Chapter 1

Simulation et estimation de performance des MPSoC au niveau CABA

1.1 Introduction

Comme ceci a été évoqué dans le chapitre d'introduction de cette thèse, le temps de simulation nécessaire pour évaluer les différentes alternatives des systèmes MPSoC au niveau RTL devient l'obstacle majeur dans la phase de conception. Pour réduire ce temps de simulation, plusieurs travaux de recherche ont été consacrés à la modélisation des systèmes MPSoC au niveau CABA (Cycle Accurate Bit Accurate). A ce niveau, les modèles de composants offrent l'avantage de garder la précision au bit et au cycle près tout en permettant une simulation plus rapide. Habituellement pour passer du niveau RTL au niveau CABA, les détails architecturaux au niveau transfert de registres sont supprimés. Ce niveau nécessite néanmoins une description détaillée de la micro-architecture du système afin de simuler un comportement correct à chaque cycle. Comme nous le verrons dans ce chapitre, la simulation d'un système MPSoC au niveau CABA nécessite par conséquent un effort considérable pour le développement des composants matériels et pour interconnecter ces composants. Par ailleurs, le temps de simulation pour évaluer les performances de chaque alternative architecturale demeure important. Toutefois une simulation préliminaire en CABA de certains composants s'avère nécessaire si l'on désire intégrer dans les niveaux de simulation à des niveaux d'abstraction plus élevés, des informations pertinentes sur les performances et la consommation d'énergie.

Ce chapitre a pour but essentiel d'étudier la problématique du temps de simulation au niveau CABA. Nous utiliserons la bibliothèque de composants SoCLib pour réaliser une étude de cas de la description d'un MPSoC au niveau CABA. Cette étude détaillée s'avère nécessaire pour deux principales raisons:

- La maîtrise de la modélisation au niveau CABA va nous aider à décrire des systèmes MPSoC à des niveaux d'abstraction plus élevés tout en gardant la possibilité d'obtenir des estimations de performance précises (chapitre **??**).

- Cette maîtrise est aussi nécessaire pour développer des modèles d'estimation d'énergie

au niveau de chaque composant pour l'évaluation de la consommation totale du système (chapitre **??**).

Ce chapitre est structuré comme suit. La section 2 détaille la description au niveau CABA construite sur la théorie des "machines à états finis (FSM) communicantes synchrones". La section 3 présente les modèles des différents composants matériels que nous avons utilisés pour composer notre plateforme MPSoC au niveau CABA. La méthodologie d'estimation des performances à ce niveau est détaillée dans la section 4. Enfin, la section 5 présente une synthèse de ce chapitre et donne un aperçu sur la méthode d'intégration des composants CABA dans notre flot de conception Gaspard.

1.2 Description d'un système au niveau CABA à l'aide de FSM

La description d'un SoC au niveau CABA émule un comportement du système à chaque cycle de façon similaire au niveau RTL. En effet, la modélisation au niveau CABA est basée sur la théorie des "machines à états finis (FSM) communicantes synchrones" (*Synchronous Communicating Finite State Machines*) [21, 11]. Une FSM est constituée d'états et de transitions entre états. Son comportement est conditionné par des variables d'entrée. L'automate passe d'un état à l'autre suivant les transitions décrites dans la FSM et en prenant en compte les valeurs des variables d'entrée. Une FSM possède par définition un nombre fini d'états.

Dans une description comportant plusieurs FSM, la communication entre ces dernières est réalisée en partageant des variables. Dans un système embarqué monoprocesseur ou multiprocesseur, chaque composant est décrit avec une ou plusieurs FSM suivant les fonctionnalités réalisées. L'ensemble des signaux qui relient les différents composants représente les variables partagées pour l'échange de données. Les deux parties calcul et communication sont exécutées à des périodes de temps discrètes. Dans le cas d'une simulation CABA, une période de temps représente un cycle d'horloge du système global.

Ainsi, pour décrire le comportement d'un composant matériel, une ou plusieurs FSM sont spécifiées et exécutées en parallèle. La figure 1.1 représente une seule FSM qui contient trois fonctions: (*Transition, Moore et Mealy*) [21]. La fonction (*Transition*) calcule le prochain état du composant en se basant sur l'état actuel et sur les valeurs des signaux d'entrée. La fonction (*Mealy*) détermine les valeurs des signaux de sortie qui dépendent des signaux d'entrée et de l'état actuel du composant. Enfin, la fonction *Moore* calcule les valeurs des signaux de sortie qui dépendent seulement de l'état actuel du composant. Dans l'implémentation des FSM que nous avons expérimentées, les spécifications du simulateur SystemC, telles que l'exécution guidée par événement discret (*event driven*), ont été utilisées pour obtenir un simulateur précis au niveau cycle.

1.3 Modèles de composant au niveau CABA pour la conception des MPSoC

La plateforme SoCLib est une bibliothèque de composants matériels qui permet de réaliser un simulateur de MPSoC au niveau CABA. Les composants de SoCLib utilisent des FSM communicantes synchrones. A travers cette plateforme, une architecture MPSoC est obtenue par instanciation de composants matériels. Ces derniers sont connectés par des signaux et respectent le protocole de communication VCI (*Virtual Component Interface*) [6]. Dans cette sec-

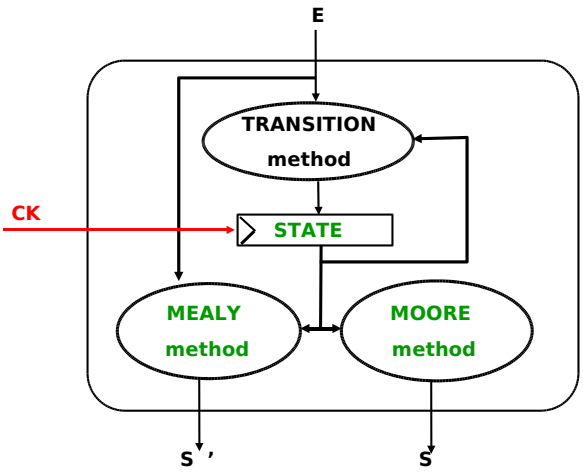


Figure 1.1: *modèle d'un composant [8]*

tion, nous allons commencer par décrire en détail ce protocole. Par la suite, les composants de SoCLib que nous avons utilisés pour concevoir notre plateforme d'expérimentation au niveau CABA seront décrits. Comme de nouveaux composants ont été intégrés à la bibliothèque de SoCLib durant cette thèse, nous profiterons de cette section pour les décrire brièvement. Ces derniers permettent de concevoir des systèmes MPSoC hétérogènes. Rappelons que l'objectif de cette section est de montrer que la description détaillée au niveau CABA d'un MPSoC nécessite un effort de développement considérable et une connaissance détaillée du comportement des composants. Cette tâche est sûrement difficile mais néanmoins nécessaire car elle permettra par la suite la modélisation de l'architecture à des niveaux d'abstraction plus élevés (cf chapitre suivant).

1.3.1 Le protocole de communication VCI

Avec l'augmentation de la complexité des architectures, il devient crucial de pouvoir réutiliser des modèles de composants disponibles. Cette approche facilite la conception et réduit le temps de développement. Pour arriver à cet objectif, il est indispensable d'utiliser un standard d'interface entre les composants. Cette approche facilite en outre l'interconnexion des modules. Il existe sur le marché plusieurs standards de protocoles.

- le standard AMBA qui a été proposé par la société ARM [2]
- le standard coreconnect de la société IBM [7]
- le standard VCI (Virtual Component Interface) proposé dans le cadre du consortium VSIA[1] (*Virtual Socket Interface Alliance*).

C'est ce dernier standard qui a été adopté dans la bibliothèque SoCLib. Le protocole VCI a été par conséquent utilisé dans cette thèse. Dans cette section, nous allons détailler le fonctionnement de VCI. Nous pensons que ces détails sont nécessaires pour comprendre la manière avec laquelle l'estimation de performance au niveau PVT sera réalisée (chapitre **??**)

[1] http://www.vsia.org

ainsi que l'estimation de la consommation (chapitre **??**). Dans cette thèse, nous supposons que les composants matériels utilisent le même protocole de communication. Une approche pour résoudre le problème d'interopérabilité entre des protocoles de communications différents est présentée dans la thèse de Lossan bonde [5]. La solution proposée est basée sur l'utilisation d'une méthodologie dirigée par les modèles.

Norme VCI

La norme VCI définit une communication point-à-point asymétrique entre un *initiateur* et une *cible*. L'initiateur émet des requêtes (de lecture ou d'écriture, simples ou par paquets) et la cible y répond. La norme VCI se décline en trois versions plus ou moins élaborées selon la nature et le rôle des composants [9, 6]. Ces trois versions sont: la Basic VCI(*BVCI*), la Peripheral VCI (*PVCI*) et l'Advanced VCI (*AVCI*). BVCI définit les signaux de base pour permettre une communication entre les initiateurs et les cibles. PVCI, comme son nom l'indique, permet de connecter des périphériques d'entrées/sorties au réseau d'interconnexion. Enfin, AVCI permet à un même initiateur d'émettre plusieurs requêtes consécutives avant de recevoir les paquets réponses. C'est cette dernière version qui a été adoptée pour interconnecter les composants de SoCLib.

Les différents signaux dans *AVCI* sont résumés dans le tableau 1.1. Le nombre entre parenthèses correspond au nombre de bits du port, et b représente le nombre d'octets par donnée. Dans cette figure $b \in \{1,2,4\}$, $e \in \{1,2,3\}$ *et* $n \in \{0..64\}$. Dans la plateforme SoCLib, nous avons $b = 4(unmot = 4octets)$, $e = 3$ *et* $n = 32$.

L'interface *AVCI* est constituée de deux paquets de signaux: les signaux de commande (*CMD*) et les signaux pour acheminer la réponse(*RSP*). Ces deux paquets fonctionnent de façon asynchrone. Ainsi, il est possible d'envoyer plusieurs requêtes consécutives avant de recevoir les premières réponses. Ces dernières peuvent en plus arriver dans un ordre différent de celui des requêtes. Pour permettre cela, dans le protocole *AVCI*, de nouveaux signaux ont été ajoutés à BVCI. Il s'agit de SCRID, TRDID, PKTID, RSCRID, RTRDID et RPKTID. Les signaux CMDACK, CMDVAL, RSPACK et RSPVAL sont utilisés pour établir un protocole de communication de type *handshake*.

Le fonctionnement du protocole *AVCI* peut être décrit comme suit: un *initiateur* envoie un signal CMDVAL à une *cible* pour l'informer que des données valides sont prêtes à être émises. Ces données peuvent être mises dans un ou plusieurs paquets et chaque paquet peut contenir un ou plusieurs mots. La cible, à son tour, informe l'initiateur par le signal RSPACK de son état (libre ou occupée). Si la cible est libre, le cycle de transfert commence. Le nombre de cycles nécessaires pour terminer la transaction dépend du nombre de mots dans le paquet de données.

Le chronogramme de la figure 1.2 illustre des séquences de transfert où la cible traite des requêtes de différentes tailles. Comme nous pouvons le constater sur ce chronogramme, quand l'initiateur a une requête à soumettre, il active le signal CMDVAL pour informer la cible. Le transfert est déclenché quand la cible active son signal RSPACK. Le signal EOP valide l'arrivée du dernier mot mettant fin au transfert. Le type d'opération lecture ou écriture est déterminé par le signal CMD. Pour implémenter le protocole de communication présenté, dans chaque interface de type VCI d'un composant initiateur ou cible nous avons besoin de deux FSM. La première pour gérer les signaux de commande (FSM VCI_CMD) et la seconde pour gérer les signaux de réponse (FSM VCI_RSP).

Nom (taille en bits)	Origine	Description
CMDACK (1)	Initiateur	Requête acceptée
CMDVAL (1)	Initiateur	Requête valide
ADDRESS (n)	Initiateur	Adresse
BE (b)	Initiateur	Byte Enable : octets concernés par la requête
CMD (2)	Initiateur	Type d'opération: lecture ou écriture (Word, Half word, Byte)
CONTIG (1)	Initiateur	Adresses contigues
WDATA (8b)	Initiateur	Donnée à écrire
EOP (1)	Initiateur	Marqueur de fin de paquet
CONS (1)	Initiateur	Adresses constantes
PLEN (8)	Initiateur	Longueur du paquet en octets
WRAP (1)	Initiateur	Adresses repliées
CFIXED (1)	Initiateur	Définition d'une chaîne
CLEN (8)	Initiateur	Nombre de paquets chaînés
SCRID (8)	Initiateur	Numéro d'initiateur
TRDID (8)	Initiateur	Numéro de thread
PKTID (8)	Initiateur	Numéro de paquet
RSPACK (1)	Cible	Réponse acceptée
RSPVAL (1)	Cible	Réponse valide
RDATA (8b)	Cible	Donnée lues
REOP (1)	Cible	Marqueur de fin de paquet
RERROR (e)	Cible	Signalisation des erreurs
RSCRID (8)	Cible	Numéro d'initiateur
RTRDID (8)	Cible	Numéro de thread
RPKTID (8)	Cible	Numéro de paquet

Table 1.1: *Ports de la norme VCI*

1.3.2 Modèle du processeur MIPS R3000

Alors que les anciennes générations de systèmes embarqués étaient réalisées uniquement à base de circuits dédiés ASIC, donc sans processeurs, actuellement le nombre de processeurs dans les systèmes embarqués augmente de plus en plus. Cette solution est attractive à plus d'un titre. En effet, l'utilisation d'unités programmables (les processeurs) permet de réduire le temps de conception, simplifie les tests du circuit et offre enfin un coût de fabrication intéressant. Ainsi, la part et le coût du logiciel dans les systèmes embarqués en général et dans les systèmes embarqués hautes performances en particulier, ne cesse d'augmenter. Néanmoins, l'utilisation des processeurs dans les systèmes embarqués pose le problème des respects des contraintes de temps et de consommation d'énergie. La solution que nous proposons dans le cadre de cette thèse consiste à utiliser une architecture multiprocesseur pilotée par une fréquence d'horloge relativement réduite d'un coté et d'adapter l'architecture du ou des processeurs aux spécificités de l'application de l'autre coté. Cette opération d'adaptation de l'architecture consiste en réalité à trouver la configuration (déterminer les paramètres) matérielle permettant d'obtenir un maximum de performances avec le minimum de consommation d'énergie.

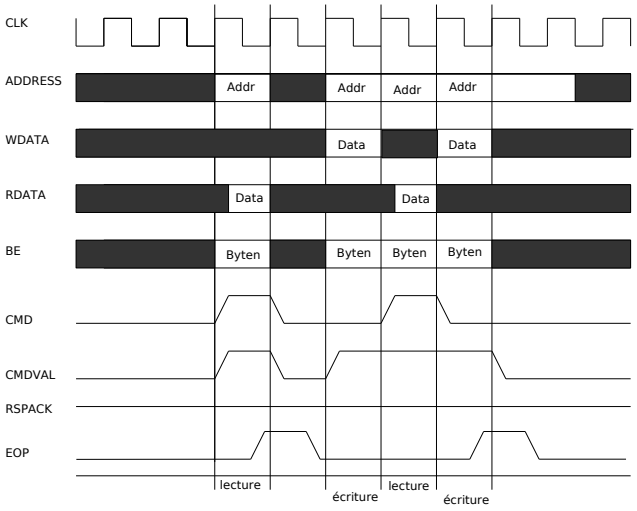


Figure 1.2: *Exemple de transactions en utilisant le protocole VCI*

Comme nous le verrons dans les chapitres suivants, notre solution n'est efficace que si nous disposons d'outils permettant d'évaluer les performances et la consommation des différentes configurations dans un temps réduit. Chaque configuration se caractérise par rapport aux autres configurations par un certain nombre de paramètres comme: le nombre de processeurs, le nombre d'UAL (Unité Arithmétique et Logique) dans chaque processeur, la taille des caches, l'intégration ou non d'unités spécialisées (co-processeurs, FPGA), le type de NoC et les débits sur ce dernier, etc. La figure 1.3, donne un aperçu sur les processeurs les plus couramment utilisés dans la réalisation des systèmes sur puce. Comme nous pouvons le voir sur cette figure, depuis plusieurs années les processeurs ARM occupent une place importante.

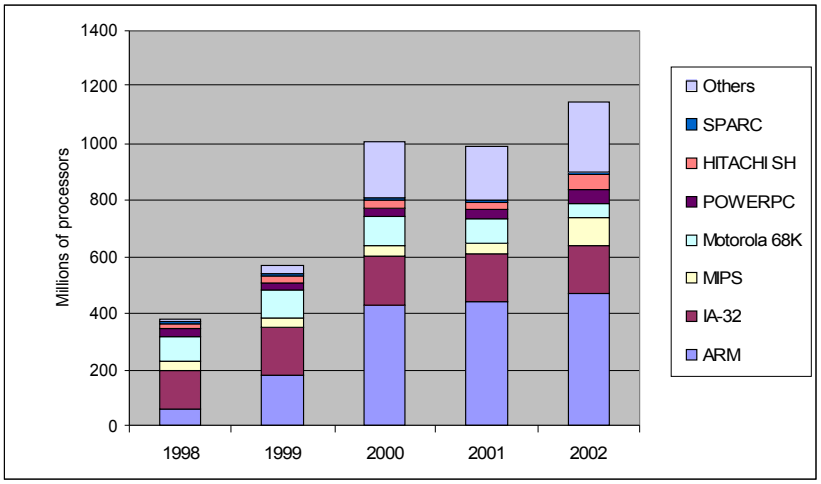


Figure 1.3: *Parts des fabricants dans le marché des processeurs embarqués [18]*

Type	Scalaire
Ordonnancement des instructions	Dans l'ordre
Étages de pipeline	5
Gestion du pipeline	solution matérielle
Prédiction du branchement	N'est pas implémentée
Latence d'exécution d'une instruction	1 cycle
Plage de Fréquence	100-200 MHz

Table 1.2: *Caractéristiques du MIPS R3000*

Le modèle de processeur qui a été utilisé est le MIPS R3000. Ce choix est justifié par l'existence d'un composant MIPS R3000 dans la bibliothèque SoCLib au moment du démarrage de la thèse. D'autres processeurs sont en cours de développement pour une future intégration à SoCLib. Le MIPS R3000 possède une architecture RISC avec les caractéristiques présentées dans le tableau 1.2. Dans cette étude, nous considérons que les mémoires caches d'instructions et de données sont des modules séparés du processeur qui seront présentées dans une section à part.

Le MIPS R3000 possède une architecture pipeline à 5 étages à savoir *Fetch Instruction* "I", *Decode Instruction* "D", *Execute Instruction* "E", *Memory Acess* "M" et *Write Back* "W". La figure 1.4 représente le chemin de données du processeur lors de l'exécution d'une opération d'addition. Une nouvelle instruction est chargée à chaque cycle dans l'étage "I", pour être décodée dans l'étage "D". Le résultat de l'instruction est par la suite calculé dans l'étage "E". Tout accès à la mémoire de données est réalisé dans l'étage "M" et, finalement, l'enregistrement du résultat dans le registre destination se fait à l'étage "W".

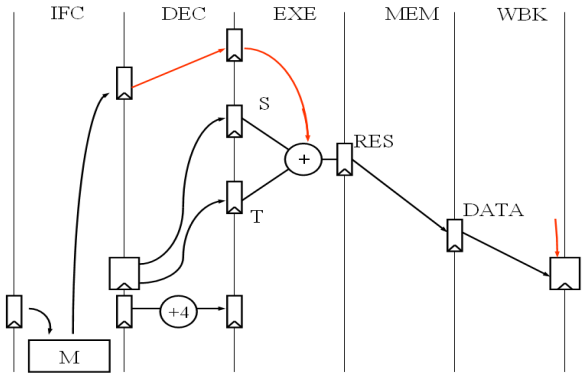


Figure 1.4: *Chemin de données pour une opération d'addition*

Le MIPS R3000 est connecté aux autres unités à travers 3 ports (figure 1.5):

1. Un port contenant des Lignes d'interruption, utilisées pour le traitement des événements asynchrones. Ici, l'interruption correspond à une requête matérielle pour un traitement spécifique par le processeur (E/S, ordonnancement des tâches, etc.).

2. Un port avec le cache d'instructions qui est composé principalement des signaux d'adresses et de données et du signal *Miss* pour informer le processeur de l'occurrence d'un défaut de cache.

3. Un port avec le cache de données qui est composé principalement des signaux d'adresses et de données, du signal *Miss* pour informer le processeur d'un défaut de cache, du signal *Type* pour définir le type d'opération (lecture ou écriture) et enfin du signal *UNC* pour indiquer au cache que la donnée ne peut pas être chargée (non-cachable).

En plus de ces 3 ports, 2 signaux: "reset" pour la remise à zéro du processeur et le signal d'horloge sont prévus. L'interface du processeur MIPS R3000, reliée aux caches d'instructions et de données, est optimisée afin de pouvoir charger une donnée et une instruction par cycle. Lors d'un défaut de cache (*cache miss*), le processeur rentre dans le mode inactif (*idle*) jusqu'à ce qu'il reçoit l'instruction ou la donnée manquante. Pendant ce temps d'inactivité du processeur, la consommation d'énergie est réduite puisqu'aucune activité n'est réalisée.

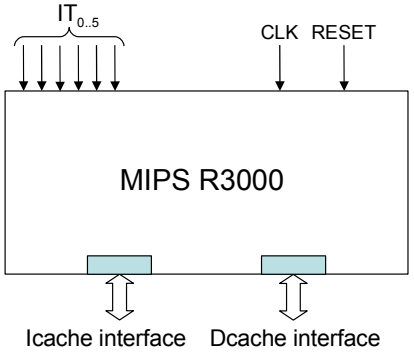


Figure 1.5: *Interface du MIPS R3000*

Cette architecture du processeur MIPS R3000 est implémentée en utilisant un composant SystemC. Le comportement du composant est décrit avec une FSM qui implémente les 3 fonctions *Transition, Moore et Mealy*, déclarées en tant que processus SC_METHOD. En effet, ces 3 fonctions doivent d'être exécutées entièrement à chaque cycle d'horloge afin de calculer en premier lieu le nouvel état du composant et en deuxième lieu l'état des signaux de sorties. Pour cette raison la fonction de transition est sensible au front montant de l'horloge alors que les fonctions de génération Moore et Mealy sont sensibles au front descendant de l'horloge. En réalité cette démarche est valable pour tous les composants décrits au niveau CABA. Pour éviter la redondance dans les descriptions des autres composants de l'architecture, nous allons insister sur chaque aspect particulier du composant. Le but final étant, s'il faut le rappeler, de quantifier l'effort nécessaire pour obtenir une description au cycle près pour ces composants. Dans chaque composant décrit au niveau CABA, nous avons besoin de déclarer au début du module composant (N.B.: les numéros d'étapes renvoient au code SystemC présenté ci-après):

1. Les noms des ports pour la communication avec les autres modules (étape 0)
2. Les noms des signaux pour modéliser des registres (étape 1)
3. Les noms des variables pour modéliser des constantes architecturales (étape 2)
4. Les fonctions décrivant le comportement (étape 3)

Le texte ci-après montre un exemple de déclaration du composant MIPS R3000 décrit au niveau CABA.

```
struct SOCLIB_MULTI_MIPS:sc_module {
    // Etape 0
    sc_in<bool>  CLK;
    sc_in<bool>  RESETN;
    // Interface avec cache d'instructions
    ICACHE_PROCESSOR_PORTS  ICACHE;
    // Interface avec cache de données
    DCACHE_PROCESSOR_PORTS  DCACHE;
    // Signaux d'interruption
    sc_in<bool>    IT_5;
    sc_in<bool>    IT_4;
    // Déclaration des registres (étape 1)
    sc_signal<int> GPR[32];
    sc_signal<int> PC;
    sc_signal<int> HI;
    sc_signal<int> LO;
    sc_signal<int> IR;
    // Constantes architecturales (étape 2)
    int  MSB_NUMBER; // nombre de bits MSB dans l'adresse
    ...
    SOCLIB_MULTI_MIPS(sc_module_name insname)
    {
        // Déclaration des fonctions pour décrire le comportement (étape 3)
        SC_METHOD (transition);
        sensitive_pos << CLK;
        SC_METHOD (genMealy);
        sensitive_neg << CLK;
        SC_METHOD (genMoore);
        sensitive_neg << CLK;
    }
}
```

1.3.3 Modèle du composant xcache

A cause de l'écart, de plus en plus important, entre les vitesses de fonctionnement des mémoires et celui des processeurs, les opérations d'accès mémoires sont devenues la cause principale de la chute des performances. Ce problème devient encore plus gênant dans le contexte des systèmes MPSoC où généralement il est nécessaire de transiter par un réseau d'interconnexion avant d'accéder à la mémoire. Cela entraîne une autre augmentation du temps d'accès. L'utilisation des systèmes à plusieurs niveaux de mémoires (ou hiérarchie mémoire) apparaît comme une solution intéressante. Cette hiérarchie comporte plusieurs niveaux. Au niveau le plus bas de la hiérarchie nous retrouvons les mémoires partagées, de taille significative mais avec des temps d'accès relativement importants. A l'opposé au niveau le plus haut, nous avons les registres en petit nombre et un temps d'accès très court.

Les mémoires caches représentent un compromis entre ces deux niveaux extrêmes. Elles sont utilisées pour mémoriser les données et les instructions récemment accédées. Ce type de mémoire se caractérise par sa rapidité d'accès, de l'ordre d'un cycle processeur et par une capacité limitées de quelques dizaines de kilo-octets (Ko) [15].

Le composant cache (nommé xcache) de la SoCLib est un composant générique construit sur une base de mémoire de type SRAM. Ce type de RAM présente l'avantage de présenter un temps d'accès réduit. Xcache contient en réalité deux mémoires caches: un cache pour les instructions et un cache pour les données. Ces deux derniers sont indépendants, mais partagent la même interface VCI (figure 1.6) pour l'accès au réseau d'interconnexion. Chaque cache est contrôlé par une FSM indépendante. Les champs VCI adresses et données sont codés sur 32 bits chacun. La taille de chaque cache est déterminée en fixant le nombre de lignes et le nombre de mots par ligne. Les deux caches instructions et données ont un degré d'associativité égale à 1 (*direct mapped cache*). La stratégie d'écriture dans le cache de données est l'écriture simultanée (*write through*).

La plateforme SoCLib utilise une solution logicielle simple pour résoudre le problème de cohérence des caches. En effet, à partir d'indications fournies par le programmeur, les données sont stockées dans deux segments (ou deux espaces d'adressage) différents de la mémoire. Ainsi, nous distinguons deux types; des données locales ou privées à un processeur et les données partagées entre les processeurs. Les données locales peuvent être chargées dans le cache. Les données partagées entre les processeurs peuvent aussi être chargées dans les caches à condition que ces dernières soient lues uniquement. Dans le cas contraire, les données ne sont jamais mises dans le cache. Comme, nous pouvons l'imaginer cette solution simple ne permet pas d'optimiser l'utilisation des caches. En effet, dans les applications à traitement de données intensif, il est possible d'avoir une quantité importante de données partagées et modifiées par un ou plusieurs processeurs. Par conséquent, l'utilisation des caches est réduite et un trafic important sur le réseau d'interconnexion est généré. Il existe dans la littérature plusieurs solutions pour résoudre ce problème [19]. En collaboration avec l'école nationale des ingénieurs de Sfax (ENIS), nous avons commencé le développement d'une solution matérielle plus performante en termes de performance et de consommation d'énergie. Dans cette thèse, nous utiliserons uniquement la solution logicielle sans gestion de la cohérence.

En cas de défaut de cache (*Miss*), une mémoire tampon (*buffer*) permet de recevoir la ligne manquante. Ce tampon est aussi utilisé pour recevoir la donnée en cas de lecture non cachée. Enfin, le xcache possède une FIFO de requêtes (*request FIFO*) pour envoyer les demandes de lecture ou d'écriture vers les autres composants (mémoire partagée ou DMAC) via le réseau d'interconnexion. Ainsi, le contrôleur de l'interface VCI lit dans la FIFO et construit un paquet de commande lorsqu'il trouve plusieurs adresses appartenant à la même page. Par défaut, une page a une taille de 4 Ko. Notons que le processeur reste dans le mode inactif (*idle*) en attente de la donnée dans les cas de défaut de cache, lecture non cachée, écriture ou lorsque la FIFO est pleine.

Pour implémenter le composant xcache au niveau CABA nous avons suivi la même démarche que celle appliquée au processeur. Ici, nous insistons sur l'effort de développement pour décrire le comportement des FSM de la xcache. Pour ce composant, nous avons besoin de 4 FSM à savoir:

1. FSM_DC pour modéliser le cache de données
2. FSM_IC pour modéliser le cache d'instructions

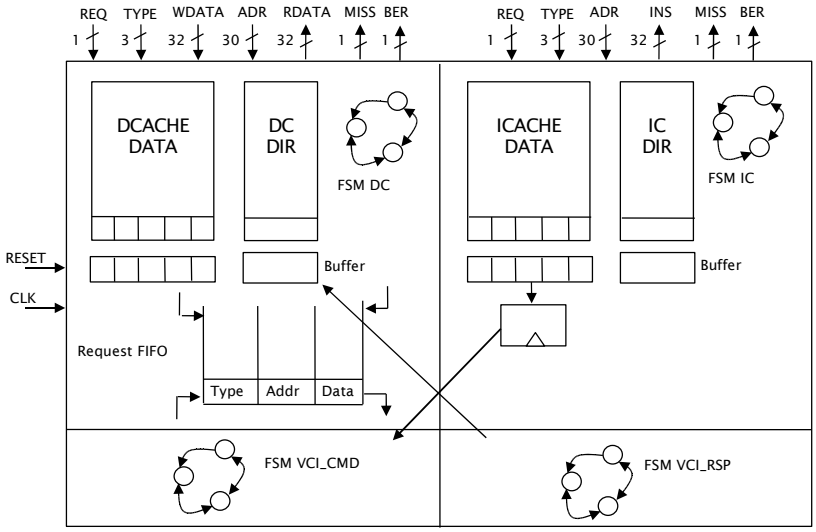


Figure 1.6: *Le composant xcache de SoCLib*

3. FSM_VCI_CMD pour modéliser l'interface de commande VCI
4. FSM_VCI_RSP pour modéliser l'interface de réponse VCI

Chaque FSM peut avoir plusieurs états. Là aussi, nous avons les 3 fonctions *Transition, Moore et Mealy*. La figure 1.7 montre l'exemple de la FSM qui décrit le cache de données (FSM_DC). Plusieurs états sont nécessaires pour assurer le bon fonctionnement du composant. Par exemple, l'état MISS_REQ correspond à une demande de donnée suite à un défaut de cache, l'état W_UPDT correspond au chargement d'une donnée dans le cache, etc.

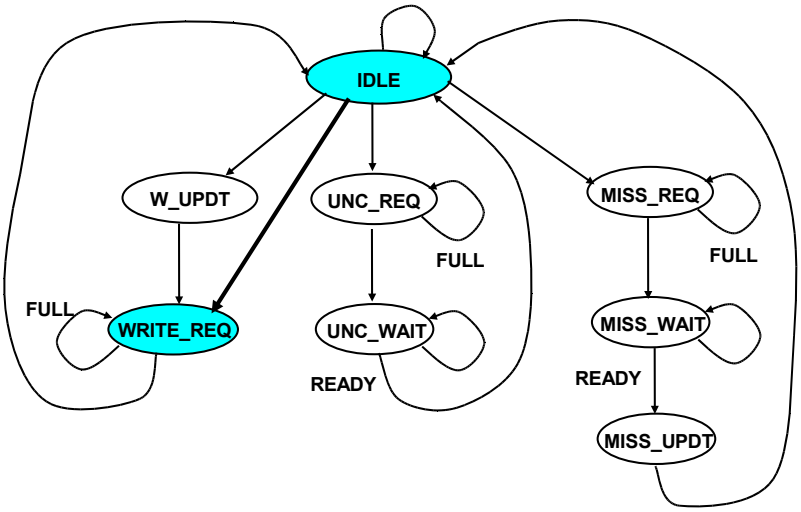


Figure 1.7: *FSM du cache de données*

Le texte ci-dessous montre un exemple de déclaration d'une FSM au niveau CABA.

```
// DCACHE_FSM STATES
switch (DCACHE_FSM) {

case DCACHE_INIT :
        DCACHE_TAG[DCACHE_CPT_INIT] = 0;
        DCACHE_CPT_INIT = DCACHE_CPT_INIT - 1;
        if (DCACHE_CPT_INIT == 0) { DCACHE_FSM = DCACHE_IDLE;}
break;

case DCACHE_IDLE :
if ((dcache_validreq == true) && (dcache_write == false) && (dcache_inval == false) &&
    (dcache_unc == false) && (dcache_hit == false))  { DCACHE_FSM = DCACHE_MISSREQ; }
else if ((dcache_validreq == true) && (dcache_write == true) &&
         (dcache_hit == false))  { DCACHE_FSM = DCACHE_WRITEREQ; }
else if ((dcache_validreq == true) && (dcache_write == true) &&
 (dcache_hit == true))  { DCACHE_FSM = DCACHE_WUPDT; }
else if ((dcache_validreq == true) && (dcache_inval == true) &&
 (dcache_hit == 1))  { DCACHE_FSM = DCACHE_INVAL; }
    ...
break;

case DCACHE_WUPDT :
    ...
    break;

case DCACHE_WRITEREQ :
    ...
break;

case DCACHE_MISSREQ :
    ...
break;

case DCACHE_MISSWAIT :
    ...
break;

    ...

} // end switch DCACHE_FSM
```

1.3.4 Modèle du réseau d'interconnexion

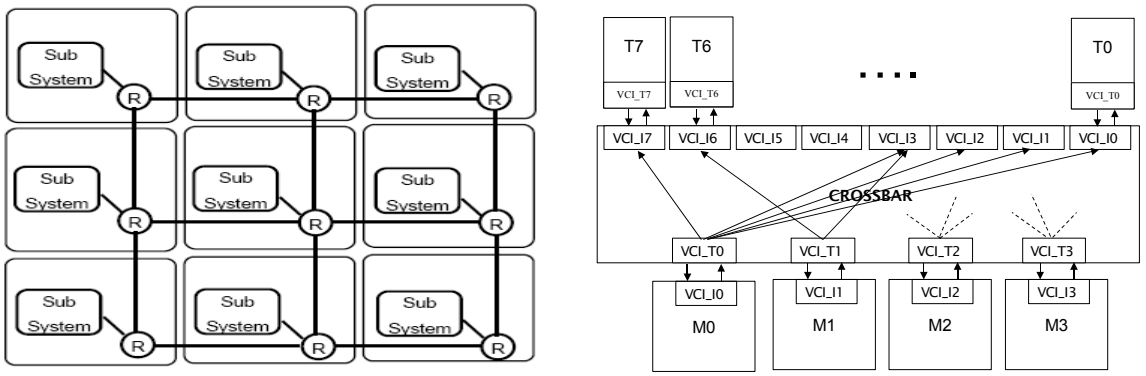
Le réseau d'interconnexion utilisé pour connecter les différents composants d'un MPSoC [4] joue un rôle d'une grande importance pour déterminer les performances du système. En effet, si les architectures MPSoC se présentent comme une alternative intéressante par rapport aux ASIC pour les prochaines générations de systèmes sur puce, il devient primordial de

trouver un moyen de communication efficace entre les processeurs et les mémoires.

Différentes topologies de réseaux d'interconnexion existent. L'architecture en bus partagé est sans aucun doute la plus populaire. Néanmoins, ces dernières années des architectures de réseaux plus performantes telles que les grilles, les bus hiérarchiques, le crossbar et réseau multi-étages ont été proposées [4]. Dans le cadre de cette thèse, l'objectif n'est pas de proposer de nouvelles architectures de réseaux d'interconnexion, mais, là aussi, de concevoir des outils permettant d'accélérer la simulation pour explorer l'espace des différentes alternatives. Pour le réseau d'interconnexion, en plus des critères: délais de transmission, qui doivent être courts, et consommation d'énergie qui doit être faible, d'autres critères peuvent être déterminants dans le choix du réseau. Parmi ces critères, nous trouvons:

- l'extensibilité du réseau pour pouvoir facilement intégrer un plus grand nombre d'éléments sur le réseau
- la redondance des chemins entre 2 composants, afin de tolérer des pannes et, surtout, pour éviter les goulots d'étranglement dans le réseau.
- La reconfigurabilité dynamique du réseau afin de s'adapter aux besoins en communication de l'application.

La plateforme SoCLib que nous avons utilisée possède deux types de réseaux d'interconnexion, le crossbar et le DSPIN (*Distributed Scalable Integrated Network*) [16]. Le crossbar permet une liaison directe entre les initiateurs (M) et les cibles (T) (figure 1.8(b)).



(a) Architecture d'un système à mémoire distribuée (b) Architecture d'un système à mémoire partagée

Figure 1.8: différents types d'architectures MPSoC

Ce support de communication possède une large bande passante (*O(max(M,T))*) mais a comme inconvénient majeur la complexité de la conception en termes de points de connexion (*O(MxT)*). Cette complexité s'accentue avec l'augmentation du nombre de modules connectés. Pour cette raison, il est préconisé pour un nombre limité de composants interconnectés.

Le réseau DSPIN a une topologie en grille à deux dimensions (figure 1.8(a)) permettant de relier un nombre plus important de composants ou de sous-systèmes avec un coût relativement faible par rapport au crossbar. Le DSPIN permet ainsi une bande passante équivalente à celle du crossbar avec une complexité de conception réduite (*O(log(M+T))*). Néanmoins, ce dernier à un diamètre ($O(\sqrt{(M+T)})$), qui correspond au nombre maximal

de routeurs à traverser, plus grand que celui du crossbar (*1*). Même si cela ne remet pas en cause la méthodologie proposée, dans cette thèse, nous nous intéressons aux systèmes multiprocesseurs à mémoires partagées. Nous nous sommes limités au crossbar qui est adapté pour les systèmes MPSoC étudiés. En effet, le nombre de composants que nous avons modélisé dans notre plateforme ne dépasse guère la dizaine de composants.

En plus du composant crossbar de SoCLib, nous avons développé le composant bus standard afin de réaliser des études comparatives. Ces deux composants sont génériques et par conséquent différents paramètres, tels que les latences de connexion entre les ports, peuvent être modifiés. Le protocole de communication utilisé par le crossbar est VCI pour la connexion avec les autres composants. Afin d'assurer un fonctionnement correct, lorsque le composant connecté est de type *initiateur*, c'est un port de type *target* dans le crossbar qui est utilisé et vice-versa. Dans la figure 1.8(b), les ports target (resp. initiateur) du crossbar sont notés VCI_T_i (resp. VCI_I_i). Le fonctionnement de notre système MPSoC est totalement synchrone et de ce fait tous les composants sont cadencés par un seul signal d'horloge.

Du point de vue du fonctionnement du crossbar, à chaque cycle ce dernier réalise un test sur toutes les entrées du côté des initiateurs afin de récupérer les nouvelles requêtes de commande. Pour permettre à chaque initiateur de transmettre plusieurs requêtes successives, nous avons associé à chaque port d'entrée du réseau crossbar une FIFO pour stocker les demandes. La taille de cette FIFO est un paramètre à fixer par le concepteur. Pour résoudre le problème d'accès simultané à une même cible, le crossbar utilise la stratégie d'arbitrage par tourniquet (ou Round-Robin). Cette fonctionnalité est implémentée dans un composant SystemC au niveau CABA.

Pour montrer la complexité de la modélisation au niveau CABA de ce crossbar, c'est la génération des signaux de sorties des ports VCI à chaque cycle d'horloge qui est détaillée ci-après. L'état de sortie de chaque signal du port VCI est calculé dans la fonction Moore de la FSM qui décrit le composant. Le texte ci-après montre la boucle appliquée sur chaque port de type initiateur du réseau d'interconnexion afin de déterminer l'état des signaux de sortie. Dans cet exemple, nous nous intéressons aux paquets "réponses" qu'il faut transmettre des ports VCI_I_i aux ports VCI_T_i.

```
void SOCLIB_VCI_GMN::genMoore()
{
// Boucle sur les ports de type initiateur
for (i=0 ; i<NB_INITIAT ; i++) {
 k     = T_ALLOC_VALUE[i]; //détermination du numéro de la cible
     if (RSP_FIFO_STATE[i][k] == not_empty) { //vérifier que le port n'est pas vide
        ptr   = RSP_FIFO_PTR[i][k]; //détermination du numéro d'ordre
        cmd   = RSP_FIFO_CMD[i][k][ptr]; //génération du mot de commande
        if (T_ALLOC_STATE[i] == true) { //vérifier que le port est prioritaire
            T_VCI[i].RSPVAL = true;     //réponse valide
            T_VCI[i].REOP = (bool) (cmd>>3  & 0x00000001);
            T_VCI[i].RTRDID = (vci_id_type) (cmd>>4  & 0x000000FF);
            T_VCI[i].RPKTID = (vci_id_type) (cmd>>12 & 0x000000FF);
            //lecture du numéro de l'initiateur
            T_VCI[i].RSCRID = (vci_id_type) (cmd>>20 & 0x000000FF);
            \\lecture de la donnée à envoyer
```

```
            T_VCI[i].RDATA = (vci_data_type) RSP_FIFO_DATA[i][k][ptr];
            ....
         }
      }
      else{
        //si port vide, alors bus au repos
        .....
      }
} // end loop initiators
```

1.3.5 Modèle de la mémoire de données et d'instructions

Les besoins en ressources mémoires des applications de traitement de données intensif deviennent de plus en plus importants. Ces dernières peuvent réduire les performances et/ou augmenter la quantité d'énergie électrique totale consommée. En effet, plus l'application est complexe, plus grands seront ces besoins en mémoire pour stocker les instructions et les données. Selon les prévisions de l'ITRS (International Technology Roadmap for Semiconductors) (figure 1.9), dans les prochaines générations de systèmes embarqués les unités de mémorisation (caches, scratchpads, buffers et autres) occuperont plus de 80% de la surface de la puce, alors que la surface de la logique réutilisable (les processeurs) et la surface des unités spécifiques (comme les ASIC) ne dépasseront pas les 5% chacun.

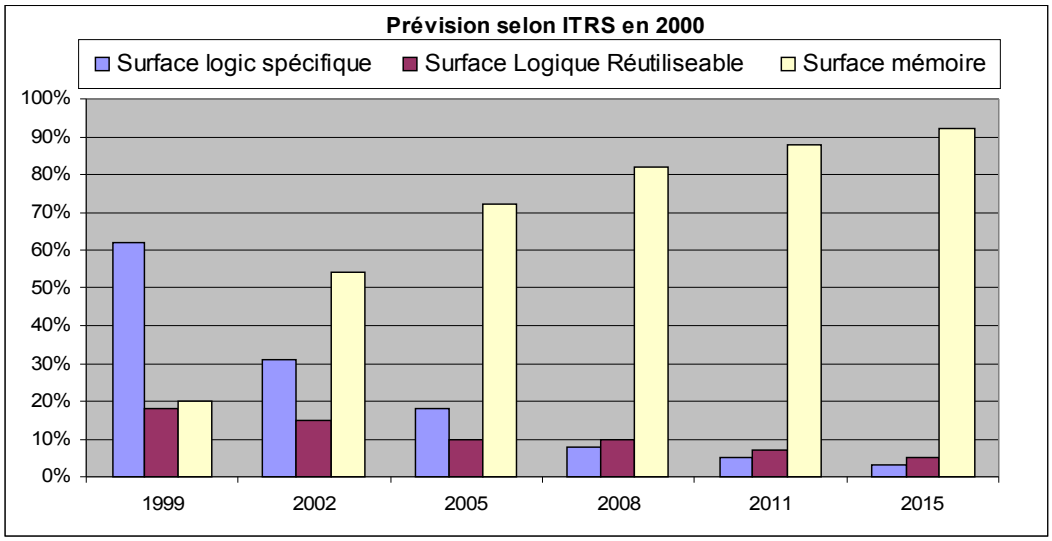


Figure 1.9: *Evolution des surfaces pour "Logique spécifique", "logique réutilisable" et "mémoire" dans les de SoC*

L'intégration des mémoires dans les systèmes embarqués peut se faire en utilisant différentes technologies: SRAM, DRAM, FRAM, EPROM, FLASH, etc. Pour chaque technologie plusieurs architectures et optimisations sont possibles dans la perspective d'augmenter les performances et réduire la consommation d'énergie [17]. Même si la part des mémoires embarquées non volatiles (ou mortes) devient de plus en plus grande pour stocker les applications dans les systèmes embarqués (tel que les mémoires FLASH), dans cette thèse nous

nous sommes limités aux mémoires RAM embarquées, et plus spécialement les mémoires SRAM qui offrent une simplicité d'intégration et un temps d'accès plus court.

Dans un système MPSoC, il existe deux types d'architecture mémoire:

1. Mémoire globale partagée accessible par tous les processeurs. Dans ce cas les données sont entièrement stockées dans cette mémoire. Ce type d'architecture est utilisé dans les applications où les données sont fortement dépendantes [15].

2. Mémoire distribuée. Dans ce cas les données sont découpées et réparties dans les mémoires de chaque processeur. La mémoire associée à un processeur peut être accessible aux autres processeurs (mémoire distribuée partagée).

Le choix entre ces deux types d'architectures pour implémenter une application donnée n'est pas simple. En effet, il est contraint en premier lieu par la nature de l'application et en deuxième lieu par le type des composants utilisés (processeur, réseau de connexion, la bande passante des bus). Dans cette thèse, nous nous limitons aux architectures MPSoC à mémoire partagée.

Au niveau CABA, les différentes tâches à exécuter sont présentées sous forme d'un code binaire exécutable. On associe à chaque segment d'instructions et de données une adresse de base et une taille en octets définies par le concepteur du système(figure 1.10). Avant de commencer la simulation, les différents segments sont stockés dans des mémoires partagées d'instructions et de données comme le montre la figure 1.10. Les mémoires d'instructions et de données utilisées dans SoCLib sont de type SRAM (*Static Random Access Memory*) monoport. Ce type de mémoire exige certes plus d'espace (4 fois plus grand) sur la puce par rapport à la technologie DRAM (*Dynamic Random Access Memory*) mais présente l'avantage de rapidité (cycles plus courts).

Notons que cette architecture du module mémoire peut être modifiée pour permettre des accès multiples à un même module et améliorer ainsi les performances. Le nombre de modules mémoire, la taille de chacun des modules et la latence des accès, peuvent être modifiés. Ces paramètres déterminent un espace de solutions pouvant être exploré. Le concepteur peut aussi modifier le placement des différents segments de données pour alléger les goulots d'étranglement dans le réseau d'interconnexion et améliorer les performances du système.

Un composant SystemC intégrant les fonctionnalités de la mémoire (de données ou d'instructions) a été conçu. Le texte ci-dessous montre l'implémentation de la fonction "Transition" pour calculer le nouvel état de la mémoire à chaque cycle d'horloge. A partir de l'état précèdent du composant (lecture, écriture ou inactif) et des valeurs des signaux d'entrées (comme CMDVAL de VCI), un nouvel état de la FSM est déterminé. Dans un souci de simplicité, nous ne détaillons que les transitions à partir de l'état "repos".

```
void transition()
{
case 0: //Etat inactif
if (VCI.CMDVAL == true) { // si requête valide
        // Identifier le type d'opération et l'adresse concernée
adrword = ((int)VCI.ADDRESS.read()) & 0xFFFFFFFC;
wdata   = (int)VCI.WDATA.read(); // données à écrire
be      = (int)VCI.BE.read(); //byte enable
```

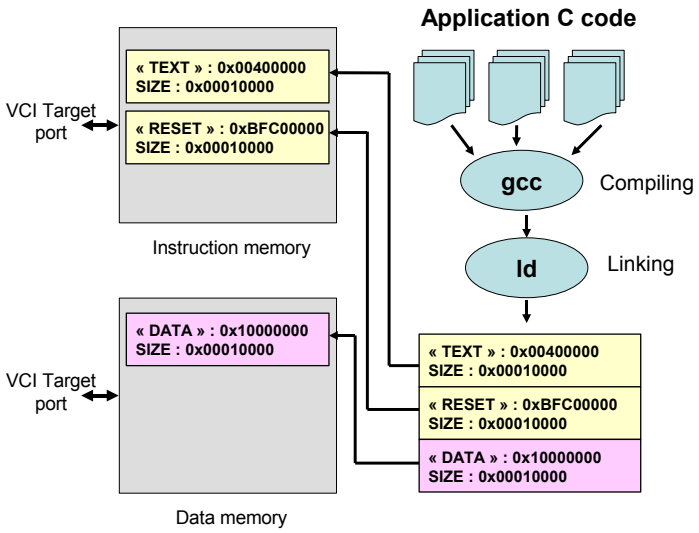


Figure 1.10: *Mémoires d'instructions et de données*

```
cmd     = (int)VCI.CMD.read(); //opération: lecture ou écriture
rdata = 0;
rerror  = 1;
for (int i = 0 ; ((i < NB_SEG) && (rerror != 0)) ; i++) {
            // Identifier le segment de données
if ((adrword >= BASE[i]) && (adrword <  BASE[i] + SIZE[i])) {
                //Dans quel segment est la donnée?
                // fonction de lecture et écriture de données dans un segment
rdata  = rw_seg(RAM[i], (adrword - BASE[i]) >> 2, wdata, be, cmd);
rerror = 0;
} // end if
} // end for
        //préparation du paquet réponse:
FIFO_RDATA[0]  = rdata;
FIFO_RERROR[0] = rerror;
FIFO_RSCRID[0] = (int)VCI.SCRID.read();
FIFO_RTRDID[0] = (int)VCI.TRDID.read();
FIFO_RSCRID[0] = (int)VCI.SCRID.read();
FIFO_REOP[0] = (int)VCI.EOP.read();
FIFO_STATE = 1; // on va à l'état 1
} else {
FIFO_STATE = 0; // si non on reste à l'état repos
}
break;
case 1:
...
case 2:
...
```

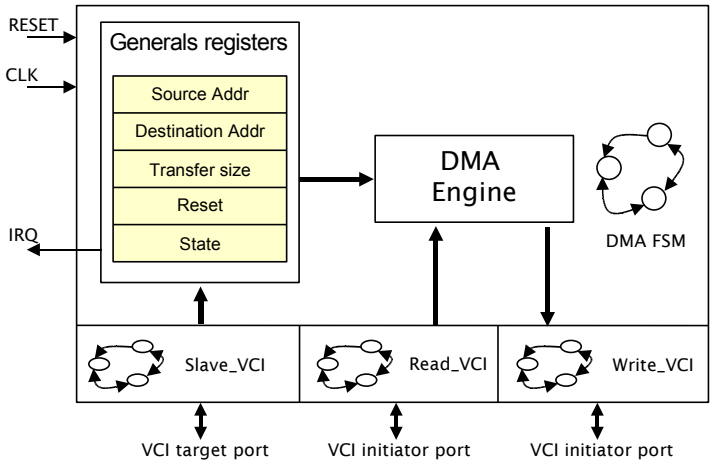


Figure 1.11: *Architecture du controleur DMA*

```
break;
}
```

1.3.6 Modèle du contrôleur DMA

Dans les applications de traitement de données, une quantité importante de données est transférée entre les modules mémoires et les périphériques. Pour réduire les délais de ces transferts, nous avons conçu un contrôleur DMA (*Direct Memory Access controler*). Ce type de composant permet d'accélérer le transfert de données, ce qui libère les processeurs de ces tâches et évite de passer par des mémoires intermédiaires.

La figure 1.11 illustre l'architecture du composant développé. Il possède 3 ports VCI, le premier est de type *cible* (Slave_VCI) pour permettre aux processeurs de configurer le DMAC via les registres du DMA. Les deux autres ports sont de type *initiateur* (Read_VCI et Write_VCI) permettant un transfert de données simultané entre une adresse source et une adresse destination. Notre contrôleur DMA contient 5 registres. Les adresses source et destination sont spécifiées respectivement dans les registres *Source addr* et *Destination addr*. Le registre *Transfer size* détermine la taille du bloc à transférer. L'écriture dans ce registre fait démarrer l'opération de transfert de données. Le contrôleur DMA active son signal *IRQ* et initialise le registre *State* à 1 pour signaler la fin du transfert avec succès. L'écriture dans le registre *Reset* permet de désactiver le signal *IRQ* et initialise les autres registres pour un nouveau transfert. La fonctionnalité du contrôleur DMA est décrite avec un composant SystemC possédant 4 FSM (figure 1.11).

1.3.7 Modèle de l'accélérateur matériel TCD-2D

Aujourd'hui avec la complexité des applications embarquées, l'utilisation d'une approche totalement logicielle pour implémenter certains algorithmes exige un nombre de processeurs dépassant les capacités des circuits en termes de surface et de consommation. Pour résoudre ce problème, les tâches soumises à de fortes contraintes temporelles peuvent être implémentées de façon matérielle. Ceci permet d'obtenir un processeur dédié pour une fonctionnalité

donnée, ce que nous appelons ici un "accélérateur matériel". Nous obtenons ainsi une architecture mixte logicielle/matérielle pour implémenter les différentes tâches d'une application.

Dans cette thèse nous nous intéressons à l'implémentation matérielle de la transformée en Cosinus Discrète (TCD). La TCD est un algorithme très utilisé dans le domaine des applications vidéo. Elle permet la transformation d'un bloc d'image du domaine spatial au domaine fréquentiel [12]. La TCD bidimensionnelle (TCD-2D) est appliquée sur des blocs de 8x8 pixels et implémentée à travers deux TCD unidimensionnelles (TCD-1D). La première TCD-1D opère sur les 8 lignes tandis que la deuxième opère sur les 8 colonnes. Pour résoudre la complexité de calcul de la TCD et satisfaire des critères de temps réel, plusieurs architectures ont été proposées [12, 10, 20]. La base de comparaison entre ces implémentations est le nombre de multiplications (MUL) et d'additions (ADD) utilisés. L'algorithme le plus optimisé est celui de Loeffler [13] qui utilise 11 MUL et 29 ADD. Nous avons implémenté cet algorithme sur la carte de développement pour FPGA STRATIX II EP2S60 d'Altera [1] en utilisant le langage VHDL. L'objectif est d'établir une analyse temporelle ainsi qu'une évaluation de la consommation de puissance de l'accélérateur matériel développé. Cette analyse nous permettra d'estimer les performances à un niveau plus haut.

	TCD
ALUTs	1569 (3%)
E/S	195 (40%)
Blocs RAMs	0%
Blocs DSPs	22 (8%)
Fmax (MHz)	120

Table 1.3: *Résultats d'implémentation de la TCD*

Une simulation temporelle en utilisant l'outil Quartus II d'Altera [14] (tableau 1.3) montre que la fréquence de fonctionnement de la TCD-2D est de l'ordre de 120 MHz. La solution matérielle de la TCD-2D développée permet une accélération du traitement par un facteur de 200 par apport à la solution logicielle exécutée avec le processeur MIPS R3000. En effet, l'utilisation de la solution matérielle permet de réaliser une TCD-2D pour un bloc d'image (8x8 pixels) en 720 cycles au lieu des 15000 cycles sur le MIPS R3000. Pour pouvoir intégrer l'accélérateur matériel TCD-2D dans l'environnement de simulation CABA, nous avons réécrit le composant en utilisant le langage SystemC. Une interface VCI de type *cible* a été ajoutée à ce composant pour le connecter au réseau d'interconnexion. Du point de vue fonctionnement, le contrôleur DMA décrit précédemment se charge de transférer un bloc d'image (8x8 pixels) d'un composant d'entrée/sortie (*Sensor*) à l'accélérateur matériel TCD-2D. Après l'exécution de la tâche TCD-2D, le bloc traité sera transféré par le contrôleur DMA vers la mémoire de données. Cette méthodologie peut être appliquée à divers algorithmes qui peuvent être implémentés sous forme matérielle en vue d'accélérer leur exécution comme la FFT (*Fast Fourier Transform*).

1.4 Estimation de performance au niveau CABA

En utilisant les composants décrits dans la section précédente, différentes architectures multiprocesseurs peuvent être modélisées, simulées et évaluées. Ces dernières peuvent

contenir un nombre variable de processeurs, d'accélérateurs matériels, de périphériques d'entrées/sorties, etc. La figure 1.12 donne un exemple d'architecture MPSoC à base de ces composants. Cet exemple d'architecture convient en général aux applications de traitement de signal intensif nécessitant des moyens de calcul parallèles importants. Pour exécuter l'application, les différentes tâches sont placées, ensuite compilées vers les processeurs cibles. Les données manipulées sont aussi placées sur les bancs de la mémoire partagée SRAM. Pour garantir un ordonnancement correct des tâches entre les processeurs, des variables de synchronisation sont utilisées. Ces variables sont stockées en mémoire. Elles sont lues et modifiées par les différents processeurs. Pour cette raison, ces variables de synchronisation (ou sémaphores) ne doivent pas être copiées dans les caches. L'estimation des performances de l'application est donnée par le simulateur d'architecture en nombre de cycles. Pratiquement, à chaque processeur est associé un registre compteur de temps dans le composant "*Timer*". Le compteur de temps est incrémenté à chaque cycle d'horloge. Ainsi, pour estimer le temps d'exécution d'une tâche sur un processeur, il suffit de lire la valeur du registre avant et après l'exécution de la tâche. Par ailleurs, le temps d'exécution de toute l'application est donné par l'horloge globale du simulateur SystemC.

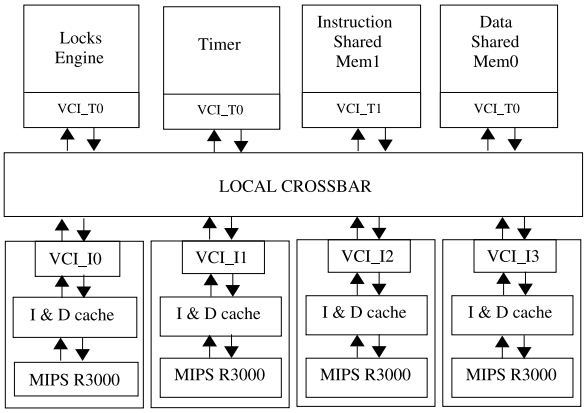


Figure 1.12: *Exemple d'architecture MPSoC*

Comme nous l'avons souligné, la description d'une architecture au niveau CABA permet d'obtenir des estimations de performances précises. Comme nous le verrons dans les chapitres suivants, les résultats d'estimations obtenus à ce niveau permettront la conception d'outils en vue d'une exploration rapide des alternatives architecturales. La figure 1.13 montre un exemple d'exploration d'architecture au niveau CABA. Dans cette figure, la taille du cache de données et d'instructions varie de 1Ko à 32Ko alors que le nombre de processeurs varie de 4 à 16. Les résultats sont rapportés pour l'application codeur H.263 qui sera détaillée dans le chapitre suivant. L'objectif ici est de monter qu'il est possible grâce à cet environnement de trouver la solution optimale en terme de performance. Ces résultats d'exploration seront discutés dans le chapitre **??**. Malgré la précision de ces estimations, le problème majeur du niveau CABA est le temps de simulation nécessaire pour obtenir ces résultats. Le tableau 1.4 donne un ordre de grandeur sur les temps de simulation à ce niveau pour l'application codeur H.263 exécutée pour une séquence vidéo de type QCIF en utilisant une machine Pentium M (1.6 GHz). La taille des caches des processeurs est fixée à 4 Ko. L'évaluation du système nécessite ainsi plusieurs heures de simulation.

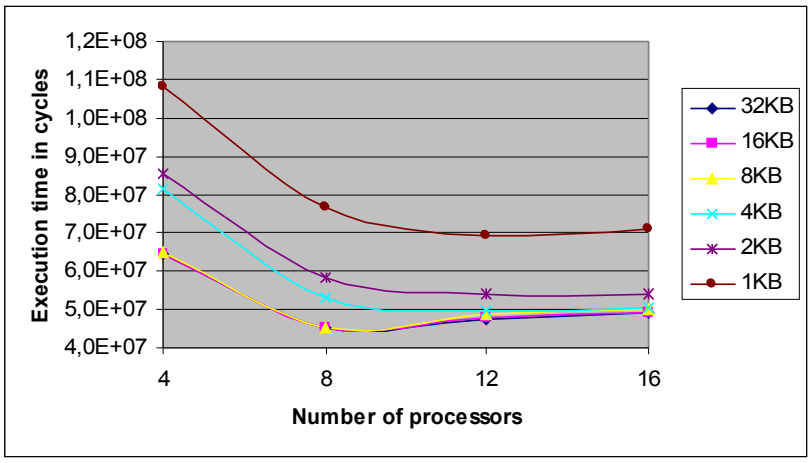


Figure 1.13: *Estimation de performance au niveau CABA pour l'application codeur H.263*

Nombre de processeurs	4	8	12	16
Temps de simulation	2h 37min	2h 57min	3h 52min	5h

Table 1.4: *Temps de simulation en fonction du nombre de processeurs pour l'application H.263*

Des expérimentations sur d'autres simulateurs MPSoC au niveau CABA comme MPARM [3] ont donné des temps de simulation comparables. Cet inconvénient s'accentue avec l'augmentation du nombre de composants dans le système MPSoC à simuler et plus spécifiquement le nombre de processeurs. La figure 1.14 donne le temps de simulation, en (*Instructions simulées par sec*) pour l'application H.263 avec un nombre croissant de processeurs. La figure 1.14 montre que le temps de simulation est inversement proportionnel au nombre de processeurs. En effet, en augmentant le nombre de processeurs, la vitesse de simulation chute du fait d'un nombre croissant de changements de contexte de l'ordonnanceur SystemC d'une part et des opérations de communication entre les différents processus qui composent la plateforme d'autre part.

1.5 Intégration dans Gaspard

Les composants matériels décrits au niveau CABA de diverses bibliothèques peuvent être intégrés dans notre environnement de conception des systèmes MPSoC Gaspard. Comme nous l'avons introduit au chapitre précédent, cet environnement vise la simulation du système à différents niveaux d'abstraction en particulier le niveau CABA. La simulation à ce niveau est obtenue à partir d'une modélisation de haut niveau et différentes transformations. La figure 1.15 détaille les phases de transformation appliquées sur la partie matérielle. Dans notre description nous allons insister sur le modèle d'architecture, le chapitre **??** fera le point sur les modèles d'application et d'association.

Notre flot de conception commence par une étape de modélisation de haut niveau du système MPSoC. Afin de profiter d'une description visuelle, nous utilisons le langage UML pour modéliser le système, il sera détaillé dans le chapitre **??**. Cette modélisation décrit

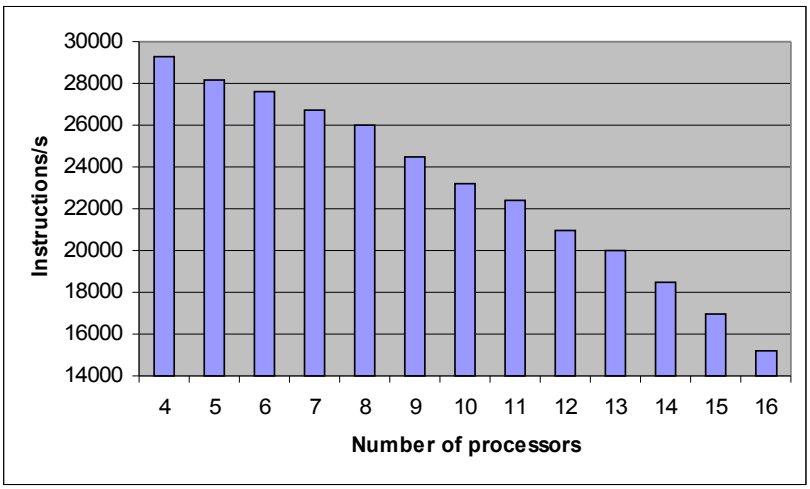


Figure 1.14: *Performance du simulateur SystemC*

les composants de l'architecture (nombre de processeur, réseau d'interconnexion, etc.) sans se référer aux composants réels. En effet, à cette étape de conception, la description doit être assez générale, puisque plusieurs niveaux d'abstraction peuvent être ciblés. Le choix d'un composant particulier de l'architecture dépend de la disponibilité de ce composant au niveau d'abstraction ciblé. De ce fait, le concepteur a besoin d'une deuxième étape de déploiement qui associe à chaque composant du modèle de haut niveau un composant réel existant dans la bibliothèque. Ceci nous permet d'obtenir un modèle d'architecture déployé. Ce dernier peut intégrer plusieurs composants décrits au niveau CABA possédant des interfaces de communication différentes. Dans ce cas, il est nécessaire de réaliser une adaptation entre les interfaces. Pour ce faire, deux méthodes sont possibles. La première est d'utiliser des adaptateurs qui sont déjà développés et mis à disposition avec la bibliothèque de composant.. La seconde approche est d'utiliser des outils de génération automatique d'adaptateurs à partir de la description de chaque interface de communication. SPIRIT [22] est un exemple de standard de description.

Cette phase de transformation nous permet d'obtenir un modèle d'architecture homogène au niveau CABA. A partir de ce modèle, nous appliquons une dernière transformation qui consiste à générer le code du système à simuler. Ce code comporte l'instanciation des composants à partir des bibliothèques avec les paramètres spécifiés par le concepteur et la connexion entre les interfaces. L'exécution du code généré va nous permettre la vérification fonctionnelle de notre système MPSoC et la mesure des performances correspondantes. Le chapitre **??** présentera une méthodologie de conception dirigée par les modèles afin de réaliser l'ensemble de transformations jusqu'a la génération de code.

1.6 Conclusion

Ce chapitre a montré que les méthodes classiques pour simuler et évaluer les performances des systèmes MPSoC à des niveaux bas sont fastidieuses, en particulier pour des architectures embarquées complexes. Notre étude a été réalisée en utilisant la bibliothèque de composants SoCLib que nous avons enrichie pour rendre possible la simulation des systèmes

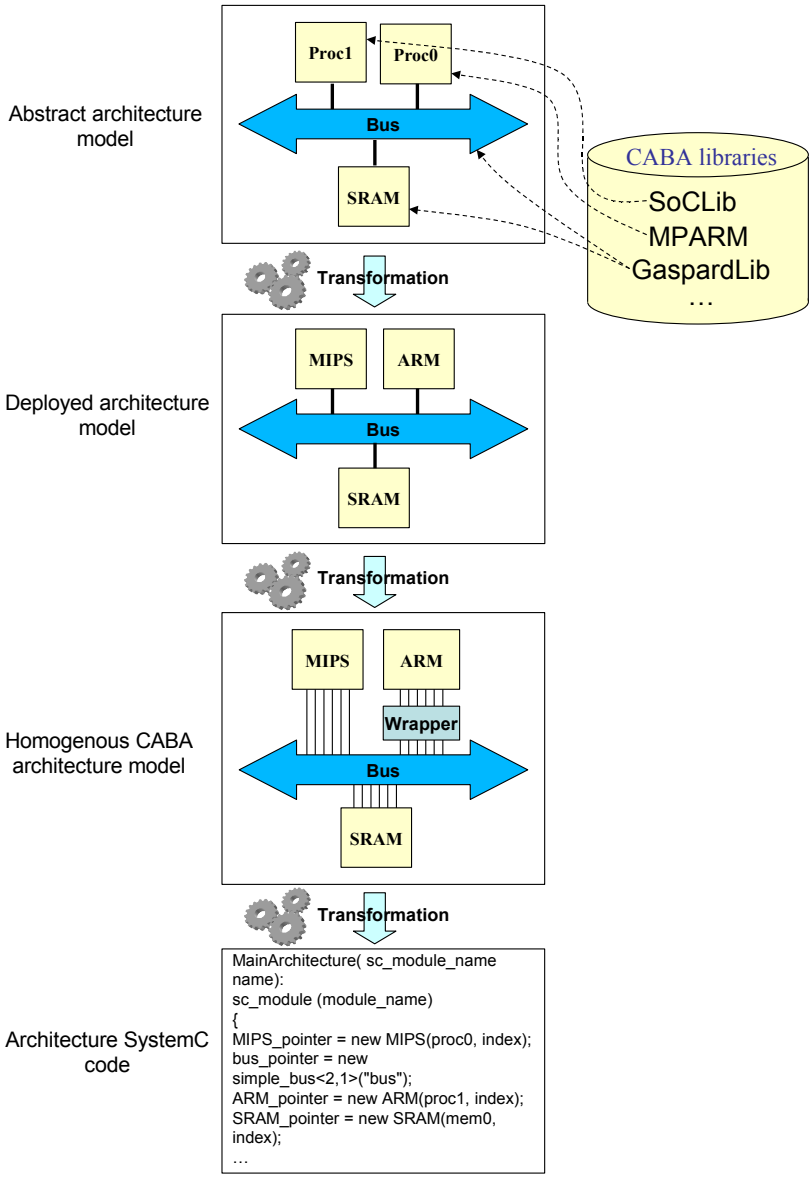


Figure 1.15: *CABA*

MPSoC hétérogènes. A travers cette étude, nous avons montré que l'effort de développement nécessaire au niveau CABA est considérable pour obtenir des estimations précises. Mais surtout, les temps de simulation obtenus à ce niveau ne sont pas suffisamment réduits pour permettre une exploration architecturale rapide des systèmes MPSoC.

Dans notre travail, nous nous sommes limités à des systèmes MPSoC à mémoires partagées et des architectures de processeurs simples tels que le processeur MIPS R3000. Le problème des temps de simulation s'accentue avec l'utilisation de processeurs plus complexes tels que les architectures superscalaires ou VLIW (*Very Large Instruction Word*) ou avec des architectures plus complexes telles que les architectures à mémoires distribuées.

Dans le chapitre suivant, nous allons présenter notre approche pour réduire le temps de simulation des systèmes MPSoC. Celle-ci permettra l'estimation des performances dans de délais relativement courts et pourra être intégrée dans un processus d'exploration de l'espace de solutions.

Bibliography

[1] Altera Startix II Architecture. http://www.altera.com/products/devices/stratix2/st2-index.jsp.

[2] AMBA protocol. http://www.arm.com/products/solutions/AMBAHomePage.html.

[3] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri. MPARM: Exploring the multi-processor SoC design space with systemc. *Springer J. of VLSI Signal Processing*, 41(2):169–182, July 2005.

[4] T. Bjerregaard and S. Mahadevan. A survey of research and practices of network-on-chip. *ACM Comput. Surv.*, 38(1), 2006.

[5] L. Bonde. *Transformations de ModÃÍles et InteropÃl'rabilitÃl' dans la Conception de SystÃÍmes HÃl'tÃl'rogÃÍnes sur Puce Ãă Base d'IP*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, UniversitÃl' de Lille 1, 2006.

[6] A. Bunker and G. Gopalakrishnan. Formal specification of the virtual component interface standard in the unified modeling language. http://www.cs.utah.edu/research/techreports/2001/pdf/UUCS-01-007.pdf.

[7] Coreconnect protocol. www.ibm.com/chips/products/coreconnect.

[8] A. Fraboulet, T. Risset, and A. Scherrer. Cycle accurate simulation model generation for SoC prototyping. In *SAMOS IV System Modeling, and Simulation*, 2004.

[9] A. Hekmatpour and K. Goodnow. DesignCon east 2005 : Standards-compliant IP design advantages, problems, and future directions. http://www.iec.org/events/2005/designcon_east/pdf/1-tp1_hekmatpour.pdf.

[10] K. Kim and J.-S. Koh. An area efficient DCT architecture for MPEG-2 video encoder. *IEEE Transactions on Consumer Electronics*, 45(1):62–67, February 1999.

[11] L. Lavagno, A. Sangiovanni-Vincentelli, and E. Sentovich. *System-level synthesis*, chapter Models of computation for embedded system design, pages 45 – 102. Kluwer Academic Publishers, 1999.

[12] H. Lim, V. Piuri, and E. Swartzlander. A serial-parallel architecture for two-dimensional discrete cosine and inverse discrete cosine transforms. *IEEE Transactions on Computers*, 49(12):1297–1309, December 2000.

[13] C. Loeffler, A. Lieenberg, , and G. Moschytz. Practical fast 1-D DCT algorithms with 11 multiplications. In *International Conference on Acoustics, Speech, and Signal Processing*, 1989.

[14] Logiciel Quartus II dâĂŹAltera. `http://www.altera.com/products/software/products/quartus2/qts-index.html`.

[15] S. Meftali. *Exploration d'architectures et allocation/affectation mémoire dans les systèmes multiprocesseurs monopuce*. PhD thesis, Laboratoire Techniques de l'Informatique et de la Microélectronique pour l'Architecture des ordinateurs, Université de Joseph Fourier, 2002.

[16] I. M. Panades, A. Greiner, and A. Sheibanyrad. A low cost network-on-chip with guaranteed service well suited to the GALS approach. In *NanoNet*, 2009.

[17] P. R. Panda. *Designing Embedded Processors, A Low Power Perspective*, chapter Power Optimization Strategies Targeting the Memory Subsystem, pages 131 – 155. Springer, 2007.

[18] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/Software Interface, 3rd Edition*. Morgan Kaufmann, 2006.

[19] F. Petrot, A. Greiner, and P. Gomez. On cache coherency and memory consistency issues in noc based shared memory multiprocessor soc architectures. In *DSD '06: Proceedings of the 9th EUROMICRO Conference on Digital System Design*, pages 53–60, 2006.

[20] R. E. C. Porto and L. V. Agostini. Project space exploration on the 2-D DCT architecture of a JPEG compressor directed to FPGA implementation. In *Design, Automation and Test in Europe Conference and Exhibition*, 2004.

[21] M. Sgroi, L. Lavagno, and A. Sangiovanni-Vincentelli. Formal models for embedded system design. *IEEE Design and Test of Computers*, 17(2):14–27, June 2000.

[22] SPIRIT consortium. `www.spiritconsortium.org`.