UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

MÉMOIRE PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN GÉNIE ÉLECTRIQUE

PAR

KAMAL ELSANKARY

DÉVELOPEMENT DE GESTIONNAIRES DE PERIPHERIQUES GÉNÉRIQUES À HAUT DÉBIT

Décembre 2001

REMERCIEMENTS

A l'issue de ce travail, je tiens à exprimer toute ma gratitude à l'ensemble des personnes qui ont contribué, chacune à leur manière, à l'accomplissement de ce mémoire.

Je tiens tout d'abord à exprimer mes profonds remerciements à mon directeur de Maîtrise, professeur à l'UQTR, Monsieur Ahmed Chériti pour l'aide compétente qu'il m'a apportée, pour sa patience et son encouragement. Son oeil critique m'a été très précieux pour structurer le travail et pour améliorer la qualité des différentes sections. Ces mots de remerciements s'adressent également à mon professeur et co-directeur Monsieur El Mostapha Aboualhamid, professeur à l'université de Montréal, pour ses recommandations qui ont significativement contribué à l'avancement de mes travaux de recherche et d'avoir accepté de corriger mon mémoire.

Mes remerciements vont aussi à la compagnie DAYNSTY et spécialement à Monsieur Kamal Sakkay, MSc, pour le soutien financier et son excellence collaboration.

Je tiens à remercier également tous les professeurs de l'UQTR du département de Génie Électrique, et spécialement Messieurs Adam Skorek et Mohamed Benslima, qui nous ont aidés à s'adapter avec le système d'étude au Québec durant notre première session de maîtrise, et pour avoir accepter à corriger mon mémoire, et Monsieur Daniel Massicotte qui nous a orienté à présenter d'une bonne manière nos travaux de recherche dans le cours de Séminaire.

Je remercie également ma famille pour leur soutien moral, ainsi que mes amis qui, de près comme de loin m'ont aidé et encouragé aux moments opportuns. D'autres

personnes m'ont encouragé à la réalisation de ce mémoire par des gestes d'amitié dont je suis reconnaissant. Je ne citerai pas de noms ici, pour ne pas en oublier certains.

Avant de terminer, je voudrais dédier ce mémoire à l'esprit de mon père que je lui souhaite la miséricorde divine, et une spéciale dédicace à ma mère et à toute ma chère famille pour leur encouragement tout au long de mes études. Enfin, j'exprime toute ma gratitude à mes amis qui ont partagé au quotidien mes espoirs et mes inquiétudes, qui m'ont réconforté dans les moments difficiles et avec qui j'ai partagé d'inoubliables instants de détente et à ceux qui m'ont côtoyé quotidiennement pour une longue période pendant mes travaux : Salma Ait Fares ,Walid Ghie. Je vous remercie tous chaleureusement.

RÉSUMÉ

L'architecture ouverte des ordinateurs basée sur l'extension des bus internes avec des bus externes est l'une des raisons principales de son succès. Par ailleurs, pour qu'une application, résidante sur les bus d'extension, puisse communiquer avec le système, un gestionnaire de périphérique, compatible avec ce type de bus, est indispensable pour contrôler le transfert des informations.

Suite aux limitations des fonctionnalités des gestionnaires de périphériques existants, vis-à-vis ces bus d 'extension, au niveau de conception et l 'apparition de ressources technologiques modernes nécessitent l'apparition des gestionnaires périphériques génériques.

Le présent travail propose une architecture des gestionnaires de périphérique(s) générique(s) d'un bus d'extension type PCI et PCI-X qui peut être utilisée par n'importe quel concepteur de cartes d'interfaçage, et d'être adaptés avec différentes applications, à savoir l'interface d'une carte de communication, l'interface d'un co-processeur avec le microprocesseur central et la mémoire centrale ou toutes autres cartes connectées au bus. De plus, ces gestionnaires génériques doivent être compatibles avec tout type de librairie cible de l'intégration et résoudre les problèmes de limitation des gestionnaires existants.

Le développement de ces gestionnaires est fait par le langage de description du matériel VHDL, et pour assurer la conformité des gestionnaires au comportement fonctionnel exigé et pour satisfaire toutes les contraintes imposées au design, l'environnement de l'utilisation de ces gestionnaires est simulé par la création des testbenches. Les résultats

des performances des gestionnaires proposés ont été évalués sur une structure logique reconfigurable (Field Programmable Gates Array) XC2V250FG256 de la compagnie Xilinix. L'évaluation des performances obtenues après synthèse à l'aide des outils Synplicity, nous a permis d'atteindre des fréquences d'opérations de 59MHz, 67.1 MHz et 72.2 MHz pour des gestionnaires type PCI avec fonctionnalité Cible/Maître, Cible et Maître respectivement, une fréquence d'opérations 51.5 MHz pour un gestionnaire type PCI/PCIX avec fonctionnalité Cible/Maître et une fréquence d'opérations 78 MHz pour un gestionnaire type PCIX avec fonctionnalité Cible/Maître.

Enfin, nous avons proposé comment améliorer les fréquences d'opérations en adaptant le code VHDL de ces gestionnaires avec les technologies cibles de l'intégration.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

ABREVIATIONS

ADB	Allowable Disconnect Boundary
AGP	Accelerated Graphics Port
ASIC	Application-Specific Integrated Circuit
CAO	Conception Assistée par Ordinateurs
CLB	Configurable Logic Block
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
DUV	Design Under Verification
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GTL	Gunning Transceiver Logic
HDL	Hardware Description Language
IEEE	Institute of Electrical and Electronics Engineers
IOB	Input Output Blocks
IP	Intellectual Property
ISA	Industry Standard Architecture
ISP	In System Programmable
LAN	Local Area Network
LPT	Line Print Terminal
LUT	Look Up Table
MSI	Message Interrupt Signal
PC	Personal Computer
PCI	Peripheral Component Interconnect
PCI-X	PCI eXtension
PIP	Programmable Interconnect Points
PLD	Programmable Logic Device
PnP	Plug and Play
RAM	Random Access Memory

SIG	Special Interest Group
SoC	System-on-a-Chip
TLC	Tool Command Language
USB	Universal Serial Bus
VHDL	VLSI Hardware Description Language
VLSI	Very Large Scale Integration circuit

Chapitre 1

Introduction

La flexibilité qu'offrent les ordinateurs (Personal Computer PC), grâce à leur capacité d'interfaçage et leur adaptabilité à différentes classes de cartes et de périphériques, représente la principale raison de leurs succès.

Aujourd'hui, un système PC moderne est constitué essentiellement d'une collection de composantes internes et externes, interconnectées par une série de lignes électriques de communications qui transmettent l'information entre un élément d'entrée à un élément de sortie. Ces lignes, nommées ' bus', relient les composantes internes et les périphériques externes du PC à l'unité centrale de traitement (Central Processing Unit) et à la mémoire centrale (Random Access Memory) du PC. Cette adaptabilité représente le concept principal de l'architecture ouverte du PC, basée sur l'utilisation d'un bus d'extension (expansion bus) qui facilite la connexion des composantes supplémentaires et les éléments principaux du PC.

Pour qu'une application, résidante sur un bus d'extension, puisse communiquer avec le système, elle a besoin d'un gestionnaire de périphérique compatible avec ce type de bus pour contrôler le transfert des informations.

Le gestionnaire de périphérique est une composante qui établit la communication entre les applications (Audio, Vidéo, Graphique, LAN, Coprocesseur..) et l'unité centrale de traitement.

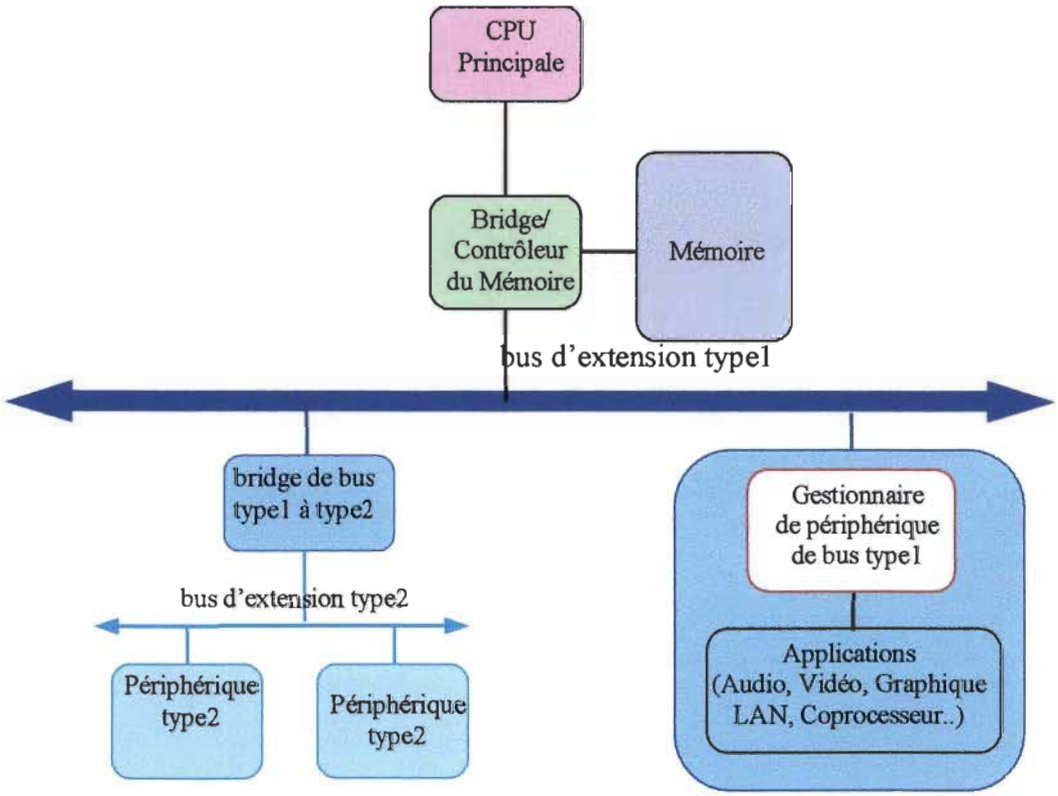

Figure 1.1 : Architecture interne d'un PC et l'emplacement d'un gestionnaire de périphérique

La figure 1.1 montre l'architecture interne d'un système informatique qui illustre l'emplacement d'un gestionnaire de périphérique. Une application raccordée au bus type 1, incorporée dans le système (Embbeded) ou ajoutée par une carte d'extension (add in card), doit être compatible avec les normes spécifiques au bus type 1 pour qu'elle

puisse échanger les informations avec le CPU ou toutes autres applications existantes dans le système.

Cette compatibilité est alors assurée par le gestionnaire de périphérique type bus 1 qui a pour rôles de [1],[2],[3]:

- Assurer la configuration du périphérique dans le système en stockant des informations sur le type d'application, dont ses ressources nécessaires seront réservées, par la suite, par le système.
- Permettre la communication entre des applications à vitesses différentes, en utilisant des signaux pour synchroniser le transfert des informations ou en jouant le rôle d'une mémoire tampon entre les applications.
- Isoler les interfaces de l'application et celles du CPU pour séparer leur évolution. Ce qui permet à la même application d'être appropriée à différents systèmes.
- Permettre d'inclure plusieurs applications dans la même carte d'extension, en assurant une logique interne sur cette carte pour l'arbitrage entre les différentes applications.
- Permettre un fonctionnement indépendant à chaque application dans le système en assignant à cette application le temps où elle peut accéder au bus.
- Donner à l'application, au temps opportun, l'ordre de transfert pour qu'elle ne perde pas l'information. Par exemple, pour éviter l'écrasement de l'information d'une carte de réception, le gestionnaire assure que les données soient lues avant la réception d'une nouvelle information.
- Signaler les erreurs au système. Par exemple, en cas d'une erreur due au calcul de la parité de l'information, le gestionnaire va signaler cette erreur au système.
- Minimiser le temps de flottement de la tension du bus, en connectant ce dernier à une valeur fixe de tension, afin de réduire la consommation de la puissance.

1.1. Problématique de recherche

La caractéristique générique d'un gestionnaire de périphérique peut être constatée à plusieurs niveaux de sa conception. Cette caractéristique répond aux différentes exigences suivantes :

1. La flexibilité de l'interfaçage du gestionnaire qui doit avoir la possibilité de communiquer avec chaque application choisie utilisant le bus.
2. La possibilité d'adapter sa configuration à une grande gamme d'applications.
3. La possibilité de son intégration dans une puce séparée ou dans la même puce de l'application arrière de la carte.
4. Son indépendance de la technologie cible de l'intégration. Il faut que le gestionnaire développé au niveau de la programmation 'software' assure la portabilité de son intégration avec son application arrière dans n'importe quelle technologie cible (FPGA, CPLD ou ASCI) utilisée par le constructeur de l'application.
5. La dotation du gestionnaire d'options versatiles programmables. Il faut que le gestionnaire profite bien de la spécification du bus d'extension, en offrant à l'utilisateur le choix des options convenables à son application.
6. Le gestionnaire générique doit avoir plusieurs fonctions indépendantes au niveau des ressources ainsi des espaces de configurations. Cette caractéristique est indispensable pour les cartes d'extensions pour les trois raisons suivantes :
   i. la vitesse énorme des bus modernes justifie le partage de cette vitesse entre plusieurs applications. Prenons l'exemple d'une carte type PCI-X d'un débit de 1.7Gbytes/sec avec une application arrière représentée par un contrôleur d'Ethernet rapide (Fast Ethernet Controller) de vitesse de 100Mbits/sec (12.5Mbytes/sec). Ce contrôleur est formé de deux FIFOs (First In First Out) une FIFO de réception et une deuxième de transmission de 32 octets chacune. Avec PCI-X de vitesse de 133MHz donc à laquelle correspond une période de 7.5nsec et un transfert de 64 bits (8 bytes). Avec le PCI-X, le transfert de 64 bytes nécessite : (64/8)*7.5=60 nsec. Ayant un débit de 12.5Mbytes/sec, le contrôleur doit accéder au bus après chaque 3µsec, pour une durée de transfert de 60nsec, pour transférer les 64 octets et ce pour ne pas écraser les données. D'après ce calcul, on a montré que le bus reste en état de repos pour une durée de 2.930µsec dans chaque période de 3µsec. Le pourcentage d'utilisation de bus qui est de 0.02 % est très médiocre, d'où l'intérêt de partager cette vitesse énorme entre plusieurs applications, ainsi on profitera de toute la performance du bus.

ii. la plate-forme PCI-X doit avoir un bridge pour pouvoir utiliser plusieurs types des cartes pour les composantes à faible débit, comme dans le cas d'un bus ISA[4]. Cependant, une plate-forme complètement de type PCI-X permet d'utiliser une seule carte multifonctions pour toutes les applications à faible vitesse utilisées au bus ISA, ainsi le nombre des bus d'extensions est réduit, par conséquent on réduit le coût du système.

iii. la technologie moderne permet actuellement la réalisation de ce type de gestionnaire, but qui était difficile avant. les composantes programmables FPGA [5] et CPLD [6] de 10Millions de portes par cm^2, disponibles actuellement, permettent l'intégration de plusieurs composantes sur une même carte ceci était possible avec les ASIC [7] mais la réalisation était d'une part beaucoup plus chère, et d'autre part la solution ASIC n'inclut pas l'avantage principal du circuit programmable ce qui est la caractéristique ISP (In System Programmable)[8] qui réside dans la possibilité de la programmation quand le système est branché et après la construction de la carte. Donc avec le FPGA et le CPLD la carte d'extension sera constituée, comme la figure 1.2, d'un circuit programmable et de simples circuits analogiques de régulation de tension et de composantes dont l'intégration dans les circuits programmables s'est avérée difficile jusqu'à présent. L'attribution au gestionnaire générique de la caractéristique multifonction sera assurée par un changement d'une nouvelle architecture dans le circuit programmable au lieu de changer la carte d'extension.

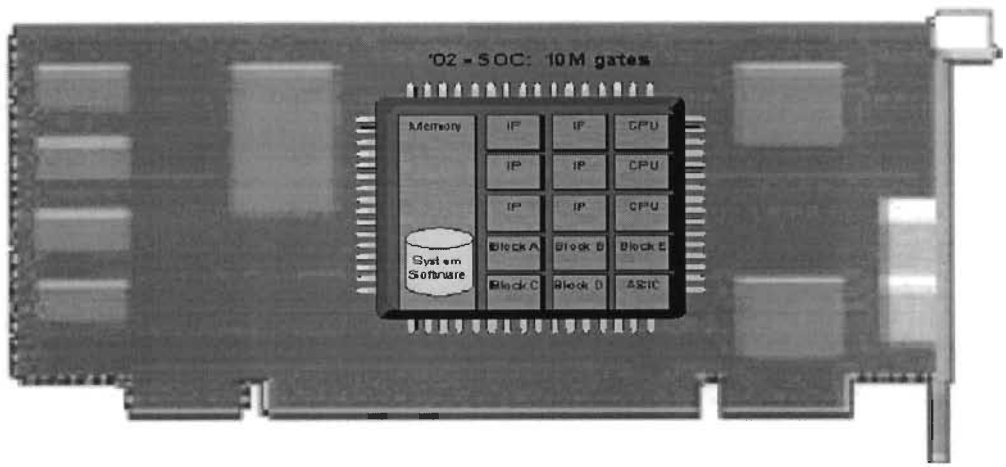

Figure 1.2 : Forme prévue de la carte d'extension

Ces trois raisons avec les autres exigences déjà cités justifient l'appellation générique d'un gestionnaire.

La nécessité d 'un gestionnaire générique pour les applications modernes et que le PCI est le bus le plus répandu parmi tous les autres types de bus existants [9] et qu'avec son extension PCI-X il assure sa dominance dans le marché des cartes d'extension [10], justifie le choix du PCI/PCI-X comme application de développement d'un gestionnaire générique en proposant une architecture qui assure la caractéristique générique de PCI/PCI-X sur différents niveaux de généralités du gestionnaire qu'on a discutés ci-dessus.

Plusieurs circuits intégrés commerciaux d'interface de type PCI ont été développés[11]. Cependant, ils imposent souvent des contraintes sur le développement de l'application et aussi sur les cartes qui vont l'inclure. En plus, en utilisant un FPGA ou un CPLD pour l'intégration de la fonctionnalité du PCI et de l'application principale, une seule puce est utilisée, ce qui n'est pas possible en cas d'utilisation de circuits intégrés préfabriqués (deux puces et plus).

Également le gestionnaire de type PCI est disponible comme noyau 'core' chez plusieurs constructeurs [12], mais, dans ces noyaux plusieurs omissions par rapport à la spécification du PCI peuvent être rencontrées. De plus ils dépendent d'une certaine libraire imposée par les constructeurs, et ne sont pas portables d'un constructeur à un autre, et ne profitent pas de toutes les options du PCI/PCI-X comme support des multifonctions sur la même carte, le décodage programmable et la fonctionnalité bridge.

Parmi les articles qui traitent le développement de gestionnaires de périphérique type PCI, les articles [12], [13] , [14] et [15] portent sur le développement d'une cible type PCI 32 bits. Les gestionnaires développés dans ces articles ne sont pas génériques à plusieurs niveaux. Ils ne peuvent être configurés que comme des cibles de 32 bits, leurs interfaçages avec leurs applications arrières sont fixes donc un re-design de leurs

architectures est nécessaire pour leur utilisation avec d'autres applications, et enfin des limitations et rencontrées concernant pour les options supportées par ces gestionnaires.

1.2. Objectifs

Ce travail a pour but de développer des gestionnaires de périphériques type PCI (Peripheral Component Interconnect) et PCI-X (PCI-eXtension) à usage général à l'aide de langages de description matériel tels que VHDL. Par ailleurs la technologie VLSI (FPGA) sera utile à l'étape de la synthèse. Les gestionnaires de type PCI développés fonctionnent à 32/64 bits et ont des fréquences situées entre 33 et 66 MHz. Ceux de type PCI-X fonctionnent à 32/64 bits et ont des fréquences situées entre 66 et 100 MHz.

L'architecture proposée répond à la caractéristique générique de ces gestionnaires, donc ces dernières peuvent être utilisées par n'importe quel concepteur de cartes d'interfaçage. Ils peuvent s'adapter avec différentes applications comme l'interface d'une carte Ethernet ou d'un co-processeur avec le microprocesseur central et la mémoire centrale ou toutes autres cartes connectées au bus, et peuvent être utilisés pour n'importe quel type de librairie cible et viennent résoudre les problèmes de limitation des gestionnaires existantes.

1.3. Méthodologie de recherche

À la suite d'une recherche bibliographique sur les différents types de bus d'extensions, en particulier le PCI et le PCI-X et leurs architectures internes, ainsi que sur les différents langages de description de matériel et leurs librairies cibles de l'intégration, une architecture générique des gestionnaires de périphériques type PCI et PCI-X est proposée. Le développement de ces gestionnaires est fait par le langage de description matériel VHDL, et pour assurer la conformité des gestionnaires au comportement fonctionnel exigé et pour satisfaire toutes les contraintes imposées au design, l'environnement de l'utilisation de ces gestionnaires est simulé par la création des

testbenches. L'évaluation de la performance de ces gestionnaires est mise en valeur via les résultats de l'intégration par la technologie FPGA de Xilinx.

1.4. Structure du rapport

Ce rapport de mémoire est constitué de quatre chapitres. Le premier chapitre sera consacré à expliciter la problématique de recherche, les objectifs ainsi que la méthodologie suivie tout au long de la réalisation de ce travail.

Dans le second chapitre décrit, l'architecture de bus PCI et les différents éléments de ce bus à développer, également les améliorations de bus PCI-X par rapport au bus PCI. Le troisième chapitre portera sur l'architecture proposée des gestionnaires et sur sa modélisation. Dans ce chapitre sont décrits, l'environnement proposé pour que les gestionnaires soient génériques, le développement des différentes unités de gestionnaire ainsi que la simulation et les résultats de l'intégration des gestionnaires développés. Une conclusion générale viendra clore le rapport.

Chapitre 2

L'architecture des bus PCI et PCI-X

Une bonne compréhension du principe de fonctionnement des bus PCI et PCI-X est primordiale et c'est que nous tenterons de faire dans les pages suivantes.
Ce chapitre présente donc une description détaillée de bus PCI : ses types de gestionnaires ainsi que leur architecture globale, ses caractéristiques électriques, ses mécanismes de transactions et on terminera par les améliorations et les modifications apportées au bus PCI-X par rapport au bus PCI [3], [4],[16], [17] et [18].

2.1. Gestionnaires type PCI

Il existe quatre types de gestionnaire de type PCI dont leurs descriptions sont données ci-après.

2.1.1. Bridge 'HOST' et Chipset PCI

Le bridge qui existe entre l'unité de traitement du PC (CPU+CACHE+RAM) et le bus PCI s'appelle 'bridge host'(voir figure 2.1). Il est actuellement intégré dans le circuit jeu de puces communément nommé 'Chipset PCI'. Ce circuit regroupe la gestion

du bus PCI comme arbitrage, la fréquence de l'horloge, et le contrôle d'échange de données entre l'unité de traitement et le bus PCI.

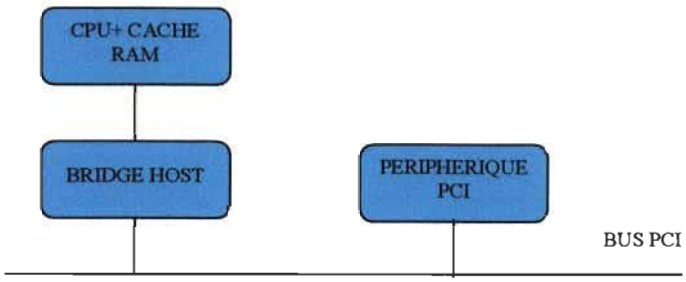

Figure 2.1 : Architecture interne d'un PC qui illustre l'emplacement du bridge host type PCI

2.1.2. Maître PCI

Le Maître PCI est un dispositif qui initialise le transfert de l'information en demandant la permission de l'accès au bus de la bridge host. Une fois la permission est obtenue, ce maître exécute le transfert avec d'autres dispositifs. Le mécanisme d'arrêt de transfert est contrôlé par une logique d'arbitrage qui partage le bus entre les applications selon ses besoins. L'effet de limiter l'accès au bus pour une durée précise pour les maîtres encourage ces derniers à ne pas insérer des états d'attentes, ce qui améliore le débit réel sur le bus.

2.1.3. Cible PCI

La Cible PCI est un dispositif capable d'identifier si une transaction donnée est dirigée vers elle et de répondre aux données correspondantes à cette demande. Si la cible identifie que la transaction n'est pas orientée vers elle, la transaction est ignorée. Notons que, tous les gestionnaires doivent être capables de fonctionner comme cible, mais pas nécessairement comme maître. Cependant, un gestionnaire peut contenir des interfaces de cible et de maître.

2.1.4. Bridge PCI

Le Bridge PCI est un dispositif qui fournit un chemin de raccordement entre deux bus indépendants de type PCI. Il permet à des transactions de se produire entre un maître

connecté à un bus PCI et une cible connectée à un autre bus PCI. Par ailleurs, il fournit aussi, aux concepteurs de systèmes et de cartes d'extensions, la capacité de surmonter les limites des charges électriques en créant la hiérarchie de bus PCI.

2.2. Architecture globale des gestionnaires type PCI

Dans cette section on s'intéressera de présenter les éléments principaux d'un gestionnaire type PCI notamment les fonctionnalités de ses principaux signaux, ses commandes, ses espaces d'adressage et de configuration.

2.2.1 Signaux de gestionnaire type PCI

L'interface d'un gestionnaire type PCI présente 124 signaux physiques pour manipuler les données et l'adressage (voir figure 2.2). Parmi ces signaux, 47 sont obligatoires pour une cible et 49 pour un maître.

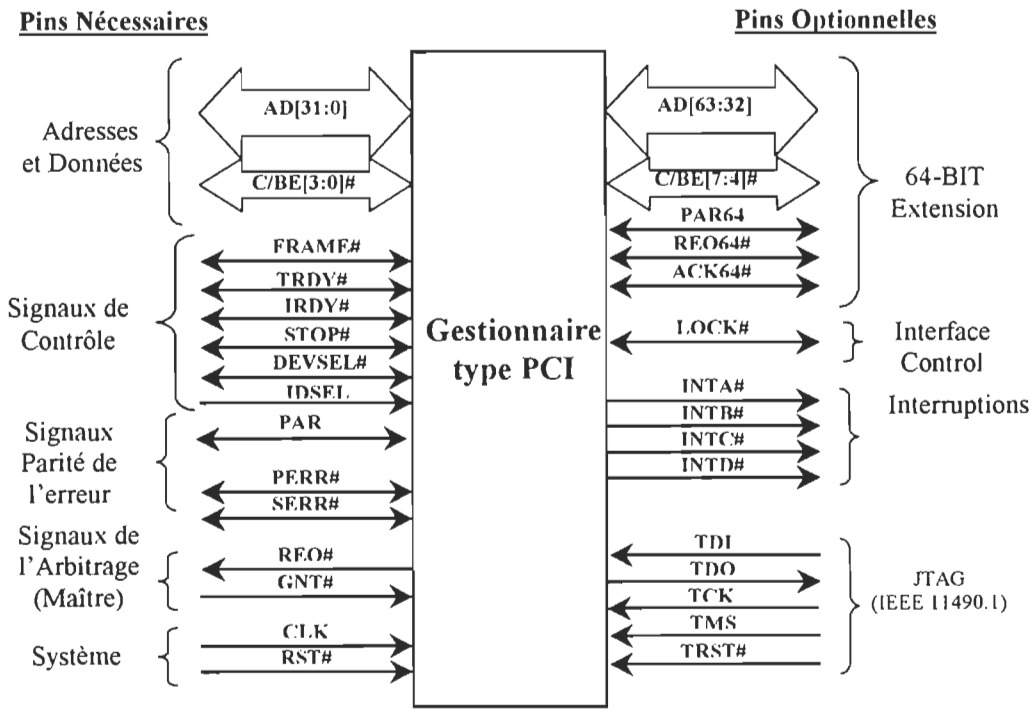

Figure 2.2 : Interface d'un gestionnaire type PCI

Il est à noter qu'il y a cinq différents types des signaux utilisées dans ce type de gestionnaire :

in signal d'entrée ordinaire (Input)

out signal de sortie ordinaire (Output)

t/s signal bidirectionnel à 3 états (Tri-state) permet aux plusieurs gestionnaires d'être connectés à ce type des signaux simultanément, mais à condition que lorsqu'un gestionnaire commande ces signaux, tous les autres gestionnaires, connectés à ces signaux; doivent mettre ces derniers en haute impédance.

o/d signal émetteur ouvert (Open Drain) permet aux plusieurs gestionnaires de le partager en permanence. Cependant le pin de type 'Open Drain' a 2 états : état 0 utilisé pour l'activer afin de signaler une erreur par exemple, et dans ce mode le pin absorbe le courant, et état de la haute impédance qui est l'état inactif. Notons que un pin type o/d n'est jamais une source de courant. Par ailleurs, des résistances de tirage vers le haut (pull-ups) maintiennent l'état inactif du signal jusqu'à ce qu'un gestionnaire connecté au bus prenne le contrôle.

s/t/s signal trois états soutenus (Sustained tri-state) permet d'améliorer la performance et la consommation de la puissance du système. Après son état active, ce type de signal est maintenu désactive pendant une durée au moins une période de l'horloge avant d'être mis en haute impédance. Donc à ce fait, il réduit le temps d'état flottant d'où la diminution de la dissipation de l'énergie.

Ces signaux principaux sont récapitulés au tableau 2.1 avec une courte description sur leur fonctionnement. 2.2.2. Commandes de bus PCI

Pendant la phase de données, les signaux de commande et de bytes valides multiplexés **C/BE[3:0]#** sont employés comme bytes valides pour indiquer les signaux de bus **AD[31:0]** qui portent les données valides. Notons qu'il est possible de transférer des données à travers certaines ou tous les signaux de bus **AD**. Pendant la phase d'adresse d'une transaction de bus PCI, les signaux **CBE[0:3]#** contiennent la commande, indiquant à la cible le type de la transaction demandée par le maître.

Il y a trois types de commande standards :

1) Les accès de configuration :

- *Configuration Read* : utilisée pour lire l'espace de configuration d'un gestionnaire.

- *Configuration Write* : utilisée pour configurer un gestionnaire en écrivant dans son espace de configuration.

Tableau 2.1 : Signaux principaux de bus PCI

Signal	Type	description courte
AD[31:0]	t/s	Le bus d'Adresses / Données multiplexés sur les mêmes pins de PCI bus, fournissent l'adresse pendant la phase d'adresse et les données pendant les phases de données.
C/BE[3:0]#	t/s	Command/Bytes Eanbles. Pendant la phase d'adresse d'une transaction, ils définissent la commande de bus. Pendant les phases de données, ils indiquent les bytes valides.
CLK	in	Horloge système de bus PCI.
DEVSEL#	s/t/s	Device Select, indique que le gestionnaire a décodé son adresse comme cible de l'accès courant.
FRAME#	s/t/s	Cycle Frame, activé par le maître de la transaction courante pour indiquer le commencement et la durée d'un accès au bus.
GNT	t/s	Grant, indique à un maître que la propriété de bus lui a été accordée (utilisé pour les maîtres seulement).
IDSEL	in	Initialisation Device Select, utilisé comme "chip select" pendant les transactions des configurations d'écriture ou de lecture.
IRDY#	s/t/s	Initiator Ready, le maître est prêt à transférer des données.
TRDY#	s/t/s	Target Ready, La cible est prête à transférer des données.
PAR	t/s	Parity, Un signal de parité utilisé pour assurer la parité égale à travers les lignes AD[31:0] et C/BE[3:0]#.
PERR#	s/t/s	Parity Error, pour indiquer les erreurs de parité de données pendant toutes les transactions de bus PCI.
SERR#	o/d	System Error, Une erreur de parité s'est produite pendant une phase d'adresse ou pendant un cycle de commande spéciale. En outre, il est utilisé pour signaler des erreurs autres que celle de la parité.
REQ#	out	Request, indique à l'arbitre que cet agent désire l'utilisation de bus.
RST#	in	Reset
STOP#	s/t/s	Stop, indique que la cible courante demande au maître pour arrêter la transaction courante.
AD[63:32]	t/s	Address/Data Bus, 32 bits additionnels multiplexés pour l'extension 64-bits.
C/BE[7:4]	t/s	Bus Command/Byte Enable pour l'extension à 64-bit.
PAR64	t/s	Parity Upper DWORD, est la parité pour les signaux AD[63:32] et C/BE[7:4].
REQ64#	s/t/s	Request 64-bit Transfer, indique que le maître demande le transfert de 64-bits.
ACK64#	s/t/s	Acknowledge 64-bit Transfer, indique que la cible est prête pour le transfert de données de 64-bits.
INTA,B,C, D#	o/d	Interrupt A,B,C et D, ces signaux sont employés pour demander un service d'interruption du système hôte.

Note: Le nom de signal terminé avec le suffixe # est actif à l'état bas.

2) Les accès au E/S :

- *IO Read* : utilisée pour lire les données d'un gestionnaire implanté dans l'espace Entrée / Sortie du système.
- *IO Write* : utilisée pour écrire les données d'un gestionnaire implanté dans l'espace Entrée / Sortie du système.

3) Les accès à la Mémoire :

- *Memory Read* : utilisée pour lire les données d'un gestionnaire implanté dans l'espace Mémoire du système;
- *Memory Write* : utilisée pour écrire les données d'un gestionnaire implanté dans l'espace Mémoire du système;
- *Memory Read Multiple* : utilisée pour lire plusieurs lignes de cache d'un gestionnaire implanté dans l'espace Mémoire du système. Signalons que, la ligne de cache consiste d'un registre implanté dans les gestionnaires type PCI qui est configuré par le système pour indiquer la taille d'un accès de transaction sur la mémoire cache du PC. Cependant, la performance du système est améliorée en utilisant cette commande à la place d'une commande de lecture de mémoire ordinaire car le système peut anticiper la recherche de données avant que les anciennes données soient consommées par leur destination finale;
- *Memory Read Line* : utilisée pour lire une ligne de cache entière d'un gestionnaire implanté dans l'espace Mémoire du système. Cependant, la performance du système est améliorée en utilisant cette commande à la place d'une commande de lecture de mémoire ordinaire car le système peut anticiper la recherche de données avant que les anciennes données soient consommées par leur destination finale;
- *Memory Write and Invalidate* (*MWI*) : utilisée pour écrire une ou multiple lignes du cache dans un gestionnaire implanté dans l'espace mémoire. Cette commande facilite la cohérence entre la mémoire cache et l'espace mémoire accédé par cette commande. En effet; dans le cas de modification d'une ligne complète de mémoire, la cohérence est maintenue entre les deux mémoires par l'invalidation de cette ligne dans la mémoire cache. En contraste, dans le cas d'accès d'une ligne incomplète l'invalidation de lignes complètes du mémoire cache ramène à la perte de donnée valide.

En outre à ces commandes standards, on rencontre trois autres commandes qui ont des fonctionnalités particulières:

- *Special Cycle* : utilisée pour diffuser des messages à tous les gestionnaires connectés au bus;

- *Interrupt Acknowledge*: c'est une lecture implicitement adressée au contrôleur d'interruption de système;
- *Dual Address Cycle* : utilisée pour transférer une adresse de 64 bits sur les lignes **AD** de 32 bits. Signalons que ce type d'adressage est employé pour l'adressage des gestionnaires implantés dans l'espace mémoire supérieur à 4 Gigaoctets.

Les codes de commandes de bus sont indiqués dans le tableau 2.2.

Tableau 2.2 : Codes de commandes de bus PCI

CBE#[0:3]	Command Type
0000	Interrupt Acknowledge
0001	Special Cycle
0010	I/O Read
0011	I/O Write
0100	Réservé
0101	Réservé
0110	Memory Read
0111	Memory write
1000	Réservé
1001	Réservé
1010	Configuration Read
1011	Configuration Write
1100	Memory Read Multiple
1101	Dual Address Cycle
1110	Memory Read Line
1111	Memory Write and Invalid

2.2.3. Décodage d'adresse sur le bus PCI

Le PCI définit trois types des espaces d'adressages : L'espace de configuration qui est défini pour la configuration de matériel type PCI, l'espace mémoire et l'espace d'E/S. Ces deux derniers sont employés par les dispositifs raccordés sur le bus pour le transfert réel des données. Le décodage d'adresse sur le PCI est distribué d'une manière que chaque cible est responsable à décoder sa propre adresse. Ainsi le bus PCI soutient deux arrangements de décodage d'adresse: décodage positif, dont une cible cherche une adresse dans sa gamme assignée du système, et décodage soustractif, dont une seule cible accepte toutes les adresses qui n'ont pas été encore décodées par les autres dispositifs. Avec l'un de ces deux arrangements, une cible indique qu'elle a décodé son adresse et réclame ainsi la transaction en affirmant le signal de trois états **DEVSEL#**. Notons que

pour le décodage positif **DEVSEL#** est piloté une, deux ou trois horloges après la phase d'adresse pour le décodage rapide, moyen ou lent respectivement. D'autre part, pour le décodage soustractif **DEVESEL#** est piloté cinq horloges après la phase d'adresse. Par ailleurs, chaque cible indique la temporisation utilisée du **DEVSEL#** dans son registre d'état dans son espace de configuration dont on verra ultérieurement.

2.2.4. Espace de configuration

Pour l'auto-configuration (Plug and Play, prêt-à-tourner) et l'initialisation totalement contrôlée par le pilote (Driver) d'un gestionnaire, un espace d'adresse séparé est réservé : l'espace de configuration. Ainsi les gestionnaires de type PCI sont exigés à assigner 256 bytes des registres de configuration à ce but. Cet espace stocke des informations sur les ressources requises ou optionnellement disponibles, ainsi il a une partie prédéfinie comprend les 64 premiers bytes et une partie, dépendant du gestionnaire, comprend les 192 bytes restants. Le tableau 2.3 décrit les premiers 64 bytes pour les gestionnaires autres que les ponts. Notons que, ce type de l'espace de configuration est nommé type 00h.

Tableau 2.3 : Espace de configuration type 00h

31 24	23 16	15 8	7 0	
DeviceID		*VendorID*		00h
StatusRegister		*Command*		04h
Class Code			*Revision ID*	08h
BIST	*Header Type*	**Latency Timer**	**Cache Line Size**	0Ch
Base Address Register 0				10h
Base Address Register 1				14h
Base Address Register 2				18h
Base Address Register 3				1Ch
Base Address Register 4				20h
Base Address Register 5				24h
CardBUS CIS Pointer				28h
Subsystem ID		*Subsystem Vendor ID*		2Ch
Expansion ROM Base Address				30h
Reserved			**Capabilities Pointer**	34h
Reserved				38h
Max-Lat	**Min-Gnt**	**Interrupt Pin**	**Interrupt Line**	3Ch

Les registres en italique sont des registres obligatoires à implanter, le reste de l'espace de configuration n'est pas nécessaire d'être implanté physiquement mais une valeur lue de zéro doit être produite lors les accès de lecture.

En outre, les accès à l'espace de configuration exigent le décodage de l'adresse accédée avec l'activation du pin **IDSEL**, qui agit en tant que "chip-select" unique pour chaque gestionnaire.

Il est à noter que l'espace de configuration ne doit impliquer aucune contrainte sur l'emplacement de l'adresse ou sur les lignes d'interruptions, cependant le système principal doit pouvoir localiser librement le gestionnaire de PCI dans ses ressources.

Dans ce qui suit, on présente les fonctionnalités des principaux registres de l'espace de configuration :

- **Vendor ID** : identifie le fabricant du gestionnaire, assigné par PCI SIG (Special Interest Group).
- **Device ID** : identifie le type du gestionnaire particulier assigné par le fournisseur.
- **Revision ID** : identifie la révision assignée par le fournisseur.
- **Class Code** : identifie la fonction générique de gestionnaire.
- **Command Register** : registre de commande, sa forme est présentée au tableau 2.4.

Tableau 2.4 : Registre de commande

Bit	Fonction
0	**IO Space**. Quand ce bit est mis à 1, le gestionnaire répond aux accès dans l'espace E/S.
1	**Memory Space**. Quand ce bit est mis à 1, le gestionnaire répond aux accès dans l'espace mémoire.
2	**Bus Master**. Quand ce bit est mis à 1, le gestionnaire acte comme un maître.
3	**Special Cycles**. Quand ce bit est mis à 1, le gestionnaire est permis de détecté les cycles spéciaux de bus.
4	**Memory Write and Invalidate Enable.** Quand ce bit est mis à 1, le gestionnaire est peut génère la commande MWI.
5	**VGA Palette Snoop.** Mise à 1 informe le gestionnaire compatible à VGA de mise à jour les registres de palette des couleurs
6	**Parity Error Response.** Mise à 1 indique que le gestionnaire est capable de signaler l'erreur de parités via PERR#
7	**Stepping Control.** Il indique si le gestionnaire est permis d'effectuer commande progressive adresses / données
8	**SERR# Enable.** Mise à 1 indique que le gestionnaire est capable de commander SERR#
9	**Fast Back-to-Back Enable.** Ce bit indique si le maître est capable de faire accès multiples.
15 :10	**Réservés**

- **Status Register** : registre d'état, sa forme est présentée au tableau 2.5.

Tableau 2.5 : Registre d'état

Bit	R/W	Fonction
3:0	R	**Réservés.** Câblées à 0.
4	R	**Capabilities List.** Il indique que le registre de Capabilities Pointer est implanté.
5	R	**66MHz-Capable.** Mise à 1 indique que le gestionnaire est capable de fonctionner à 66MHz
6	R	**Réservé.**
7	R	**Fast Back-to-Back Capable.** Indique la capacité de gestionnaire d'effectuer des accès multiples au bus.
8	R/W	**Master Data Parity Error**. Ce bit est mis à 1 par le maître qui a signalé l'erreur au pin PERR# au bus ou il l'a reçu par une cible sur l'une de ses transactions.
9 :10	R	**Device Select(DEVSEL#)Timing.** Indique la rapidité de la cible pour décoder son adresse. 00b = rapide, 01b = médium, 10b = lente, 11b = réservé
11	R/W	**Signaled Target Abort**. Mis à 1 par la cible quand elle signale Target Abort.
12	R/W	**Received Target Abort**. Mis à 1 par le maître quand elle reçoit Target Abort
13	R/W	**Received Master Abort**. Mis à 1 par le maître quand une de ses transactions termine avec Master Abort
14	R/W	**Signaled System Error.** Indique que le gestionnaire a signalé erreur au système via SERR#.
15	R/W	**Detected Parity Error**. Indique que le gestionnaire à détecter une erreur des parités

- **Base Addresses** : les registres d'Adressages.

Il faut noter que l'une des fonctions la plus importante pour permettre une configuration supérieure et une facilité d'utilisation est la possibilité de placer librement par le système les gestionnaires PCI dans les espaces d'adressage. Par ailleurs, sur la mise sous tension, le système doit configurer automatiquement les espaces d'adressage mutuellement exclusifs pour chaque gestionnaire dans l'espace mémoire ou l'espace d'E/S. Afin d'assurer cette configuration, le système doit pouvoir détecter le type et la taille de l'espace à allouer pour chaque gestionnaire. Par conséquent, les registres de base (*BARs*) sont employés pour informer le système sur la taille de l'adresse requise de chaque application.

Le fonctionnement du registre *Base Address* est montré dans le tableau 2.6.

Tableau 2.6 : Registre Base Address

Bit	Fonction
0	1 pour adresse dans l'espace E/S, 0 pour adresse dans l'espace mémoire
2:1	00 pour implanter dans l'espace d'adresse de 32 bits, 01 et 11 réservé, 10 pour implanter dans l'espace d'adresse de 64 bits
3	Mis à 1 dans le cas de la mémoire avec possibilité de pré-lecture (Prefechable Memory)
32:4	Il dépend de la taille de l'espace d'adressage du gestionnaire. Par exemple un gestionnaire qui veut posséder une espace d'adresse de 1 MB de mémoire doit établir les 12 bits poids fort du registre d'adresse et câbler les autres à 0.

L'espace de configuration consacré aux bridges est nommé type 01h. Cependant, les registres de configurations les plus intéressants pour les bridges et qui sont différents de ceux de la type 00h sont les suivantes :

- **Primary Bus Number Register** : Ce registre est initialisé par le système avec le nombre du bus de bridge qui a une architecture plus proche au processeur central.

- **Secondary Bus Number Register** : Ce registre est initialisé par le système avec le nombre du bus d'extension ajouté par rapport au bridge.

- **Subordinate Bus Number Register** : Ce registre est initialisé par le système avec le nombre du bus d'extension le plus dérivé par rapport au bridge.

- **Primary** et **Secondary Status Register** : ont le même fonctionnement que le *Status Register* décrit pour la configuration type 00h à l'exception que pour le bridge le premier registre décrit l'état du bus primaire et le deuxième décrit celui du bus secondaire du bridge.

- **Memory Limit** et **Memory Base registers** : La base de mémoire et les registres de limite de mémoire sont les registres exigés qui définissent une plage d'adresse implantée dans l'espace mémoire de l'E/S qui est employée par le bridge pour déterminer le temps d'expédition des transactions de mémoire d'une interface à l'autre.

Notons que les 12 bits poids fort des registres *Memory Base* et *Memory Limit* sont de lecture/écriture et correspondent aux 12 bits d'adresse poids fort assignés au bridge.

Cependant, afin de décoder les adresses, le bridge suppose que les 20 bits poids faible de registre *Memory Base* sont mis à 00000h zéros. Également, il suppose que les 20 bits poids faible du registre *Memory Limit* sont ainsi mis à FFFFFh. Le minimal de la plage d'adresse de mémoire définie sera aligné sur la borne de 1 MB.

La figure 2.3 montre un exemple d'un système PCI avec des valeurs assignées aux registres concernant les numéros de bus et les limites des espaces mémoires réservés par le système.

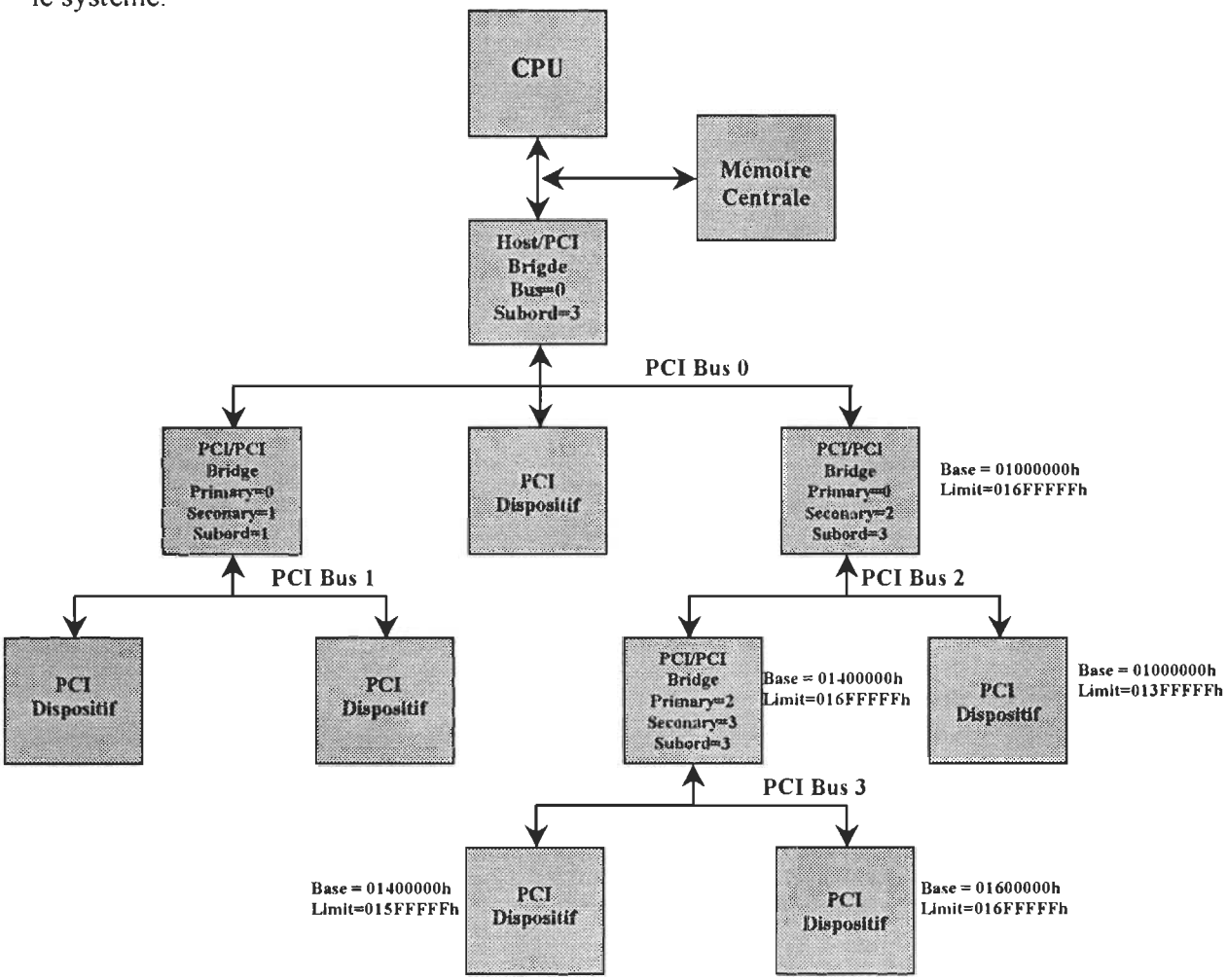

Figure 2.3 : Initialisation des registres concernant le numéro de bus et l'espace mémoire pour chaque dispositif connecté au bus PCI

- **Capability pointer** : pointeur d'une nouvelle capacité.

Ce registre facultatif est utilisé pour se diriger à une liste de nouvelles capacités mises en application par un gestionnaire. D'ailleurs ce registre est seulement valide si le bit de liste

de possibilités dans le registre de statut est mis à 1. Cependant, si ce registre est implanté, les deux bits inférieurs sont réservés et devraient être placés à 00b.
La forme générale des registres d'une nouvelle capacité est montrée dans le tableau 2.7

Tableau 2.7 : Registres d'une nouvelle capacité

31 — 16	15 — 8	7 — 0	
Feature-specific	Pointer to Next Capability	Capability ID	Dword0
Configuration Registers			Dword1 ... Dwordn

- Capabiltiy ID : l'identité de la nouvelle capacité définit par le SIG de PCI par exemple 02h dans le cas d'une carte avec capacité AGP (Accelerated Graphic Port).

- Pointer to Next Capability : ce registre contient un pointeur vers une nouvelle capacité si elle existe où il est câblé à 00h dans le cas contraire.

- Feature-specific Configuration Registers : Contient des caractéristiques spécifiques de cette nouvelle capacité, par exemple les registres d'état et de commande pour nouvelle capacité type AGP sont montrés dans le tableau 2.8.

Tableau 2.8 : Registres d'une nouvelle capacité type AGP

31 — 24	23 — 20	19 — 16	15 — 8	7 — 0	
Reserved	Major Rev	Minor Rev	Pointer to Next Capability	02h	Dword0
AGP Status Register					Dword1
AGP Command Register					Dword2

2.3. Caractéristiques électriques

Quelques caractéristiques s'avèrent nécessaires et intéressantes de connaître pour le fonctionnement du bus PCI :

- Tension d'alimentation fonctionne en 5 volts entre 4,75 et 5,25 V et aussi en 3,3 volts entre 3,0 et 3,6 V. Le tableau 2.9 présente les valeurs maximales et minimales pour lesquelles la tension se comporte à l'état haut ou bas;

- Capacité d'entrée maximale par pin est de 10 pF;
- La limite de longueur de trace (sur une carte de PCI) est de 1,5 pouces (2,5 pouces pour le signal CLK);
- Biais d'horloge (clock skew) maximum permit est de 2 ns à 33 MHz, 1ns à 66 MHz;
- T_slew (temps de monté / descende des signaux à 33 MHz) est entre 1 à 5 V/ns;
- T_prop (temps maximal de propagation dans le bus à 33 MHz est de 10ns, 5ns à 66 MHz.

Tableau 2.9 : Limites maximales et minimales de la tension de fonctionnement dans le bus PCI

Symbole	Paramètre	5 volt env.	3.3 volt env.
VIH	Input High Voltage (min)	2.0 V	1.65 V
VIL	Input Low Voltage (max)	.8 V	1 V
VOH	Output High Voltage (min)	2.4 V	3 V
VOL	Output Low Voltage (max)	.55 V	.3 V

2.4. Transactions sur le bus PCI

Il faut noter que dans les transactions sur le bus PCI, on rencontre des transactions des données normales et anormales, des transactions de configuration, des transactions pour la signalisation de parité et des erreurs, des transactions pour la fermeture d'une cible et pour la génération des interruptions et des transactions pour l'arbitrage entre les maîtres connectés au bus PCI.

2.4.1. Transactions normales des données

Le mécanisme de base de transfert sur le bus PCI est celui de transfert en mode continu (Burst) composé d'une phase d'adresse et d'une ou plusieurs phases de données. La figure 2.4 montre une transaction sur le bus PCI pour le mode de transfert continu.
Le premier cycle de la transaction est la phase d'adresse et l'autre cycle(s) est la phase de données (selon le mode de transfert de données choisit continu ou simple). La transaction commence par la phase d'adresse qui se produit quand **FRAME#** est activé (active bas) pour la première fois. Pendant cette phase d'adresse, le bus AD contient une adresse

valide et le bus **C/BE#** contient une commande de bus valide. Cependant, pour le transfert de la configuration, un signal supplémentaire **IDSEL** est affirmé (active haut) durant la phase d'adresse pour sélectionner le gestionnaire à configurer.

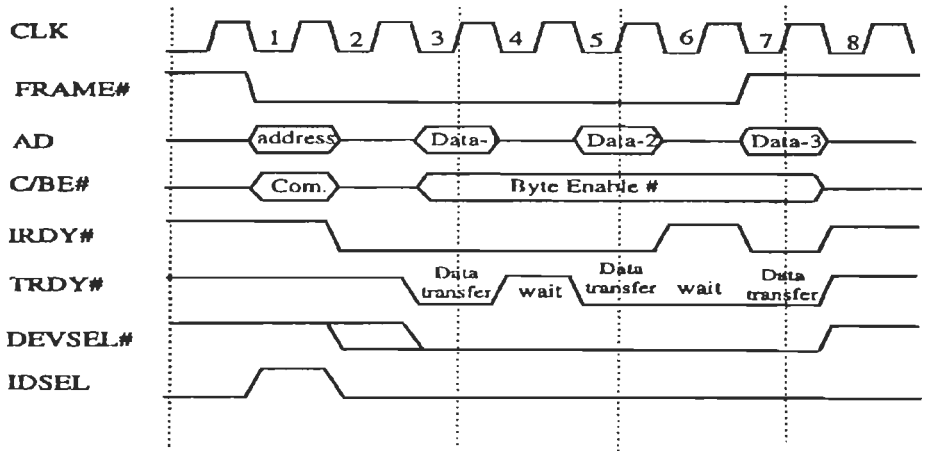

Figure 2.4 : Transfert des données dans le bus PCI

Après la phase d'adresse, le maître affirme **IDRY#** par la suite pour indiquer qu'il est prêt à recevoir les données (pour une transaction de lecture) ou pour indiquer que le bus **AD** contient des données valides (pour une transaction d'écriture). Par ailleurs la cible répond en affirmant les signaux **TRDY#** et **DEVSEL#** en indiquant que le bus **AD** contient des données valides ou que la cible est prête à recevoir des données pour une transaction de lecture ou d'écriture respectivement. Pendant les phases des données le bus **C/BE#** est utilisé pour indiquer les bytes valides **BE#** parmi les bytes transférés dans le bus **AD**. Le signal **DEVSEL#** est affirmé jusqu'à la fin de la transaction. Notons que, le transfert de données se produit seulement si **IRDY#** et **TRDY#** sont affirmés, le cycle d'attente peut être inséré dans la phase de données par le maître ou par la cible quand **IRDY#** ou **TRDY#** est désactivé (cycles 4 à 6 dans la figure 2.4). Le dernier cycle se produit quand **IRDY#** et **TRDY#** sont affirmés et **FRAME#** est désactivé (cycle 7).

Il est important de signaler que le gestionnaire type PCI peut accéder à la transaction selon son mode d'opération. Si ce gestionnaire est une cible, il recevra les signaux **FRAME#**, **IDSEL#**, **IRDY#**, l'adresse **AD**, les commandes et les bytes valides en **C/BE#**. Si le gestionnaire de PCI est un maître, il doit demander l'autorisation de bus de l'arbitre (seulement un maître peut commander le bus de PCI à un moment donné) en

activant **REQ#** et doit attendre jusqu'à ce que l'arbitre lui accorde l'autorité du bus en affirmant le signal **GNT#**, afin d'accéder à la transaction comme maître. Par ailleurs, Le maître envoie les signaux **FRAME#**, **IRDY#** et **C/BE#**, et l'adresse sur **AD**, et reçoit **TRDY#**, **DEVSEL#**. Le transfert de données s'effectue d'une cible à un maître pour une transaction de lecture et d'un maître à une cible pour une transaction d'écriture.

2.4.2 Transactions anormales

Ce sont des transactions normales mais qui sont interrompus par des arrêts lancés par un maître ou une cible lors de leur incapacité de les achever. Tandis que ni le maître ni la cible ne peut réellement arrêter une transaction unilatéralement, cependant le maître contrôle son achèvement final en apportant toutes les transactions à un arrêt ordonnée et systématique indépendamment de ce qui a causé l'arrêt. Toutes les transactions sont conclues quand **FRAME#** et **IRDY#** sont désactivés, en indiquant un état de repos sur le bus.

2.4.2.1. Arrêts lancés par le Maître

Notons que le mécanisme utilisé dans l'arrêt lancé par le maître est quand **FRAME#** est désactivé et **IRDY#** est activé. Cette condition signale à la cible que la phase finale de données est en marche. Le transfert final de données se produit quand **IRDY#** et **TRDY#** sont activés. Les transactions sont achevées quand **FRAME#** et **IRDY#** sont désactivés.
Le maître peut lancer l'arrêt en utilisant ce mécanisme pour une des deux raisons :

L'accomplissement : se rapporte à l'arrêt quand le maître a conclu sa transaction prévue. C'est la raison la plus commune des arrêts (figure 2.4).

Timeout : se rapporte à l'arrêt quand la ligne **GNT#** du maître est désactivée et le temporisateur de latence interne (*Latency Timer*) de celui-ci est expiré (figure2.5). Dans ce cas la transaction prévue n'est pas nécessairement conclue. Le temporisateur a pu

avoir expiré en raison de la latence induite par l'accès à la cible ou l'opération prévue était très longue. Par ailleurs, la commande *Memory Write and Invalidate* n'est pas régie par le temporisateur de latence (*Latency Timer*) excepte qu'aux frontières de ligne de cache.

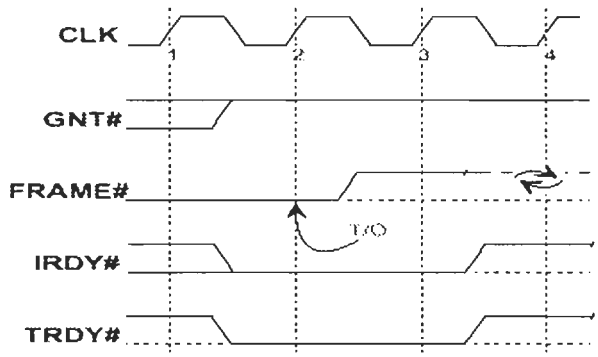

Figure 2.5 : Arrêt dû à un timeout

Il est à noter qu'un maître qui lance une transaction avec cette commande ignore le temporisateur de latence jusqu'à la frontière de ligne de cache. Cependant, quand la transaction atteint une frontière de ligne de cache et le temporisateur de latence a expiré (et **GNT#** est désactivé), le maître doit terminer la transaction.

Master-Abort : un maître détermine qu'il n'y aura aucune réponse lancée par une cible liée à une transaction quand **DEVSEL#** reste désactiver après la sixième horloge de la phase d'adresse, donc c'est après l'expiration du temps de décodage soustractif le plus lent pour **DEVSEL#** (figure 2.6).

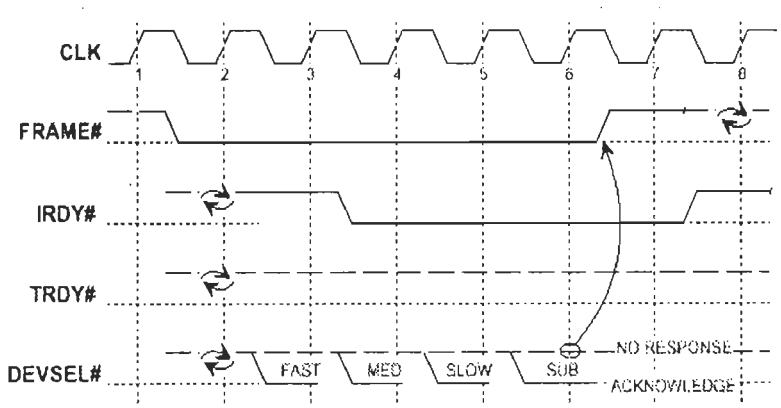

Figure 2.6 : Arrêt dû à Master_Abort

2.4.2.2. Arrêts lancés par la cible

Dans la plupart des conditions, la cible est capable de générer ou de recevoir les données demandées par le maître jusqu'à ce que le maître termine la transaction. Mais, dans le cas qu'elle est incapable d'accomplir la demande, elle peut employer le signal **STOP#** pour lancer l'arrêt de la transaction. Notons que la façon que la cible combine **STOP#** avec d'autres signaux indiquera au maître la condition qui mène à l'arrêt.
Il est à noter que les trois types d'arrêt lancés par la cible sont:

RETRY : se rapporte à l'arrêt demandé avant que n'importe quelle donnée soit transférée du fait que la cible est occupée et temporairement ne peut pas traiter la transaction. Cette condition peut se produire, par exemple, dans le cas où il y a un conflit avec les ressources internes. Cependant, la cible signale *RETRY* en activant **STOP#** et désactivant **TRDY#** dans la phase initiale de la transaction (figure 2.7). Toutefois, si une cible signale *RERTY* à un maître, aucune donnée n'est transférée lors de cette transaction et par conséquent, le maître est obligé de re-essayer le transfert ultérieurement.

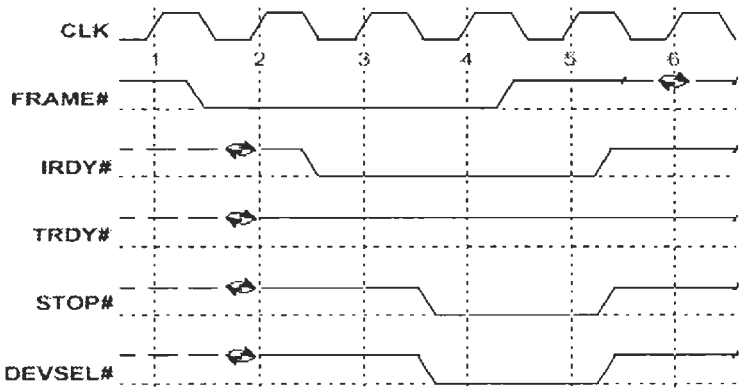

Figure 2.7 : Réponse RETRY par la cible

DISCONNECT : se rapporte à l'arrêt demandé après que la phase initiale des données sera transférée du fait que la cible ne peut pas continuer le transfert. Par exemple le maître veut transférer des données en mode continu alors que la cible est incapable à répondre qu'à une seule phase de donnée.
Il est à noter que la signalisation de *DISCONNECT* diffère d'un *RETRY*. Ce dernier s'est produit toujours sur la phase initiale de données, et aucun transfert de données ne

s'effectue, alors que *DISCONNECT* peut être signalée à n'importe quelle phase de données en activant **STOP#** (figure 2.8).

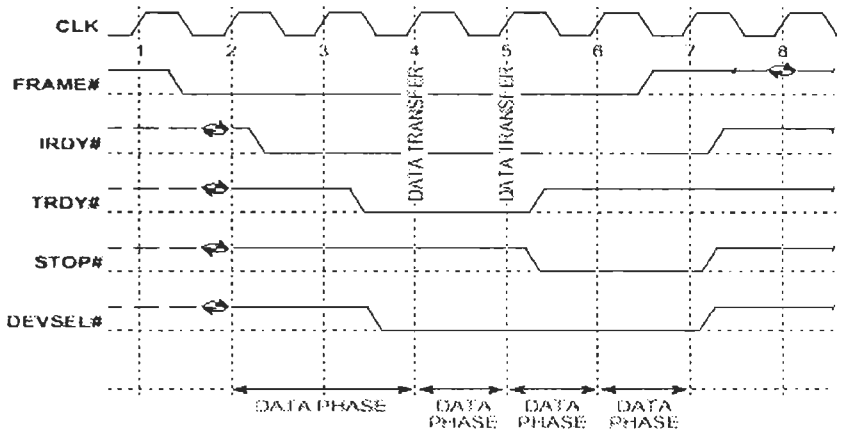

Figure 2.8 : DISCONNECT

Target-Abort : se rapporte à une fin anormale d'une demande de transfert du fait que la cible a détecté une erreur fatale ou elle ne pourra jamais accomplir la demande. Par exemple, un maître demande que tous les bytes dans un espace d'adresse d'E/S d'une cible doivent être lus, mais soit la conception de la cible limite l'accès à des bytes précis dans cette gamme, ou soit le transfert demandé par le maître dépasse la capacité de la cible. Par ailleurs, puisque la cible ne peut jamais accomplir la demande, elle termine le transfert avec *Target-Abort*. Une fois que la cible a réclamé une demande d'accès sur ses ressources en activant **DEVSEL#**, elle peut signaler *Target-Abort* sur n'importe quelle horloge suivante. Notons que la cible signale *Target-Abort* en désactivant **DEVSEL#** et en activant **STOP#** simultanément (figure2.9).

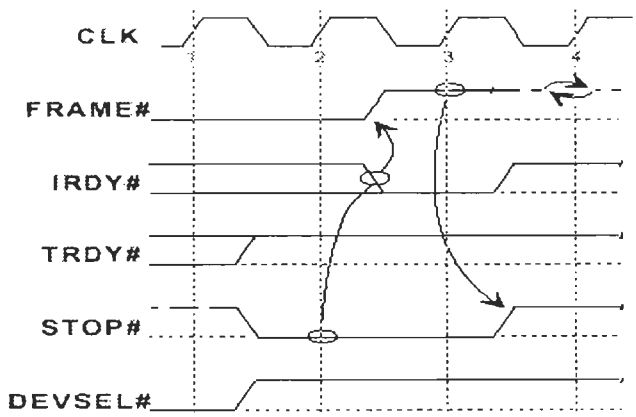

Figure 2.9 : Target_Abort

2.4.3. Transaction de la configuration

En raison de limitation de la charge électrique, le nombre des gestionnaires qui peuvent être supportés sur un bus donné est limité. Pour permettre à des systèmes d'être construit au-delà d'un bus simple, des bridges de type PCI/PCI sont introduits. Signalons qu'un bridge PCI/PCI exige un mécanisme pour savoir comment et quand expédier des accès de configuration aux gestionnaires qui résident derrière lui. Cependant, pour supporter la hiérarchie de bus PCI, deux types de transactions de configuration sont employés. Ils ont les formats illustrés par la figure 2.10, qui montre l'interprétation des lignes **AD** pendant la phase d'adresse d'une transaction de configuration.

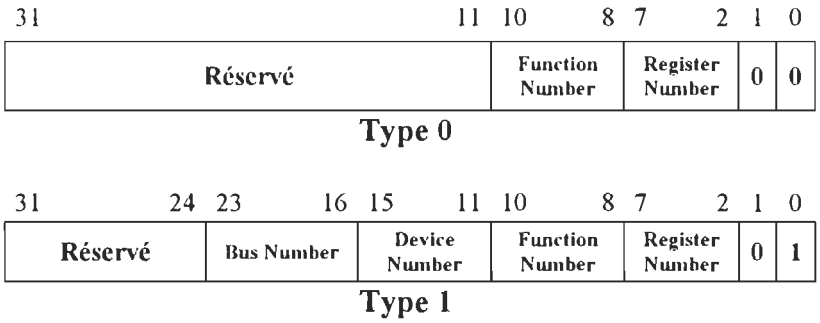

Figure 2.10 : Les deux types de transaction de configuration

Il est à noter que les transactions de configurations type 1 et de type 0 sont différenciées par les valeurs sur **AD[1:0]**. Une transaction de configuration type 0 (quand **AD[1:0]** = "00") est utilisée pour sélectionner un gestionnaire sur le bus où la transaction est exécutée. Une transaction de configuration type 1 (quand **AD[1:0]** ="01") est utilisée pour passer une demande de configuration vers un autre bus de la hiérarchie. Notons que les bridges (host et PCI/PCI) qui ont besoin de générer une transaction de configuration type 0 utilisent le champ *Device Number* pour choisir le gestionnaire correspondant à la transaction en affirmant son **IDSEL**. Le nombre de fonction est fourni sur **AD[10:8]**. Également le nombre de registre fournit sur **AD[7:2]**. **AD[1:0]** doit être"00" pour une transaction de configuration du type 0.

Par ailleurs, un gestionnaire à simple fonction affirme **DEVSEL#** pour réclamer une transaction de configuration quand ces trois conditions sont avérées simultanément :

- Une commande de configuration est décodée;
- Le pin **IDSEL** de ce gestionnaire est affirmé;
- **AD[1:0]** est "00 " (commande de configuration type 0) pendant la phase d'adresse.

Également, si un gestionnaire implante des fonctions multiples indépendantes avec des espaces de configuration indépendantes pour chaque fonction (gestionnaire à multifonction), ce gestionnaire doit affirmer **DEVSEL#** pour réclamer une transaction de configuration quand ces quatre conditions sont avérées simultanément.

- Une commande de configuration est décodée;
- Le pin **IDSEL** de ce gestionnaire est affirmé;
- **AD[1:0]** est "00 ";
- **AD[10:8]** correspond à l'une des ses fonctions implantées.

En outre, tous les types du gestionnaire sauf les bridges PCI/PCI ignorent les transactions de configuration de type 1. Les bridges PCI/PCI décodent le champ *Bus Number* pour déterminer si le bus de destination de la transaction de configuration réside derrière lui. Cependant, si le *Bus Number* n'est pas un bus derrière le bridge, la transaction est ignorée. Si le *Bus Number* n'est pas un bus secondaire du bridge, la transaction est simplement passée par lui sans changement. Et finalement, si le *Bus Number* correspond à un bus secondaire de ce bridge, il convertit la transaction en une transaction de configuration type 0, il change **AD[1:0]** en "00 ", il passe à la suite **AD[10:02]** sans changement et à la fin il lance une transaction de configuration type 0 avec l'affirmation **IDSEL** correspondant au gestionnaire accédé.

2.4.4. Transaction pour la fermeture d'une cible

Un maître donné, sous certaines conditions, peut verrouiller (LOCK) une ressource de système dont il peut l'accéder uniquement. Cette technique de verrouillage est utilisée aux accès à une mémoire partagée ou à une mémoire à portes duales (Dual Port RAM), pour que le maître assure qu'aucun autre maître que lui change sa mémoire de travail lorsqu'il perd l'autorité au bus. Cependant, un maître peut verrouiller une cible en activant le signal **LOCK#** juste après la phase d'adresse. Le signal **LOCK#** est

maintenu activé même après l'achèvement de la transaction. Il faut signaler qu'une cible verrouillée ne peut pas répondre à toutes les transactions tant que **LOCK#** est activé. Par conséquent, le maître de verrouillage ne peut pas accéder à aucune autre cible que celle verrouillée jusqu'à ce qu'il désactive le signal **LOCK#**.

2.4.5. Transaction pour la génération des interruptions

Un gestionnaire type PCI qui veut produire des interruptions pour demander des services de son pilote (Device Driver) peut le faire en utilisant l'une des deux méthodes.

Méthode 1. N'importe quel gestionnaire peut signaler une interruption via un pin d'interruption. Notons que, chaque gestionnaire peut posséder jusqu'à quatre pins d'interruption : **INTA#**, **INTB#**, **INTC#** et **INTD#**, et comme ces pins d'interruption sont de type émetteur ouvert, ils peuvent être partagés par plusieurs gestionnaires simultanément. Or si plusieurs pins d'interruption sont utilisés par un gestionnaire, chaque pin doit correspondre à une fonction différente dans ce gestionnaire. Par ailleurs, toutes les interruptions sont conduites à un contrôleur d'interruption qui signale au système hôte qu'une interruption est exigée. Le système hôte exécute en conséquence la routine d'interruption basée sur le vecteur d'interruption retourné par le contrôleur d'interruption (figure 2.11).

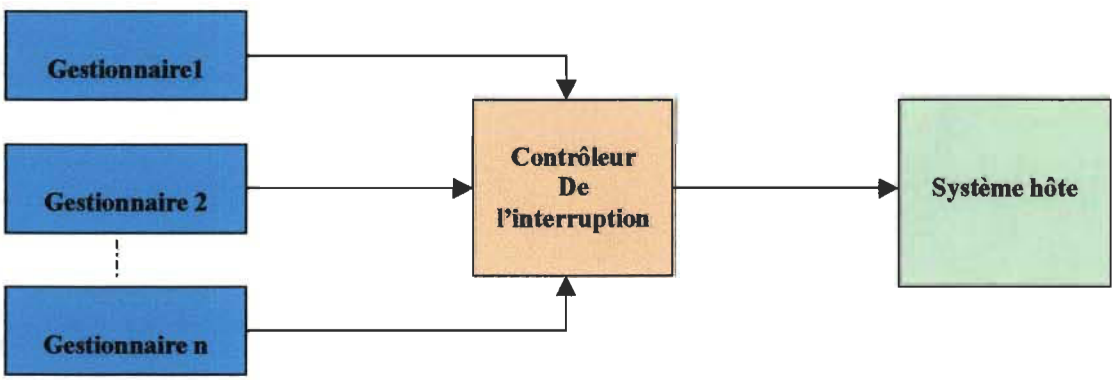

Figure 2.11 : Demande d'une interruption basée sur la méthode 1

Notons que le système hôte contient la trace de l'interruption qui lui permet de déterminer l'identité du gestionnaire signalant l'interruption, en se basant sur le vecteur

d'interruption. Il faut noter que le gestionnaire signalant l'interruption doit automatiquement enlever la demande de l'interruption une fois que le transfert de données requis est complet. Dans le cas où il y aurait une trace d'interruption partagée par plusieurs gestionnaires, le processeur lit des registres spécifiques pour l'interruption dans ces gestionnaires pour savoir celui qui a demandé l'interruption.

Méthode 2. Alternativement, le gestionnaire peut implanter la capacité de générer une interruption comme un message MSI (Message Interrupt Signal). L'interruption par ce message est générée comme une transaction d'écriture sur un emplacement mémoire réservé au gestionnaire qui demande l'interruption. Notons que cette méthode élimine le besoin de trace d'une interruption et élimine également le besoin de partager la même trace de l'interruption par plusieurs gestionnaires sur l'entrée du contrôleur de l'interruption. Par ailleurs, un gestionnaire de PCI indique son soutien de MSI comme nouvelle capacité. L'adresse de cette nouvelle capacité est pointée par le pointeur d'adresse de la nouvelle capacité décrite dans la section 2.2.4 de l'espace de configuration. Les registres de la capacité MSI pour un message à 32 bits d'adresse ont les formats illustrés au tableau 2.10.

Tableau 2.10 :Registres de la capacité MSI

31 — 16	15 — 8	7 — 0	
Message Control Register	**Pointer to Next Capability**	**Capability ID=05h**	Dword0
Message Address Register			Dword1
	Message Data Register		Dword2

- **Capacity ID**: ce registre identifie la capacité MSI et il est câblé à 05h.
- **Pointer to Next ID** : contient un pointer vers une nouvelle capacité s'il existe, et câblé à 00h dans le cas contraire.
- **Message Address Register** : l'adresse allouée par le système pour ce gestionnaire pour l'écrire dans son vecteur d'interruption.
- **Message Data register** : Le message à exécuter par le processeur central quand ce gestionnaire génère son message d'interruption.

- **Message Control Register** : Contrôle l'état de message à exécuter, si c'est un message simple ou multiple.

Signalons que les registres de la capacité MSI pour un message de 64 bits d'adresse ont le même format que ceux de 32 bits avec une simple différence d'avoir en deux registres type Message Address au lieu d'un seul dans le cas de 32 bits.

2.4.6.Transaction pour la signalisation de parité et des erreurs

L'intégrité de données est vérifiée en utilisant un signal de parité : **PAR**. Le signal **PAR** est utilisé pour assurer la parité égale, c. à. d s'il existe un nombre pair de '1' dans les 36 bits formés par les signaux **C/BE #** et **AD**. Si une erreur de parité est détectée, le signal **PERR#** est activé. Si l'erreur de parité se produit pendant la phase d'adresse ou pendant une commande *Special Cycle*, le signal **SERR#** est activé. Également, n'importe quel gestionnaire peut utiliser le signal **SERR#** pour signaler une erreur au système et comme ce signal est de type émetteur ouvert donc il est partagé entre tous les gestionnaires. D'ailleurs, pour que le système sache le gestionnaire signalant l'erreur, il doit consulter les registres d'état de tous les gestionnaires branchés au bus PCI pour savoir le générateur de la signalisation de l'erreur (Polling method).

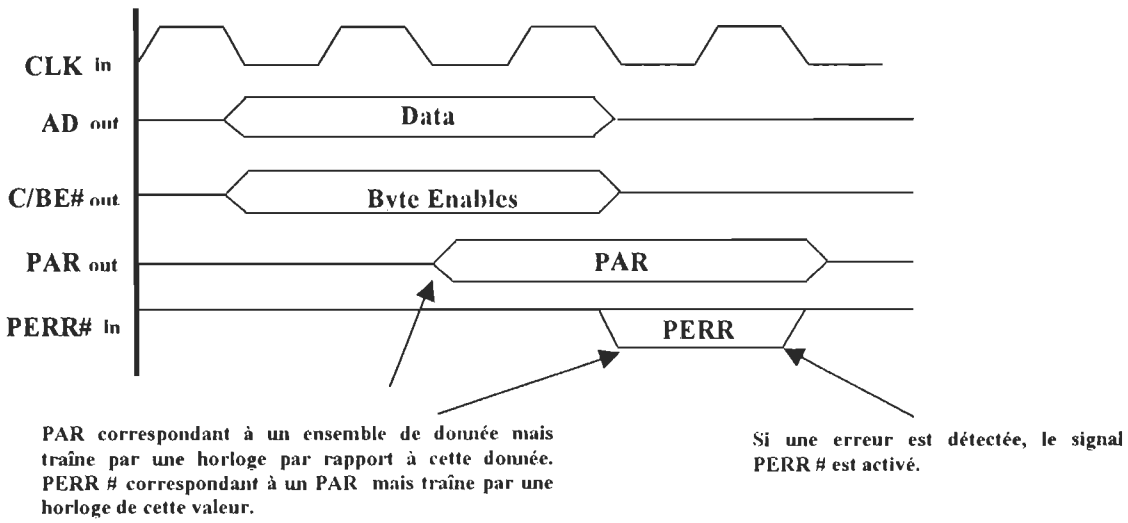

Figure 2.12 : Signalisation de la parité

2.4.7. Transaction de l'arbitrage entre les maîtres connectés au bus PCI

2.4.7.1. Arbitre de bus PCI

À un instant donné, un ou plusieurs maîtres connectés au bus PCI peuvent demander l'utilisation de ce bus pour exécuter un transfert de données avec une autre cible connecté au bus. Cependant, chaque maître demande une transaction en activant sa sortie **REQ#** pour informer l'arbitre de bus de sa demande suspendue pour l'utilisation de bus. La figure 2.13 illustre la relation entre les maîtres de PCI avec la ressource centrale de bus connue sous le nom d'arbitre de bus.

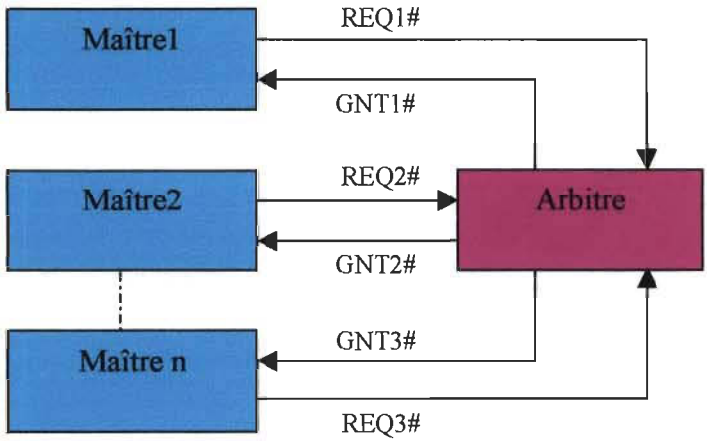

Figure 2.13 : Liaisons entre l'arbitre et les maîtres

Dans cet exemple, il y a trois maîtres possibles reliés à l'arbitre par l'intermédiaire d'une paire séparée de signaux **REQ#/GNT#**. Bien que l'arbitre soit montré comme composant séparé, il est intégré habituellement dans le jeu de puces (chip set) de PCI; et particulièrement, il est typiquement intégré dans le host bridge ou dans le bridge d'extension de bus.

2.4.7.2 Algorithme de l'arbitrage

Les spécifications du bus PCI ne définissent pas l'algorithme de l'arbitrage pour décider le maître de la transaction courante. Quand plusieurs maîtres demandent

simultanément la propriété de bus, l'arbitre peut utiliser n'importe quel algorithme, tel qu'un basé sur une priorité fixe, sur une priorité rotationnelle (Round Robin) ou sur une combinaison de ces deux (rotationnelle parmi un groupe de maîtres et fixe pour un autre groupe). En outre les spécifications déclarent que l'arbitre est requis de mettre en application un algorithme d'équitabilité afin d'éviter le blocage de bus par un maître.
Idéalement, l'arbitre de bus peut déterminer le temps d'accès à assigner à chaque maître connecté au bus par la lecture du contenu du registre de latence maximale de l'espace de configuration (*Max_LAT*) lié à chacun de ces maîtres. Par ailleurs, le concepteur du gestionnaire type maître programme ce registre pour indiquer, dans des incrémentations de 250ns, à quelle rapidité le maître exige l'accès au bus, afin de réaliser une exécution appropriée.
Or, afin d'accorder le bus PCI à un maître, l'arbitre active le signal **GNT#** lié à ce maître. Par conséquent, si le maître produit une demande, et il n'a pas encore lancé une transaction (active **FRAME#**) après 16 périodes de l'horloge du bus PCI qui était à un état de repos, l'arbitre peut supposer que le maître a un mal fonctionnement. Dans ce cas, la décision prise par l'arbitre serait dépendante de la conception du système.

2.5. Améliorations de bus PCI-X par rapport au bus PCI

Le bus PCI-X présente plusieurs améliorations principales par rapport au bus PCI. Ces améliorations permettent au PCI-X d'avoir des débits de transfert et des fréquences plus élevés jusqu'à 1.4 Gbytes/sec et 133 MHz respectivement.
Parmi ces améliorations on rencontre :

1. Le principe de transfert d'un registre à un registre;
2. Des transactions retardées en PCI conventionnel, dues à un *RETRY*, sont remplacées par des transactions divisées en PCI-X;
3. Des nouvelles informations, sous forme d'attribue, sont ajoutées à chaque transaction afin d'aboutir à des arrangements plus efficaces pour le transfert;
4. Des règles restreintes pour l'insertion des d'états d'attentes et pour la signalisation de *DISCONNECT* pour mener à un usage de bus plus efficace.

Dans les paragraphes suivants, ces améliorations vont être discutées.

2.5.1 Principe de transfert d'un registre à un registre

De toute évidence, pour le PCI conventionnel, un pin de sortie d'un gestionnaire est commandé par un registre à chaque front montant de l'horloge PCI, et un pin d'entrée n'a aucun registre pour assurer la rapidité de réponse. Ce principe de l'entrée non-registré présente des problèmes quand la fréquence de bus excède 66 MHz.

En effet; la figure 2.14 présente l'effet d'accroissement de la fréquence versus le temps de synchronisation du design. Dans cette figure l'expéditeur c'est le gestionnaire qui affirme un signal quelconque pour demander une réponse et le récepteur c'est le gestionnaire qui doit décoder le signal de l'expéditeur pour fournir une réponse.

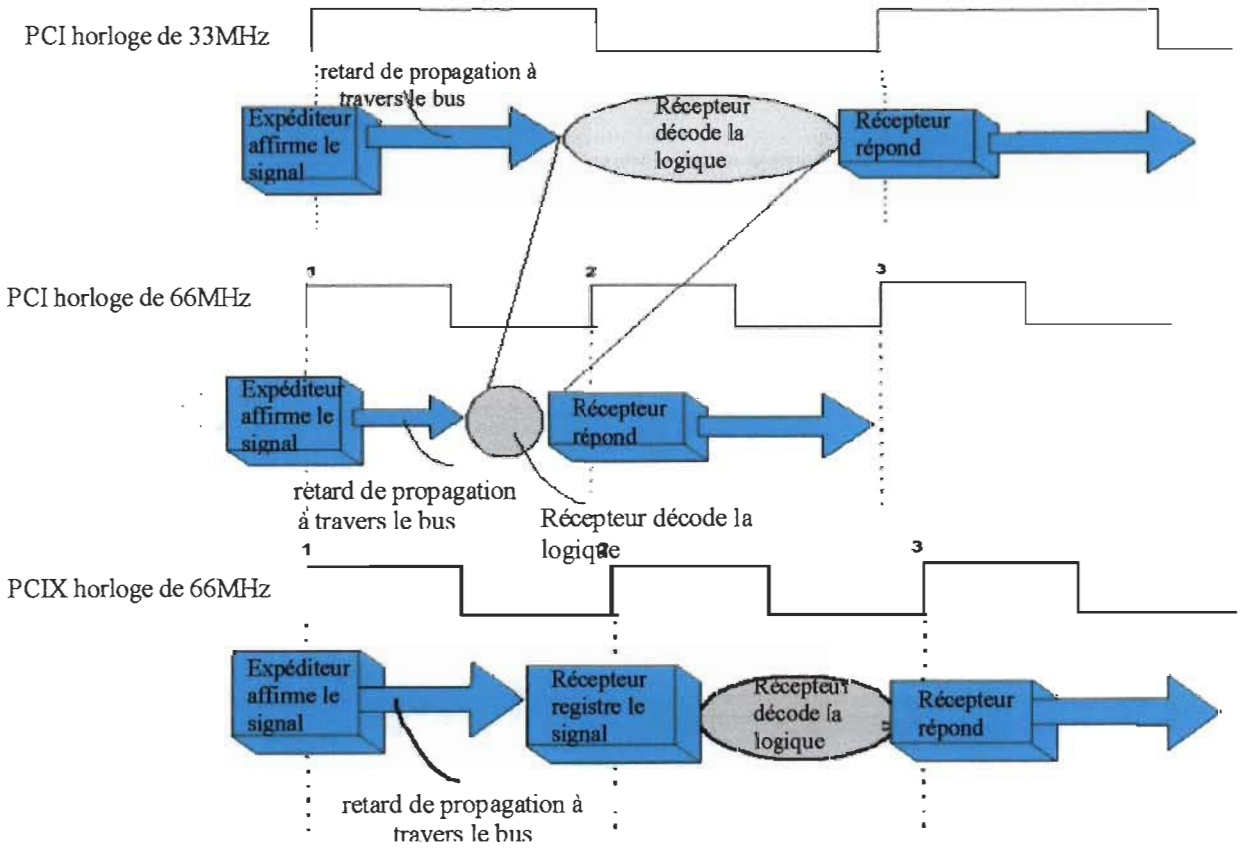

Figure 2.14 : Principe de transfert d'un registre à registre

À la fréquence 33 MHz, une horloge de période 30 nsec, signifie que le récepteur a 30 nsec pour décoder un tel signal et répondre à ce dernier tout en mettant en considération le temps de setup de 7 nsec nécessaire pour le signal pour être stable. Ainsi même après

la soustraction de délai de la propagation sur le bus de 2 nsec et le biais horloge (clock skew) de 1 nsec, on trouve que la logique du récepteur a 20 nsec comme temps de décodage. Donc à cette fréquence (33 MHz), il est facile pour le récepteur de répondre à cette exigence de temps de décodage pour accorder une réponse à l'expéditeur.

Or, à la fréquence 66 MHz, une horloge de période de 15 nsec, signifie que le récepteur a 15 nsec pour décoder un signal émis par l'expéditeur et répondre à ce dernier tout en répondant à l'exigence de temps de setup de 3 nsec, et de soustraction de 2 nsec pour le temps de propagation et de 1 nsec pour le biais de l'horloge. Or la soustraction de ces délais du 15nsec, laisse 9 nsec pour que la logique du récepteur décode le signal. Donc à cette fréquence (66 MHz), Il est difficile à atteindre cette condition de synchronisation, particulièrement pour le décodage de l'adresse. Notons que, la conception du transfert d'un registre à un registre sert comme base pour l'architecture du gestionnaire type PCI-X. En effet, dans cette conception, les signaux des entrées et des sorties du PCI-X sont registrés et sont commandés au front montant de l'horloge. Dans ce concept, un gestionnaire possède un cycle d'horloge complet pour répondre après le verrouillage à ses signaux d'entrées, donc on peut travailler à la fréquence 66 MHz pour un PCI-X avec la même logique de décodage que celle donnée au PCI à la fréquence 33 MHz, du fait qu'on a la même durée de décodage des signaux pour assurer une réponse.

Il faut signaler que l'inconvénient du concept du PCI-X par rapport à celui de PCI est que la réponse de ce dernier sa à un signal à l'instant t sera à l'instant t+1, en revanche, pour le PCI-X à l'instant t+1, le récepteur verrouille les signaux d'entrées et la réponse sera à l'instant t+2, donc on a une latence d'une période d'horloge pour avoir la réponse. Cette latence a l'influence seulement pour la première phase de transfert et à cause de la pipelinage on aura après une réponse à chaque période de l'horloge.

2.5.2. Transactions divisées (Split Transaction)

Notons que, la nouvelle caractéristique la plus significative du PCI-X est l'ajout des transactions divisées (*Split transaction*). Ces transactions ont pour rôle principalement pour substituer les transactions retardées de PCI. Il faut signaler que la transaction retardée du PCI se rapporte à une transaction de lecture, exécutée par un

maître dont lequel la cible termine le transfert par un *RETRY*. En outre, ce maître relance plus tard la transaction, et la cible répond si elle est prête pour faire l'échange des données. Par conséquent, une transaction divisée du PCI-X résout l'inconvénient principal d'une transaction retardée du PCI (terminée par une cible par *RETRY*) qui force le maître à relancer à plusieurs reprises la transaction jusqu'à ce que la cible ait les données, ce qui réduit d'une manière significative la performance du système. Le gestionnaire PCI-X type cible utilise des transactions divisées pour résoudre ce problème dans lequel toutes les transactions doivent être terminées soit immédiatement soit en utilisant le principe de la transaction divisée. Par ailleurs, dans le principe de la transaction divisée, la cible ou le completeur dans la terminologie de PCI-X, termine une transaction en signalant la réponse divisée (*Split Response*), puis cette cible exécute la commande sur son bus intérieur, lance sa propre transaction d'accomplissement (*Split Competion*) pour compléter la transaction et finalement elle envoie les données ou un message d'accomplissement à l'initiateur de la transaction (figure 2.15).

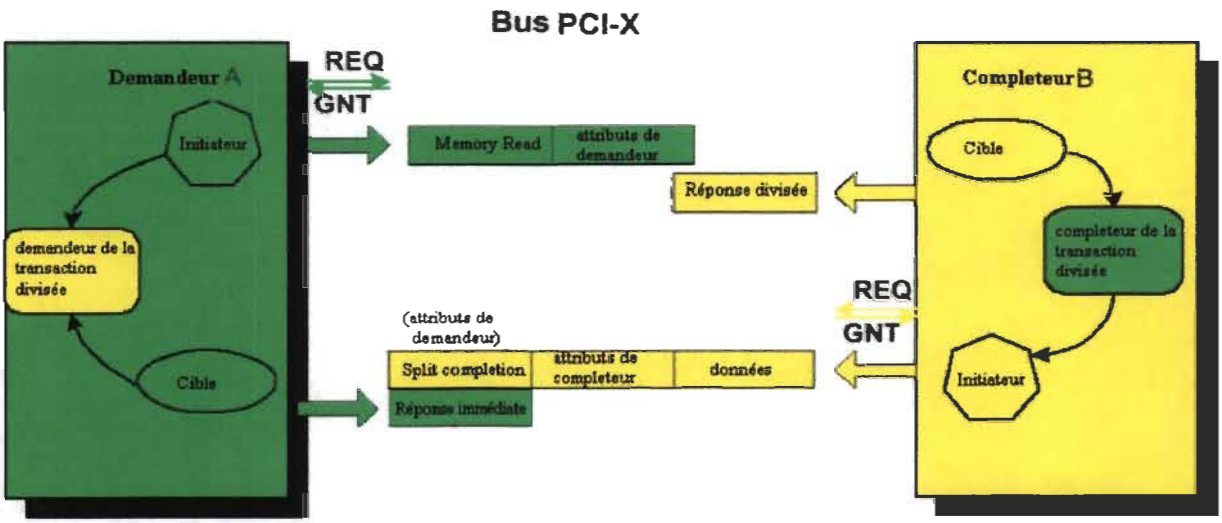

Figure 2.15 : Principe de la transaction divisée (Split Transaction)

Dans la terminologie du PCI-X le nom du maître est changé à l'initiateur car maintenant la cible a la possibilité de maîtriser le bus. Il faut noter que cette approche augmente la complexité des gestionnaires type PCI-X. Ainsi, dans ce principe, il faut ajouter à chaque

transaction des informations concernant l'initiateur de la transaction afin que le completeur puisse acheminer à l'initiateur les données demandées quand il est prêt, d'où l'ajout d'une phase d'attribut à chaque transaction.

2.5.3. Attributs

Les attributs sont des informations supplémentaires incluses dans chaque transaction, et donnant des informations sur l'initiateur de la transaction et sur le type du transfert. Cependant, le maître de chaque transaction commande des attributs sur les bus **C/BE[3:0]#** et **AD[31:00]** dans la phase d'attribut. Notons que la phase d'attribut a toujours une durée d'une horloge et elle vient après la phase d'adresse indépendamment de la longueur de données à transférer ou la longueur de l'adresse (cycle d'adresse simple ou duelle). Les bus **AD[63:32]** et **C/BE[7::4]#**, des gestionnaires qui font le transfert à 64 bits, sont réservés et forcés aux valeurs FFFFFFFFh et FFh respectivement pendant la phase d'attribut.

2.5.4. Nouvelles règles pour les états d'attentes et la déconnexion

Le bus PCI-X améliore le débit du transfert des données en ajoutant des règles pour les états d'attentes et la signalisation de *DISCONNECT*. Il ne permet ni à des cibles ni à des maîtres d'insérer des états d'attente pendant le transfert des données, sauf au début du transfert pour les cibles. Par conséquent, une fois que la cible et le maître commettent à un transfert, ils ne peuvent insérer aucun état d'attente durant le transfert. La nouvelle règle pour la permission d'insertion des états d'attentes implique une utilisation plus efficace du bus mais implique aussi la nécessite d'augmenter le nombre des mémoires tampons dans le design ce qui ramène à un design plus coûteux.

Également, le bus PCI-X optimise le débit en permettant aux transactions en mode continu d'être plus longues que celles du bus PCI, et en limitant les zones auxquelles l'initiateur ou la cible peut arrêter un transfert de données. Un nouveau concept de la borne permise de débranchement (Allowable Disconnect Boundary, ADB) sert comme

une base de ce concept dans lequel les gestionnaires en mode PCI-X peuvent arrêter les transferts seulement au frontière d'un ADB, qui est 128 octets de longueur.

2.6. Modifications sur le bus PCI-X dues aux améliorations

2.6.1. Transaction dans le bus PCI-X

Comme pour le PCI, le maître demande d'abord la transaction au bus PCI-X et après la réception du signal **GNT#** activé, il active à la suite le signal **FRAME#** pour débuter le transfert et il commande l'adresse sur le bus **AD** simultanément. Le chronogramme de ces signaux pour une transaction dans le bus PCI-X est illustré dans la figure 2.17.

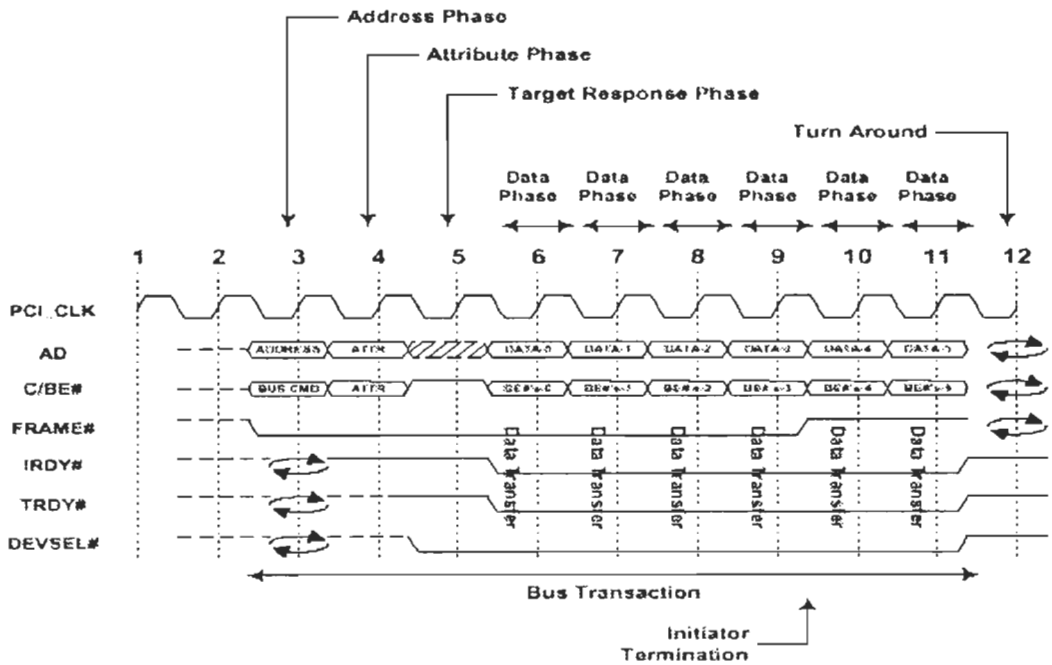

Figure 2.16 : Transaction dans le bus PCI-X

Notons que le cycle dans lequel l'adresse est commandée est la phase d'adresse. Juste après, la phase d'attribut se produit dont le maître commande l'attribut sur le bus **AD**. Après cette phase, la phase de réponse de la cible se produit dont la cible répond au transfert en activant le signal **DEVDEL#**. Signalons que la cible peut prendre jusqu'à quatre horloges pour activer **DEVSEL#** selon sa vitesse de décodage de son adresse.

Après la production de ces phases la cible indique qu'elle est prête à transférer les données en activant **TRDY#**.

Notons qu'une phase de données se produit dans n'importe quelle horloge dont **IRDY#** et **TRDY#** sont activés simultanément. Néanmoins, il faut signaler que grâce au temps de latence introduit dans le bus PCI-X, le maître peut indiquer son achèvement du transfert en désactivant **FRAME#** avant la désactivation de **IRDY#** durant deux périodes de l'horloge, au lieu d'une seule période de l'horloge pour une transaction en mode PCI. Et finalement la phase de la rotation de bus(turn around) se produit dont le maître et la cible cèdent les signaux de transfert vers des nouveaux maîtres et cibles connectés au bus.

2.6.2. Espace de Configuration

Les gestionnaires du bus PCI-X incluent des nouveaux registres d'état et de commande qui sont situés dans les espaces de configuration. Le système de configuration détermine si un gestionnaire supporte le protocole du PCI-X par la présence d'un pointeur vers les registres de cette capacité dans la liste de nouvelles capacités, selon la méthode expliquée dans le paragraphe 2.2.4. Par ailleurs, un gestionnaire à multifonction type PCI-X doit mettre en application ces registres dans l'espace de configuration de chaque fonction. La figure 2.17 présente les registres de la capacité PCI-X pour un dispositif à espace de configuration du type 00h et type 01h (un bridge de PCI-X).

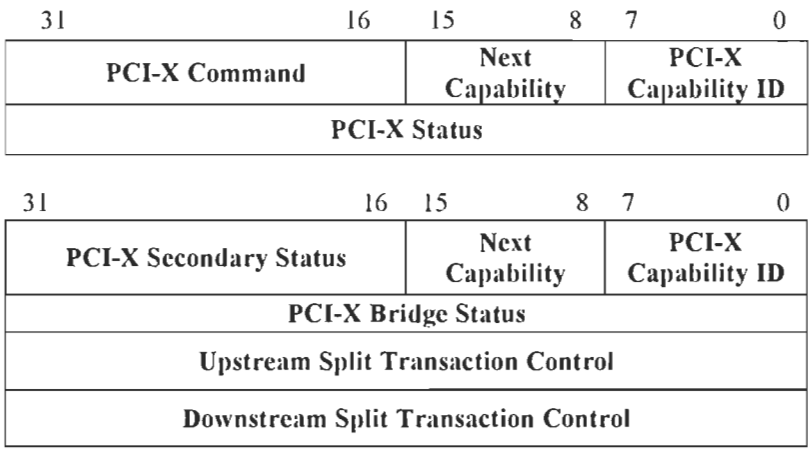

Figure 2.17 : Espace de configuration type 00h(en haut) et 01h(en bas) pour la capacité PCI-X.

2.6.3. Commandes de PCI-X

Le tableau 2.11 présente une comparaison entre les encodages des commandes de bus PCI-X et celles de PCI conventionnel. En conception du système, les initiateurs ne doivent pas produire des commandes réservées, et les cibles doivent ignorer toutes les transactions utilisant une commande réservée

Tableau 2.11 : Codes de commandes de bus PCI-X

CBE#[0:3]	PCI-X Commandes	PCI Conventionnel
0000	Interrupt Acknowledge	Interrupt Acknowledge
0001	Special Cycle	Special Cycle
0010	I/O Read	I/O Read
0011	I/O Write	I/O Write
0100	Réservé	Réservé
0101	Réservé	Réservé
0110	Memory Read DWORD	Memory Read
0111	Memory write	Memory Write
1000	Alias to Memory Read Block	Réservé
1001	Alias to Memory Write Block	Réservé
1010	Configuration Read	Configuration Read
1011	Configuration Write	Configuration Write
1100	Split Completion	Memory Read Multiple
1101	Dual Address Cycle	Dual Address Cycle
1110	Memory Read Block	Memory Read Line
1111	Memory Write Block	Memory Write and Invalid

2.6.4. Initialisation des gestionnaires en mode PCI-X

Un gestionnaire type PCI-X doit avoir la capacité de travailler en mode PCI. La configuration d'un dispositif pour fonctionner en mode PCI-X ou en mode PCI conventionnel se fait à l'initialisation du système. À la mise en marche, le système lance une séquence d'initialisation pour les dispositifs qui composent le système. Cependant, les sorties des broches de **PCIXCAP** et de **M66EN**, de chaque gestionnaire, indiquent au système si le gestionnaire supporte le protocole PCI-X et à quelle fréquence. Le tableau 2.12 présente la façon utilisée par un gestionnaire pour indiquer sa capacité PCI-X au système de configuration en combinant ses broches **M66EN** et **PCIXCAP**.

Tableau 2.12 : Détection de la capacité du gestionnaire PCI/PCI-X par le système de la configuration

M66EN	PCIXCAP	Conventional Device Frequency Capability	PCI-X Device Frequency Capability
Ground	Ground	33 MHz	Not capable
Not connected	Ground	66 MHz	Not capable
Ground	Pull-down	33 MHz	PCI-X 66 MHz
Not connected	Pull-down	66 MHz	PCI-X 66 MHz
Ground	Not connected	33 MHz	PCI-X 133 MHz
Not connected	Not connected	66 MHz	PCI-X 133 MHz

D'ailleurs, après la détermination des capacités du gestionnaire par le système hôte, le host bridge commande un modèle de configuration selon le tableau 2.13 pour initialiser le système avec **RST#** activé. Notons que tous les gestionnaires connectés au système hôte décodent le modèle d'initialisation et déterminent le mode de fonctionnement si c'est PCI ou PCI-X. Si à l'état de repos (**FRAME#** et **IRDY#** désactivés) un gestionnaire détecte, un ou plusieurs signaux de **DEVSEL#**, **TRDY#**, et **STOP#** activés avec **RST#** activé, le gestionnaire entre dans le mode PCI-X; autrement, il entre le mode de PCI conventionnel.

Tableau 2.13 : Modèles d'initialisation d'un gestionnaire au mode PCI ou PCI-X

DEVSEL#	STOP#	TRDY#	Mode	Max Clock Period (ns)	Min Clock Period (ns)	Min Clock Freq (MHz) (ref)	Max Clock Freq (MHz) (ref)
Deasserted	Deasserted	Deasserted	Conventional 33	∞	30	0	33
			Conventional 66	30	15	33	66
Deasserted	Deasserted	Asserted	PCI-X	20	15	50	66
Deasserted	Asserted	Deasserted	PCI-X	15	10	66	100
Deasserted	Asserted	Asserted	PCI-X	10	7.5	100	133
Asserted	Deasserted	Deasserted	PCI-X	Reserved	Reserved	Reserved	Reserved
Asserted	Deasserted	Asserted	PCI-X	Reserved	Reserved	Reserved	Reserved
Asserted	Asserted	Deasserted	PCI-X	Reserved	Reserved	Reserved	Reserved
Asserted	Asserted	Asserted	PCI-X	Reserved	Reserved	Reserved	Reserved

2.7. Conclusion

Au niveau de ce chapitre, nous avons présenté une étude exhaustive sur les différents types des gestionnaires type PCI. Par la suite, nous avons discuté sur leurs architectures globales, leurs caractéristiques électriques et surtout leurs mécanismes de transfert des informations. Ces gestionnaires type PCI présentent des limitations au niveau de la fréquence d'opération pour les applications demandant des débits élevés, d'où l'apparition du bus PCI-X qui vient de combler ces limitations.

Par ailleurs, cette étude nous a permis de bien comprendre la fonctionnalité des gestionnaires type PCI/PCI-X afin de pouvoir incorporer leurs différents blocs à intégrer dans nos gestionnaires génériques développés; ce qui va être présenté dans le chapitre 3.

Chapitre 3

Développement de gestionnaire générique type PCI/PCI-X

Le développement d'un gestionnaire de périphériques consiste à le décrire avec un outil de description de matériels afin de pouvoir le matérialiser dans un circuit intégré. Cette description peut se faire avec un dessin de masque, un schéma logique, des tables de vérité, des équations logiques, un langage évolué, ou encore avec un langage de description de matériels.

Cependant, le temps de préparation pour le marché (time-to-market), l'abondance des technologies cibles de circuits programmables (PLD), et la complexité des nouveaux designs ont forcé les designers à adopter un langage de programmation HDL (Hardware Description Language) comme le VHDL, le Verilog et le C/C++, pour le développement de grands systèmes.

En outre, un avantage d'utiliser un langage HDL pour la conception du matériel est l'abstraction de niveau de la conception des designs. En effet, la conception des circuits logiques utilisant un langage HDL améliore la productivité en permettant aux designers de fonctionner avec des opérations et des comportements plutôt qu'avec l'approche traditionnelle des circuits schématiques des fils et de portes logiques.

Un deuxième avantage important est la portabilité de design. Peut-être la raison la plus importante d'adopter un langage standard pour la conception c'est qu'il facilite le passage d'un design source d'un constructeur de PLD à l'autre. Par ailleurs, les ingénieurs décrivent typiquement les designs schématiques avec des symboles de logique spécifique au vendeur de PLD. Ces schémas exigent la substitution de composantes et de révisions si on désire refaire le design en utilisant des composantes appartiennent aux autres vendeurs de PLDs. Tandis qu'idéalement, le modèle synthétisable d'un HDL devrait être le même pour toutes les technologies de cible afin d'assurer l'aspect générique au niveau du choix de la technologie cible [19].

3.1. Langages de description de matériels.

Actuellement, il y a trois langages fortement utilisés pour la description des matériels : Verilog, VHDL et C/C++. Le Verilog et le VHDL sont les langages les plus utilisés, ils se comportent comme des standards pour le développement des matériels et cependant grâce à des outils disponibles présentement comme Renoir de Mentor Graphics, on peut concevoir des systèmes mixtes entre ces deux langages.

Le Verilog est limité pour la description des systèmes, (ce qui est illustré dans la figure 3.1), à cause de son amoindrissement au niveau des librairies pour les opérations arithmétiques, et de sa limitation concernant les types d'éléments existants pour la construction de nouveaux types [20].

Au contraire le VHDL et plus versatile et il facilite mieux que le Verilog le développement au niveau de grand système[21]. Le C/C++ commence à apparaître

largement dans le domaine du développement des matériels et d'après l'avis de l'expert Michael A. Bohm de la compagnie Exemplar Logic : le C/C++ va connaître une utilisation croissance et majeure parmi les développeurs de matériels dans les quatre ans imminents [22].

D'ailleurs, Il faut citer qu'il y a des efforts pour étendre les librairies de Verilog en faisant des extensions comme le Superlog [23]. D'autre part, au niveau du VHDL, le VHDL+ [24] et SUAVE [25] ont été développés comme des extensions mais jusqu'à maintenant elles ne sont pas approuvées par le comité de standardisation du langage VHDL, et selon notre avis ces extensions proposent des grandes améliorations au niveau des développements des grands systèmes.
La figure 3.1 présente la capacité de ces langages HDL versus le niveau de description des systèmes [22].

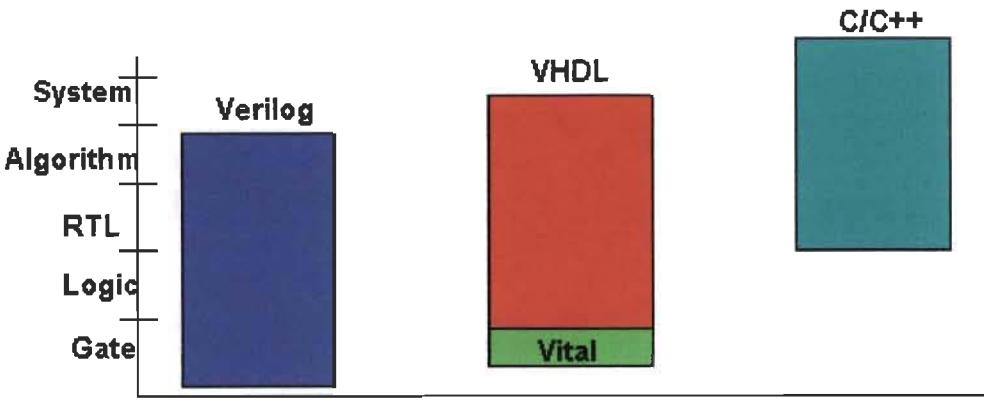

Figure 3.1 : Capacité des langages HDL versus les niveaux de description des systèmes

Il est intéressant de constater que le VHDL couvre plus que les autres langages l'intervalle de développement dans les systèmes actuels.
Dans ce mémoire, le gestionnaire type PCI/PCI-X a été décrit avec le langage de description de matériels VHDL. Étant donné que ce langage est normalisé par IEEE(Institute of Electrical and Electronics Engineers), plusieurs outils d'aide CAO (Conception Assistée par Ordinateurs) compatibles avec celui-ci sont actuellement disponibles. Cependant, il existe aussi plusieurs librairies du VHDL pouvant faciliter la conception et la simulation d'un système numérique. Le langage VHDL permet d'obtenir

conception et la simulation d'un système numérique. Le langage VHDL permet d'obtenir une description hiérarchisée composée de petites unités assemblées. De plus, tel que présenté à la figure 3.2, son haut niveau d'abstraction facilite la compréhension, la conception logique ainsi que le développement d'un système aisément modifiable[26].

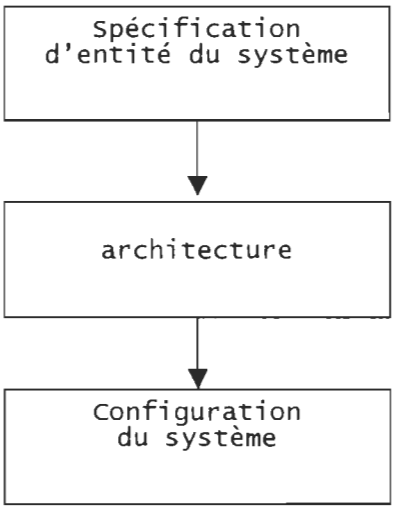

Figure 3.2 : Niveaux d'abstraction

Ces principaux niveaux d'abstraction ont lieu en trois étapes : la spécification d'entité, l'architecture et la configuration du système. Le VHDL permet donc de décrire un système en faisant l'abstraction des niveaux inférieurs, en créant sa spécification d'entité, son architecture et, finalement, sa configuration. Par définition, la spécification d'entité permet de spécifier les entrées et les sorties d'un système numérique, l'architecture permet de décrire le comportement ou la structure d'un système, ou même de mixer les deux dans une description hybride, et la configuration permet de définir les liens physiques entre les composants dans des architectures et des entités spécifiques.

3.2 Méthodologie pour le développement du gestionnaire

La méthodologie suivie pour le développement du gestionnaire est constituée de huit étapes telle que représentée à la figure 3.3.

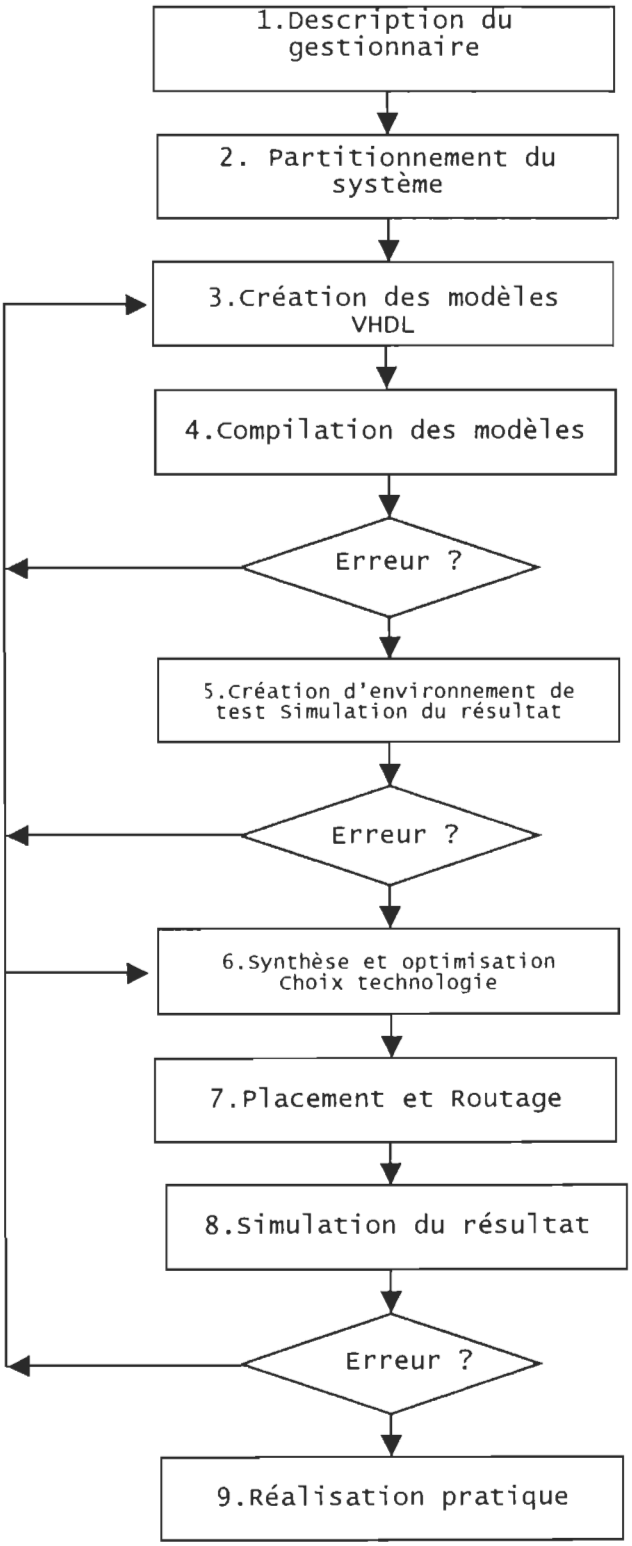

Figure 3.3 : Étapes de conception du gestionnaire

Étape 1 – Définitions des spécifications matérielles du gestionnaire afin d'être générique;

Étape 2 – Partitionnement du système. En effet, le système est partitionné en plusieurs unités qui sont, à leur tour d'une manière hiérarchique, partitionnées en plusieurs sous-unités;

Étape 3 – Création des modèles VHDL. À ce stade, un modèle VHDL est crée pour chaque sous-unité.

Étape 4 – Compilation des modèles. Le modèle est compilé dans l'environnement d'aide à la conception CAO Synplify de Synplicity. En cas d'erreurs, il faut corriger la syntaxe du code VHDL et il faut recompiler à nouveau.

Étape 5 – Création un environnement pour simuler le gestionnaire.
Elle se consiste en :
-Création d'un environnement de test (testbench) semblable à celui où le gestionnaire va être utilisé.
-Création des vecteurs de tests qui couvrent les transactions possibles à trouver dans un bus type PCI/PCI-X.
-Connections de gestionnaire dans cet environnement de test.
-Simulation avec l'outil Modelsim de Mentor Graphics sans tenir compte du délai.
-Retour à l'étape 3 en cas de la détection d'erreurs afin de corriger notre modèle VHDL.

Étape 6 – Synthèse et optimisation.
Ces opérations sont réalisées avec le logiciel Synplify Pro de Synplicity et les librairies des circuits programmables FPGA de la famille Xilnix.

Étape 7 – Placement et Routage.
Elle se consiste en :
-Optimisation de notre conception du gestionnaire dans les librairies cibles.

-Amélioration de la surface occupée et de la vitesse du fonctionnement du circuit en utilisant le langage TCL (Tool Command Language)

Étape 8 – Simulation du résultat.
Elle a lieu en deux étapes :
-Simulation avec Model Sim en tenant compte des délais engendrés par les caractéristiques matérielles des circuits programmables.
-Retour à l'étape 6 afin d'optimiser le système au niveau d'intégration ou à l'étape 3 pour corriger notre modèle VHDL.

Étapes 9 – Réalisation pratique
Génération des fichiers permettant la configuration des circuits programmables pour la réalisation pratique.

3.3. Description des spécifications matérielles du gestionnaire générique

3.3.1 Interface du gestionnaire avec le bus PCI et les applications arrières

Avant de présenter l'architecture proposée pour notre gestionnaire générique, toutefois on va discuter sur ses environnements qui nous ont amenés à notre proposition. Par ailleurs, pour que le gestionnaire type PCI/PCI-X développé, soit générique il faut qu'il supporte plusieurs applications sur la même carte d'extension (multifunction device). En effet, le support de plusieurs fonctions exige au gestionnaire d'être indépendant d'une application spécifique, en revanche, il doit recevoir les informations de chaque fonction lorsque cette dernière est l'application de la transaction courante avec le système (figure 3.4). Néanmoins, de côté de bus PCI, le gestionnaire doit avoir une interface fixe conforme à l'interface de bus PCI (vue dans le chapitre 2), et de l'autre coté, (côté des applications locales implantées devant ce gestionnaire), il doit assurer une architecture qui fait la sélection entre les bus de commandes et les bus de données pour les différentes fonctions.

Également, chaque fonction type PCI doit inclure un espace de configuration indépendant des autres fonctions pour assurer son auto configuration dans le système hôte ou pour informer ce système de sa capacité. Par conséquent, le système hôte de configuration doit avoir la capacité d'accès à plusieurs espaces de configuration via le gestionnaire.

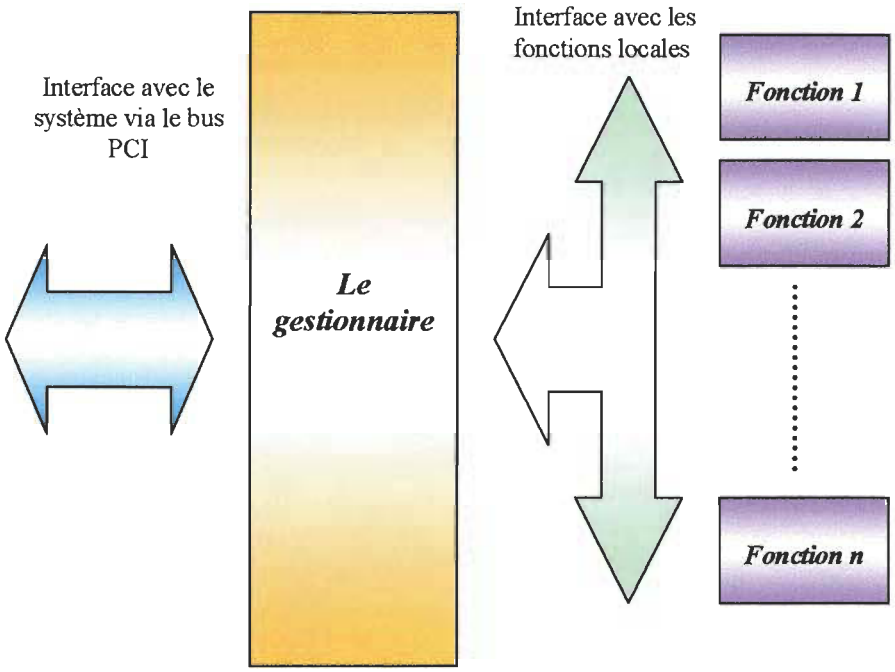

Figure 3.4 : Support des plusieurs fonctions par le gestionnaire

En effet, étudions l'exemple de deux applications résidentes derrière le gestionnaire : une application à une fonctionnalité maître et l'autre à une fonctionnalité cible. Le système doit lire les deux registres *Max_lat* et *Min_lat* de l'espace de configuration du maître pour savoir le temps nécessaire de ses transferts sur le bus, et également pour l'espace de configuration de la cible, le système doit savoir sa rapidité pour décoder son adresse, donc le système doit avoir la capacité d'accès à deux espaces de configuration à travers le gestionnaire l'une est attribuée pour le maître et l'autre est attribuée pour la cible.

Pourtant, si on a intégré tous les espaces de configuration à l'intérieur du gestionnaire : il va être trop encombrant, et lors de sa conception, le nombre des applications doit être fixé, d'où la notion générique sur le nombre des applications n'est pas respectée. Par conséquent, la possibilité d'augmenter le nombre des applications reliées à ce gestionnaire est impossible. Et par ailleurs, si le nombre des applications utilisées est

inférieur au nombre des espaces de configuration intégré lors de la conception du gestionnaire, on aura une utilisation d'espace de configuration inutile. La figure 3.5 illustre un gestionnaire cible développé dans l'un des articles [12] et [15] où on constate que la logique interne est liée à une seule fonction d'où la notion générique sur le nombre des applications n'est pas respectée.

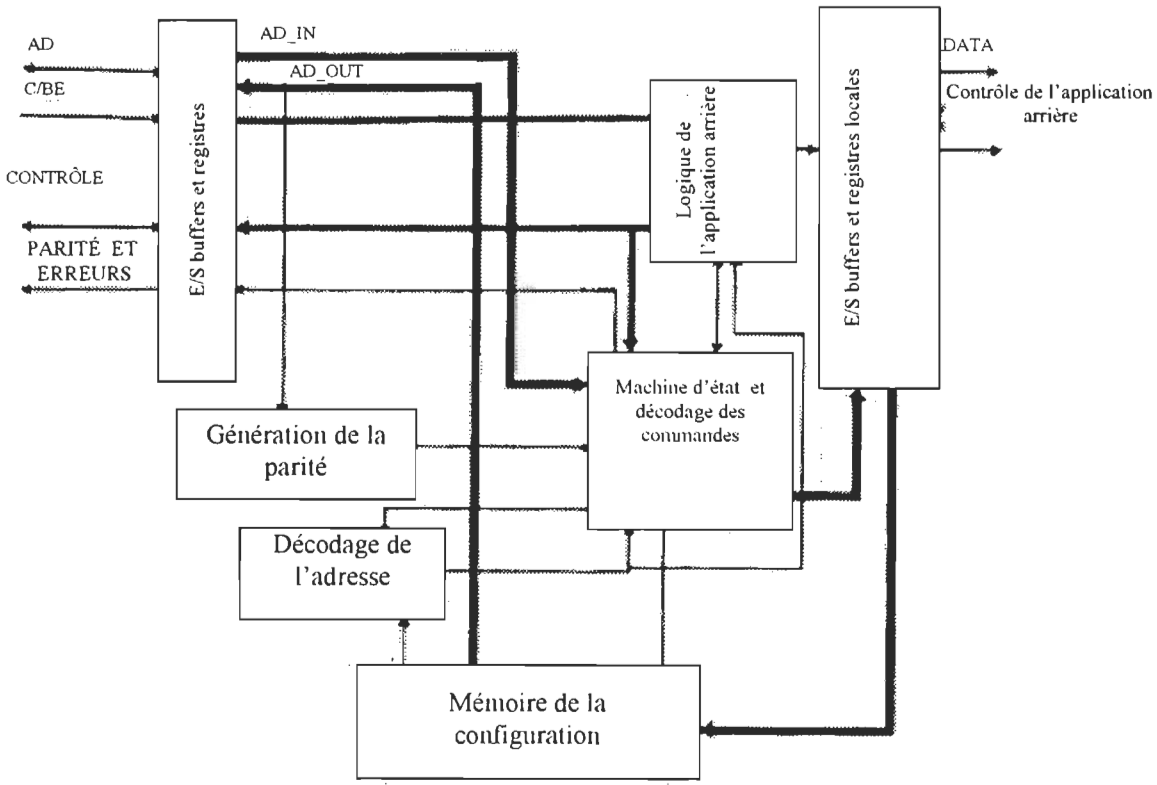

Figure 3.5 : Mémoire de configuration est à l'intérieur de gestionnaire [12]

Pour résoudre à ces problèmes précités, on a proposé :

- d'extraire les espaces de configuration à l'extérieur du gestionnaire;
- de créer une interface générale entre le gestionnaire et les espaces de configuration qui a comme rôle de sélectionner l'espace de configuration de l'application courante de la transaction. Par conséquent, le gestionnaire fonctionne d'une manière qu'il communique avec un seul espace de configuration.

Toutefois, en séparant l'espace de configuration on peut implanter des espaces de configuration selon le nombre des fonctions désirées. Par la suite, en adoptant à ces

propositions, on abouti à un gestionnaire non encombrant et à un nombre de fonction illimité.

D'ailleurs, dans le cas où plusieurs fonctions type cible connectées au gestionnaire, ce dernier, doit répondre aux différents espaces d'adressage lorsqu'un maître connecté au côté de bus PCI veut accéder à l'une de ces cibles. Comme on a actuellement les espaces de configuration, où résident les adresses des cibles à décoder, à l'extérieur du gestionnaire, on doit ajouter une unité de décodage qui a pour rôle de décoder les adresses fournies par les espaces de configuration implantés et d'informer le gestionnaire de demander la transaction dans le cas d'accéder l'une des ces cibles.
Également, dans le cas où plusieurs maîtres connectés au bus PCI à travers ce gestionnaire, on a avéré la nécessité d'ajouter une logique d'arbitrage simple qui a pour rôle de donner l'ordre de la transaction qui voudraient accéder à ce bus simultanément. Les unités de décodage et de l'arbitrage sont rassemblées dans une seule unité nommée unité de ***Decodage_arbitrage***.

D'après ces propositions sur l'architecture du gestionnaire on a trouvé une interface entre le gestionnaire et les applications arrières représentée à la figure 3.6. Cette figure montre le gestionnaire PCI/PCI-X en interface avec deux fonctions arrières. La première fonction est un bridge par exemple avec un espace de configuration type 1 (cfig1_typ1). D'ailleurs, Les fonctions type bridge doivent avoir la capacité maître et cible donc on a pour ce bridge deux composantes : un maître (Maître1) et une cible (Cible1). La seconde fonction a un espace de configuration type 0. Pour cette deuxième fonction, on a présenté comme exemple deux composantes un maître (Maître2) et une cible (Cible2) mais il faut signaler que l'implantation d'une seule composante ou deux, dépend de la capacité de la fonction indiquée dans son espace de configuration (cfig2_typ0).

Cependant, en ce qui concerne l'interface du gestionnaire, elle est indépendante du nombre des fonctions arrières ce qui nous permet d'implanter plusieurs fonctions connectées à l'interface de gestionnaire par des multiplexeurs commandés par l'unité de ***Decodage_arbitrage***.

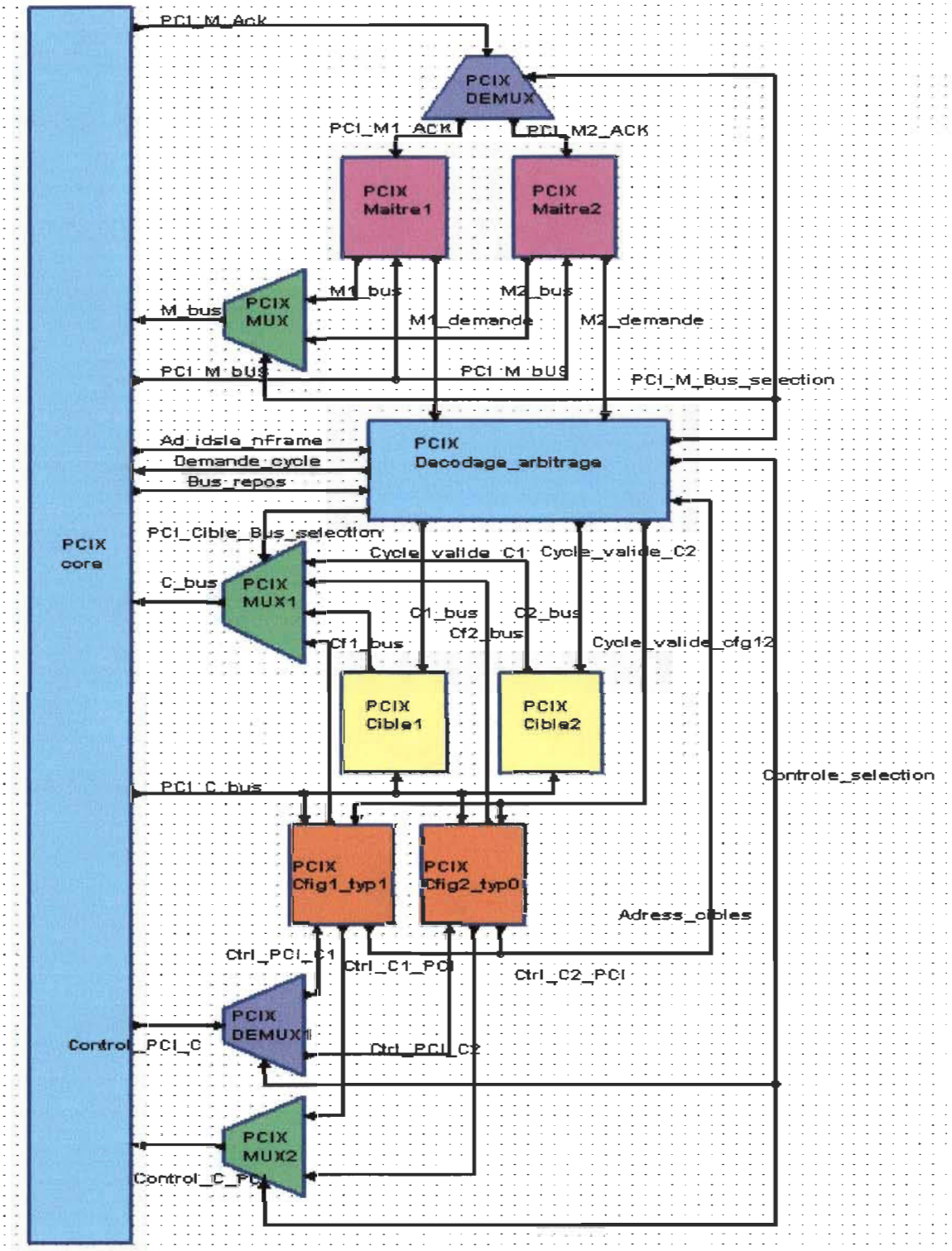

Figure 3.6 : Interface entre le gestionnaire et les applications arrières

3.3.2. Unité de Décodage_Arbitrage

L'architecture de l'unité ***Décodage_Arbitrage*** est présentée dans la figure 3.7, notons que cette unité présentée est conçue pour l'utilisation avec deux applications arrières.

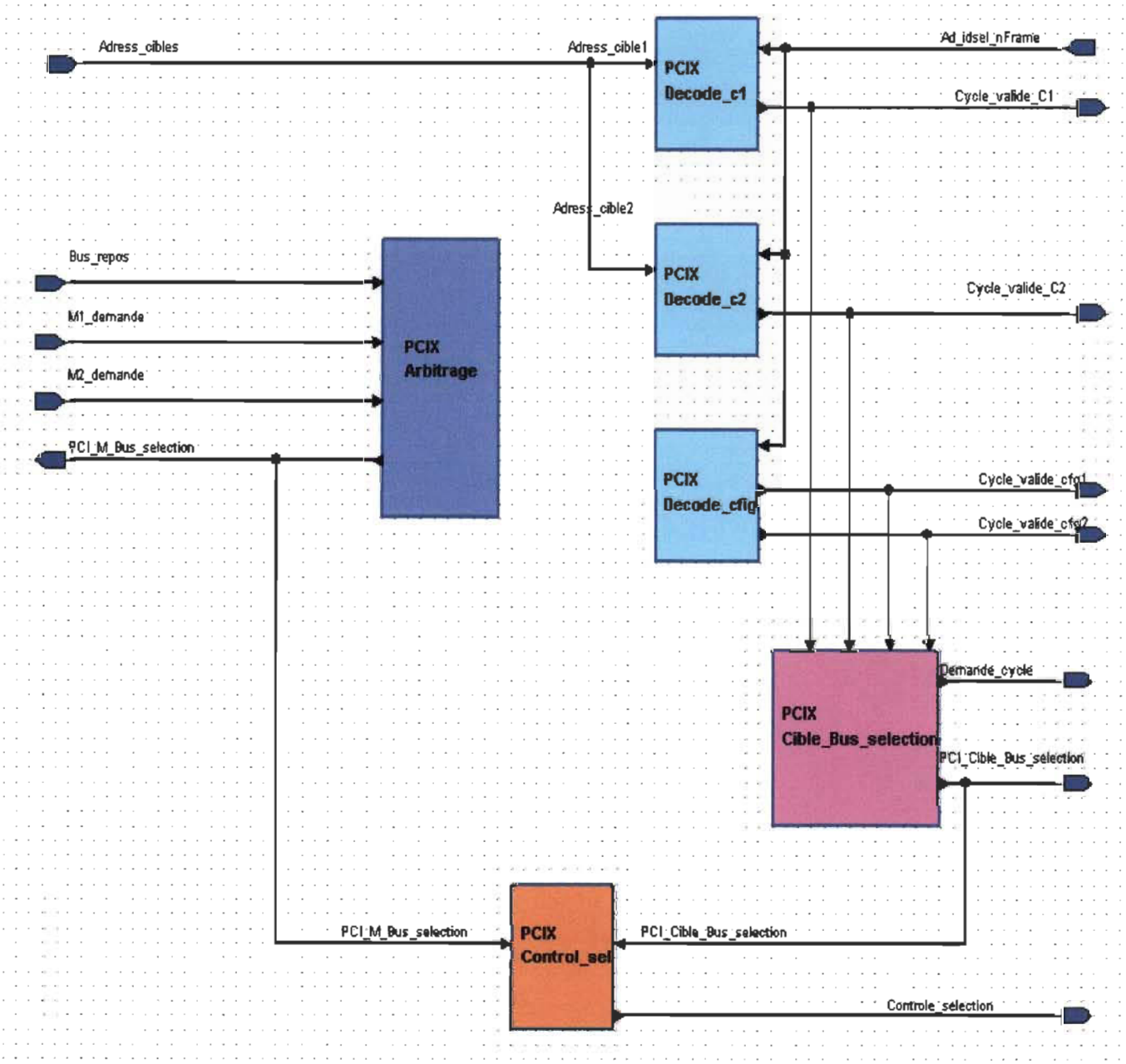

Figure 3.7 : Unité Décodage_Arbitrage

Cette unité de ***Décodage_Arbitrage*** est divisée en deux sous-unités : arbitrage et décodage.

- **Unité de l'arbitrage :**

Elle a pour rôle de choisir le maître actuel de la transaction selon un algorithme de priorité indépendant du gestionnaire. Cet algorithme peut être un algorithme équitable entre les applications types maîtres ou un algorithme qui favorise certaines applications selon leurs types.

Il est intéressant de signaler que dans le cas où plusieurs maîtres demandent l'accès au bus PCI simultanément, l'unité de l'arbitrage décide qui sera le maître de bus. Lorsqu'on a une transaction au bus, il faut éviter que l'unité de l'arbitrage donne l'ordre de la transaction à un autre maître interne qui demande l'accès au bus (en activant le signal **Mi_demande**). En conséquence le gestionnaire fournit le signal **bus_repos** qui informe l'unité de l'arbitrage que le bus est à l'état de repos et à ce moment elle peut accepter seulement une demande d'un maître interne.

Notons que lorsqu'un maître, (Maître i tel que $i \in \{1,2\}$ dans la figure 3.6), demande l'accès au bus en activant le signal **Mi_demande**, l'unité de l'arbitrage active de son tour le signal **PCI_M_Bus_selection** pour commander le multiplexeur MUX qui connecte le bus du maître (Maîtrei) au gestionnaire et aussi pour commander un Demultiplexeur, DEMUX, pour connecter le bus de PCI vers le maître qui a demandé l'accès.

- **Unité de décodage**

Elle a pour rôle d'informer le gestionnaire qu'un maître, connecté au côté de bus PCI, accède l'une des ses cibles ou que le système de configuration principal veut configurer l'un des espaces de configurations résidants derrière ce gestionnaire.
Cette unité de sa part a composé des sous unités de décodage qui fonctionnent en parallèle pour accélérer le décodage. Il y a deux types de sous unités de décodage : une

pour décoder l'adresse d'une cible (ex. Decode_c1, figure 3.7) et l'autre pour décoder l'accès à un espace de configuration (ex. Decode_cfig, figure 3.7).

D'ailleurs, le premier type de sous unité compare l'adresse d'une cible reçue de son espace de configuration et l'adresse **AD** à décoder reçue du gestionnaire. Si ces adresses sont égales, elle active le signal **Demande_cycle** pour informer au gestionnaire de répondre à la transaction courante. Or le signal **Frame#**, fournit par le gestionnaire, active à son état 0 au début d'un cycle, le décodage par ces sous unités lors du début d'un cycle.

Le deuxième type de sous unité de décodage est activé seulement quand le signal **IDSEL**, fournit par le gestionnaire, est à l'état 1, ce qui indique l'accès sur l'une des espaces de configuration de ce gestionnaire. Également le signal **IDSEL** désactive les sous unités de décodage type 1. En plus de signal **IDSEL** ce type des unités de décodage, a des signaux d'interfaçage avec le gestionnaire comme l'adresse à décoder **AD** et le signal **FRAME#** qui indique le début d'un cycle.

À ce stade, et en résumé sur le fonctionnement de l'unité de décodage tout entier on trouve :

Quand **IDSEL** =1 et **FRAME#** =0 l'unité de décodage décode le champ *Function number* de l'adresse fournit par le gestionnaire via l'adresse **AD** pour connaître l'unité de configuration cible de la transaction, puis elle commande le signal **cycle_valide_Cfgi** pour activer l'espace de configuration concerné par la transaction.

Quand **IDSEL** =0 et **FRAME#** =0 l'unité de décodage décode les adresses qui sont fournies par les espaces de configuration afin d'activer la cible concernée à la transaction par l'activation de son signal **cycle_valide_Ci**.

Dans ces deux états l'unité de décodage commande un multiplexeur MUX1 (figure 3.6) en activant le signal **PCI_cible_bus_selection** pour connecter le bus de la cible de la transaction au gestionnaire.

Quand l'un des signaux **cycle_valide_Ci** ou **Cycle_valide_cfgi** est activé le signal **Demande_cycle**, connecté de l'unité de décodage au gestionnaire, est activé pour que ce dernier demande le cycle sur le bus PCI en activant son signal **DEVSEL#** de côté de bus PCI.

Enfin le signal **controle_selection** est activé pour connecter les signaux de contrôle de l'unité de configuration de la composante (cible ou maître) de la transaction au gestionnaire. Par ailleurs, ce signal commande le Demultiplexeur (DEMUX1) et le multiplexeur (MUX 2) pour connecter les signaux de contrôle de la composante maître ou cible lié à la transaction actuelle avec le gestionnaire et les signaux de contrôle de gestionnaire avec l'unité de configuration de cette composante respectivement.
Il faut noter que les bus de données et d'adresse du gestionnaire sont connectés vers les cibles, les maîtres et les unités de configuration directement(figure 3.6).

Dans le cas où le gestionnaire est implanté dans un circuit programmable PLD (notre technologie cible de l'implantation) séparé des applications arrières type cible et maître implantées derrière lui. Le choix de bus à deux états entre le gestionnaire et l'application arrière laisse le nombre limité de pins trois états de PLD à l'interface entre le gestionnaire et le bus PCI qui nécessite beaucoup de signaux de trois états, c'est la raison pour laquelle qu'on choisit des signaux type deux états entre le gestionnaire et ses applications arrières.

3.3.3. Unité de l'espace de configuration

Cette unité possède deux interfaces : interface avec le gestionnaire et interface avec l'unité de ***Decodage_Arbitrage***. Elle est formée de deux unités : l'unité de transfert des informations et l'unité du banc des registres. La figure 3.8 illustre l'architecture de l'unité de l'espace de configuration, ses deux sous unités, ses interfaces combinatoire et synchrone avec le gestionnaire.

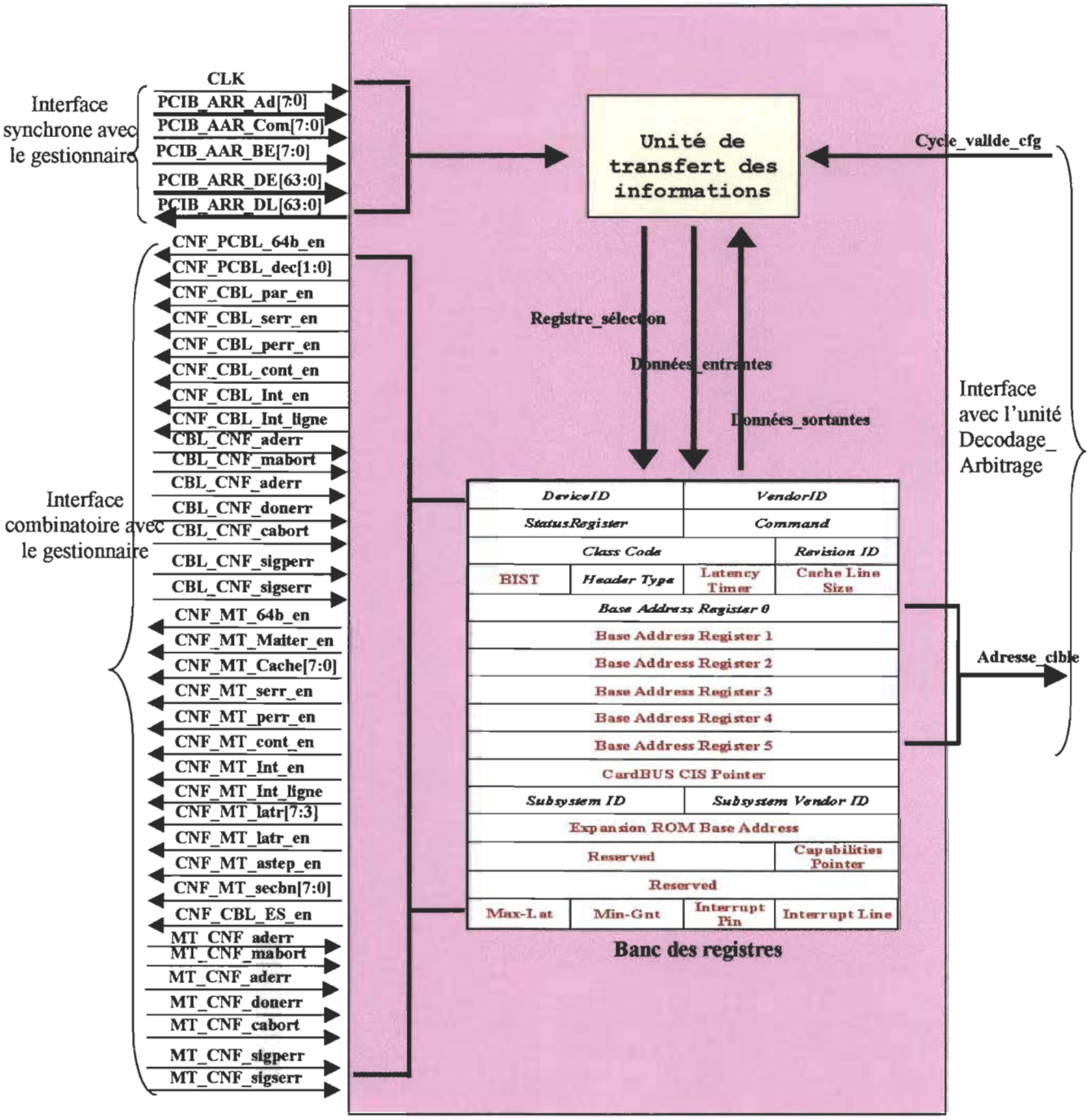

Figure 3.8 : Unité de l'espace de configuration

L'unité de transfère de l'information a deux interfaces synchrones, l'une reliée au gestionnaire, et l'autre reliée à l'unité de banc des registres. À travers cette unité, ces deux interfaces ont pour rôle d'échanger les informations entre le gestionnaire et le banc des registres à travers cette unité de transfère avec la cadence de l'horloge de système. Parmi les signaux principaux des interfaces synchrones avec le gestionnaire, on

trouve le signal **PCIB_AAR_Com** utilisé par l'unité de transfert pour connaître si on a une commande de lecture ou d'écriture générée par le système à travers le gestionnaire vers le banc des registres. Cependant, tous les signaux restants sont des signaux d'adressage et des données entre le gestionnaire et le banc des registres. Cette unité commande ses sorties au front montant de l'horloge, en revanche ses entrées sont non échantillonnées et décodées directement pour assurer la réponse rapide vers le gestionnaire au prochain front montant de l'horloge.

L'unité de banc des registres est une mémoire vivante mais possède des emplacements pour lecture ou d'écriture seule. Parmi les emplacements de lecture seule on rencontre les registres dépendants de l'application qui utilise cet espace de configuration, par exemple le registre *Vendor ID* qui ne peut pas être modifié par le système. Les emplacements de lecture et écriture sont soit des registres configurés par le système, comme *Base Adresse*, ou soit des registres modifiés par le gestionnaire lors de la détection d'une erreur, comme le *Status Register*.

L'interface entre l'unité de banc des registres et le gestionnaire est combinatoire utilisée pour faire une mise à jour rapide pour l'état de l'application courante qui utilise cet espace de configuration. Pour éclaircir le fonctionnement de cette interface combinatoire, prenons l'exemple où l'espace de configuration est utilisé pour une application à capacité maître. Donc si le système de configuration via le gestionnaire et l'unité de transfert de l'information change le contenu du registre *Lattency Timer* par une transaction d'écriture dans l'espace de configuration, il faut que le gestionnaire met à jour le compteur qui est chargé par le nouveau contenu du *Lattency Timer* et limite l'accès de ce maître au bus. D'où la nécessité d'introduire une interface combinatoire directe qui lie directement le registre *Lattency Timer* au gestionnaire à travers les signaux de sortie de l'unité de configuration: **CNF_MT_latr[7:3]**. Cet exemple décrit comment le système modifie directement les informations du gestionnaire en modifiant les informations de l'espace de configuration, sans avoir réservé une mémoire pour stocker les informations dans ce gestionnaire.

Également, on peut décrire dans un deuxième exemple, comment le gestionnaire informe le système sur les modifications subies lors d'une transaction dans le bus en modifiant l'espace de configuration : Quand le gestionnaire détecte une erreur dans le bus, par exemple un *Target_Abort* ou une erreur lors d'une phase d'adressage, il faut qu'il informe le système en changeant les contenus du bit **Signaled Target Abort** , ou du bit **Signaled System Error** respectivement dans le registre d'état *Status register* .
Notons que, ce changement doit être immédiat pour que le système puisse lire les modifications directement quand il exécute une lecture à ce registre d'état. D'où la nécessité d'introduire une interface combinatoire directe qui lie ces deux bits directement au gestionnaire à travers les signaux de sorties de l'unité de configuration: **CBL_CNF_cabort** et **CBL_CNF_sigserr**.

Par ailleurs, on peut trouver des signaux qui lient le gestionnaire avec l'unité de configuration afin d'assurer la notion générique de l'espace de configuration : Le signal de sortie de l'unité de configuration **CNF_MT_Maiter_en** indique la capacité de l'application (qui fait recours au gestionnaire), si elle est capable d'être un maître ou non.
À partir de ce qui précède, l'interface de cette unité de configuration avec l'unité de ***Decodage_arbitrage*** est limitée par les deux signaux : le signal **Adresse_cible** qui fournit l'adresse de cible à décoder à l'unité de décodage et le signal **valid_cycle_cfig** qui informe l'unité de l'espace de configuration qu'un accès est destiné vers elle.

Il faut noter que les registres qui définissent les caractéristiques de l'application arrière qui utilise le gestionnaire, notamment les registres *DEVICE ID* ou *Class Code* etc..., sont conçus comme des registres configurables pour assurer la capacité de ce gestionnaire d'être utilisé par n'importe quelle application pour différents manufacturiers.

3.3.4 Unité application arrière type maître

Cette unité est une application qui nécessite de générer des transactions sur le bus PCI. Elle possède trois interfaces : une interface avec le gestionnaire, une interface avec l'unité ***Decodage_Arbitrage*** et une interface vers l'extérieur, et est formée de deux

unités: l'unité de transfert des informations et l'unité de l'application interne type maître et sa logique de contrôle. Sa description globale est illustrée dans la figure (3.9).

L'unité de l'application type maître peut être n'importe quelle application qui maîtrise le bus. Prenons l'exemple d'une carte de communication qui lorsqu'elle reçoit, via son interface vers l'extérieur, les données à transférer, ensuite elle demande la maîtrise de bus pour générer des transferts d'écriture vers la cible des données (mémoire principale par exemple), connectée au bus PCI, et de même elle lance des transferts de lecture des données à transférer d'une cible vers l'interface de l'extérieur à l'instant opportun.

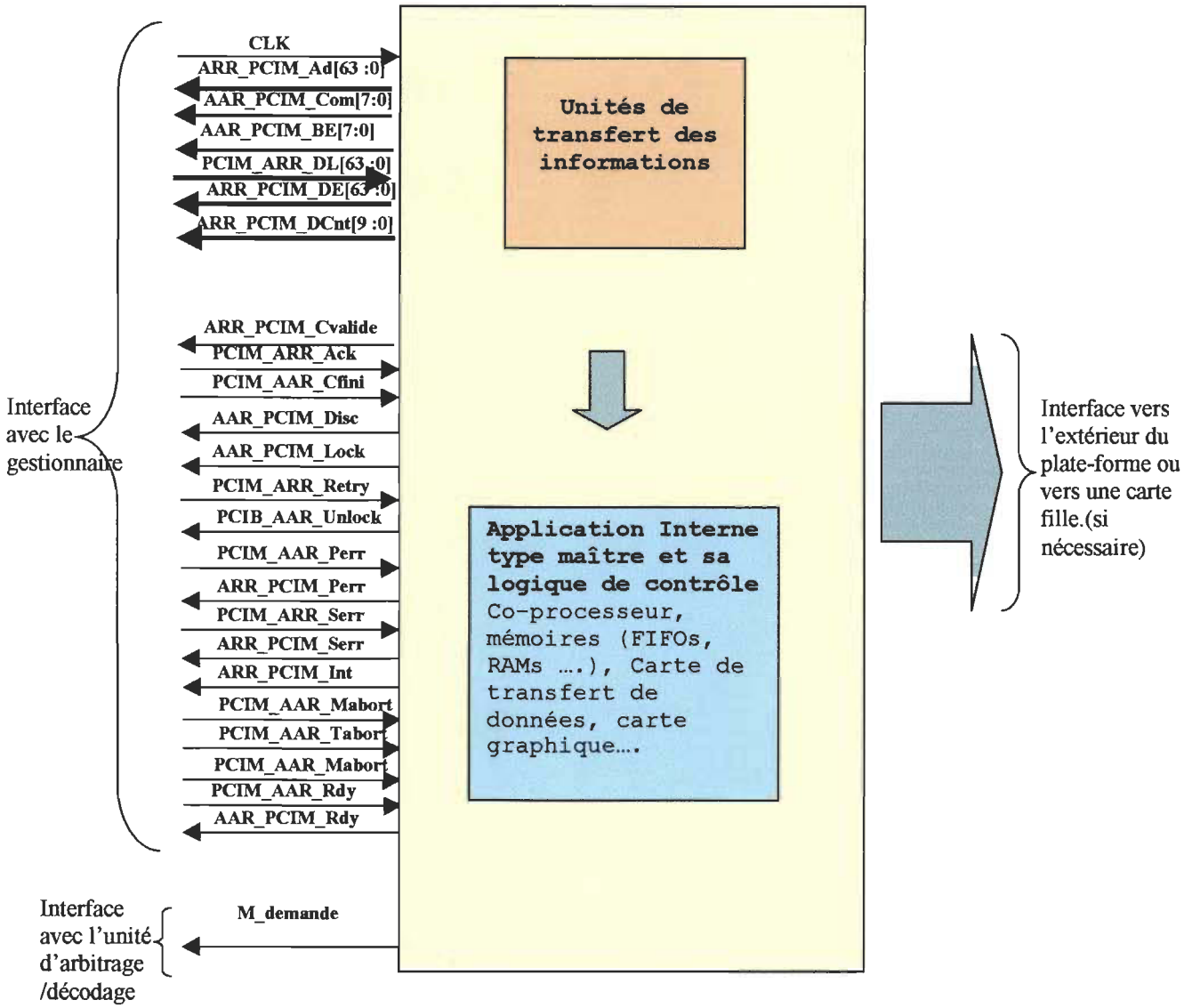

Figure 3.9 : Unité de l'application type maître

Il faut signaler que la structure de cette unité dépend de l'application intégrée dedans. Elle peut avoir une interface avec l'extérieur de la plate-forme, par exemple dans le cas d'une carte de communication, on a besoin d'une porte d'interface RJ45 ou une porte d'interface USB, pour la lier avec l'extérieur. Également, cette unité peut ne pas avoir une interface avec l'extérieur de la plate-forme comme dans le cas d'un coprocesseur dédié pour une application spécifique, car il reçoit les données à traiter de système hôte de la plate-forme et lorsqu'il achève son traitement des données, il demande l'accès au bus pour transférer le résultat traité vers le système hôte. Par ailleurs, Si l'application intégrée dans cette unité est extensible, ou l'espace occupé par ses composantes nécessite plus qu'une carte, l'interface vers l'extérieur peut se faire par une carte fille (daughter card).

Cette unité doit simplement demander la transaction sur le bus PCI en activant le signal **M_demande** pour l'unité d'arbitrage et quand elle reçoit le signal **PCIM_ARR_Ack,** activé par le gestionnaire, elle commence ses transactions sur le bus.
Notons que, son interface avec le gestionnaire type PCI est formée simplement par des bus d'adresse, de données et de commandes. Le bus de commande est souvent utilisé par l'application pour générer une transaction de lecture ou d'écriture (**AAR_PCIM_Com[7:0]**). La synchronisation de transfert entre cette unité et le gestionnaire s'effectue par les signaux **PCIM_AAR_Rdy** et **aar_pcim_rdy,** utilisés pour échanger les états d'attente entre eux. À coté de ces signaux liés au transfert de données, le gestionnaire fournit à cette unité des signaux de contrôle notamment : le signal **PCIM_AAR_Cfini** qui annonce l'achèvement du transfert, les signaux **PCIM_AAR_Serr** et **PCIM_AAR_Perr** qui indiquent si le gestionnaire a détecté des erreurs lors du transfert d'adresse ou données et des signaux qui informent cette unité si la cible de la transaction a lancé un *RETRY* ou *DISCONNECT* **(PCIM_ARR_Retry AAR_PCIM_Disc)**, etc. Signalons que l'utilisation de ces signaux de contrôle par cette unité dépend du type de l'application intégrée. Ces signaux peuvent ne pas être implantés dans l'application type maître si leurs existante n'est pas nécessaire.

L'application type maître peut utiliser ces signaux de contrôle pour informer le système hôte de son état en modifiant le contenu des registres d'états internes et qui seront lus à la suite par le système hôte quand il lance un transfert de lecture.

Signalons pour terminer que cette unité commande ses signaux de sortie au front montant de l'horloge et vu que le gestionnaire d'un PCI, d'après sa spécification, doit commander ses signaux au front montant de l'horloge ça lui permet d'avoir une période complète pour décoder une réponse au bus.

3.3.5 Unité de l'application arrière type cible

Cette unité a la capacité de répondre à un transfert demandé par le bus, elle possède trois interfaces : une interface avec le gestionnaire, une interface avec l'unité de ***Decodage_Arbitarge*** et une interface vers l'extérieur, et est formée de deux unités: l'unité de transfert des informations et l'unité de l'application interne type cible et sa logique de contrôle (qui peut être n'importe quelle application). Sa description globale est illustrée dans la figure 3.10.

L'unité de l'application type cible peut être n'importe quelle application qui ne maîtrise pas le bus. Prenons l'exemple d'un coprocesseur qui recevant une demande par le système hôte de la plate-forme, afin de traiter des données. Lorsqu'il achève son traitement, soit il attend que le système lit le résultat de son traitement, soit il peut utiliser une interruption pour demander le service de routine d'interruption du système hôte. D'autre part, cette unité peut avoir l'accès à l'extérieur de sa plate forme, en utilisant des portes de communication (série : USB, parallèle : LPT, ..), comme par exemple le cas d'un coprocesseur où le résultat de traitement est demandé par une autre plate forme. Si l'application intégrée dans cette unité est extensible, ou l'espace occupé par ses composantes nécessite plus qu'une carte, l'interface vers l'extérieur peut se faire par une carte fille (daughter card) .
Notons que son interface avec le gestionnaire type PCI est formée simplement par des bus d'adresse, données et commandes. Le bus de commande est utilisé par l'application

afin de connaître si la transaction courante est de lecture ou d'écriture (**PCIB_AAR_Com[7:0]**). La synchronisation de transfert entre cette unité et le gestionnaire s'effectue par les signaux **PCB_AAR_Rdy** et **AAR_PCB_Rdy**, utilisés pour échanger les états d'attente entre eux.

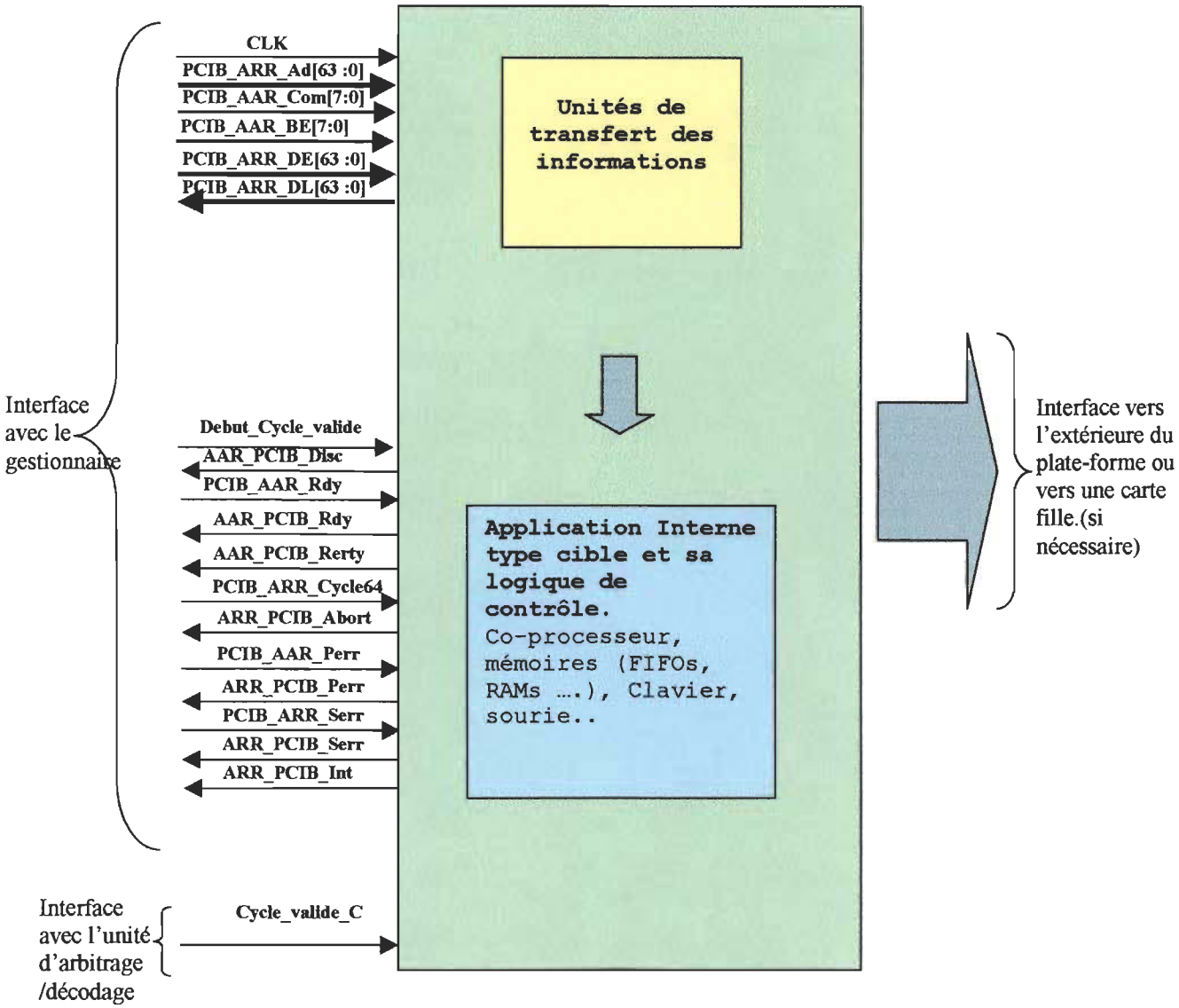

Figure 3.10 : Unité de l'application type cible

Cette unité doit répondre à la transaction courante de bus PCI lorsque le signal **Cycle_valide_C** est activé par l'unité de décodage.

À coté de ces signaux liés au transfert de données, le gestionnaire fournit à cette unité des signaux de contrôle comme : le signal de transfert **Debut_Cycle_valide** qui annonce le début du transfert, les signaux **PCIB_AAR_Serr** et **PCIB_AAR_Perr** qui indiquent

si le gestionnaire a détecté des erreurs lors du transfert d'adresse ou de données. Par ailleurs d'autres signaux (**AAR_PCIB_Disc** et **AAR_PCIB_Rerty** etc..) informent le gestionnaire si l'application type cible, intégrée dans cette unité, veut signaler *DISCONNECT* ou *RETRY* etc.. Signalons également que L'utilisation de ces signaux de contrôle par cette unité dépend du type de l'application intégrée. Ces signaux peuvent ne pas être implantés dans l'application type cible si leurs existante n'est pas nécessaire.

Par ailleurs l'application type cible peut utiliser ces signaux de contrôle pour informer le système hôte de son état en modifiant le contenu des registres d'états internes et qui seront lus à la suite par le système hôte quand il lance un transfert de lecture.

Signalons pour terminer que cette unité commande ses signaux de sorties au front montant de l'horloge et vu que le gestionnaire d'un PCI, d'après sa spécification, doit commander ses signaux au front montant de l'horloge ça lui permet d'avoir une période complète pour décoder une réponse au bus.

3.4. Développement du gestionnaire générique

3.4.1 Gestionnaire générique type PCI avec fonctionnalité Cible

À partir de chapitre 2, on a décrit l'architecture interne d'un gestionnaire cible type PCI et son interface avec le bus PCI, et d'après les structures proposées, dans ce chapitre, sur les composantes résidantes derrière ce gestionnaire pour assurer sa caractéristique générique, on a décomposé ce gestionnaire en neuf sous unités.
La figure 3.11 illustre les sous unités du gestionnaire, son interfaçage avec le bus PCI et les applications résidentes derrières lui : l'application arrière, l'espace de configuration et l'unité de ***Decodage_Arbitrage***.
À la suite, on présentera les structures des sous-unités de ce gestionnaire et leurs interfaçages.

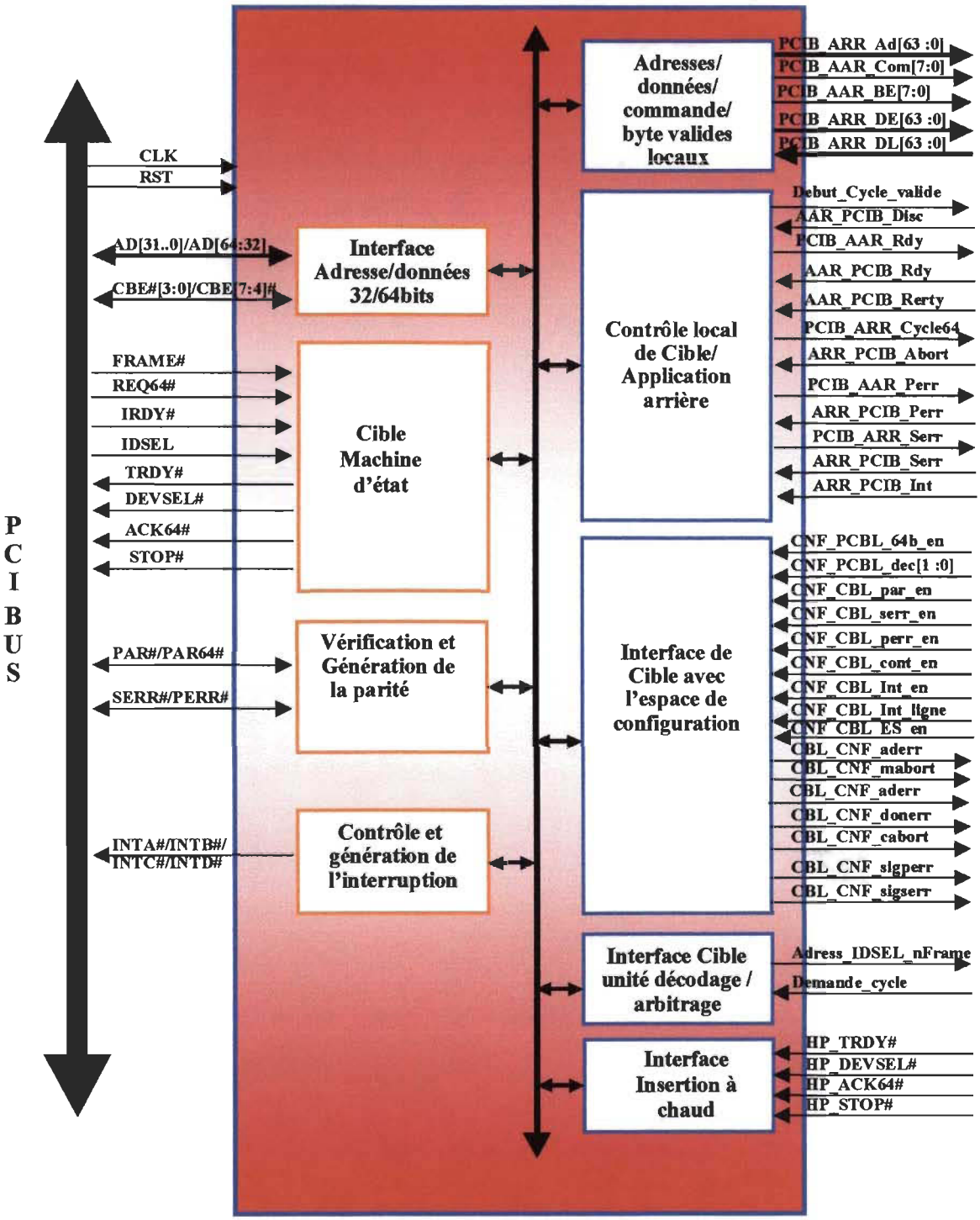

Figure 3.11 : Architecture interne d'un gestionnaire cible, ses interfaces avec le bus PCI et les applications arrières

3.4.1.1. Unité de l'interface Adresses / Données 32/64 bits

Cette unité a pour rôle de programmer les bus (**AD** et **CBE**) : en mode sortie lors de l'envoi des informations vers l'extérieur (cas d'une commande de lecture), et en mode d'entrée lors de la réception des informations de l'extérieur(cas d'une commande d'écriture). Par ailleurs, elle met à haute impédance ces bus **AD** et **CBE** lorsqu'elle n'est pas concernée par les transactions courantes sur le bus PCI.

Cette unité manipule les signaux d'entrée et de sortie du gestionnaire en acheminant directement les signaux d'entrée à l'intérieur de la logique de décodage, pour garantir une réponse rapide du gestionnaire vers l'émetteur (un tel maître) dès le prochain front montant de l'horloge qui suit l'émission des informations; et en commandant, au front montant de l'horloge également, les signaux de sorties par des bascules pour assurer la synchronisation entre l'émetteur (le gestionnaire) et le récepteur des données (figure 3.12).

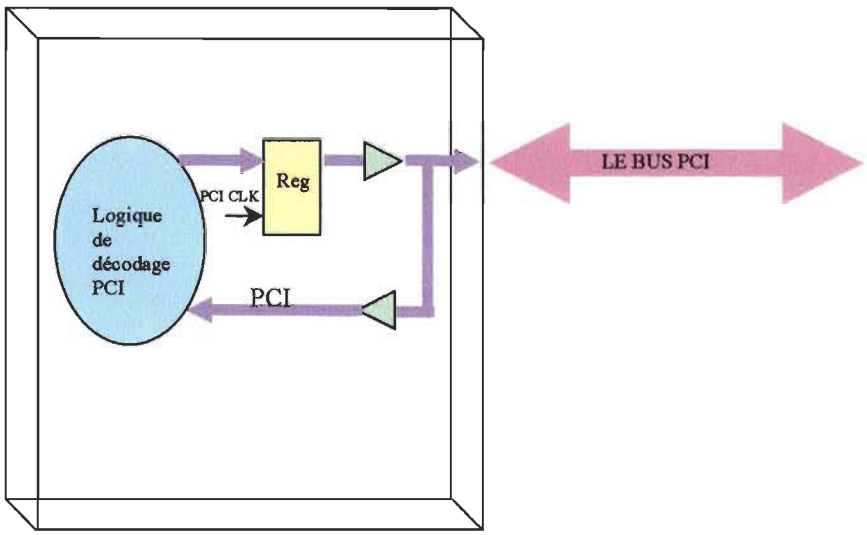

Figure 3.12 : Acheminement des entrées et des sorties du gestionnaire vers l'extérieur et l'intérieur.

3.4.1.2. Unité machine d'état Cible

Cette unité est le cœur du gestionnaire, elle est responsable de générer des états logiques à partir desquels tous les signaux externes (entrés / sorties) et les signaux internes du gestionnaire sont encodés. Cette machine d'état est divisée en deux sous-machines : Machine d'état de

transfert par la cible qui contrôle le transfert des informations et la réponse de la cible sur les transactions, et Machine d'état de commande de la fermeture de la cible par un maître. On va décrire les états de ces différentes sous machines et leurs fonctionnalités en adaptant une description algorithmique.

- **Machine d'état de transfert par la cible :**

Elle est constituée de sept états : **Repos, Decodage, C_Retry, Tran_donnees, C_Disconnect, C_Tabort, Retour** (figure 3.13).

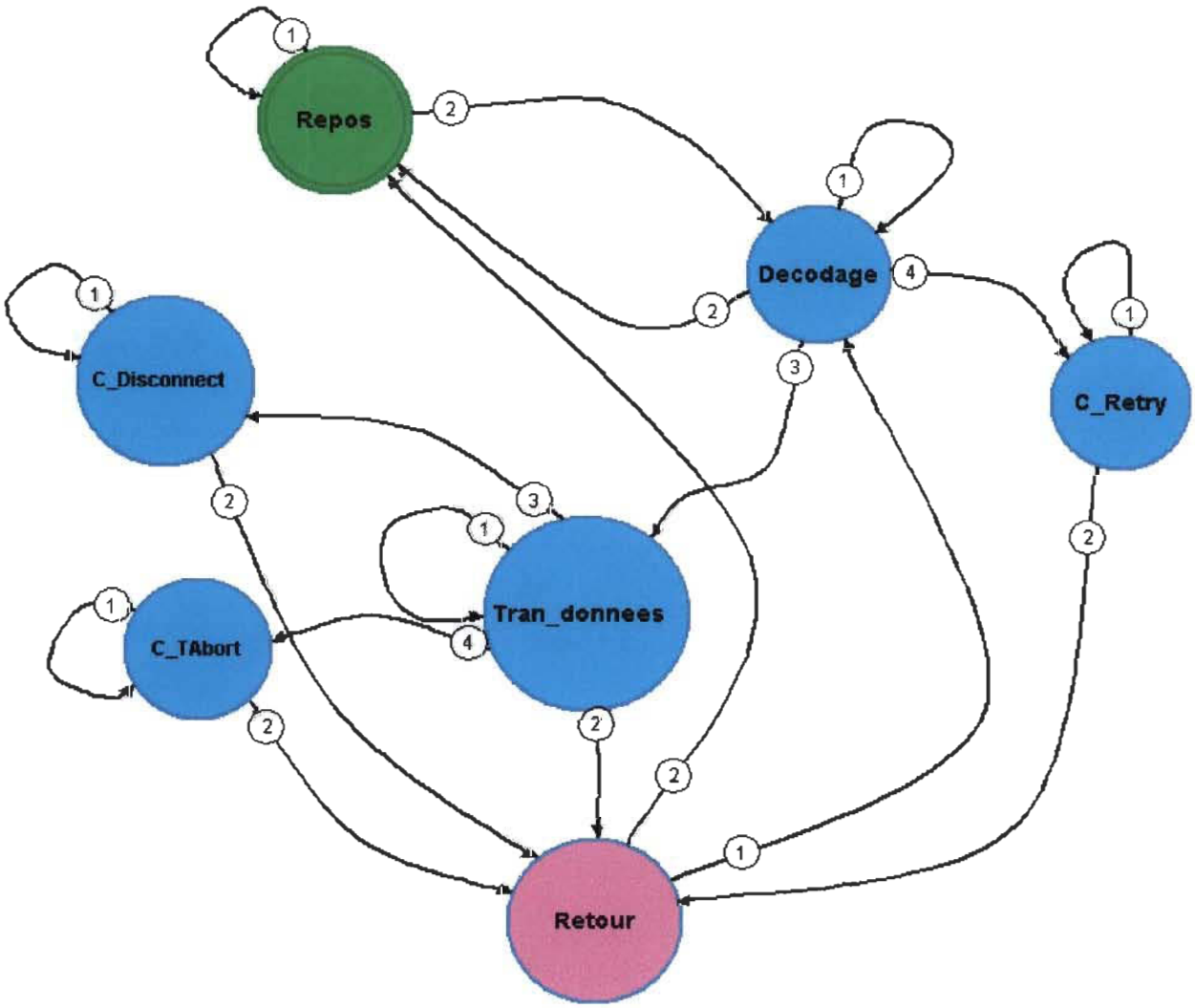

Figure 3.13 : Machine d'état de transfert par la cible

- Repos : État de repos : il n'y a aucun demande d'accès à la cible

1. *Aller* à **Repos,** si on n'a aucun accès à partir d'un maître donc **FRAME#** =1;
2. *Aller* à **Decodage,** si un maître commence une transaction par une phase d'adresse donc **FRAME#** =0;

- Decodage : Cible est occupée par le décodage de son adresse : Elle n'a pas encore accompli le décodage de l'adresse lancée par le maître. Le nombre de cycle dans lequel la cible demeure à cet état dépend de sa rapidité de décoder son adresse.

1. *Aller* **Decodage,** la cible reste à l'état **Decodage** si le décodage de son adresse n'est pas terminé.
2. *Aller* **Repos,** soit l'adresse ne correspond pas à la cible, soit le maître retire sa demande de transaction en mettant **FRAME#** =1;
3. *Aller* **Tran_donnees**, quand la transaction correspond à une adresse de la cible en activant **DEVSEL#**.
4. *Aller* **C_Retry,** si l'adresse correspond à la cible mais cette dernière veut signaler *RETRY* tandis que le maître n'a pas retiré encore sa demande de transfert.

- C_Retry : Cet état indique que la cible est occupée et elle veut signaler *RETRY*

1. *Aller* **C_Retry**, la cible active les signaux **STOP#** et **DEVSEL#** sans activer **TRDY#** et demeure à cet état tant que le maître n'a pas encore désactivé ses signaux de demande de transfert.
2. *Aller* **Retour,** si le maître a désactivé ses signaux de demande de transfert

- Tran_donnees : Cet état indique que la cible a accepté le transfert des données. Durant le temps du transfert, la cible demeure à cet état jusqu'à ce qu'un arrêt est lancé par le maître ou par la cible.

1. *Aller* **Tran_donnees,** si le transfert continue et aucun problème n'empêche la cible de continuer le transfert.
2. *Aller* **Retour,** si c'est la dernière phase de transfert, donc le maître a désactivé **FRAME#** avec **IRDY#** activé.
3. *Aller* **C_Disconnect,** quand la cible veut signaler *DISCONNECT*,
4. *Aller* **C_TAbort,** quand la cible veut signaler *TARGET_ABORT*,

- **C_Disconnect** : Cet état indique que la cible veut interrompre la transaction courante et elle veut signaler *DISCONNECT.*

1. *Aller* **C_Disconnect,** la cible active les signaux **STOP#** et **DEVSEL#** en désactivant **TRDY#** et demeure à cet état tant que le maître n'a pas encore désactivé ses signaux de demande de transfert.
2. *Aller* **Retour,** si le maître a désactivé ses signaux de demande de transfert.

- **C_TAbort** : Cet état indique que la cible veut terminer à cause d'une erreur rencontrée et elle veut signaler *TARGET_ABORT.*

1. *Aller* **C_TAbort,** la cible active les signaux **STOP#** en désactivant **DEVSEL#** et **TRDY#** et demeure à cet état tant que le maître n'a pas encore désactivé ses signaux de demande de transfert.
2. *Aller* **Retour,** si le maître a désactivé ses signaux de demande de transfert.

- **Retour** : Cet état indique la dernière phase de transfert : La cible utilise cette phase pour désactiver ses signaux de commande type : trois états soutenus(sustained tristate) avant de les mettre à haute impédance pour que ces signaux puissent être commandés par une autre cible.

1. *Aller* **Decodage**, si le maître a réactivé une nouvelle demande de transaction.
2. *Aller* **Repos,** si le maître n'a pas réactivé de nouveau ses signaux de demande de transfert.

- **Machine d'état de commande de la fermeture de la cible par un maître**

Elle est constituée de deux états : **LIBREE, FERMEE** (figure 3.14).

Figure 3.14 : Machine d'état de transfert par la cible

- **LIBREE** : Cet état indique que la cible est libre pour répondre à toutes les transactions

1. *Aller* **FERMEE,** si le maître a demandé un accès de fermeture de la cible en activant le signal **LOCK#**.
2. *Aller* **LIBREE** dans le cas contraire;

- FERMEE : la cible ne peut plus répondre sauf au maître qui l'a fermé;

1. *Aller* **LIBREE,** si le maître a désactivé son signal de fermeture.
2. *Aller* **FERMEE**, dans le cas contraire

3.4.1.3. Unité de vérification et de génération de la parité

Cette unité a pour rôle de calculer la parité entre les bus **AD** et **CBE** quand le maître exécute une commande de lecture ou écriture sur sa marge d'adressage.

Il est intéressant de signaler que dans le cas d'une lecture exécutée par un maître de côté de bus PCI sur ce gestionnaire type cible, cette unité génère la parité mais à condition que le flag de la génération de cette parité soit activé dans l'espace de configuration de l'application cible courante connectée au gestionnaire. Autrement dit, le signal **CNF_CBL_par_en,** émet par cet espace de configuration et reçu par le gestionnaire cible, doit être activé. Par ailleurs, la parité pour une donnée émise par la cible à l'instant t doit être validée au bus PCI à l'instant t+1 pour que le maître qui reçoit cette donnée à l'instant t, calcule la parité lui-même et la compare à l'instant t+1 par la parité générée par la cible.

Également, dans le cas d'une écriture d'un maître exécuté par un maître sur ce gestionnaire type cible, cette unité vérifie la parité, et à la suite, elle active le signal **SERR#** dans le cas de la détection d'une erreur durant une phase d'adresse, soit elle active le signal **PERR#** dans le cas de la détection d'une erreur durant une phase de donnée. L'activation de ces deux signaux est liée à l'état de la configuration de ce gestionnaire s'il a la permission de les générer. Donc les signaux d'interconnexion entre le gestionnaire et l'espace de configuration notamment **CNF_CBL_Serr_en** et **CNF_CBL_perr_en** sont activés respectivement.

Cependant, lors de la détection de l'une de ces erreurs, cette unité doit modifier les emplacements mémoires correspondants à ce type d'erreur dans l'espace de configuration, en activant les signaux d'interconnexion entre le gestionnaire et l'espace de configuration **CBL_CNF_sigserr** ou **CBL_CNF_sigperr.**

3.4.1.4. Unité de contrôle et génération de l'interruption.

Cette unité a pour rôle de générer l'interruption au bus PCI via le signal **INT#** quand l'application arrière demande un service d'interruption du système hôte en activant le signal **ARR_PCIB_Int**. Il faut noter que l'utilité de cette unité, d'accepter de générer une interruption, et de connaître sur quelle ligne d'interruption (A, B, C ou D) elle va être générée, dépendent aux informations obtenues par l'espace de configuration via les signaux **CNF_CBL_Int_en** et **CNF_CBL_Int_ligne.**

3.4.1.5. Unité d'adresses / données / commandes / bytes valides locaux.

Cette unité a pour rôle d'interchanger via des signaux d'interconnexion avec le gestionnaire et l'application arrière les informations : adresse, données, commandes et les bytes valides locaux..

Notons que ces signaux d'interconnexion sont choisis comme des signaux à deux états pour deux raisons :

- pour laisser les pins type trois états aux signaux d'interconnexion entre le gestionnaire et le bus PCI, dans le cas où ce gestionnaire est intégré dans un circuit programmable séparé de l'application arrière.
- pour ne pas consommer toutes les ressources des mémoires tampons trois états et des mémoires internes du circuit programmable dans le cas où ce gestionnaire est intégré avec l'application arrière dans la même puce.
- pour simplifier la conception des applications résidentes derrières ce gestionnaire au niveau d'interconnexion avec le gestionnaire, et de simplifier ainsi par la logique de commandes de ces signaux d'interconnexion.

3.4.1.6. Unité contrôle local de la Cible / Application arrière

Cette unité a pour rôle d'interchanger les informations de contrôle via des signaux d'interconnexion entre la cible et les applications arrières. Ces signaux de contrôle sont utilisés

pour ajouter les états d'attentes, signaler les arrêts, les interruptions, le début d'un transfert, et ainsi les erreurs entre le gestionnaire et les applications arrières.

Par exemple, dans les cas de la signalisation des arrêts par les applications arrières : si une application arrière veut arrêter le transfert, elle active le signal **AAR_PCIB_Disc** pour que cette unité informe la Machine d'état de transfert par la cible et effectue le changement de l'état **Tran_donnees** à l'état **C_Disconnect** afin de signaler *DISCONNECT* au maître lié à la transaction courante. Et par ailleurs, dans le cas de la signalisation des erreurs pendant une phase d'adresse, cette unité active le signal **PCIB_ARR_SERR** pour informer l'application arrière.
Il faut noter que l'utilisation de ces signaux par l'application arrière dépend de la nature de cette application. Autrement dit, s'ils ne sont pas utiles pour cette application, cette unité peut les ignorer ou les maintenir déconnectés.

3.4.1.7. Unité de l'interface de la cible avec l'espace de configuration

Cette unité a trois rôles :

- fournir au gestionnaire l'état de la configuration des applications par le système hôte.

Par exemple, Quand le système de configuration hôte active l'accès aux espaces d'E/S de gestionnaire par l'activation du bit 0 dans le registre de commande de son espace d configuration. Par conséquent, l'activation de ce bit active à son tour le signal **CNF_CBL_ES_en** de cette unité d'interface pour donner la permission au gestionnaire de répondre aux accès à son espace d'E/S.

- fournir au gestionnaire les caractéristiques programmées par les constructeurs pour les applications arrières

Par exemple, si une application arrière supporte un décodage rapide, donc son emplacement dans l'espace de configuration dans le registre d'état (bits 9 et 10) doit être programmé à une valeur correspondante à ce décodage. Cette valeur sera transmise au gestionnaire via le signal **CNF_PCBL_dec[1:0]** lié à cette unité de transfert.

- mettre à jour les espaces de configuration des applications arrières pour informer le système, lorsqu'il génère des commandes de lectures sur ces espaces de configuration sur les modifications produites.

Par exemple, dans le cas de la détection de l'erreur de parité, cette unité doit mettre à jour le registre d'état (bit 14) afin d'informer le système de la détection de cette erreur, en utilisant le signal **CBL_CNF_sigperr**.

3.4.1.8. Unité de l'interface de la cible avec l'unité de Decodage_Arbitrage

Cette unité a pour rôle de fournir à l'unité de **Decodage_Arbitrage** l'adresse détectée sur le bus PCI par ce gestionnaire via le signal **FRAME#**, donc l'unité de **Decodage_Arbitrage** peut connaître le début de la transaction afin d'effectuer le décodage de l'adresse.

À la suite, cette unité fournit le signal **IDSEL** pour que l'unité de **Decodage_Arbitrage** puisse faire la distinction entre l'accès sur une application arrière ou sur un espace de configuration.

3.4.1.9. Unité de l'interface de l'insertion à chaud

Il est intéressant de savoir qu'un contrôleur de l'insertion à chaud est un dispositif, communiquant avec le système, pour configurer des nouvelles cartes insérées sur le bus durant le fonctionnement du système global. Cette configuration appelée configuration à chaud est utilisée pour ne pas configurer le système global lors de l'insertion d'une nouvelle carte

Si une application, parmi les applications arrières utilisant ce gestionnaire, est un contrôleur de l'insertion à chaud, le gestionnaire doit avoir la capacité de générer la séquence d'initialisation pour une carte insérée à chaud sur le bus où elle réside.

Cependant, ce contrôleur commande les signaux, liés à cette unité, **HP_TRDY#, HP_STOP#, HP_ACK64# HP_DEVSEL#** pour initialiser la carte insérée par un modèle d'initialisation selon le tableau 2.13. Le gestionnaire, de son tour, doit commander les signaux **DEVSEL#, ACK64#, STOP#** et **TRDY#** sur le bus PCI avec la même séquence que celle des signaux d'interface de contrôleur..

3.4.2 Gestionnaire générique type PCI avec fonctionnalité Maître

D'après le chapitre 2, on a décrit l'architecture interne d'un gestionnaire maître type PCI et son interface avec le bus PCI et d'après les structures proposées, dans ce chapitre, sur les composantes résidantes derrière ce gestionnaire pour assurer sa caractéristique générique, on a décomposé ce gestionnaire en huit unités (figure 3.15). Ces unités sont semblables à celles décrites pour le gestionnaire type cible, figure (3.11), mais de fonctionnalité différente.
Dans ce qui suit, on présentera la différence entre ces unités non semblables (type maître et cible),

3.4.2.1. Unité de l'interface Adresses / Données 32/64 bits

Il faut signaler que du fait que les commandes sur le bus PCI sont interprétées par rapport au maître, cette unité doit programmer les signaux **AD** et **CBE** en sortie pour une commande d'écriture, et mode d'entrée pour une commande de lecture (vice versa pour un gestionnaire type cible).

3.4.2.2. Unité machine d'état maître

Cette unité est le cœur du gestionnaire, elle est responsable de générer des états logiques à partir desquels tous les signaux externes (entrés / sorties) et les signaux internes du gestionnaire sont encodés.

Cette machine d'état est divisée en deux sous-machines : Machine d'état de transfert par le maître qui contrôle le transfert des informations lancées par ce maître et Machine d'état de commande de la fermeture d'une cible par le maître qui contrôle la fermeture d'une cible par le maître sur le bus PCI.

À la suite, les états de ces sous machines et leurs fonctionnalités vont être décrits en adoptant une description algorithmique.

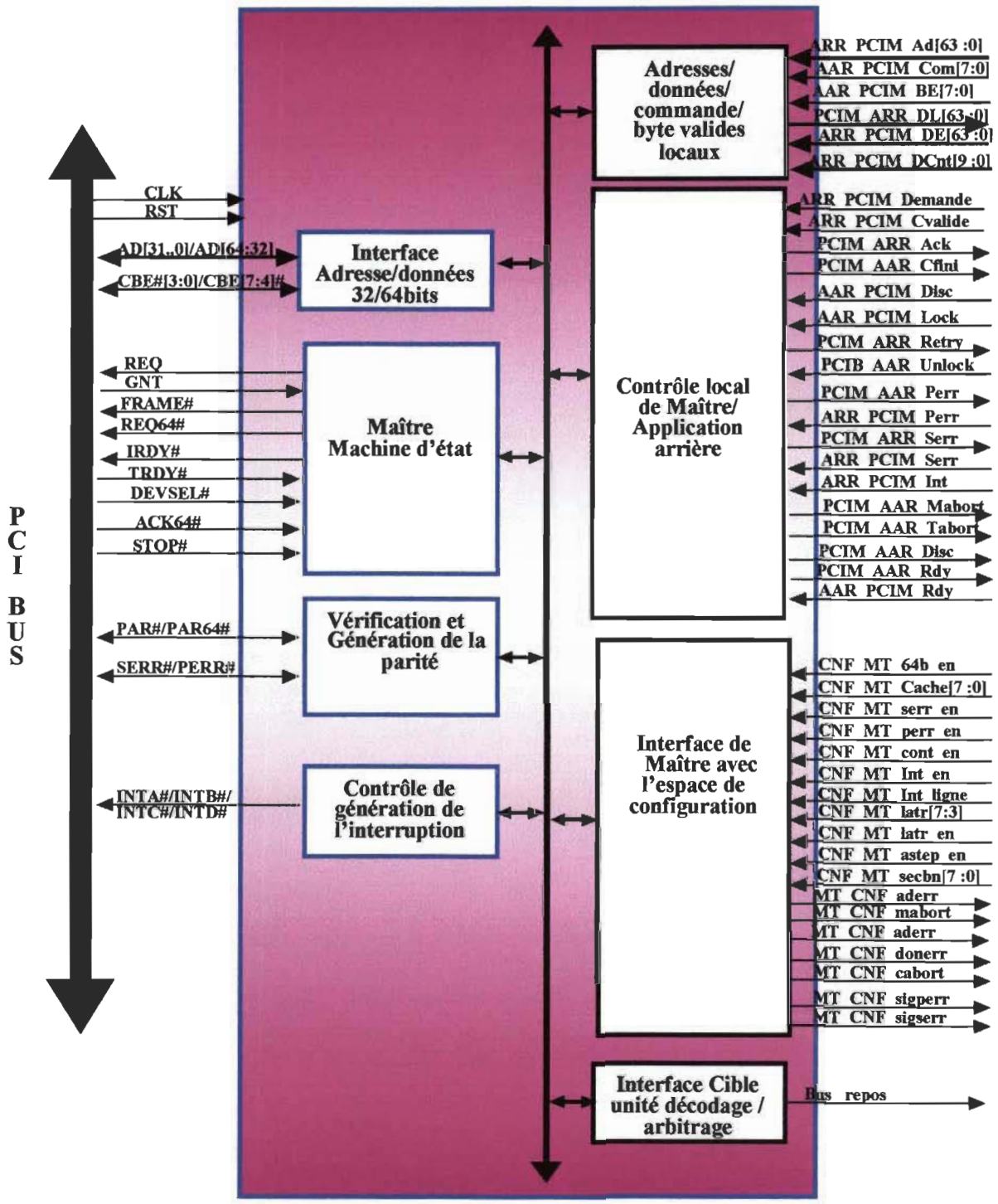

Figure 3.15 : Architecture interne d'un gestionnaire maître et ses interfaces avec le bus PCI et les applications arrières

- **Machine d'état de transfert par le maître**

Elle est constituée de dix états : **Repos, Demande_bus, Adresse, Adresse_double, Trans_donnees, Signale_Mabort, Signale_Retry, Signale_Tabort, Park_bus, Retour** (voir figure 3.16).

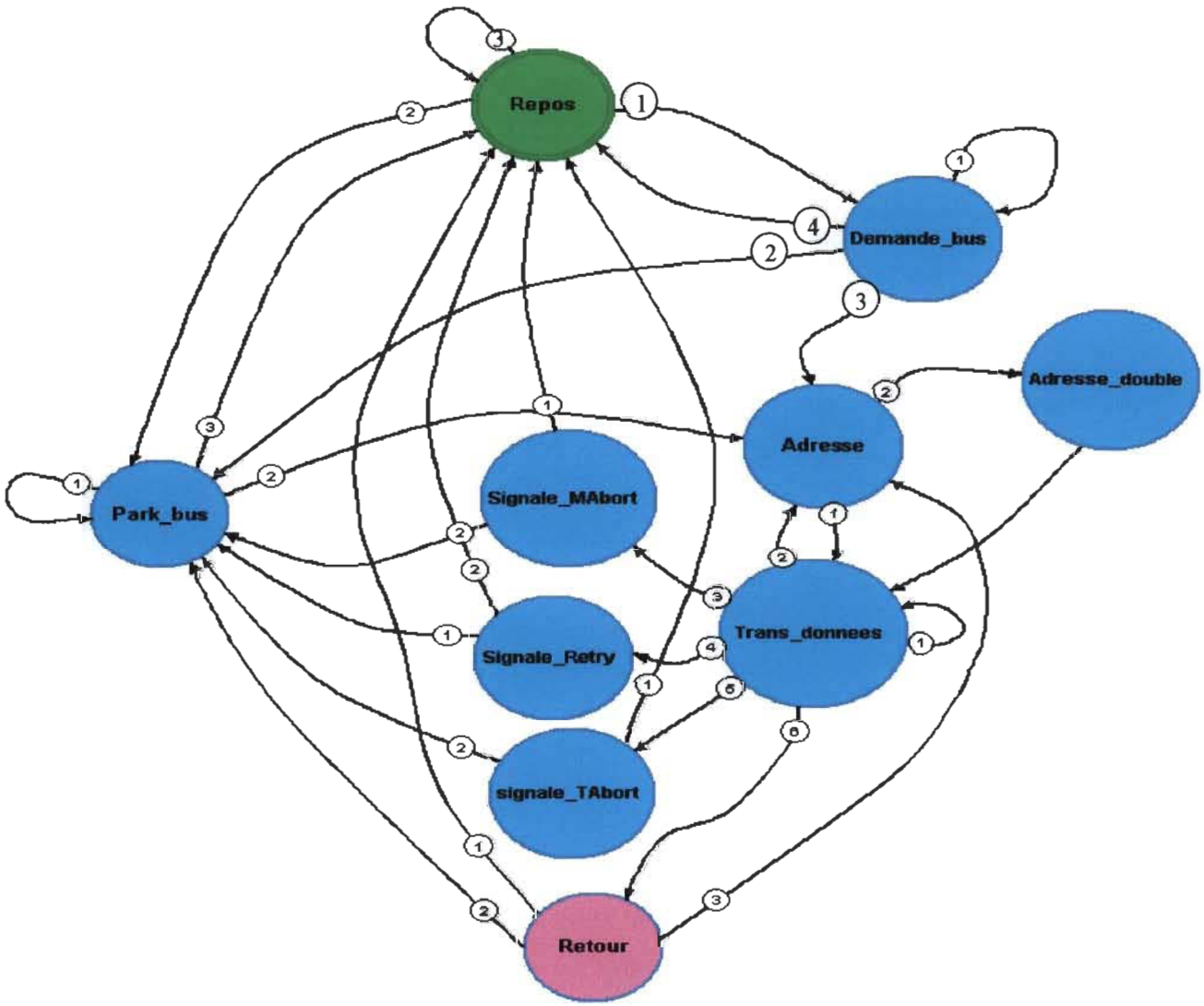

Figure 3.16 : Machines de transfert par le maître

-Repos : C'est l'état de repos lorsqu'il n'existe aucune transaction sur le bus PCI;

1. *Aller* **Demande_bus,** quand l'application arrière demande un transfert de donnée, en activant le signal **AAR_ PCIM_Demande,**

2. *Aller* **Park_bus,** quand il n'y a pas des transactions sur le bus PCI et que l'application arrière n'a pas une transaction à transférer mais l'arbitre de bus PCI veut que ce maître commande les signaux **AD** et **CBE**, sur le bus PCI, pour minimiser leurs temps de flottement.
3. *Aller* **Repos,** Dans les cas contraires.

-Demande_bus : À cet état, le maître demande l'accès au bus PCI en activant le signal **REQ,**

1. *Aller* **Demande_bus,** quand l'arbitre de bus PCI n'a pas activé le signal **GNT#** pour donner au maître l'accès au bus
2. *Aller* **Park_bus**, quand l'application arrière a désactivé sa demande de transfert et que l'arbitre a accordé au maître l'accès au bus.
3. *Aller* **Adresse**, quand la demande de l'application arrière demeure activée et que l'arbitre de bus PCI active le signale **GNT#**
4. *Aller* **Repos**, Dans les cas contraires.

-Adresse : À cet état, le maître commande l'adresse de la transaction sur le bus PCI dans le cas d'une commande simple ou le poids faible de l'adresse d'une transaction utilisant la commande *Dual Address Cycle* ;

1. *Aller* **Trans_donnees,** au prochain front montant de l'horloge si on a cas d'adresse simple (adressage d'une cible implantée dans l'espace mémoire inférieure à 4 Gigaoctets).
2. *Aller* **Adresse_double,** Cas de la génération d'une commande *Dual Address Cycle* (adressage d'une cible implantée dans l'espace mémoire supérieur à 4 Gigaoctets).

-Adresse_double, : À cet état, le maître commande le poids fort de l'adresse d'une transaction utilisant la commande *Dual Address Cycle*;

Aller **Trans_donnees**, au prochain front montant de l'horloge.

-Trans_donnees : À cet état, le maître transfère les données.

1. *Aller* **Trans_donnes,** le maître doit maintenir à cet état si l'une de ces trois conditions est réalisée :

i. Si au début de la transaction, la cible, liée au transfert, n'a pas encore répondu au transfert (**DEVSEL#** reste désactivé) et que le temps de réponse maximale d'une cible, 6 périodes d'horloge, n'a pas expiré.

ii.

a- Si **DEVSEL#** est activé avant la 6éme période, d'horloge donc la cible a répondu au transfert,

b- le nombre de bytes à transférer, obtenu par le gestionnaire par le signal **ARR_PCIM_DCnt[9:0]**, n'est pas achevé,

c- le maître n'est pas arrêté (preempted) par l'arbitre en désactivant **GNT#**,

d- le temps de l'accès sur le bus conféré au maître par le système via le signal de l'espace de configuration **CNF_MT_latr[7:3]** n'est pas expiré.

e- Si on a une commande autre que **MWI** et que la cible n'a pas activé une fin de la transaction due à *RETRY* ou *DISCONNECT* ou *Target_Abort.*

Ces conditions doivent être effectuées simultanément.

iii.

Dans le cas d'une commande **MWI,** le maître doit maintenir à l'état **Trans_donnees** dans les mêmes conditions de ii(a-d), mais lorsque **GNT#** sera désactivé et le *Latency_Timer* détecté par le gestionnaire via le signal **CNF_MT_latr[7:3]** sera expiré, le maître doit continuer le transfert jusqu'à ce que la limite d'accès à une ligne de cache mémoire soit expirée. Donc, le maître doit vérifier le signal de l'espace de configuration **CNF_MT_Cache[7:0]** qui indique la taille d'une ligne de cache.

2. *Aller* **Adresse,** si le maître a terminé le transfert de données en échangeant tous les bytes de la transaction courante, et que sa maîtrise de bus reste activé (**GNT# =0**) et en même temps qu'il y a une autre demande de transfert lancée par une de ces applications arrières (**ARR_PCIM_Demande** =1);
3. *Aller* **Signale_Mabort,** quand la cible de la transaction n'a pas répondu après 6 périodes de la phase d'adresse, le maître doit arrêter le transfert et signaler *Master_Abort*,
4. *Aller* **Signale_Retry**, quand la cible a activé les deux signaux **DEVSEL#** avec **STOP#** sans activer **TRDY#** lors du commencement du transfert pour signaler *RETRY*, ou elle a activé **DEVSEL#** avec **STOP#** après transfert de données pour signaler *DISCONNECT*,

5. *Aller* **Signale_Tabort,** quand la cible veut arrêter le transfert après un certain nombre de phases de données à cause d'une erreur en signalant au *Target_Abort*.
6. *Aller* **Retour**, Dans les conditions contraires.

- Signale_Mabort: Arrêt de la transaction lancée par la cible dû à un *Master_Abort*. Dans cet état le maître active le signal **PCIM_AAR_Mabort** lié à l'application arrière pour l'informer que la cible de la transaction n'a pas répondu à son transfert. Par ailleurs, le maître met à jour le registre d'état de l'espace de configuration lié à cette application en activant le signal **MT_CNF_mabort** de son interface avec l'espace de configuration.

1. *Aller* **Repos**, si **GNT#** est désactivé; le maître doit céder le bus à un autre maître;
2. *Aller* **Park_bus**, si **GNT#** reste activer; le maître doit commander le bus.

- Signale_Retry: Arrêt de la transaction lancée par la cible dû à un *Retry* or *DISCONNECT*. Dans cet état le maître active le signal **PCIM_ARR_Retry** lié à l'application arrière pour l'informer que la cible de la transaction a interrompu son transfert

1. *Aller* **Repos**, si **GNT#** est désactivé; le maître doit céder le bus à un autre maître;
2. *Aller* **Park_bus**, si **GNT#** reste activer; le maître doit commander le bus.

- Signale_Tabort: Arrêt de la transaction lancée par la cible dû à un *Target_Abort*. Dans cet état le maître active les signaux **PCIM_AAR_Tabort** et **MT_CNF_cabort** pour informer à l'application arrière de cet arrêt et pour mettre à jour l'espace de configuration de cette application respectivement.

1. *Aller* **Repos**, si **GNT#** est désactivé; le maître doit céder le bus à un autre maître;
2. *Aller* **Park_bus**, si **GNT#** reste activer; le maître doit commander le bus.

-Retour : Cet état indique que la transaction est terminée. Cependant Le maître utilise cette phase pour désactiver ses signaux de commande type trois états soutenus (sustained tristate) avant de les mettre à haute impédance pour que ces signaux puissent être commandés par un autre maître.

1. *Aller* **Repos**, si **GNT#** est désactivé.
2. *Aller* **Park_bus**, si on n'a pas une transaction demandée par une application arrière mais l'arbitre laisse le signal **GNT#** activé, dans ce cas, le maître doit commander le bus.

3. *Aller* **Adresse**, si la maîtrise de bus du maître reste activé (**GNT# =0**) et qu'il y a une autre demande de transfert lancée par une application arrière(**ARR_PCIM_Demande** =1);

- **Park_bus** : l'arbitre de bus PCI veut que ce maître commande les signaux **AD** et **CBE**, sur le bus PCI, pour minimiser leurs temps de flottement.

1. Aller **Park_bus**, s'il n'y a aucune demande de transfert de l'application arrière et que **GNT#** reste activé.
2. *Aller* **Adresse,** s'il y a une demande d'une application arrière est que **GNT#** est encore activé.
3. *Aller* **Repos,** si **GNT#** est désactivé.

- **Machine d'état de commande de la fermeture d'une cible par le maître.**

Elle est constituée de deux états : **LIBRE, OCCUPEE** (figure 3.17).

Figure 3.17 : Machine d'état de commande de la fermeture d'une cible par le maître

-LIBRE : Cet état indique que le signal **LOCK#** n'est pas utilisé par un maître;

1. *Aller* **LIBRE,** si le signal **LOCK#** est désactivé donc il n'est pas utilisé par un maître;
2. *Aller* **OCCUPEE** si le maître détecte que le signal **LOCK#** est activé.

-OCCUPEE: le signal **LOCK#** est utilisé par un maître.

1. *Aller* **LIBRE**, lorsque les signaux **LOCK#** et **FRAME#** sont désactivés simultanément; le maître détecte que le signal **LOCK#** est libre;
2. *Aller* **OCCUPEE** dans le cas contraire.

3.4.2.3. Unité contrôle local de Maître / Application arrière

Cette unité a le même rôle que l'unité **contrôle locale de cible / Application arrière** sauf que ses signaux sont liés au maître. Par exemple le signal **ARR_PCIM_Demande** indique au maître une requête de transaction générée par l'application arrière et le signal

ARR_PCIM_Cvalide est utilisé pour valider l'adresse commandée par l'application arrière dans le bus **ARR_PCIM_Ad[63:0]**.

La négociation du transfert entre le gestionnaire et les applications arrières s'effectue par les signaux **PCIM_ARR_Rdy** et **ARR_PCIM_ Rdy** de cette unité.

Le maître fournit le résultat du transfert à l'application arrière en utilisant des signaux notamment **PCIM_AAR_Mabort** et **PCIM_AAR_Tabort.** Notons que, l'utilisation de ses signaux dépend de l'application arrière. Prenons le cas où le maître a détecté une erreur lors d'une phase de données, il l'en informe à l'application arrière via le signal **ARR_PCIM_Perr**. Par ailleurs, l'application arrière juge, selon son type, si c'est nécessaire de reprendre le transfert de cette donnée corrompue. Toutefois, quand l'application exécute une commande sur un espace de configuration d'une cible connectée au bus PCI, il est souvent nécessaire de reprendre la donnée défectueuse. Dans le cas d'une application multimédia, la manque d'une phase de données n'influe pas beaucoup sur la qualité d'une image ou d'un son d'une chanson, dans ce cas l'application peut ne pas reprendre la phase de donnée perdue.

3.4.2.4. Unité de l'interface de maître avec l'espace de configuration

Cette unité a pour rôle d'informer le gestionnaire sur l'état de configuration de l'application arrière type maître, par exemple le signal **CNF_MT_Cont_en** informe le gestionnaire si l'application arrière supporte ou non le transfert en mode continue. Par ailleurs, cette unité est utilisée pour mettre à jour, par le gestionnaire type maître, l'état de l'application arrière dans l'espace de configuration.

3.4.2.5. Unité de l'interface Maître / Decodage_Arbitrage

Cette unité a pour rôle d'informer l'unité de **Decodage_Arbitrage** que le bus est à l'état de repos pour qu'elle puisse accepter les demandes d'accès au bus par les applications arrières type maître.

Notons pour terminer que les unités : **Vérification et génération de la parité**, **Contrôle de la génération de l'interruption**, et **Adresse/donnée/commandes/bytes valides locaux** ont la même fonctionnalité que les unités contreparties dans le gestionnaire type cible.

3.4.3 Gestionnaire générique type PCI/PCI-X

Comme on a vu dans le chapitre 2, un gestionnaire type PCI-X doit inclure la possibilité du fonctionnement en mode PCI, donc il doit fonctionner comme un gestionnaire type PCI dans le cas de son existence dans un bus où des gestionnaires type PCI sont connectés. C'est la raison pour laquelle qu'on a introduit l'unité de sélection mode PCI-X qui permet à la logique interne du gestionnaire de permuter entre les modes PCI et PCI-X.

Par ailleurs, on a déjà cité qu'un gestionnaire de type PCI-X doit échantillonner ses signaux d'entrées et de sorties. Cependant l'architecture interne d'un gestionnaire type PCI-X doit avoir la forme illustrée à la figure 3.18.

Notons que d'après cette figure 3.18, le gestionnaire possède deux chemins d'entrés séparés à sa logique interne, un chemin échantillonné utilisé pour sa logique type PCI-X; et un chemin directement connecté à sa logique type PCI. Le passage d'une logique PCI à une logique PCI-X se fait par le signal PCIX_Selection qui a pour rôle la validation de la capacité du gestionnaire de fonctionnement en mode PCI-X.

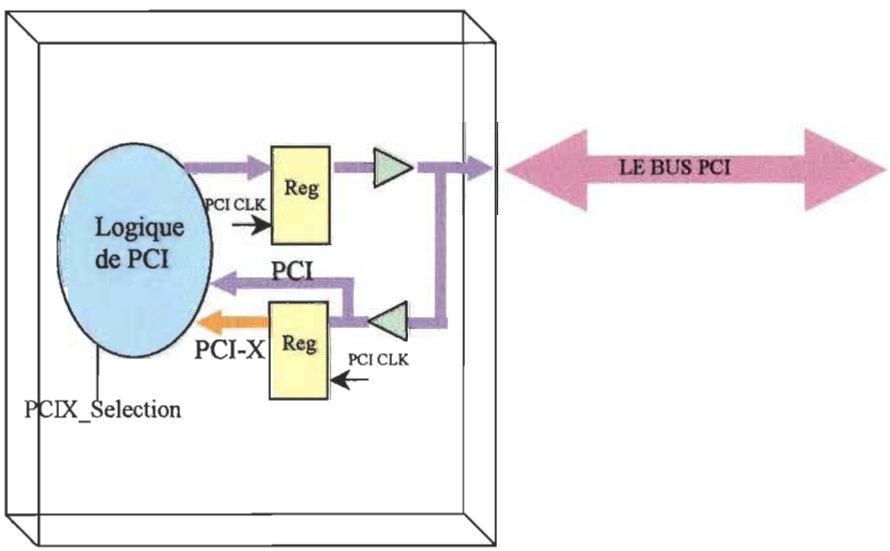

Figure 3.18 : Architecture interne de gestionnaire type PCI/PCI-X

Donc chaque signal interne doit être une sortie d'un multiplexeur commandé par le signal **PCIX_Selection**. Ce multiplexeur doit choisir à ce signal interne une logique pour le mode PCI quand le signal **PCIX_Selection** = 0 et une logique pour le mode PCI-X quand le signal **PCIX_Selection** =1 (figure 3.19).

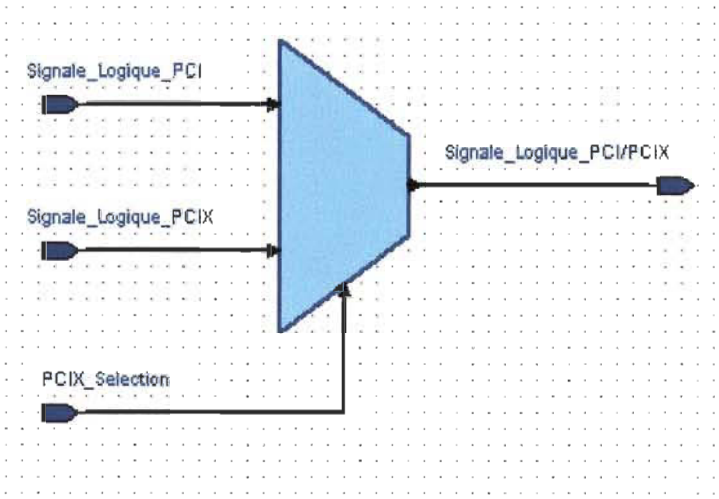

Figure 3.19 : Permutation entre les logiques type PCI et PCIX d'un signal interne

Ainsi d'après le chapitre 2, à cause de la commande *SPLIT Transaction*, une cible type PCI-X doit inclure la capacité maître pour demander l'accès au bus PCI-X afin d'acheminer les données demandées par un maître (initiateur) durant une transaction divisée. De même un maître doit avoir la capacité cible afin d'accepter les réponses des transactions divisées adressées vers lui.

D'après ces conditions, la figure 3.20 représente l'architecture interne du gestionnaire Cible/Maître où on a rassemblé les unités des deux gestionnaires type cible et maître et en ajoutant une logique pour son fonctionnement en mode PCI-X.

À la suite; on décrira les nouvelles unités par rapport au gestionnaire type PCI tout en présentant la différence entre les unités communes avec les gestionnaires précédents.

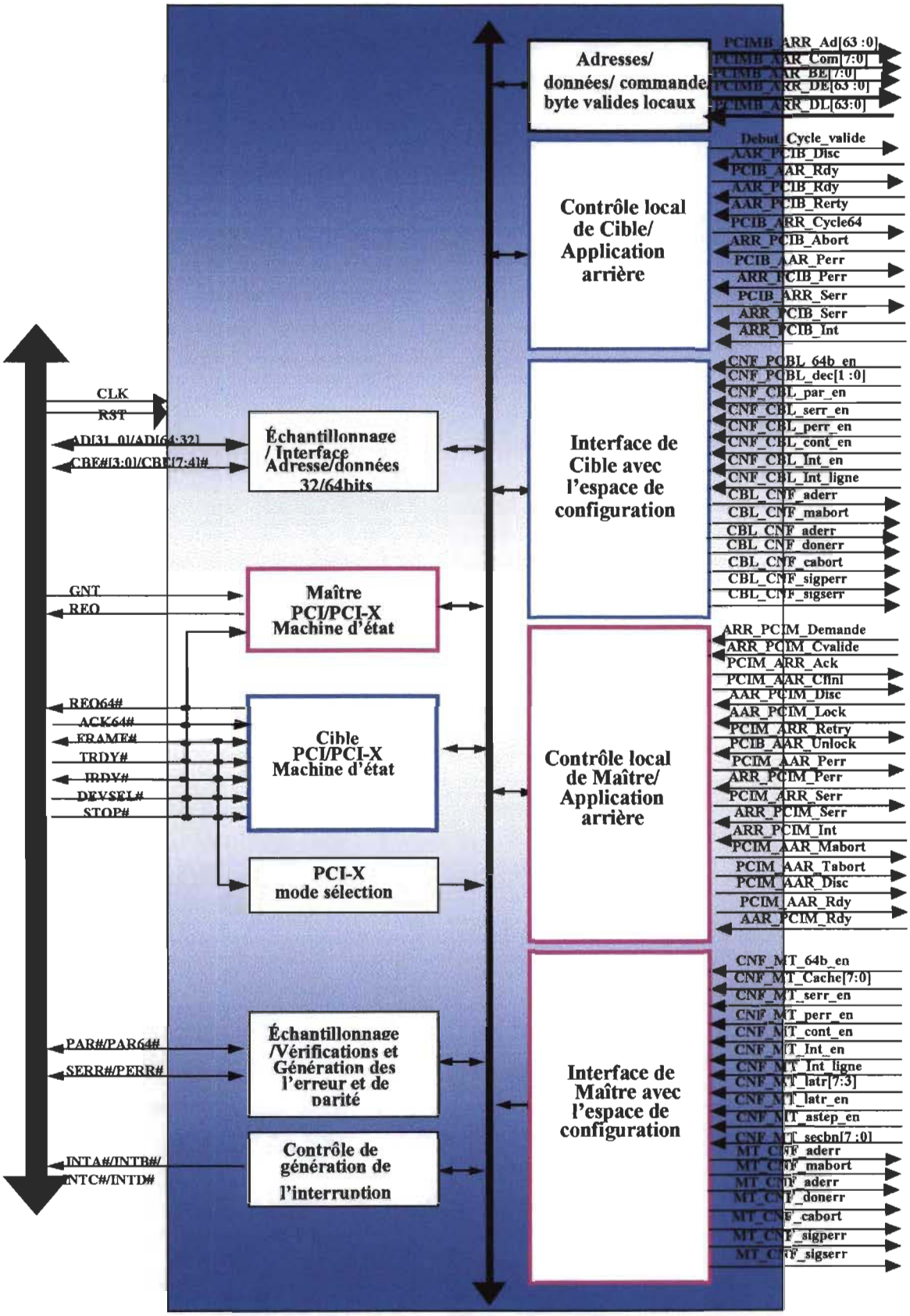

Figure 3.20 : Architecture interne d'un gestionnaire cible/Maître, ses interfaces avec le bus PCI/PCI-X et les applications arrières

3.4.3.1. Unité sélection mode PCI-X

Cette unité a pour rôle de détecter le mode d'initialisation de la carte en mode PCI ou PCI-X en se basant sur un modèle d'initialisation selon le tableau 2.13 donné dans le chapitre 2. D'après ce tableau on trouve que la logique de PCI-X est activée quand le bus est à l'état de repos; c.à.d **FRAME#** =1 et **IRDY#** =1, et quand **TRDY#** =0 ou **STOP#** =0 ou **DEVSEL#** =0.

Cependant la figure 3.21 illustre une structure interne du circuit qui fait de la capacité PCI-X.

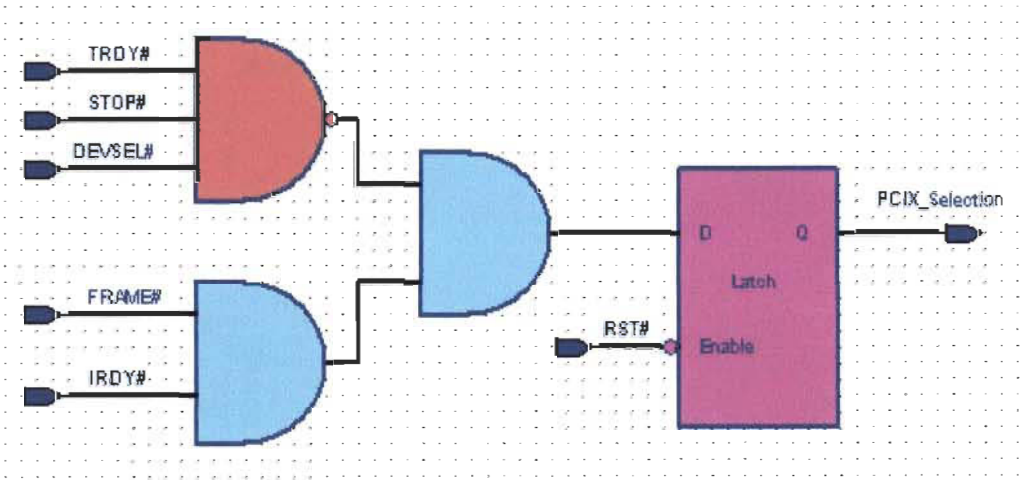

Figure 3.21 : Structure interne de l'unité sélection mode PCI-X

On trouve que la mémoire tampon verrouille la valeur D produit par les signaux de modèle d'initialisation, quand **RST#** =0, c.à.d lors de l'initialisation du gestionnaire par le système hôte.

3.4.3.2 Unité machine d'état

Cette unité est divisée en deux machines : machine d'état cible, qui gère le gestionnaire en mode fonctionnement cible, et machine d'état maître qui le gère en mode maître. De toute évidence, ces deux machines doivent contrôler le fonctionnement de gestionnaire en mode PCI et mode PCI-X.

Constatons que, ces deux machines sont semblables à celles présentées aux gestionnaires PCI type maître et cible; mais la différence fondamentale entre ces deux types du gestionnaire réside dans le fait qu'on a ajouté des états concernant le cas où le gestionnaire entre dans le mode PCI-X.

- ***La machine d'état de gestionnaire cible type PCI/PCI-X***

La figure 3.22 présente la machine d'état cible de ce gestionnaire étudié : les états **C_Attribue** et **C_Attentes** sont les états ajoutés pour le mode PCI-X

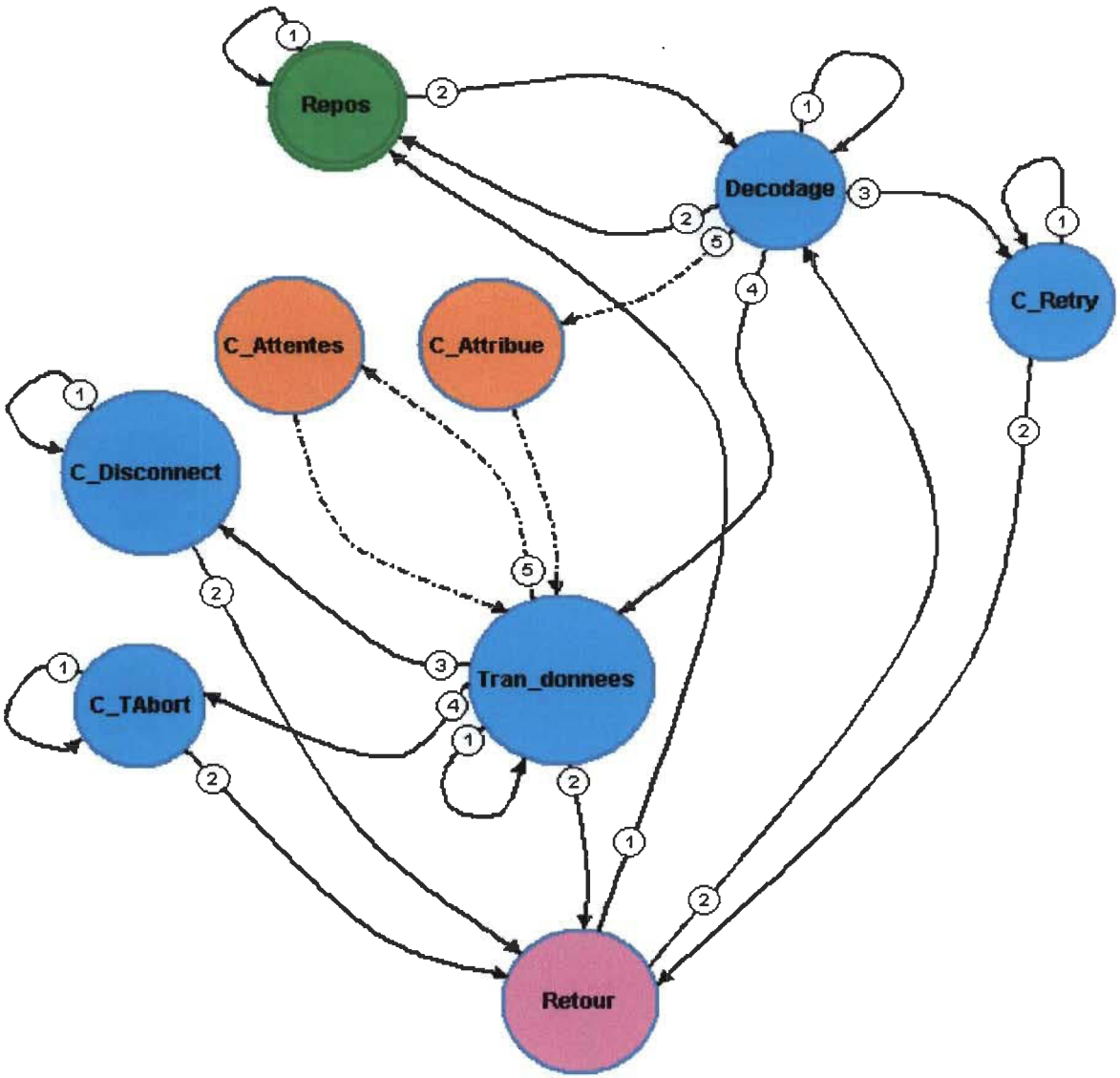

Figure 3.22 : Machine d'état du gestionnaire cible type PCI/PCI-X

- L'état **C_Attribue** est ajouté entre le passage de l'état **Decodage** à l'état **Tran_donnees**. Cet état est ajouté en mode PCI-X à cause de l'ajout de la phase d'attribue de la transaction. Pendant cet état, le maître fournit l'attribue de la transaction tandis que la cible doit le mémoriser. En plus, il est utilisé pour faire l'échange du bus **AD** entre le maître et la cible liés à la transaction dans le cas où ce maître exécute une commande de lecture à cette cible. Ainsi, de l'état **C_Attribue** vers l'état **Tran_donnees**, on a passage inconditionnel car la phase de l'attribue s'effectue pendant une seule période d'horloge. Notons que durant le fonctionnement en mode PCI (quand le signal **PCIX_Selection** est désactivé), le passage de l'état **Decodage** à l'état **Trans_donnees** s'effectue directement.

- Pour bien éclaircir l'intérêt de l'ajout de la phase **C_Attentes**, nous allons examiner l'exécution d'une transaction d'écriture sur le bus PCI-X où la cible de la transaction ajoute des états d'attentes.

En se basant sur la nouvelle notion d'échantillonnage des signaux d'entrées et de sorties du gestionnaire type PCI-X, si le maître n'ajoute pas des états d'attentes durant le transfert de la transaction, il doit fournir la donnée DATA_0 avec **IRDY#** activé à la phase suivante de l'attribue (la cinquième horloge, figure 3.23).

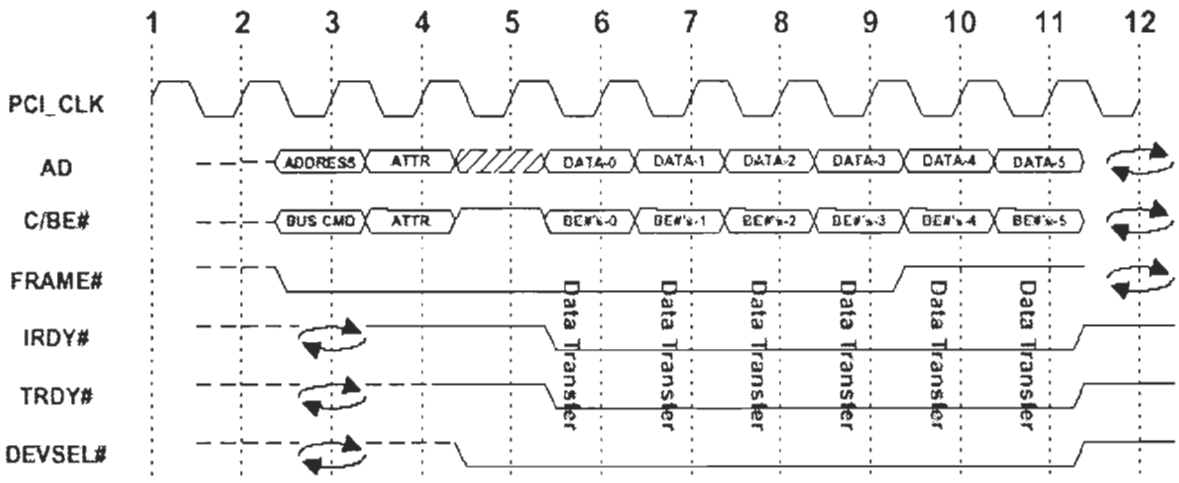

Figure 3.23 : Transaction d'écriture sur le bus PCI-X

Également, si la cible n'ajoute pas des états d'attentes au début du transfert, elle doit activer TRDY# à la cinquième horloge, pour signaler qu'elle est capable de recevoir les données. À la sixième horloge, le maître, à la suite, échantillonne la réponse (**TRDY#** = 0) qui indique que la cible a accepté DATA_0 et il fournit la phase DATA_1 simultanément [18].

Étudions l'exemple le cas d'une cible qui, à la cinquième horloge, n'a pas activé **TRDY#** donc à la sixième horloge, le maître sait que la cible n'a pas reçu DATA_0 et cette connaissance se produit après sa transmission de la deuxième phase de données DATA_1. En effet, durant la période entre la sixième et la septième horloge, le maître va décider de renvoyer de nouveau DATA_0 à la septième horloge. Or, si la cible active TRDY# à la septième horloge, le maître, à la huitième horloge, va connaître que la cible a accepté DATA_0 et par conséquent, il va fournir de nouveau DATA_1 simultanément (figure 3.24).

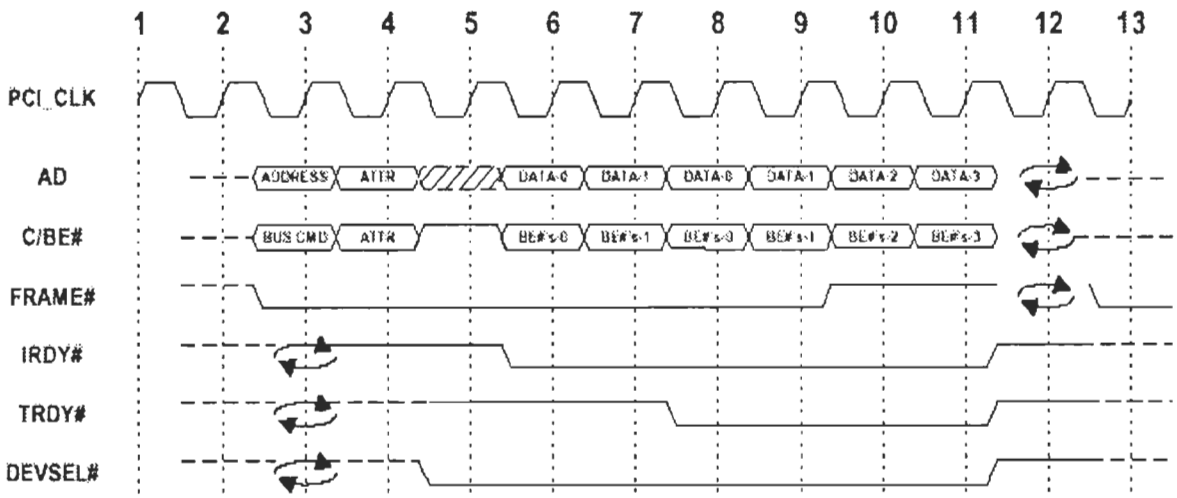

Figure 3.24 : Insertion de deux périodes d'attentes.

Donc si **TRDY#** demeure désactivé pour deux périodes de l'horloge (cinquième et sixième horloge) et la cible l'a activé à la septième horloge, la transaction s'effectue de façon séquentielle (DATA_0 avant DATA_1), mais si la cible l'a activé à la sixième horloge en même temps que le maître fournit DATA_1 et échantillonne que **TRDY#**

était désactivé à la cinquième horloge pour la phase de donnée DATA_0, donc le maître doit envoyer de nouveau DATA_0 après que DATA_1 a été reçue par la cible. Donc la transaction ne s'effectue pas de façon séquentielle (DATA_0 après DATA_1).

Par conséquent, Pour éviter de tomber à des transactions non-séquentielles, le gestionnaire cible type PCI- X doit insérer des états d'attente en nombre paire. Par exemple si à la septième horloge, la cible n'a pas activé **TRDY#**, elle doit attendre à la neuvième horloge pour laisser au maître le temps de permuter entre DATA_0 et DATA_1(figure3.25).

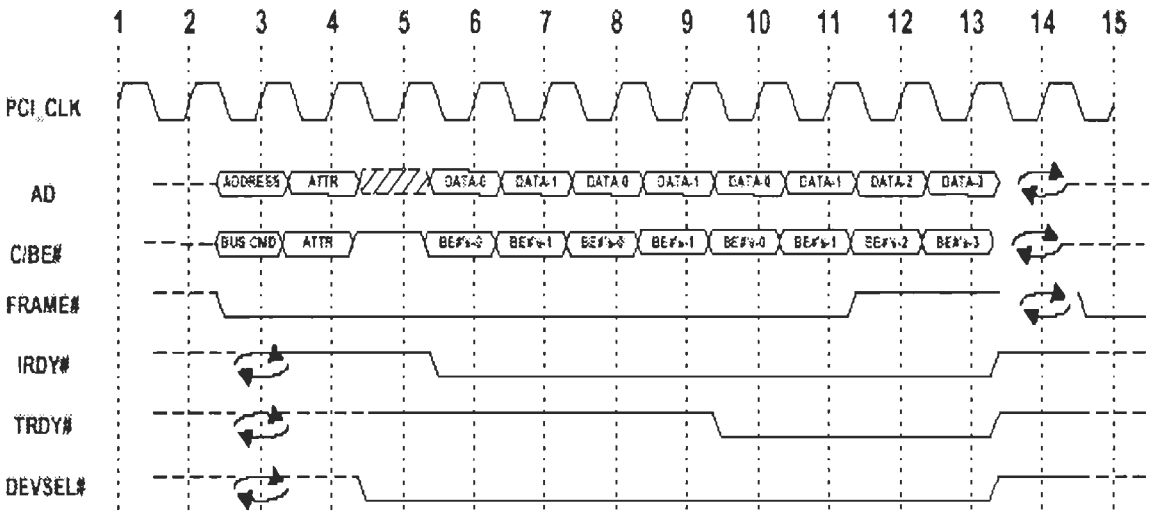

Figure 3.25 : Insertion de quatre périodes d'attentes.

Signalons que la cible passe de l'état **Tran_donnees** à **C_Attentes** quand ces conditions sont avérées simultanément :

- le signal **PCIX_Selection** est activé ;
- on a une transaction d'écriture exécutée sur notre gestionnaire cible;
- l'application arrière veut ajouter des états d'attentes.

Dans ces conditions, la cible désactive **TRDY#** et passe à l'état **C_Attentes**, elle revient à la suite vers l'état **Tran_donnees**, inconditionnellement au prochain front montant de l'horloge, tout en maintenant **TRDY#** désactivé.
À l'état **Tran_donnees**, si la cible est prête à effectuer le transfert, elle reste à cet état en activant **TRDY#**, dans le cas échéant, elle reprend le passage à l'état **C_Attentes** afin désactiver **TRDY#** pendant deux périodes. Donc l'ajout de l'état **Tran_donnees** sert la cible à insérer des états d'attentes paires lors des transactions d'écriture.

Remarque : Dans le cas de l'exécution d'une transaction de lecture sur le bus PCI-X, on ne tombe pas au problème de l'insertion des états d'attentes paires, car c'est la cible qui fournit les données donc elle est capable d'ajouter des états d'attentes quand elle n'est pas prête à faire de la transaction.

En plus de l'ajout de ces deux états, **C_Attentes** et **C_Attribue**, des autres modifications sur les conditions du passage entre certaines états par rapport à la machine d'état cible type PCI ont été manifestés :

- Le passage entre l'état **Tran_donnees** et **Retour** : Rappelons qu'en mode PCI, ce passage se fait quand la cible détecte le signal **FRAME#** désactivé, ce qui annonce la fin de la transaction.

Notons que pour le PCI-X, le nombre de bytes est fournit à la cible pendant la phase de l'attribue donc pour que la cible sache la fin de la transaction il faut intégrer dans la cible un compteur qui décrémente le nombre des bytes transférés et lorsqu'il atteint la valeur 0 on aura le passage à l'état **Retour**.

- Le passage de l'état **Tran_donnees** à l'état **C_Disconnect** : Rappelons qu'en mode PCI le passage se fait quand la cible signale *DISCONNECT*.

Pour le PCI-X, il faut ajouter le cas où l'application arrière veut diviser le transfert en signalant *SPLIT Transaction*.

De même, si la cible signale *DISCONNECT*, cette déconnexion est acceptable seulement au limite permise de la déconnexion (Allowable Disconnect Boundry) , c.à.d **AD[7 :0]**=00h. Par conséquent, pour détecter le passage acceptable **Tran_donnees** à l'état **C_Disconnect** il faut intégrer dans le gestionnaire cible un compteur qui a pour valeur initiale celle de l'adresse **AD[7 :0]** de la transaction et qui décrémente à chaque transfert des données. Lorsqu'il atteint la valeur 00h, le gestionnaire cible peut signaler *DISCONNECT* et après il fait le passage à l'état **C_Disconnect**.

Notons que ce compteur se réinitialise à la valeur FFh dés qu'il atteint la valeur 0 afin de continuer à compter les données transférées lorsqu'il n'y a pas une demande de déconnexion.

➢ ***La machine d'état de gestionnaire maître type PCI/PCI-X***

La figure 3.26 présente la machine d'état maître de ce gestionnaire étudié : les états **M_Attribue** et **M_Attente** sont les états ajoutés pour le mode PCI-X.

- Pour la même raison sur l'ajout de l'état **C_Attribue** dans la machine d'état d'une cible on a ajouté l'état **M_Attribue** pour fournir la phase d'attribue sur le bus.

- De l'état **M_Attribue** vers l'état **Trans_donnees**, on a un passage inconditionnel car la phase de l'attribue s'effectue pendant une seule période d'horloge.

- Également l'état **M_attentes** est ajouté de la même raison que **C_Attentes** pour la machine d'état de la cible. Cet état est utilisé par le maître pour faire la permutation entre les phases des données DATA_0 et DATA_1 quand la cible insère des états d'attentes.

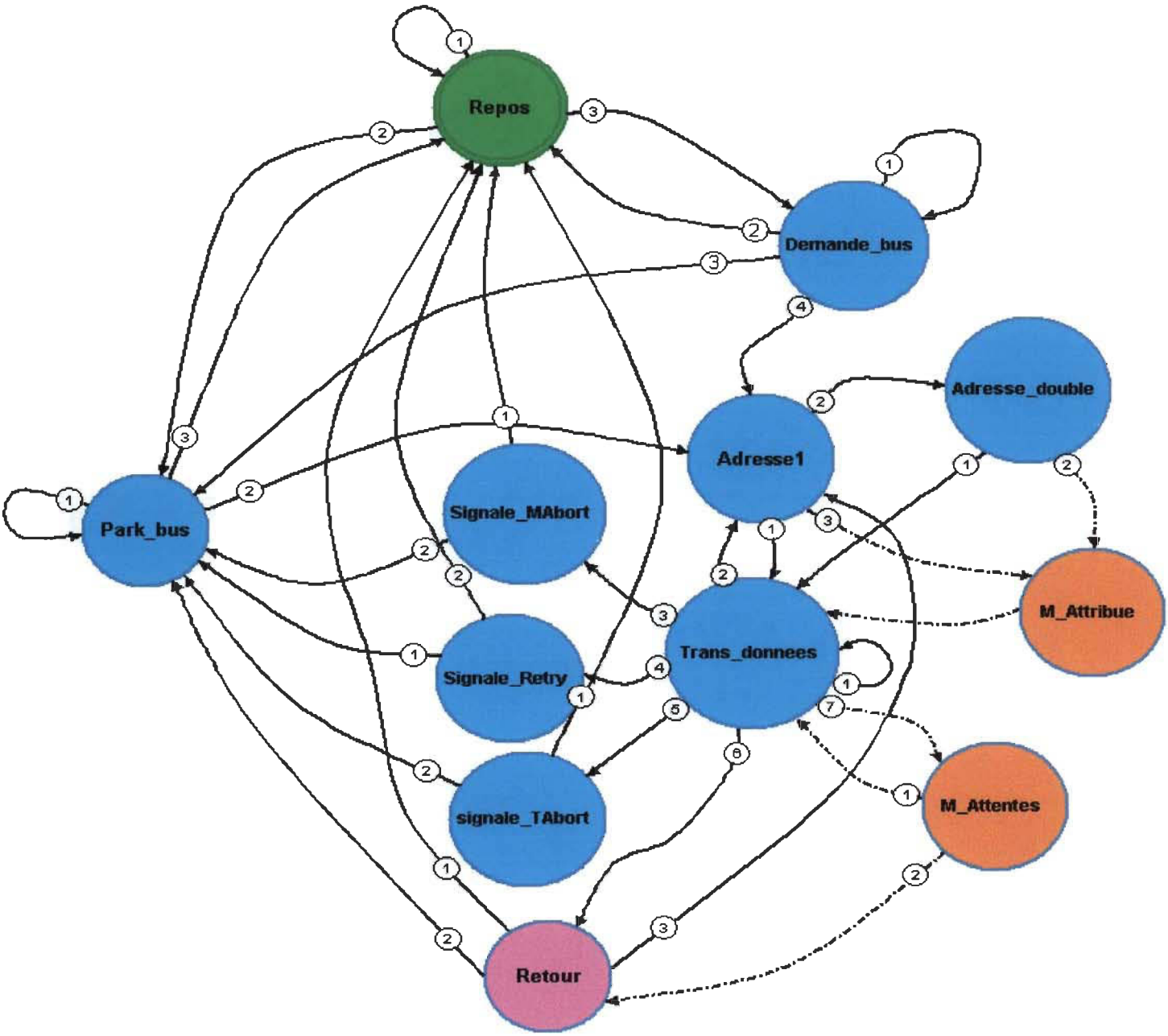

Figure 3.26 : Machine d'état de gestionnaire maître type PCI/PCI-X

Les autres unités possèdent en outre, des fonctionnalités identiques que celles du PCI mais elles présentent des logiques supplémentaires pour faire le passage entre les modes PCI et PCI-X.

3.5. Validation de gestionnaire développé

Dans les sections précédentes nous avons décrit la conception d'un gestionnaire générique type PCI/PCI-X comme combinaison des circuits logiques. Après un procédé de conception, nous comptons que le gestionnaire conçu va fonctionner comme il faut. Mais comment on vérifie que le circuit final atteint en effet les objectifs de la conception? Nous verrons par la suite, les différents niveaux adoptés afin de tester les gestionnaires développés et afin d'évaluer leurs performances.

3.5.1. Au niveau de la simulation

Il est essentiel de s'assurer que le gestionnaire conforme avec le comportement fonctionnel exigé et qu'il rencontre toutes les contraintes de synchronisation et de fonctionnalité qui sont imposées au design. Nous avons discuté sur les sujets de synchronisation des signaux du gestionnaire ainsi que son développement. Dans cette section, nous montrerons les méthodes d'essai adoptées pour vérifier la fonctionnalité de notre gestionnaire développé.

La phase de la compilation exécute au design des contrôles qui sont entièrement statiques en nature et qui exercent des vérifications simples: elle ne demande ni des stimulus, ni une description de la sortie prévue du circuit testé. Les outils de compilations ne peuvent pas identifier tous les problèmes en code source, ils peuvent seulement trouver les problèmes qui peuvent être statiquement déduits en examinant la structure du code, mais pas des problèmes dans l'algorithme ou dans les flux de données comme l'absence d'une branche d'une condition 'if' dans l'algorithme. Par ailleurs, ces outils sont semblables au contrôle orthographique; ils identifient des mots mal épelés, mais ne détectent pas les fautes de syntaxe. Par exemple, un programme peut contenir plusieurs exemples d'utilisation des mots 'with' et 'width' mais l'outil de la compilation ne peut pas détecter l'erreur de la confusion de leurs utilisations puisque ces deux mots sont réservés dans le code.

Afin de vérifier le bon fonctionnement des gestionnaires développés, on a besoin de créer une liste de contrôle (checklist) qui décrit les différentes transactions possibles exécutées avec ce gestionnaire. Cependant, après la génération de toutes ces transactions possibles de lecture, écriture etc.. et tous les arrêts possibles lancés par un maître ou une cible, on doit côcher dans cette liste de contrôle la transaction réussite et déboguer la transaction échouée au niveau du développement de gestionnaire. Il est essentiel de signaler que pour effectuer les transactions lancées par le gestionnaire vers les applications arrières et vis versa, on doit simuler l'environnement où on va utiliser ce gestionnaire après l'intégration. D'où la nécessité de développer un testbench pour tester ce gestionnaire. Le terme testbench, dans VHDL et Verilog, se rapporte habituellement au code employé pour appliquer à un design des ensembles de vecteurs d'entrée prédéterminés, et observer sa réponse. Il est généralement mis en application en utilisant VHDL ou Verilog, mais il peut inclure également des fichiers de données externes ou des sous-programmes en langage C. Par ailleurs, étant donnée la proportion significative de la littérature consacrée à l'écriture des codes VHDL ou Verilog synthétisables d'une part, et comparée avec l'écriture des testbenches pour vérifier l'exactitude fonctionnelle de ces codes d'autre part, on peut être tenté de conclure que la première tâche est plus intimidante que la deuxième. Mais la réalité, trouvée dans toutes les équipes de conception du matériel, pointe vers le contraire. En effet, aujourd'hui c'est l'ère des circuits à plusieurs millions des portes ASICs, des propriétés intellectuelles réutilisables (Intellectual Property, IP) et l'intégration d'un système dans la même puce (System-on-a-Chip SoC), donc la vérification des designs consomme 70% de l'effort de son développement. En plus, parmi les équipes de la conception du matériel, on rencontre celles consacrées à la création du design, et d'autres consacrées à la vérification du design. Le nombre du personnel destiné à la vérification est habituellement deux fois le nombre du personnel destiné à la création de design. Cependant, quand la création d'un design est achevée, le code dédié à la création de testbenches consomme jusqu'à 80% du volume totale de code de projet. La figure 3.27 illustre comment un tel testbench interacte avec un design sous la vérification (Design Under Verification DUV). Le testbench fournit des entrées au design et contrôle toutes ses sorties. Remarquons que ce système est complètement fermé: aucune entrée ou sortie n'interchange avec l'extérieur [27].

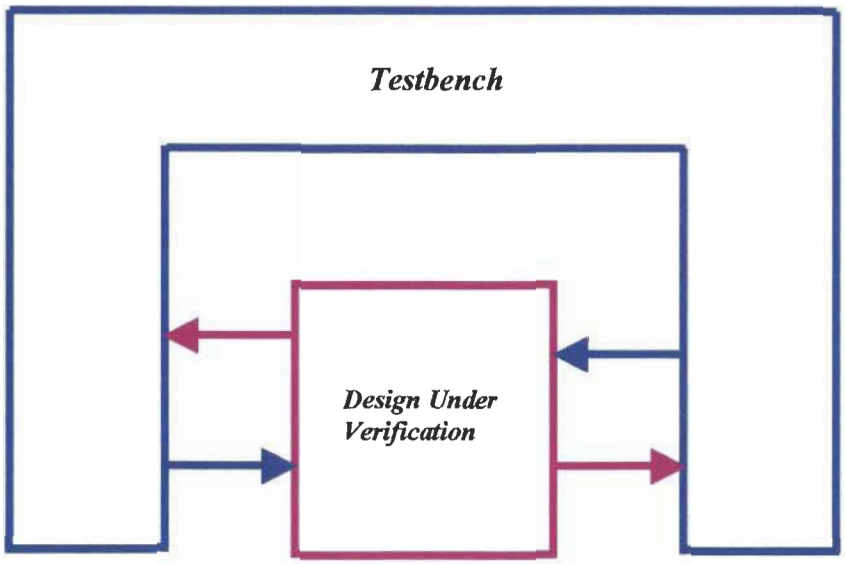

Figure 3.27 : Interaction entre le testbench et le design sous la vérification(Design Under Test)

À la suite, nous décrivons les différents testbenches qu'on a développé pour tester les gestionnaires type PCI et PCI-X.

3.5.1.1. Testbenches pour tester le gestionnaire type PCI

Puisque le gestionnaire type PCI peut avoir la fonctionnalité maître ou cible donc on a dû créer deux testbenches qui ont pour rôle de tester l'une de ces deux fonctionnalités. Le testbench de gestionnaire développé a été décomposé en deux parties entourant le gestionnaire, une partie représente l'interface du gestionnaire avec le bus PCI/PCI-X et l'autre représente l'interface avec les applications arrières.

- **Testbench pour tester la fonctionnalité type cible**

La figure 3.28 illustre le testbench créé pour tester la fonctionnalité type cible de ce gestionnaire. Notons que, son interface avec le bus PCI joue deux rôles : le rôle d'être un système de configuration qui initialise les transferts destinés pour configurer l'espace de configuration de l'application arrière via ce gestionnaire, et le rôle d'être un maître, connecté au bus PCI, qui lance des transactions de lecture et d'écriture aux applications implantées derrières ce gestionnaire. Ces deux rôles sont assurés par l'unité : Générateur

des vecteurs de test de côte de bus PCI, qui est formée d'une composante qui génère les différentes transactions au bus PCI sous forme des vecteurs de test. Ainsi, cette unité est liée à une autre unité nommée Générateur de l'horloge qui a pour rôle de générer l'horloge du système (figure 3.28).

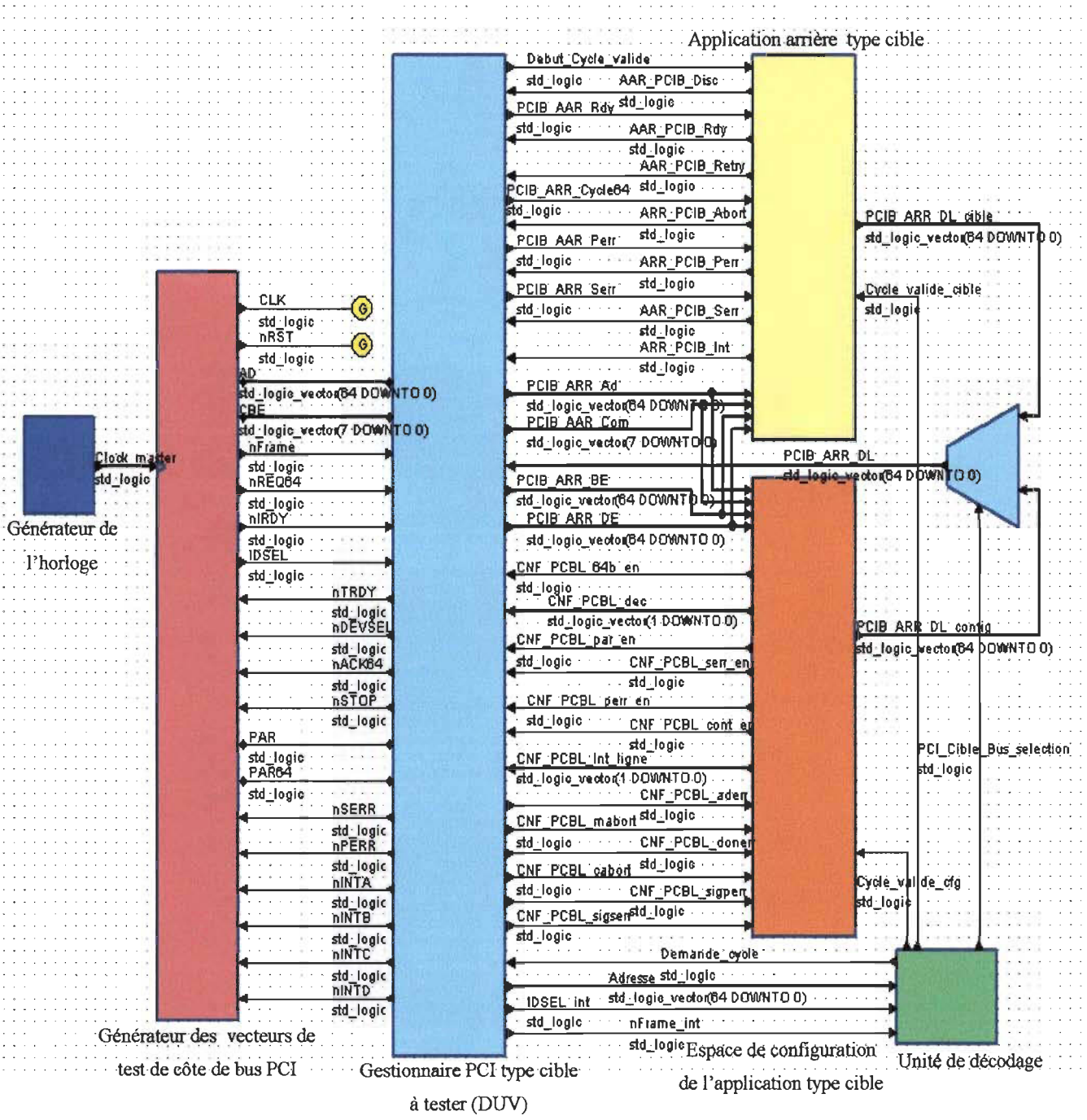

Figure 3.28 : Testbench créé pour tester le gestionnaire PCI type cible

Par ailleurs, l'interface de testbench avec les applications arrières est divisée en trois parties : la première représente l'application arrière type cible pour laquelle on a choisit une mémoire vivante avec sa logique de contrôle, la seconde partie représente l'espace de configuration de cette application arrière et la dernière partie est responsable au décodage de l'adresse lancée par le bus PCI. Notons que l'application arrière type mémoire vivante a été conçue avec la possibilité d'accès au niveau 32 ou 64 bits. Elle possède des logiques supplémentaires qui sont programmées d'une façon à générer les arrêts des transactions type *RETRY* et *DISCONNECT* à des moments précis des transactions et de signaler l'arrêt *Target_Abort* lorsqu'une transaction en mode continu excède sa capacité.

L'unité Générateur des vecteurs de test joue le rôle d'une composante qui lit les vecteurs de test à partir d'un fichier texte et les applique aux entrées de notre gestionnaire. Ces vecteurs de test initialisent au début l'espace de configuration afin de programmer le gestionnaire d'être une cible de 32 bits. Ils implantent à la suite l'application mémoire vivante en réservant un espace d'adressage de 256 bytes dans l'espace d'E/S du système hôte. Après cette initialisation, des transactions comme I/O Read et I/O Write ont été généré par ces vecteurs de test pour tester le bon fonctionnement du gestionnaire. Et pour tester la signalisation *Target_Abort* de notre gestionnaire, on a généré des accès en mode continu à la mémoire qui dépasse sa capacité de 256 bytes. Également, pour tester la signalisation *RETRY* de notre gestionnaire, la mémoire vivante a été programmée pour indiquer au gestionnaire qu'elle est occupée lors d'un accès d'un maître de bus au début d'une transaction.

En outre, lors de la génération des transactions notamment *IO Read* et *IO Write* avec des adresses différentes que l'adresse de la mémoire et des transactions *Memory Read* et *Memory Write*, nous avons constaté que le gestionnaire ne répond plus à ces transactions, ce qui était attendu.

Or, des nouveaux vecteurs de test ont été générés pour assurer l'implantation de ce gestionnaire de 32 /64 bits dans l'espace mémoire du système hôte. Par conséquent, pour tester le fonctionnement de ce gestionnaire dans ce cas, on a repris les mêmes procédures précitées auparavant toutefois en activant les signaux liés aux transactions de 64 bits.

- **Testbench pour tester la fonctionnalité du gestionnaire type maître**

La figure 3.29 illustre le testbench créé pour tester la fonctionnalité type maître.

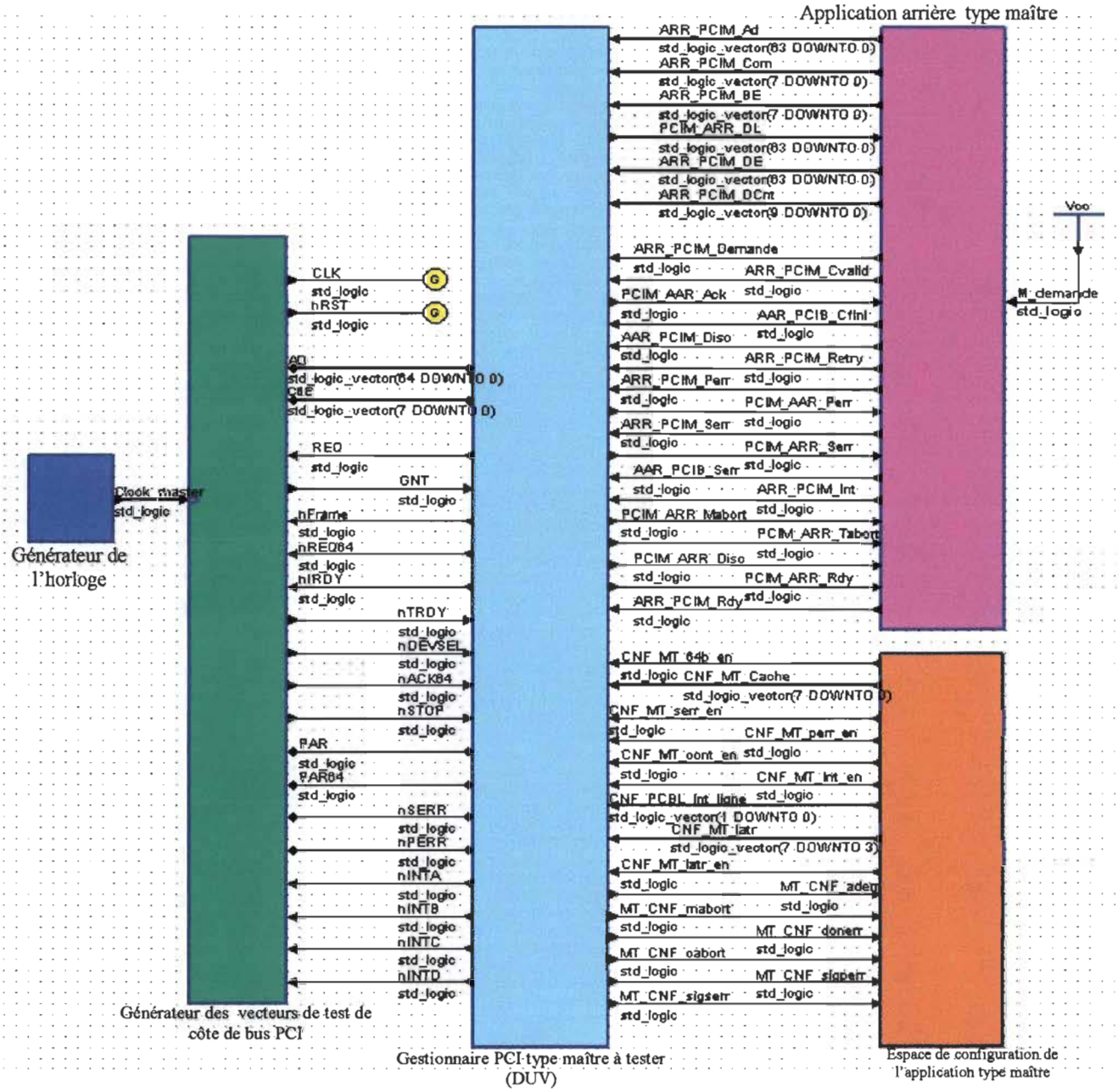

Figure 3.29 Testbench créé pour tester le gestionnaire PCI type maître

Son architecture est semblable à celle de la cible, mais diffère dans son fonctionnement, par exemple :

- l'unité Générateur des vecteurs de test fournit les signaux de contrôle d'une cible au lieu de fournir les signaux de contrôle d'un maître et en plus elle joue le rôle de l'arbitre qui donne l'accès au bus PCI du gestionnaire maître.
- . dans le cas de l'application type maître implantée derrière le gestionnaire, nous avons employé deux FIFOs au lieu d'une mémoire vivante, l'une pour l'écriture des données dans le bus PCI et l'autre pour la lecture. Ces deux FIFOs possèdent des capacités de 64 bytes et peuvent être accédées aux niveaux de 32 ou 64 bits. En plus nous avons implanté des logiques avec ces deux FIFOs qui ont pour rôle d'initialiser et contrôler les transferts sur le bus PCI via le gestionnaire.

Par ailleurs, des vecteurs de test ont été crée pour initialiser au début l'espace de configuration afin de programmer le gestionnaire d'être un maître et de programmer les registres *Latency Timer* et *Cache Line Size* pour limiter l'accès du gestionnaire au bus PCI. Après cette phase d'initialisation, nous avons testé les transactions des lectures et écritures, lancées par ce gestionnaire, par la création des séquences dans lesquelles la cible liée à ces transactions répond directement ou insert des états d'attente. Et pour tester en suite la réponse *Master_Abort* de notre gestionnaire, d'autres vecteurs de test ont été créés pour simuler une cible accédée au bus PCI qui n'active pas le signal **DEVSEL#** après 6 périodes d'horloge du début de transfert. Et pour tester la limitation de l'accès au bus par le gestionnaire au moment de l'expiration de la durée programmé au registre *Latency Timer* (16 périodes dans notre cas), nous avons généré, par les FIFOs via le gestionnaire, des transactions qui ont une durée supérieure à 16 périodes d'horloge. Après l'expiration de cette durée, nous avons constaté que le gestionnaire cède le bus en désactivant tous ses signaux de contrôle. Et pour terminer, nous avons rétabli ce test de limitation en adoptant la commande *MWI*, dans laquelle le maître doit maintenir le transfert tant que la valeur programmée au registre *Cache Line Size* (24 bytes dans notre cas d'essai) ne vaut pas zéro, malgré l'expiration de la durée programmée au registre *Latency Timer*, nous avons constaté que le gestionnaire a maintenu le transfert jusqu'a ce que le *Cache Line Size* décompte à la valeur 0. Et finalement, nous avons testé la fonctionnalité normale du gestionnaire maître concernant la génération de différentes transactions de lecture et d'écriture sur les bus PCI.

3.5.1.2 Testbench pour tester le gestionnaire type PCI/PCIX

La figure 3.30 illustre le testbench créé pour tester le gestionnaire maître/cible type PCI /PCI-X.

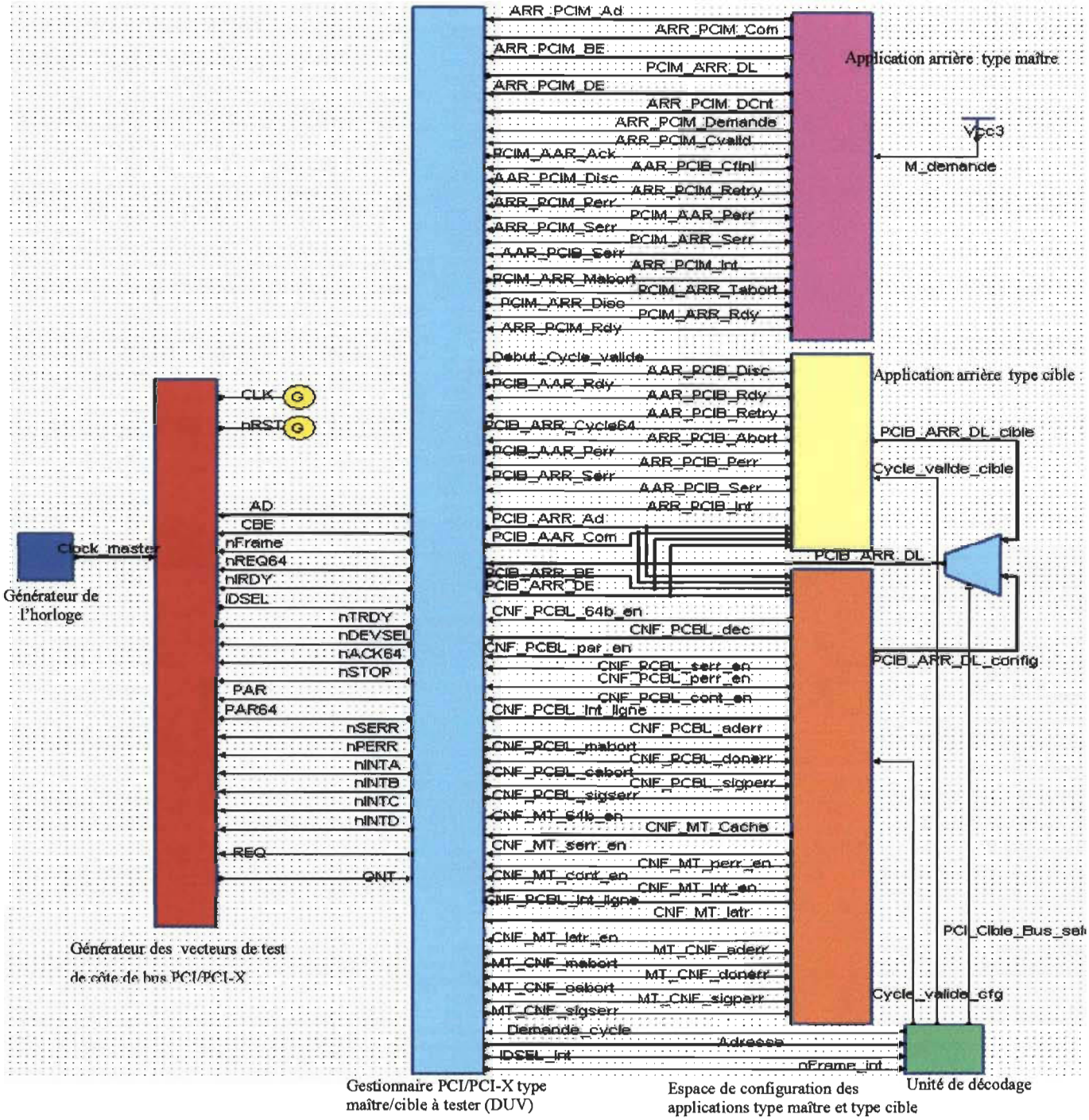

Figure 3.30 : Testbench créé pour tester le gestionnaire type PCI/PCI-X

Pour cet environnement de test, nous avons rassemblé les deux testbenches précédents (maître et cible) dans un seul testbench. Cependant, nous avons ajouté des vecteurs de test pour vérifier la fonctionnalité en mode PCI-X du gestionnaire, en générant les commandes *Split Transactions* et les nouvelles conditions de déconnexion sur le bus. Et pour terminer la phase de la simulation, nous avons testé la capacité du gestionnaire de permuter entre les modes PCI et PCI-X selon les modèles d'initialisation lancés sur le bus.

3.5.2. Au niveau de la synthèse et l'optimisation

Après avoir assemblées et simulées toutes les sous-unités, il est nécessaire d'effectuer la synthèse de la description VHDL du gestionnaire en un ensemble des portes logiques. À la suite de cette synthèse, le circuit obtenu est optimisé avec des contraintes de temps et de surface. Signalons que ces opérations ont été effectuées avec le logiciel Synplify Pro de Synplicity et de la librairie des circuits FPGAs de la famille Xilinx. Le résultat obtenu, c'est-à-dire du circuit final, c'est la génération des fichiers permettant la configuration des circuits programmables pour la réalisation pratique.

3.5.2.1. Réalisation de gestionnaire

Normalement, la réalisation de gestionnaire peut se faire avec plusieurs types de technologies de circuits tels que : les composantes normalisées ou programmables, les prédiffusés, les cellules normalisées et les circuits dédiés. Généralement, les concepteurs de circuits intégrés cherchent à obtenir une performance optimale et à réduire le temps de conception, le temps de fabrication ainsi que le coût de production. Par ailleurs, Il faut signaler que les composantes programmables représentent une solution avantageuse qui respecte, pour un faible taux de production, les différents critères précités.

Le gestionnaire a été conçu pour être mis en œuvre avec des circuits FPGAs (Field Programmable Gate Arrays) reprogrammables de la famille Xilinx. Ces circuits réutilisables, performantes, reconfigurables et économiques permettent d'obtenir très rapidement un prototype fonctionnel.

En outre, les circuits FPGAs de Xilinx sont organisés en matrice de cellules, chacune d'elles, peut être reliée par des connexions locales et globales programmables. Ces circuits contiennent essentiellement trois types de ressources : les cellules logiques CLB (Configurable Logic Block), les blocs d'entrées et de sorties IOB (Input Output Block) et les interconnexions. Les CLBs sont des cellules programmables pour réaliser les fonctions combinatoires et séquentielles de notre architecture à implanter. Chaque CLB contient un nombre des LUTs (Look Up Table) à 4 entrées (8 pour la famille Virtex2 de Xilinx) et un nombre de bascules (de même 8 pour la famille Virtex2 de Xilinx). Les IOBs, qui sont comme tampons aux entrées/sorties de l'FPGA, peuvent être programmés comme des entrées ou sorties avec ou sans bascules, et elles peuvent être également programmées comme des entrées / sorties à trois états avec ou sans résistance de tirage (Pullup Resistor). Par ailleurs, les unités des interconnexions sont formées par des transistors appelés PIPs (Programmable Interconnect Points) qui assurent les interconnexions entre les différentes cellules du circuit programmable. Notons que toutes ces cellules sont programmées lors de chargement des fichiers de configuration générés par la phase de la synthèse du système à implanter [28].

Or, la dernière version de Xilinx est la famille Virtex, son architecture interne est illustrée dans la figure 3.31.

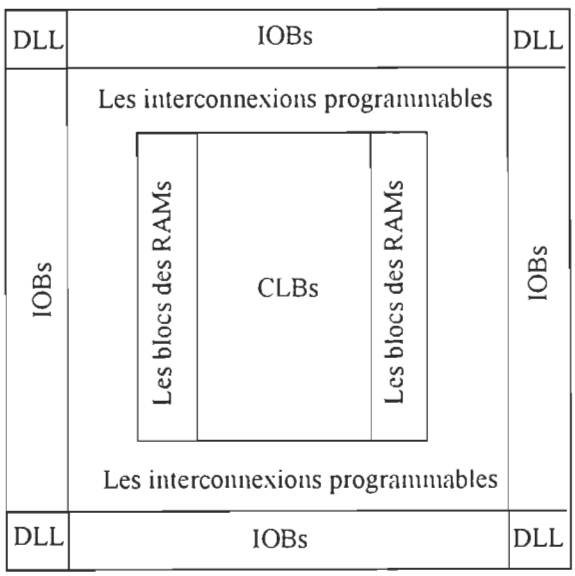

Figure 3.31 : Architecture interne d'un circuit de la famille Virtex

Les Virtex sont établies avec une lithographie de 0,22 microns (avec une carte routière de 0,18 microns) et une technologie à cinq couches de métal. Les circuits de cette famille possèdent de l'ordre de 75.000.000 transistors. Ces nouveaux circuits ont des caractéristiques intéressantes : ils incluent des entrées différentielles à basse tension pour supporter des bus en mode GTL (Gunning Transceiver Logic), des blocs de SRAM à porte-double de 4096 bits, des cellules de RAM distribuées, des boucles de verrouillage à retard commandé pour fournir des horloges à retard programmable, et de routage de mode vectoriel permettant d'avoir un routage flexible vers le haut / vers le bas / à gauche / droit entre les CLBs.

Ces circuits reprogrammables fonctionnent à 2,5 volt, mais leurs E/Ss tolèrent l'interface avec des blocks à plus haute tension. Les IOBs de Virtex incluent : des résistances de tirage programmables vers le haut ou vers le bas, des circuits type Faible Garde (Weak Keeper) qui maintiennent les valeurs des entrées ou sorties quand leurs commandes sont retirées, et des entrées avec des circuits de retard programmables afin de décaler un signal d'entrée pour effectuer la synchronisation avec l'horloge du circuit [29].

Signalons que les résultats de synthèse, après le placement et le routage par les outils Synplify Pro de Synlicity en utilisant le circuit XC2V250FG256-6 de la famille Virtex de Xilinx, sont résumés dans le tableau 3.1.
Néanmoins, le gestionnaire développé type PCI peut fonctionner comme maître ou cible et le gestionnaire PCI-X doit fonctionner comme maître et cible et de plus il doit fonctionner en mode PCI, donc on a présenté les résultats d'intégration, en fonction des nombres des ressources consommées et fréquence obtenu pour un gestionnaire PCI Cible, PCI Maître, PCI Cible/Maître, et PCI/PCI-X Cible/Maître. Il faut signaler pour terminer qu'afin de montrer l'amélioration apportée au gestionnaire type PCI-X par rapport à un gestionnaire type PCI on a présenté le résultat de l'intégration d'un gestionnaire type PCI-X sans introduire les logiques liées au mode PCI.

Tableau 3.1 : Résultats de synthèse sur FPGA

	Fréquence	Période (nsec)	LUTs (1548)	Bascules (3100)	I/Os (1557)
PCI Cible	67.1 MHz	14.9	277 (18%)	306 (9.96%)	333 (21%)
PCI Maître	72.2 MHz	13.8	468 (30%)	403 (13%)	364 (23%)
PCI Cible/Maître	59 MHz	16,9	724 (46%)	686 (22%)	386(24%)
PCI/PCI-X Cible/Maître	51.5 MHz	19.4	1425 (92%)	1018 (33%)	394 (25%)
PCI-X Cible/Maître	78 MHz	12.8	624 (40%)	769 (24%)	372(23%)

D'après les résultats obtenus ci-dessus, on constate que pour le gestionnaire type PCI/PCI-X, on a obtenu une fréquence inférieure à celle du gestionnaire type PCI. Ceci est dû à la quantité énorme des composantes logiques constituant le gestionnaire type PCI/PCI-X puisqu'il doit inclure à la fois la fonctionnalité maître et cible des types PCI et PCI-X. Également, on peut remarquer que lorsqu'on a implanté le gestionnaire en mode PCI-X Cible/Maître sans introduire les logiques liées au mode PCI, on a obtenu un meilleur résultat au niveau de la vitesse par rapport au gestionnaire type PCI Cible/Maître vu le mode de transfert amélioré et le pipelinage utilisés dans les gestionnaires type PCI-X.

3.6. Conclusion

Au niveau de ce chapitre, nous avons pu proposer une architecture d'un gestionnaire générique type PCI/PCI-X adaptable avec les différents types d'applications. Cette architecture développée assure, au gestionnaire d'avoir la fonctionnalité Cible/Maître type PCI et la fonctionnalité en mode PCI-X.. Ainsi, cette architecture a été réalisée en langage VHDL comportemental, indépendant de la structure d'une certaine librairie d'intégration. À la suite, afin d'évaluer et de valider cette architecture, on a créé des environnements de test utilisés pour tester la fonctionnalité de chaque type de gestionnaire. Également, les performances de ces gestionnaires ont été mises en œuvre

par la synthèse de ces gestionnaires sur une structure logique reprogrammable de la compagnie Xilnix. D'après les résultats obtenus par la synthèse, la fréquence obtenue pour le gestionnaire type PCI/PCI-X était inférieure à celle du gestionnaire type PCI, ce qui était attendu grâce à la quantité énorme des composantes logiques constituant le gestionnaire type PCI/PCI-X. Également, le gestionnaire en mode PCI-X Cible/Maître a donné des résultats favorables au niveau de la fréquence d'opération par rapport au gestionnaire type PCI Cible/Maître.

Chapitre 4

Conclusion générale

Les bus d'extension sont indispensables dans les systèmes informatiques modernes, leur principale rôle est d'assurer la flexibilité de l'interfaçage interne et externe pour les applications d'une plate forme et aussi de séparer l'évolution de cette plate forme de celle des applications développées.

Le bus PCI est le bus le plus répondu, qui depuis son apparition a occupé la majorité des plates formes informatisées; son extension le bus PCI-X, apparaissant au début de l'an 2000, a introduit une solution efficace pour les serveurs et les applications demandant des débits élevés.

Le gestionnaire de périphérique est la composante qui lie les applications arrières avec le système hôte via un bus d'extension, il assure d'une part, l'intégration de l'application arrière dans le système de point de vue initialisation et réservation de ressources; et d'autre part, la synchronisation de transfert avec toutes les applications résidentes sur le bus d'extension. Par ailleurs, le gestionnaire type PCI est disponible sous forme de circuits intégrés, qui imposent souvent des contraintes sur le développement des applications derrière ce type de gestionnaire. Également, il est disponible sous forme de

noyaux qui présentent des omissions par rapport à la spécification du bus PCI et qui présentent aussi une dépendance à certaines libraires d'intégration imposées par leurs constructeurs; ce qui conduit à des noyaux qui ne sont pas portables d'un constructeur à un autre, et qui ne profitent pas de toutes les options du PCI/PCI-X.

Il est à noter que les articles rencontrés, dans la littérature, traitant le développement des gestionnaires de périphérique type PCI, portent sur le développement de gestionnaires faisant des tâches fixes et qui ne peuvent pas fonctionner avec plusieurs types d'applications à cause de leurs dépendances à des applications précises; par conséquent, ils ne sont pas génériques à plusieurs niveaux.

Dans ce cadre, la présente étude nous a permis de développer une architecture des gestionnaires génériques afin de surmonter les contraintes précitées.

L'architecture développée assure, en effet, aux gestionnaires d'avoir la fonctionnalité Cible/Maître type PCI et la fonctionnalité en mode PCI-X. Elle assure également la possibilité de leur utilisation avec n'importe quelle application arrière due à son interface flexible. Les gestionnaires développés ne présentent pas, donc, des limitations sur le nombre d'applications résidantes derrière eux; même après leur intégration sur des puces, grâce à la séparation des espaces de configuration de ces gestionnaires. Également, ces gestionnaires peuvent être configurés pour n'importe quel constructeur d'applications arrières. Ils sont construit d'une façon à laisser, dans le cas de nombreuses applications implantées derrière ces gestionnaires, aux utilisateurs le choix de partager le temps d'accès au bus PCI entre les applications type maître et le choix de la rapidité de répondre aux transactions pour les applications type cible grâce à la présence de l'unité Decodage_Arbitrage.

En outre, le choix des signaux à deux états entre les gestionnaires et les applications arrières laisse, d'une part, dans le cas de leur intégration dans une même puce reprogrammable, les pins 3 états pour l'interfaçage entre ces gestionnaires et le bus PCI/PCI-X et d'autre part, dans le cas de leur intégration dans des puces séparées. Ce choix facilite le développement de ces applications arrières.

Il est à noter que, l'architecture développée a été réalisée en langage VHDL comportemental, indépendant de la structure d'une certaine librairie d'intégration, afin d'assurer la portabilité de ces gestionnaires entre les différentes technologies cibles de

l'intégration. Par ailleurs, afin de valider le fonctionnement de ces gestionnaires, des environnements de test, semblables à ceux où les gestionnaires peuvent être intégrés, ont été crées pour assurer que la fonctionnalité de ces gestionnaires s'inscrive bien dans la perspective de cette étude.

Également, les performances de ces gestionnaires ont été mise en œuvre par la synthèse de ces gestionnaires sur une structure logique reprogrammable (Field Programmable Gates Array) XC2V250FG256 de la compagnie Xilnix à l'aide des outils Symplify PRO de Synplicity. Les résultats de la synthèse, nous ont permis de déduire qu'on peut atteindre des fréquences d'opération de 59MHz, 67.1 MHz et 72.2 MHz pour des gestionnaires type PCI avec fonctionnalité Cible/Maître, Cible et Maître respectivement, une fréquence d'opération 51.5 MHz pour un gestionnaire type PCI/PCIX avec fonctionnalité Cible/Maître et une fréquence d'opération 78 MHz pour un gestionnaire type PCIX avec fonctionnalité Cible/Maître.

D'après les résultats obtenus ci-dessus, on constate que pour le gestionnaire type PCI/PCI-X, on a obtenu une fréquence inférieure à celle du gestionnaire type PCI. Ceci est dû à la quantité énorme des composantes logiques constituant le gestionnaire type PCI/PCI-X puisqu'il doit inclure à la fois la fonctionnalité maître et cible des types PCI et PCI-X. Également, on a pu remarquer que lorsqu'on a implanté le gestionnaire en mode PCI-X Cible/Maître sans introduire les logiques liées au mode PCI, on a obtenu un meilleur résultat au niveau de la vitesse par rapport au gestionnaire type PCI Cible/Maître vu le mode de transfert amélioré et le pipelinage utilisés dans les gestionnaires type PCI-X.

En dernier lieu, pour voir l'effet d'utilisation d'un code VHDL structurel par rapport à un code comportemental, on a testé le développement d'un gestionnaire cible type PCI de 32 bits d'une façon structurelle en ayant recours à l'instantiation manuelle des instructions de code VHDL dans des composantes de la librairie XC4000 de Xilinx, qui possède des caractéristiques beaucoup moins efficaces que celles de la famille Virtex de la même compagnie. Comme résultat de synthèse de ces gestionnaires, comportemental et structurel, introduits dans le même type de circuit XC4002APC84-5, on a obtenu une fréquence de 33.7 MHz pour le code comportemental et une fréquence de 61.1 MHz pour le code structurel. On remarque que le code structurel donne une meilleure performance

que le code comportemental. Cependant, ce type de code structurel présente l'inconvénient d'être pénible à développer pour les designs complexes (c'est la raison pour laquelle on a introduit comme un test le gestionnaire cible qui est le plus simple à décrire sous forme structurelle), et ainsi il est non portable.

Il est important de signaler que notre objectif était de développer un gestionnaire portable entre les technologies cible de l'intégration, néanmoins on a effectué ce test pour proposer une suite à ce travail, qui est le développement des noyaux de ces gestionnaires avec la possibilité de spécifier à un haut niveau les instructions qui sont affectées le plus par la technologie d'intégration (FPGA, CPLD ou ASIC) et en offrant la possibilité d'un remplacement automatique des instructions ayant une influence sur l'intégration, par de nouvelles instructions adaptées avec la technologie cible.

BIBLIOGRAPHIE

[1] W.K. Dawson, R.W. Dobinson, "Buses and bus standards" ELSEVIER Computer Standards & Interfaces, p. 201-224, 1999.

[2] I. Englander, "The architecture of computer hardware systems software : an information technology approach", New York, 2000.

[3] Edward Solari, George Willse, "PCI Hardware and Software, Architecture and Design", Annabooks, 1998.

[4] D. Addison , T. Shanley, "PCI system architecture", Addison Wesley,1999.

[5] L. Dutrieux , D. Demigny, "Logique programmable", Eyrolles, 1997

[6] "PLDs for all", Digital Design, Electronics World, 1999.

[7] D. Robinson, P. Lysaght, G. McGregor, H. Dick, "Performance evaluation of a full speed PCI initiator and target subsystem using FPGAs", Publication Lecture Notes in Computer Science, v.1304, p. 41-50, 1997.

[8] "Use In-system programming to simplify Field Upgrades", Digital Applications, Electronic Design, p. 93-96, 1998.

[9] S. Weiss , E. Finkelstein, "Extending PCI performance beyond the desktop ", IEEE Computer Society, p. 80-87, 1999.

[10] Laverty Nwaekwe, Syeed Chowdhury, Synopsys Inc, "PCI-X boosts bus bandwidth to 1 Gbps", EDN, p. 135-144, 2000.

[11] http:// www.plxteh.com, PLX technology : the I/O interconnect solution.

[12] K. Kuusilinna, Timo Hämäläinen, Jukka Saarinen, "Field Programmable gate arry-

based PCI interface for a coprocessor system" ELSEVIER, Microprocessors and Microsystems, p. 373-388, 1999.

[13] K. Kuusilinna, Timo Hämäläinen, Jukka Saarinen, "Practical VHDL optimisation for timing critical FPGA applicationsm", ELSEVIER, Microprocessors and Microsystems, p. 459-469, 1999.

[14] E. Finkelstein, S. Weiss, "Implementation of PCI-based systems using programmable logic", IEE Proceeding Circuit Devices System, p.171-174, 2000.

[15] H. Saleh, R. Engels, R. Reinartz, P. Reinhart, F. Rongen, "A flexible compatible PCI Interface for nuclear experiments", IEEE Transactions On Nuclear Science, v. 45, p. 849-851, 1998.

[16] "Le chipset PCI Publication", PC Expert Paris, v. 55, p.112-136,1996.

[17] "PCI Local Bus Specification Revision 2.2", SIG (Special Interest Group), 1999.

[18] "PCI-X Local Bus Specification Revision 1.0", SIG (Special Interest Group), 2000.

[19] Troy Scott, ORCA Inc, "Adopting VHDL for PLD design and simulation", EDN, p. 139-148, 1998.

[20] Douglas Smith, "VHLD and Verilog fundamentals- expresiions, operands, and operators", EDN, p.207-214, 1997.

[21] Douglas Smith, "HDL basic training : top down chip design using Verilog and VHDL", EDN, p.103-112,1996.

[22] www.TechOnLine.com : Education, infromation And Resources For Electonics Engineers WorldWide.

[23] Peter Clarke, "Superlog subset to be standardized head of full language", EE Times, 2001.

[24] "VHDL+ L.R.M.Extensions to VHDL for System Specification ", ICL Design Automation,1999.

[25] Peter Ashenden, Philip Wilsey, Dale Martin, "Suave : Extending VHDL to improve Data Modeling Support", IEEE Design & Test Of Computer, p. 34-44, 1998.

[26] K. Chang, "Digital Design And Modeling With VHDL And Synthesis", IEEE Computer Society Press, 1997.

[27] Janick Bergeron, " Writing Testbenches Functional Verification of HDL Models", Kluwer Academic Publishers,2000.

[28] Ken Koffman, " Real World FPGA Design With Verilog", Prentice Hall, 2000.

[29] "VirtexII Field Programmable Gate Array", data sheet, Xilinx, 2001