# THÈSE DE DOCTORAT DE L'UNIVERSITÉ PARIS VI

SPÉCIALITÉ INFORMATIQUE

Présentée par

**Denis HOMMAIS**

Pour obtenir le titre de

DOCTEUR DE L'UNIVERSITÉ PARIS VI

Sujet

## UNE MÉTHODE D'ÉVALUATION ET DE SYNTHÈSE DES COMMUNICATIONS DANS LES SYSTÈMES INTÉGRÉS MATÉRIEL–LOGICIEL.

Soutenue le 13 septembre 2001

Devant le jury composé de

| | | |
|---|---|---|
| M. Frédéric | PÉTROT | directeur de thèse |
| M. Jean-Paul | CALVEZ | rapporteur |
| M. Ahmed | JERRAYA | rapporteur |
| M. Alain | GREINER | examinateur |
| M. Jean-Yves | BRUNEL | examinateur |
| M. Michel | ROBERT | examinateur |

# Résumé

Cette thèse propose une méthode de synthèse et d'évaluation des communications dans les sytèmes intégrés matériel–logiciel. Le point de départ de la synthèse des communications est une description parallèle de l'application que le système doit réaliser sous la forme d'un graphe de tâches et l'affection de ces tâches à la partie matérielle ou à la partie logicielle.

Notre méthode de synthèse des communications est basée sur l'instanciation d'un schéma de communication pour chaque canal du graphe. Ces schémas décrivent les moyens utilisés pour transmettre les données dans le canal et synchroniser l'état du canal en fonction de la nature matérielle ou logicielle de la tâche productrice et de la tâche consommatrice.

De ces schémas, nous déduisons les spécifications d'un module matériel générique permettant de réaliser la partie matérielle des communications. Nous proposons une architecture de ce module utilisant la norme VCI pour lui permettre de s'adapter aux différents média de communication des systèmes intégrés. La réalisation d'un schéma de communication consiste alors pour la partie matérielle à instancier le ou les modules d'interface nécessaires, et pour la partie logicielle à remplacer les primitives utilisées dans la description parallèle par celle correspondant au schéma. L'application de notre méthode à la réalisation d'une application démontre qu'il est possible, à partir de ces schémas, de réaliser automatiquement la synthèse des communications.

Pour l'évaluation des systèmes réalisés, les différents modules sont modélisés par des machines à états. Nous utilisons les proprités de ces machines pour calculer un ordre statique d'évaluation et pour définir un algorithme simple de simulation du système au cycle près.

Cette méthode de simulation a été mis en œuvre dans le logiciel CASS. Les performances de simulation d'un système par CASS, comparé à un simulateur industriel de même niveau, montrent la pertinence de notre algorithme.



# Mots clés

Systèmes intégrés – Systèmes embarqués – synthèse des communications

simulation de systèmes – codesign matériel–logiciel

# Abstract

This thesis proposes a method for synthesis and evaluation of communications in hardware–software systems on a chip. The starting point of communication synthesis is a task graph describing the application that the target system should achieve as well as an assignment of the various tasks of the graph to the software and hardware components of the system.

The method for communication synthesis proposed here is based on the instanciation of a communication scheme for each channel of the graph. These communication schemes describe the means used to transfer data through the channel and to manage the channel status depending on the emitting and receiving tasks implementation, i.e hardware or software.

These schemes allow to draw the specifications of a generic hardware interface module to implement the hardware part of the communications. We propose an architecture of this module offering a VCI interface so that it can be adapted to various on chip interconnect. The implementation of a communication scheme consists then in instanciating one or more interface modules for hardware communications, and for software communications, in substituting the primitives used in the parallel description by those corresponding to the scheme.

This method has been used in the design of an application and demonstrates that automation of communication synthesis is feasable using predefined communication schemes..

In order to evaluate systems designed using the synthesis method proposed, components of an embedded system are modeled using communicating finite state machines. Advantage is taken from properties of state machines to determine statically an evaluation order and to use a simple and cycle-accurate simulation algorithm for system evaluation.

This simulation technique has been implemented in CASS, the Cycle Accurate System Siumlation tool. Significant speedup obtained compared to existant simulators demonstrates the relevancy of the simulation method chosen.



# Keywords

Systems on chip – embedded systems – communication synthesis

systems simulation – harware-software codesign

# Remerciements

Je tiens d'abord à remcercier Monsieur Frédéric Pétrot pour son travail, ses conseils, ses corrections et ces heures de *debug* pendant ces années où il a été mon directeur de thèse.

Je remercie Monsieur Alain Greiner, directeur de l'équipe ASIM du LIP6, pour m'avoir accueilli au sein de l'équipe et pour son intérêt pour mes travaux.

Je remercie également Monsieur Jean-Paul Calvez, professeur à l'école polytechnique de l'Université de Nantes et Monsieur Ahmed Jerraya, directeur de recherche au CNRS, de m'avoir fait l'honneur d'être les rapporteurs de mes travaux. J'y associe Monsieur Michel Robert, professeur au LIRMM à l'université de Montpellier, qui a accepté de faire parti du jury de thèse.

Je remercie Monsieur Jean-Yves Brunel, de Cadence design systems, d'avoir accepter de faire partie de mon jury de thèse et, associé à Monsieur Wido Kruijtzer de Philips Research, pour la collaboration fructueuse que fut pour moi le projet COSY.

Salut à tous les docteurs ou doctorants du laboratoire avec qui j'ai passé de bons moments et qui sont mes amis : Amal Zerrouki, Laurent Vuillemin, Julien Dunoyer, Nizar Abdallah, Vuong Huu Nghia, Gilles-Eric Descamps, Cyril Spasevski. Salut aussi à tous les membres de l'équipe ASIM et en particulier à Madame Anne Derieux.

Merci enfin à mes parents et à mes frères pour leur soutien pendant mes longues études.

# Table des matières

# Table des figures

# Liste des tableaux

# Chapitre 1

# Introduction

La technologie permettant l'intégration sur silicium des circuits numériques n'a cessé d'évoluer. Dans le même temps, presque toutes les étapes de la conception des circuits intégrés ont été automatisées : en particulier, l'utilisation de bibliothèques de cellules pré-caractérisées, la modélisation des circuits à l'aide de langages évolués et la synthèse. Ceci permet une conception rapide des circuits à partir de leur modèle en langage de haut-niveau.

Aujourd'hui, la densité d'intégration est telle que des systèmes complets peuvent être mis sur une seule puce. Un système intégré, tel que nous le définissons, est conçu pour une application précise (télévision numérique, gps, . . . ). Il contient généralement une partie matérielle et une partie logicielle. Un ou plusieurs processeurs intégrés exécutent la partie logicielle. La partie matérielle est, elle, composée de composants spécifiques à l'application. Ces composants exécutent une partie de l'application beaucoup plus rapidement et en consommant moins d'énergie que si elle était exécutée par le processeur.

Les méthodes de conception des circuits intégrés ne sont pas adaptées aux systèmes intégrés : elles ne visent, en effet, que la réalisation de la partie matérielle.

Des méthodes de conception des systèmes intégrés ont été proposées. Ces méthodes utilisent les techniques de conception des circuits intégrés pour la partie matérielle, mais elles la précèdent d'étapes visant, à partir de l'application et de ses contraintes, à déterminer ce qui doit être fait en matériel et ce qui doit être fait en logiciel, à concevoir l'architecture de la partie matérielle et à permettre la réalisation des communications entre la partie matérielle et la partie logicielle.

L'objectif de cette thèse est d'étudier le problème de la synthèse des communications et de leur évaluation pour la conception des systèmes mixtes matériel–logiciel. La technique de synthèse doit permettre aisément l'exploration des solutions. L'évaluation doit permettre de comparer rapidement ces solutions, d'en vérifier la viabilité, mais aussi de permettre la mise au point des composants matériels et logiciels utilisés par les communications.

Ce document est organisé de la façon suivante :

Le chapitre 2 présente la problématique de cette thèse. Il se base sur un flot de conception adapté aux systèmes embarqués. Nous cherchons à traiter deux sous-problèmes : la synthèse des communications et l'évaluation du coût des communications. Nous situons la synthèse des communications dans le flot de conception pour déterminer les hypothèses préalables à son déroulement. Nous déterminons le niveau de modélisation nécessaire à une évaluation précise du coût des communications.

Le chapitre 3 présente un état de l'art sur la synthèse des communications en précisant pour chaque méthode les hypothèses sur lesquelles elle repose. Ces hypothèses portent sur l'architecture du système cible et les spécifications de l'application, en particulier les sémantiques liées aux primitives de communication.

Le chapitre 4 formalise notre approche de la synthèse des communications. À partir du modèle fonctionnel des communications, nous déduisons un ensemble d'implémentations optimisées qui font intervenir du logiciel et du matériel. Ceci nous permet d'établir la spécification des ressources matérielles nécessaires pour supporter ces différentes implémentations.

Le chapitre 5 décrit l'architecture d'un module d'interface générique qui est la partie matérielle de la synthèse des communications. Pour cela, nous décrivons d'abord l'architecture des systèmes que nous visons, ainsi que les protocoles utilisés. Nous détaillons ensuite les services nécessaires aux communications fournis par ce module générique.

À partir des contraintes nécessaires à l'évaluation du coût des communications des systèmes intégrés, le chapitre 6 présente la modélisation en langage C et l'algorithme de simulation que nous avons défini. Il présente aussi la méthode utilisée pour intégrer des composants décrits comme des tâches logicielles.

Le chapitre 7 présente les performances obtenues par le simulateur, le compare avec un simulateur industriel ayant le même niveau de description. Il montre par ailleurs la faisabilité de la synthèse des communications sur l'exemple d'un système permettant de décoder des flux d'images JPEG où l'on a utilisé différentes architectures cibles et différents types de communications.

# Chapitre 2

# Problématique

Le contexte de notre travail est la conception des systèmes numériques embarqués. Un système embarqué est conçu pour répondre à un besoin particulier, c'est-à-dire pour réaliser une application précise dans un domaine particulier (TV numérique, télécommunication, identification, musique, automobile, spatial, ...). Il serait d'ailleurs plus juste de parler d'application embarquée. Cependant, il doit pouvoir s'adapter aux évolutions, gammes de produits, ou à l'utilisation de l'application dans des environnements différents.

La capacité d'intégration sur silicium actuelle permet sur une seule puce de créer des systèmes complexes qui étaient jusqu'à présent réalisés par une carte portant plusieurs puces. On parle alors de systèmes intégrés. Malheureusement, la méthode classique de conception des circuits intégrés à application spécifique (ASICs) n'est, par contre, pas adaptée.

Une méthode de conception adaptée à la conception de systèmes intégrés numériques doit avoir les caractéristiques suivantes :

- elle doit permettre la réutilisation systématique de composants logiciels ou matériels existants. Pour le matériel, cette réutilisation peut se faire à partir de plusieurs niveaux, de la description comportementale synthétisable (*soft core*) au dessin des masques (*hard core*). La méthode doit favoriser cette réutilisation, en permettant d'adapter les composants existants à chaque environnement spécifique ;
- elle doit s'appuyer sur des protocoles de communication normalisés entre composants comme celui défini par la norme VCI [VSI00]. Ces protocoles permettent de simplifier la

conception du système en fixant l'interface des composants matériels et facilitent leur réutilisation [GBA+99] ;
- la spécification de l'application ne doit pas préjuger des parties qui seront réalisées en matériel de celles qui le seront en logiciel. Un langage de spécification ne différenciant pas matériel et logiciel est donc nécessaire ;
- le flot de conception suppose l'existence d'un environnement de simulation mixte matériel/logiciel permettant une évaluation précise du système.

Pour concevoir de tels circuits, la méthode classique de conception de circuits doit être précédée d'étapes permettant l'évaluation de l'application et son partitionnement matériel–logiciel. En effet, un niveau de conception plus élevé considérant la conception de systèmes dans son ensemble est nécessaire. Des méthodes de conception des systèmes intégrés répondant à ces besoins ont été proposées [BSVKK98, Cal93]. Ces méthodes partent des spécifications de l'application pour obtenir d'une part le logiciel pour le ou les processeurs embarqués sous forme de langage de programmation ou de code compilé ; et d'autre part, la partie matérielle constituée des cœurs réutilisés et de la description synthétisable des accélérateurs matériels *ad-hoc*. Cette partie matérielle est le point d'entrée de la méthode de conception des circuits intégrés.

Dans la suite de ce chapitre, nous présentons les grandes lignes d'une méthode générale de conception de systèmes intégrés afin de mettre en avant les problèmes liés à la mise en œuvre et à l'évaluation des mécanismes des communications qui font l'objet de notre travail.

## 2.1 Méthode de conception des systèmes intégrés

La méthode que nous allons présenter est décrite par la figure 2.1.

### 2.1.1 Les spécifications

Sachant qu'un système intégré est conçu pour une application particulière, la spécification de ce système est donc l'application elle-même.

Deux niveaux de spécification peuvent cependant être distingués :
- la spécification fonctionnelle : elle décrit le comportement attendu de l'application ; ce

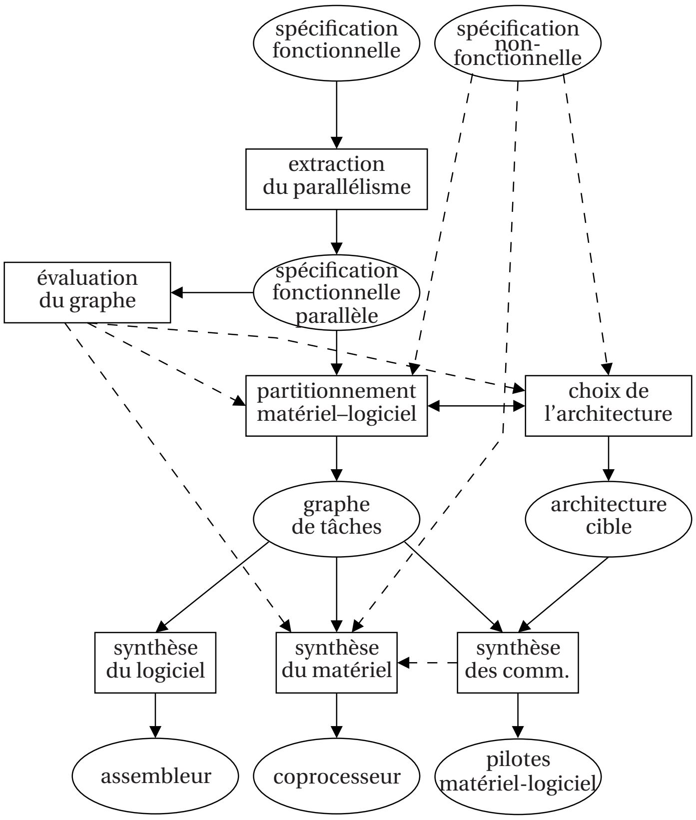



FIG. 2.1 – Flot de conception

que le système doit être capable de faire (jouer une séquence vidéo mpeg2, déclencher le gonflement d'un air bag, ...),

– les spécifications non-fonctionnelles : ce sont les contraintes externes que le système devra supporter (environnement, consommation, surface de silicium, débit),

La spécification fonctionnelle est généralement décrite dans un langage de programmation séquentiel standard, réalisant l'application sur une station de travail. Le comportement de ce programme constitue la référence pour la conception du système. Il permet entre autres de définir un ensemble de stimuli qui seront utilisés pour la validation des différentes étapes de la conception.

Les spécifications non-fonctionnelles sont des paramètres pris en compte à divers niveaux de la conception : un outil de synthèse de haut-niveau peut, par exemple, tenir compte du débit de données souhaité pour un coprocesseur ou de sa fréquence pour régler la profondeur des pipelines ou augmenter le nombre de ressources nécessaires à la réalisation de l'algorithme.

### 2.1.2 Un ensemble de tâches communicantes

L'étape suivante consiste à expliciter le parallélisme intrinsèque de l'application.

Dès que l'on veut partitionner l'application en parties matérielles et logicielles, la spécification fonctionnelle séquentielle ne peut plus être utilisée directement, car cette description séquentielle ne donne aucune indication sur le parallélisme à gros grain inhérent à l'application. Or, ce parallélisme gros grain est indispensable pour obtenir une accélération significative du système par l'ajout de coprocesseurs matériels.

#### 2.1.2.1 Loi d'Amdahl

La loi d'Amdahl [Amd67, HP90] rappelle que le gain en performances obtenu par l'utilisation d'un mode rapide d'exécution pour une partie d'une application est limité par la fraction de temps pendant laquelle ce mode est utilisé.

$$SpeedUp = \frac{1}{1 - Fraction_{enhanced} + \frac{Fraction_{enhanced}}{SpeedUp_{enhanced}}} \tag{2.1}$$

$SpeedUp_{enhanced}$ est le gain en performance de la partie de l'application passée en matériel. C'est le rapport entre le temps d'exécution de la partie accélérée avant et après l'accélération.

$Fraction_{enhanced}$ est le rapport du temps d'exécution de la partie à accélérer et du temps d'exécution total de l'application.

La loi d'Amdahl montre que le temps d'exécution maximum que l'on peut gagner est le temps d'utilisation de la partie que l'on a accélérée.

La figure 2.2 donne l'exemple d'une application comprenant quatre fonctions. Le temps d'exécution de l'application est la somme des temps d'exécution des quatre fonctions : $T_{app} = \sum_{i=1}^{4} T_i$

Si l'on souhaite accélérer la fonction 2 par un coprocesseur matériel (figure 2.3), l'accélération que l'on peut obtenir est donnée par l'équation (2.1), soit :

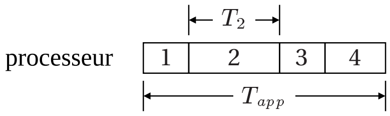



FIG. 2.2 – Une application séquentielle

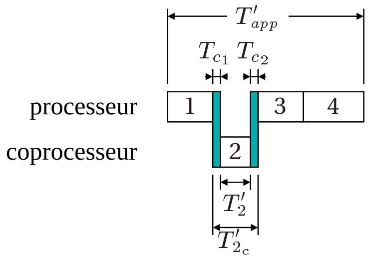



FIG. 2.3 – Accélération de la fonction 2

$$S = \frac{1}{1 - \frac{T_2}{T_{app}} + \frac{\frac{T_2}{T_{app}}}{\frac{T'_{2_c}}{T_2}}} \tag{2.2}$$

où $T_{c_1}$ et $T_{c_2}$ sont le temps nécessaire au communication.

Pour obtenir une amélioration significative des performances, il faut donc paralléliser les tâches. M. Edwards a noté qu'à cause de cela, l'utilisation de carte de prototypage rapide était généralement sans grand intérêt.

Si on exprime le parallélisme de cette application et les dépendances de données comme le montre la figure 2.4, on peut utiliser le parallélisme inhérent à l'utilisation de coprocesseurs matériels pour améliorer les performances.

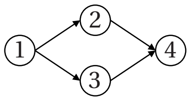



FIG. 2.4 – Graphe de dépendance des fonctions

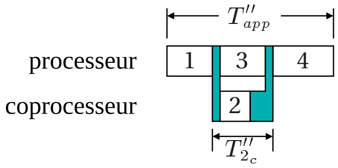



FIG. 2.5 – Utilisation de parallélisme

Dans le cas de notre exemple, le graphe de la figure 2.4 montre que l'on peut paralléliser la fonction 2 réalisée en matériel et la fonction 3 restée en logiciel. La figure 2.5 montre l'exécution de l'application obtenue. Pour cet exemple, on aura l'accélération :

$$S'' = \frac{1}{1 - \frac{T_2+T_3}{T_{app}} + \frac{\frac{T_2+T_3}{T_{app}}}{\frac{T''_{2_c}}{T_2+T_3}}} \tag{2.3}$$

Ceci montre que l'expression du parallélisme est nécessaire pour bénéficier pleinement de l'accélération apportée par l'existence de plusieurs processeurs et/ou par l'ajout de coprocesseurs matériels.

#### 2.1.2.2 Expression du parallélisme

Afin de mettre en évidence le parallélisme de l'application, il est assez usuel de décrire le système sous forme d'un ensemble de tâches communicantes [LS99, Cal93, dKES+00, HCM+99, Goe95].

Dans le cadre de notre travail, nous utilisons le formalisme KPN (*Kahn Process Network*) [Kah74, KM77].

Le modèle de Kahn est un graphe orienté (figure 2.6). Les nœuds sont les tâches séquentielles et les arcs les canaux de communication.

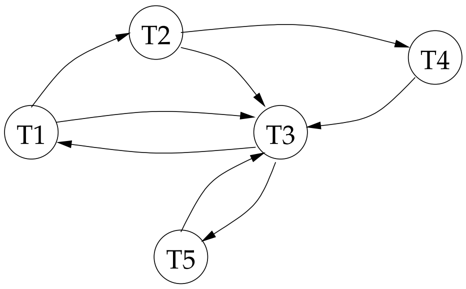



FIG. 2.6 – un graphe de tâches communicantes

Dans ce modèle théorique, chaque tâche exécute un programme séquentiel et communique en utilisant des canaux unidirectionnels de type FIFO de taille infinie. Chaque fifo possède une unique tâche productrice et une unique tâche consommatrice. Une tâche peut lire dans ses canaux d'entrée et écrire dans ses canaux de sortie, mais ne peut pas vérifier la présence de données dans un canal d'entrée. Si une tâche essaye de lire dans un canal vide, elle est bloquée jusqu'à ce que le canal ne le soit plus. Les fifos étant de profondeur infinies, les écritures ne sont jamais bloquantes. Kahn démontre que ces contraintes (producteur et consommateur unique, tâches séquentielles et lecture bloquante) font que les tâches sont monotones, ce qui rend le graphe déterministe. Un tel graphe garantit que l'ordre des données est toujours le même et ne dépend pas de l'ordre d'évaluation des tâches. Bien évidemment, la profondeur infinie des fifos n'est pas réaliste pour une réalisation matérielle. Ajouter une taille de mémorisation limitée aux canaux dans le modèle de Kahn implique qu'une tâche peut aussi être bloquée si elle tente d'écrire dans un canal plein. Un résultat intéressant du à Parks [Par95] est que le graphe est alors toujours déterministe mais que des inter-blocages peuvent intervenir. Des techniques ont été proposées pour calculer la profondeur minimum (c'est-à-dire optimale) des tampons nécessaires dans chaque canal [KLdM93]. Ces techniques fonctionnent pour des applications de type flot de données, mais ne sont pas applicables aux communications qui sont fonction des données circulant dans les canaux.

Dans le modèle de Kahn, les tâches échangent par ces canaux des messages à l'aide de primitives connues (`read`, `write`). Les tâches ne sont synchronisées entre elles que par ces messages.

Ce graphe de tâches sera appelé spécification parallèle dans ce manuscrit (par opposition à la spécification fonctionnelle initiale qui est souvent un programme séquentiel). Pour construire ce graphe, il faut extraire le parallélisme à gros grains du système et détecter les dépendances de données ou encore, comme le propose Calvez, extraire les « variables fondamentales » du système pour en déduire les tâches [Cal93]. La granularité des tâches doit être suffisamment faible pour ne pas avoir à repartitionner les tâches lors de leur synthèse car les étapes suivantes de la méthode s'appuient sur ce partitionnement en supposant les tâches atomiques. Cependant, plusieurs tâches peuvent être exécutées par un même élément de traitement (processeur général ou coprocesseur matériel spécialisé).

Ce graphe ne représente que les spécifications fonctionnelles de l'application. Il ne modélise pas les contraintes externes du système, telles que les contraintes temporelles.

Cette description peut être faite dans différents langages. Il peut s'agir :

– soit d'un langage de programmation classique : T. Kuhn ou J. Fleischmann utilisent Java [KRK99, FBK99] ;
– soit d'un langage spécifique ou un langage classique enrichi : D. Gajski utilise SpeC+ [GDWL92], A. Wenban utilise Promela [WO93], L. Lavagno utilise Esterel/C (ECL) [LS99] ;
– soit un langage de description de matériel qui exprime directement le parallélisme (VHDL, Verilog, ...).

### 2.1.3 Validation et évaluation quantitative du graphe de tâches

La simulation du graphe de tâches est une étape importante du flot de conception. Elle permet d'évaluer la complexité de chaque tâche pour en déduire un temps de calcul suivant qu'elle est réalisée en matériel ou en logiciel. La simulation du graphe permet aussi de mesurer les communications entre les tâches en terme de volumes de données transférées sur chaque canal. Ceci permet de déterminer le coût en communication impliqué par un changement de nature d'une tâche de logiciel en matériel ou inversement en fonction de la nature des tâches avec lesquelles elle communique. La spécification parallèle doit donc être exécutable.

### 2.1.4 Choix de l'architecture cible

Le choix de l'architecture cible consiste à déterminer le nombre et le type de processeurs que le système doit contenir, le nombre et le type de coprocesseurs implémentant les parties matérielles ainsi que l'organisation et le choix des média de communications.

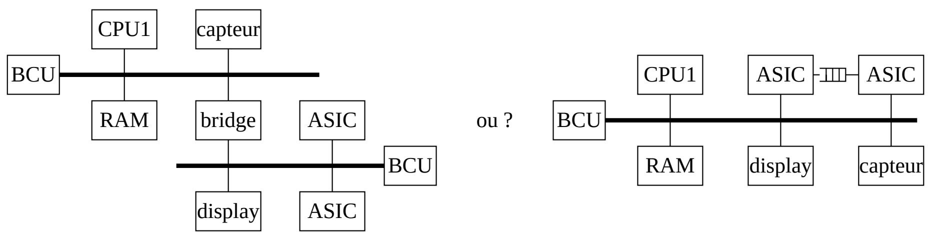



FIG. 2.7 – Choix de l'architecture du système

Le choix de l'architecture et le partitionnement ne peuvent pas être fait séparément [KM98]. Il existe plusieurs systèmes réalisant la même fonctionnalité en respectant les mêmes contraintes. Le choix du bus de communications par exemple dépend du débit qui est nécessaire aux différents éléments pour communiquer et des tâches affectées à ces éléments. À l'inverse, suivant le débit du bus choisi, l'affectation de tâches aux éléments du système sera ou non réalisable.

### 2.1.5 L'affectation des tâches

Après définition de l'architecture cible qui spécifie le nombre et la nature des processeurs et des coprocesseurs, ainsi que les moyens de communication dont ils disposent, il s'agit d'affecter les tâches aux différents éléments de traitement. C'est à dire, de définir pour chaque tâche si elle sera réalisée en matériel ou en logiciel et par quel processeur ou coprocesseur du système elle sera réalisée. Les choix de cette étape sont faits par le concepteur en fonction des spécifications non-fonctionnelles de l'application, des composants disponibles et d'une évaluation précise du graphe des tâches pour obtenir la complexité de chaque tâche et le volume de données échangées entre les tâches.

Plusieurs tâches peuvent être regroupées dans un même élément de traitement. De même, dans le cas d'un coprocesseur la réutilisation du même matériel pour effectuer plusieurs tâches est une question de compromis entre parallélisme et surface du système résultant.

La validité des choix effectués à cette étape ne peut être vérifiée qu'après l'étape suivante qui synthétise les communications entre les différentes tâches.

### 2.1.6 La synthèse des communications

Dans le graphe, les tâches échangent des messages par des canaux de communication représentés par les arcs. Une fois les tâches affectées aux ressources du système, il faut adapter ces échanges de données aux supports physiques effectivement disponibles pour la communication (bus, lignes d'interruptions, communications point-à-point comme une fifo matérielle dédiée entre deux coprocesseurs, mémoires, ...) et à la nature matérielle ou logicielle de chaque tâche.

Il faut remplacer les primitives (`read`, `write`) utilisées par les tâches pour communiquer par les méthodes matérielles ou logicielles du système qui sont le support effectif des communications. Les méthodes d'échange de données et de synchronisation qui peuvent être mises en œuvre sont nombreuses : mémoire partagée ou files de messages pour les données, interruptions ou scrutations pour la synchronisation.

### 2.1.7 La synthèse du logiciel

Les tâches logicielles doivent être compilées pour le processeur cible et exécutées par lui. Si plusieurs tâches sont affectées à un même processeur, soit ces tâches sont fusionnables soit elles peuvent être ordonnées statiquement, soit il faut utiliser un système d'exploitation multi-tâches [CKL+00].

### 2.1.8 La synthèse des coprocesseurs matériels

Une fois l'affectation des tâches effectuée et validée, il faut synthétiser les tâches matérielles. Ceci est fait par des outils de synthèse de haut niveau prenant en entrée :

- la spécification fonctionnelle de la tâche,
- les contraintes non fonctionnelles (débit, fréquence, consommation) soit provenant des spécifications de départ, soit issues de l'évaluation du système,
- les directives du concepteur [ABG+97].

## 2.2 Synthèse des communications

Le comportement du système est décrit par le graphe de tâches. Chacune des tâches du graphe peut être réalisée indifféremment en logiciel ou en matériel. Si une tâche est réalisée en logiciel, elle sera compilée pour être exécutée par un processeur du système. Si une tâche est réalisée en matériel, elle sera soit synthétisée en un coprocesseur, soit un coprocesseur existant sera utilisé pour la réaliser. Un système contenant des parties matérielles et des parties logicielles aura donc besoin d'interfaces pour adapter les communications entre ces différentes parties.

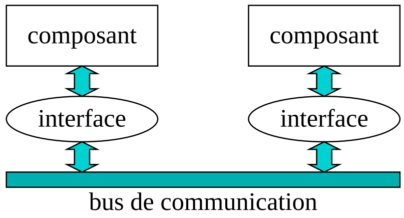



FIG. 2.8 – Interfaces permettant à deux composants de communiquer par un bus

Ces interfaces sont des méthodes matérielles et logicielles permettant de réaliser de façon transparente pour les tâches les primitives de communication qu'elles utilisent.

Pour cela, les interfaces utilisées doivent gérer les canaux de communication. Elles doivent permettre d'acheminer les données d'un bout à l'autre du canal, de gérer le tampon interne à chaque canal et permettre la synchronisation, c'est à dire rendre les primitives bloquantes.

L'utilisation de tampon permet à la lecture et à l'écriture d'être totalement désynchronisée, comme le veut la sémantique de Kahn : l'écriture se termine lorsque le canal a absorbé les données et non lorsque une tâche à lu les données avec la primitive de lecture. Les primitives ne sont bloquantes que lorsque le canal est plein pour la primitive d'écriture et vide pour la primitive de lecture.

Les canaux ont une capacité de stockage de type FIFO ; ce qui permet aux écritures, dans la mesure ou cette fifo n'est pas pleine, de ne pas être bloquantes même si la tâche consommatrice qui doit réaliser la lecture n'est pas prête. Du point de vue d'une tâche productrice, une écriture est réalisée au retour de la primitive `write`. Ceci correspond non pas à la lecture par la tâche consommatrice des données par la primitive `read` mais à la copie des données dans la fifo.

La communication, quelle que soit la forme qu'elle prendra est, et doit rester, transparente pour la tâche, car il n'est pas envisageable de modifier la spécification initiale pour pouvoir effectuer la synthèse des communications.

Que les tâches soient matérielles ou logicielles, la réalisation d'une communication contient deux parties distinctes : une partie matérielle et une partie logicielle.

La partie matérielle doit permettre aux éléments réalisant les tâches de communiquer avec le reste du système. Par exemple si la communication se fait par un bus, la partie matérielle implémente le protocole d'accès au bus.

La partie logicielle est, dans le cas d'une tâche réalisée en logiciel, la primitive de communication elle-même et l'utilisation de la partie matérielle pour accéder à la fifo du canal. Dans le cas d'une tâche réalisée par un coprocesseur matériel, il s'agit de l'utilisation d'un processeur pour réaliser la communication (programmation d'un DMA, génération d'une interruption, ...).

Pour réaliser ces deux parties, deux approches sont envisageables. La première consiste à faire appel aux outils de synthèse généraux lorsqu'on synthétise les coprocesseurs matériels. La deuxième solution consiste à utiliser une bibliothèque de protocoles de communication prédéfinis. Cette solution permet d'utiliser des protocoles standards pour la communication. Une bibliothèque peut être définie à différents niveaux :

– au niveau du protocole. La bibliothèque contient une description de différents protocoles que l'outil de synthèse utilise pour réaliser l'interface du composant.

– au niveau bloc. La bibliothèque contient alors des blocs paramétrables que l'outil de synthèse peut instancier pour réaliser la communication. Deux niveaux de paramétrisation sont possibles : lors de la synthèse et ces blocs sont alors des générateurs ou lors de l'exécution, ces blocs étant configurable logiciellement à l'initialisation du système.

## 2.3 Évaluation du système

Parallèlement au flot de synthèse du système, il faut à chaque étape non seulement pouvoir vérifier le comportement du système, mais également en évaluer les performances pour valider les choix du concepteur. En particulier, il est nécessaire de pouvoir le faire après trois des étapes du flot de conception que nous venons de décrire :

**Après la construction du graphe de tâches :** pour valider le comportement de la spécification parallèle par rapport à la référence que représente la description séquentielle de l'application. Le langage utilisé pour décrire les tâches est soit un langage de programmation, soit un langage spécifique. Dans le premier cas, il existe un compilateur permettant d'obtenir un exécutable qui sera exécuté sur une station de travail avec un système multi-tâches. Dans le second, une plate-forme de simulation ou de compilation dédiée doit être fournie [dKES$^+$00],

**Après la synthèse des communications :** pour valider le choix de partitionnement matériel–logiciel et évaluer les coûts de communication qui en résulte. Ceci permet de définir précisément les contraintes à fournir à l'outil de synthèse de haut-niveau chargé de la synthèse des coprocesseurs matériels.

**Après la synthèse des coprocesseurs matériels :** pour valider les protocoles de communication mis en œuvre et s'assurer que les contraintes de performances fournies à la synthèse de haut-niveau ont été respectées.

Notre travail se situe au niveau de la synthèse des communications. A ce niveau, la description du système est hétérogène. Nous disposons de tâches devant être réalisées en logiciel qui doivent être compilées pour le ou les processeurs embarqués et de tâches affectées au matériel qui deviendront, à terme, des coprocesseurs.

Pour simuler l'ensemble d'un système matériel–logiciel, deux solutions sont envisageables : une simulation « classique » ou une co-simulation.

Dans la simulation « classique », tous les composants matériels du système, y compris les processeurs, sont modélisés dans le langage d'entrée du simulateur. Les tâches logicielles sont chargées dans les mémoires du système et exécutées par les modèles des processeurs.

Dans la co-simulation, les parties matérielles et logicielles sont simulées séparément. La partie matérielle est simulée de façon classique ou sur un émulateur matériel. Plusieurs techniques existent pour exécuter la partie logicielle [Row94] :

– sans modèle du processeur : le logiciel est compilé sur la machine hôte et exécuté par celle-ci,
– avec un modèle du processeur : ce modèle peut être précis au cycle ou seulement émuler correctement le jeu d'instructions [ŽM96].

La communication entre le logiciel et le matériel se fait à travers des interfaces. Les simulateurs possèdent une interface permettant de communiquer avec d'autres application comme l'interface CLI de Synopsys [Syn98a]. La figure 2.9 donne un exemple. Le processeur relié au bus physique est remplacé par une interface transformant les accès faits par l'application en accès au bus. Dans l'application, les accès aux entrées/sorties ont été remplacés par une connexion à l'interface externe du simulateur.

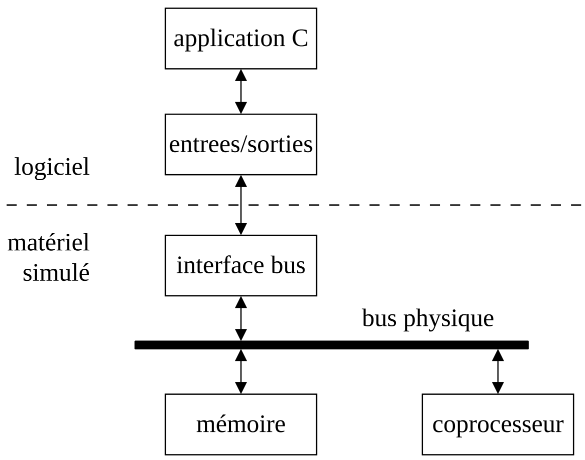



FIG. 2.9 – Co-Simulation

Le principal intérêt de la co-simulation est de pouvoir autoriser une validation fonctionnelle sans que le processeur embarqué ne soit décrit [Naç98]. Le défaut est que l'application compilée pour la station hôte n'a pas les mêmes performances temporelles que sur le processeur embarqué. Par exemple, une instruction qui dure un cycle sur la station hôte comme une division peut prendre plusieurs centaines de cycles sur le processeur embarqué qui ne possède pas l'instruction de division. Dans ce contexte, le temps de traitement d'une interruption, par exemple, ne peut être mesuré dans un tel environnement.

## 2.4 Conclusion

La synthèse des communications repose sur une bibliothèque définissant des schémas de communication associant les primitives bloquantes (`read`, `write`) utilisées par les tâches aux moyens matériels ou logiciels utilisés par les différents processeurs ou coprocesseurs. Les éléments de cette bibliothèque dépendent des primitives employées et de l'architecture choisie.

L'objectif de cette thèse est d'une part de proposer des éléments matériels et logiciels adaptés aux communications des systèmes embarqués et d'autre part de permettre l'évaluation des performances de l'application après cette synthèse pour pouvoir valider les choix architecturaux et les moyens matériels et logiciels mis en jeu.

Notre travail consistera donc à :

- définir l'interface matérielle d'un coprocesseur synthétisé à partir d'une spécification utilisant les primitives de communication des réseaux de Kahn,
- définir un module d'interface générique permettant de faire communiquer un coprocesseur avec le reste du système,
- définir les services que ce module devra fournir pour permettre ces communications,
- fournir un environnement de simulation permettant l'évaluation fiable des communications du système après leur synthèse.

# Chapitre 3

# État de l'art de la synthèse des communications

Dans le chapitre précédent, nous avons situé la synthèse des communications dans le flot de conception des systèmes embarqués. Nous avons vu que les communications dépendent fortement de la description initiale et de l'architecture du système choisi.

Pour décrire les différentes approches existantes pour la synthèse des communications, nous présentons pour chaque approche la description initiale de l'application utilisée et l'architecture cible puis le modèle de communication utilisé.

## 3.1 Kalavade

L'architecture visée par A. Kalavade [Kal95] est décrite par la figure 3.1. Dans cette architecture un seul processeur exécute la partie logicielle et plusieurs coprocesseurs implémentent la partie matérielle. Ces modules sont connectés à un unique bus.

Un coprocesseur matériel (figure 3.2) comprend un *kernel* contenant le contrôleur et le chemin de données du module. Le module communique avec le reste du système par deux interfaces : une en entrée et une en sortie.

Le *kernel* possède deux signaux de commande `ready` et `completion`. Le premier indique que

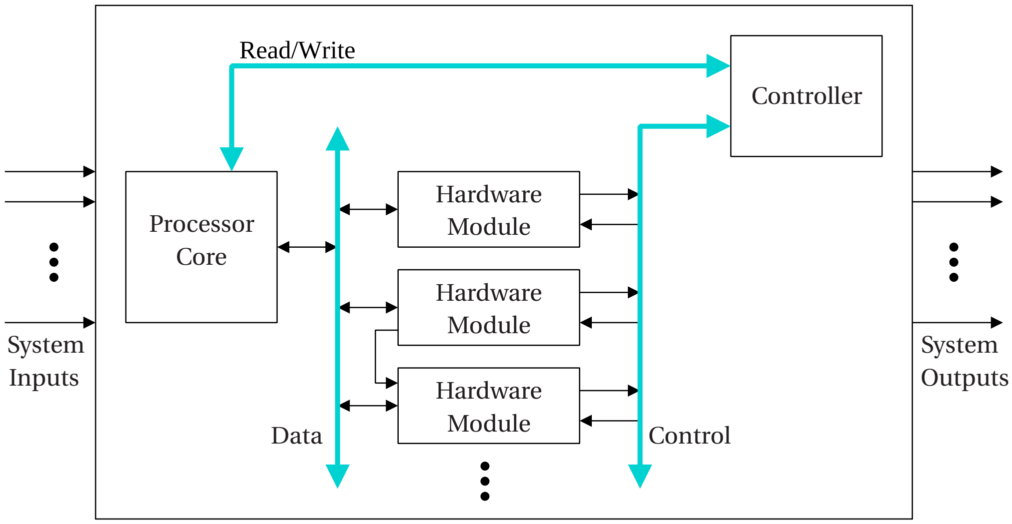



FIG. 3.1 – Architecture type de systèmes

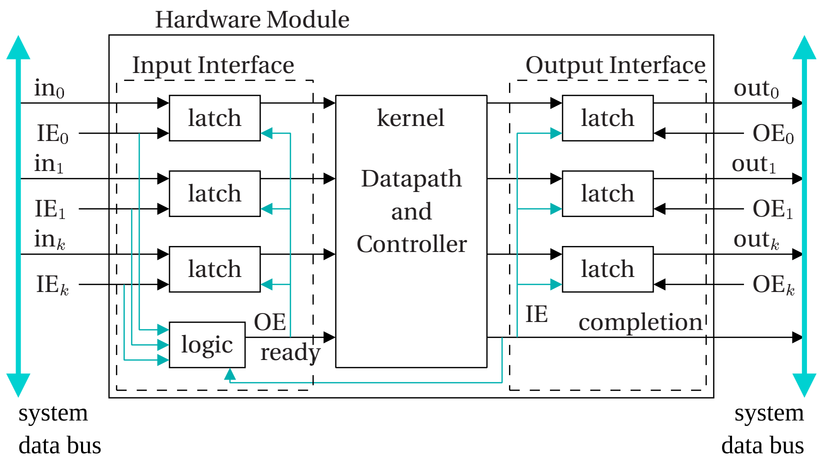



FIG. 3.2 – Architecture type de systèmes

toutes les données sont présentes dans les latchs de l'interface d'entrée. Le second indique que le *kernel* a terminé le calcul et que les résultats sont disponibles dans les latchs de l'interface de sortie. Chaque latch possède un *input enable* (IE) et un *output enable* (OE) pour contrôler son accès.

Toutes les entrées ou sorties de tous les modules matériels du système sont vues par le processeur comme une adresse unique.

Dans cette architecture, un contrôleur central câblé gère les communications entre le processeur et les modules matériels. Pour cela, il connaît *a priori* l'ordre dans lequel les modules matériels doivent être activés. Cet ordre est issu du partitionnement matériel–logiciel du système. Il ne peut donc pas être dépendante des données. À chaque requête du processeur, le contrôleur active un input ou un output enable d'un latch d'un module matériel suivant cet ordre.

Les communications matériel→matériel sont réalisées en connectant simplement les latchs du module producteur au latch du module consommateur. Le signal `completion` du module producteur autorise l'écriture des latchs du module consommateur.

Dans ce modèle, les communications matériel→logiciel sont de simples lectures ou écritures en mémoire pour le processeur. Lorsque le processeur fait une écriture en direction d'un module matériel, le contrôleur valide l'écriture dans le latch. Si tous les latchs du module matériel ont été écrits, le signal `ready` commande au *kernel* du module de lire les données pour commencer le traitement. Lorsque le processeur fait une lecture en direction d'un module matériel, le contrôleur vérifie que le signal `completion` est actif. S'il ne l'est pas, il bloque le processeur jusqu'à ce que `completion` soit valide. Dès qu'il le devient, le contrôleur sélectionne le latch pour que le processeur lise la donnée.

Cette architecture et les communications entre matériel et logiciel qui lui sont associées, sont assez efficaces : le transfert d'une donnée entre le logiciel et le matériel est un simple accès mémoire. Le remplacement du décodeur d'adresses classique par un contrôleur permet de s'affranchir complètement de la notion d'adresse dans le matériel. Cependant, le système matériel et logiciel résultant est figé. Il n'est pas possible de changer le logiciel, puisque l'ordre d'accès des données du matériel est codé dans le contrôleur. Or, l'un des principaux avantages de l'utilisation du logiciel est la souplesse qu'il procure.

## 3.2 Vulcan

### 3.2.1 Modélisation

Dans [GCM94] et [GM96], R.K. Gupta présente l'environnement Vulcan. Dans cet environnement, une application est décrite à l'aide de HardwareC. HardwareC est un langage de description de matériel dont la syntaxe est proche de celle du langage C, mais qui permet d'exprimer des contraintes au niveau des ressources utilisées et des contraintes temporelles.

Les contraintes temporelles sont exprimées par des délais qui sont appliqués aux opérations. Ces délais peuvent être dépendants des données ou ils peuvent dépendre de synchronisations extérieures.

Un langage de description de matériel n'est pas idéal pour décrire une application complète, en raison des limitations inhérentes au modèle matériel, mais il permet de s'appuyer sur des outils de synthèse existants pour l'implémentation de la partie matérielle.

Un système est décrit dans ce langage comme plusieurs processus communicants. Les communications intra-processus se font par mémoire partagée. Les communications inter-processus se font par passage de messages. Le mécanisme de passage de messages mis en œuvre est un simple *handshake* et les opérations d'émission et de réception sont simultanées.

Chaque processus est compilé en un graphe dit « flot de données » (*dataflow graphs*). Dans ce graphe, les sommets sont des opérations et les arcs représentent les dépendances de données entre les opérations. Le graphe d'un système est constitué de composantes *dataflow* concurrentes qui sont ordonnées par le *system control flow*.

Enfin, l'utilisateur peut associer des contraintes de débits aux entrées–sorties.

### 3.2.2 Architecture cible

L'architecture cible de système est décrite par la figure 3.3.

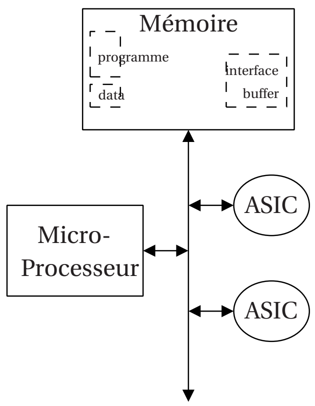



FIG. 3.3 – Architecture cible

Plusieurs restrictions ont été faites :

- le système ne comprend qu'un seul processeur. L'utilisation de plusieurs processeurs nécessiterait, selon les auteurs, un ajout de synchronisation et de protection de la mémoire,
- il n'y a qu'un seul bloc mémoire,
- le processeur est le seul maître sur le bus,
- le processeur dispose de suffisamment de ligne d'interruptions pour la réalisation de l'application. Ces lignes d'interruption sont non-vectorisées et il existe une adresse associée à chaque ligne d'interruption.

### 3.2.3 Partitionnement

Le graphe est partitionné en fonction de sa faisabilité et des contraintes de débit de données de l'application. Le résultat du partitionnement est un ensemble de modèles matériels et logiciels. Ces modèles sont représentés par des graphes hiérarchiques acycliques. Les opérations d'entrées–sorties sont réalisées par passage de messages. Les boucles dépendant des données sont considérées comme des opérations dont le délai n'est pas borné.

L'utilisation de la hiérarchie permet de transformer le problème d'exécution conditionnelle en délais incertain d'exécution. L'incertitude de la boucle est le nombre de fois où elle est exécutée.

En l'absence de point de synchronisation multiple, un graphe peut être implémenté comme une simple fonction. Un système hiérarchique sera implémenté comme un ensemble de fonctions où chaque graphe acyclique est une fonction. L'implémentation d'un graphe est une *thread* à cause de la sérialisation de l'exécution sur un processeur.

Une application est transformée en un ensemble de graphes. Chaque graphe devient une *thread*. Le parallélisme de l'exécution des *threads* est obtenu en entrelaçant l'exécution des *threads* sur le processeur.

### 3.2.4 Partie logicielle

La partie logicielle de l'application est découpée en *threads* de telle façon qu'une *thread* est définie comme un ensemble d'opérations linéarisées (*basic block*) Seule la première opération peut être une opération dont le délais n'est pas borné.

Un graphe de flot de données spécifie un ordre partiel pour l'exécution des opérations. Suivant les opérations dont le délais n'est pas borné, plusieurs *threads* peuvent être créées à partir d'un graphe. Les *threads* sont construites de façon à ce que cet ordre partiel soit respecté. Deux *threads* peuvent être soit concurrentes, soit il existe une relation de hiérarchie entre elles. Dans ce dernier cas les *threads* sont créées avec une opération d'autorisation leur permettant d'attendre que leur prédécesseur soit terminé.

Le modèle de logiciel utilisé repose sur l'exécution séquentielle de chaque *threads*. L'exécution concurrente est obtenue en entrelaçant l'exécution des *threads*. Plusieurs *threads* peuvent être créées à partir du graphe, chaque *thread* commençant par une opération dont le délai n'est pas connu. La synchronisation logicielle est nécessaire pour assurer l'ordre d'exécution des opérations d'une *thread* et entre les *threads*.

### 3.2.5 Synchronisation matériel–logiciel

A cause de l'exécution séquentielle du logiciel, un transfert de données entre matériel et logiciel doit être explicitement synchronisé. Pour tenir compte des différentes vitesses d'exécution des composants matériels et logiciels, et des opérations dont le délai n'est pas connu, un ordonnancement dynamique des *threads* est utilisé. Cet ordonnancement est basé sur la disponibilité des données. Il est obtenu par une fifo de contrôle. L'interface matérielle–logicielle consiste en une fifo de données par canal et une fifo de contrôle qui gère les identificateurs des *threads* contrôlant ainsi l'ordre des données. La taille de la fifo de contrôle dépend du nombre de *threads*. La fifo de contrôle est associée à de la logique de contrôle qui peut être implémenté en matériel ou en logiciel. Dans le cas d'une implémentation logicielle, aucune logique n'est nécessaire puisque la fifo de contrôle est déjà logicielle. Dans ce cas, les signaux rapportant la disponibilité des données dans les file de données sont connectés au processeur comme des lignes d'interruptions non-vectorisées et les routines de gestion d'interruptions correspondantes sont utilisées pour ajouter les identificateurs de *threads* dans la fifo de contrôle. Pendant la gestion de ces interruptions, les interruptions sont masquées pour préserver l'intégrité de la logique de contrôle.

### 3.2.6 Conclusion

Dans cette approche, la description du système permet d'expliciter le parallélisme. La génération de l'échéancier permet de bénéficier de ce parallélisme aussi bien pour le matériel que pour le logiciel. Cependant, cette génération ne permet pas l'utilisation de systèmes d'exploitation existants. L'interface entre matériel et logiciel, comme l'utilisation de lignes d'interruption non-vectorisées pour prévenir de la disponibilité des données, ne permet pas d'utiliser une architecture système générique. Et ces fifos de contrôle imposent des contraintes fortes sur l'échéancier. De plus, le fait d'imposer qu'il n'y ait qu'un processeur et qu'il soit le seul maître sur le bus limite grandement les débits et charge le processeur puisqu'il doit réaliser les communications matériel–matériel.

## 3.3 IPCHINOOK

IPCHINOOK est un outil de synthèse pour systèmes embarqués distribués [COH+99]. Il est orienté vers la réutilisation de composants.

L'entrée de IPCHINOOK est une description comportementale, une description de l'architecture cible et une fonction d'allocation qui définit les relations entre les deux descriptions. La description comportementale contient plusieurs modules concurrents et communicants appelés *modal processes*. La description de l'architecture cible décrit les processeurs, les entrées/sorties, les bus de communication et la topologie du système cible. La fonction d'allocation décrit l'affectation des *modal processes* aux processeurs du système. Cette description structurée en trois parties permet de changer une partie indépendamment des deux autres.

### 3.3.1 Description comportementale

#### 3.3.1.1 *modal processes*

Un *modal process* comprend des *ports*, des *handlers* et des *modes*. Les *ports* d'un *modal process* lui permettent de communiquer avec les autres processus. Un *port* est connecté à un canal. Un canal est connecté à un *port* de sortie et à un ou plusieurs *ports* d'entrée.

Le comportement d'un *modal process* est spécifié par ses *handlers*. Un message arrivant sur un *port* d'entrée est un événement qui est géré par l'appel d'un *handler*. Un *handler* encapsule le code de l'application, il envoie les messages sur les *ports* de sortie et renvoie des requêtes de changement de *mode*.

Les *modes* sont des drapeaux permettant d'activer ou de désactiver les *handlers*. Un *mode* associe un *handler* à un *port*. Lorsqu'un *mode* est actif, si un message est reçu sur le *port* $p$ et qu'il existe une association entre $p$ et le *handler* $h$, alors $h$ sera appelé. Plusieurs *handlers* peuvent être associés à un même *port*. Comme plusieurs *modes* peuvent être actifs en même temps, plusieurs *handlers* peuvent être appelés par message. Dans ce cas, les *handlers* sont appelés dans un ordre statiquement défini.

L'exécution des *handlers* d'un *modal process* est divisé en étapes discrètes. À chaque étape, les messages présents sur les *ports* d'entrées sont envoyés aux *handlers* qui sont actifs en fonction des *modes*. Lors de son exécution, un *handler* peut envoyer un ou plusieurs messages sur ces *ports* de sortie, mais il y a au moins une étape de délai avant que les *handlers* destinataires ne reçoivent le message. Tous les tampons d'entrée ont donc au moins une case de profondeur.

Les changements sur les *modes* se font en trois phases. La première phase, la collecte des votes, a lieu immédiatement après que tous les *handlers* d'un étape aient été exécutés. Chaque *handler* revoie un ensemble de votes qui indique quels *modes* doivent être activés ou désactivés. Un *handler* ne peut demander le changement que des *modes* du *modal process* auquel il appartient. Cependant, un changement de *mode* local peut avoir un impact global par les ACTs[1] décrit au paragraphe suivant. La deuxième phase, la réconciliation des votes, les votes et les ACTs sont examinés pour déterminer si des *modes* doivent être changés à la fin de cette étape. Après la réconciliation, la dernière phase, la distribution des votes, chaque *modal process* reçoit le nouvel état de ses *modes*.

#### 3.3.1.2 Contrôle entre processus

Pour que les composants se coordonnent entre-eux, ils doivent partager un protocole. Pour cela, IPCHINOOK utilise des primitives appelées ACT (*Abstract Control Types*). Ces primitives sont

[1] Abstract Control Types

des règles établissant des relations entre les *modes*. Elles augmentent les votes de requêtes permettant de maintenir ces relations.

Les ACTs peuvent être utilisées par un *mode* pour surveiller le changement d'un autre *mode*, pour corréler deux *modes* de façon à ce qu'ils soient toujours actifs ou inactifs en même temps, et pour établir des relations d'exclusivité entre les *modes*. Des ACTs plus complexes forment des FSM similaire à StateCharts ou aux *watchdogs* de Esterel. IPCHINOOK propose une librairie de ACTs et permet la définition par l'utilisateur de ses propres ACTs.

Un ACT peut définir un protocole pour la composition entre processus. Un composant peut nécessiter une adaptation pour composer avec un autre, et cette organisation permet cette adaptation sans modification du composant. Ceci permet également aux composants de communiquer autrement que par passage de messages.

### 3.3.2 Description de la cible

La description de la cible définit l'architecture du système et la fonction d'allocation qui associe les *modal processes* et les canaux à l'architecture.

IPCHINOOK ne cherche pas à partitionner automatiquement l'application ni à associer automatiquement les processus aux composants. Ceux-ci doivent être fournis par l'utilisateur et peuvent être issus d'un outil de génération automatique.

Une architecture est définie par ses processeurs, ses systèmes d'exploitation et ses protocoles de communication. La fonction d'allocation associe les éléments de la description comportementale à ceux de l'architecture. Un élément de l'architecture peut être un processeur exécutant du logiciel ou un bloc de logique programmable comme un FPGA. La définition de l'architecture comprend aussi la topologie des communications décrivant l'interconnection des composants et les protocoles de communication qu'ils utilisent. IPCHINOOK dispose d'une librairie de protocoles comme $I^2C$, CAN, SCSI, USB, ....

La fonction d'allocation associe les processus aux processeurs et les canaux de communication aux médias de communication de l'architecture. L'association processus–processeur est telle que les processus sont indivisibles mais un processeur peut exécuter plusieurs processus. L'association canaux–médias n'a pas cette restriction : un canal peut utiliser plusieurs médias et un média peut être utilisé par plusieurs canaux. Même si la spécification du comportement utilise des passages de messages, le code synthétisé peut être exécuté sur un système à mémoire partagée.

### 3.3.3 Synthèse

La synthèse consiste à transformer la représentation de haut niveau vers une description plus proche de l'implémentation. Cette synthèse comprend deux étapes : le synthèse du *mode-manager* et la synthèse des communications et des interfaces.

#### 3.3.3.1 Synthèse du *mode-manager*

Le *mode-manager* est synthétisé pour coordonner les processus. Si aucune architecture n'est spécifiée, IPCHINOOK synthétise un *mode-manager* centralisé utilisable dans une architecture monoprocesseur pour la simulation. Par défaut, les processus s'exécutent en mode synchronisé : tous les processus sont bloqués et aucun *handler* n'est exécuté jusqu'à ce que les changements des *modes* soient résolues. D'autres modèles de synchronisation plus évolués sont supportés comme *mode synchrony* qui demande un *handshake* pour chaque changement d'état. Ces options sont spécifiées pour chaque architecture cible sans changement de la spécification.

#### 3.3.3.2 Synthèse des communications et des interfaces

La synthèse des communications et des interfaces implémente une infrastructure de communication spécifique à l'application permettant de gérer efficacement les messages de données de l'application et les messages de contrôle du *mode-manager*.

La synthèse des communications transforme les communications du protocole abstrait vers celui de l'architecture cible. La synthèse d'interface génère de la logique et des pilotes bas-niveau pour inter-connecter les éléments de traitement entre eux.

Les protocoles de communication abstraits sont utilisés pour spécificier le système indépendamment de l'architecture cible. Ces protocoles abstraits doivent être transformés et implémentés sur le protocole bas-niveau du bus utilisé par l'architecture cible. La synthèse des communications transforme ce modèle de communication indépendant de l'architecture en un modèle dépendant de l'architecture et génère le code des communications du *mode manager* pour chaque processeur de l'architecture. La synthèse des communications tient compte du protocole de bus, du routage et des contraintes temporelles pour toutes les communications du système.

Cette approche globale permet au concepteur d'appliquer les spécifications haut-niveau sur une architecture du système comportant n'importe quelle topologie de bus. Le concepteur est isolé des détails comme le protocole de bus ou les pilotes nécessaires pour obtenir un prototype.

La synthèse des communications dans IPCHINOOK est constituée de quatre étapes : la distribution des échéances multi-saut, le calcul des attributs du protocole de bus, la génération du routeur de messages et l'instanciation des pilotes bas-niveaux. Elle est suivi de la synthèse des interfaces qui instancie les pilotes à partir de l'architecture du système.

La synthèse multi-sauts crée des processus de saut lorsque des processus connectés logiquement sont exécutés par des processeurs qui ne sont pas connectés directement. Le processus de saut route le message à travers les processeurs et les bus intermédiaires. Comme les liens de communication abstraits peuvent avoir une entrée et plusieurs sorties, un message doit pouvoir prendre plusieurs chemin pour atteindre toutes les destinations. L'échéance associée au message est distribuée sur les chemins empruntés par le message de telle sorte que les paramètres du protocole de bus pour chaque message puissent être déterminés.

La synthèse des attributs du protocole de bus détermine les paramètres du protocole appropriés à chacun des messages. Une classification des protocoles en fonction de la politique d'arbitrage a été définie pour permettre la synthèse de ces paramètres. Ces attributs de protocoles de bus peuvent être un identificateur de message, un identificateur de processeur ou une priorité en fonction du protocole. Ils sont synthétisés de sorte que les messages ayant des contraintes temporelles plus fortes aient une plus grande priorité lors de l'arbitrage pour l'accès au bus. Les messages contiennent aussi des informations de routage permettant au système d'exploitation de chaque processeur de délivrer le message à son destinataire.

Le pilote bas-niveau ainsi que les messages de bus contenant les attributs synthétisés du protocole permettent au concepteur de s'abstraire des particularités du protocole de bus. Par exemple, le Controller Area Network (CAN) ne transfert que huit octets données par message. Le pilote adapté pour le processeur émettant sur ce bus découpe automatiquement les messages plus larges en segments de huit octets et le pilote exécuté par le processeur récepteur reconstruit le message. Le message reconstruit est passé au routeur synthétisé qui examine les informations de routage du message et le délivre sur le port d'entrée correspondant. Le port d'entrée synthétisé inclue le tampon spécifié dans la description haut-niveau du système.

Les pilotes sont automatiquement instanciés depuis la librairie de protocole. La synthèse d'interfaces adapte ces pilotes qui lisent et écrivent directement sur les connecteurs du processeur. Ces routines doivent refléter toutes la logique sauvage introduite. Pour réaliser l'interface entre un processeur et du matériel, on utilise soit les ports d'entrées–sorties du processeur, soit les adresses d'entrées–sorties.

La synthèse des communications permet d'explorer plusieurs affectations des tâches, plusieurs topologies de bus et plusieurs protocoles de communication. Ce qui est nécessaire pour trouver une solution efficace qui vérifie les contraintes.

### 3.3.4 Conclusion

L'approche proposée permet de synthétiser les communications d'un système en utilisant une bibliothèque décrivant les protocoles que l'on souhaite pouvoir utiliser. De plus, l'utilisation de technique de routage permet à tout processeur du système de communiquer avec tous les autres. Ces deux techniques doivent permettre de pouvoir utiliser cette méthode pour réaliser des systèmes dont les architectures sont complexes comme celles comportant des hiérarchies de bus. L'effort de formalisation n'est semble-t-il pas encore assez poussé pour rendre le système de synthèse accessible.

Cependant, la description du système est très singulière et fort complexe. Le comportement des tâches y est décrit comme des gestionnaires d'évènements. Cette description semble inadapté par exemple à la description de systèmes de type flot de données.

## 3.4 Sélection et allocation de communication à partir de protocoles abstraits

### 3.4.1 Modèle de communication utilisé

L'approche proposée par J.M. Daveau dans [DMBIJ97] décrit le système comme des processus communicants à travers de canaux abstraits. Pour cela, les processus utilisent des primitives prédéfinies, les services.

Lorsque les processus font appel à ces primitives, le canal réalise la communication de façon transparente du point de vue du processus. Ces procédures sont la seule partie visible du canal. Ceci permet aux processus de communiquer par des schémas de communication haut-niveau en masquant les détails de leur implémentation.

### 3.4.2 Modélisation des unités de communication

Pour réaliser les canaux abstraits, des unités de communication sont utilisées. Une unité de communication est une abstraction d'un composant physique. Une unité de communication est choisie dans une bibliothèque et instanciée lors de la synthèse des communications.

Une unité de communication est un objet qui peut exécuter une ou plusieurs des primitives de communication en utilisant un protocole défini. Ces services peuvent partager des ressources communes (comme un bus, un arbitre de bus ou une mémoire tampon). Une unité de communication peut inclure un contrôleur qui détermine le protocole utilisé. Les services interagissent avec le contrôleur pour modifier l'état de l'unité de communication et synchroniser la communication. Tous les accès à l'interface d'une unité de communication sont faits par ces services. Ceux-ci déterminent aussi le protocole utilisé pour échanger des paramètres entre le processus et une unité de communication. L'usage de services permet de masquer les détails du protocole dans la librairie, un service pouvant avoir différentes implémentations suivant l'architecture cible.

Les unités de communication diffèrent des canaux abstraits en implémentant un protocole spécifique et sa réalisation matériel–logiciel. Plusieurs canaux abstraits peuvent être réalisés par une seule unité de communication si elle fournit tous les services nécessaires. Ceci est appelé la fusion de canaux abstraits.

Ce modèle permet à l'utilisateur de définir un large choix de schémas de communication tel que le passage de messages ou la mémoire partagée.

### 3.4.3 Synthèse des communications

Deux étapes sont nécessaires à la synthèse des communications : la première, la sélection du protocole et allocation des unités de communication, fixe la structure du réseau et les protocoles utilisés. La seconde, la synthèse des communications, adapte les interfaces des différents processus au réseau de communication.

#### 3.4.3.1 Sélection du protocole et allocation des unités de communication

L'allocation des unités de communication utilise la description du système en processus communicant par les canaux abstraits et une bibliothèque d'unités de communication. Ces unités sont une abstraction de composants physiques. Cette étape choisit dans la bibliothèque les unités de communication nécessaires à la réalisation des services utilisés par les processus. Elle fige ainsi le protocole utilisé par chaque primitive de communication, puisqu'une unité de communication utilise un protocole spécifique pour chaque canal abstrait. Plusieurs canaux abstraits peuvent être réalisés par une seule unité de communication si elle peut gérer différentes communications indépendantes. Regrouper plusieurs canaux abstraits dans une unité permet de partager un média de communication.

L'allocation des unités de communication est relative au compromis surface/coût. Le choix d'une unité de communication ne dépend pas que de la communication elle-même, mais aussi des performances requises des processus communicants. Ce compromis est implémenté par une fonction de coût utilisée par l'algorithme d'allocation.

#### 3.4.3.2 Synthèse des interfaces

La synthèse des interfaces choisit une implémentation pour chaque unité de communication. Elle génère les interfaces nécessaires avec chaque processus utilisant les unités de communication. Une implémentation de chaque unité de communication est choisie en fonction des débits de données, de la capacité des tampons et du nombre de signaux de données et de contrôle. Les interfaces des processus sont adaptées en fonction de l'implémentation choisie et de l'interconnection.

Le résultat de la synthèse des communications est un ensemble de processeur communicant par des signaux, des bus et des composants additionnels comme un contrôleur de bus ou une fifo. Cette méthode permet de synthétiser les communications d'une application vers n'importe quel protocole comme un simple *handshake* ou un protocole plus complexe.

### 3.4.4 Conclusion

L'approche proposée permet conceptuellement de réaliser la synthèse des communications comme un problème d'allocation. Cette approche permet la sélection automatique de protocole. La synthèse des interfaces peut être réalisée automatiquement. La contrepartie à la généricité de cette approche est qu'elle repose sur l'existence d'une bibliothèque contenant tous ce que l'on veut pouvoir faire en matière de communication et la réalisation pratique reste difficile à concevoir.

## 3.5 Conclusion

A. Kalavade propose dans sa thèse d'utiliser un controleur central pour réaliser les communications. L'ordre des communications est décrit statiquement dans ce contrôleur. Cette approche ne permet pas de modifier la partie logicielle sans resynthètiser ce contrôleur matériel.

L'approche proposée dans Vulcan utilise une description explicitant le parallélisme de l'application. Le transport des données pour les communications avec les coprocesseurs matériels et entre les coprocesseurs est réalisé par l'unique processeur qui doit être le seul maître sur le bus. Cette approche limite l'usage du processeur pour l'exécution de tâches puisqu'il doit réaliser toutes les communications.

Dans IPChinook, la description du système est faite par des gestionnaires (*handlers*) réagissant à des événements. Les gestionnaires communiquent entre eux par des variables partagées (*modes*). La synthèse des communications est basé sur une bibliothèque décrivant les protocoles que le concepteur souhaite utilisé. La méthode utilisée repose sur la description du système qui ne semble pas adapté à des systèmes de type « flot de données ».

Enfin, la dernière approche présentée propose de réaliser les communications comme une allocation de composants d'une bibliothèque. La description du système en processus communicant par des canaux explicite le parallélisme de l'application et normalise les communications. Cette approche ne présente pas de caractéristique pratique de ces bibliothèques.

Les approches reposant sur des bibliothèques décrivent peu les mécanismes de communication mis en jeu et l'implémentation des primitives de communication qu'elles soient matérielles ou logicielles.

Notre approche est donc de plus bas-niveau, elle vise cependant à l'implémentation et à l'évaluation des composants de ces bibliothèques. Ces dernières sont nécessaires à la réalisation rapide de systèmes embarqués en permettant la réutilisabilité de composants par l'usage de protocole standard.

# Chapitre 4

# Synthèse des communications

La synthèse des communications consiste à réaliser physiquement les canaux permettant les échanges de données entre les tâches. Nous partons pour cela d'une description dans laquelle le concepteur du système a choisi, selon des critères qui dépassent le cadre de cette thèse, d'affecter un sous-ensemble de tâches à une exécution logicielle et le complément à une exécution matérielle. Ce choix n'est pas forcément définitif et la synthèse des communications doit être rapide à mettre en œuvre pour permettre plusieurs itérations. En effet, ceci permet d'explorer rapidement plusieurs alternatives d'implémentation du système.

Le but de ce chapitre est double : d'une part nous présentons un ensemble, certainement non-exhaustif mais cependant utilisable en pratique, de protocoles de communication entre les tâches matérielles ou logicielles ; d'autre part nous définissons les ressources matérielles et logicielles nécessaires à l'échange des données et à la synchronisation.

## 4.1 Spécification initiale

La synthèse des communications est réalisée à partir de trois descriptions : la spécification parallèle de l'application, l'architecture cible choisie pour le système et le choix d'implémentation de chaque tâche en matériel ou en logiciel. Nous définissons maintenant un peu plus précisément ces trois éléments, ainsi que les hypothèses d'implémentation que nous avons retenues.

### 4.1.1 La spécification parallèle de l'application

La spécification parallèle écrite à l'aide de la bibliothèque FSS (*Fonctional System Specification*). Cette bibliothèque réalise les canaux et les primitives de communication. Les tâches sont des *threads* POSIX. Les primitives de communication bloquantes READ() et WRITE() du modèle de Kahn sont implémentées en utilisant les moyens de synchronisation standard POSIX. Le comportement des tâches est décrit en langage C. L'utilisation d'un langage très répandu et d'une bibliothèque standard pour le parallélisme permet d'exécuter cette spécification sur la plupart des systèmes d'exploitation pour station de travail en vue de la validation et de l'évaluation du comportement de l'application. Un gros avantage de cette approche est que les tâches logicielles peuvent être exécutées sur des systèmes d'exploitation embarqués sans modification de la spécification.

Dans la spécification parallèle, les tâches communiquent entre elles par des canaux. La figure 4.1 illustre les caractéristiques d'un canal.

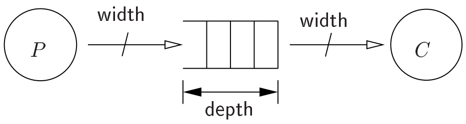



FIG. 4.1 – Un canal de communication entre deux tâches

Un canal contient une fifo de profondeur finie. Les caractéristiques de cette fifo sont sa profondeur (depth) qui est le nombre de cases dont elle dispose, et la largeur (width) de ses signaux de données en entrée et en sortie.

Une tâche écrit ou lit dans un canal à l'aide de deux primitives bloquantes :

$$\textsc{channelWrite}(channel, tampon_p, size_p);$$

$$\textsc{channelRead}(channel, tampon_c, size_c);$$

La variable $size$ est la quantité de données que la tâche souhaite lire ou écrire dans un canal. C'est un multiple de width. L'ordre de grandeur de $size$ est très variable (pour une tâche traitant des images par exemple, on peut envisager qu'elle échange un pixel, un bloc de pixels, ou même une image complète). La variable $tampon$ est une zone mémoire locale à la tâche suffisamment grande pour contenir $size$ éléments. Les primitives retournent le nombre de données lues ou écrites dans le canal, soit respectivement $size_c$ ou $size_p$ puisqu'elles sont bloquantes.

L'implémentation des primitives de communication dans FSS utilise deux moyens de synchronisation des *threads* POSIX : `mutex` pour l'accès aux données partagées et `condition` pour la suspension des tâches sur les entrées–sorties. Les figures 4.2 et 4.3 donnent l'algorithme de l'écriture et de la lecture du canal. On notera $data$ le tampon du canal, $status$ le nombre de cases pleines dans ce tampon, $write$ son pointeur d'écriture et $read$ son pointeur de lecture.

```
CHANNELWRITE(channel, buf, size)
1  n ← size
2  forever
3  do p ← depth − status
4     m ← min(n, p)
5     if m ≤ depth − write
6       then MEMCPY(data + write, buf, m)
7       else end ← depth − write
8            MEMCPY(data + write, buf, end)
9            MEMCPY(data + write, buf + end, m − end)
10    write ← (write + m) mod depth
11    n ← n − m
12    MUTEX_LOCK(channel)
13    status ← status + m
14    if n = 0
15      then break
16    COND_WAIT(channel)
17    MUTEX_UNLOCK(channel)
18 COND_SIGNAL(channel)
19 MUTEX_UNLOCK(channel)
20 return size
```

- 3 calcul de la place disponible
- 4 $m$ vaut soit le nombre de données à copier s'il y a suffisamment de place et la place disponible sinon
- 5–10 copie de $m$ données dans le tampon circulaire et mise à jour du pointeur d'écriture
- 12–13 mise à jour de l'état de la fifo protégée par un verrou
- 14–15 si tout a été copié, sortie de la boucle
- 16 sinon attente de la lecture de la fifo
- 17–18 relâchement du verrou
- 19 réveille la tâche attendant éventuellement l'écriture de la fifo

FIG. 4.2 – Écriture dans un canal

Les primitives d'accès à un canal reposent sur deux moyens de synchronisation :

**mutex :** c'est un verrou permettant de protéger l'accès concurrent à des données. Ce verrou est une variable partagée par les *threads* accédant aux données. La protection est obtenue en posant le verrou avant d'exécuter les portions de programme qui y accèdent. Deux *threads* ne peuvent acquérir en même temps le même verrou. La fonction MUTEX_LOCK est bloquante si une thread $y$ l'exécute alors qu'une autre *thread* $x$ l'a déjà fait sur le même verrou, la *thread* $y$ reste bloquée jusqu'à ce que le verrou soit libéré par l'appel par $x$ de la fonction MUTEX_UNLOCK. Un **mutex** est utilisé dans le cas des primitives CHANNELWRITE et CHANNELREAD des figures 4.2 et 4.3 pour interdire la modification concurrente de l'état

```
CHANNELREAD(channel, buf, size)
 1  n ← size
 2  forever
 3  do p ← status
 4     m ← min(n, p)
 5     if m ≤ depth − read
 6       then MEMCPY(buf, data + read, m)
 7       else end ← depth − read
 8            MEMCPY(buf, data + read, end)
 9            MEMCPY(buf + end, data + read, m − end)
10     read ← (read + m) mod depth
11     MUTEX_LOCK(channel)
12     status ← status − m
13     n ← n − m
14     if n = 0
15       then break
16     COND_WAIT(channel)
17     MUTEX_UNLOCK(channel)
18  COND_SIGNAL(channel)
19  MUTEX_UNLOCK(channel)
20  return size
```

3 lecture du nombre de données présentes dans la fifo,

5–11 lecture des données du tampon circulaire,

14–15 si *size* données ont été lues ; sortie de la boucle de lecture,

16 sinon, attente de l'écriture de données par une autre tâche,

18 signalement de la lecture à la tâche attendant pour écrire d'autres données dans la fifo,

19 relâchement du verrou.

FIG. 4.3 – Lecture dans un canal

d'une fifo par les deux tâches productrice et consommatrice,

**condition :** une condition est une variable associée à des données partagées par plusieurs *threads*. Elle permet à une *thread* de se suspendre (COND_WAIT) pour attendre la modification des données partagées. Elle est réveillée lorsqu'une autre *thread* signale que la condition a changé (COND_SIGNAL), indiquant que les données partagées ont été modifiées. Un COND_WAIT est toujours associé à un verrou (**mutex**) qu'il relâche afin de permettre à une autre *thread* d'accéder à la ressource. Il sert dans notre cas à suspendre une tâche essayant d'effectuer une lecture ou une écriture impossible résultant de l'absence de données ou de place disponible dans la fifo. A chaque accès, la tâche ayant modifié l'état de la fifo signale la condition pour réveiller la tâche éventuellement en attente.

Seul le registre $status$ est protégé par un verrou car c'est la seule ressource de la fifo qui peut être modifiée par le consommateur et le producteur. Le pointeurs de lecture et d'écriture ne sont accédés et modifiés que par une seule tâche (respectivement le consommateur et le producteur). Les données sont protégées par le registre $status$ : le consommateur ne lit pas plus de données qu'il y en a dans la fifo d'après le registre $status$. De même, le producteur n'écrit pas plus de données qu'il n'y a de cases vides dans la fifo d'après ce même registre.

Cette description du graphe peut être compilée et exécutée sur une station de travail car tous les systèmes d'exploitation actuels implémentent les *threads* POSIX. Son exécution permet de valider la spécification parallèle sous forme de graphe de tâches communiquantes par rapport à la spécification séquentielle initiale en comparant leur comportement vis-à-vis des mêmes stimuli d'entrée, comme par exemple l'exécution d'une plage mp3 ou d'une séquence vidéo.

FSS permet aussi d'extraire les caractéristiques des communications :

– une analyse de l'exécution de la spécification permet de déterminer la complexité de chaque tâche pour un processeur. Cette analyse est d'autant plus précise qu'elle est réalisée sur le processeur embarqué du système cible,
– les canaux de communication de FSS permettent de connaître le volume de données échangé entre les tâches et la granularité des échanges si celle-ci est dépendante des données.

### 4.1.2 L'architecture du système

L'architecture du système décrit le nombre de processeurs et de coprocesseurs présents dans le système et les moyens de communication utilisables pour chaque élément pour communiquer.

Une architecture cible possible est présentée par la figure 4.4. Elle fait l'hypothèse que la cohérence de la mémoire est assurée par le matériel.

Il s'agit d'une architecture classique avec un ou plusieurs processeurs exécutant la partie logicielle de l'application ainsi que les pilotes des coprocesseurs matériels, et un bus central permettant l'échange des données à travers une mémoire partagée.

Un micro-noyau supportant plusieurs processeurs (SMP) supportant les threads POSIX est utilisé pour répartir les tâches sur les processeurs. Il utilise un échéancier centralisé et les tâches s'exécutent indifféremment sur n'importe quel processeur.

Les tâches « matérielles » sont réalisées par un ou plusieurs coprocesseurs. Chaque coprocesseur possède un module d'interface lui permettant de communiquer avec les autres compo-

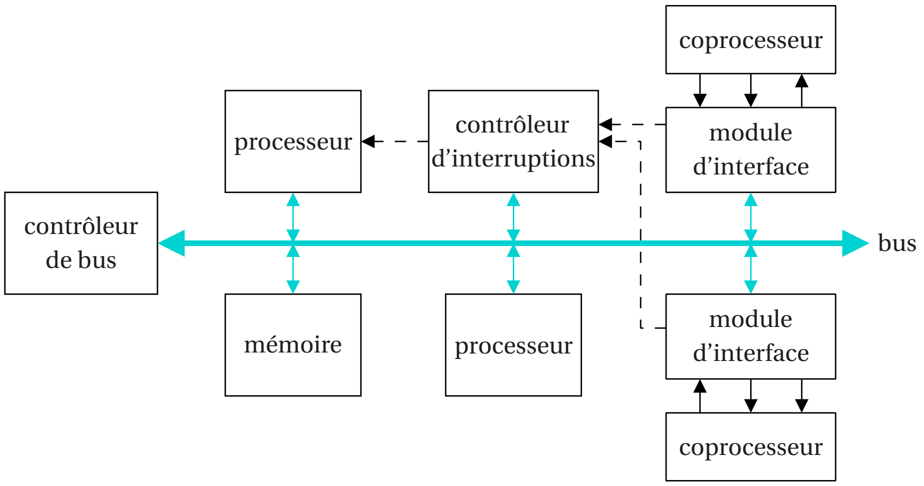



FIG. 4.4 – Architecture type de systèmes

sants du système par l'intermédiaire du bus.

### 4.1.3 L'affectation de chaque tâche

Le dernier élément nécessaire à la synthèse des communications est le choix d'affectation de chaque tâche soit en matériel soit en logiciel ainsi que le choix des coprocesseurs pour les tâches réalisées en matériel.

#### 4.1.3.1 Une tâche logicielle

Les tâches sont écrites en langage C Elles utilisent les primitives de communication bloquantes CHANNELREAD() et CHANNELWRITE(). Celles-ci s'appuient sur les threads POSIX. Puisque nous disposons d'un système d'exploitation multi-tâches, multiprocesseurs, implémentant les threads POSIX, la description des tâches logicielles n'est pas modifiée par rapport à la spécification utilisée par le graphe des tâches.

#### 4.1.3.2 Une tâche matérielle

Une tâche est réalisée en matériel par la synthèse d'un coprocesseur spécialisé ou l'utilisation d'un coprocesseur existant capable d'en implémenter les spécifications (fonctionnelles et non-fonctionnelles). Plusieurs tâches peuvent être regroupées dans un même coprocesseur ;

dans ce cas, la synthèse des communications entre ces tâches est à la charge de l'outil de synthèse de haut niveau synthétisant les différentes tâches réalisées par ce coprocesseur.

Il faut donc définir un protocole de communication permettant à un coprocesseur matériel de communiquer avec le reste du système. Nous avons choisi d'utiliser des protocoles les plus proches possible des primitives de communication utilisées dans la spécification et d'insérer entre le coprocesseur et le support physique de la communication un module d'interface permettant de réaliser l'ensemble des actions nécessaires à la communication. Ce module devra en particulier permettre le transfert des données et la synchronisation entre l'émetteur et le récepteur. Ce choix a été fait pour plusieurs raisons :

- l'outil de synthèse de coprocesseur matériel n'a pas à synthétiser un protocole complexe comme celui d'un bus, ni à gérer de ressources externes comme les interruptions, les adresses systèmes, ...
- le coprocesseur synthétisé étant indépendant du protocole de bus, celui-ci est plus facilement réutilisable,
- les contraintes temporelles du bus (disponibilité des signaux, ...) n'ont pas besoin d'être prise en compte par l'outil de synthèse.

### 4.1.4 Interface matérielle d'un coprocesseur

L'interface matérielle d'un coprocesseur utilise un protocole simple proche des primitives de communication logicielles CHANNELREAD et CHANNELWRITE. Nous avons étudié deux protocoles : le protocole fifo et le protocole vecteur. Nous utilisons le protocole vecteur parce qu'il reprend la sémantique des primitives de communication de la spécification parallèle. Cependant, le module d'interface supporte également le protocole fifo qui est utilisé par l'outil de synthèse de haut-niveau de notre laboratoire d'accueil.

#### 4.1.4.1 Le protocole fifo

Les canaux de communication étant de type fifo, le premier protocole est donc naturellement un protocole de type fifo. Ses signaux et sa sémantique sont décrits par le figures 4.5 et 4.6.

Un signal de commande indique que la tâche souhaite écrire une donnée dans la fifo et un

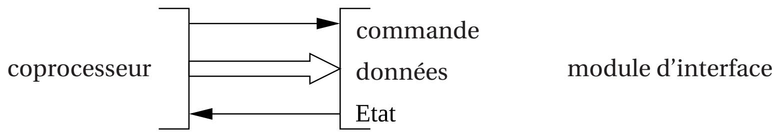



FIG. 4.5 – Interface du protocole fifo

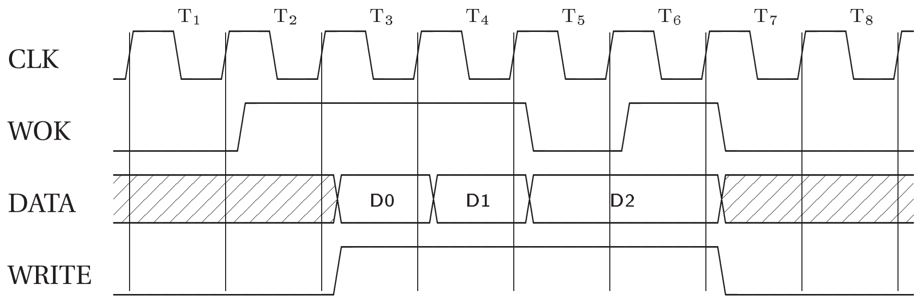



FIG. 4.6 – Le protocole fifo

signal d'état permet à la fifo d'indiquer que de la place est disponible pour cette écriture. Si ces deux signaux sont actifs alors un échange est réalisé.

Ce protocole à le mérite de la simplicité et permet la synchronisation requise. Le coprocesseur est bloqué dès qu'il souhaite écrire et que le signal d'autorisation n'est plus valide (la fifo est pleine) ou qu'il souhaite lire et que la fifo est vide.

Cependant, ce protocole ne tient pas compte de l'écriture par blocs des primitives. En effet, il faut $n$ écritures successives dans une fifo pour traduire un appel de la primitive CHANNELWRITE si $n$ est la taille du transfert donnée en paramètre à CHANNELWRITE. C'est pourquoi nous avons défini un autre protocole : le protocole vecteur.

#### 4.1.4.2 Le protocole vecteur

Le protocole vecteur (figure 4.7 et 4.8 reprend la sémantique des primitives CHANNELREAD et CHANNELWRITE : la tâche définit un tampon matériel à remplir ou à vider et indique le nombre de données à échanger. Elle est bloquée tant que le transfert n'est pas terminé.

Le coprocesseur possède un signal de requête (`req`) accompagné de la taille du transfert à effectuer (`length`). Le signal `step` demande au coprocesseur de présenter la donnée suivante au cycle suivant. Le signal `done` indique la fin du transfert (figure 4.9).

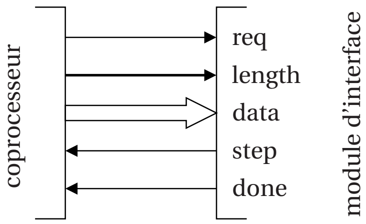



FIG. 4.7 – Signaux d'interface du protocole vecteur pour un CHANNELWRITE

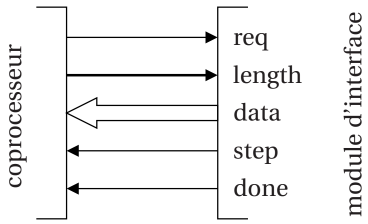



FIG. 4.8 – Signaux d'interface du protocole vecteur pour un CHANNELREAD

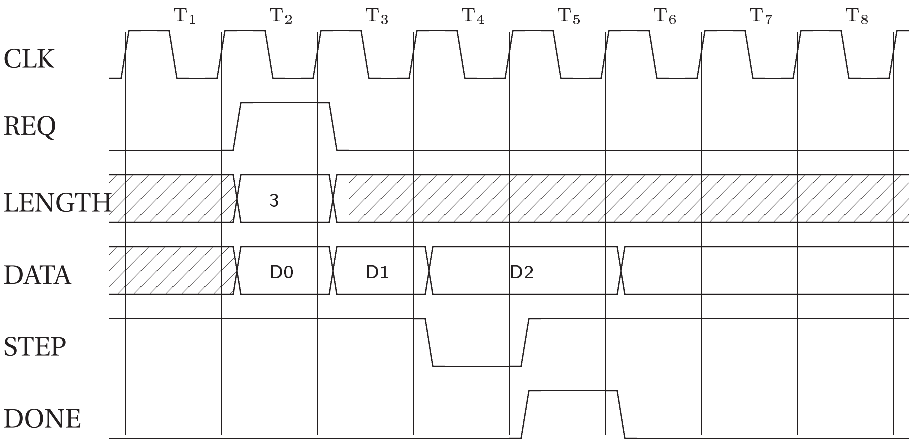



FIG. 4.9 – Le protocole vecteur

L'assertion du signal `req` correspond dans le protocole matériel à l'appel de la primitive du logiciel. Il est accompagné de la taille du transfert donné par le signal `length` et il garantit que le coprocesseur peut fournir ou recevoir les données à la cadence souhaité par l'interface. Il signifie qu'un tampon est disponible pour la communication.

Le tampon est interne au coprocesseur. Il doit l'adresser pour lire ou écrire les données échangées avec l'interface à chaque signal `step` valide. En effet, le module d'interface ne fournit pas d'adresse, ni d'index. Ce choix a été fait pour laisser libre l'implémentation du tampon en fonction de sa taille et de son utilisation par le coprocesseur. Ce peut être une fifo, un banc de registres multi-accès, une mémoire simple accès, ...

Le protocole vecteur permet de réaliser les primitives bloquantes pour l'émission ou la réception des données par une tâche réalisée en matériel : le coprocesseur est en attente du signal `done`. Ce signal représente en quelque sorte le retour de la primitive de communication. Il faut maintenant acheminer les données vers l'autre point de la communication.

## 4.2 Communications matériel–logiciel

Dans la spécification parallèle, l'appel à une primitive de communication qui ne peut être satisfaite en raison de l'état de remplissage de la fifo suspend la tâche. Dans l'implémentation POSIX, ceci est réalisé par une condition propre à chaque canal permettant de bloquer la primitive en suspendant la tâche. Les tâches peuvent aussi être suspendues lors de tentatives d'acquisition du verrou protégeant la mise à jour de l'état du canal contre les accès concurrents.

Dans le cas des communications matériel–logiciel, une tâche matérielle doit pouvoir réveiller une tâche logicielle suspendue par une condition. Une tâche matérielle ne peut pas, comme le fait une tache logicielle, changer la condition et faire appel au système d'exploitation pour qu'il rende éligible la tâche attendant sur cette condition. Il doit faire appel à un processeur pour qu'il exécute le code correspondant.

Pour ce faire, le matériel dispose d'un mécanisme de signalisation asynchrone : l'interruption. Le gestionnaire d'interruptions sera alors programmé pour changer la condition. Ceci pose un problème car l'interruption est émise de façon asynchrone et le la tâche en cours d'exécution n'est pas suspendue lorsqu'on exécute le gestionnaire d'interruptions. Du point de vue de l'échéancier, le processeur exécute la fonction correspondant à l'interruption comme s'il continuait à exécuter la tâche. Suspendre le gestionnaire d'interruptions par un verrou ou une condition suspend en réalité la tâche qui était exécutée au moment où l'interruption a été reçue. Le gestionnaire d'interruptions ne peut donc pas être suspendu en attendant une ressource, sans risquer que la tâche possédant la ressource soit celle que le processeur exécutait lorsqu'il a été interrompu. Ceci conduirait à un inter-blocage puisque la tâche interrompue ne pourrait relâcher la ressource. Le gestionnaire d'interruptions ne peut donc pas prendre de verrou.

Or, Pour que le gestionnaire d'interruptions puisse changer la condition sur laquelle la tâche logicielle s'était suspendue. Il doit s'assurer que la tâche est bien en attente sur cette condition avant de signaler son changement sans quoi la tâche ne serait jamais réveillée. Lorsque deux tâches se synchronisent par une condition, si une tâche exécute COND_SIGNAL avant que l'autre n'ait exécuté COND_WAIT, alors cette dernière ne verra pas le signalement de la condition. C'est pourquoi, on protège les conditions par des verrous. Le cas se présente typiquement lorsque le temps de traitement du coprocesseur est très petit comparé au temps d'exécution du code de suspension, permettant au coprocesseur d'interrompre le processeur avant que la tâche qui a lancé le coprocesseur ne soit effectivement suspendue. Le gestionnaire d'interruptions ne pouvant pas utiliser un verrou pour s'assurer que la tâche attend bien sur la condition, il faut utiliser une autre méthode.

Dans l'exemple de la figure 4.10, on voit que l'interruption peut être reçue après la ligne 4. Si elle survient avant la suspension effective de la tâche par le COND_WAIT de la ligne 5, la tâche ne sera pas en attente sur la condition et si le gestionnaire d'interruptions signale la condition sans s'assurer que la tâche est bien en attente, le signalement ne sera pas vu.

```
PRIMITIVE(channel)
1  ...
2  MUTEX_LOCK(mutex)
3  ...
4  UNMASKINTERRUPT(line)
5  COND_WAIT(cond, mutex)
6  ...
```

2 prise du verrou protégeant la condition,

4 autorisation de l'interruption par laquelle la tâche matérielle va communiquer,

5 suspension de la tâche sur la condition (cette fonction relâche aussi le verrou).

FIG. 4.10 – Exemple de primitive de communication utilisant une interruption

Les threads POSIX possèdent un autre moyen de synchronisation que nous n'avons pas utilisé pour les communications logiciel→logiciel : les sémaphores. Un sémaphore est un compteur dont l'accès est atomique. Deux fonctions sont utilisées pour y accéder : SEM_POST incrémente la valeur du compteur de manière atomique et ne peut en aucun cas suspendre la tâche. Si la valeur du compteur est nulle, SEM_WAIT bloque la tâche qui l'exécute jusqu'à ce que la valeur du compteur soit non nulle. Elle décrémente aussi le compteur.

Les sémaphores présentent pour nous, dans le cas des interruptions, un avantage par rapport aux conditions : si la tâche exécute SEM_POST avant que l'autre n'ait exécuté SEM_WAIT, le compteur gardera la trace de SEM_POST et SEM_WAIT ne provoquera pas la suspension de la tâche.

Nous avons donc utilisé des sémaphores pour réaliser les primitives de communication logiciel→matériel et matériel→logiciel.

## 4.3 Schémas de communications

Afin d'évaluer le matériel nécessaire à la réalisation des communications, nous décrivons dans cette partie différents schémas de communication. Tous ces schémas correspondent à différentes implémentations des primitives CHANNELREAD et CHANNELWRITE et ont des comportements identiques du point de vue de la tâche.

### 4.3.1 Communication logiciel→logiciel

Puisque le système d'exploitation que nous utilisons implémente les threads POSIX, les communications entre les tâches logicielles peuvent être les mêmes que celles utilisées pour l'implémentation du graphe de tâches (figures 4.2 et 4.3). Nous noterons ce schémas SS1.

Dans le cas où le système possède plusieurs processeurs identiques, le système d'exploitation utilisé affectera les tâches logicielles indifféremment à n'importe quel processeur, la mémoire étant partagée. La synchronisation est assurée par les mécanismes de communication des threads POSIX, car ils s'appliquent aussi bien aux systèmes mono-processeurs qu'aux systèmes multi-processeurs.

### 4.3.2 Communication logiciel→matériel

La tâche productrice est réalisée en logiciel, la tâche consommatrice en matériel.

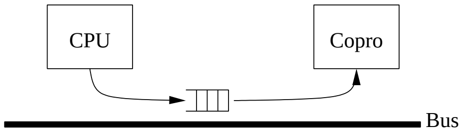



FIG. 4.11 – communication logiciel→matériel

Le tampon de communication peut soit exister physiquement dans le module d'interface du coprocesseur matériel, soit être en mémoire (comme dans le cas logiciel→logiciel).

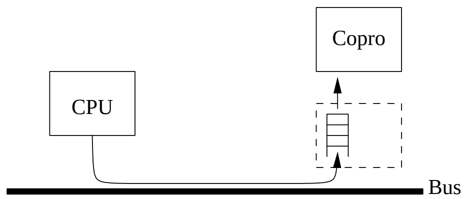



FIG. 4.12 – communication logiciel→matériel

#### 4.3.2.1 la fifo est dans le module d'interface : SH1

Si la fifo est dans l'interface (figure 4.12), la gestion de la fifo — la mise à jour de son état en fonction des lectures du coprocesseur et des écritures par le bus — est faite par le matériel du module d'interface. Cependant, le processeur doit pouvoir lire l'état de la fifo pour vérifier qu'il dispose de suffisamment de place libre pour réaliser son écriture. Si cela n'est pas le cas, la tâche productrice doit être suspendue comme pour une communication entre tâches logicielles. En effet, si la primitive d'écriture ne tenait pas compte de l'état de la fifo et que celle-ci était pleine, le module d'interface introduirait des « cycles d'attentes » sur le bus. Cela entraînerait un gaspillage de bande passante sur le bus et le gel du processeur (interdisant l'exécution d'autres tâches logicielle sur ce processeur). Ceci risquerait également de causer des inter-blocages lorsque par exemple ce processeur exécute aussi la tâche lisant les résultats fournis par le coprocesseur. S'il n'y a pas assez de place dans la fifo du module d'interface, la tâche productrice est donc suspendue, et la consommation de données dans la fifo par le coprocesseur doit alors réveiller la tâche productrice. Pour suspendre cette tâche, un sémaphore (`sem_wait`) est utilisé, et seule une tâche logicielle peut la réveiller (`sem_post`). Pour réveiller la tâche productrice, le module d'interface dispose d'une ligne d'interruption qu'il peut activer lorsque le nombre de cases vides dans la fifo est celui voulu par la tâche productrice. Le gestionnaire d'interruptions réveille la tâche productrice par une opération `sem_post`.

Les figures 4.13 et 4.14 présentent une réalisation de ce schéma de communication nommé SH1. Deux fonctions sont nécessaires : la primitive d'écriture CHANNELWRITE() et la routine de gestion de l'interruption. Elles utilisent trois ressources matériels interne au module d'interface : `status` qui est le nombre de cases pleines de la fifo, `threshold` le niveau programmable générant l'interruption et `fifo` la fifo elle-même.

```
CHANNELWRITE(channel, buf, size)
 1  n ← size
 2  forever
 3  do p ← depth − status
 4      m ← min(n, p)
 5      for i ← 0 to m − 1
 6      do fifo ← buf[i]
 7      n ← n − m
 8      if n = 0
 9        then break
10      threshold ← min(n, depth)
11      UNMASKINTERRUPT(itline)
12      SEM_WAIT(sem)
13  return size
```

3 calcul du nombre de cases vides dans la fifo,

5–6 écriture dans la fifo,

8–9 si tout à été écrit, sortie de la boucle,

10 sinon, programmation du threshold soit avec le nombre de données $n$ restant à écrire, soit avec la profondeur de la fifo si elle est inférieur à $n$,

11 démasquage de la ligne d'interruption du threshold,

12 attente de l'interruption pour reprendre l'écriture,

FIG. 4.13 – Écriture suivant le schéma SH1

```
THRESHOLDINTERRUPTHANDLER(channel)
 1  MASKINTERRUPT(itline)
 2  SEM_POST(sem)
```

1 masquage de l'interruption,

2 réveil de la tâche attendant la lecture de la fifo

FIG. 4.14 – Gestionnaire d'interruptions pour le schéma SH1

Cette méthode est bien adaptée lorsque la profondeur de la fifo permet que la fifo soit incluse dans le module d'interface. Autrement, il faut réaliser la fifo dans l'espace mémoire du système.

#### 4.3.2.2 la fifo est en mémoire : SH2

La fifo est réalisée comme un tampon circulaire dans la mémoire du système, accessible à la fois par le processeur et par le module d'interface (figure 4.15).

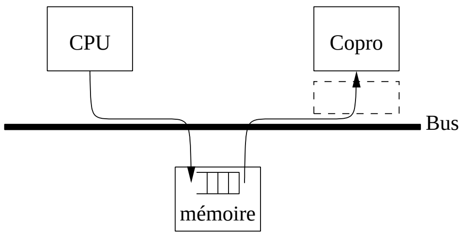



FIG. 4.15 – communication logiciel→matériel suivant le schéma SH2

L'écriture est faite par le processeur. La réalisation de la fifo en mémoire est la même que dans le cas logiciel→logiciel. La primitive d'écriture sera donc la même.

Pour la lecture, il faut fournir les mêmes services que pour une lecture effectuée par une tâche logicielle : copie des données, gestion de l'état de la fifo interdisant l'accès concurrent, et réveil de la tâche productrice.

Une lecture sans aucune intervention du logiciel nécessiterait que le module d'interface soit capable, en plus de la lecture des données, de mettre à jour l'état de la fifo logicielle, d'empêcher l'accès simultané du processeur et de réveiller une tâche logicielle. Ces actions sont très dépendant du système d'exploitation, il est préférable que ceci soit fait par le logiciel.

On pourrait imaginer qu'une tâche logicielle auxiliaire effectue le transfert de la mémoire vers le module d'interface fonctionnant en mode esclave (figure 4.16). Cette tâche possède deux canaux. Un premier en lecture pour lire les données de la mémoire. Un second pour écrire les données dans le module d'interface. La lecture s'apparente à la lecture d'une communication logiciel→logiciel, l'écriture à celle du schéma SH1.

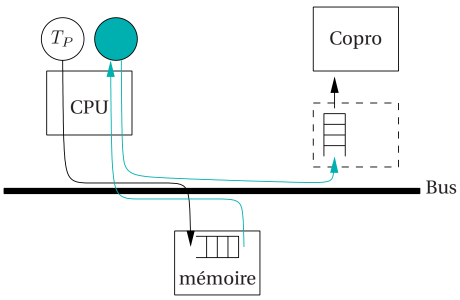



FIG. 4.16 – insertion d'une tâche logicielle pour réaliser la lecture

Ce mécanisme triple le coût de la communication par rapport au cas où la fifo est dans le module d'interface.

Pour diminuer l'utilisation du processeur, le transfert des données peut être fait par le module d'interface fonctionnant en mode maître sur le bus. Le module d'interface doit demander au processeur à chaque accès le nombre de données présentes dans la fifo. Ceci peut être fait par une interruption émise à chaque requête de lecture du coprocesseur. Le gestionnaire d'interruptions répond en écrivant dans un registre du coprocesseur le nombre de données disponibles. A la fin du transfert, le module d'interface interrompt à nouveau le processeur pour qu'il mette à jour l'état de la fifo en fonction des lectures qu'il a effectué.

Pour réaliser la lecture le module d'interface doit connaître les adresses du début et de la fin du tampon circulaire. Ces deux adresses sont constantes pour la communication et peuvent être programmées à l'initialisation. Il doit aussi connaître le nombre de données présentes dans la fifo. C'est le rôle de la tâche auxiliaire de transmettre cette information au module d'interface à chaque interruption.

Pour réduire le nombre d'interruptions, la tâche productrice peut indiquer au module d'interface le nombre de données présentes dans la fifo avant qu'il ne le demande. Pour cela, lorsque la tâche productrice effectue une écriture, la primitive indique au module d'interface le nombre de données qu'il peut lire, comme le faisait, dans le schéma précédent, le gestionnaire d'interruptions à la demande du module d'interface.

Cependant, le module d'interface ne mettant pas à jour l'état de la fifo en mémoire, le processeur doit le faire en lisant le nombre de données que le module d'interface a effectivement lu. Une interruption est toujours nécessaire, mais elle n'est utilisée que si la fifo devient pleine et que la tâche productrice doit être suspendue en attendant que le coprocesseur lise les données.

Ce mécanisme est décrit par les figures 4.17 et 4.18 ; il est nommé schéma SH2. Le quota de données que le module maître peut lire est nommé `transertLength`, le nombre de données de ce quota encore non-utilisé est nommé `remaining` et l'interdiction d'émettre des requêtes sur le bus est nommé `disconnect`.

```
CHANNELWRITE(channel, buf, size)
1  n ← size
2  forever
3  do disconnect ← 1
4     status ← status − (transfertLentgth − remaining)
5     p ← depth − status
6     m ← min(n, p)
7     if m ≤ depth − write
8        then MEMCPY(data + write, buf, m)
9        else end ← depth − write
10            MEMCPY(data + write, buf, end)
11            MEMCPY(data + write, buf + end, m − end)
12    write ← (write + m) mod depth
13    n ← n − m
14    status ← status + m
15    transfertLentgth ← status
16    disconnect ← 0
17    if n = 0
18       then break
19    UNMASKINTERRUPT(itline)
20    SEM_WAIT(sem)
21 return size
```

3 stoppe la lecture des données dans la fifo par le module d'interface pour garantir la cohérence des valeurs des ressources du module,

4 mise à jour de l'état de la fifo en fonction des lectures réalisées par le module d'interface,

5 calcul du nombre de cases vides,

6–12 écritures des données,

15 mise à jour de l'état de la fifo,

16 programmation du nombre de données lisible par le module d'interface à la valeur du nombre de cases pleines dans la fifo,

17 redémarrage du module d'interface,

18–19 si toutes les données ont été écrites, sortie,

20 démasquage de l'interruption correspondant à la fin de la lecture des données par le module d'interface,

21 attente de la lecture des données par le module d'interface.

FIG. 4.17 – Écriture suivant le schéma SH2

```
DISCONNECTINTERRUPTHANDLER(channel)
1  MASKINTERRUPT(itline)
2  SEM_POST(sem)
```

1 masquage de l'interruption,

2 réveil de la tâche attendant la lecture de la fifo

FIG. 4.18 – Gestionnaire d'interruptions (SH2)

#### 4.3.2.3 Optimisation : SH3

Si l'on regarde le travail effectué par le processeur dans la primitives de communication du schéma SH1 (figures 4.13), on peut remarquer que le processeur effectue deux transferts sur le bus pour copier les données du tampon interne à la tâche productrice qui est passé en paramètre à la primitive d'écriture vers la fifo.

On peut éviter un transfert en faisant fonctionner le module d'interface en mode maître plutôt qu'en mode esclave. La primitive CHANNELWRITE au lieu de copier les données dans le module d'interface, programme le module d'interface pour qu'il lise les données du tampon vers sa fifo. Cependant, la primitive CHANNELWRITE ne peut pas se terminer tant que le module d'interface n'a pas fini de lire le tampon. Un sémaphore peut être utilisé pour suspendre la tâche et

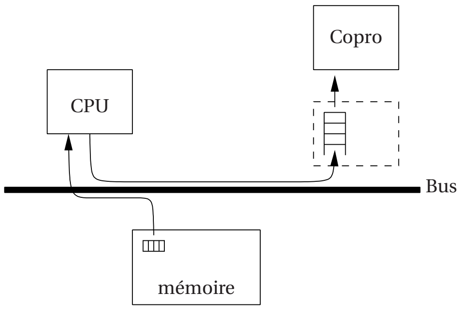



FIG. 4.19 – copie des données par le processeur

permettre au processeur d'en traiter une autre en attendant la fin de la lecture du tampon par le module d'interface. Une interruption émise par le module d'interface lorsqu'il a achevé la lecture réveillera alors la tâche.

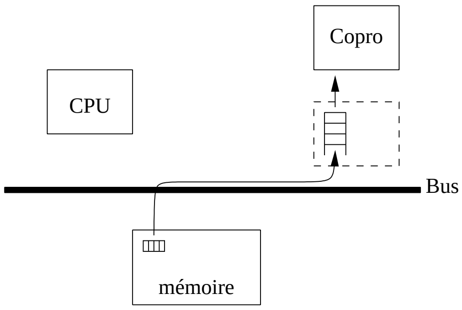



FIG. 4.20 – communication logiciel→matériel (SH3)

Ce schéma permet de soulager le processeur de la copie des données du tampon vers la fifo. Il n'est cependant pas efficace pour des tampons trop petits où la programmation de l'interface coûte plus cher que la copie des données. La figure 4.21 présente l'algorithme de la primitive CHANNELWRITE associée à ce schéma, nommé SH3. Le gestionnaire d'interruptions est celui de la figure 4.18.

CHANNELWRITE($channel, buf, size$)
1 `startAddress` $\leftarrow buf$
2 `transfertLentgth` $\leftarrow size$
3 `disconnect` $\leftarrow 0$
4 UNMASKINTERRUPT(itline)
5 SEM_WAIT($sem$)
6 **return** $size$

1–3 programmation du module maître pour lire $m$ données dans le tampon et démarrage du module d'interface,

4 démasquage des interruptions correspondant à la fin de la lecture des données par le module d'interface,

5 attente de la lecture des données par le module d'interface,

FIG. 4.21 – Écriture suivant le schéma SH3

```
DisconnectInterruptHandler(channel)
    maskInterrupt(itline)
    sem_post(sem)
```

1. masquage de l'interruption,
2. réveil de la tâche attendant la lecture de la fifo

FIG. 4.22 – Gestionnaire d'interruptions (SH3)

### 4.3.3 Communication matériel→logiciel

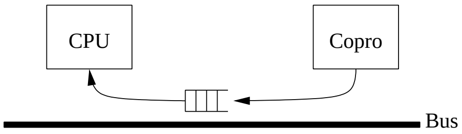



FIG. 4.23 – communication matériel→logiciel

La mise en œuvre de ce type de communication est similaire à la précédente. Le sens des données est inversé. On retiendra donc les trois mêmes types de schémas de communication : le schéma HS1 avec la fifo dans le module d'interface (figures 4.25 et 4.26), le schéma HS2 où la fifo est en mémoire (figure 4.28 et 4.29) et le schéma HS3 où la fifo est supprimée.

Les ressources nécessaires à la mise en œuvre de ces trois schémas sont aussi identiques à celles des schémas pour la communication logiciel→matériel.

#### 4.3.3.1 La fifo est dans le module d'interface : HS1

Dans le schéma HS1, la fifo du canal est réalisée en matériel dans le module d'interface. Celui-ci fonctionne en mode esclave. L'état de la fifo est géré par le module d'interface. Un sémaphore permet de suspendre la tâche si la fifo est vide. Un interruption est utilisée pour changer le sémaphore et réveiller la tâche. Cette interruption est émise par le module d'interface lorsqu'un nombre programmable (`threshold`) de données est présent dans la fifo.

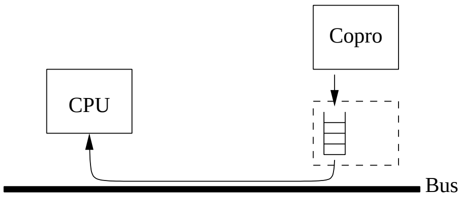



FIG. 4.24 – communication matériel→logiciel (HS1)

```
CHANNELREAD(channel, buf, size)
1  n ← size
2  forever
3  do p ← status
4      m ← min(n, p)
5      for i ← 0 to m − 1
6      do buf[i] ← fifo
7      n ← n − m
8      if n = 0
9         then break
10     threshold ← min(n, depth)
11     UNMASKINTERRUPT(itline)
12     SEM_WAIT(sem)
13 return size
```

3 lecture du nombre de cases pleines dans la fifo,

5–6 lecture des données,

8–9 si $size$ données ont été lues ; sortie de la boucle de lecture,

10 sinon, programmation du threshold soit avec le nombre de données $n$ restant à écrire, soit avec la profondeur de la fifo si elle est inférieur à $n$,

11 démasquage de la ligne d'interruption du threshold,

12 attente de l'interruption pour reprendre l'écriture,

FIG. 4.25 – Lecture suivant le schéma HS1

```
THRESHOLDINTERRUPTHANDLER(channel)
1  MASKINTERRUPT(itline)
2  SEM_POST(sem)
```

1 masquage de l'interruption,

2 réveil de la tâche attendant la lecture de la fifo

FIG. 4.26 – Gestionnaire d'interruptions (HS1)

#### 4.3.3.2 La fifo est en mémoire : HS2

Le schéma HS2 permet d'utiliser un canal dont la fifo est réalisée par un tampon circulaire dans la mémoire du système. Un accès en mode maître permet au module d'interface d'écrire dans la fifo. Un registre est utilisé pour indiquer au module d'interface le nombre de cases vides dont il dispose dans la fifo. Un sémaphore est utilisé pour suspendre la tâche si la fifo est vide. Une interruption permet au module d'interface de changer le sémaphore pour réveiller la tâche.

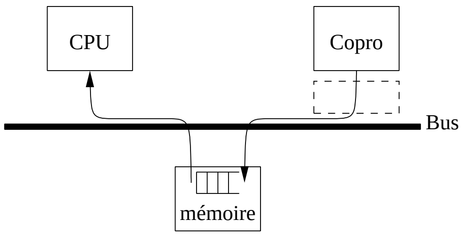



FIG. 4.27 – communication matériel→logiciel (HS2)

CHANNELREAD($channel$, $buf$, $size$)
1 $n \leftarrow size$
2 **forever**
3 **do** `disconnect` $\leftarrow 1$
4 $status \leftarrow status + ($`transfertLentgth` $-$ `remaining`$)$
5 $m \leftarrow \min(n, status)$
6 **if** $m \leq$ depth $- \ read$
7 **then** MEMCPY($buf$, $data + read$, $m$)
8 **else** $end \leftarrow depth - read$
9 MEMCPY($buf$, $data + read$, $end$)
10 MEMCPY($buf + end$, $data + read$, $m - end$)
11 $read \leftarrow (read + m) \mod$ depth
12 $n \leftarrow n - m$
13 $status \leftarrow status - m$
14 `transfertLentgth` $\leftarrow$ depth $- \ status$
15 `disconnect` $\leftarrow 0$
16 **if** $n = 0$
17 **then break**
18 UNMASKINTERRUPT(itline)
19 SEM_WAIT($sem$)
20 **return** $size$

3 stoppe l'écriture des données dans la fifo par le module d'interface pour garantir la cohérence des valeurs des ressources du module,

4 mise à jour de l'état de la fifo en fonction des écritures réalisées par le module d'interface,

6–12 écritures des données,

13 mise à jour de l'état de la fifo,

14 programmation du nombre de données inscriptibles par le module d'interface à la valeur du nombre de cases vides dans la fifo,

15 redémarrage du module d'interface,

16–17 si toutes les données ont été lues, sortie,

18 démasquage de l'interruption correspondant à la fin de l'écriture des données par le module d'interface,

19 attente de l'écriture des données par le module d'interface.

FIG. 4.28 – Lecture suivant le schéma HS2

DISCONNECTINTERRUPTHANDLER($channel$)
1 MASKINTERRUPT(itline)
2 SEM_POST($sem$)

1 masquage de l'interruption,

2 réveil de la tâche attendant la lecture de la fifo

FIG. 4.29 – Gestionnaire d'interruptions (HS2)

#### 4.3.3.3 Optimisation : HS3

Le schéma HS3 utilise le module d'interface en mode maître pour copier les données de la fifo vers le tampon interne à la tâche consommatrice. Il utilise un sémaphore pour suspendre la tâche jusqu'à ce que la copie soit terminée. Une interruption permet de modifier le sémaphore pour réveiller la tâche.

CHANNELREAD($channel$, $buf$, $size$)
1 `startAddress` $\leftarrow buf$
2 `transfertLentgth` $\leftarrow size$
3 `disconnect` $\leftarrow 0$
4 UNMASKINTERRUPT(itline)
5 SEM_WAIT($sem$)
6 **return** $size$

1–3 programmation du module maître pour écrire $m$ données dans le tampon et démarrage du module d'interface,

4 démasquage des interruptions correspondant à la fin de l'écriture des données par le module d'interface,

5 attente de l'écriture des données par le module d'interface.

FIG. 4.30 – Lecture suivant le schéma HS3

| | |
|---|---|
| DISCONNECTINTERRUPTHANDLER($channel$) | |
| 1 MASKINTERRUPT(itline) | 1 masquage de l'interruption, |
| 2 SEM_POST($sem$) | 2 réveil de la tâche attendant la lecture de la fifo |

FIG. 4.31 – Gestionnaire d'interruptions (HS3)

### 4.3.4 Communication matériel→matériel

Le tâche productrice et la tâche consommatrice sont réalisées par deux coprocesseurs distincts (figure 4.32).

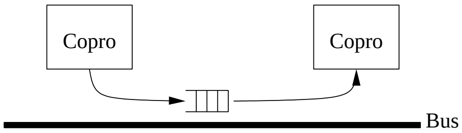



FIG. 4.32 – communication matériel→matériel

Les communications entre deux coprocesseurs matériels peuvent se faire soit directement, c'est à dire d'une interface à l'autre (figure 4.33) soit en passant par une mémoire. Dans ce dernier

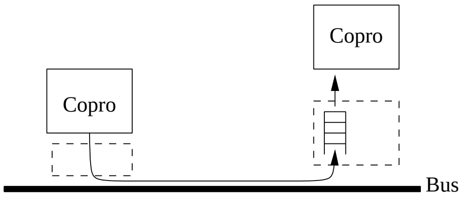



FIG. 4.33 – communication direct entre deux modules d'interface

cas, le module d'interface du coprocesseur producteur écrit dans une mémoire que le module d'interface du coprocesseur consommateur lit (figure 4.34).

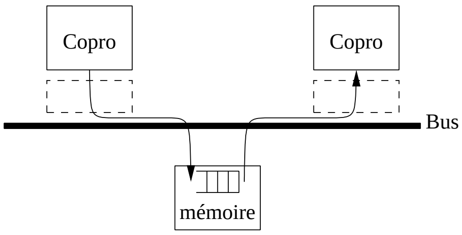



FIG. 4.34 – communication par la mémoire

Cette dernière solution est utile pour réordonner les données ou lorsque la production et la consommation des données sont très désynchronisées, ce qui peut nécessiter un stockage important de données. Que la consommation soit directe ou qu'elle passe par la mémoire la difficulté réside dans la synchronisation : stopper le module d'interface concerné lorsque la fifo est vide ou pleine suivant le sens du transfert. Comme les deux tâches sont matérielles, l'intervention du processeur doit être réduite à son minimum.

#### 4.3.4.1 Transfert direct

Dans le cas d'un transfert direct, soit l'un des modules d'interface possède un accès maître sur le bus et l'échange de données se fait directement entre les deux interfaces, soit la copie des données est assurée par un processeur ou par la programmation d'un contrôleur DMA (figure 4.35 et 4.36).

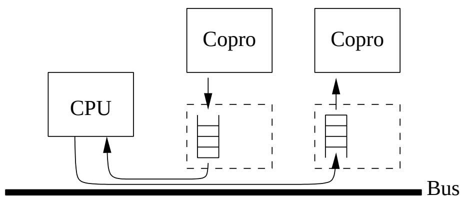



FIG. 4.35 – copie des données par le processeur

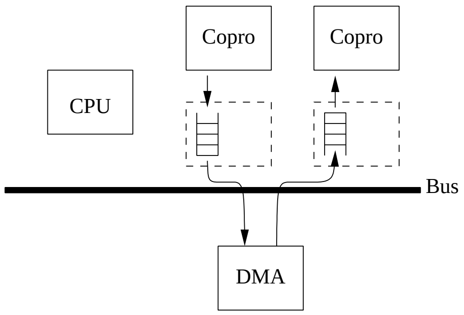



FIG. 4.36 – utilisation d'un DMA

Si la copie est assurée par le processeur ou par un contrôleur DMA, la synchronisation est faite par le processeur puisque c'est lui qui copie les données ou programme le DMA. Cette méthode est très lente si c'est le processeur qui copie les données et ne peut être utilisé que pour transférer très peu de données. Dans le cas de la programmation d'un DMA, le transfert est plus efficace, mais il double l'occupation du bus par rapport à un transfert direct entre les deux interfaces.

Si l'un des deux modules d'interface possède un accès maître sur le bus pour réaliser le transfert des données, la synchronisation consiste alors pour le maître à vérifier que la fifo du module d'interface esclave contient suffisamment de place (ou de données) pour qu'il puisse écrire (ou lire) les données qu'il souhaite échanger. Ce contrôle *a priori* peut être évité si le bus utilisé fournit un mécanisme de « reprise » : le refus d'un transfert par l'esclave stoppe le transfert jusqu'à ce que l'esclave indique au maître que le transfert peut reprendre.

Le module d'interface possédant le mode maître entre le producteur et le consommateur peut être choisi en fonction des débits des deux coprocesseurs. Il est préférable de connecter au coprocesseur ayant le plus faible débit le module d'interface en mode maître pour limiter le nombre d'arrêts/démarrages du transfert. Le coprocesseur souhaitant transmettre des données avec le plus gros débit se trouve bloqué par le protocole de communication entre le coprocesseur et le module d'interface et non par le mécanisme de synchronisation du bus.

#### 4.3.4.2 Transfert par mémoire

Dans le cas d'un transfert par mémoire, la synchronisation peut se faire à l'aide du processeur par le même mécanisme que celui des schémas SH2 et HS2. Les deux modules d'interface utilisent une interruption pour que le processeur mette à jour l'état de la fifo et reprogramme les quotas de données dans les modules.

## 4.4 Conclusion

L'analyse effectuée dans ce chapitre éclaire la démarche que nous préconisons pour la synthèse des communications : il n'y a pas à proprement parler d' « outil de synthèse » mais plutôt une « instanciation » de composants matériels et logiciels prédéfinis, décidée par le concepteur, en fonction des besoins de l'application. Il faut donc simplement associer à chaque canal du graphe de tâches un schéma de communication particulier.

Pour les schémas logiciel→matériel et matériel→logiciel, nous avons vu qu'au moins trois types de schémas sont possibles, suivant que l'on utilise une fifo matérielle locale au module d'interface ou une fifo située en mémoire semblable à celle utilisée pour une communication logiciel→logiciel, Les communications matériel→matériel peuvent être réalisées par un module d'interface en mode maître et l'autre en mode esclave en utilisant une régulation par le protocole du bus. Dans ce cas, le processeur n'est pas sollicité. Elle peuvent aussi être réalisées sous le contrôle du processeur avec ou sans utilisation d'une fifo en mémoire.

Nous pouvons résumer la liste des ressources matérielles et logicielles nécessaires aux communications que nous venons de décrire :

**pour le logiciel :**

- les tâches sont réalisées comme des *threads* POSIX,
- les tâches ne doivent communiquer entre elles que par des appels aux deux fonctions CHANNELREAD et CHANNELWRITE. Le corps de ces fonctions dépend de la nature des tâches aux deux extrémités du canal et du schéma de communication choisi,
- le gestionnaire d'interruptions doit répondre aux requêtes des interfaces matérielles en agissant comme une tâche logicielle (c'est à dire en utilisant des `sémaphores`). Ces différentes interruptions peuvent être vues comme la partie logicielle des primitives de communication lorsque la tâche est réalisée en matériel. Le gestionnaire d'interruptions ne doit jamais être bloqué,

**pour le matériel :**

- le module d'interface doit offrir coté coprocesseur un protocole permettant de bloquer le coprocesseur s'il ne peut réaliser le transfert qu'il demande (fifo vide pour une lecture ou pleine pour une écriture),
- il doit pouvoir effectuer des accès maîtres ou esclaves sur le bus,
- il doit posséder une fifo de taille variable réalisant soit la fifo du canal, soit un simple tampon afin de réaliser efficacement le transfert lorsque la fifo du canal est en mémoire,
- l'interface esclave doit :
  - permettre la visibilité de l'état de la fifo par le processeur afin qu'il puisse déterminer la place (ou les données) disponible,
  - posséder une système d'interruption programmable en fonction de la place (ou la quantité de données) disponibles dans la fifo,
- l'interface maître doit :
  - savoir adresser une fifo et un tampon circulaire,
  - posséder un quota de données à transférer programmable par logiciel et indiquer à tout instant le degré d'utilisation de ce quota,
  - pouvoir interrompre le processeur lorsque ce quota est atteint.

Les besoins ainsi définis vont servir à la spécification d'un module d'interface matériel générique permettant de supporter l'ensemble des schémas de communication retenus.

# Chapitre 5

# Architecture du module d'interface générique

Ce chapitre présente l'architecture interne du module d'interface générique permettant la communication entre un coprocesseur et le reste du système. Les services proposés par ce module découlent de l'analyse des moyens de communication qui ont été décrits dans le chapitre précédent.

Nous précisons d'abord l'environnement d'utilisation de ce module pour en définir les interfaces externes et les protocoles mis en œuvre. Du coté du coprocesseur, il s'agit du protocole vecteur décrit dans le chapitre précédent. Pour les communications avec le reste du système, nous décrirons brièvement le standard VCI.

La deuxième section décrit l'architecture interne du module d'interface composée de sous-modules que détaillent les sections suivantes.

## 5.1 Architecture externe

La figure 5.1 présente grossièrement l'architecture cible des systèmes mixtes matériel–logiciel que nous souhaitons réaliser. Cette architecture peut comporter plusieurs processeurs et plusieurs bus. Le rôle du module d'interface est de permettre à un coprocesseur matériel de communiquer avec le reste du système en déchargeant le coprocesseur de la gestion explicite du protocole de bus de communication. Ce module d'interface possède donc deux interfaces correspondant au deux types d'interlocuteurs : le bus et le coprocesseur. Du coté du coprocesseur, nous utilisons le protocole vecteur défini dans le chapitre précédent. Pour le bus, nous avons choisi d'utiliser le standard VCI, afin de permettre la réutilisation de ce module dans différents contextes. Il suffit de remplacer le *wrapper* pour s'adapter à n'importe quel bus système « on-chip ». A titre de démonstration, nous avons développé des *wrappers* pour le PI-Bus.

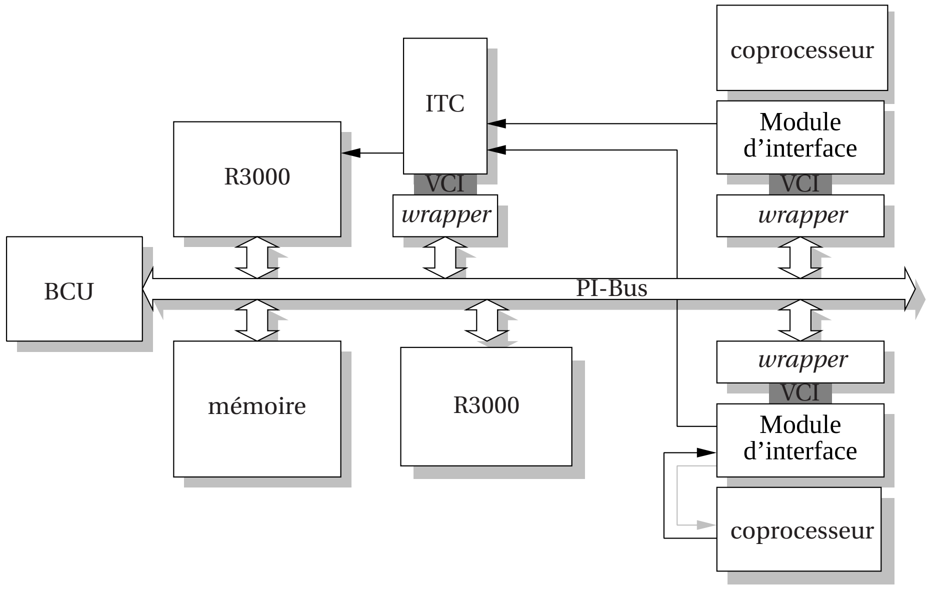



FIG. 5.1 – Architecture type de systèmes

### 5.1.1 Le protocole VCI

Le module d'interface utilise le protocole VCI (*Virtual Component Interface*) pour s'abstraire du bus utilisé [VSI98, VSI00]. Le protocole VCI définit un protocole intermédiaire entre le bus et le composant. Ceci permet au composant d'être utilisé dans des systèmes utilisant des bus différents. Il suffit pour cela de posséder un *wrapper* faisant la conversion entre VCI et le bus choisi. Ce protocole a été défini par le groupe de travail OCB du projet VSI regroupant la majeure partie des entreprises de micro-électronique et de CAO.

Le principe de VCI est que deux composants (un initiateur et une cible) communiquent entre eux à travers un espace adressable partagé comme s'ils étaient reliés par une liaison point à point (figure 5.2). À chaque transaction, l'initiateur fournit une adresse dont les poids forts sont constants et désigne la cible. Les poids faibles de l'adresse sont décodés par la cible.

VCI n'est pas un bus, mais une interface. La norme VCI définie le protocole d'échange en paires requêtes–réponses. Elle spécifie le contenu et l'encodage des requêtes et des réponses et le protocole permettant de les transférer. Ce protocole possède trois niveaux de complexité (*peripheral*, *basic* et *advanced*). Le niveau périphérique est un sous-ensemble du niveau basic pour les composants simples. Le niveau avancé ajoute des identificateurs de requêtes permettant la gestion désordonnée des requêtes. Nous utilisons le niveau basique qui est le plus courant et qui suffit à nos besoins.

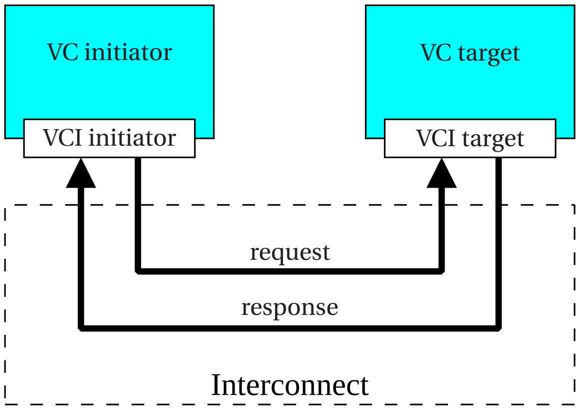



FIG. 5.2 – VCI : connexion point à point

Pour être utilisé dans un système contenant un bus, un *wrapper* est connecté à chaque composant. La figure 5.3 en montre un exemple.

Tous les composants du système n'ont pas besoin d'être VCI. Un système peut contenir des composants directement connectés au bus physique utilisé, des composants VCI reliés au bus par un *wrapper* et d'autres composants dont le protocole a été converti par un *wrapper* non-VCI pour être connectés au bus physique.

### 5.1.2 Le PI-Bus

Le PI-Bus est un standard européen (OMI 924) pour systèmes intégrés [OMI96]. Nous utilisons son extension 64 bits, définie par Philips [Kla97].

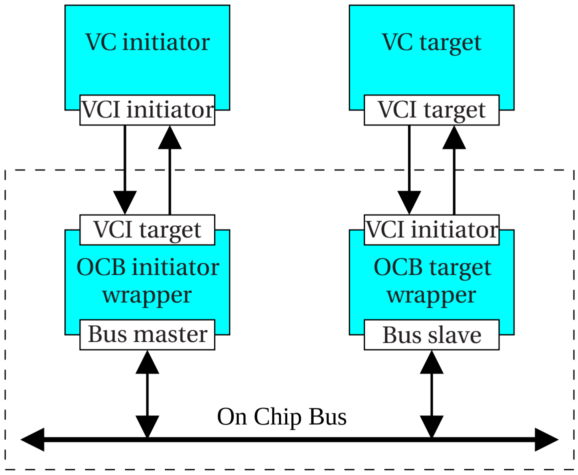



FIG. 5.3 – VCI : implémentation via un bus

Le PI-Bus (*Peripheral Interconnect Bus*) est un bus pipeliné à adresses et données dé-multiplexées (figure 5.4). La largeur des adresses est de 32 bits et celle des données de 8,16,24,32 ou 64 bits. Son débit maximum est de une donnée par cycle soit 200 Mo/s à 50 MHz pour 32 bits.

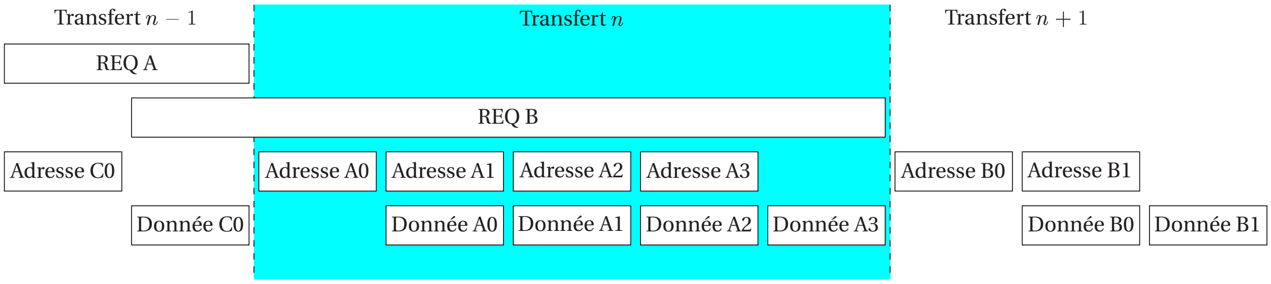



FIG. 5.4 – le pipeline du PI-Bus

Le PI-Bus différencie trois types d'agents :

– les maîtres qui initient le transfert et adressent les requêtes,
– les esclaves sélectionnés en fonction de l'adresse émise par le maître, qui répondent à ses requêtes,
– le contrôleur de bus (BCU) qui arbitre entre les maîtres pour désigner le possesseur du bus et qui sélectionne les esclaves.

Un transfert sur le PI-Bus peut se décomposer en trois phases :

– une phase de requêtes où le maître demande le bus. Cette phase ce termine lorsque le maître acquiert le bus,

– une phase d'adresse qui dure un cycle d'horloge. Le maître émet l'adresse et une opération qui contient la direction du transfert, le nombre de données à transférer et la taille de ces données parmi un ensemble prédéfini. Le contrôleur de bus décode la partie haute de l'adresse pour sélectionner l'esclave correspondant qui peut lire l'opération demandée par le maître,
– une phase de données où l'esclave répond à la requête du maître. Il lit la donnée émise par le maître dans le cas d'une écriture ou écrit la donnée sur le bus dans le cas d'une lecture. L'esclave dispose d'un code d'acquittement pour indiquer la fin ou non de la phase. Cette phase dure au minimum un cycle si l'esclave est en mesure de répondre immédiatement à la requête du maître. Le contrôleur du bus dispose d'un signal lui permettant de terminer le transfert si l'esclave tarde à répondre.

Ces trois phases s'exécutent simultanément. Un transfert est composé d'une phase de requête et d'autant de paires (adresse, donnée) qu'il y a de données à transférer.

Le PI-Bus est un bus non préemptif : le maître qui possède le bus peut le garder aussi longtemps qu'il le souhaite.

L'algorithme utilisé par le contrôleur de bus pour élire un maître parmi ceux le demandant n'est pas spécifié par la norme : il est laissé au choix du concepteur en fonction des caractéristiques du système.

Il existe un mécanisme de maître par défaut. Ce mécanisme affecte le bus à ce maître, lorsqu'aucun maître ne le veut. Ceci permet au maître par défaut d'économiser le cycle de requêtes et de passer directement au cycle d'adresse. En conséquence, le maître par défaut est le maître le plus prioritaire sur le bus.

### 5.1.3 Problèmes liés à la conversion de protocole

Si l'intérêt de l'utilisation du protocole VCI est évident pour ce qui est de la réutilisation, sa mise en œuvre pose des problèmes de pertes d'informations. L'initiateur VCI considère qu'il dispose d'une liaison point-à-point avec sa cible. Il lui envoie des paquets de requêtes contenant des informations sur le nombre de données, leur taille, la façon avec laquelle elles sont adressées, ...

Le *wrapper* maître doit transcrire le maximum d'information présentes dans le paquet requête VCI en utilisant les primitives du bus, pour que le *wrapper* esclave puisse les retranscrire du bus vers la cible VCI. Les informations que le *wrapper* esclave envoie vers la cible VCI doivent être les plus proches possibles de celles émises par l'initiateur VCI. Cependant, le protocole de bus ne peut pas forcément traduire toutes les informations présentes dans la requête VCI. Les informations manquantes ne peuvent alors pas être transmises vers la cible.

Par exemple, si deux composants 128 bits communiquent par un bus 32 bits comme le PI-Bus, le *wrapper* maître convertira un mot de 128 bits en une rafale de 4 mots de 32 bits sur le bus. Cependant, le *wrapper* esclave sera incapable de savoir s'il s'agit d'un mot de 128 bits, de 4 mots de 32 bits, ou de 2 mots de 64 bits, l'opcode du PI-Bus ne permettant pas de différencier ces trois scénarios.

Les *wrappers* doivent utiliser au maximum les informations présentes dans le paquet requête VCI pour réaliser un transfert le plus efficace possible en utilisant les primitives du bus. Par exemple, si le bus utilisé possède des adresses et des données multiplexées, le *wrapper* maître a tout intérêt à utiliser les signaux `contig` et `wrap` de VCI pour ne pas avoir à émettre l'adresse avant chaque données, ce qui permet un transfert deux fois moins coûteux. Toutefois, les ressources du bus qui n'ont pas d'équivalent dans le protocole VCI ne pourront pas forcément être utilisées par le *wrapper* maître. Par exemple, le protocole VCI ne prévoit pas de mécanisme de « snoop » qui permette de garantir la cohérence des caches par espionnage des transferts sur le bus.

Dans le protocole VCI, la transmission d'un paquet doit être atomique. Cette atomicité ne peut être garantie avec le PI-Bus puisqu'un esclave peut interrompre un transfert (le maître doit alors relâcher le bus) et que le protocole du PI-Bus ne permet pas de verrouiller une cible comme le permettent d'autres protocoles de bus comme Amba ou PCI. Deux solutions sont envisageables. La première est de considérer cela comme une erreur. Si un esclave interrompt le transfert par un wrapper maître d'un paquet de $n$ mots au $i^{\text{ième}}$ mot, le wrapper enverra $n$ réponses vers l'initiateur, les $n - i$ dernières réponses portant un code d'erreur. C'est alors à l'initiateur de ré-émettre les mots qui n'ont pas été transmis. La deuxième solution est de pas le transmettre l'interruption à l'initiateur : le wrapper maître redémarrant le transfert à l'endroit où l'esclave l'a interrompu. Nous avons choisi la deuxième solution car l'atomicité d'un paquet n'est pas utile dans notre cas puisque les composants VCI que nous utilisons servent à réaliser des canaux de communications n'ayant qu'un producteur et qu'un consommateur.

## 5.2 Architecture interne du module d'interface

Le but de ce module d'interface est de réaliser la partie matérielle de l'implémentation des canaux de communication reliés à un coprocesseur matériel.

Quatre types de service sont nécessaires à la réalisation des canaux : la fifo esclave d'entrée (du coprocesseur), la fifo esclave de sortie, la fifo maître d'entrée et la fifo maître de sortie. Un coprocesseur nécessite autant de services qu'il a de canaux. Il n'est pas raisonnable de connecter sur un bus autant de composants maître ou esclave qu'il y a de canaux reliés au coprocesseur. Nous avons donc regroupé les services nécessaires à un coprocesseur dans un module d'interface. Ceci permet de partager certains sous-modules entre les services : par exemple la cible VCI est utilisée par les fifos esclaves pour les données et par les fifos maîtres pour la configuration.

Le nombre et le type de ces canaux peuvent varier d'un coprocesseur à l'autre. L'architecture du module d'interface doit donc être modulaire pour contenir autant de canaux que nécessaire au coprocesseur, comme le montre la figure 5.5. Chaque sous-module n'est instancié que si les services utilisés par le coprocesseur l'exige.

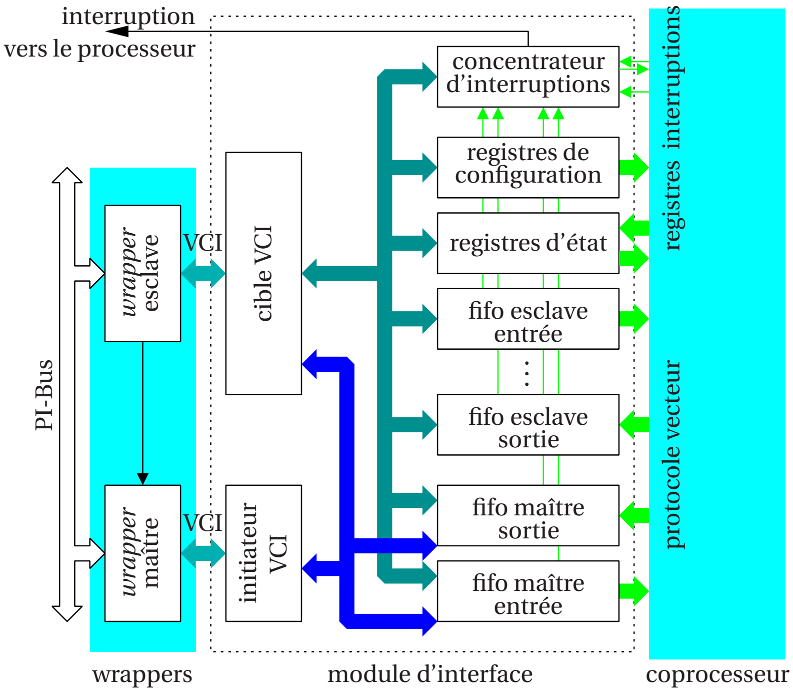



FIG. 5.5 – architecture interne du module d'interface

### 5.2.1 Décomposition en sous-module

Entre le bus physique et le module d'interface se trouvent les *wrappers* permettant l'accès au bus en mode maître et esclave. Le module d'interface comprend donc deux sous-modules à interface VCI. Le premier est une cible VCI qui communique avec le *wrapper* esclave sur le bus. Il est utilisé pour accéder à toutes les ressources adressables du module d'interface : les fifos esclaves et tous les registres de configurations et d'états des sous-modules. Le second est un initiateur VCI. Il communique avec le *wrapper* maître sur le bus. Il est utilisé par les fifos maîtres pour émettre des requêtes vers un esclave sur le bus.

Sept types de sous-modules fonctionnels peuvent être instanciés. Chaque sous-module correspond à un service spécifique. Si un service n'est pas utilisé le sous-module correspondant n'est pas instancié lors de la génération du module d'interface. Il s'agit donc d'un générateur de module d'interface. Ces sept sous-modules sont :

- un module fifo esclave d'entrée utilisé dans les schémas de communication pour réaliser la fifo du canal lorsque le coprocesseur en est le consommateur,
- un module fifo esclave de sortie utilisé dans les schémas de communication pour réaliser la fifo du canal lorsque le coprocesseur en est le producteur,
- un module fifo maître d'entrée permettant au coprocesseur de lire, par un accès en mode maître sur le bus, soit un tampon circulaire en mémoire, soit une fifo esclave dans un autre module d'interface,
- un module fifo maître de sortie permettant au coprocesseur d'écrire, par un accès en mode maître sur le bus, soit un tampon circulaire en mémoire, soit une fifo esclave dans un autre module d'interface,
- un concentrateur d'interruptions vectorisées regroupant les interruptions émise par les modules fifo du module d'interface et des interruptions externes permettant de n'utiliser qu'une ligne d'interruption par module d'interface.
- un ensemble de registres de configuration. Ces registres sont écrits par le processeur via le bus et lus par le coprocesseur sur son interface. Ils ne sont pas utilisés dans les schémas de communication, mais ils peuvent être utilisés pour paramétrer le coprocesseur à l'initialisation du système,

– un ensemble de registres d'état. Ces registres sont lisibles par le processeur. Ils ne sont inscriptibles que par le coprocesseur. Ils sont utilisés pour connaître l'état du coprocesseur.

Les sections suivantes décrivent chacun des sous-modules et des *wrappers*.

### 5.2.2 Autorégulation

Le § 4.3.4 du chapitre précédent fait apparaître la nécessité d'un mécanisme d'autorégulation entre deux modules d'interfaces communicant. Ce mécanisme fait appel à un type de transfert particulier sur le bus, le *split* qui permet à un esclave d'interrompre et transfert pour le redémarrer plus tard. Contrairement à d'autres bus comme Amba [ARM99], le PI-Bus ne possède pas un tel mode de transfert et nous avons donc été amené à en spécifier un.

Le principe de cette autorégulation est de permettre à une fifo esclave ne pouvant pas satisfaire la demande d'un maître (écriture dans une fifo pleine ou lecture dans une fifo vide) de stopper le transfert et de le reprendre lorsque la demande devient possible.

Pour cela, à la demande de la fifo esclave, la cible VCI envoie un code d'erreur au *wrapper* esclave. Ce code d'erreur demande au *wrapper* d'émettre l'acquittement *split* sur le PI-Bus qui libère le bus et bloque le maître effectuant le transfert. Lorsque la fifo esclave peut satisfaire la demande du maître, elle émet une requête à l'initiateur VCI d'écriture à une adresse spéciale du module d'interface, qui la transmet au *wrapper* maître. Le *wrapper* esclave du module d'interface destinataire lisant l'adresse de la requête ne la transmet pas à la cible VCI, mais active le signal `wakeup` qui débloque le *wrapper* maître afin qu'il réitère le transfert qui l'a bloqué.

Ce mécanisme nécessite que la fifo esclave puisse faire des requêtes à l'initiateur VCI, qu'elle connaisse l'adresse du module d'interface communicant avec elle et qu'un signal joigne les deux *wrappers* d'un module d'interface.

La figure 5.6 illustre ce mécanisme dans le cas où une fifo maître transfère des données dans une fifo esclave. Le cheminement des données est représenté en grisé, celui de la requête de réveil est en pointillé.

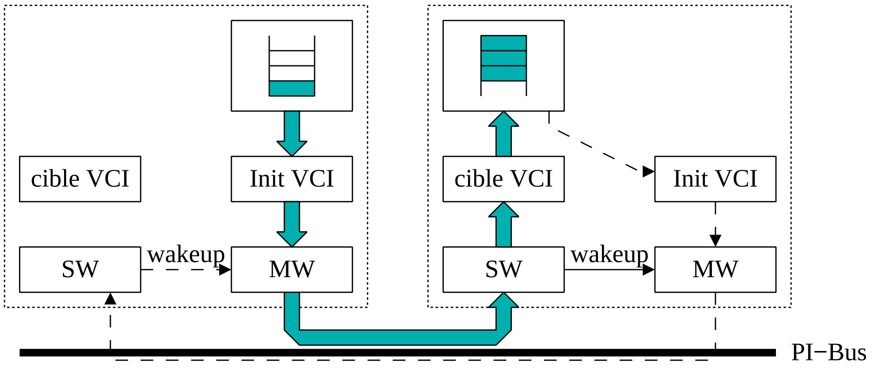



FIG. 5.6 – Mécanisme d'autorégulation

## 5.3 Le module VCI initiateur

Le rôle du module VCI initiateur est de traduire les requêtes faites par les modules fifo maître en requêtes VCI et de leurs transmettre les réponses. Il reçoit également les requêtes de réveil émises par les fifos esclaves. Son rôle est essentiellement un rôle d'arbitre puisque le protocole utilisé par les fifos maîtres est le protocole VCI. Le mécanisme d'arbitrage utilisé est à priorité fixe : la fifo maître d'entrée est plus prioritaire que la fifo maître de sortie. Les requêtes des fifos esclaves sont les moins prioritaires.

## 5.4 Le module VCI cible

Le module VCI cible reçoit les requêtes VCI, décode l'adresse pour sélectionner le sous-module et lui transmet la requête VCI.

Deux types de ressources sont adressables : les registres qui peuvent toujours répondre à la requête et les fifos susceptible de ne pas pouvoir satisfaire la requête si, par exemple dans le cas d'une écriture, la fifo est pleine.

Dans le premier cas, le module VCI cible peut émettre le paquet réponse sans demander confirmation au sous-module, le transfert est accepté.

Dans le second cas, le module VCI cible pour pouvoir répondre, doit d'abord demander au sous-module si il peut satisfaire la demande. Ceci nécessite un décodage de l'adresse pour sélectionner le module fifo esclave parmi ceux du module d'interface pour lire l'état de la fifo correspondante. Ceci constitue une chaîne combinatoire trop longue pour être évaluée dans le cycle d'horloge dont dispose le *wrapper* et le module VCI cible pour générer l'acquittement à partir d'une sélection sur le PI-Bus.

Pour pouvoir accepter des lectures ou des écritures en rafale dans les fifos, le module VCI cible dispose d'un cache contenant l'état de la dernière adresse accédée. Pour cela, le module mémorise la dernière adresse accédée et l'état de la fifo correspondant à cette adresse. Si l'adresse demandée par le *wrapper* esclave est l'adresse mémorisée, la cible émet la réponse à partir de l'état qu'elle a mémorisé, sinon, elle émet un état d'attente, mémorise la nouvelle adresse et l'état correspondant pour pouvoir répondre au cycle suivant. Ce mécanisme permet de supporter des accès en rafale pour la lecture ou l'écriture de la même fifo.

## 5.5 Le module fifo esclave d'entrée

La figure 5.7 détaille l'architecture du sous-module fifo esclave d'entrée.

Il se compose d'un contrôleur pour le protocole vecteur convertissant celui-ci en protocole fifo, d'un sérialisateur transformant les données à la taille du bus vers la taille du coprocesseur, d'une fifo vue par le module VCI comme une adresse unique et de cinq registres :

- un registre d'état (`status`), en lecture seule, donnant le nombre de cases pleines de la fifo,
- un registre `clear` dont l'écriture entraîne la mise à zéro de la fifo,
- un registre `threshold`, optionnel, contenant un niveau programmable de remplissage de la fifo, en deçà duquel une requête d'interruption est émise vers le concentrateur d'interruptions,
- les deux registres nécessaires au protocole *split* : l'adresse du maître écrivant dans la fifo (`splitAddress`) et le drapeau autorisant l'usage du *split* (`splitMode`).

Le contrôleur pour le protocole vecteur peut, optionnellement, émettre une requête d'interruption vers le concentrateur d'interruptions lorsqu'une requête de lecture est émise par le coprocesseur (lorsque le signal `req` est valide).

Les paramètres de ce sous-module sont :

- la profondeur de la fifo,

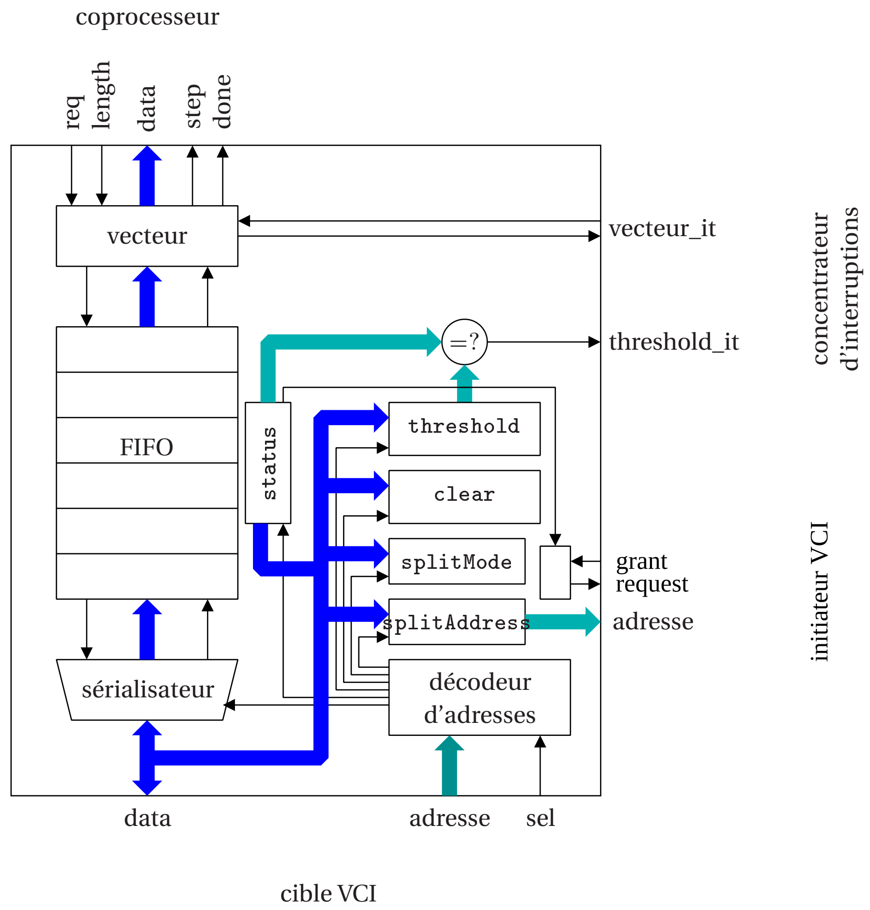



FIG. 5.7 – fifo esclave d'entrée

- la largeur du bus de données du coprocesseur,
- la présence ou non du registre `threshold`,
- la possibilité du contrôleur pour le protocole vecteur d'émettre une requête d'interruption,
- la présence ou non des registres de split.

La fifo esclave d'entrée est utilisé par les schémas de communication matériel→logiciel avec la fifo dans le module d'interface et matériel→matériel. Le registre d'état est utilisé par le processeur pour connaître le nombre de données qu'il peut écrire dans la fifo. Le registre `threshold` et la requête d'interruption qui lui est associée sont utilisés dans ce schéma pour réveiller la tâche productrice lorsque suffisamment de place est disponible dans la fifo : le processeur programme le registre pour que l'interruption soit émise lorsque le nombre de cases vides dont il a besoin est disponible si la fifo contient suffisamment de cases et sinon, lorsque toute la fifo est vide.

Le registre `clear` n'est pas utilisé dans les schémas de communications que nous avons présenter dans le chapitre précédent, puisque la sémantique des réseaux de Kahn que nous utilisons ne permet pas de vider un canal de communication. Cependant, ce registre peut être utilisé à l'initialisation des communications dans le cas, par exemple, d'un graphe dynamique.

## 5.6 Le module fifo esclave de sortie

Comme le montre la figure 5.8, son architecture est la même que celle de la fifo esclave d'entrée. Le sens du bus de données est inversé de sorte que l'adresse des données de la fifo permet de la lire et le coprocesseur peut écrire dans la fifo par le protocole vecteur.

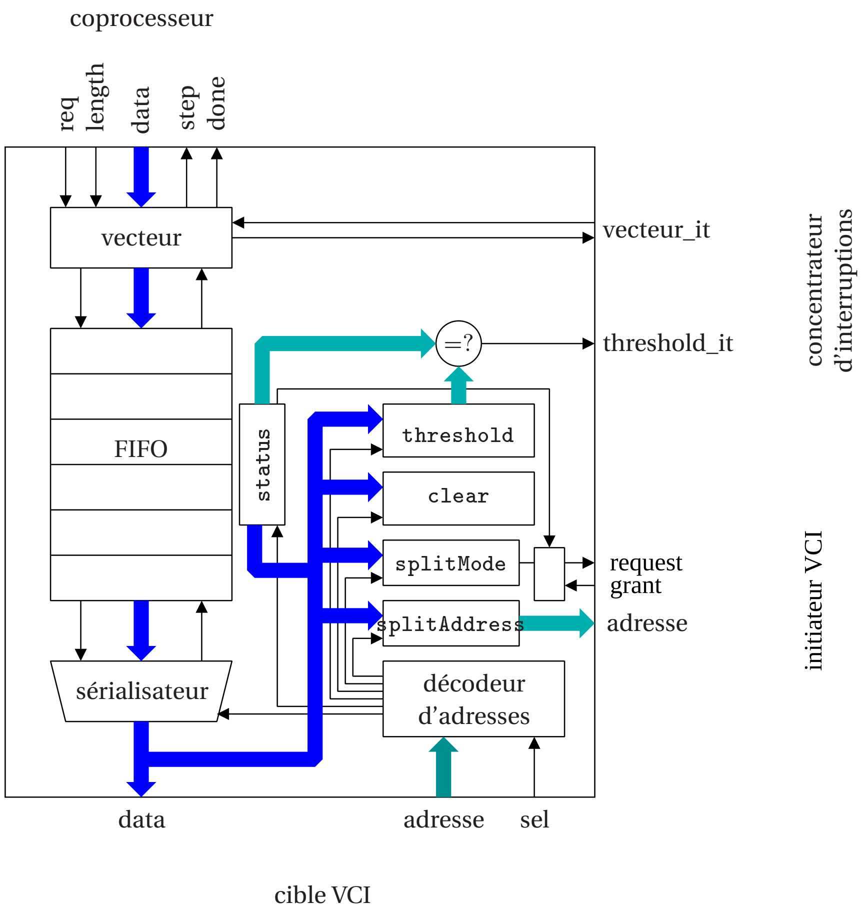



FIG. 5.8 – fifo esclave de sortie

Ce sous-module est utilisé dans le schéma de communication matériel→logiciel avec la fifo dans le module d'interface. Ce schéma utilise les ressources de ce sous-module de la même manière que le schéma logiciel→matériel utilise la fifo esclave d'entrée.

## 5.7 Le module fifo maître d'entrée

Le module fifo maître est semblable à la fifo esclave et en possède toutes les ressources (`status`, `threshold`, `clear`). Il ne possède pas de sérialisateur et les données du coprocesseur doivent être de la taille du bus VCI. Il diffère par deux aspects : les données de la fifo ne sont pas inscriptibles en utilisant le mode esclave du bus, le module utilise un accès maître sur le bus pour lire les données et remplir la fifo.

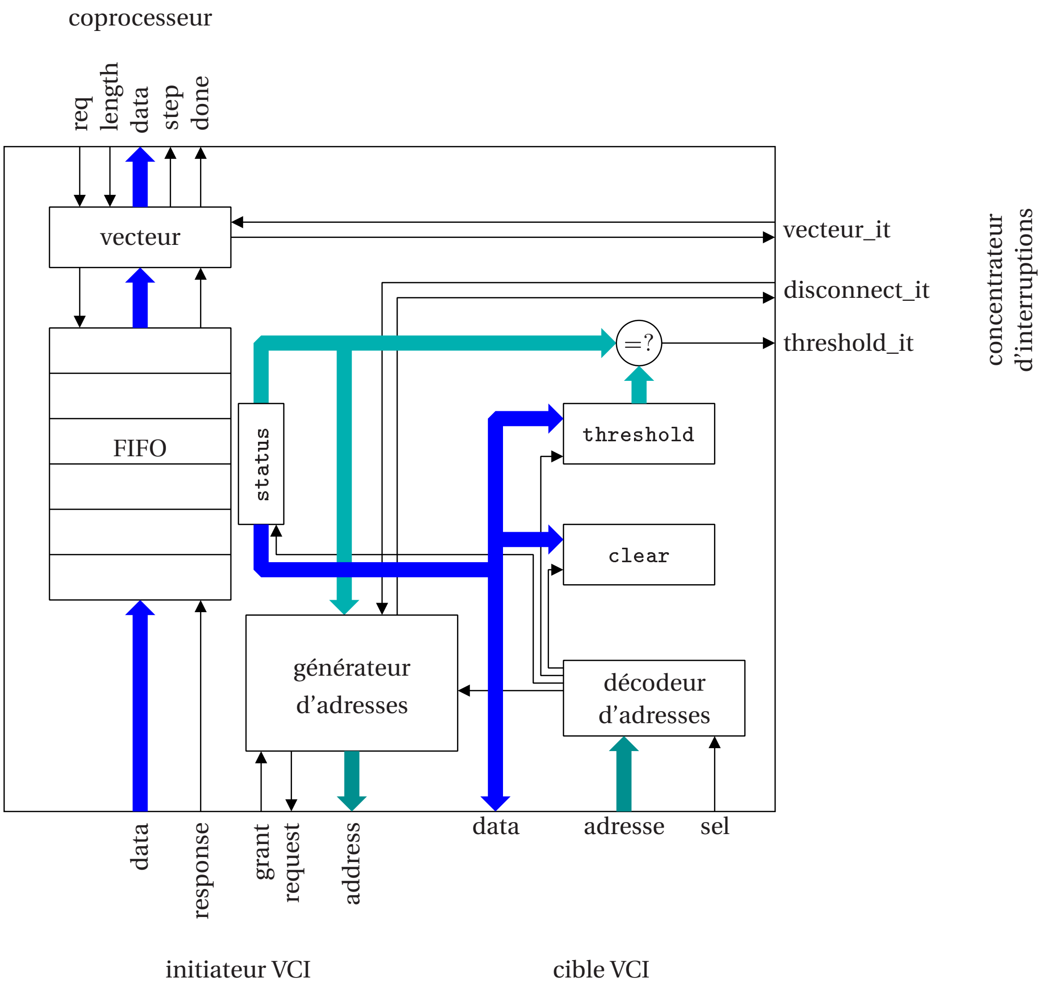



FIG. 5.9 – fifo maître d'entrée

Pour pouvoir réaliser cet accès en mode maître, le module doit connaître les adresses auxquelles il peut lire les données. Ces adresses sont issues d'un générateur paramétrable. Ce générateur, illustré par la figure 5.10, permet d'adresser des tampons continus ou circulaires avec un pas programmable. Ces différents adressages permettent d'adresser les fifo esclaves d'autres modules d'interface et les tampons circulaires utilisés par les schémas de communication.

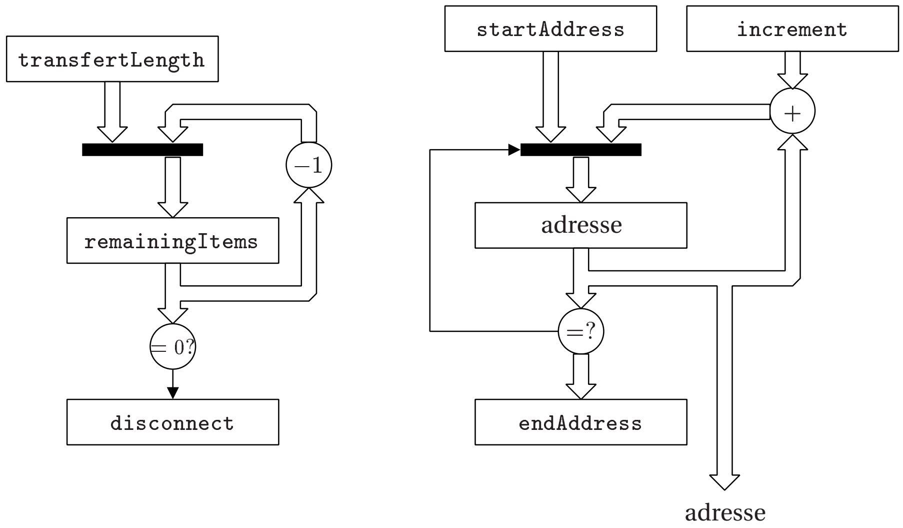



FIG. 5.10 – le générateur d'adresses

Les ressources permettant de programmer le générateur sont les suivantes :

- le registre `startAddr` contient l'adresse de la première requête,
- le registre `endAddr` contient l'adresse de la fin du tampon circulaire si le mode choisi pour la génération d'adresses est circulaire,
- le registre `transfertLength` contient la taille du transfert lorsque le mode de transfert est borné. Une requête d'interruption peut être émise vers le concentrateur d'interruption pour signaler la fin du transfert au processeur,
- un registre `mode` possédant deux champs indépendants :
  - le premier détermine si le tampon que l'on souhaite adresser est circulaire ou non. C'est à dire, si le générateur après avoir émis une donnée à l'adresse contenue dans le registre `endAddr` en émettra une à l'adresse contenue dans le registre `startAddr` ou s'il continuera en ignorant la valeur du registre `endAddr`,
  - le deuxième registre détermine si le transfert est ou non borné. Si c'est le cas, le registre `transfertLength` contient le nombre de données à transférer.
- le registre `increment` contient la valeur à additionner à l'adresse courante pour déterminer l'adresse suivante. Ce registre vaudra, par exemple, 0 pour accéder aux données d'une

fifo esclave d'un module d'interface et 4 pour un tableau d'entiers 32 bits en mémoire.

- le registre `burstLength` contient la taille d'un paquet sur le bus. Le sous-module n'émettra de requête que si `burstLength` cases sont vides dans la fifo. Il ne fera des paquet que de `burstLength` sauf si le mode est borné et qu'il reste moins de `burstLength` données à transférer,
- le registre `disconnect` est un booléen qui autorise (faux) ou interdit (vrai) l'émission de requêtes par le sous-module vers le bus. Il vaut vrai à l'initialisation pour inhiber le sous-module tant qu'il n'a pas été programmé par le processeur. Ce registre est positionné à vrai par le matériel lorsque le mode est borné et que `transfertLength` données ont été transférées,
- le registre `remainingItems` est l'unique registre en lecture seule du générateur d'adresses. Il contient le nombre de données restant à transférer pour atteindre `transfertLength` lorsque le mode est borné.

Les paramètres de la fifo maître d'entrée sont :

- la profondeur de la fifo,
- la largeur du bus de données du coprocesseur,
- la présence ou non du registre `threshold`,
- la possibilité du contrôleur d'émettre une requête d'interruption pour le protocole vecteur,
- la possibilité pour le générateur d'adresse d'émettre une requête d'interruption lorsque le transfert est terminé.

## 5.8 Le module fifo maître de sortie

Le module fifo maître de sortie est semblable à celui d'entrée à l'exception du sens des données : elles sont émises par le coprocesseur dans la fifo, et de la fifo vers le bus. Le registre `burstLength` ne donne pas la taille des paquets sur le bus, mais la taille maximum des paquets puisque une requête d'écriture est émise dès qu'une donnée est présente dans la fifo.

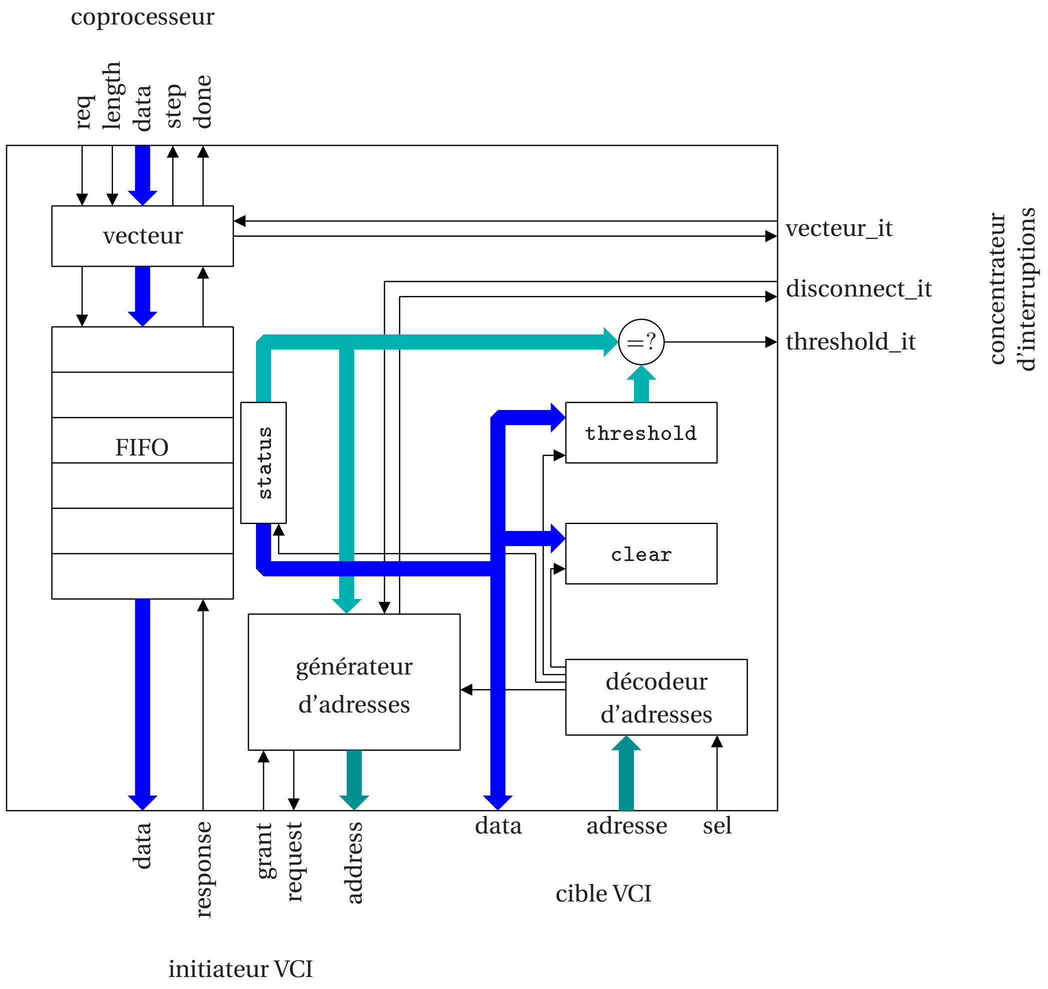



FIG. 5.11 – fifo maître de sortie

## 5.9 Le module concentrateur d'interruptions

Le rôle du concentrateur d'interruptions est de recevoir les requêtes d'interruptions internes au module d'interface et externes provenant du coprocesseur pour n'émettre qu'une requête vers le processeur. Pour chaque requête d'interruption du concentrateur existe un vecteur permettant au processeur de déterminer la source de l'interruption.

Ce concentrateur est cascadable et la requête d'interruption émise par un concentrateur d'interruption n'est pas forcément reliée au processeur, mais peut être reliée à un autre concentrateur d'interruption. Un structure hierarchique de requêtes d'interruptions peut ainsi être créé.

Le concentrateur d'interruptions peut posséder jusqu'à 32 lignes de requêtes d'interruption et autant de lignes d'acquittement. C'est à l'émetteur de maintenir l'interruption en maintenant

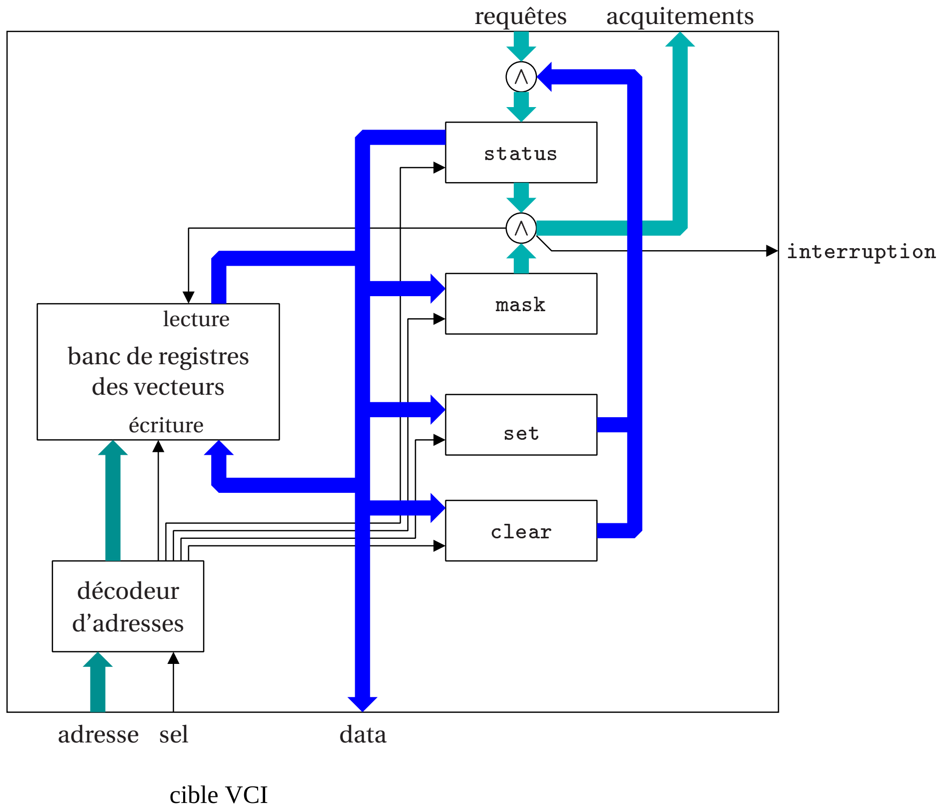



FIG. 5.12 – concentrateur d'interruptions

la requête active et la ligne d'acquittement permet de signaler à l'émetteur que l'interruption est traitée par le processeur ce qui lui permet de relâcher la requête.

Chaque requête peut être masquée indépendamment par un bit du registre `mask`. L'état de chaque ligne d'interruption est lisible par le processeur dans le registre `status`. Une priorité fixe est associée à chaque requête : la ligne 0 correspond au bit de poids faible des registres `status` et `mask` est la plus prioritaire. La ligne 31 est la moins prioritaire. Le registre `mask` peut être modifié soit par l'écriture directe de sa valeur, soit en écrivant aux adresses de deux autres ressources : `setMask` et `clearMask` qui, respectivement, masque et démasque les interruptions dont les bits correspondant à la donnée écrite sont à 1. Ces deux ressources permettent une modification atomique du registre `mask`. Cette atomicité est nécessaire dans le contexte multi-tâches d'utilisation du module d'interface. En effet, un module d'interface peut être utilisé pour la réalisation de plusieurs canaux et le registre `mask` modifié par plusieurs tâches.

À chaque requête est associé un vecteur d'interruption. Un vecteur est un mot de 32 bits initialisé par le processeur au démarrage. Il permet au processeur d'identifier parmi les 32 requêtes possibles simultanément, celle qu'il doit gérer. Pour cela, lorsqu'il reçoit une interruption, il lit à une adresse du concentrateur indépendante de la requête. Le concentrateur lui donne le vecteur correspondant à la requête non masquée la plus prioritaire. Si la requête est relâchée, un vecteur par défaut est émis.

## 5.10 Le module des registres de configuration

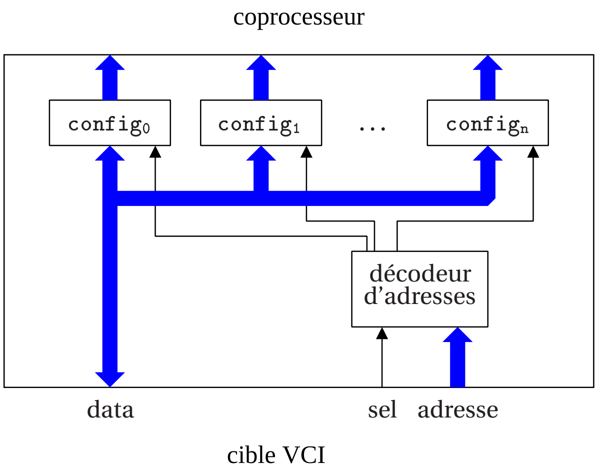



FIG. 5.13 – registres de configuration

Ces registres peuvent être lus et écrits par le processeur. Ils peuvent servir à programmer un coprocesseur. Ils ne servent pas aux schémas de communication, mais peuvent être utiles dans d'autres contextes. Ils peuvent servir notamment à spécifier des variables globales d'un système, comme la taille d'une image.

La valeur de ces registres est accessible comme un nappe de fils, ce qui implique que le processeur peut les modifier sans le notifier au coprocesseur.

## 5.11 Le module des registres d'état

Ces registres peuvent être lus par le processeur. Ils ne possèdent pas de commande pour le coprocesseur. Ils recopient à chaque cycle la valeur présente sur les nappes de fils provenant du

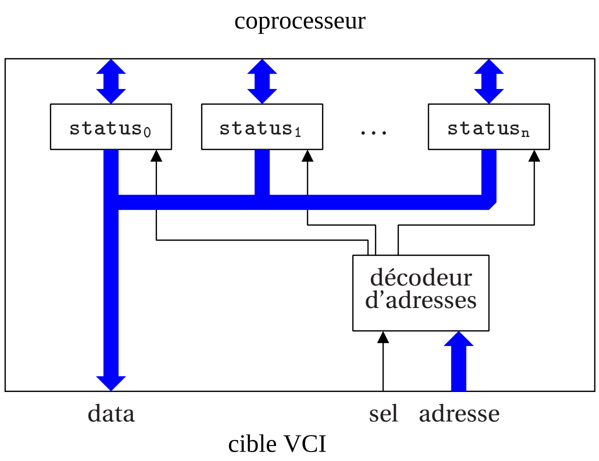



FIG. 5.14 – registres d'état

coprocesseur. Il permettent de refléter l'état d'un coprocesseur à un instant donné, comme par exemple un code d'erreur lors d'un dysfonctionnement.

Ces registres ne sont pas utilisés dans les schémas car ils ne respectent pas la sémantique des réseaux de Kahn, mais ils peuvent être utiles dans d'autres contextes.

## 5.12 Le *wrapper* maître

Le *wrapper* maître est une cible VCI et un maître sur le PI-Bus. Il reçoit les requêtes provenant d'un initiateur VCI et les convertit en requêtes sur le PI-Bus. Son architecture est illustrée par la figure 5.15.

Il se compose d'un automate de contrôle du protocole du PI-Bus et d'une fifo lui permettant de gérer les aléas du bus (ré-émission de l'adresse lorsque l'esclave introduit des cycles d'attente au cours d'une rafale et reprise du transfert si l'esclave le demande). Les requêtes sont traduites de VCI vers le PI-Bus à l'entrée de la fifo.

Ce *wrapper* n'est pas capable de gérer plusieurs requêtes simultanées comme le permet le standard VCI. Ceci n'est pas nécessaire puisque le bus ne le supporte pas. Un paquet de longueur supérieur à un se traduit en une rafale sur le PI-Bus.

Les informations concernant l'arrangement des adresses (continues, circulaires ou constantes) ne sont pas converties puisque le PI-Bus ne les supporte pas.

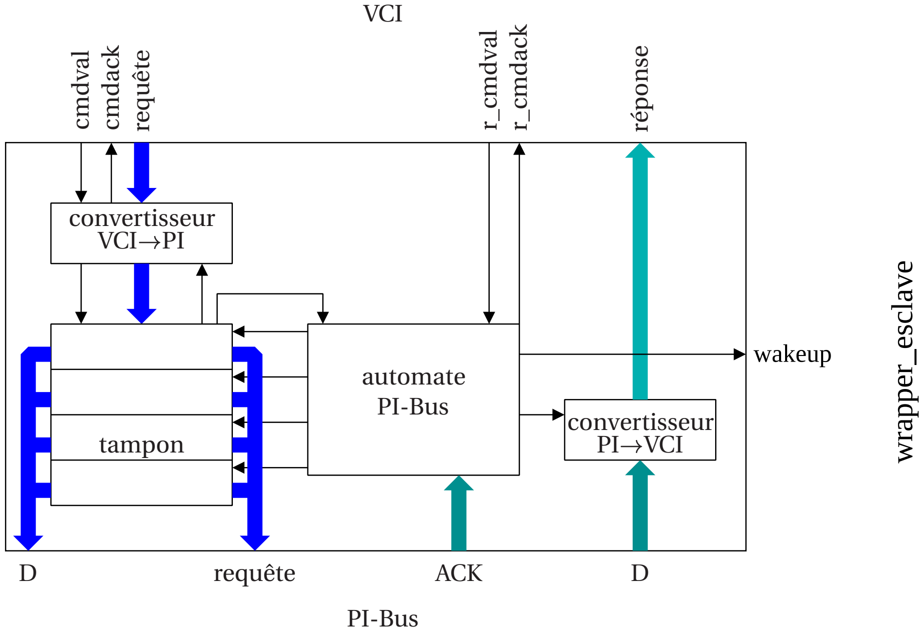



FIG. 5.15 – architecture interne du *wrapper* maître

Afin d'éviter toutes ambiguïtés sur la largeur des données transférées dont nous avons parlé au § 5.1.3, la largeur des mots de donnée sur les interfaces VCI et PI-Bus doivent être égales.

Pour les réponses, le *wrapper* suppose que l'initiateur est prêt à les recevoir dès qu'elles sont émises comme le conseille la norme VCI. Ceci permet d'éviter un tampon pour les réponses.

## 5.13 Le *wrapper* esclave

Le *wrapper* esclave est un initiateur VCI et un esclave sur le PI-Bus. Il reçoit des requêtes provenant d'un maître PI-Bus et les convertit en requêtes VCI. Son travail est de constituer des paquets VCI à partir des requêtes qui lui sont faites par les maîtres sur le PI-Bus.

Le protocole PI-Bus ne fournit pas au *wrapper* esclave suffisamment d'informations pour constituer des paquets de plus d'un élément. Pour chaque sélection sur le PI-Bus, le *wrapper* esclave construit donc un paquet requête de longueur un.

Dans le protocole PI-Bus, les données et les adresses sont décalées d'un cycle d'horloge qu'il s'agisse d'une lecture ou d'une écriture. Dans le protocole VCI, les adresses et les données des écritures doivent être émises simultanément. Un tampon permet cette synchronisation.

Pour obtenir le débit maximum sur le PI-Bus, le *wrapper* esclave dispose d'un cycle à partir de l'activation du signal de sélection par le contrôleur de bus pour fournir ou stocker la donnée. Ceci n'est pas problématique en écriture puisque le *wrapper* peut mémoriser les requêtes. Pour la lecture, cela signifie que le *wrapper* dispose d'un cycle pour émettre la requête VCI, recevoir la réponse et émettre la donnée. Pour cela, la lecture est essentiellement combinatoire dans le *wrapper* . Il utilise le positionnement par défaut de l'acquittement de la cible VCI pour émettre la requête et le positionnement par défaut de l'acquittement de l'initiateur pour attendre la réponse.

## 5.14 Conclusion

Nous avons conçus et développés un modèle synthétisable précis au cycle du modules d'interface que nous venons de décrire. Ce module possède une interface simple avec le coprocesseur compatible avec la sémantique des primitives de communication utilisées pour décrire le coprocesseur.

Il intègre les ressources matérielles nécessaire à la réalisation des schémas de communication que nous avons décrit au chapitre précédent ainsi que ceux utilisés dans le cadre du projet COSY.

Son interface VCI lui permet d'être connecté vers les bus utilisés pour la réalisation de système embarqué sur un seul circuit et vers d'autres moyens d'interconnection comme le réseau SPIN.

Il intègre un mécanisme de communication optimisé lui permettant de réaliser une communication matériel–matériel entre un module maître et un module esclave en minimisant la bande passante du bus utilisé pour gérer les contentions.

Pour permettre son développement et son évaluation nous avons développé deux *wrappers* maître et esclave permettant l'utilisation du module pour la réalisation de systèmes basés sur le PI-Bus.

Cette évaluation a conduit à l'étude de l'architecture d'un système *multi-jpeg* qui sera décrit dans le chapitre 7.

# Chapitre 6

# Simulation « au cycle près » de systèmes mixtes matériel/logiciel

Le but de ce chapitre est premièrement de présenter une approche optimisée de modélisation précise au cycle des composants d'un système intégré, et deuxièmement de détailler l'outil de simulation CASS permettant d'évaluer les systèmes décrits en interconnectant ces modèles.

Nous précisons d'abord le type de systèmes que nous souhaitons modéliser. Nous présentons ensuite des méthodes existantes pour la simulation de ces systèmes. Puis, nous décrivons la modélisation en langage C et la méthode d'évaluation des modèles que nous proposons. Nous décrivons une méthode permettant d'utiliser comme modèles de simulation les tâches de la spécification initiale qu'on souhaite réaliser comme des coprocesseurs. Cette méthode permet d'évaluer précisément les communications matérielles–logicielles avant synthèse des coprocesseurs, c'est à dire sans que l'architecture interne des coprocesseurs ne soit définie. Enfin, nous présentons les moyens d'instrumentation permettant la mise au point et l'évaluation des systèmes.

## 6.1 Contexte

Les systèmes que nous cherchons à modéliser sont constitués de modules comme des processeurs, des coprocesseurs, des mémoires où des passerelles, dont l'interface est précise au bit près. Ces composants sont interconnectés par des signaux.

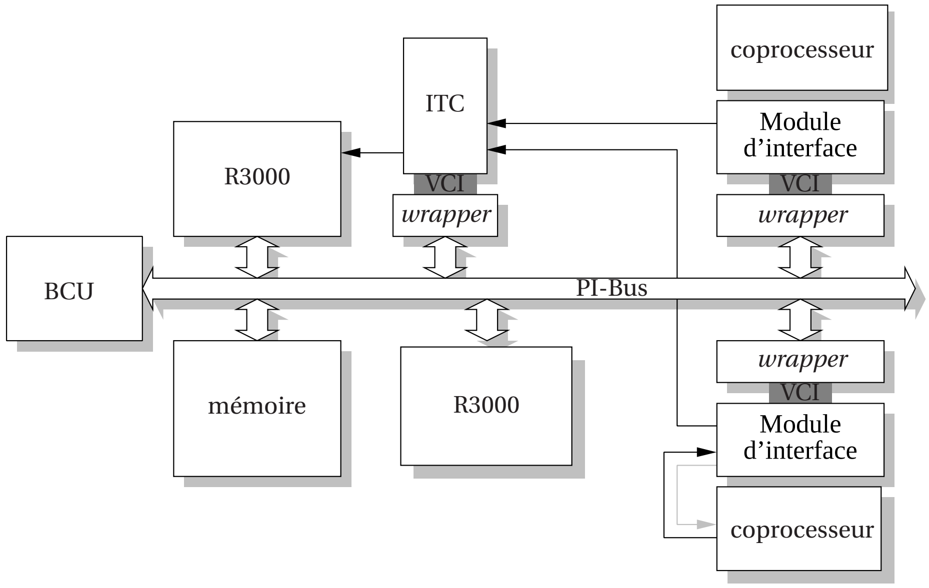



FIG. 6.1 – Exemple d'architecture de système

Ces systèmes sont construits à partir de la spécification parallèle. Ils possèdent donc un ou plusieurs processeurs exécutant la partie de l'application qui est logicielle et de coprocesseurs matériels. Chaque coprocesseur matériel communique avec le reste du système à travers un module d'interface tel que celui décrit au chapitre 5. La figure 6.1 montre le type de système que nous cherchons à simuler.

Le comportement d'un module au niveau cycle peut être modélisé comme un automate d'état synchrone [Hoa85, GVNG94]. Simuler un système au niveau cycle revient donc à simuler un ensemble d'automates synchrones communicant par des signaux. La figure 6.2 représente les automates du système de la figure 6.1.

En tirant parti du modèle des automates synchrones, nous allons développer une technique de simulation précise au cycle suffisamment performante pour permettre l'exécution de systèmes mixtes matériel–logiciel complets. La vitesse de simulation que nous cherchons à at-

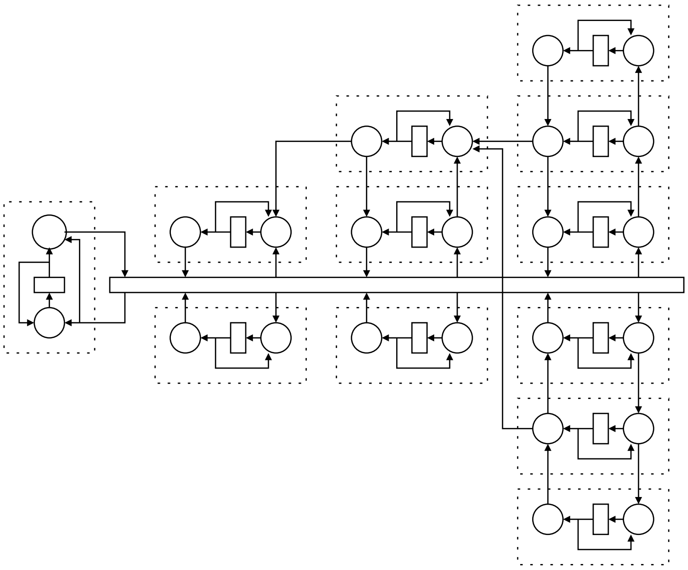


FIG. 6.2 – Automates d'un système

teindre est de l'ordre de 100000 cycles par seconde.

## 6.2 Approches existantes

Ce paragraphe présente les techniques utilisés pour la simulation de systèmes. Nous détaillons d'abord les algorithmes de simulation utilisés. Puis, nous présentons des langages permettant de modéliser les composants ainsi que des outils existants.

### 6.2.1 Algorithmes de simulation

Le matériel est intrinsèquement parallèle, en conséquence un circuit intégré peut être décrit comme un ensemble de processus parallèles communicant par des signaux. Les processus sont généralement décrits comme une suite d'instructions séquentielles. La granularité de ces processus est très variable. Cela peut être une simple assignation concurrente ou un bloc séquentiel beaucoup plus complexe. La fonction première d'un algorithme de simulation est donc d'exécuter ces processus sur une machine intrinsèquement séquentielle comme s'ils étaient exécutés en parallèle. Pour cela, deux algorithmes sont principalement utilisés : la relaxation et le pilotage événementiel.

#### 6.2.1.1 l'algorithme de relaxation

La relaxation est l'algorithme le plus simple, il consiste à évaluer l'ensemble des processus jusqu'à stabilisation des signaux, c'est-à-dire lorsqu'aucun signal ne change de valeur entre deux évaluations successives. Cet algorithme peut être accéléré en ordonnant les processus en fonction des dépendances de données qu'ils ont entre eux. Il faut alors évaluer d'abord le processus qui produit les données et ensuite celui qui les consomme. Mais l'existence de boucles combinatoires ne permet généralement pas de déterminer une ordre statique à la compilation. Le nombre d'itérations nécessaire pour obtenir la stabilisation est cependant moins important.

#### 6.2.1.2 le pilotage événementiel

Le pilotage événementiel ou *event-driven*, qui a été popularisé par la norme VHDL, a pour but de minimiser le nombre d'évaluations des processus en n'évaluant que ceux qui doivent l'être. Pour cela, à chaque processus est associé une liste de sensibilité. Cette liste est composée des signaux sur lesquels un évènement nécessite l'évaluation du processus. Pour cela, le pilotage événementiel se découpe en deux étapes : *execute* et *update*.

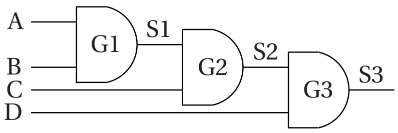
L'étape *execute*, exécute les processus qui doivent être évalués en fonction des évènements sur les signaux. Les processus sont évalués séquentiellement, cependant l'ordre de l'évaluation des processus n'influe pas sur le résultat puisque la modification des signaux est postée dans un échéancier.

L'étape *update* fait la mise à jour des signaux à partir de l'échéancier et détecte les évènements qui ont lieu à la date courante.

Par rapport à la relaxation, cet algorithme réduit le nombre d'évaluations de chaque processus nécessaire pour atteindre la stabilité car seuls les processus pour lesquels au moins un signal de la liste de sensibilité a été modifié sont réévalués. Cependant plusieurs facteurs réduisent son efficacité :

– la gestion de l'échéancier et des listes de sensibilité induit un sur-coût d'autant plus important que l'évaluation des processus ne demande que peu d'instructions,
– tous les événements générés par les composants actifs ne sont pas nécessaire pour produire des sorties. La figure 6.3 donne un exemple de circuits générant des événements

inutiles.




| temps | A | B | C | D | S1 | S2 | S3 | portes à évaluer |
|---|---|---|---|---|---|---|---|---|
| Init | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | G1 G2 G3 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | G2 |
| 2 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | G3 |
| 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |

FIG. 6.3 – Exemple de circuits générant des événements inutiles

Ainsi, les évaluations de G2 et G3 au temps 0 ne sont pas utiles. Des techniques d'ordonnancement peuvent être utilisées pour supprimer ces événements qui nous sont pas utiles pour une simulation zéro-délai [WM90]. Il s'agit d'ordonner les évaluations de processus de façon à ce qu'elles suivent la propagation des signaux. Dans l'exemple, l'évaluation de G1 puis G2 et G3 donne le résultat correct avec un nombre d'itération optimal. Cette technique d'ordonnancement ne peut être appliquée aux circuits comprenant des boucles combinatoires, comme celui de la figure 6.4.

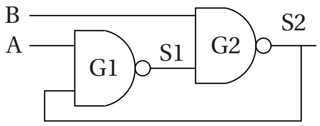



FIG. 6.4 – Exemple de circuits possédant une boucle combinatoire

Z. Wang propose dans [WM90] une solution consistant à ajouter une boucle de relaxation locale pour simuler efficacement ce genre de circuit. Il définit un composant fortement connecté comme un ensemble de composants dont tous les signaux sont utilisés comme entrées et comme sorties de portes. Dans l'exemple de la figure 6.4, S1 est la sortie de G1 et l'entrée de G2, et S2 est la sortie de G2 et l'entrée de G1. Il ordonne le composant fortement connecté en ignorant la boucle et il ajoute une boucle pour stabiliser le composant. Le nombre d'itérations est pré-calculé et si un composant comporte $m$ boucles, alors la limite est $2^m + 1$. La figure 6.5 donne le code de la boucle utilisée pour simuler l'exemple de la figure 6.4.

Par définition, deux composants fortement connectés ne peuvent être imbriqués. Cepen-

COMPONENTLOOP()

$count \leftarrow 0$
**do**
  $x0 \leftarrow Y2$
  $Y1 \leftarrow \neg(A \wedge x0)$
  $Y2 \leftarrow \neg(Y1 \wedge B)$
  $count \leftarrow count + 1$
**while** $x0 \neq Y2$ and $count < 3$
**if** $count = 3$ and $x0 \neq Y2$
  **then** UNSTABLE_ERROR()

FIG. 6.5 – Boucle de simulation d'un composant fortement connecté

dant, d'après Z. Wang, il peut être avantageux de traiter hiérarchiquement un composant contenant des boucles combinatoires de tailles différentes.

### 6.2.2 Modélisation

#### 6.2.2.1 Modélisation en langage VHDL

Le VHDL est un langage de description de circuit [IEE87, IEE93]. La description d'un circuit en VHDL se fait par l'utilisation de processus séquentiels communiquant par des signaux. Les assignations concurrentes représente une façon compacte de décrire un processus. Deux niveaux de description sont communément différenciés [Jac99] : Une description algorithmique qui ne comporte aucune information architecturale et temporelle du circuit et une description de type transfert entre registres (RTL). Le niveau qui nous concerne est le niveau RTL, car il donne le comportement du circuit cycle par cycle du point de vue de ses entrées–sorties.

La norme VHDL, en plus de la grammaire, spécifie la sémantique de simulation associée à l'utilisation de chacune des instructions. Cette sémantique de simulation est le pilotage événementiel.

Il existe plusieurs techniques permettant de simuler des composants décrits en langage

VHDL.

Les méthodes interprétées utilisant une machine virtuelle génèrent un pseudo-code à partir du VHDL. Le simulateur est un programme séparé qui interprète le pseudo-code pour effectuer la simulation. L'avantage du pseudo-code est de permettre la définition d'un jeu d'instructions et d'une machine virtuelle dédiée à la simulation.

Les méthodes interprétées utilisant des arbres syntaxiques représentent le comportement de chaque composant par des structures de données [Vuo97]. Ces structures de données sont évaluées lors de l'exécution de la simulation. Le simulateur Asimut de la chaîne de CAO Alliance construit à partir d'une description VHDL des arbres de décision binaire (BDD) pour représenter les expressions booléennes. Cela suppose donc la transformation des instructions séquentielles en équation à assignation unique. Il utilise un algorithme à pilotage événementiel pour évaluer les arbres.

Les méthodes compilées traduisent le code VHDL en un langage de programmation qui est ensuite compilé sur la station hôte et lié avec un noyau de simulation *ad-hoc* [Sin91, FLLO95]. Le langage de programmation utilisé est en général le C. L'avantage des méthodes compilées est quelles bénéficient des capacités d'optimisation des compilateurs existants. L'outil VSS de Synopsys [Syn98b] utilise cette technique en traduisant le VHDL en langage C.

Le langage VHDL ne définie aucune sémantique pour la description d'un composant. Le simulateur ne peut donc faire aucune hypothèse lui permettant d'accélérer la simulation. De plus, les processus VHDL sont suspendus par des clauses WAIT qui peuvent intervenir à l'intérieur de procédure appelée par le process. De ce fait, toutes les informations concernant l'endroit où le process s'est suspendu doivent être sauvegardé. C'est donc fondamentalement une simulation multi-tâches qui est forcément plus coûteuse qu'une exécution séquentielle, puisqu'elle nécessite des changements (des sauvegardes et des restaurations) de contextes. C'est pourquoi l'utilisation de l'algorithme à propagation événementiel est nécessaire car le coût de l'évaluation d'un process est important.

#### 6.2.2.2 Modélisation en langage C

Le langage C est certainement un des langages les plus utilisés pour la programmation séquentielle. C'est un langage assez proche de la machine et sa compilation est très efficace. Cependant, ce n'est pas un langage de description de matériel et une sémantique particulière doit être définie pour l'utiliser dans un tel contexte.

Le simulateur TSS (Tool for System Simulation) a été développé par Philips Research comme un outil interne d'évaluation des performances des systèmes intégrés [KVdV97]. Il utilise le langage C pour modéliser les composants d'un système.

Dans TSS chaque composant est modélisé par un ensemble de processus décrit chacun comme une fonction. D'autres fonctions sont utilisées pour l'initialisation de chaque instance d'un même modèle et pour effectuer des mesures. Chaque processus possède une liste de sensibilité déclarée dans la fonction d'initialisation du modèle. Deux types de processus sont utilisés : les processus dit « synchrones » dont la liste de sensibilité est réduite à une horloge et les processus dits « asynchrones » décrivant les parties combinatoires d'un composant. TSS permet de simuler des systèmes contenant plusieurs domaines d'horloge à condition que le rapport entre les fréquences d'horloge soit une nombre rationnel. La figure 6.6 montre l'exemple d'un composant décrit comme trois processus $P_1$, $P_2$, $P_3$.

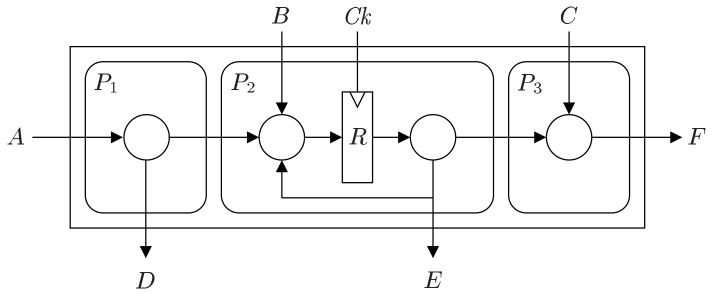



FIG. 6.6 – modélisation d'un composant pour TSS

$P_2$ est un processus synchrone, la sortie $E$ ne dépend que de la valeur du registre $R$. $P_1$ et $P_3$ sont des processus asynchrones, la sortie $D$ dépend de l'entrée $A$ et la sortie $F$ dépend de l'entrée $C$ et de la valeur du registre $R$.

TSS est un simulateur précis au cycle. Pour modéliser la logique combinatoire, il divise le cycle d'horloge en delta-cycles. Le nombre de delta-cycles dans un cycle d'horloge dépend du nombre d'évènements générés pendant la simulation. Il utilise un algorithme à pilotage événementiel pour simuler les processus asynchrones.

Un cycle d'horloge correspond à un cycle de simulation. Un cycle de simulation est constitué de deux phases :

- durant la première phase, tous les processus ayant un port connecté à une horloge active dans leur liste de sensibilité sont exécutés,
- la seconde phase est constitué d'un certains nombres de delta cycles. Pendant un delta cycle, tous les processus, ayant un port de leur liste de sensibilité sur lequel il y a un évènement, sont exécutés. Il y a autant de delta-cycles que nécessaire jusqu'à ce qu'il n'y ait plus d'évènement. Notons que l'écriture d'un signal génère un événement, même si la valeur écrite est la même que la valeur précédente, ce qui peut entraîner un boucle sans fin alors que la valeur des signaux est stable. Ceci doit être pris en compte lors de l'écriture d'un process asynchrone.

La figure 6.7 donne l'algorithme de la boucle de simulation de TSS. Chaque itération de la boucle extérieure correspond à un cycle de simulation. Cette figure montre le coût lié à l'algorithme à pilotage événementiel : deux listes (ligne 4 et 5) doivent être parcouru pour déterminer les processus qui doit être évalués. L'algorithme nécessite aussi que chaque modification de signaux soit signalée dans une liste.

TSS peut simuler les systèmes ayant plusieurs domaines d'horloge. Le modèle d'un composant sensible à plusieurs horloges comprend autant de processus qu'il a d'horloge. Pour simuler les différentes fréquences d'horloge, TSS considère le rapport entre les périodes des horloges. Pour déterminer si une horloge est active à un cycle de simulation, TSS divise la période de l'horloge par un pas égale à la plus petite période. Par exemple, si un système contient deux horloges $ck_0$ et $ck_1$ dont les périodes sont respectivement 2 et 3, on considère une horloge de période 1. La table 6.1 donne les cycles ou les horloges $ck_0$ et $ck_1$ sont actives. Les cycles où aucune horloge du système sont actives sont inutiles. La table 6.2 donne les horloges actives pour les quatre cycles du pas de simulation.

Pour modéliser les signaux, TSS utilise des entiers pour les signaux de largeur inférieure ou égale à 32 bits. Pour les signaux de plus de 32 bits, le concepteur de modèle doit utiliser un mécanisme de passage de message. Ces deux méthodes sont incompatibles et ne permettent pas

```
SIMULATIONLOOP()
while simulate
do EVALUATE_ALL_CLK_PROCESSES()
   while there_are_modified_output_ports
   do for all_modified_output_ports
      do for all_connected_input_ports
         do if type = EVAL_IMM
               then EVALUATE_PROCESS()
               else if type = EVAL_DEL and not (marked_process)
                       then MARK_PROCESS()
                            QUEUE_PROCESS()
      EVALUATE_QUEUED_PROCESSES()
```

FIG. 6.7 – boucle de simulation de TSS

| cycle | horloges actives |
|---|---|
| 0 | $ck_0$, $ck_1$ |
| 1 | |
| 2 | $ck_0$ |
| 3 | $ck_1$ |
| 4 | $ck_0$ |
| 5 | |
| 6 | $ck_0$, $ck_1$ |

TAB. 6.1 – Simulation d'un système ayant deux domaines d'horloge

| cycle | horloges actives |
|---|---|
| 0 | $ck_0$, $ck_1$ |
| 1 | $ck_0$, |
| 2 | $ck_1$ |
| 3 | $ck_0$ |

TAB. 6.2 – Cycles simulés

de faire communiquer par exemples des composants 32 et 64 bits sur un bus supportant les deux modes.

### 6.2.3 Conclusion

Nous avons rappelé que les descriptions de matériel se font à l'aide de processus parallèles communicants par des signaux. Pour simuler ce parallélisme, deux algorithmes sont utilisés : la relaxation qui évalue tous les processus jusqu'à la stabilisation du système et l'algorithme à propagation événementiel qui cherche à réduire le nombre d'itération de processus. Ce dernier algorithme est le plus utilisé. Son utilisation est justifié pour des langages de description dont l'évaluation est coûteuse par rapport au coût de la gestion de l'échéancier que nécessite l'algorithme.

Les simulateurs VHDL améliorent la vitesse de simulation en compilant le langage pour qu'il soit exécuté directement par la machine hôte. Cependant, la sémantique liée au langage ne permet pas une génération de code et une évaluation efficace.

Le simulateur TSS associe au langage C une sémantique de modélisation lui permettant de simuler efficacement un système complet. Toutefois, il utilise un algorithme à propagation événementiel dont la gestion est lourde par rapport aux évaluations supplémentaires des modèles que nécessiterait un algorithme de relaxation. De plus la gestion des signaux n'est pas extensible.

Ainsi, nous chercherons à définir une sémantique de modélisation orientée vers la rapidité d'évaluation et à minimiser le coût de l'algorithme de simulation.

## 6.3 Machines à états communicantes

Un système synchrone peut être décrit comme un ensemble de machines à états communicant par des signaux [Hoa85, GVNG94].

### 6.3.1 Définition d'une machine à états

Une machine à états est définie par le quintuplet $\{I, O, S, t, g\}$, dans lequel :

- $I$ est l'ensemble des valeurs possibles des entrées de la machine à états. On note **I** le vecteur de bits représentant l'encodage de $I$,
- $O$ est l'ensemble des valeurs possibles des sorties. **O** est le vecteur de bits représentant

l'encodage de $O$,

- $S$ est l'ensemble des états possible de la machine. **S** est le vecteur de bits représentant l'encodage de $S$,
- $t$ est la fonction de transition. Elle calcule l'état suivant en fonction de l'état courant et des entrées de la machine. $t$ est telle que $t : I \times S \to S$. **T** est un vecteur de fonctions booléennes calculant **S** en fonction de **I** et de **S**,
- $g$ est la fonction de génération. Elle calcule l'ensemble des sorties de la machine en fonction de l'état courant et de l'ensemble des entrées. $g$ est tel que $g : I \times S \to O$. **G** est un vecteur de fonctions booléennes calculant **O** en fonction de **I** et de **S**.

Deux types de machines à états sont communément différenciés : les machines de Moore dont la fonction de génération ne dépend que de l'état courant, et les machines de Mealy dont la fonction de génération dépend de l'état courant et des entrées. Les figures 6.8 et 6.9 montrent une représentation des deux machines et des dépendances de données des fonctions $t$ et $g$.

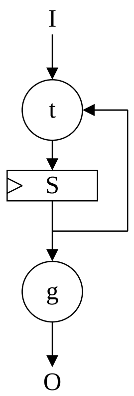



FIG. 6.8 – machine de Moore

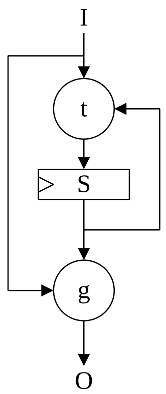



FIG. 6.9 – machine de Mealy

### 6.3.2 Définition des fonctions

On note $\mathbf{I}_0$ le vecteur de bits représentant l'ensemble des entrées primaires du système.

Les équations (6.1) et (6.2) donnent les propriétés d'une machine de Moore et de Mealy dans le système. Nous noterons $\underset{ck}{\Leftarrow}$ l'affectation sur le front actif de l'horloge. L'indice $t$ indique le cycle d'horloge.

$$MooreFSM = \begin{cases} \mathbf{S}^{t+1} & \underset{ck}{\Leftarrow} & \mathbf{T}(\mathbf{I}^t, \mathbf{S}^t) \\ \mathbf{O}^t & = & \mathbf{G}(\mathbf{S}^t) \end{cases} \tag{6.1}$$

$$MealyFSM = \begin{cases} \mathbf{S}^{t+1} & \underset{ck}{\Leftarrow} & \mathbf{T}(\mathbf{I}^t, \mathbf{S}^t) \\ \mathbf{O}^t & = & \mathbf{G}(\mathbf{I}^t, \mathbf{S}^t) \end{cases} \tag{6.2}$$

Pour les machines de Mealy, **O** peut être séparées en deux vecteur de bits : **OS** : le vecteur de bits des sorties ne dépendant pas des entrées et **OC** : le vecteur de bits des sorties dépendant combinatoirement des entrées. On aura :

$$\begin{cases} \mathbf{OS}^t = \mathbf{GS}(\mathbf{S}^t) \\ \mathbf{OC}^t = \mathbf{GC}(\mathbf{S}^t, \mathbf{I}^t) \\ \mathbf{S}^{t+1} = \mathbf{T}(\mathbf{S}^t, \mathbf{I}^t) \end{cases} \tag{6.3}$$

Où **GS** et **GC** sont deux vecteurs de fonctions booléennes représentant respectivement la partie de **G** calculant le vecteur de bits des sorties ne dépendant pas combinatoirement des entrées **OS** et la partie de **G** calculant le vecteur de bits des sorties dépendant combinatoirement des entrées **OC**.

Avec cette notation, une machine de Moore est une machine de Mealy dont l'ensemble des sorties combinatoires est vide : $OC = \varnothing$. Cette séparation permet de dissocier pour une machine à états deux fonctions : une fonction « séquentielle » *Seq* comprenant la fonction de transition et la fonction de génération de Moore, et une fonction « combinatoire » *Comb* qui est la fonction de génération de Mealy.

$$\{\mathbf{S}^{t+1}, \mathbf{OS}^{t+1}\} = \mathbf{Seq}(\mathbf{I}^t, \mathbf{S}^t) = \{\mathbf{T}(\mathbf{I}^t, \mathbf{S}^t), \mathbf{GS} \circ \mathbf{T}(\mathbf{I}^t, \mathbf{S}^t)\} \tag{6.4}$$

$$\mathbf{OC}^t = \mathbf{Comb}(\mathbf{I}^t, \mathbf{S}^t) = \mathbf{GC}(\mathbf{I}^t, \mathbf{S}^t) \tag{6.5}$$

## 6.4 Évaluation

L'évaluation d'une machine à états consiste à exécuter d'abord la fonction séquentielle et ensuite la fonction combinatoire.

### 6.4.1 Évaluation d'un système ne comportant que des automates de Moore

Comme le montre l'équation (6.4), les fonctions séquentielles ne dépendent que des valeurs des signaux et des états calculés au cycle précédent.

L'évaluation d'une fonction séquentielle ne dépend donc pas de l'évaluation d'une autre fonction séquentielle. L'évaluation des fonctions séquentielles peut donc être faite dans n'importe quel ordre, puisqu'il n'y a pas de dépendances entre les résultats de ces fonctions.

La simulation du parallélisme est obtenue en dissociant la valeur courante de la valeur future des signaux de sorte que la modification d'un signal par une machine ne modifie pas instantanément la valeur en entrée d'une autre machine. Les signaux sont mis à jour après l'évaluation de toutes les fonctions séquentielles. Un système ne comportant que des machines de Moore pourra donc être évalué par l'algorithme de la figure 6.10.

```
MOORESIMULATION()
1  do
2      for i ← 1 to n
3      do O_i ← SEQ(I_i, S_i)
4      I ← O
5  while simulate
```

FIG. 6.10 – Boucle de simulation d'un système composé uniquement de machines de Moore

La ligne 4 copie la valeur future des signaux dans la valeur courante après l'exécution de toutes les fonctions séquentielles.

### 6.4.2 Évaluation d'un système comportant aussi des automates de Mealy

Ici, les signaux de Mealy, c'est à dire les signaux $\mathbf{OC}_i$ introduisent des dépendances entre les fonctions combinatoire. Comme nous l'avons vu, deux solutions sont envisageables : le pilotage événementiel et la relaxation. Le pilotage événementiel impose la gestion d'une liste d'événements sur les signaux et d'une liste de sensibilité de chaque fonction combinatoire. Or, en pratique, on peut espérer que l'évaluation d'une fonction combinatoire sera très rapide et que le sur-coût lié à la gestion de l'échéancier sera plus important que les évaluations des fonctions combinatoires qu'il permet d'éviter.

Nous avons donc choisi d'utiliser la relaxation. L'algorithme de relaxation le plus simple consiste à évaluer toutes les fonctions combinatoires jusqu'à obtenir la stabilité. La figure 6.11 donne cet algorithme. La variable *unstable* est utilisée pour détecter la stabilisation du système. Le changement par une fonction de la valeur d'un signal entraîne la réévaluation de toutes les fonctions.

```
MealySimulation()
do
    for i ← 1 to n
    do O_i ← Seq(I_i, S_i)
    I ← O
    do
        unstable = 0
        for i ← 1 to n
        do O_i ← Comb(I_i, S_i)
            if I ≠ O
                then unstable ← 1
    while unstable = 1
while simulate
```

FIG. 6.11 – Évaluation des fonctions de Mealy

Les performances peuvent être améliorées en ordonnançant l'évaluation des fonctions combinatoires en fonction des dépendances liées aux signaux de Mealy. Pour cela nous construisons un graphe de dépendances dont les nœuds sont les modules du système qui possèdent des signaux de Mealy (c'est à dire des modules qui possèdent une fonction combinatoire non vide). Il existe un arc entre deux nœuds $i$ et $j$ lorsque le module $i$ envoie un signal de Mealy au module $j$.

La figure 6.12 donne un exemple de système d'automate. Les automates $A_0$, $A_1$ et $A_2$ sont des automates de Mealy, $A_3$ est un automate de Moore. La figure 6.13 montre le graphe de dépen-

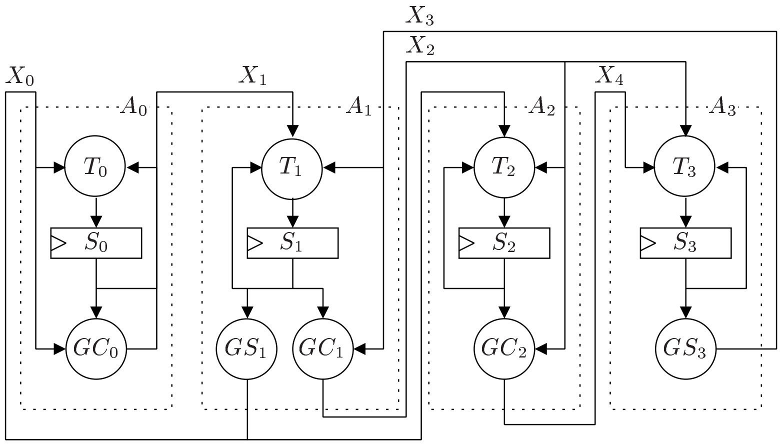



FIG. 6.12 – Exemple de système

dances construit à partir de ce système. Il n'existe de nœuds que pour les automates de Mealy, c'est à dire ceux qui possède une fonction combinatoire. Les arcs ont été nommés par le nom des signaux qui les construit. L'automate $A_1$ possède deux ports de sortie reliés au signaux $X_0$ et $X_2$. Cependant, comme le fait apparaître les vecteur de fonctions **GS** et **GC** de cet automate, seul $X_2$ dépend combinatoirement d'une entrée ($X_3$). Il n'y a donc pas d'arc $X_0$ sur le graphe de dépendance. Le signal $X_4$ est un signal combinatoire, il n'y a pas d'arc correspondant à ce signal dans le graphe car il est relié à l'automate $A_3$ qui n'a pas de fonction combinatoire et donc pas de nœud associé.

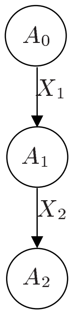



FIG. 6.13 – Graphe de dépendances

S'il n'existe pas de boucles dans le graphe des dépendances de données, un ordre peut être trouvé statiquement avant la simulation de telle sorte qu'une seule évaluation de chaque fonction est nécessaire. Sinon, nous avons choisi d'utiliser pour chaque cycle dans le graphe de dépendance du système, une boucle locale de relaxation permettant d'évaluer correctement chaque cycle plutôt qu'une boucle globale comme celle de la figure 6.11 pour pouvoir minimiser le nombre de fonctions combinatoires appelées lorsque le système contient des boucles combinatoires.

Les deux paragraphes suivant décrivent la méthode que nous utilisons pour ordonner les appels des fonctions combinatoires et insérer les boucles locales de relaxation lorsque cela est nécessaire.

#### 6.4.2.1 Ordonnancement des fonctions combinatoires

Pour ordonner les fonctions combinatoires, nous utilisons le graphe de dépendances de données. Ce graphe peut être composé de plusieurs sous-graphes indépendants ou composantes connexes. Les composantes connexes du graphe peuvent être traitées indépendamment et l'ordre d'évaluation des composantes n'a pas d'importance puisque par définition, elles ne partagent pas d'arcs.

Si le graphe est acyclique, alors les fonctions de Mealy peuvent être ordonnées statiquement. Rechercher cet ordre équivaut à étiqueter les nœuds du graphe par un tri topologique. Le tri topologique consiste à numéroter les nœuds de telle sorte que pour deux nœuds $u$ et $v$ s'il existe un chemin de $u$ vers $v$ alors $u$ aura un numéro plus petit que $v$ [CLR90].

L'ordre obtenu en triant les nœuds par étiquette croissante est celui avec lequel les fonctions combinatoires doivent être appelées.

#### 6.4.2.2 Gestion des boucles combinatoires

Si le système possède des boucles combinatoires, le graphe de dépendance n'est pas acyclique. Un graphe est dit fortement connexe si, étant donnés les sommets quelconques $i$ et $j$ (dans cet ordre), il existe un chemin d'extrémité initiale $i$ et d'extrémité terminale $j$. Si le graphe n'est pas acyclique, soit il est fortement connexe, soit il possède des composantes fortement connexes. Nous effectuons alors une « condensation » du graphe qui consiste à construire le graphe réduit dont les composantes fortement connexes sont des nœuds (les figures 6.14 et 6.15 donnent deux exemples de condensation). Le graphe réduit obtenu est forcément acyclique et donc un tri topologique de ce graphe nous donne, comme au § 6.4.2.1, un ordre d'évaluation ou les boucles combinatoires sont considérées comme une seule fonction.

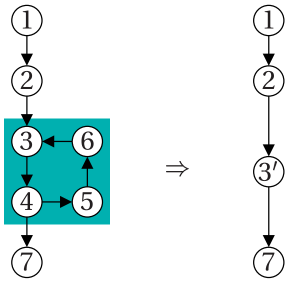



FIG. 6.14 – Exemple de graphe contenant un cycle et le graphe réduit correspondant

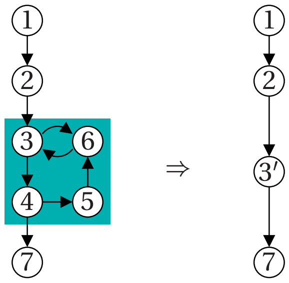



FIG. 6.15 – Autre exemple de graphe contenant plusieurs boucles imbriquées

Pour évaluer les fonctions qui font partie d'une composante fortement connexe, nous avons besoin de boucles de relaxation. Pour ordonner les fonctions à l'intérieur d'une boucle, nous cherchons un chemin hamiltonien dans la composante fortement connexe correspondante. Une chemin hamiltonien est un chemin passant une fois et une fois seulement par chacun des sommets du graphe [GM95]. L'ordre de parcours des nœuds dans ce chemin est l'ordre d'appel des fonctions combinatoires de la boucle. Si la composante fortement connexe ne possède pas de chemin hamiltonien, nous ajoutons des arcs pour rendre la composante hamiltonienne. Ce chemin peut être trouvé par l'algorithme décrit dans [Chr75].

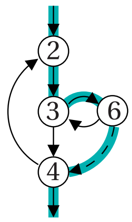



FIG. 6.16 – Exemple de graphe ne possédant pas de chemin hamiltonien

La figure 6.16 montre un exemple de graphe ne contenant pas de chemin hamiltonien. L'arc en pointillé est celui ajouté pour qu'un chemin hamiltonien existe : 2→3→6→4.

Une présentation plus formelle de ces résultats se trouve dans [PHG97a] et [HP98].

## 6.5 Modélisation d'un module en langage C

Pour modéliser une machine à états en langage C, trois fonctions sont nécessaires : une fonction permettant l'initialisation de l'instance, une réalisant la fonction séquentielle et une autre la fonction combinatoire. Un module est décrit par une structure de données contenant d'une part les ports d'entrées–sorties et d'autre part les différentes variables représentant l'état interne du module. À chaque instance de module est attachée une telle structure car différentes instances peuvent être dans des états différents. Les fonctions SEQ et COMB ne doivent pas utiliser de variable globales ou statiques afin de permettre l'existence de plusieurs instances du même modèle dans un système.

La fonction d'initialisation alloue la mémoire nécessaire pour représenter chaque instance et déclare les ports du module. Les paramètres de cette fonction sont les paramètres de l'instance. Par exemple, si l'on souhaite écrire un modèle de mémoire dont la taille est un paramètre, ce paramètre sera passé lors de l'instanciation de la mémoire comme un argument de la fonction d'initialisation. Chaque port est un champ de la structure décrivant l'instance et la valeur de ce champs est un entier décrivant le signal auquel il est connecté. Pour accéder aux valeurs des signaux reliés aux ports, les fonctions utilisent des fonctions d'entrées–sorties spécifiques.

La fonction séquentielle décrit le changement d'état de la machine et l'affectation des sorties. Cette fonction est appelée une fois par cycle d'horloge. Elle peut être décomposée en deux temps :

1. changement d'état en fonction des entrées,

2. affectation des sorties en fonction de la valeur du nouvel état.

La fonction combinatoire décrit le changement de valeur des sorties en fonction des entrées. Cette fonction peut être appelée plusieurs fois par cycle si elle fait partie d'une boucle combinatoire. La fonction combinatoire est facultative et un modèle décrivant une machine de Moore n'en contient pas.

## 6.6 Modélisation des signaux

Chaque signal du système est un vecteur de bits. Si sa taille est inférieur à 32 bits, il est représenté par deux entiers non-signés de 32 bits représentant ses valeurs présentes et futures. L'ensemble des signaux du système est donc modélisé par deux tableaux d'entiers.

Pour lire ou écrire un signal, les fonctions du modèle utilisent des primitives d'accès. La structure de données de l'instance contient un champ par port d'entrée-sortie. Ces champs sont initialisés à la déclaration des connecteurs dans la fonction d'initialisation ; un champ contient l'index du signal auquel il est connecté dans le tableau des signaux. Les primitives utilisent ces index pour accéder aux signaux. Ces primitives sont les mêmes dans la fonction séquentielle et la fonction combinatoire, même si leur comportement est légèrement différent, puisque dans la fonction séquentielle, la valeur lue est la valeur présente du signal et la valeur écrite est la valeur future du signal alors que dans la fonction combinatoire, la valeur écrite et la valeur lue sont les mêmes. Pour cela, lors de l'exécution des fonctions séquentielles, les primitives utilisent les deux tableaux de signaux, alors que lors de l'exécution des fonctions combinatoires, elle n'utilise plus qu'un tableau. Lorsque la stabilité des signaux est atteinte, ce tableau est recopié dans l'autre qui représente alors les entrées du cycle suivant.

Les signaux de plus de 32 bits sont modélisés par autant d'entiers que nécessaire suivant la formule 6.6 où $n$ est le nombre d'entiers et $l$ la largeur du signal en nombre de bits.

$$n = \frac{l-1}{32} + 1 \tag{6.6}$$

Des cases consécutives du tableau de signaux et des primitives particulières d'accès aux signaux sont utilisés. Ce mécanisme permet de conserver une homogénéité dans la représentation des signaux et de permettre à des modèles n'utilisant qu'une partie de la largeur du signal de pouvoir lire cette partie. Par exemple, le PI-Bus peut avoir un bus de données de 32 ou 64 bits. S'il est 64 bits, il sera représenté par deux entiers consécutifs dans les tableaux de signaux. Un modèle utilisant les 64 bits du bus de données accédera aux deux entiers du signal et un modèle n'utilisant que les 32 bits de poids faibles pourra les lire ou les écrire en utilisant les primitives normales comme si le signal était 32 bits.

## 6.7 Simulation

Les principes que nous venons de décrire ont été implémentés dans le simulateur CASS (Cycle Accurate System Simulator).

Les modèles des modules du système y sont décrits par trois fonctions en langage C : une fonction d'initialisation, une fonction séquentielle et une fonction combinatoire. Ces fonctions sont compilées et archivées dans une librairie. La description d'un module pour CASS est donc une librairie contenant ces trois fonctions.

Un système est décrit par une liste d'interconnections des modules par des signaux. Cette liste peut être hiérarchique.

Pour tirer profit de la modélisation en automates communicant des systèmes, La simulation d'un système par CASS se découpe en deux phases :

1. le but de la première phase est de pré-calculer tous ce qui peut l'être avant le début de la simulation. On compile alors ces résultats au lieu d'avoir à exécuter les calculs lors de la simulation. Durant cette phase, le simulateur lit la liste d'interconnections du système, il construit le graphe de dépendance de données des fonctions combinatoires et calcule l'ordre dans lequel les fonctions combinatoires doivent être appelées. À partir de ces informations, le simulateur génère la boucle de simulation contenant les appels aux fonctions des modèles dans un fichier en langage C. Ce fichier est compilé et lié avec les librairies des modèles des modules du système. Cette phase permet au simulateur de ne pas avoir à parcourir de graphe ni de tableau pour connaître les fonctions à appeler et de même évite de passer par des pointeurs sur fonction pour ces appels. Ceci est possible grâce à l'utilisation de boucles de relaxation au lieu d'un algorithme à pilotage événementiel où les évaluations ne peuvent pas être connues statiquement.
2. la deuxième phase est la simulation elle-même qui est l'exécution du programme obtenu. L'interface de l'utilisateur est un moniteur qui lui permet de faire avancer le programme pas-à-pas, poser des points d'arrêt, afficher la valeur des signaux ou des registres internes aux composants, …

La figure 6.18 donne en exemple la boucle de simulation générée pour un système comprenant huit composants et dont le graphe des dépendances de données entre les fonctions combinatoires sont décrites par la figure 6.17.

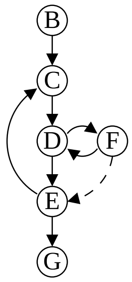



FIG. 6.17 – Exemple de graphe de dépendance

```
SIMULATIONLOOP()
do
    SEQUENTIALA(insA)
    SEQUENTIALB(insB)
    SEQUENTIALC(insC)
    SEQUENTIALD(insD)
    SEQUENTIALE(insE)
    SEQUENTIALF(insF)
    SEQUENTIALG(insG)
    SEQUENTIALH(insH)
    COMBINATIONALB(insB)
    unstable ← 0
    do
        COMBINATIONALC(insC)
        COMBINATIONALD(insD)
        COMBINATIONALF(insF)
        COMBINATIONALE(insE)
    while unstable = 1
    COMBINATIONALG(insG)
while simulate
```

FIG. 6.18 – Boucle de simulation correspondante

## 6.8 Systèmes multi-synchrones

Un système multi-synchrone est un système possédant plusieurs domaines d'horloge. Dans un tel système, un module est sensible à une ou plusieurs horloges.

Un module sensible à $n$ horloges peut être modélisé par le n-uplet $\{\mathbf{I}, \mathbf{O}, \mathbf{S}, \mathbf{T_0}, \cdots, \mathbf{T_n}, \mathbf{G}\}$ possédant autant de fonctions de transition que le module a d'horloges. La figure 6.19 représente l'automate d'un module sensible à deux horloges.

Comme dans le cas où l'automate n'est sensible qu'à une horloge, le vecteur de fonctions de génération $\mathbf{G}$ peut être séparé en un vecteur de fonctions de génération de Moore $\mathbf{GS}$ et un vecteur de fonctions de génération de Mealy $\mathbf{GC}$. On peut alors définir $n$ fonctions séquentielles $Seq_i$ et la fonction combinatoire $Comb$.

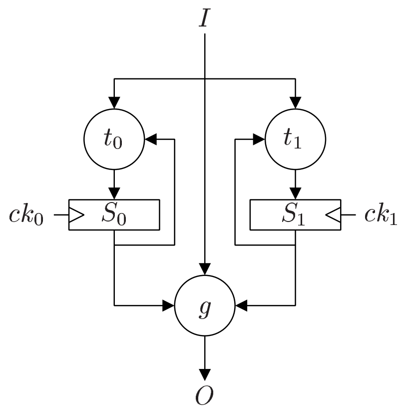



FIG. 6.19 – Automate sensible à deux horloges

CASS n'utilise qu'une fonction séquentielle et qu'une fonction combinatoire par modèle. Cette fonction séquentielle est appelée dès qu'au moins une horloge auquel le modèle est sensible est active. Il peut y avoir plusieurs horloges actives simultanément. La valeur des signaux d'horloge indique au modèle lesquelles sont actives.

On fait l'hypothèse qu'il existe un rapport rationnel entre les fréquences des différentes horloges : par exemple, si un système possède deux horloges dont les fréquences sont 66Mhz et 100Mhz, les fonctions séquentielles sensibles à la première horloge doivent être appelées $3/2$ fois plus souvent que celles sensibles à la deuxième. Pour respecter ce rapport, on calcule le plus petit commun multiple $m$ des fréquences d'horloge du système. Une itération de la boucle de simulation est composée de $m$ cycles numérotés de $1$ à $m$. À chaque cycle, les horloges actives sont celles qui sont divisibles par le numéro du cycle. Les fonctions séquentielles sensibles aux horloges actives sont exécutées. Les fonctions combinatoires, par nature indépendantes des horloges, sont exécutées à tous les cycles.

## 6.9 Simulation de coprocesseurs matériels avant synthèse

Dans les systèmes que nous cherchons à simuler au niveau cycle, tous les modules n'ont pas un modèle précis au bit et au cycle puisque les tâches de la spécification parallèle qui seront réalisées en matériel n'ont pas été synthétisées. Ceci est d'autant plus vrai que le concepteur peut fortement influencer le processus de synthèse d'un coprocesseur matériel.

L'évaluation des communications avant la synthèse et la mesure des débits de données sont nécessaires pour fournir des contraintes réalistes à la synthèse. Il faut pouvoir évaluer dans l'environnement que nous venons de décrire des modules dont l'architecture n'est pas définie. Un tel module est décrit comme une ou plusieurs tâches communicant par des primitives à travers des canaux. C'est-à-dire comme un sous-graphe du graphe de tâches de la spécification parallèle initiale.

La figure 6.20 donne un exemple de système réalisé à partir d'une description sous forme de graphe de tâches communicantes. La tâche $T_1$ y est réalisée en logiciel alors que le tâche $T_2$ doit être réalisé en matériel. La tâche $T_1$ peut être compilée pour le processeur du système en utilisant les schémas de communication matériel–logiciel. Cependant, il reste à connecter la tâche $T_2$ au module d'interface matériel contenant les canaux de communication.

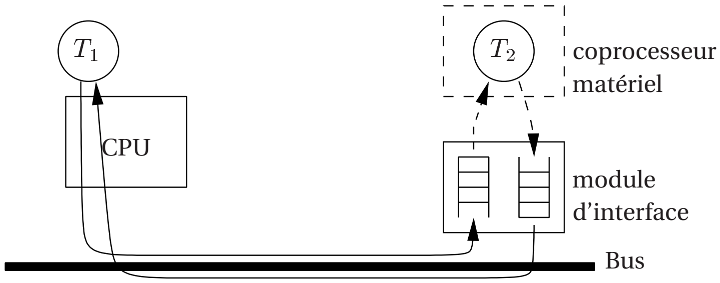



FIG. 6.20 – Exemple de système

Dans la spécification parallèle initiale, chaque tâche est une *thread*. Les tâches réalisées en logiciel sont compilées pour être exécutées par le processeur embarqué. Les tâches matérielles sont réalisées par des coprocesseurs matériels. Ces coprocesseurs après leur synthèse utilisent le protocole vecteur pour communiquer avec le module d'interface auquel ils sont connectés. Pour utiliser la tâche logicielle de la description initiale pour évaluer un coprocesseur matériel, il faut donc permettre la communication entre les primitives utilisées par la tâche logicielle et le protocole vecteur utilisé par le modèle CASS du module d'interface. Comme il n'est pas souhaitable de modifier le code de la tâche, nous devons garder la description de la tâche sous forme de *thread*.

Pour cela, nous générons un adaptateur : un module CASS possédant l'interface utilisant le protocole vecteur qu'aura le coprocesseur après la synthèse. Du point de vue de CASS, l'adaptateur est le coprocesseur. L'adaptateur, à l'initialisation, crée les canaux de communication utilisés par la tâche et lance la *thread* décrivant la tâche. Pendant l'exécution, la fonction séquentielle de l'adaptateur lit les données sur les canaux pour lesquels la tâche est productrice et les transmet au module d'interface en utilisant le protocole vecteur, et il écrit dans les canaux pour lesquels la tâche est consommatrice les données lu en utilisant le protocole vecteur dans le module d'interface. Du point de vue de la tâche, l'ensemble de système simulé par CASS représente un ensemble de tâches avec lesquelles elle communique.

La simulation d'un système dont les coprocesseurs ne sont pas synthétisés est donc un programme multi-*threads* comprenant autant de *threads* qu'il y a de tâches réalisées par des coprocesseurs matériels et une *thread* qui est le simulateur CASS et qui lance les autres *threads* par les modules adaptateurs. La figure 6.21 montre les tâches de l'exemple de la figure 6.20 où il y a deux *threads* : $T_2$ et CASS.

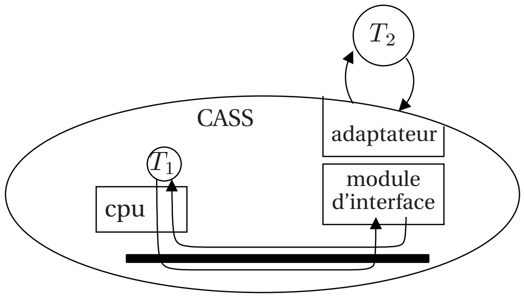



FIG. 6.21 – *threads* exécutées

Pour évaluer plus précisément les communications et les besoins pour la réalisation de la tâche, le modèle de l'adaptateur généré peut être paramétré pour préciser le débit des données entrant et sortant de la tâche. Ceci permet au concepteur de déterminer le débit des données dont il a besoin pour la réalisation du coprocesseur et il pourra l'utiliser pour paramétré l'outil de synthèse qu'il utilisera.

## 6.10 Conclusion

Dans ce chapitre, nous avons décrit les principes de la modélisation sous forme d'automates synchrones des modules matériels d'un système. Cette méthode permet de simplifier la simulation pour l'accélérer. Le modèle d'un module consiste en deux fonctions : une fonction séquentielle et une fonction combinatoire.

La méthode de simulation que nous avons définie tire parti de la modélisation des modules matériels sous forme d'automate. En effet, elle permet d'exploiter au maximum la compilation de la boucle de simulation pour éviter le parcours de structure dynamique durant la simulation. En effet, les fonctions séquentielles sont évaluées une seule fois par cycle. De plus, les dépendances de données permettent de déterminer statiquement l'ordre d'évaluation des fonctions combinatoires. Les boucles combinatoires sont elles, résolues localement par un algorithme de relaxation.

Enfin, nous proposons une méthode et un module rendant possible l'évaluation des coprocesseurs matériels avant même la définition de leur architecture interne en utilisant leur description sous forme de tâches logicielles.

L'outil CASS développé est utilisé dans le chapitre suivant d'une part pour mesurer les performances de simulation obtenues et d'autre part pour l'évaluation d'architectures pour la réalisation d'un système à partir d'une description parallèle.

# Chapitre 7

# Résultats expérimentaux

Nous présentons dans ce chapitre la mise en œuvre de notre approche de conception des systèmes embarqués et l'évaluation des performances du simulateur CASS utilisant la méthode de simulation décrite au chapitre précédent. Ces expérimentations visent cinq objectifs.

Le premier objectif est de montrer qu'à partir de la description d'une application sous forme d'un graphe de tâches communicantes respectant la sémantique des réseaux de Kahn que permet FSS, on peut effectivement évaluer l'application pour déterminer un partitionnement matériel–logiciel, réaliser la synthèse automatique des communications et évaluer les performances du système obtenue. Nous illustrons cette approche sur l'exemple de la réalisation d'un décodeur JPEG à partir de sa description parallèle.

Le deuxième objectif est d'évaluer la qualité d'un système obtenu à partir de sa description utilisant FSS. Pour cela, nous revenons sur le système de décodage d'images JPEG conçu à partir de sa description FSS et nous le comparons à un autre système de décodage d'images JPEG conçu sans utiliser FSS.

L'évaluation précoce des performances du système repose sur un adaptateur permettant d'utiliser la description sous forme de tâche logicielle d'un coprocesseur matériel pour la simulation de l'architecture cible choisie. Le troisième objectif est de montrer que cet adaptateur permet aussi de définir des contraintes externes pour la synthèse des coprocesseurs matériels. Ces contraintes sont nécessaires à la synthèse d'une architecture interne du coprocesseur adaptés au besoins du système.

Nous avons utilisé la définition de schémas de communication pour déterminer les ressources internes du module d'interface générique. Le quatrième objectif est de montrer que le module d'interface générique offre une généricité satisfaisante pour être utilisé dans le cadre de différents systèmes. Pour cela, nous présentons son utilisation dans trois contextes différents.

Le chapitre 6 décrit la modélisation et l'algorithme utilisés par l'environnement de simulation CASS. Chaque module d'un système y est décrit par un modèle en langage C comme des machines à états à l'aide de deux fonctions : une fonction séquentielle et un fonction combinatoire. L'algorithme de simulation de CASS utilise cette modélisation, pour déterminer et compiler un ordre statique d'évaluation des fonctions des modèles. La cinquième objectif est de valider la pertinence de l'algorithme de simulation de CASS. Pour cela, nous comparons les performances de CASS avec un simulateur industriel TSS qui utilise la même modélisation, mais un algorithme de simulation à pilotage événementiel. Cette comparaison est faite sur cinq systèmes en utilisant les mêmes modèles pour CASS et pour TSS.

## 7.1 Synthèse automatique des communications

Cette partie a pour but de montrer, en l'illustrant par le cas d'un décodeur JPEG, qu'à partir de la spécification parallèle d'une application, du choix du partitionnement et de l'architecture cible, il est possible de réaliser un système en instanciant automatiquement des schémas de communication prédéfinis.

### 7.1.1 Le décodage d'une image JPEG

Le JPEG est un format standard de description d'images compressées avec pertes. Il repose sur la compression par la méthode de Huffman de la représentation en fréquence de l'image. Trois étapes sont nécessaires au codage d'une image en JPEG.

La première étape calcule la transformée en cosinus discrète (DCT) qui permet d'obtenir une représentation en fréquence de l'image.

La seconde étape réalise le seuillage qui filtre les fréquences hautes de l'images auxquelles l'œil est peu sensible. C'est cette étape qui provoque les pertes de qualité de l'image,

La troisième étape effectue le codage de Huffman qui compresse l'image à l'aide d'une table : les séquences de bits les plus souvent utilisées sont codés avec le minimum de bits possibles.

Une image JPEG se compose de trois parties séparées par des balises :

- la description de l'image : sa taille, le nombre de tables, l'auteur, . . .
- les tables de Huffman et de seuillage,
- les données compressées représentant l'image.

Le décodage du jpeg consiste à inverser successivement les trois étapes du codage :

1. le décodage de Huffman,
2. le seuillage inverse,
3. la transformée en cosinus inverse.

L'application multi-JPEG que nous souhaitons réaliser doit être capable de décoder un flux vidéo contenant une séquence d'images JPEG. La taille des images est de $320 \times 240$ pixels. Un pixel est un niveau de gris parmi 256. Un pixel est donc codé sur un octet. La taille d'une image décodée est donc de 76800 octets. Le décodeur à réaliser doit être capable de décoder le flux à la vitesse de 25 images par seconde.

### 7.1.2 Spécification parallèle du décodeur JPEG

Cette section décrit la spécification parallèle d'un décodeur d'images JPEG à l'aide de FSS. FSS permet de décrire une application comme un graphe de tâches communicant par des canaux à l'aide de primitives bloquantes.

Le décodage d'une image JPEG étant réalisé en trois étapes distinctes, chacune de ces étapes peut être modélisée par une tâche pour former un macro-pipeline :

- la tâche *vld* réalise le décodage de huffman,
- la tâche *iq* calcule la seuillage inverse,
- la tâche *idct* réalise la transformée en cosinus discrète inverse.

Dans le flux JPEG, une image contient une en-tête composée de différentes informations séparées par des balises (taille de l'image, tables, ...). Il faut donc, une tâche supplémentaire pour séparer le flux en fonction des balises et n'envoyer aux tâches que les données qui leur sont utiles : la table de Huffman et les données vers la tâche *vld*, les tables de seuillage vers la tâche *iq*. Nous appellerons cette nouvelle tâche *demux*.

La tâche *idct* manipule des blocs de $8 \times 8$ pixels organisés d'abord en lignes puis en colonnes. Le seuillage inverse est effectuée en parcourant un bloc suivant le « zigzag scan » illustré par la figure 7.1. Il faut donc insérer une tâche pour réordonnes les données avant de les fournir à la tâche *idct*. Nous noterons cette tâche *zz*.

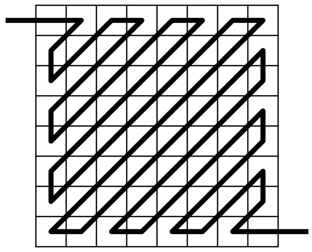


FIG. 7.1 – *zigzag scan*

De même, la tâche *idct* produit des blocs $8 \times 8$ pixels. Le flux de sortie envoyé vers le visualisateur (*ramdac*) doit être organisé suivant un balayage de l'image ligne par ligne de gauche à droite horizontalement et de haut en bas verticalement. Une tâche supplémentaire (*libu*) est donc nécessaire pour reconstituer à partir des blocs fournis par la tâche *idct*, les lignes complètes envoyées au ramdac.

La spécification parallèle de l'application JPEG illustrée par la figure 7.2 comprend au total huit tâches communicant à travers onze canaux.

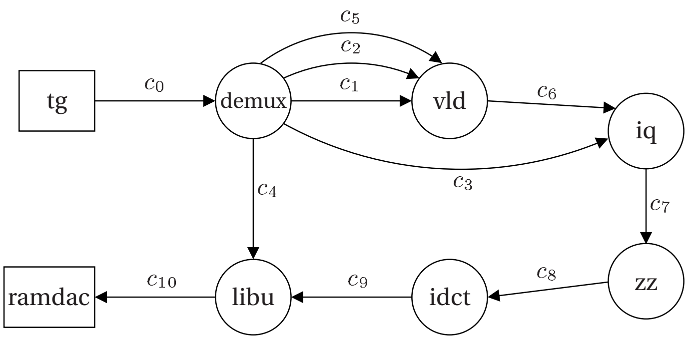



FIG. 7.2 – Spécification parallèle du décodeur JPEG

### 7.1.3 Évaluation de la spécification parallèle

Pour réaliser un système à partir de cette description, il est nécessaire de déterminer quelles tâches seront réalisées en logiciel et quelles tâches seront réalisées par des coprocesseurs matériels. Pour cela, il faut évaluer la spécification parallèle. Une évaluation précise de l'application peut être obtenue par une première architecture où toutes les tâches sont réalisées en logiciel. Cette évaluation donne le coût de chaque tâche pour le processeur embarqué. Cette évaluation est plus précise qu'une évaluation faite avec FSS sur une station de travail car l'exécution d'une tâche sur le processeur embarqué peut prendre beaucoup plus de temps que sur la station de travail. En effet, si par exemple, une tâche réalise des calculs sur des nombres réels et que le processeur embarqué ne possède pas d'unité de calculs flottants, l'exécution de cette tâche sur le processeur embarqué durera beaucoup plus longtemps que sur la station de travail possédant une unité de calculs flottants.

Pour réaliser cette évaluation pour l'application JPEG, il faut utiliser le schéma de communication logiciel–logiciel SS1 pour tous les canaux de communication à l'exception du canal d'entrée $c_0$ et du canal de sortie $c_{10}$. $c_0$ est réalisé par le schéma matériel→logiciel HS1. $c_{10}$ est réalisé par le schéma logiciel→matériel SH1.

Le système simulé est décrit par la figure 7.3. Les modules du système interconnecté par un PI-Bus sont les suivants :

**r3000** est un processeur MIPS R3000 et de ses caches d'instructions et de données. Chacun des deux caches est à correspondance directe et possèdent une interface maître PI-Bus [KH92]. Leur taille est de deux kilo-octets ;

**bcu** est un contrôleur de bus (BCU) pour le PI-Bus. Ce contrôleur peut être paramétrer pour utiliser plusieurs algorithme d'arbitrage. Nous utilisons un arbitrage à priorités circulaires (*round robin*) ;

**ram** est un modèle de RAM à interface esclave PI-Bus. Cette RAM peut être configurer à l'initialisation par le contenu d'un fichier contenant soit des données, soit un programme au format ecoff pour le processeur ;

**tty** est un modèle de TTY affichant des caractères dans une fenêtre ;

**tg** un générateur de trafic pour le flux JPEG. Il lit une séquence d'images dans un fichier et l'écrit sur une interface FIFO ;

**ramdac** un ramdac à interface FIFO affichant une image ;

**sw** est un modèle de *wrapper* VCI esclave PI-Bus transmettant les requêtes PI-Bus à une cible VCI ;

**mw** est un modèle de *wrapper* VCI maître PI-Bus communicant avec un initiateur VCI ;

**vcims** est un modèle du module d'interface décrit dans le chapitre 5. Deux instances de ce module sont utilisées : l'une permettant de réaliser le schéma HS1 pour communiquer avec le générateur de trafic (tg) contient un sous-module fifo esclave de sortie et l'autre permettant de réaliser le schéma SH1 pour communiquer avec le ramdac contient un sous-module fifo esclave d'entrée. Dans les deux cas, une interruption peut être programmée en fonction du remplissage de la fifo pour réveiller la tâche logicielle qui communique avec le coprocesseur. Ces lignes d'interruptions sont reliés au concentrateur d'interruptions (itc) ;

**itc** est un modèle du module d'interface décrit dans le chapitre 5 utilisé comme concentrateur d'interruptions ;

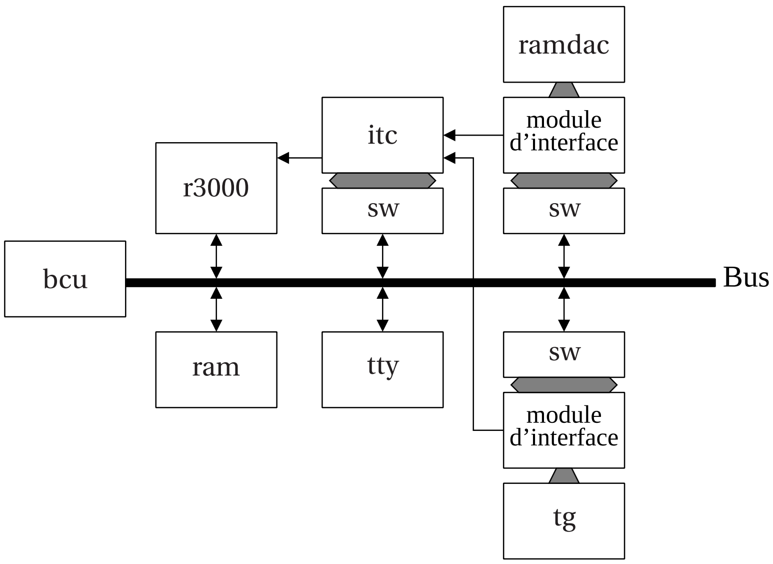



FIG. 7.3 – Système *jpegsoft*

Le modèle CASS du processeur r3000 possède une fonction de profilage permettant de connaître le temps passé dans chaque tâche et dans chaque fonction, le temps pris par les changements de contextes (commutation) et le volume de données échangées dans les canaux. Le tableau 7.1 montre le nombre de cycles d'horloge utilisés par chaque tâche pour le décodage de 30 images JPEG et la participation de chaque tâche au temps total du décodage.

| tâche | temps d'exécution en nombre de cycles | % |
|---|---:|---:|
| demux | 28647486 | 1,49% |
| vld | 330883992 | 17,17% |
| iq | 226908373 | 11,77% |
| zz | 234007968 | 12,14% |
| idct | 826102419 | 42,86% |
| libu | 196336723 | 10,19% |
| commutation | 84197961 | 4,37% |
| initialisation | 578021 | 0,03% |
| total | 1927662943 | 100% |

TAB. 7.1 – Profil de l'application JPEG

| canal | volume de données (octets) |
|---|---:|
| $c_0$ | 255616 |
| $c_1$ | 245536 |
| $c_2$ | 7936 |
| $c_3$ | 2232 |
| $c_4$ | 248 |
| $c_5$ | 620 |
| $c_6$ | 9502720 |
| $c_7$ | 9502720 |
| $c_8$ | 9502464 |
| $c_9$ | 2375552 |
| $c_{10}$ | 2373120 |

TAB. 7.2 – Volume de données transportées par les canaux

La table 7.2 donne le volume de données échangées par les canaux. Les canaux $c_0$ et $c_1$ transportent les données compressées. Le canal $c_{10}$ transporte les pixels décompressés vers le *ramdac*. Après la décompression par le vld, l'image est représentée en fréquence par un entier codé sur 32 bits par pixel (canaux $c_6$, $c_7$ et $c_8$), soit quatre fois plus de données que pour la représentation spatiale (canaux $c_9$ et $c_{10}$).

### 7.1.4 Réalisation du système JPEG

Le profil de la spécification parallèle montre qu'il faut plus de 1,9 milliard de cycles pour décoder 30 images. Sa réalisation purement logicielle nécessiterait une fréquence d'horloge de 1,6 GHz pour être en mesure de décoder le flux à 25 images par seconde.

La tâche *idct* utilise plus de 40% du temps du processeur, mais sa réalisation en matériel ne suffit pas à obtenir une fréquence de fonctionnement plus raisonnable. La tâche *demux* ne représente que 1,5% du temps de l'application et peut être réalisée en logiciel. Les autres tâches utilisant plus de 10% du temps de l'application doivent être réalisées en matériel pour avoir un système fonctionnant à moins de 100 MHz. Nous avons donc choisi de réaliser toutes les tâches à l'exception de la tâche *demux* par des coprocesseurs matériels. Chaque tâche matérielle est réalisée par un coprocesseur distinct. La figure 7.4 présente le système réalisé.

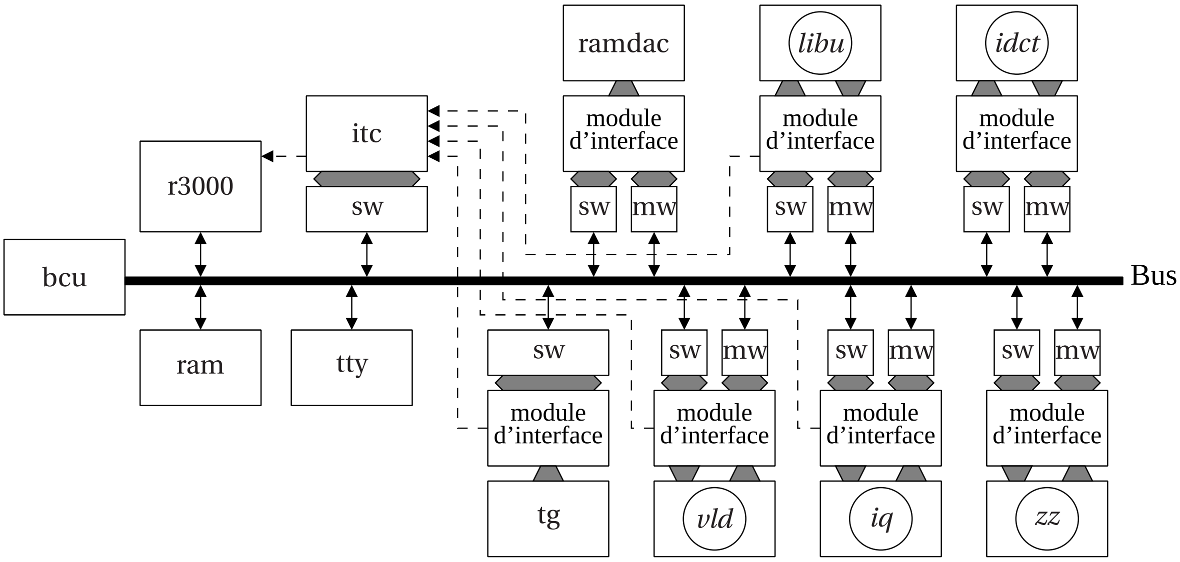



FIG. 7.4 – Architecture du système JPEG

Pour réaliser ce système, il faut choisir un schéma de communication pour chaque canal en fonction de la nature des tâches auxquelles il est relié et des données qu'il transporte. Les canaux $c_6$, $c_7$, $c_8$, $c_9$ et $c_{10}$ reliant des tâches matérielles sont réalisés suivant le schéma matériel→matériel HH1. Le canal $c_0$ est réalisé suivant le schéma matériel→logiciel HS1. Les canaux $c_1$, $c_2$, $c_3$, $c_4$ et $c_5$ sont réalisés suivant le schéma logiciel→matériel SH1. La figure 7.5 présente le graphe de tâches sur lequel sont spécifiés la nature de chaque tâche et les schémas de communication.

La figure 7.6 montre le format de description du graphe de tâches. Pour chaque canal, la description précise le schéma utilisé, la tâche productrice, la tâche consommatrice et le numéro de port des tâches auxquelles il est relié y est précisé. Des paramètres supplémentaires peuvent être ajouté, précisant, par exemple, la taille de la fifo du canal. Les descriptions générées sont :

– l'organisation générale de la mémoire et la définition de la zone mémoire permettant d'accéder aux ressources matérielles des modules d'interface. Une zone mémoire dans la

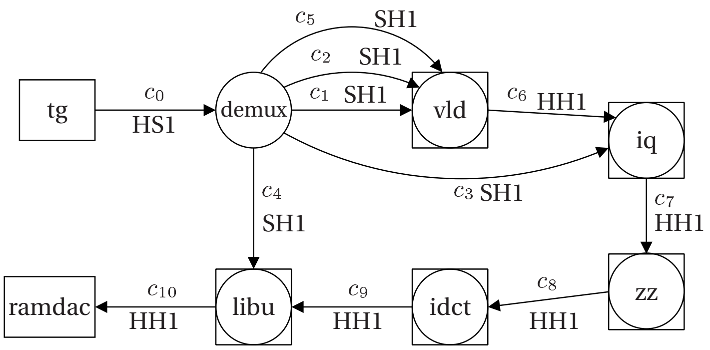



FIG. 7.5 – Spécification parallèle du décodeur JPEG

$c_0$ : HS1 $tg(0) \to demux(0)$

$c_1$ : SH1 $demux(1) \to vld(0)$

$c_2$ : SH1 $demux(2) \to vld(1)$

$c_3$ : SH1 $demux(3) \to iq(1)$

$c_4$ : SH1 $demux(4) \to libu(1)$

$c_5$ : SH1 $demux(4) \to vld(3)$

$c_6$ : HH1 $vld(2) \to iq(0)$

$c_7$ : HH1 $iq(2) \to zz(0)$

$c_8$ : HH1 $zz(1) \to idct(0)$

$c_9$ : HH1 $idct(1) \to libu(0)$

$c_{10}$ : HH1 $libu(2) \to ramdac(0)$

FIG. 7.6 – Description du graphe de tâches

partie non-cachable de l'espace d'adressage du processeur doit être allouée pour chaque coprocesseur,

- la topologie des lignes d'interruptions pour le concentrateur d'interruptions. Comme l'architecture du système possède un concentrateur d'interruptions vectorisé, il faut allouer une ligne de requête d'interruption pour chaque module d'interface émettant des interruptions,
- les fichiers de configuration de chacun des modules d'interface connectant chaque coprocesseur au bus en fonction du nombre de canaux et des schémas de communication utilisés,
- la fonction logicielle initialisant les canaux, les modules d'interfaces et les tâches logicielles. Cette fonction alloue les fifos logicielles en mémoire, programme les vecteurs d'interruptions en fonction de la topologie des lignes d'interruptions, initialise les générateurs d'adresses et les vecteurs d'interruptions des modules d'interface,
- la liste d'interconnection décrivant le système.
- pour chaque coprocesseur, un adaptateur permettant d'utiliser la tâche logicielle que le coprocesseur doit réaliser pour décrire son comportement. Un adaptateur est utilisé pour

remplacer le modèle CASS d'un coprocesseur.

Par exemple, la tâche *vld* possède quatre canaux : trois canaux d'entrée réalisés suivant le schéma SH1 et un canal de sortie réalisé suivant le schéma HH1. Le module d'interface doit donc posséder trois modules fifo esclave d'entrée et une fifo maître de sortie. Chaque fifo esclave doit être capable de générer une interruption en fonction du niveau de remplissage de la fifo. La figure 7.7 donne le fichier de configuration du module d'interface connecté au *vld*.

```
splitTransaction=0 ;
master {
        output (  width=32, depth=8  , vector, it(disconnect)) ;
}
slave {
        input (0, width=32, depth=64, vector, it(threshold)) ;
        input (1, width=32, depth=64, vector, it(threshold)) ;
        input (2, width=32, depth=4  , vector, it(threshold)) ;
}
```

FIG. 7.7 – Fichier de configuration du module d'interface connecté au coprocesseur *vld*

### 7.1.5 Évaluation du système

Pour évaluer le système réalisé, nous utilisons l'adaptateur présenté dans le chapitre précédent au paragraphe 6.9. Ceci nous permet d'avoir un modèle de simulation d'un coprocesseur matériel en utilisant directement la description logicielle de la tâche qu'il doit réaliser.

Dans cette évaluation, les coprocesseurs sont idéaux : leurs performances ne sont limités que par les débits de canaux d'entrée/sortie.

Cette évaluation montre que le système utilise 15721918 cycles pour décoder 30 images. Le système doit donc fonctionner à la fréquence de 13 MHz pour décoder un flux d'image JPEG à 25 images par seconde.

Cette évaluation montre aussi que le bus est utilisé à plus de 95%. Il est possible d'améliorer cette architecture en intégrant deux tâches dans un même coprocesseur ou en modifiant certains paramètres des modules d'interface comme la taille des rafales émises par les fifos maîtres.

Nous avons donc modifié le système pour que les tâches *iq* et *zz* soient réalisées par le même coprocesseur matériel. La figure 7.8 montre le système obtenu. Il nous suffit pour cela de fusionner les deux tâches dans le graphe de tâche décrivant les communications. Pour évaluer le système, nous générons un adaptateur instanciant les deux tâches *iq* et *zz*.

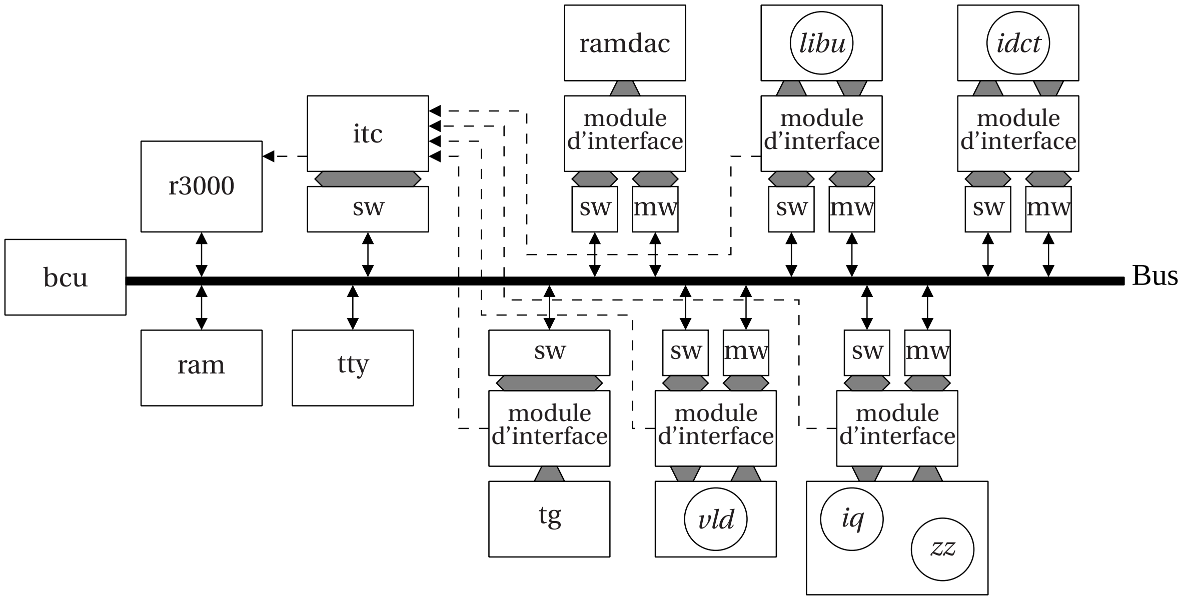



FIG. 7.8 – Amélioration du système JPEG

L'évaluation du système montre que la charge du bus n'est plus que de 66%. Le temps mis par le système pour décoder 30 images et de 15475051 cycles, soit une amélioration de seulement 1,5%. Ceci signifie que le débit du bus était suffisant pour les besoins de l'application et que les fifos des modules d'interface absorbent la latence due à la charge du bus.

### 7.1.6 Conclusion

La réalisation d'un système permettant le décodage des images JPEG à partir de la description de l'application sous forme d'un graphe de tâches montre qu'il est possible d'automatiser la synthèse des communications. Cette automatisation repose sur l'instanciation de schémas de communication prédéfinis et l'utilisation d'un module d'interface générique permettant la communication et la synchronisation avec les coprocesseurs matériels.

Une fois les communications instanciés, nous pouvons directement évaluer le système réalisé et en modifier simplement l'architecture si nécessaire.

## 7.2 Performances du système obtenu par synthèse

L'objectif de cette section est d'évaluer la qualité d'un système réalisé à partir de sa description sous forme de tâches logicielles suivant la méthode que nous avons présentée dans la section précédente. Pour cette évaluation, nous comparons la réalisation du système JPEG réalisé dans la section précédente et un système JPEG dont l'architecture est *ad-hoc*.

### 7.2.1 Description du système JPEG *ad-hoc*

Un système de décodage d'image JPEG a été conçu dans notre laboratoire d'accueil avant le développement de notre méthode de réalisation de système à partir de sa description FSS. L'architecture de ce système est présenté par la figure 7.9.

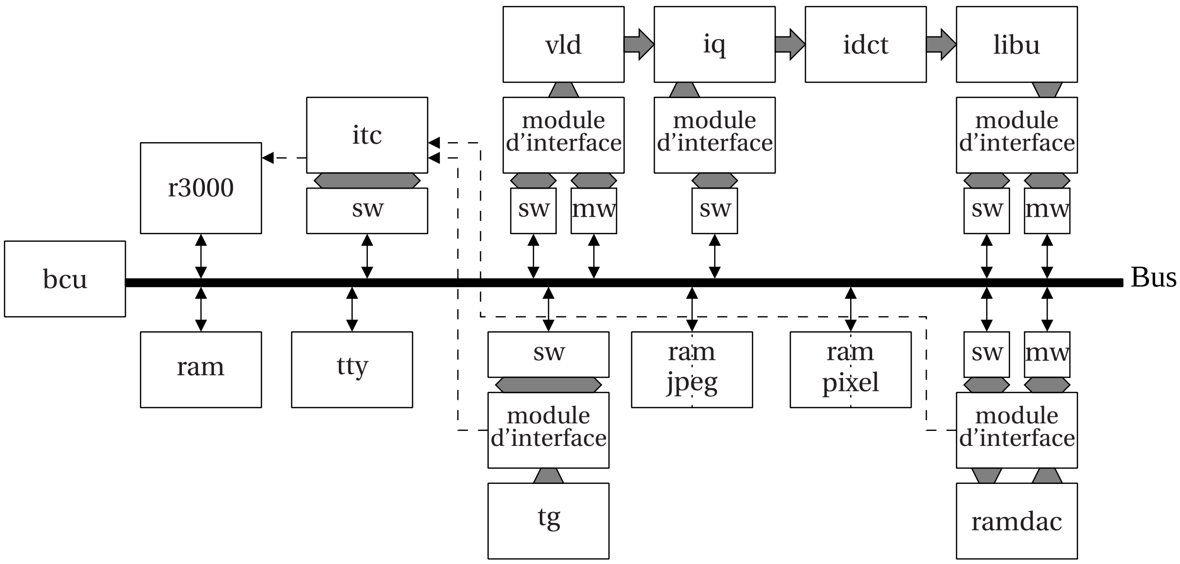



FIG. 7.9 – Système *ad-hoc* réalisant le décodage d'images JPEG

Comme l'architecture du système conçu avec FSS, le système est construit autour d'un PI-Bus et comprend un processeur MIPS R3000. Un générateur de trafic (*tg*) envoie les images compressées au format JPEG et un RAMDAC affiche les images décodées. Le but de ce système est d'afficher le flux d'images JPEG à 25 images par seconde.

Les quatre coprocesseurs de traitement du décodage JPEG ont été développés spécialement pour ce système. Ils constituent les quatre étages d'un macro-pipeline au niveau bloc puisqu'ils échangent par des liens dédiés les blocs $8 \times 8$ de l'image. Ces coprocesseurs n'échangent aucune donnée par le bus.

Le système JPEG est un macro-pipeline au niveau image de trois étages. Chaque étage travaille sur une image distincte. Pendant qu'un étage écrit une image, l'étage suivant lit l'image précédente. Lorsque le premier étage, le générateur de trafic, envoie l'image compressée $n + 1$ dans le RAM *jpeg*, le second étage formé des quatre coprocesseurs lit l'image $n$ dans cette RAM et écrit le résultat du décodage dans la RAM *pixel* et le *ramdac*, le troisième étage, lit l'image $n - 1$ dans la ram *pixel* et l'affiche. Les deux RAM *jpeg* et *pixel* contiennent chacune deux images consécutives, pendant que l'une est lue, l'autre est écrite.

Il n'y a pas de synchronisation entre le *ramdac* et l'étage de décompression : la RAM *pixel* contient deux images, le module d'interface du *ramdac* lit dans cette mémoire comme dans un tampon circulaire, l'étage de décompression doit avoir fini de décompresser l'image $n$ lorsque le *ramdac* a fini de lire l'image $n - 1$.

La synchronisation entre le générateur de trafic et le *vld* est faite à l'aide d'une interruption. Lorsque le générateur de trafic a écrit une image complète dans la RAM *jpeg*, il émet une interruption vers le processeur. Le gestionnaire d'interruptions, vérifie que le *vld* a terminé la décompression de l'image précédente, lit les tables de Huffman, les écrit dans la fifo esclave du module d'interface du *vld*, lit les tables de seuillage, les écrit dans la fifo esclave du module d'interface de *iq* et programme la fifo maître du module d'interface du *vld* pour qu'il lise les données de l'image dans la RAM *jpeg*.

### 7.2.2 Comparaison

La table 7.3 donne le temps mis par le système conçu avec FSS et le système *ad-hoc* pour décoder 30 images. Les paramètres des modèles utilisés pour ces deux simulations : les caches du processeur R3000 sont de 2 kilo-octets chacun, le débit des coprocesseurs est de 1 mot par cycle.

Le système *ad-hoc* décompresse une image quatre fois plus rapidement que le système

| | nombre de cycles | charge du bus | fréquence d'horloge |
|---|---|---|---|
| Système conçu avec FSS | 15721918 | 95% | 13MHz |
| Système *ad-hoc* | 3629919 | 51% | 4MHz |

TAB. 7.3 – Comparaison entre les deux architectures

conçu avec FSS. Ce dernier devra donc fonctionner à une fréquence quatre fois plus importante que le système *ad-hoc* pour décompresser un flux à 25 images par seconde. Ces performances ne peuvent être atteintes que par l'utilisation de liens de communication directs entre les coprocesseurs.

Les coprocesseurs matériels utilisés dans les deux systèmes ont les mêmes caractéristiques, la perte de performance est donc due à la partie logicielle. Dans les deux systèmes, le processeur lit les tables et les écrit dans les fifos des modules d'interface du *vld* et de l'*iq*. Cependant, dans le système *ad-hoc*, le processeur ne lit pas les données, il indique simplement au module d'interface du *vld* où il doit les lire alors que la tâche *demux* dans le système conçu avec FSS lit les données venant de générateur de trafic et les écrit dans le canal de communication pour le *vld*. Cette différence explique la perte de performances. Cependant, le mécanisme utilisé dans le cas du système *ad-hoc* permettant au coprocesseur *vld* de lire directement les données émise par le générateur de trafic ne peut pas être modélisé avec FSS.

Toutefois, le macro-pipeline du système *ad-hoc* nécessite que le système puisse mémoriser quatre images (deux compressées et deux décompressées) soit 170 kilo-octets pour des images $320 \times 240$ de 256 niveaux de gris contrainte que n'impose pas le système conçu avec FSS.

### 7.2.3 Conclusion

La comparaison du système JPEG conçu avec FSS et un système *ad-hoc* fait apparaître que les performances obtenues par le premier sont limitées par le modèle FSS. Celui-ci utilise le passage de message comme moyens de communication qui nécessite systématiquement la copie des données.

## 7.3 Aide à la synthèse des coprocesseurs matériels

Le but de cette partie est de montrer qu'à partir de l'évaluation d'un système dont la modélisation des coprocesseurs matériels a été faite en utilisant leur description sous forme de tâches logicielles et d'un adaptateur, on peut déterminer des contraintes pour la synthèse des coprocesseurs matériels.

L'IDCT (transformée en cosinus discrète inverse) utilisée pour le décodage du JPEG est un algorithme transformant un bloc $8 \times 8$ en fréquence en un bloc $8 \times 8$ spatial. Pour cela, il applique un algorithme nommé *idct1D* sur chaque ligne du bloc calculant ainsi une nouvelle ligne. Il applique ensuite l'algorithme sur les colonnes du bloc formé par les lignes issues de la première étape de calcul.

L'algorithme de l'*idct1D* est constitué de quatre étapes et nécessite quatorze multiplications. L'étape la plus coûteuse en multiplication nécessite le calcul de huit produits partiels. L'*idct1D* étant appliqué une fois par ligne et une fois par colonne d'un bloc, le décodage d'un bloc $8 \times 8$ nécessite donc 224 multiplications.

Quand on cherche à synthétiser un coprocesseur matériel orienté flot de données, la problématique est la suivante :

- le principale paramètre est le débit de traitement : en jouant sur le parallélisme interne du coprocesseur (c'est à dire sur le nombre d'opérateurs ou sur le nombre de tampons mémoire) on peut faire varier le débit. Par exemple, un coprocesseur *idct* pourra calculer un pixel à chaque cycle (parallélisme élevé) ou bien un pixel en 64 cycles (calcul séquentiel).
- l'environnement système, et plus précisément les canaux d'entrée/sortie du coprocesseur, définissent une borne maximale sur le débit : puisqu'on ne peut transférer qu'un seul mot par cycle, le débit maximal est fixé par la largeur de la fifo.

Le problème posé consiste donc à déterminer quel est le débit minimal de calcul que doit supporter le coprocesseur pour satisfaire les besoins du système. Ceci permet de déterminer quelles directives doivent être données à l'outil de synthèse du coprocesseur.

Pour cela, une évaluation peut être réalisée en utilisant un adaptateur permettant d'exploiter directement la description FSS de la tâche dans l'environnement de simulation CASS. Cet adaptateur est décrit au paragraphe 6.9 du chapitre 6. Pour déterminer le débit minimal acceptable, le concepteur peut paramétrer l'adaptateur pour réduire le débit du coprocesseur et en mesurer l'impact sur les performances du système.

Nous avons utilisé ce paramétrage pour déterminer les performances du système en fonction du débit de l'*idct*. Le système utilisé est le système JPEG où les tâches *demux* et *vld* sont réalisées en logiciel. La figure 7.10 donne les résultats de cette expérimentation.

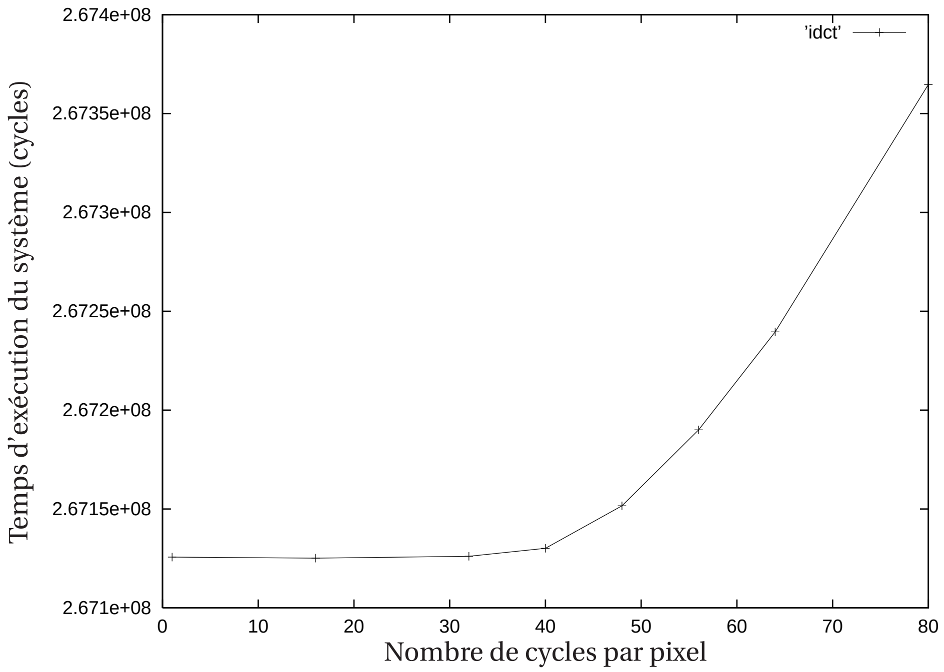



FIG. 7.10 – Temps d'exécution du système en fonction du débit de l'*idct*

Cette figure montre la limite à partir de laquelle les performances du système sont dégradées par la baisse du débit de l'*idct*. Tant que le débit est supérieur à un pixel pour 32 cycles, le coprocesseur *idct* n'est pas le facteur limitant.

## 7.4 Généricité du module d'interface

Dans cette partie, nous nous intéressons à l'adéquation entre les services offerts par le module d'interface générique et les besoins variés de différentes architectures matérielles.

Pour cela, nous étudions l'emploi d'un module d'interface générique dans trois contextes différents :

– le premier résume l'utilisation du module d'interface dans la réalisation du système JPEG présentée dans la section précédente,
– le second décrit l'utilisation de module d'interface générique dans une autre architecture de décodeur JPEG dont la modélisation n'a pas été faite à l'aide de FSS,
– le troisième décrit l'utilisation de module d'interface générique pour réaliser un coprocesseur de filtrage dans le projet COSY. Ce système présente la particularité d'utiliser des schémas de communication différents de ceux que nous avons défini.

### 7.4.1 Le système JPEG conçu avec FSS

Dans le système présenté figure 7.4, les modules d'interface sont utilisés pour la réalisation des schémas de communication présenté dans le chapitre 4.

Ces schémas utilisent :

– des modules fifo maître de sortie pour communiquer avec des fifos esclave d'entrée. Les générateurs d'adresses sont configurés pour avoir une adresse constante.
– des modules fifo esclave de sortie. Ces modules génèrent des interruptions en fonction du nombre de données présentes dans la fifo.
– des modules fifo esclave d'entrée. Ils sont utilisés dans deux cas. Pour la réalisation de schémas HS1, ils peuvent alors générer des interruptions en fonction des données présentes dans la fifo. Pour la réalisation de schémas HH1, ils peuvent alors stopper le module fifo maître qui émet les données et le redémarrer par un accès en mode maître comme décrit dans le paragraphe 5.2.2.

La taille des fifos varie de 2 à 64 mots de 32 bits. Le nombre de sous-modules par module d'interface varie de 1 à 4. Les coprocesseurs communiquent avec les modules d'interface par des protocoles vecteur.

### 7.4.2 Un système de décodage JPEG à architecture *ad-hoc*

Une architecture d'un décodeur JPEG non-basé sur une description par un graphe de tâches a été présentée au § 7.2.1. Cette architecture utilise le module maître esclave de façon différente. La figure 7.9 présente ce système.

Dans cette architecture, les modules d'interfaces utilisés ne génèrent pas d'interruptions. Les modules fifo maîtres sont utilisés pour adresser des tampons circulaires. Le module d'interface du coprocesseur VLD possède quatre registres de configuration permettant au processeur de lui spécifier la taille de l'image et un registre d'état permettant au processeur de déterminer si le coprocesseur VLD a achevé le traitement d'une image. L'interface entre les coprocesseurs et leur module d'interface utilise le protocole fifo.

Dans cette architecture, le logiciel est fondamentalement un gestionnaire d'interruptions écrit spécifiquement pour cette application. Une interruption est émise par le générateur de trafic ($tg$) lorsqu'il a terminé la réception de l'image. Le gestionnaire d'interruption vérifie que le VLD a terminé le traitement d'une image, écrit les tables dans les fifos du VLD et de l'IQ et programme le module maître du VLD pour lire les données dans la mémoire.

### 7.4.3 L'utilisation du module d'interface dans le projet COSY

Le projet COSY est un projet européen esprit regroupant Cadence, Philips et Siemens dans le but de définir une méthode de conception de système permettant la réutilisation de blocs (*IP-based*) [BSVKK98].

La méthode proposée par COSY pour la conception d'un système embarqué part d'une application sous forme de graphe de tâches communicantes respectant la sémantique des réseaux de Kahn. L'outil VCC de Cadence (*Virtual Component Co-design*) permet d'évaluer l'application, de décrire l'architecture et de choisir l'affectation des tâches à la partie logicielle ou à la partie matérielle [VCC01]. Pour réaliser les communications entre les tâches matérielles ou logicielles, des schémas de communication différents de ceux présentés ici ont été définis par Philips. Leur implémentation repose cependant sur le module d'interface générique que nous avons défini. L'architecture cible est basée sur un processeur MIPS R3000, un PI-Bus et un système d'exploitation embarqué pSOS. Ces schémas de communication sont différents de ceux décrits au chapitre 4 car il ne repose pas sur les *threads* POSIX, ils utilisent les moyens de synchronisation propre au système d'exploitation pSOS.

Pour illustrer ces schémas, un système réalisant un filtre a été décrit. Le graphe de tâches réalisé est décrit par la figure 7.11. Les tâches $p$ et $c$, respectivement la tâche produisant l'image et la tâche lisant l'image filtrée, sont réalisés en logiciel. La tâche $f$, le filtre, est réalisé par un coprocesseur matériel. Le système réalisé est décrit par la figure 7.12.

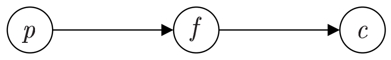



FIG. 7.11 – Système réalisant un filtre

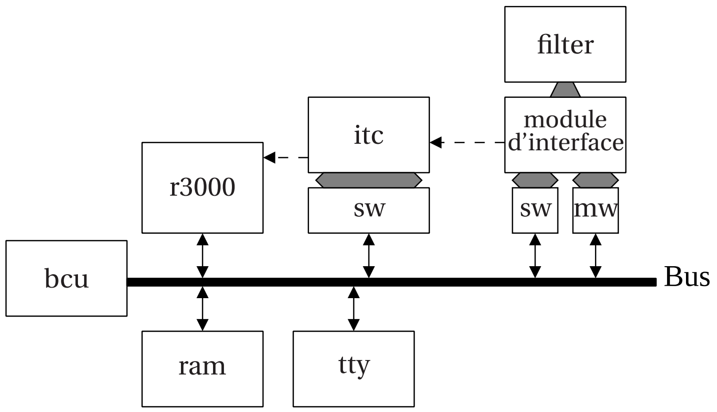



FIG. 7.12 – Système réalisant un filtre

Le schéma de communication logiciel→matériel utilise une fifo en mémoire, un module fifo maître d'entrée lit les données dans la fifo. À chaque requête de lecture par le coprocesseur d'un vecteur, le module d'interface émet une interruption. Le gestionnaire d'interruption configure le module maître pour lire les données dans la mémoire.

Le schéma de communication matériel→logiciel utilise un module fifo esclave de sortie dont la fifo est la fifo du canal. La synchronisation est faite en utilisant la requête d'interruption du module fifo en fonction du nombre de données dans la fifo : si la tâche logiciel $c$ lit $n$ données dans le canal alors que la fifo ne les contient pas, la primitive de lecture programme le module d'interface pour qu'il émette une requête d'interruption lorsque les $n$ données sont présentes et stoppe la tâche. Le gestionnaire de cette interruption réveille la tâche $c$.

### 7.4.4 Conclusion

Ces trois exemples montrent que le module d'interface peut être utilisé dans différents contextes pour la réalisation des communications dans un système intégré. Son architecture générique lui permet d'être utilisé pour réaliser d'autres schémas de communication que ceux que nous proposons dans le chapitre 4

À ces exemples, on peut ajouter que l'utilisation de la norme VCI lui permet de communiquer avec d'autre bus que le PI-Bus que nous avons utilisé dans nos exemples.

## 7.5 Évaluation des performances de CASS

Pour valider les choix que nous avons fait pour l'algorithme de simulation de CASS, nous l'avons comparé à un simulateur industriel, TSS utilisé par Philips, visant le même niveau de description et utilisant une modélisation proche de la notre.

Comme nous l'avons présenté au chapitre 6, pour simuler un système, CASS utilise des modèles décrits par deux fonctions : une fonction séquentielle et une fonction combinatoire. Cette modélisation permet de simplifier l'algorithme de simulation. Les fonctions séquentielles sont d'abord évaluées, leur ordre d'évaluation n'important pas. Les fonctions combinatoires sont évaluées ensuite. L'ordre d'évaluation est calculé statiquement en fonction des dépendances de données. Lorsqu'il y a une boucle dans les dépendances de données, une boucle de relaxation est insérée localement pour évaluer la boucle. Pour chaque système, CASS compile une boucle de simulation décrivant l'ordre d'évaluation des fonctions. Il n'y a donc pas d'échéancier.

Dans TSS, un modèle est décrit par des processus. TSS distingue deux types de processus, les processus synchrones sensibles uniquement à l'horloge et les processus asynchrones qui décrivent les parties combinatoires d'un module. La fonction séquentielle de CASS est un processus synchrone pour TSS, la fonction combinatoire un processus asynchrone. Un modèle CASS peut donc être utilisé sous TSS sans modifications majeures : seules les fonctions d'entrées/sorties et les fonctions d'initialisation sont changées. TSS évalue les processus synchrones de la même façon que CASS évalue les fonctions séquentielles. Mais pour les fonctions combinatoires, il utilise un algorithme à pilotage événementiel. Cet algorithme nécessite que toutes les modifications de la valeur d'un signal lors de l'évaluation d'un processus, même synchrone, soient mémorisées dans une liste servant à déterminer dynamiquement quels sont les processus asynchrones qui doivent être évalués.

Nous avons mesuré les performances de CASS et de TSS sur cinq systèmes :

**simple** est un système minimal dont les modules sont interconnectés par un PI-Bus. Il comprend un MIPS R3000, un contrôleur de bus, un TTY permettant l'affichage et quatre RAM correspondant aux différentes plages d'adresses nécessaires au R3000. Le programme exécuté par le R3000 est un programme séquentiel ;

**1proc** est un système *simple* auquel est ajouté un deuxième TTY. Le programme exécuté par le R3000 est multi-tâches ;

**2proc et 3proc** sont des extensions de *1proc* par un, deux et trois processeurs et autant de TTY ;

**jpegsoft** est un système *1proc* auquel est ajouté un afficheur graphique et un générateur de flux d'images JPEG. La particularité du point de vue de la simulation de ce système est qu'il comporte une boucle combinatoire entre le contrôleur de bus et les wrappers esclaves de l'afficheur et du générateur de flux.

Nous avons utilisé les mêmes modèles pour CASS et TSS de sorte à pouvoir comparer les performances de l'algorithme de simulation. Nous avons utilisé une station de travail Sun Ultrasparc dont le processeur est cadencé à 450 MHz. La figure 7.13 donne les résultats obtenus. Sur les systèmes simulés, le gain ($\frac{\text{vitesse de CASS}}{\text{vitesse de TSS}}$) varie de 1,2 à 3,25.

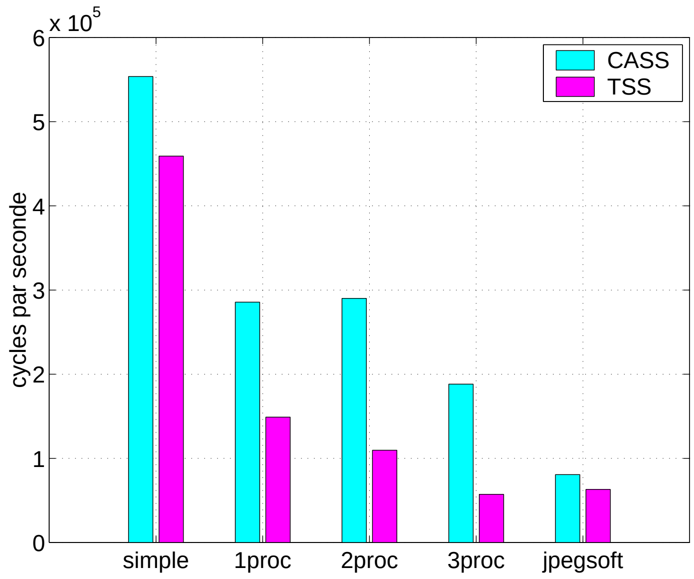



FIG. 7.13 – Performances de CASS et de TSS

La baisse de performance entre les systèmes *simple* et *1proc* s'explique par la différence du nombre d'accès mémoire que réalise le processeur. En effet le taux d'occupation du bus est de 12% pour le système *simple* et de 36% pour le système *1proc*. Le programme exécuté sur le système *1proc* est un programme multi-tâches nécessitant des accès à la mémoire pour les communications et les changements de contexte. Lorsque le processeur ne réalise pas d'accès à la mémoire, l'exécution des autres modèles est très rapide puisqu'ils sont oisifs.

Nous l'avons vu, la principale différence entre CASS et TSS réside dans l'utilisation par TSS d'un algorithme à pilotage événementiel visant à minimiser le nombre d'évaluation des fonctions combinatoires. Cet algorithme nécessite une gestion dynamique de l'évaluation des fonctions combinatoires. Pour mettre en évidence le bénéfice que CASS tire de son algorithme d'ordonnancement statique, nous avons mesuré pour chaque système, le nombre moyen de signaux commutant par cycle. La figure 7.14 présente le gain de CASS par rapport à TSS en fonction du nombre de signaux commutant par cycle.

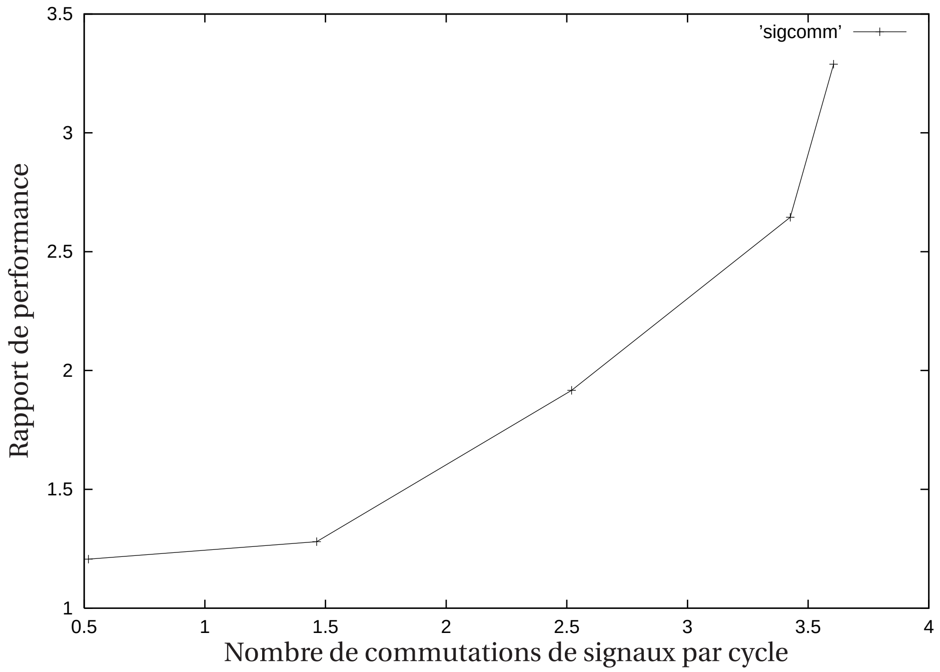



FIG. 7.14 – Gain en fonction du nombre de signaux commutant par cycle

Cette courbe montre les performances que CASS tire de l'absence d'échéancier et de la compilation de la boucle de simulation.

## 7.6 Conclusion

Dans ce chapitre, nous avons montré que l'architecture générique du module d'interface permet de réaliser automatiquement la synthèse des communications d'un système en instanciant des schémas de communication prédéfinis.

Nous avons également montré que le module d'interface possédait une généricité suffisante pour être utilisé pour la réalisation de système utilisant d'autres schémas de communication que ceux que nous proposons.

Nous avons illustré sur l'exemple du coprocesseur *idct* comment la possibilité d'utiliser une tâche FSS directement sous le simulateur CASS permet non seulement d'évaluer le système avant la synthèse des coprocesseurs matériels mais aussi de déterminer des contraintes pour leur synthèse.

Enfin, nous avons montré l'efficacité de l'algorithme de simulation que nous avons mis en œuvre dans le simulateur CASS. Cet algorithme utilise une modélisation au cycle près sous forme de machine à états. Il utilise cette modélisation pour ordonner statiquement l'ordre d'évaluation des modèles. Nous avons montré en le comparant à un simulateur utilisant la même modélisation que cette algorithme était plus efficace qu'un ordonnancement dynamique.

# Chapitre 8

# Conclusion

La densité d'intégration actuelle des circuits intégrés numériques permet la réalisation sur une seule puce d'un système complet. Ces systèmes conçus pour réaliser une application particulière contiennent une partie matérielle et une partie logicielle.

La spécification d'un système est donc une application. Cette application est définie sans préjuger des parties qui seront réalisées en matériel de celles qui le seront en logiciel.

Une méthode de conception de systèmes intégrés consiste, à partir de l'application, à déterminer ce qui doit être fait en matériel et ce qui doit être fait en logiciel et à synthétiser ces deux parties. Une étape de la conception est la synthèse des communications permettant de relier les deux parties de l'application.

Le point de départ de notre travail est une spécification parallèle de l'application sous la forme d'un graphe de tâches , l'affectation de chacune de ces tâches à la partie matérielle ou à la partie logicielle et le choix du architecture cible pour le système. Dans la spécification parallèle, les tâches communiquent à travers des canaux par deux primitives bloquantes, une primitive de lecture et une primitive d'écriture. Un canal de communication est relié à deux tâches : un producteur utilisant la primitive d'écriture pour accéder au canal et un consommateur utilisant la primitive de lecture. Cette spécification possède les propriétés des réseaux de Kahn garantissant la conservation du comportement de l'application quelque soit l'ordre d'évaluation des tâches.

Nous avons montré qu'à partir de ces descriptions, une méthode automatique peut être mise en œuvre pour la réalisation d'un système. La méthode que nous proposons repose sur la réalisation des communications entre les tâches suivant des schémas prédéfinis. Les interfaces de ces schémas sont les primitives de communication utilisées par les taches dans la spécification parallèle. Les schémas ne nécessitent pas de modification des tâches. Ils reposent sur un module d'interface générique permettant la communication entre un coprocesseur matériel et le reste du système.

Nous avons montré que la généricité du module d'interface permet d'implémenter les schémas de communication que nous avons défini, mais aussi qu'elle est suffisante pour que le module d'interface soit utilisé pour la réalisation des communications entre un coprocesseur matériel et le reste du système en utilisant des méthodes de communication différentes des nôtres.

Pour évaluer les systèmes, nous avons besoins d'un environnement de simulation. Le niveau de précision que nous avons retenu est le cycle d'horloge car il permet d'évaluer précisément le coût des protocoles utilisés. Nous avons montré qu'en modélisant les modules d'un système sous forme de machines à états, nous pouvons définir un algorithme simple de simulation au cycle près. La modélisation d'un module est faite par deux fonctions : l'une décrivant la partie séquentielle de la machine à états, l'autre la partie combinatoire. L'algorithme de simulation utilise cette modélisation pour calculer statiquement l'ordre d'évaluation des fonctions décrivant les modules. Si un tel ordre ne peut être trouver en raison de boucles de dépendances de données entre les fonctions combinatoires, nous insérons une boucle de relaxation locale. Cet ordonnancement statique nous permet d'utiliser la compilation de la boucle de simulation pour bénéficier pleinement de la simplicité de l'algorithme de simulation. Nous avons montré, en nous comparant à un simulateur utilisant les mêmes modèles mais un ordonnancement dynamique et un pilotage événementiel, que notre méthode de simulation permet une amélioration notable des performances.

Notre travail peut être amélioré dans les directions suivantes :

**Spécification de l'application :** La spécification parallèle que nous utilisons est basé sur les réseaux de Kahn. Les propriétés des réseaux de Kahn nous assurent que le comportement de l'application ne sera pas modifiée par l'ordre d'évaluation des tâches. Toutefois, certaines applications ne peuvent pas être modélisées par cette méthode. Par exemple, si l'on souhaite modéliser une télévision numérique, le décodage de la réception se modélise très bien avec la spécification parallèle. Par contre, l'appui par l'utilisateur sur la télécommande pour changer de chaîne ne peut pas être modélisée. Il est donc souhaitable d'étendre le nombre de primitives pour pallier ce manque. La lecture ou l'écriture non-bloquante ou le « select » sont des primitives qui peuvent être synthétisées en utilisant des schémas semblable aux schémas que nous avons mis en œuvre pour la lecture et l'écriture bloquantes. Cependant, le comportement décrit à l'aide de ces primitives dépend de l'ordre dont les tâches sont évaluées.

**Optimisation des communications entre tâche logicielles :** Le mécanisme de communication par passage de message est bien adapté à la communication entre deux tâches matérielles ou à la communication entre une tâche matérielle et une tâche logicielle. Cependant, il est moins adapté à la communication entre deux tâches logicielles alors que l'architecture cible suppose l'existence d'un espace d'adressage partagé par les (co)processeurs. Pour améliorer ces communications, il est souhaitable d'utiliser des primitives de communication permettant l'usage de la mémoire partagée.

**Séquentialisation :** Un processeur exécute de façon plus efficace un programme séquentiel qu'un programme parallèle multi-tâches car il n'a pas de changement de contexte à effectuer et les communications peuvent être fait par des variables partagées. Une fois les tâches logicielles déterminées, il est donc préférable de transformer ces tâches en un unique programme séquentiel.

**Simulation :** Le simulateur CASS utilise au plus deux fonctions pour modéliser un module. Ceci permet de minimiser le nombre d'appels de fonction dans la boucle de simulation. Mais cela peut générer des fausses boucles combinatoires qui nécessitent l'usage d'une boucle de relaxation. Ces fausses boucles peuvent être évitées en permettant au concepteur de définir plusieurs fonctions combinatoires et pour chaque fonction combinatoire la liste de ses entrées.

# Bibliographie


[ABG+97] Ivan Augé, Rajesh K. Bawa, Pierre Guerrier, Alain Greiner, Ludovic Jacomme, and Frédéric Pétrot. User guided high level synthesis. In Ricardo Reis and Luc Claensen, editors, *VLSI : Integrated Systems on Silicon*, pages 464–475, Gramado, Brazil, August 1997. IFIP, Chapman & Hall.

[Amd67] G.M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, Reston, Va., April 1967. AFIPS Press.

[APH01] Ivan Augé, Frédéric Pétrot, and Denis Hommais. A pragmatic approach to the design of embedded systems. In *Proceedings of Design Automation and Test in Europe Conference, Designer's Forum*, pages 170–174, Munich, Germany, March 2001.

[ARM99] ARM. *AMBA Specification Rev 2.0*, May 1999.

[BSVKK98] Jean-Yves Brunel, Alberto Sangiovanni-Vincelli, Rainer Kress, and Wido Kruijtzer. COSY : a methodology for system design based on reusable hardware & software ip's. In *the EMMSEC*, pages 709–716, Bordeaux, September 1998.

[Cal93] Jean-Paul Calvez. *Embedded Real-Time Systems.* John Wiley & Sons, 1993.

[Chr75] Nicos Christofides. *Graph Theory, An Algorithmic Approach*, chapter 10, Hamiltonian Circuits, Paths and the Traveling Salesman Problem, pages 214–235. Academic Press, 1975.

[CKL+00] Jordi Cordatella, Alex Kondratyev, Luciano Lavagno, Marc Massot, Sandra Moral, Claudio Passerone, Yosinori Watanabe, and Alberto Sangiovanni-Vicentelli. Task generation and compile-time scheduling for mixed data-control embedded software.

In *Proceedings of the 37th Design Automation Conference*, pages 489–494. IEEE, June 2000.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, chapter 23, pages 485–488. MIT Press, 1990.

[COH+99] Pai Chou, Ross Ortega, Ken Hines, Kurt Partridge, and Gaetano Borriello. IPCHINOOK : An integrated IP-based design framework for distributed embedded systems. In *Proceedings of the 36th Design Automation Conference*, pages 44–49, 1999.

[dKES+00] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers. Yapi : Application modeling for signal processing systems. In *Proceedings of the 37th Design Automation Conference*, pages 402–405. IEEE, June 2000.

[DMBIJ97] Jean-Marc Daveau, Gilberto Fernandes Marchioro, Tarek Ben-Ismail, and Ahmed Amine Jerraya. Protocol selection and interface generation for hw-sw codesign. *IEEE Transaction on VLSI Systems, Special Issue On Design Automation of Complex Integrated Systems*, March 1997.

[FBK99] Josef Fleischmann, Klaus Buchenrieder, and Rainer Kress. Java driven codesign and prototyping of networked embedded systems with java. In *Proceedings of the 36th Design Automation Conference*, pages 794–797, New Orleans, June 1999. IEEE.

[FLLO95] Robert S. French, Monica S. Lam, Jeremy R. Levitt, and Kunle Olukotun. A general method for compiling event-driven simulations. In *Proceedings of the 32th Design Automation Conference*, pages 151–156, June 1995.

[GBA+99] Daniel Geist, Giora Biran, Tamara Arons, Michael Slavkin, Yvgeny Nustov, Monica Farkas, Karen Holtz, Andy Long, DAve King, and Steve Barret. A methodology for the verification of a "system on chip". In *Proceedings of the 36th Design Automation Conference*, pages 574–579. IEEE, 1999.

[GCM94] Rajesh K. Gupta, Claudionor N. Coelho Jr, and Giovanni De Micheli. Program implementation schemes for hardware-software systems. *IEEE Computer*, June 1994.

[GDWL92] Daniel D. Gajski, Nikil Dutt, Allen Wu, and Steve Lin. *High Level Synthesis, Introduction to Chip and System Design*, chapter 1. Kuwer Academic Publisher, 1992.

[GM95] Michel Gondran and Michel Minoux. *Graphes et algorithmes*, chapter 8, page 317. Eyrolles, 1995.

[GM96] Rajesh K. Gupta and Giovanni De Micheli. A co-synthesis approach to embedded system design automation. *Design Automation for Embedded Systems*, 1(1-2) :69–120, June 1996.

[Goe95] Mudit Goel. Process networks in ptolemy II. Master's thesis, Electronics Research Laboratory, Berkeley, 1995.

[GVNG94] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong. *Specification and Design of Embedded Systems*. Prentice Hall, 1994.

[HCM+99] F. Hessel, P. Coste, Ph. Le Marrec, N.E. Zergainoh, J.M. Daveau, and A.A. Jerraya. Communication interface synthesis for multilanguage specifications. In *rsp99*, Clearwater, USA, June 1999.

[HD96] D. Hommais and A. Derieux. Educational project : Design of a compatible macrocell 6800 using the alliance cad system. In *Proceedings of Mixed Design of Integrated Circuits and Systems*, pages 564–567, May 1996.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HP90] John L. Hennessy and David A. Patterson. *Computer architecture, a quantitative approach*. Morgan Kaufmann Publisher, Inc, 1990.

[HP98] Denis Hommais and Frédéric Pétrot. Efficient combinational loops handling for cycle precise simulation of system on a chip. In *24th Euromicro*, pages pages 51–54, Vesteras, Sweden, August 1998. IEEE.

[HPA01] Denis Hommais, Frédéric Pétrot, and Ivan Augé. A practical tool box for system level communication synthesis. In *Codes*, 2001.

[HPG97] Denis Hommais, Frédéric Pétrot, and Alain Greiner. Un environnement de simulation pour les systèmes embarqués. In *Actes du premier colloque CAO de circuits et systèmes*, pages 66–69, Grenoble, France, January 1997.

[IEE87] IEEE, New York, USA. *IEEE standard VHDL Language Reference Manual*, 1987.

[IEE93] IEEE, New York, USA. *IEEE standard VHDL Language Reference Manual*, 1993.

[Jac99] Ludovic Jacomme. *Analyse sémantique de descriptions VHDL synchrones en vue de la synthèse*. Thèse de doctorat de l'université paris 6, October 1999.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing*, pages 471–475, 1974.

[Kal95] Asawaree Kalavade. *System-Level Codesign of Mixed Hardware-Software Systems*. PhD thesis, Electronics Research Laboratory, Berkeley, 1995.

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[Kla97] Peter Klapproth. *PRISC Architecture Framework, PI-Bus Specification*. Philips Semiconductors ASIC Service Group, Embedded Systems Technology Centre Eindhoven, August 1997.

[KLdM93] Tilman Kolks, Bill Lin, and Hugo de Man. Sizing and verification of communication buffers for communicating processes. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 660–664, Santa Clara, CA, USA, November 1993.

[KM77] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *Information Processing*, pages 993–998, 1977.

[KM98] Peter Voigt Knudsen and Jan Madsen. Communication estimation for hardware/software codesign. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (Codes/CASHE)*, 1998.

[KRK99] Tommy Kuhn, Wolfgang Rosenstiel, and Udo Kebschull. Description and simulation of hardware/software systems with java. In *Proceedings of the 36th Design Automation Conference*, pages 790–793, New Orleans, June 1999. IEEE.

[KVdV97] Sjaak Koot, Arjan Versluys, and Pieter de Visser. *TSS User Manual*. Philips Semiconductors, September 1997.

[LS99] Luciano Lavagno and Ellen Sentovich. ECL : A specification environment for system-level design. In *Proceedings of the 36th Design Automation Conference*, pages 511–516. IEEE, 1999.

[Naç98] François Naçabal. *Outils pour l'exploration d'architectures programmables embarquées dans le cadre d'applications industrielles.* PhD thesis, Institut Nationnal Polytechnique de Grenoble, 1998.

[OMI96] Open Microprocessor systems Initiative. *OMI 324 : Pi-Bus*, April-May 1996. Available at `http://www.omimo.be/public/data/_indstan.htm#OMI324`.

[Par95] Thomas M. Parks. *Bounded Scheduling of Process Networks.* PhD thesis, BSE Princeton University, 1995.

[PH99] Frédéric Pétrot and Denis Hommais. Cycle accurate system simulation. In *Medea-Esprit Conference*, Antwerpen, Belgium, September 1999.

[PHG97a] Frédéric Pétrot, Denis Hommais, and Alain Greiner. Cycle precise core based hardware/software system simulation with predictable event propagation. In *the 23$^{rd}$ Euromicro Conference*, pages 182–187, Budapest, Hungary, September 1997. IEEE.

[PHG97b] Frédéric Pétrot, Denis Hommais, and Alain Greiner. A simulation environment for core based embedded systems. In *Proceedings of the* 30$^{th}$ *IEEE International Simulation Symposium*, pages 86–91, Atlanta, Georgia, April 1997. IEEE.

[Row94] J.A. Rowson. Hardware-software co-simulation. In *Proceedings of the 31th Design Automation Conference*, pages 439–440, June 1994.

[Sin91] Sunder Singhani. Code generation for vhdl - interpreted vs. compiled code. In *EuroVHDL*, pages 336–339, 1991.

[Syn98a] Synopsys. *VSS Interfaces*, August 1998.

[Syn98b] Synopsys. *VSS Reference Manual*, August 1998.

[VCC01] Cadence. *Virtual Component Co-design*, 2001. `http://www.cadence.com/datasheets/vcc.html`.

[VSI98] VSI Alliance. *On Chip Bus Development*, August 1998. `http://www.vsi.org/library/specs/summary.htm#ocb`.

[VSI00] VSI Alliance, On-Chip Bus Development Working Group. *Virtual Component Interface Standard (OCB 2 1.0)*, March 2000.

[Vuo97] Huu Nghia Vuong. *Une nouvelle méthode de simulation par évaluation directe des expressions logiques représentées par des graphes : application à des circuits modélisés* *par un sous-ensemble du langage VHDL*. Thèse de doctorat de l'université paris 6, December 1997.

[WM90] Zhicheng Wang and Peter M. Maurer. LECSIM : A levelized event driven dompiled logic simulator. In *Proceedings of the 27th Design Automation Conference*, pages 491–496, 1990.

[WO93] Alan S. Wenban and John W. O'Leary. Codesign of communication protocol. *IEEE Computer*, 26(12) :46–52, December 1993.

[ŽM96] Vojin Živojnović and Heinrich Meyr. Compiled hardware-software co-simulation. In *Proceedings of the 33th Design Automation Conference*. IEEE, June 1996.