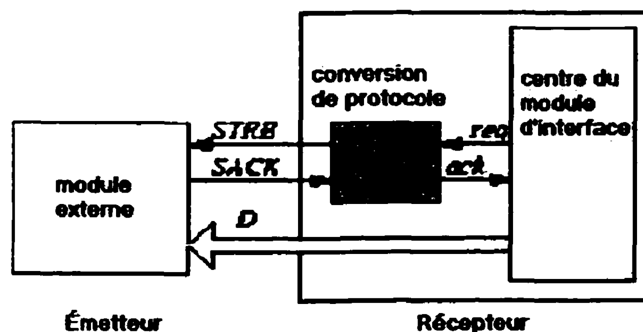


Figure F.4. Module d'interface synthétisée avec Integral

Les opérations arithmétiques et logiques sont raffinées via une bibliothèque de modules. On associe donc les opérations à des modules matériels qui effectue la tâche voulue. La communication entre le module principal et les modules ajoutés se fait à travers un protocole de handshake à quatre phases (avec les signaux request et acknoledge). On ajoute donc au réseau de Pétri initiale les transitions de signaux nécessaires au protocole utilisé pour communiquer avec le nouveau module. Le raffinement des communications internes se fait d'une manière similaire. On transforme les fonctions send et receive par les transitions de signaux associées au protocole utilisé.

F.5.3. Génération du module de conversion de protocole

Une fois le raffinement du module interne terminé, on génère un module auxiliaire pour faire la conversion de protocole et raffiner les communications externes. On identifie d'abord les actions de communications qui sont reliées au même protocole externe et qui se tiennent dans un même cycle de communication. On les groupe pour former un cluster. Pour chaque groupe identifié, il faut ajuster à la description les transitions de signaux nécessaires à l'implantation d'un protocole de communication avec l'extérieur dans la représentation du module interne (de la même façon qu'avec les communications internes).

Ensuite, il faut ajouter les réseaux de Pétri associés aux protocoles utilisés par le module interne pour les communications externes dans la bibliothèque. Cependant, la génération de ces protocoles est très directe puisque dans la majorité des cas le protocole utilisé est le handshake à quatre phases. Les protocoles du module interne sont appelés protocoles primitifs. Les protocoles des modules externes sont appelés les protocoles extérieurs. Le module de conversion de protocole permet de lier les protocoles extérieurs au protocole primitifs.

Pour lier les protocoles, il faut d'abord ajuster les STG des processus récepteurs en fonction des émetteurs. Par exemple, si le module externe est le récepteur, il faut ajouter devant chaque attente d'une donnée valide, l'attente d'un signal request montant. De plus, devant chaque transformation d'une donnée valide en invalide, il faut ajouter la génération d'un signal acknoledge montant. Il faut faire une modification similaire dans le cas où le récepteur serait le module interne.

À partir des nouveaux STG (réseaux de Pétri) des deux protocoles à relier, on génère les miroirs de ceux-ci. Le miroir correspond au STG où les entrées et les sorties sont inversées. On peut ensuite lier les deux STG pour obtenir un seul graphe qui pourra être synthétiser par le logiciel Assassin et ainsi obtenir une description au niveau porte du module de conversion de protocole.

ANNEXE G : PROCESSEUR ARM7DTMI

Dans cet annexe, des figures et des tableaux pertinents à la conception d'une interface de communication pour le ARM sont données. Il faut noter que les images, diagrammes et tableaux ont été tirés de la documentation sur le ARM disponible sur l'internet [ARM00a] [ARM00b] [ARM00c] [ARM00d] [ARM00e] dans le but de faciliter la compréhension du lecteur.

G.1. Signaux du ARM7TDMI

Cette section relate le diagramme fonctionnel du ARM (Figure G.1) et la desctiption des signaux du ARM utilisés dans ce projet (Tableau G.1).

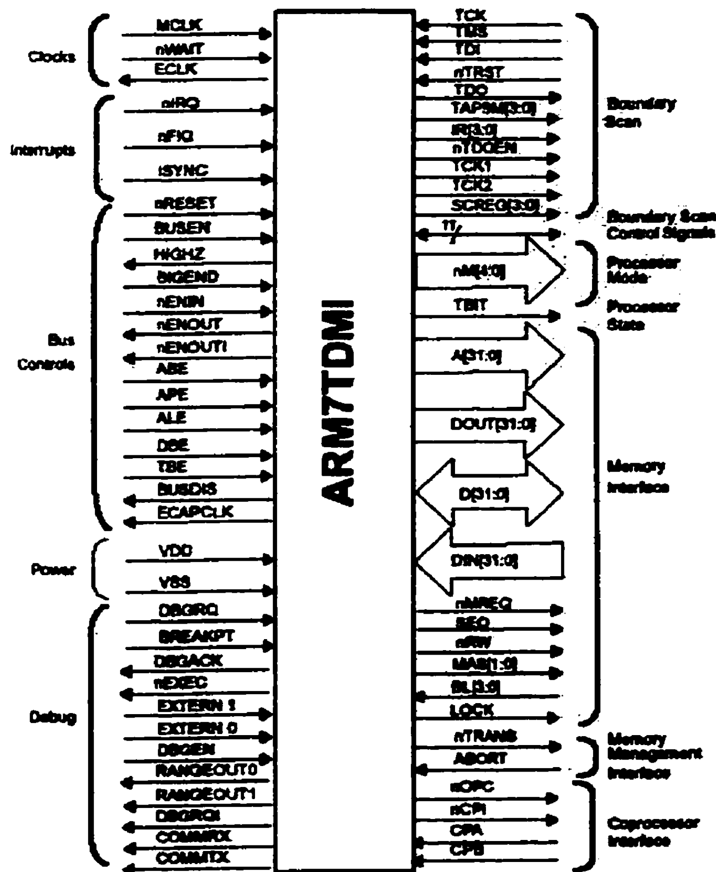


Figure G.1. Diagramme fonctionnel du ARM7TDMI

Tableau G.1. Description des signaux importants du ARM

Nom	Type	Description
A[31 :0]	Sortie	Bus d'adresse du processeur dont les délais sont contrôlés par les signaux ALE et APE.
ABE Address bus enable	Entrée	Ce signal est utilisé pour mettre le bus d'adresses (et signaux associés) en haute impédance (lorsque ABE = 0).
ABORT Memory Abord	Entrée	Ce signal est utilisé par la système mémoire pour avertir que l'accès en cours n'est pas possible.
ALE Adress latch enable	Entrée	Ce signal est utilisé pour contrôler les délais du bus d'adresses. Il est présente pour garder la compatibilité avec les anciennes versions et devrait être gardé à 1 lorsque le signal APE doit être utilisé.
APE Address pipeline enable	Entrée	Ce signal est utilisé pour mettre la génération d'adresse en mode pipeline et ainsi chager les délais du bus d'adresse (voir Figure G.2 et Figure G.3)
BIGEND	Entrée	Lorsque ce signal est à 1, les octets sont considérés en mode Gros-Boutiste et lorsqu'il est à 0, les octets sont considérés en mode Petit-Boutiste.
BL Bytes latch control	Entrée	Ce signal permet d'accéder indépendamment les différents octets d'un mot (l'équivalent d'un "Byte Enable")
BUSEN	Entrée	Ce signal indique au processeur si le bus bidirectionnel (D si BUSEN est à 0) ou les bus unidirectionnels seront utilisés (DIN et DOUT si BUSEN est à 1).
D[31:0]	Entrée/Sortie	Bus de données bidirectionnel
DBE Data bus enable	Entrée	Ce signal permet de mettre le bus de données D en haute impédance (DBE = 0). Il devrait être utilisé seulement pour faire des tests.
Din[31:0]	Entrée	Bus d'entrée de données utilisé pour les lectures mémoire
Dout[31:0]	Sortie	Bus de sortie de données utilisé pour les écritures mémoire
ISYNC	Entrée	Lorsque ce signal est a 0, il indique que les entrées d'interruption doivent être synchronisée avec le processeur.
LOCK	Sortie	Ce signal permet de faire des accées en mode blocage
MAS[1 :0]	Sortie	Cette sortie indique le type d'accès mémoire du processeur 00 => un octet 01 => deux octets 10 => un mot de 4 octets 11 => réservé
MCLK	Entrée	Horloge du processeur.
NFIQ	Entrée	Interruption logicielle (mode de transfert de données donc de permière priorité)
NIRQ	Entrée	Interruption logicielle (mode normal donc de seconde priorité)
NMREQ	Sortie	Ce signal indique aux composants externes que le prochain cycle sera un accès mémoire.
NRESET	Entrée	Signal actif bas pour la restauration.
NRW	Sortie	Ce signal indique un cycle de lecture (à 0) et un cycle d'écriture (à 1)
NWAIT	Entrée	Ce signal permet de mettre le processeur en mode d'attente lorsque le périphérique est lent.
SEQ Sequential address	Sortie	Ce signal indique que le prochaine adresse sera reliée à la précédente.

G.2. Diagrammes temporels du ARM7TDMI

Cette section donne les informations temporelles de l'interface mémoire du processeur. Les diagrammes temporels associés au deux modes d'adressages est d'abord donné. Puis, les délais associés aux différentes transactions sont montrés.

G.2.1. Modes d'adressage

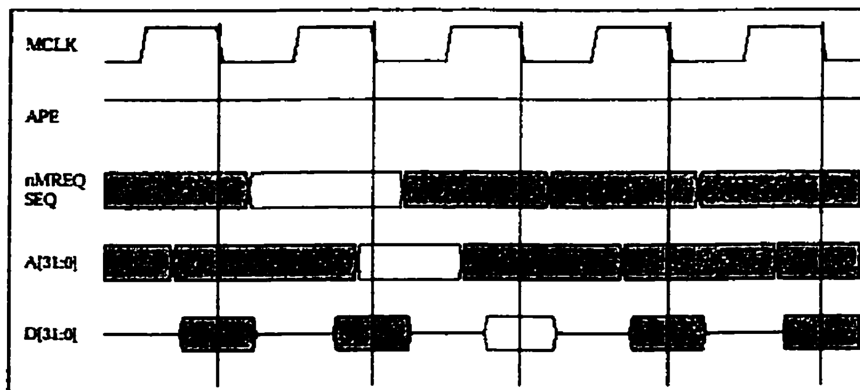


Figure G.2. Diagramme temporel pour un adressage en mode pipeline

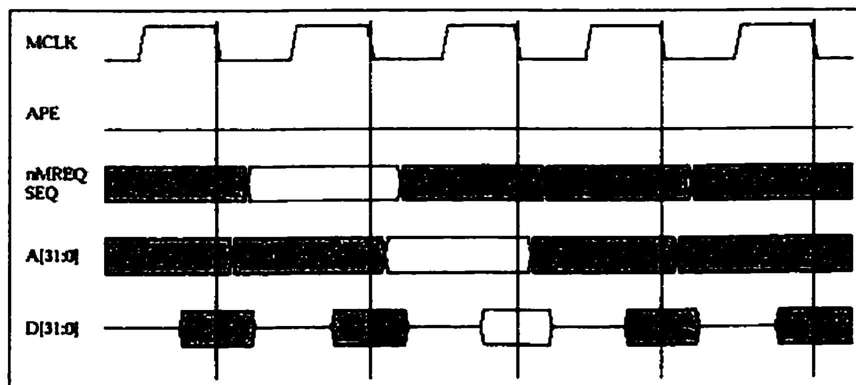


Figure G.3. Diagramme temporel pour un adressage en mode normal

G.2.2. Délais des transactions

Dans la Figure G.4, le signaux nWait, ABE, ALE et ABE sont hauts. Les valeurs numériques des délais suivent les figures des diargammes temporels.

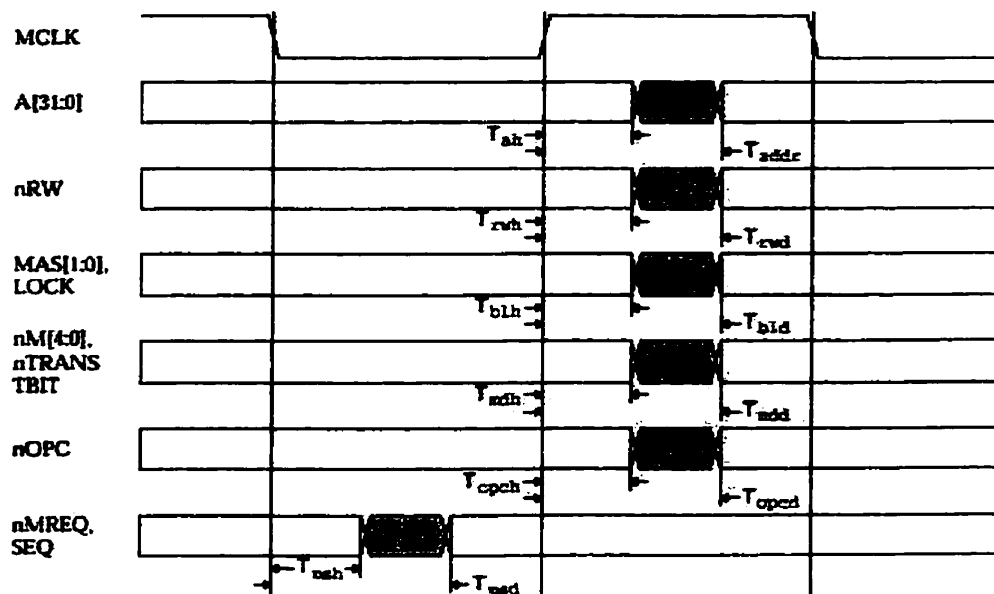


Figure G.4. Diagramme temporel général

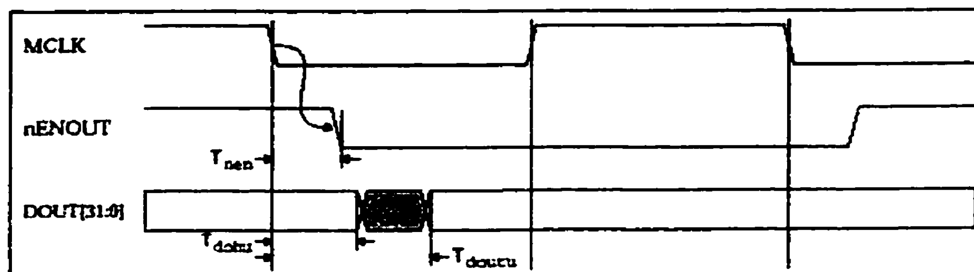


Figure G.5. Diagramme temporel d'une écriture

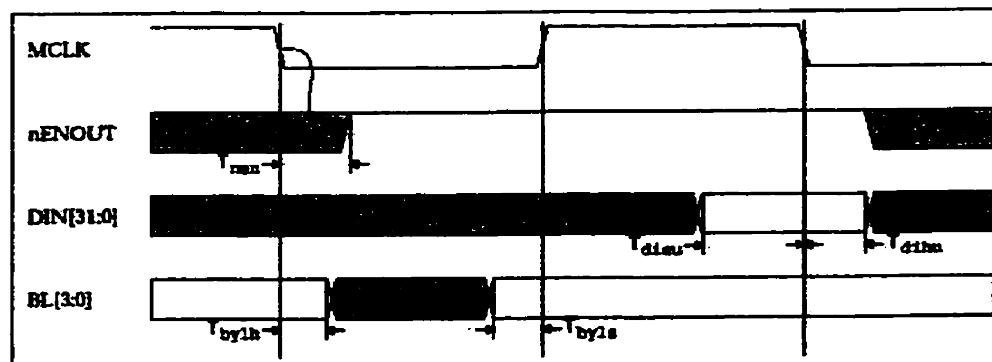


Figure G.6. Diagramme temporel d'une lecture

Tableau G.2. Délais pour les transactions de lectures et écritures

Symbole	[illegible]	Minimum (ns)	Maximum (ns)
Taddr	Mclk à une adresse valide		14
Tah	Temps de maintient de l'adresse (MClk)	2.4	
Tdoutu	Mclk à DOUT valide		17
Tdohu	Temps de maintient de DOUT (Mclk)	2.4	
Tdsiu	Temps de préconditionnement de DIN (Mclk)	1.8	
Tdohu	Temps de maintient de DIN (MClk)	1.7	
Tnen	Mclk à nENOUT valide		11.2
Tnenh	Temps de maintient de nENOUT (MClk)	2.4	
Tbyls	Temps de préconditionnement de BL (Mclk)	0.7	
Tbylh	Temps de maintient de BL (MClk)	0.1	
Trwd	Mclk à une nRW valide		14
Trwh	Temps de maintient de nRW (MClk)	2.4	
Tmsd	Mclk à nMREQ et SEQ valide		17.9
Tmsh	Temps de maintient de nMREQ et SEQ (MClk)	2.4	
Tbld	Mclk à MAS et LOCK valide		18.9
Tblg	Temps de maintient de MAS et LOCK (MClk)	2.4	
Tmdd	Mclk à nTRANS et nM et TBIT		19.5
Tmdh	Temps de maintient de nTRANS et nM et TBIT (MClk)	2.4	
Topcd	Mclk à nOPC valide		10.6
Topch	Temps de maintient de nOPC (MClk)	2.4	

G.3. Éléments de conception intéressants

Dans cette section relate des informations utiles à la conception d'interface pour le ARM. Basé sur les instructions données dans [ARM00d], le schéma d'une architecture mémoire typique est donné. Puis, une définition d'une espace d'adressage typique est donné. Ensuite, un schéma logique du circuit permettant d'acviter les banques mémoires tiré de [ARM00b] est fourni. Enfin, un schéma logique de base d'un registre d'interuption tiré de [ARM00e] est donné.

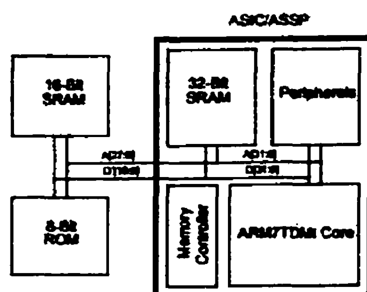


Figure G.7. Architecture de mémoire typique

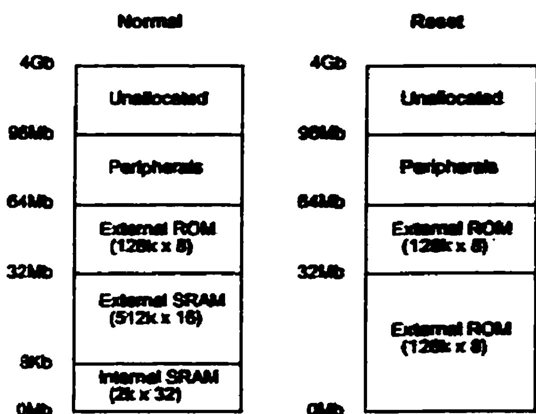


Figure G.8. Définition d'un espace d'adressage typique

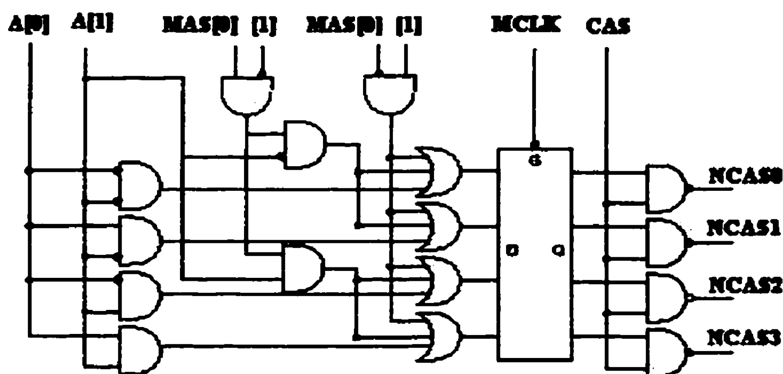


Figure G.9. Schéma logique d'un activateur de banque mémoire

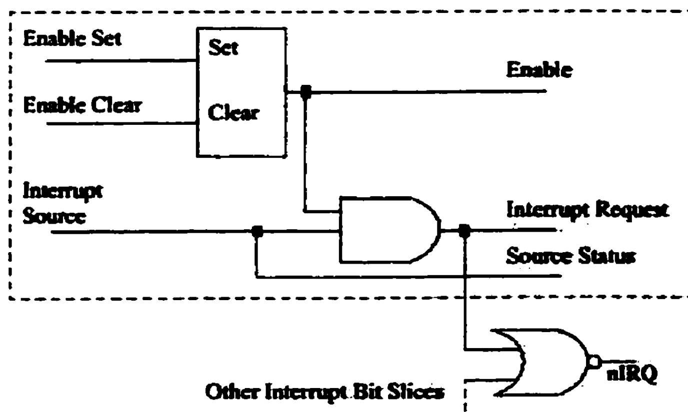


Figure G.10. Schéma logique de base d'un registre d'interruption

ANNEXE H : PROTOCOLE BVCI

Cette annexe donne les diagrammes fonctionnels, la descriptions de signaux de BVCI et le diagramme temporelle de certaines opérations BVCI. Il faut noter que les images, diagrammes et tableaux ont été tirés de la documentation de VSIA disponible sur l'internet aux membres seulement [VSIA00a] dans le but de faciliter la compréhension du lecteur.

H.1. Diagramme fonctionnel du protocole BVCI

Les figures qui suivent montrent les signaux utiles au transferts d'une requête et d'une réponse du protocole BVCI. Le sens des flèches indique qui envoie et qui reçoit le signal, l'initiateur (maître) ou la cible (esclave). La requête est envoyée par l'initiateur et la réponse par la cible.

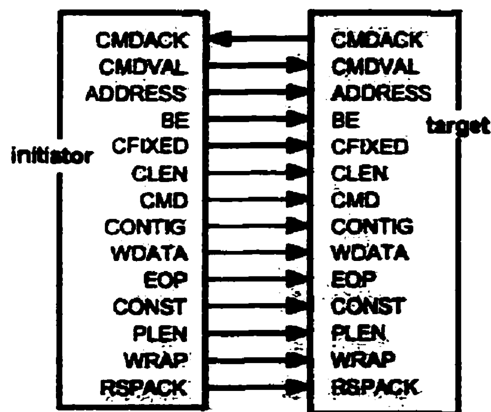


Figure H.1. Signaux du transfert d'une requête BVCI

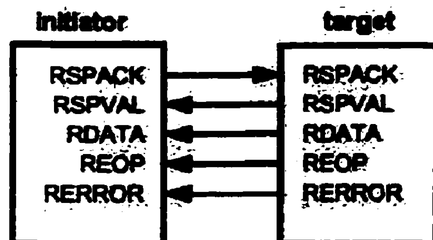


Figure H.2. Signaux du transfert d'une réponse BVCI

H.2. Description des signaux BVCI

Voici la description des signaux VCI (Tableau G.3). Les signaux facultatifs ne sont pas utilisés dans le projet ne sont pas décrits. Ils servent, pour la plupart, aux transferts par paquets complexes.

Tableau G.3. Description des signaux VCI

Nom	Générateur	Description
Signaux système		
CLOCK	Système	Horloge du système.
RESETN	Système	Signal de restauration du système.
Signaux de requête		
CMDACK	Cible	*Command acknowledge* : Signal de poingnée de main indiquant que la requête à été captée.
CMDVAL	Initiateur	*Command valid* : Signal de poingnée de main indiquant que la requête est prête sur le bus.
ADDRESS[n-1 :0]	Initiateur	Adresse
BE[b-1 :0|0 :b-1]	Initiateur	*Byte enable* : Signal indiquant quels octets considérés.
CMD[1 :0]	Initiateur	*Command* : Commande à effectuer 00 => aucune opération 01 => lecture 10 => écriture 11 => lecture bloquante
WDATA[8b-1 :0]	Initiateur	Données à écrire.
EOP	Initiateur	*End of Paquet* : Signal qui indique la fin du paquet de requête.
CFIXED	Initiateur	*Chain fixed* (pas utilisé dans ce projet)
CLEN[q-1 :0]	Initiateur	*Chain Length* (pas utilisé dans ce projet)
CONTIG	Initiateur	*Contiguous* (pas utilisé dans ce projet)
CONST	Initiateur	*Constant* (pas utilisé dans ce projet)
PLEN[k-1 :0]	Initiateur	*Packet Length* (pas utilisé dans ce projet)
WRAP	Initiateur	*Wrap* (pas utilisé dans ce projet)
Signaux de réponse		
RSPACK	Initiateur	*Response acknowledge* : Signal de poingnée de main indiquant que la reponse à été captée.
RSPVAL	Cible	*Response valid* : Signal de poingnée de main indiquant que la reponse est prête sur le bus.
RDATA[8b-1 :0]	Cible	Données lues.
REOP	Cible	*Response end of Paquet* : Signal qui indique la fin du paquet de réponse.
RERROR	Cible	*Response Error* : Signal d'erreur.

H.3. Diagramme temporel d'opérations BVCI

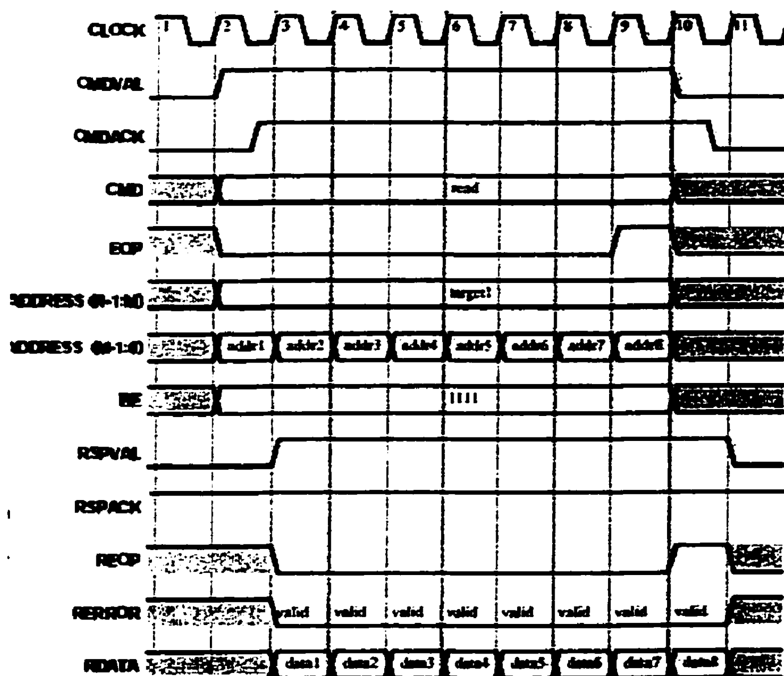


Figure H.3. Diagramme temporel d'une opération de lecture

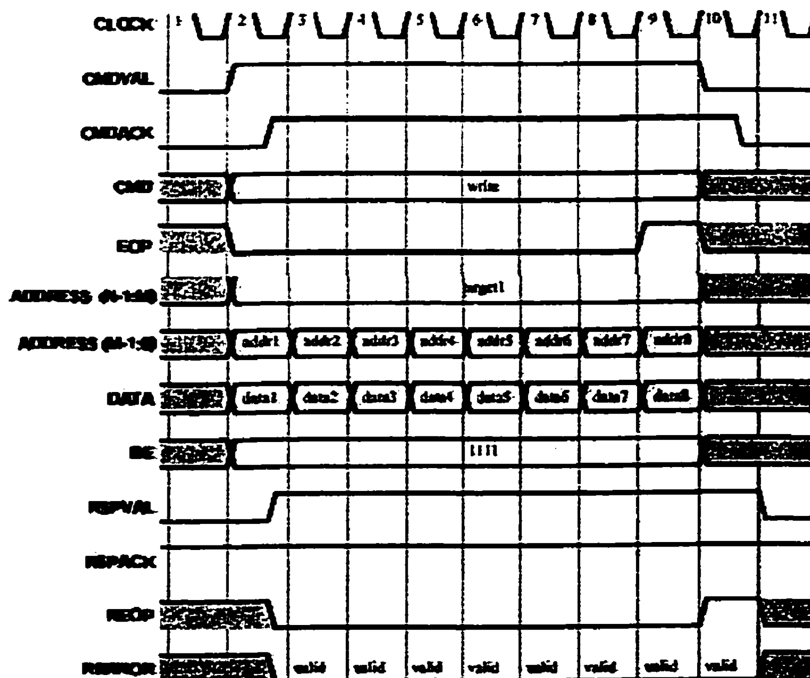


Figure H.4. Diagramme temporel d'une opération d'écriture

H.4. Paramètres BVCI

L'interface BVCI peut être configurable pour faciliter l'intégration au système. Il est bine important de bine documenté l'interface pour que l'intégrateur système puisse la comprendre facilement. Les paramètres peuvent avoir différentes portées. D'abord, tous les paramètres doivent être documentés (Doc). Ensuite, les paramètres peuvent être configurés à l'intérieur du code ("Soft"). Il peuvent aussi être configurés au niveaux matériels au moment de l'assemblage du système ("Hard"). Enfin, les paramètres peuvent être configurés dynamiquement (Dyn).

Comme expliqué à la section 1.3.1.2, il y a différents types de IP. Un "Hard" IP pourra avoir des paramètres documentés, dynamique et "Hard". Un "Soft" IP pourra avoir tous les types de paramètres. Il convient au concepteur de bien documentés les paramètres et leur portée. Le tableau qui suit donne la liste des paramètres ainsi que la portée qu'ils peuvent avoir selon le type de IP.

Tableau H.1. Paramètres BVCI

Paramètres	Description	Intervalles	Portées
INITIATOR	Indique si le IP est un initiateur	0,1	Doc
ADDRSIZE(N)	Grandeur du bus d'adresse en bit	0 à 32	Doc, Soft
CELLSIZE(B)	Nombre d'octet dans une cellule	1,2,4,8	Doc, Soft
BIGENDIAN	Vrai si gros-boutiste et faux autrement	0,1	Doc, Soft, Hard,Dyn
NOENDIAN	Vrai si le stockage des octets n'a pas d'important	0,1	Doc
RESETLEN	Nombre minimum de cycle d'horloge nécessaire à la restauration	entier	Doc
ERRLEN(E)	Nombre de bit d'extension pour l'erreur	0 à 2	Doc, Soft
CHAINING	Vrai si le chaînage est possible	0,1	Doc, Soft
PLENSIZE(K)	Grandeur du signal PLEN en bit	0 à 9	Doc, Soft
CLENSIZE(Q)	Grandeur du signal CLEN en bit	0 à 8	Doc, Soft
DefCMDACK	Vrai si le CMDACK peut etre 1 sans que le CMDVAL le soit	0,1	Doc, Soft, Hard
DefRSPACK	Vrai si le RSPACK peut etre 1 sans que le RSPVAL le soit	0,1	Doc, Soft, Hard
DCFIXED	Signal CFIXED est par défaut à 0 ou 1	0,1	Doc, Hard
DCONST	Signal CONST est par défaut à 0 ou 1	0,1	Doc, Hard
DCONTIG	Signal CONTIG est par défaut à 0 ou 1	0,1	Doc, Hard
DWRAP	Signal CWRAP est par défaut à 0 ou 1	0,1	Doc, Hard

ANNEXE I : MODÈLES SEAMLESS

I.1. DPRAM

Cette annexe donne les éléments de base pour comprendre le fonctionnement de la mémoire SRAM à doubles ports du modèle Seamless. D'abord, le Tableau I.1 donne la description des ports et le Tableau I.3 les modes de fonctionnement. Les attributs du modèle Seamless sont la grandeur du bus de données (1 à 8 et 16-24-32) et du bus d'adresse (1 à 32). En plus des attributs, il y a des options : le nombre de sémaphores (0 à 8) et l'adresse des boîte aux lettres de droite et de gauche (MBR, MBL).

Une mémoire à port duale permet un accès simultané par deux composants différents et donc les ports sont doublés. L'écriture se fait sur la montée du premier des signaux CE (ou SEM) et WE et la lecture se fait sur la descente du dernier des signaux CE (ou SEM) et OE.

Tableau I.1. Description des ports d'une DPRAM

Nom du signal	Direction	Niveau	Description
ADDRL, ADDRR	Entrée		Signal d'adresse de l'accès mémoire
DQL, DQR	Entrée/Sortie		Signal de donnée lue (sortie) ou écrite (entrée)
CEL, CER	Entrée	Bas	Signal d'activation de la puce
OEL, OER	Entrée	Bas	Signal d'activation de la sortie
WEL, WER	Entrée	Bas	Signal d'écriture
SEML, SEMR	Entrée	Bas	Signal d'activation des sémaphores
INTL, INTR	Sortie	Bas	Signal d'interruption généré à l'écriture d'une boîte aux lettre
BUSYL, BUSYR	Entrée/Sortie	Bas	Signal indiquant que la location accédée est occupée (sortie si MS = 0 et entrée si MS = 1)
MS	Entrée		Signal indiquant si l'arbitration est interne (MS = 0) ou externe (MS = 1)

Les boîtes aux lettres sont des espaces mémoire spéciaux (à l'intérieur de l'espace d'adressage régulier) permettre de générer une interruption du côté opposé. Une écriture à la boîte de gauche à partir du côté droit permet de générer une interruption à

gauche. Une lecture subséquente à cette boîte à partir du côté gauche permet de désactiver l'interruption.

Les sémaphores sont aussi des espaces mémoires spéciaux, mais en dehors de l'espace d'adressage régulier (activer par le signal SEM). Lorsqu'un côté écrit 0 dans une sémaphore, une lecture subséquente de cette sémpahore indiquera une prise de contrôle (données à 0). Si l'autre côté écrit dans cette même sémaphore, la lecture subséquente n'indique pas une prise de contrôle (donnée pas à 0). Pour libérer une sémaphore, il faut lui écrire 1. Alors, le port opposé peut en prendre le contrôle.

Il est possible d'avoir de l'arbitration interne. Si le signal MS est à 1, la mémoire empêche deux écritures simultanées à une même location en sélectionnant un port gagnant. Le signal BUSY (sortie) du port perdant est alors activé. Si le signal MS est à 0, les signaux BUSY sont des entrées et l'arbitration est externe. Si plusieurs mémoires sont mises en parallèle, il faut s'assurer que les résultats d'arbitration des mémoires sont les mêmes. Pour ce faire, il est suggérer que déterminer un maître (signal MS à 1) et d'utiliser les signaux BUSY du maître pour arbitrer les autres mémoires (MS à 0). La figure 3.7 montre cette architecture.

<u>Tableau I.2.</u> **Paramètres de la DPRAM**

Symbole	Description	Valeur (ns)
LECTURE		
Trc	Temps d'une lecture	15
Taa, Tace	Temps d'accès à l'adresse, au sélecteur de puce	15
Taoe	Temps d'accès à OE	10
Toh	Temps de maintient de la données après changement d'adresse	3
Tsaa	Temps d'accès à une sémaphore	15
ÉCRITURE		
Twc	Temps d'une écriture	15
Taw	Temps entre adresse ou sélecteur de puce et fin écriture	12
Twp	Temps données valides et fin de l'écriture	10

Selon le manufacturier, les mémoires peuvent différer. Entre autre, les délais varient beaucoup d'une mémoire à l'autre. Dans l'interface développée, les délais utilisés

étaient les mêmes que la mémoire IDT7009L15 de IDT, DPRAM CMOS de 17 bits d'adresse et 8 bits de données (128Kx8). Le Tableau I.6 donne certains paramètres, leur description et leur valeur pour la mémoire sélectionnée.

<u>Tableau I.3.</u> Mode de fonctionnement d'une DPRAM

[illegible]	CE	SEM	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
G ou D	1	1	X	X	X	X	Haute impédance	Attente
G ou D	0	1	0	X	X	X	Données en entrée	Écriture mémoire
G ou D	0	1	1	0	X	X	Données en sortie	Lecture mémoire
G ou D	0	1	1	1	X	X		Ignoré
G ou D	1	0	0	X	X	X	Données en entrée	Écriture sémaphore
G ou D	1	0	1	0	X	X	Données en sortie	Lecture sémaphore
G ou D	1	0	1	1	X	X		Ignoré
G ou D	0	0	X	X	X	X		Pas permis
G	0	1	0	X	MBR	0	Données en entrée	Activer interruption droite
D	0	1	1	0	MBR	1	Données en sortie	Désactiver interruption droite
D	0	1	0	X	MBG	0	Données en entrée	Activer interruption gauche
G	0	1	1	0	MBG	1	Données en sortie	Désactiver interruption gauche

I.2. SRAM

Cette annexe décrit la mémoire SRAM à simple port du modèle Seamless. D'abord, le Tableau I.4 donne la description des ports et le Tableau I.5 les modes de fonctionnement. Les attributs du modèle Seamless sont la grandeur du bus de données (1 à 8 et 16-24-32) et du bus d'adresse (1 à 32). L'écriture set l'écriture se fait sur les même fronts que la mémoire duale.

<u>Tableau I.4.</u> Description des ports d'une SRAM

[illegible]	[illegible]	[illegible]	[illegible]
ADDR	Entrée		Signal d'adresse de l'accès mémoire
DQ	Entrée/Sortie		Signal de donnée lue (sortie) ou écrite (entrée)
CE	Entrée	Bas	Signal d'activation de la puce
OE	Entrée	Bas	Signal d'activation de la sortie
WE	Entrée	Bas	Signal d'écriture

Tableau I.5. Modes de fonctionnement de la SRAM

CE	OE	WE	DQ	[illegible]
1	X	X	Haute impédance	Attente
0	0	1	Données en sortie	Lecture
0	X	0	Données en entrée	Ecriture
0	1	1	Haute impédance	Sortie désactivée

Selon le manufacturier, les délais et spécifications des mémoires varient. Les délais choisis sont ceux de la mémoire IDT71V416L de IDT, SRAM statique CMOS 3.3V de 18 bits d'adresse et 16 bits de données (256Kx16). Le Tableau I.6 donne certains paramètres, leur description et leur valeur pour la mémoire sélectionnée.

Tableau I.6. Paramètres de la SRAM

Symbole	Description	Valeur (ns)
LECTURE		
Trc	Temps d'une lecture	10
Taa, Tacs	Temps d'accès à l'adresse et au sélecteur de puce	10
Toe	Temps entre OE bas et la sortie valide	5
ÉCRITURE		
Taw	Temps entre adresse, sélecteur de puce ou WE valide et fin écriture	8
Twp	Temps données valides et fin de l'écriture	5

I.3. FIFO

Cette annexe donne les éléments de base pour comprendre le fonctionnement de la mémoire FIFO du modèle Seamless. D'abord, le Tableau I.7 donne la description des ports et le Tableau I.7 les modes de fonctionnement. Les attributs du modèle Seamless sont la grandeur du bus de données (1 à 8 et 16-24-32), la profondeur de la FIFO (0 à 256), ainsi que les barières pour la génération des indicateurs.

Tableau I.7. Description des ports de la FIFO

[illegible]	[illegible]	[illegible]	[illegible]
RS	Entrée		Signal de restauration
D	Entrée		Signal de données écrites
Q	Sortie		Signal de données lues
WE	Entrée	Bas	Signal d'écriture
RE	Entrée	Bas	Signal de lecture
EF, FF, HF, AEF, AFF	Sorties	Bas	Signal indicateurs de l'états de la FIFO

Le fonctionnement de la FIFO est simple, l'écriture se fait d'un côté et la lecture de l'autre. Les signaux WE et D sont utilisé pour faire l'écriture et les signaux RE et Q pour la lecture. Les délais typiques utilisés sont de 15 ns. Des indicateurs sont disponibles pour permettre de savoir l'état de la FIFO. Une lecture dans une FIFO vide et une écriture dans une FIFO pleine ne sont jamais effectuées.

ANNEXE J : COMPLÉMENTS DIVERS

J.1. "Package" de base

Comme expliqué à la section 2.3, un "package" est utilisé pour fixer les paramètres de l'interface. Les paramètres du "package est donné ci-dessous.

```
-- PERIODE D'HORLOGE
 constant PERIOD_VCI   : time := 40 ns ;
 constant PERIOD_INT   : time := 4 ns ;
 constant HAFT_INT_VCI_FACTOR : integer := 5;
 constant INT_VCI_FACTOR : integer := 10;

 -- DEFINITION DE L'ESPACE MEMOIRE
 -- sram read only memory of the processor (for program)
constant SRAM_RO_RANGE_BOTTOM : std_logic_vector(31 downto 0) :=
"00000000000000000000000000000000";   -- 00000000
constant SRAM_RO_RANGE_TOP : std_logic_vector(31 downto 0) :=
"00000000010000000000000000000000";  -- 00200000
 -- sram memory of the processor (for data)
constant SRAM_RANGE_BOTTOM : std_logic_vector(31 downto 0) :=
"00000000010000000000000000000000";  -- 00200000
 constant SRAM_RANGE_TOP : std_logic_vector(31 downto 0) :=
"00000000010000000000000000000000";  -- 00200000
 -- off chip memory (not implemented but used here for screen)
constant OFF_CHIP_MEM_RANGE_BOTTOM : std_logic_vector(31 downto 0) :=
"00000000010000000000000000000000";  -- 00400000
constant OFF_CHIP_RANGE_TOP : std_logic_vector(31 downto 0) :=
"00000000011000000000000000000000";  -- 00600000
 --screen used only for debbug
 constant SCREEN_RANGE : std_logic_vector(31 downto 0) := OFF_CHIP_MEM_RANGE_BOTTOM ;
 --dual port memory
constant DPRAM_RANGE_BOTTOM : std_logic_vector(31 downto 0) :=
"00000000011000000000000000000000";  -- 00600000
constant DPRAM_RANGE_TOP : std_logic_vector(31 downto 0) :=
"00000001011000000000000000000000"; -- 01600000
 -- semaphores
constant SEM_RANGE_BOTTOM : std_logic_vector(31 downto 0) :=
"00000001011000000000000000000000";  -- 01600000
constant SEM_RANGE_TOP : std_logic_vector(31 downto 0) :=
"00000001011000000000000000100000";   -- 01600020
 --mailbox right
constant MAILBOX_RIGHT_VCI : std_logic_vector(31 downto 0) := "00000000011000000000000000000100";
constant MAILBOX_RIGHT_DPRAM : INTEGER := 1;
 --mailbox left
 constant MAILBOX_LEFT_VCI : std_logic_vector(31 downto 0) := "00000000011000000000000000000000";
 constant MAILBOX_LEFT_DPRAM : INTEGER := 0;
 -- interrupts register
constant INTREG_RANGE_BOTTOM : std_logic_vector(31 downto 0) :=
"00000001011000000000000000100000";  -- 01600020
constant INTREG_RANGE_TOP : std_logic_vector(31 downto 0) :=
"00000001011000000000000000110000";  -- 01600030
 -- interrupts generation
constant INTGEN_RANGE_BOTTOM : std_logic_vector(31 downto 0) :=
"00000001011000000000000000110000";  -- 01600030
```

```
constant INTGEN_RANGE_TOP : std_logic_vector(31 downto 0) :=
"00000001011000000000000000110100";  -- 01600034
 -- fifo memory
constant FIFO_VERS_EXT_RANGE_BOTTOM : std_logic_vector(31 downto 0) :=
"00000001011000000000000000110100";     -- 01600034
constant FIFO_VERS_EXT_RANGE_TOP : std_logic_vector(31 downto 0) :=
"00000001011000000000000000111000";  -- 01600038
constant FIFO_VERS_INT_RANGE_BOTTOM : std_logic_vector(31 downto 0) :=
"00000001011000000000000000111000";  -- 01600038
constant FIFO_VERS_INT_RANGE_TOP : std_logic_vector(31 downto 0) :=
"00000001011000000000000001000010";  -- 01600042
 -- espace d'adressage reserve aux acces externes
 -- 01700000
 constant EXTERN_RANGE_BOTTOM : std_logic_vector(31 downto 0) :=
"00000001011100000000000000000000";
 -- 02700000
 constant EXTERN_RANGE_TOP : std_logic_vector(31 downto 0) := "00000010011100000000000000000000";
 -- semaphores externes
constant SEM_EXT_RANGE_BOTTOM : std_logic_vector(31 downto 0) :=
"00000010011100000000000000000000";  -- 02700000
constant SEM_EXT_RANGE_TOP : std_logic_vector(31 downto 0) :=
"00000010011100000000000000100000";  -- 02700020
 -- mailbox externe
constant MAILBOX_EXTERNE_BOTTOM_LEFT : std_logic_vector(31 downto 0):=
"00000001011100000000000000000000"; -- 01700000
constant MAILBOX_EXTERNE_BOTTOM_RIGHT : std_logic_vector(31 downto 0):=
"00000001011100000000000000000100";  -- 01700004
constant MAILBOX_EXTERNE_TOP: std_logic_vector(31 downto 0):=
"00000001011100000000000000001000";  -- 01700008


 -- DEFINITION DES GROSSEURS DES MEMOIRES
constant FIFO_MEM_PROFONDEUR : integer := 4;
constant BIT_SRAM_RO : integer := 19;
constant BIT_SRAM : integer := 19;
constant BIT_DPRAM : integer := 22;
constant BIT_SCREEN : integer := 8;
constant BIT_ROM : integer := 1;

 -- DEFINITION DES PARAMETRES De LA FIFO DE L'INTERFACE VCI_TARGET
constant VTI_PROFONDEUR_FIFO : integer := 2;
constant  VTI_FIFO_BIT : integer := 2;
constant  VTI_FIFO_SIZE : integer := VTI_FIFO_BIT - 1;
constant VTI_FIFO_VIDE : std_logic_vector(VTI_FIFO_SIZE downto 0) := (0 => '1', others => '0');

 -- DEFINITION DES INTERRUPTIONS
 constant BIT_INT : integer := 4;
 constant INT_NUM : integer := BIT_INT - 1;

 -- DEFINITION DES SEMAPHORES
constant SEM_NUM : integer := 8;
constant BIT_SEM : integer := 3;
```

J.2. Exemple de code assembleur pout les routines d'interruptions

L'exemple donné est simple et permet de voir la base d'une routine d'interruption. Seulement deux registres sont utilisés, deux routines d'interruptions sont possibles et les boîtes aux lettres externes n'existe pas.

```
; ; Enregistrement du contexte actuel
    STMDB  sp!,{R12}
    STMDB  sp!,{R11}
    STMDB  sp!,{R10}

    ;; identification du numero de la routine et de sa provenance
    ;; provient-elle d'une boîte aux lettres
    MOV    R12, #Mailbox_add
    LDR    R11, [R12]
    ;; éteindre l'interruption
    MOV    R10, #Write0
    STR    R10, [R12]
    ;; si l'interruption vient d'une boîte, on exécute la routine appropriée
    CMP    R11, #Int_1
    BEQ    premiere_routine
    CMP    R11, #Int_2
    BEQ    deuxieme_routine
    ;; si l'interuption ne vient pas d'une boîte aux lettres
    MOV    R12, #Treated
    LDR    R11, [R12]
    AND    R10, R11, #Int_1
    CMP    R10, #Int_1
    BEQ    premiere_routine
    AND    R10, R11, #Int_2
    CMP    R10, #Int_2
    BEQ    deuxieme_routine

    ; Au cas ou aucune interruption n'est identifiée
    ; le contexte doit être restauré
    BEQ    restauration

restauration
    MRS    R11,SPSR          ; Pickup CPSR again
    ORR    R11,R11,#F_Bit    ; Build lockout value
    MSR    SPSR,R1           ; Lockout interrupts
    LDMIA  sp,{r11-r12}      ; restore registers
    SUBS   pc,lr,#4   ; return to the point of the interrupt

premiere_routine  (deuxième routine semblable mais appelle une routine C différente)
    ;; indiquer que l'interruption est en traitement
    MOV    R12, #Treated
    MOV    R11, #Int_1
    STR    R11,[R12]
```

```
      ; mémoriser le reste des registres
     STMDB  sp!,{R0-R9}
     MOV          R0, lr
     STMDB  sp!,{R0}
; appeler la routine C
      BL    int1    ; Call the interrupt handler
; restaurer contexte
MOV     R12, #Treated
MOV     R11, #Write0
STR     R11,[R12]
BEQ     restauration
```

J.3. Banc de test générateur de requêtes

Voici le banc de test permettant de simuler une interface maître. D'abord, la structure des éléments du tableau utilisé est données, puis le code VHDL est fournit.

Tableau J.1. Structure du tableau du le banc de test

	# envoie vci_addr vci_be vci_wdata vci_eop vci_cmd val_att int_clear int_enable int_disable vci_rdata vci_reop vci_rerror ack_att
	1 => (16#0065#,16#0000#, "1111" , 16#0000#,16#0000#, '0',"01",100, 0, 0, 0, 16#00AA#,16#AA00#, '0', '0', 4),

4.5.3.1. Code VHDL d'un module du banc de test

```
--processus permettant de faire les requêtes VCI (vérifcation de l'interface esclave)
   envoie_p : process
          variable position_envoie : integer;
   begin
          -- initialisation
          Targ_vci_cmdval <= '0';
          Targ_vci_addr <= (others => '0');
          Targ_vci_be <= (others => '0');
          Targ_vci_cmd<=(others => '0');
          Targ_vci_eop<='0';
          Targ_vci_wdata<=(others => '0');
          fin_interruption <= '1';

          wait until(nrst = '1');
          -- après un coup d'horloge après la restauration, les
         -- requête commencent à être envoyée
          position_envoie := 1;
          wait until (vci_clk'event and vci_clk = '1');
```

```
 -- la boucle est traversée tant que tous les envoies du tableaux
-- ne sont pas faits
 envoie_loop : while position_envoie <= MAX_ENVOIE loop

-- selon l'envoie l'attente est plus ou moins longue
    -- avant d'envoyer le val
    if(vci_sig(position_envoie).VAL_ATT > 0) then
            attente_val : for i in 1 to vci_sig(position_envoie).VAL_ATT loop
               Targ_vci_cmdval <= '0';
               wait until (vci_clk'event and vci_clk = '1');
            end loop attente_val;
    end if;

    -- communication avec le module de réception pour
   -- indiquer une interruption effacée
    if(vci_sig(position_envoie).INT_CLEAR = 1) then
            fin_interruption <= '1';
    else
            fin_interruption <= '0';
    end if;

    Targ_vci_cmdval <= '1';
   -- à partir des données du tableaux, les signaux de requête sont identifiés


   -- il est impossible de représenter un signal de 32 bits en hexadécimal et donc
   -- deux signaux hexadécimaux de 16 bits sont raboutés pour obtenir l'adresse
   -- de 32 bits
    Targ_vci_addr <=   CONV_LONG_INTEGER(vci_sig(position_envoie).ADRESSE2,
                                                  vci_sig(position_envoie).ADRESSE1);
    Targ_vci_be <= vci_sig(position_envoie).ENABLE;
    Targ_vci_cmd<= vci_sig(position_envoie).CMD;
    Targ_vci_eop<= vci_sig(position_envoie).EOP;
    Targ_vci_wdata<=   CONV_LONG_INTEGER(vci_sig(position_envoie).WDATA2,
                                                   vci_sig(position_envoie).WDATA1);

 -- une note est affichée pour indiquer le numero de l'envoie
assert false report "Envoie "&to_string(position_envoie,3)&" done" severity NOTE ;

 --message pour interruption vers extérieur par MB
 assert (not(vci_sig(position_envoie).INT_ENABLE and interruption = '1')) report
     "Bonne generation d'interruption : "&to_string(position_envoie,3) severity NOTE ;
 assert (vci_sig(position_envoie).INT_ENABLE and interruption = '1') report "Erreur de
      generation d'interruption : "&to_string(position_envoie,3) severity ERROR ;
 assert (not(vci_sig(position_envoie).INT_DISABLE and interruption = '0')) report
     "Bonne clear d'interruption : "&to_string(position_envoie,3) severity NOTE ;
 assert (vci_sig(position_envoie).INT_DISABLE and interruption = '0') report "Erreur
```

```
            du clear d'interruption : "&to_string(position_envoie,3) severity ERROR ;

        -- augmentation du numero de l'envoie
            position_envoie := position_envoie + 1;

        -- attente du coup d'hologe suivant
            wait until (vci_clk'event and vci_clk = '1');
        -- attente de réception du ack avant de poursuivre
            attente_ack: while (Targ_vci_cmdack = '0') loop
                    wait until (vci_clk'event and vci_clk = '1');
            end loop attente_ack;

        end loop envoie_loop;

        -- remise des signaux `s 0
        Targ_vci_cmdval <= '0';
        Targ_vci_addr <= (others => '0');
        Targ_vci_be <= (others => '0');
        Targ_vci_cmd<=(others => '0');
        Targ_vci_eop<='0';
        Targ_vci_wdata<=(others => '0');
        wait;
end process envoie_p;


-- processus permettant de verifier la valeur des sorties
reponse_p : process
        variable position_reponse : integer;
begin

        Targ_vci_rspack <= '0';

        wait until(nrst = '1');
        position_reponse := 0;

        reponse_loop : while position_reponse < MAX_ENVOIE loop

      -- on attend d'abord au moins une montee d'horloge
            wait until (vci_clk'event and vci_clk = '1');
            wait until (int_clk'event and int_clk = '1');
            wait until (int_clk'event and int_clk = '1');
            -- on continue a attendre tant que l'on a pas le val
            attente_val : while (Targ_vci_rspval = '0') loop
                    wait until (vci_clk'event and vci_clk = '1');
                    wait until (int_clk'event and int_clk = '1');
                    wait until (int_clk'event and int_clk = '1');
            end loop attente_val;

            position_reponse := position_reponse + 1;

      -- celon le numero de la reponse, on attend un certain nombre de cycles
            -- avant de genere le ack
```

```
            if(vci_sig(position_reponse).ACK_ATT > 0) then
                attente_ack : for i in 1 to vci_sig(position_reponse).ACK_ATT loop
                    Targ_vci_rspack <= '0';
                    wait until (vci_clk'event and vci_clk = '1');
                    wait until (int_clk'event and int_clk = '1');
                    wait until (int_clk'event and int_clk = '1');
                end loop attente_ack;
            end if;

            --on peut envoyer le ack
            assert (Targ_vci_rspval = '1') report " Problem with the rspval generation" severity ERROR;
            Targ_vci_rspack <= '1';

            -- on genere des message d'erreur si mauvaises reponses
                                                        assert          (Targ_vci_rdata          =
CONV_LONG_INTEGER(vci_sig(position_reponse).RDATA2,vci_sig(position_reponse).RDATA1))
                                        report "Read data erreur a l'envoie : "&to_string(position_reponse,3)
severity ERROR ;
            assert (Targ_vci_reop = vci_sig(position_reponse).REOP)
                                                    report "End of packet data erreur a l'envoie :
"&to_string(position_reponse,3) severity ERROR ;
            assert (Targ_vci_rerror = vci_sig(position_reponse).RERROR)
                                                        report "Error generation erreur a l'envoie :
"&to_string(position_reponse,3) severity ERROR ;

            -- on genere une note si on n'a pas d'erreur de reponses mais
            -- mais seulement pour la donnees
                                                        assert          (Targ_vci_rdata          /=
CONV_LONG_INTEGER(vci_sig(position_reponse).RDATA2,vci_sig(position_reponse).RDATA1))
                                        report "Good read data a l'envoie : "&to_string(position_reponse,3)
severity NOTE ;
        end loop  reponse_loop;

        -- remet les signaux a 0
        Targ_vci_rspack <= '0';
        wait;
  end process reponse_p;
```