THÈSE DE DOCTORAT DE L'UNIVERSITÉ PARIS VI

SPÉCIALITÉ INFORMATIQUE

Présentée par François DONNET

Pour obtenir le titre de

DOCTEUR DE L'UNIVERSITÉ PARIS VI

SYNTHÈSE DE HAUT NIVEAU CONTRÔLÉE PAR L'UTILISATEUR

Soutenue le 20 janvier 2004, devant le jury composé de

Mr Ahmed JERRAYA	Rapporteur
Mr Eric MARTIN	Rapporteur
Mr Alain GREINER	Examinateur
Mr Henri MICHEL	Examinateur
Mr Frédéric PÉTROT	Examinateur
Mr Tanguy RISSET	Examinateur
Mr Ivan AUGÉ	Directeur de thèse

Avant-propos et remerciements

Cette thèse a été effectuée dans le Laboratoire d’Informatique de Paris VI au département Architecture des Systèmes Intégrés et Micro-électronique dirigé par Alain Greiner, Professeur à l’Université Paris VI.

Je tiens à remercier particulièrement Ivan Augé, Maître de Conférence, enseignant à l’Institut d’Informatique d’Entreprise à Evry, qui a dirigé mes recherches. Il a toujours été disponible malgré l’éloignement et a constamment suivi mes travaux. Je tiens à le remercier pour sa relecture attentive de ce manuscrit ainsi que ses conseils avisés. Son soutien m’a été d’une aide plus que précieuse.

J’adresse mes remerciements à Ahmed Amine Jerraya et Eric Martin qui m’ont fait l’honneur d’être les rapporteurs de ma thèse, ainsi qu’à l’ensemble des membres du jury.

Je remercie l’ensemble des membres du département ASIM avec qui j’ai eu des échanges fructueux. J’ai eu un grand plaisir à travailler au sein de l’équipe SoC menée par Frédéric Pétrot. Il m’a fait bénéficier de sa large expérience couvrant tous les domaines de la micro-électronique. Il a de plus contribué pour une part non négligeable à notre outil de synthèse de haut niveau *ugh*.
J’ai aussi une pensée particulière pour les premiers utilisateurs de notre outil de synthèse Mouhamadou Diaby, Etienne Faure, Jean-Baptiste Lhomme et Mathieu Tuna. Bien qu’ils aient eu leurs propres échéances à tenir, leur patience et leur tolérance ont été appréciables. De plus, ils ont soulevé des questions avant qu’elles ne deviennent des problèmes.

Je remercie le Laboratoire d’ ´Électronique des Systèmes Temps Réels (LESTER), pour m’avoir permis d’utiliser l’outil de synthèse *gaut* et plus particulièrement, Pierre Bomel, pour avoir répondu de façon très réactive à mes questions incessantes sur leur outil.

Enfin, j’ai une pensée particulière pour Fabienne Cori qui m’a soutenu pendant ces trois années.

Résumé

La synthèse de haut niveau de coprocesseur dédié est un problème complexe. Elle doit répondre à différents objectifs tels que la fréquence de fonctionnement, la surface, la consommation, etc...
Ces critères caractérisent le circuit généré et donc l'efficacité de l'outil de synthèse de haut niveau.

Cette thèse présente une approche en deux étapes de la synthèse de haut niveau.
Dans un premier temps, on s'intéresse à minimiser la surface et à maximiser le parallélisme.
Dans un deuxième temps, le circuit est ajusté pour prendre en compte les caractéristiques électriques du matériel et produire un circuit fonctionnant à la fréquence désirée.

L'outil de synthèse peut donc se concentrer sur l'optimisation d'un nombre réduit de caractéristiques visées par le concepteur.
Cependant, pour les circuits dont le contrôle dépend des données, l'objectif ne peut pas être caractérisé par une directive globale car une telle directive est inadaptée et ne permet pas de qualifier le degré d'optimisation à atteindre. Les répercussions sur les autres caractéristiques du circuit ne sont pas contrôlées par le concepteur.
Le respect de la directive globale est difficilement tenable par les algorithmes de synthèse. Les phases d'ordonnancement, d'allocation et d'affectation nécessaires à la synthèse de haut niveau possèdent une complexité non polynomiale. Elles se résolvent la plupart du temps par des heuristiques qui ont des conséquences imprédictibles sur les caractéristiques du circuit généré.

L'automatisation de la synthèse de haut niveau passe par une description précise des attentes du concepteur. Nous étudions dans cette thèse les directives nécessaires pour cibler précisément une solution et rester compatibles avec notre démarche en deux phases de la synthèse. Il faut que la définition de ces directives soit la moins contraignante possible pour le concepteur, tout en étant attentif à la perte d'efficacité qu'engendre leur simplification.

Mots clefs : synthèse de haut niveau, coprocesseur dédié, fréquence de fonctionnement, surface, parallélisme, contrôle dépendant des données, ordonnancement, allocation, affectation, micro-architecture, circuit intégré.

Abstract

The high level synthesis for dedicated coprocessor is a very complex problem. It has to cope with different aims like the working frequency, the surface, the consumption, etc...
These criteria specify the generated circuit and the high level synthesis tool efficiency.

This Ph.D thesis presents a two steps approach in the high level synthesis.
In the first step, we minimize the surface and maximize the parallelism.
In the second step, the circuit is modified to take in account the electrical characteristics and to fetch the wanted working frequency.

The synthesis tool has to optimize the circuit on a very few number of criteria which are specified by the designer.
By the way, for the data control dominated circuits, the aim can't be qualified by a global guidance, because a such guidance is unadapted and can't qualify the optimization level. The collateral effects on the other characteristics of the circuit aren't controlled by the designer.
The respect of the global guidance is difficult to reach by the synthesis algorithms. The scheduling, allocation and binding phases of the high level synthesis are NP-complex. Most of these problems are solved with heuristics which have unpredictable consequences on the circuit characteristics.

The automation of the high level synthesis needs a very precise description of the designer's attempts. We are studying in this Ph.D thesis the necessary guidances to precisely aim a solution and keep the compatibility with our two-phases approach. We have to reduce the guidance definition to facilitate the designer's work and be aware of the loss of efficiency which has been induced.

Keywords : high level synthesis, dedicated coprocessor, working frequency, surface, parallelism, data control dominated, scheduling, allocation, binding, micro-architecture, integrated circuit.

Table des matières

Chapitre 1

Introduction

Dans le domaine de la conception assistée par ordinateur de circuits intégrés, l'intérêt porté à la synthèse de haut niveau réside essentiellement dans la réduction des temps de conception des circuits [Cam96]. La synthèse de haut niveau automatise certaines tâches de la conception.

L'une des évolutions récentes du flot de conception est l'utilisation systématique des outils de synthèse. La synthèse de haut niveau consiste à traduire une description comportementale séquentielle comme un programme en langage 'C' ou en 'PASCAL' en un circuit intégré.

Les performances d'un circuit à l'intérieur d'un système intégré sont difficilement évaluables autrement que par la simulation du système. Il faut donc construire un modèle au niveau matériel pour pouvoir évaluer le circuit dans son contexte. Le concepteur doit pouvoir tenter plusieurs architectures matérielles de son circuit afin de pouvoir choisir le rapport entre les performances et le coût de son circuit. Cependant les temps de conception "à la main" deviennent rédhibitoires car ils entraînent un coût d'ingénierie très important. Seule l'automatisation de la synthèse peut amener à une solution en un temps raisonnable. La synthèse de haut niveau facilite le travail d'exploration architecturale puisqu'elle accélère la production d'essais successifs pour un circuit.

A ce jour des résultats satisfaisants ont été obtenus pour des domaines d'application spécifiques comme le traitement du signal. En revanche, les outils de synthèse de haut niveau non spécialisés ne sont pas encore opérationnels pour des circuits complexes. La raison essentielle réside dans la faiblesse de ces outils à prendre en compte les contraintes temporelles ou structurelles du cahier des charges du circuit. Le concepteur ne peut décider des performances de son circuit.

Dans notre laboratoire *LIP6/ASIM*, un outil de synthèse de haut niveau a été développé. Il permet à l'utilisateur de spécifier des contraintes structurelles et temporelles du circuit.
Notre expérience avec cet outil sur des réalisations complexes nous a montré que des recherches doivent être menées pour étendre son champ d'application :

1. la description des contraintes structurelles est lourde pour l'utilisateur, alors qu'un bon nombre d'entre elles pourraient être déduites automatiquement ;

2. l'absence de contraintes temporelles ferme les portes de tout une classe de circuits, en particulier les circuits synchrones.

ugh (User Guided High level synthesis) est l'outil de synthèse de haut niveau sur lequel nous allons réaliser les expérimentations. Il fait partie d'un ensemble d'outils d'aide à l'exploration architecturale appelé *DiSyDEnt* (Digital System Design Environment [ADG+02]) développé au laboratoire *LIP6/ASIM*.

1.1 Plan du manuscrit

Le chapitre 2 présente les différentes phases de la synthèse de haut niveau et leur complexité. Il définit le cadre et les objectifs de cette thèse.

Le chapitre 3 présente de façon critique les approches classiques de la synthèse de haut niveau ainsi que les outils existants.

Le chapitre 4 est la présentation globale de la méthode logicielle utilisée par notre outil de synthèse de haut niveau. Cette méthode permet de s'affranchir des contraintes temporelles inhérentes à la synthèse et de simplifier la problématique.

Dans le chapitre 5, nous comparons les directives structurelles que le concepteur peut donner à l'outil de synthèse. Nous choisissons les plus efficaces et les plus adaptées à notre méthode logicielle.

Le chapitre 6 présente les algorithmes utilisés par notre outil de synthèse de haut niveau.

Le chapitre 7 présente les résultats expérimentaux. Il compare les résultats obtenus par la méthode développée dans cette thèse à d'autres approches existantes.

Enfin dans le chapitre 8, nous dressons le bilan des travaux effectués et quelques perspectives de recherche.

Chapitre 2

Problématique

Sommaire

Ce chapitre a pour objet de poser les problèmes abordés dans cette thèse. Il traite des problèmes rencontrés aujourd'hui lors de la synthèse de haut niveau dans le domaine des circuits intégrés.

Les outils de synthèse de haut niveau ne sont pratiquement pas utilisés dans le domaine industriel. Ils ne répondent pas aux attentes des concepteurs. Car dans la plupart des cas, les circuits issus de la synthèse ne respectent pas les objectifs de performance.
La synthèse de haut niveau est un problème très complexe. L'outil de synthèse doit prendre beaucoup de décisions et respecter de nombreuses contraintes pour produire un résultat acceptable. Nous pensons que la synthèse de haut niveau est un problème trop complexe pour être totalement automatisé, et nous proposons une méthode qui permet au concepteur d'aider l'outil de synthèse dans ses décisions.

L'outil *ugh* (User Guided High level synthesis) développé au laboratoire *LIP6* est un outil de synthèse de haut niveau produisant des coprocesseurs spécialisés. *ugh* oblige le concepteur à donner des directives pour réduire le nombre de décisions et simplifier le travail de l'outil de synthèse. Le résultat de la synthèse est un circuit respectant les contraintes données par le concepteur.
Nous utiliserons l'outil *ugh* pour tester expérimentalement la validité de la méthode de synthèse de haut niveau proposée dans cette thèse.

Tout d'abord, il s'agit de positionner la synthèse de haut niveau telle que nous la voyons dans le flot de conception et d'identifier le domaine auquel elle est destinée. Nous verrons les difficultés de la synthèse de haut niveau d'un point de vue général. Pour les résoudre, *ugh* doit disposer d'un certain nombre d'informations que nous énumérerons. Nous nous poserons la question de la nature de ces informations et ce qu'elles modifient dans la méthode de synthèse de haut niveau. Enfin, nous nous demanderons comment intégrer de manière efficace le résultat de la synthèse dans le flot de conception.

2.1 La synthèse

2.1.1 Conception de circuit assistée par ordinateur

Pour définir la synthèse de haut niveau (HLS), nous nous basons sur le schéma de la figure 2.1 inspiré du *schéma en Y de Gajski* [HG85] [RKG+92]. Sur la figure, les cercles concentriques représentent des niveaux d'abstraction croissant du centre vers l'extérieur :

- ⇒ Circuit : conception d'une cellule à partir de transistors.
- ⇒ Logique : conception d'une macro-cellule à partir de transistors.
- ⇒ Flot de données : conception d'un chemin de données à partir de macro-cellules.
- ⇒ Séquence d'instructions : conception d'un coprocesseur à partir d'un chemin de données et d'une macro-cellule (i.e l'automate à états finis).
- ⇒ Système : conception d'une application à partir de coprocesseurs et de macro-cellules (bus, RAM, ...).

Les trois axes du Y représentent trois vues différentes d'un circuit. La vue comportementale donne la fonctionnalité du circuit. La vue structurelle indique les éléments qui composent le circuit. La

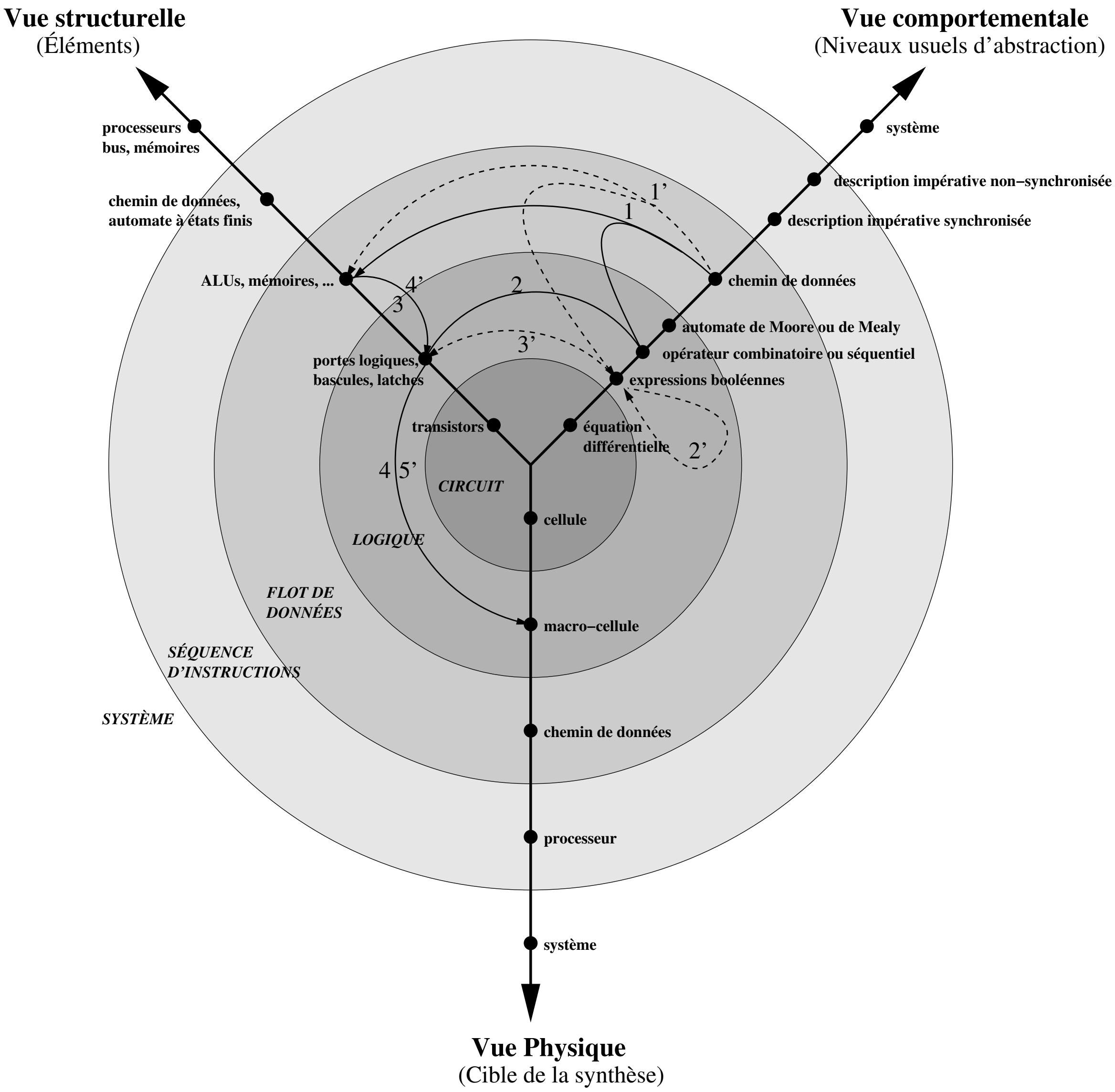


FIG. 2.1 Deux différents flots de conception.

vue physique contient les masques de fabrication du circuit. Les points sur les axes donnent les représentations usuelles des circuits introduites par les méthodes de conception.

La figure 2.1 permet de représenter les outils de conception assistée par ordinateur (CAO) de circuit. Les outils effectuent des transformations permettant de passer d'un point à un autre point (potentiellement le même). Ces transformations sont représentées par des arcs. Par exemple, l'*arc 4* représente un outil de placement/routage de cellules, l'*arc 3* représente une mise à plat d'une ALU en portes logiques et l'*arc 2'* une optimisation booléenne.

Ce schéma permet de visualiser les différents chaînages d'étapes (les arcs) menant de la vue comportementale à la vue physique (les points sur les axes).
Voici deux exemples de flot de conception, qui, partant d'un chemin de données et arrivant à une vue physique, passent par des étapes intermédiaires différentes :

1. Dans cet exemple, les transformations sont représentées par des arcs pleins dans la figure 2.1. L'*étape 1* produit une vue structurelle avec des instances comportementales d'opérateurs. La forme en hyper-arc de l'*arc 1* symbolise la production conjointe de la description structurelle d'opérateurs et des descriptions comportementales des opérateurs instanciés. Chaque comportement d'opérateur est synthétisé séparément en une vue structurelle de portes logiques grâce à l'*étape 2*. La vue structurelle d'opérateurs est mise à plat (*arc 3*) pour obtenir une vue structurelle au niveau portes de notre chemin de données entier. Le chemin de données est placé et routé de manière globale par l'*étape 4*.
2. Dans cet exemple, les transformations sont représentées par des arcs en pointillés dans la figure 2.1.
   L'*étape 1'* produit une vue structurelle dont les feuilles sont des descriptions comportementales exprimées uniquement sous la forme d'expressions booléennes. L'*étape 2'* représente l'optimisation booléenne de ces expressions afin d'en réduire, par exemple, le nombre de littéraux. Les expressions booléennes sont synthétisées en portes logiques (*arc 3'*). La vue structurelle est mise à plat lors de l'*étape 4'* pour pouvoir placer et router l'ensemble grâce à l'*étape 5'*.

2.1.2 Descriptions comportementales

La figure 2.1 introduit de nouveaux niveaux de description par rapport au schéma en Y de Gajski pour différencier les différentes étapes et algorithmes de la synthèse. Ceux-ci sont généralement pris en compte par un même outil et sont donc masqués pour l'utilisateur.

- L'équation différentielle donne le comportement électrique de la cellule.
- Les expressions booléennes sont un flot d'assignations concourantes exclusivement composées de fonctions booléennes.
- L'opérateur est soit combinatoire soit séquentiel. Il est décrit sous la forme d'un flot d'assignations concourantes à l'aide de fonctions arithmétiques et booléennes.
- L'automate de Moore ou de Mealy est une description d'automate à états finis. Un automate étant défini par :
  soient
  $\mathcal{E}$: l'ensemble fini des états de l'automate,
  $\mathcal{I}$: l'ensemble fini des entrées de l'automate,
  $\mathcal{S}$: l'ensemble fini des sorties de l'automate.
  On définit la *fonction de transition* de l'automate telle que $\mathcal{E} \otimes \mathcal{I} \longmapsto \mathcal{E}$.
  On définit la *fonction de génération* de l'automate de Moore telle que $\mathcal{E} \longmapsto \mathcal{S}$.
  On définit la *fonction de génération* de l'automate de Mealy telle que $\mathcal{E} \otimes \mathcal{I} \longmapsto \mathcal{S}$.
- Le chemin de données est un flot d'assignations concourantes. Il comporte des opérateurs séquentiels, arithmétiques, logiques...
- La description impérative est une description procédurale comme les langages Pascal, C ou Ada.
- Dans la description synchronisée, les frontières de cycles sont données. La *fonction de transition* est donc connue.

- Dans la description non synchronisée, les frontières de cycles ne sont pas données.
- Le système représente le traitement complet d'une ou plusieurs tâches et les échanges de données nécessaires entre les différents éléments d'un système.

Dans la littérature, la description impérative synchronisée et le chemin de données sont des descriptions de niveau transferts de registres (RTL). Dans ces descriptions, les registres, les opérations fonctionnelles qui leurs sont assignées, ainsi que le cycle dans lequel elles sont faites, sont déjà identifiés.

2.2 La synthèse de haut niveau

2.2.1 Définition de la synthèse de haut niveau

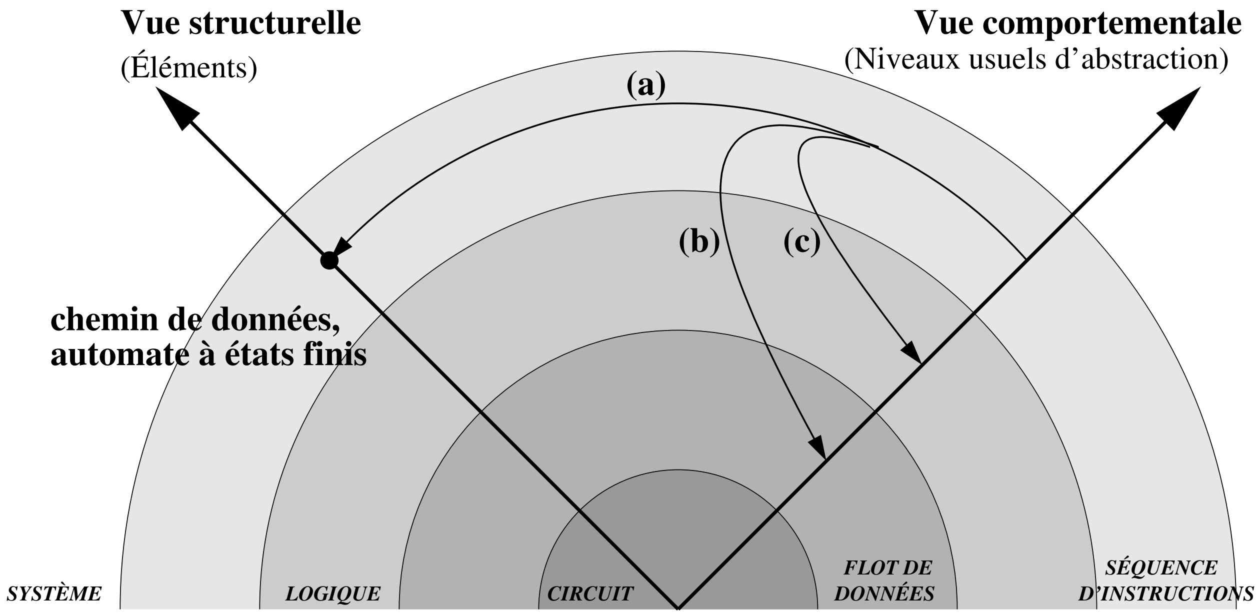


FIG. 2.2 La synthèse de haut niveau dans le flot de conception.

L'entrée de la synthèse de haut niveau est une description séquentielle non synchronisée représentant un coprocesseur. L'objectif final est d'obtenir la vue physique du circuit.

La synthèse est matérialisée dans la figure 2.2 par un arc *(a)* allant de la vue comportementale à la vue structurelle et de deux autres (*(b)* et *(c)*) retournant à la vue comportementale. Il symbolise la production d'une vue structurelle comprenant les descriptions comportementales synchronisées d'un chemin de données (DP) et d'un automate à états finis qui envoie des commandes vers le chemin de données (FSM).

Les applications visées

Le type d'application visé est une application dont le comportement est très dépendant des données (control dominated). Elle se caractérise par des traitements fonctionnels irréguliers comportant de nombreuses instructions de branchement avec des tests sur les données (un décodeur de Huffmann par exemple).
Elle diffère des applications pipelinées peu dépendantes des données qui ont un fonctionnement très régulier telle une DCT (Discret Cosinus Transform).

Les entrées et les sorties de la synthèse

Le niveau d'entrée que nous nous fixons est une description séquentielle non synchronisée décrivant le comportement d'un coprocesseur spécialisé.

L'entrée de la synthèse est écrite dans un langage impératif ne comportant aucune référence à une horloge. Le niveau de la sortie est une vue structurelle cadencée par une horloge comprenant un chemin de données ainsi qu'un automate de commandes.

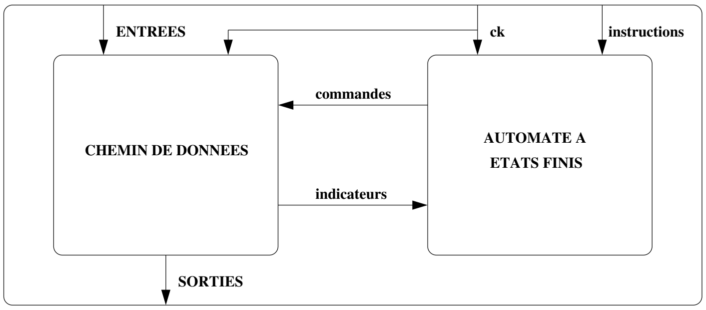


FIG. 2.3 Cible de la synthèse de haut niveau.

La figure 2.3 décrit de manière schématique la structure de la sortie de la synthèse. Le chemin de données est contrôlé par l'automate à états finis. Cet automate envoie des commandes sur le chemin de données reproduisant les opérations fonctionnelles décrites dans la spécification comportementale. Le chemin de données effectue l'ensemble des traitements sur les données.

Le chemin de données contient deux types d'opérateurs matériels :

- les opérateurs combinatoires (logiques ou arithmétiques).
- les composants séquentiels (éléments mémorisants de type registres, banc de registres, etc...).

Exploration de l'espace des solutions

Le concepteur a une vision globale du résultat qu'il veut obtenir. Son premier objectif est d'arriver à une réalisation matérielle dont les caractéristiques physiques tels que les temps de propagation, la surface, la puissance consommée, ne soient pas rédhibitoires pour le circuit.

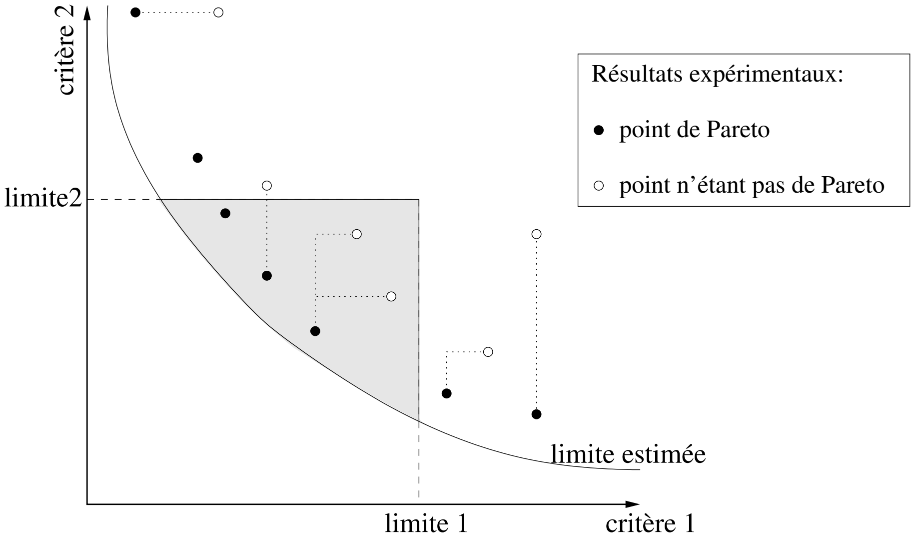


FIG. 2.4 Espace des solutions à deux dimensions.

La figure 2.4 représente un espace des solutions proposées par un outil par rapport à deux critères quelconques (par exemple : la surface et la puissance consommée). Un *point de Pareto* est un point pour lequel il n'existe aucune solution meilleure atteignable par l'outil sur les axes de l'espace des solutions [Mic94a]. La courbe de la *limite estimée* représente le résultat "idéal" estimé. Elle peut représenter, par exemple, une conception "à la main" par un concepteur expérimenté. Le concepteur pose des contraintes sur chaque axe pour délimiter l'espace des solutions acceptables (surface en grisé sur la figure 2.4).

Nous cherchons à mesurer la qualité de l'outil dans l'espace des solutions restreint (surface grisée) que cible le concepteur. La distance des *points de Pareto* à la courbe "idéale" nous informe sur l'efficacité de l'outil.

Cet espace ne se limite souvent pas à deux critères, explorer l'espace des solutions revient à considérer l'hyper-espace décrite par les *points de Pareto*. L'objectif de la synthèse de haut niveau est de pouvoir couvrir cet espace des solutions en obéissant aux directives du concepteur.
Comme pour une démarche "à la main", le concepteur doit pouvoir contrôler les critères d'optimisation de son circuit. De plus, l'outil a pour tâche de produire le résultat le plus efficace possible dans l'espace des solutions acceptables.

2.2.2 Environnement du coprocesseur

Le coprocesseur synthétisé doit s'insérer dans un ensemble matériel déjà existant ou caractérisé. Il en résulte un certain nombre de contraintes à respecter par le coprocesseur synthétisé.

Communications fonctionnelement asynchrones

Pour la plupart des applications visées par les systèmes intégrés sur puce (télécommunication, automobile, vidéo), le comportement dynamique du système auquel appartient le coprocesseur est imprédictible. Tout au plus, le comportement du système peut imposer une borne maximale sur le nombre de cycles autorisés pour réaliser un traitement. Nous considérons donc que les communications avec l'environnement sont fonctionnelement asynchrones : cela signifie que les lectures et les écritures effectuées par le coprocesseur doivent être bloquantes.

Remarque : Un système fonctionnelement asynchrone peut être électriquement synchrone (les composants ont la même horloge). Ce sont les communications entre les différents composants du système qui sont asynchrones.

Fréquence d'horloge

Le résultat de la synthèse est un circuit logique synchrone cadencé par une horloge.

La chaîne longue est le chemin fonctionnel entre deux registres où le temps de propagation est le plus long en tenant compte des temps de préétablissement et de maintien des registres. Par conséquent, la fréquence de fonctionnement est l'inverse du temps de propagation de la chaîne longue.

Le système dans lequel est plongé le coprocesseur impose son horloge au coprocesseur synthétisé. Le non-respect de cette fréquence de fonctionnement est rédhibitoire pour l'intégration de ce circuit. Du fait de l'importance de ce critère, l'outil *ugh* prend en compte la fréquence de l'horloge comme une contrainte d'entrée et non comme un résultat de la synthèse. Ainsi, les résultats produits à ce stade respecteront cette fréquence. Ce qui facilitera la continuité entre la synthèse de haut niveau et les autres outils de CAO de circuits pour produire la vue physique.

- ➠ L'entrée de la synthèse de haut niveau est une description comportementale séquentielle sans référence à une horloge.
- ➠ La fréquence est une entrée de la synthèse de haut niveau.
- ➠ Les communications externes du coprocesseur sont fonctionnelement asynchrones. C'est à dire que les lectures et écritures sont bloquantes.
- ➠ Les sorties de la synthèse de haut niveau constituent un ensemble couvrant le plus largement possible l'espace des solutions.
- ➠ Chaque solution est une description structurelle synchrone, c'est à dire cadencée par une unique horloge. Elle est composée d'un chemin de données et d'un automate d'états finis.

2.3 Les principes de la synthèse de haut niveau

2.3.1 Phases de la synthèse de haut niveau

Trois phases composent la synthèse de haut niveau. Il s'agit de l'élaboration du graphe de flot de données et de contrôle, de l'allocation de ressources et de l'ordonnancement temporel au niveau cycle.

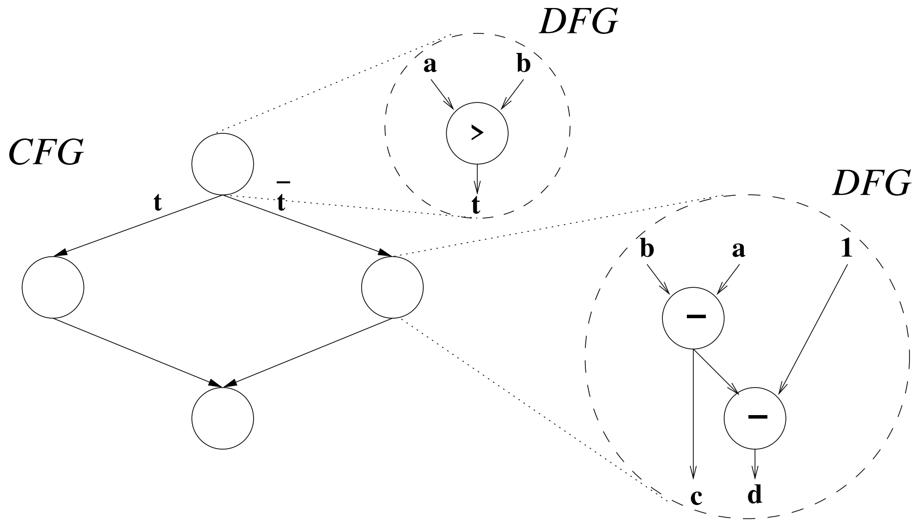
Élaboration du graphe de flot de données et de contrôle

Le but de cette phase est de construire le squelette de l'automate à états finis qui contrôle le coprocesseur et, en particulier, d'identifier les branchements conditionnels qui seront présents dans l'automate.

Le graphe de flot de données et de contrôle (CDFG) est la fusion du graphe de flot de données (DFG) et du graphe de flot de contrôle (CFG). Dans le CFG, les séquences d'opérations sans branchement sont représentées par des noeuds et les branchements entre deux séquences (boucles, sauts conditionnels) par des arcs. Le DFG représente les dépendances de données à l'intérieur d'un bloc d'opérations séquentielles.

Le CDFG représente les dépendances entre toutes les opérations fonctionnelles de la description comportementale. Dans ce graphe, chaque noeud est une opération séquentielle et chaque arc symbolise une contrainte de précédence dans l'ordonnancement des opérations fonctionnelles. Les contraintes de précédence matérialisent l'impossibilité de faire une action avant une autre pour garder la cohérence globale du comportement initial. La figure 2.5 donne une représentation graphique d'un CDFG.

```
if ( a > b )
{
   c = a – b ;
   d = c + 1 ;
}
else
{
   c = b – a ;
   d = c – 1 ;
}
e = d + a :
```



(séquence d'instructions) **(CDFG)**

FIG. 2.5 Un CDFG d'une séquence d'instructions.

La phase d'élaboration du CDFG cherche à optimiser la topologie de l'automate en vue de l'ordonnancement et de l'allocation. En effet, le CDFG peut être optimisé pour minimiser le nombre de cycles ou la surface occupée par le coprocesseur. Certains DFG composant le CDFG peuvent être regroupés en un seul DFG.
Les changements de topologie du CDFG peuvent être des déroulements de boucles, des branchement câblés dans le chemin de données, etc... Deux DFG séparés par un branchement (dans l'exemple de la figure 2.5, il s'agit des deux noeuds du CDFG séparés par les conditions t et $\overline{t}$) peuvent être réunis pour former un branchement câblé.

Le regroupement des DFG supprime les branchements dans l'automate à états finis. L'ordonnancement est plus efficace sur un gros DFG que sur plusieurs petits DFG. Le regroupement des DFG optimise donc le nombre de cycles.
D'un autre côté, le regroupement de DFG transfère le test de branchement dans le chemin de données et augmente la complexité de la partie opérative. Le regroupement des DFG augmente donc la surface du coprocesseur.

L'ordonnancement

Le but de l'ordonnancement est de construire la *fonction de transition* de l'automate à états finis constituant la partie contrôle du coprocesseur.

L'ordonnancement ne remet pas en cause les branchements présents dans le CDFG. Chaque DFG du CDFG est ordonnancé séparément.

A partir des temps de propagation des opérateurs matériels, l'ordonnancement détermine les frontières temporelles des cycles sur les opérations fonctionnelles du DFG [Mic94b]. Chaque cycle nécessite des opérateurs matériels pour réaliser les opérations fonctionnelles de la description comportementale. Plusieurs opérations fonctionnelles peuvent être parallélisées dans un même cycle.

Remarque : Le nombre minimal d'opérateurs matériels (combinatoires et séquentiels) nécessaires pour faire le chemin de données est connue à la fin de la phase d'ordonnancement.

Allocation matérielle

Le but de l'allocation est la définition de la structure du chemin de données du coprocesseur.

L'allocation sert à déterminer le nombre d'opérateurs matériels nécessaires ainsi que leurs types et la façon dont ils sont interconnectés. L'allocation peut soit ajouter un opérateur, soit partager un opérateur déjà existant en liant cet opérateur à plusieurs opérations fonctionnelles de la description comportementale [GDWL92].

1. allocation et affectation des registres : détermination du nombre de registres. Les registres sont liés aux variables de la description comportementale pour stocker les données à chaque cycle. On décide à ce stade, si les variables correspondent à des registres ou à de simples signaux combinatoires.
2. allocation et affectation des opérateurs combinatoires : détermination du nombre d'opérateurs combinatoires. A chaque cycle, on connaît la liste des opérations fonctionnelles à exécuter. Pour cela, des opérateurs combinatoires sont sélectionnés et attachés à ces opérations fonctionnelles.
3. mise en place de la connectique (bus et multiplexeurs).

Chaque opérateur occupe de la surface et cela a un coût lors de la production en série. L'allocation cherche à diminuer la surface occupée par le coprocesseur. Pour ce faire, elle doit optimiser l'utilisation des opérateurs en les partageant entre différentes opérations fonctionnelles.

Remarque : Multiplexer une entrée d'opérateur pour le partager revient à ajouter de la logique combinatoire au circuit. Cela réduit le gain de surface engendré par le partage de cet opérateur.

2.3.2 Cercles vicieux de la synthèse

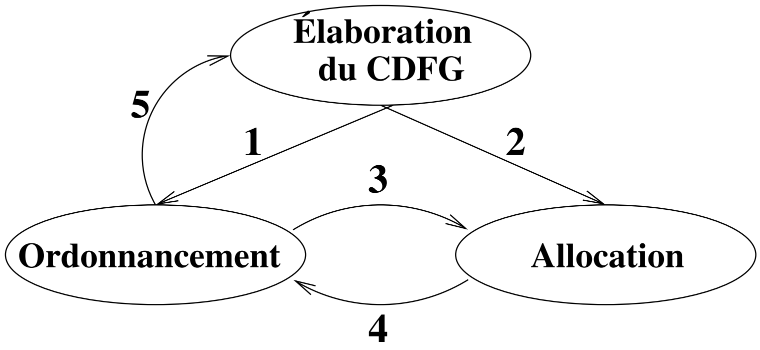


FIG. 2.6 Cercles vicieux de la synthèse de haut niveau.

Les optimisations effectuées dans les trois étapes de la synthèse de haut niveau sont très liées et les décisions ne peuvent être prises indépendamment dans chaque phase sans risquer de compromettre la qualité du résultat.

La figure 2.6 représente les dépendances entre les phases :

Arc 1 L'*ordonnancement* se base sur les DFG extraits du CDFG.

Arc 2 Le CDFG influence le nombre minimal d'opérateurs en nécessitant des opérateurs pour les branchements câblés.

Arc 3 L'allocation se base sur la parallélisation des opérations fonctionnelles pour déterminer le nombre de ressources matérielles nécessaires.

Arc 4 Dans la phase d'allocation et d'affectation, le partage d'un opérateur entre deux opérations fonctionnelles implique le plus souvent l'ajout de combinatoire sur ses entrées (multiplexage). Le partage d'un opérateur augmente le temps de propagation de ce dernier.
La phase d'allocation influence donc l'ordonnancement car il modifie l'évaluation temporelle sur laquelle se base l'ordonnancement.

Arc 5 La détermination des blocs d'instructions séquentielles dans le CDFG se fait indépendamment des contraintes temporelles des opérateurs matériels. Un mauvais ordonnancement peut alors empêcher une parallélisation efficace des opérations fonctionnelles.
L'ordonnancement doit pouvoir remettre en cause la phase d'élaboration du CDFG.

La synthèse peut toujours produire une solution de base par une démarche linéaire : construction du CDFG, ordonnancement puis allocation. Mais celle-ci est souvent inutilisable du fait des piètres caractéristiques du résultat.

Les arcs *4* et *5* de la figure 2.6 sont des phases d'optimisations des caractéristiques du circuit. Ce sont elles qui constituent les cercles vicieux.
C'est l'existence de cycles dans le graphe de la figure 2.6 qui constitue la principale difficulté de la synthèse de haut niveau.

Du fait de la complexité des algorithmes, les outils de synthèse de haut niveau sont très lents. Le travail d'exploration architecturale en est rendu plus difficile pour le concepteur.

Échapper aux cercles vicieux

Deux approches sont possibles pour traiter les dépendances croisées :

⇒ une automatisation totale de la synthèse où on itère les différentes phases de la synthèse jusqu'à obtenir une solution satisfaisante. Elle peut éventuellement impliquer des heuristiques pour converger plus rapidement vers une solution.

⇒ des informations supplémentaires fournies par l'utilisateur pour casser les cycles de dépendance et réduire le nombre de choix.

L'outil de synthèse de haut niveau *ugh* s'appuie sur un ajout d'informations de la part de l'utilisateur. Le concepteur doit définir le nombre d'éléments mémorisants. Chaque registre est identifié dans la description comportementale. La synthèse ne pourra pas ajouter d'autres registres pour construire l'architecture cible. D'emblée, l'affectation des registres de la phase d'allocation (cf. chapitre 2.3.1) est résolue.

2.3.3 Multiplicité et antagonisme des objectifs

L'évaluation d'une solution est assez complexe car elle est multi-critères. Le concepteur s'attache surtout aux caractéristiques suivantes de son circuit :

- la fréquence d'horloge.
- le temps d'exécution global.
- la surface.
- la consommation.

Dès que l'on cherche à optimiser la surface, les temps de propagation ou la vitesse du coprocesseur, la synthèse de haut niveau se heurte à l'antagonisme de ces critères. L'optimisation d'un critère particulier peut avoir des répercussions néfastes sur les autres.

L'antagonisme des critères augmente la complexité de la synthèse de haut niveau. L'élaboration du CDFG peut, par exemple, choisir d'optimiser la surface ou bien le temps d'exécution global.

Échapper à la multiplicité des objectifs

Pour simplifier la synthèse, nous décidons de réduire le nombre de critères caractérisant la solution pour avoir un objectif plus simple.

Le changement de fréquence, de surface et de temps d'exécution influence la consommation. Si l'utilisateur vise un coprocesseur à faible consommation, il devra, soit réduire la fréquence de fonctionnement, soit cibler une surface plus réduite. Nous faisons l'hypothèse que le changement de la fréquence ou de la surface permet de contrôler la consommation. La consommation n'est plus un objectif de la synthèse mais devient une conséquence des choix d'optimisation des autres critères.

La fréquence d'horloge étant une contrainte d'entrée de notre outil de synthèse de haut niveau, il reste deux degrés de liberté qui sont la surface de silicium et le temps d'exécution (en nombre de cycles).

- ➠ L'outil de synthèse haut niveau doit être suffisamment rapide pour permettre l'exploration architecturale.
- ➠ Pour réduire la complexité de la synthèse de haut niveau, nous faisons le postulat que l'affectation des registres est faite. Les registres sont identifiés et liés aux opérations fonctionnelles de la description comportementale.
- ➠ Nous supposons que la consommation est gouvernée par la fréquence et la surface. La consommation ne fait pas partie des objectifs directs de la synthèse.
- ➠ L'espace des solutions se réduit aux critères suivants :
  - le temps d'exécution global.
  - la surface.

2.4 Les problèmes ouverts

2.4.1 Comment restreindre l'espace des solutions ?

Nous avons vu dans le chapitre 2.2.2 que la fréquence d'horloge est une contrainte d'entrée pour la synthèse de haut niveau. Dans le chapitre 2.3.2, nous décidons aussi pour des raisons algorithmiques de définir explicitement les registres. D'autres informations sont-elles nécessaires pour ne pas trop s'écarter d'une conception "à la main" idéale, tout en permettant une exploration efficace de l'espace des solutions ?

La description comportementale et la fréquence ne sont pas des entrées suffisantes pour circonscrire le domaine des solutions. Il doit y avoir directives supplémentaires pour mieux correspondre aux objectifs que s'est fixé le concepteur. A savoir :

- le temps d'exécution global.
- la surface.

Comment cibler un intervalle dans cet espace ?
Le concepteur peut fournir à l'outil de haut niveau plus de contraintes sur la surface du circuit ou sur sa vitesse.
Par exemple, en plus du nombre de registres, le concepteur peut fournir l'affectation de ces registres. Il peut aussi fournir des informations sur la taille en nombre de bits des opérateurs du chemin de données. Cela permet de cibler plus précisément la surface du circuit à synthétiser.

Les informations supplémentaires données à l'outil pour cibler précisément une solution coûte du temps au concepteur. Il faut veiller à ce que ce coût soit le plus faible possible.
Quelles sont les informations indispensables à la réduction de l'espace des solutions ?

✓ Comment réduire le champ des solutions que donne la synthèse de haut niveau ?

✓ Quelles informations supplémentaires faut-il ajouter à la description pour qu'elle soit synthétisable dans l'espace des solutions visées ?

2.4.2 Qu'apporte la connaissance des registres sur l'algorithme de la synthèse ?

Nous avons vu que pour restreindre l'espace des solutions, nous pouvions fournir l'affectation des registres à l'outil de synthèse de haut niveau. Quelles en sont les implications dans l'algorithme de la synthèse de haut niveau ?

Pour éviter le cercle vicieux de la synthèse et produire un résultat rapidement, le chapitre 2.3.2 pose une hypothèse forte : les registres sont connus. Le fait que la synthèse haut niveau connaisse une partie de l'architecture cible nous ramène à un problème de compilation logicielle.

R1 ⟵ `R10 + R11` `--` *(a)*
`R2` ⟵ **R1** `+ 1` `--` *(b)*
R1 ⟵ `R10 + R12` `--` *(c)*
`R3` ⟵ **R1** `+ 1` `--` *(d)*

FIG. 2.7 Partage du registre **R1**.

En compilation, le partage d'éléments mémorisants illustré par la figure 2.7 entraîne des contraintes sur l'utilisation de ces registres. Les différents accès au registre *R1* en lecture ou en écriture ne peuvent être ordonnancés de n'importe quelle façon. Pour garder la cohérence de la description initiale, on ne peut, par exemple, intervertir l'instruction *(b)* et l'instruction *(c)*. La valeur contenue dans *R1* serait alors remplacée avant d'avoir pu être lue.

Ces contraintes d'ordonnancement ne sont pas évoquées dans la synthèse de haut niveau classique car les registres ne sont pas identifiés dans la description comportementale.

Dans notre synthèse ciblée de haut niveau, les registres sont déjà affectés. Le partage éventuel de ces registres nous amène à considérer les mêmes contraintes sur l'ordonnancement que pour la compilation logicielle. La construction du CDFG doit respecter les contraintes de précédence issues du partage des registres.

La connaissance des registres a aussi des répercussions sur l'algorithme d'ordonnancement et d'allocation. Dans le chapitre 2.4.1, nous nous demandons s'il faut ajouter des informations pour mieux cibler la solution. Est-ce que ces informations simplifieront la complexité de l'algorithme ?

✓ Est-ce que l'affectation des registres résout tous les cercles vicieux de la synthèse ?

✓ Quelles sont les contraintes issues de la compilation logicielle dont la synthèse hérite ?

✓ Est-ce que l'affectation des registres est suffisante pour produire un algorithme simple et efficace ?

2.4.3 Quelle formes ont les communications externes ?

Dans le chapitre 2.2.2, nous constatons que les communications du coprocesseur sont fonctionnelement asynchrones avec l'environnement. Les accès aux ports d'entrées/sorties doivent être encapsulés par des opérations de synchronisation. Pour garantir la compatibilité avec l'environnement, le protocole de communication doit être suffisamment répandu ou assez simple pour être adapté via une interface. Doit-on laisser le choix du mode d'accès à l'utilisateur ?

- ✓ Quel protocole de communication respectant les contraintes de l'asynchronisme choisir ?
- ✓ Devons-nous laisser leur description à la charge de l'utilisateur ?

2.4.4 Comment prendre en compte les caractéristiques électriques du circuit ?

L'un des objectifs de la synthèse de haut niveau est de prendre en compte les caractéristiques électriques de la bibliothèque de cellules cibles.

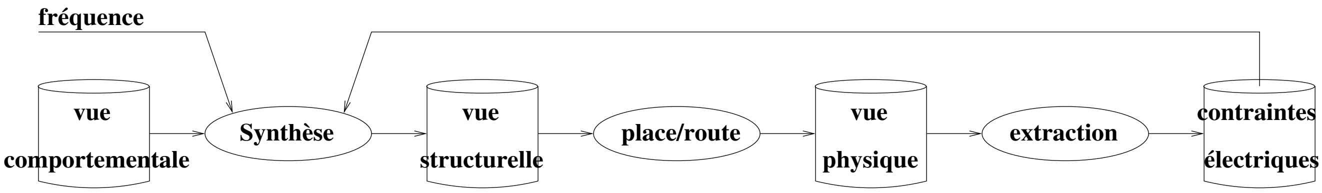


FIG. 2.8 Dépendance de la synthèse par rapport aux caractéristiques physiques

Les contraintes électriques sont obtenues une fois le circuit placé et routé. Elles sont constituées des temps de propagation des opérateurs combinatoires et des temps de préétablissement et de maintien des registres et des bancs mémoire.

Pour respecter la fréquence d'horloge, l'outil de synthèse de haut niveau doit adapter son ordonnancement en fonction des caractéristiques temporelles des opérateurs du chemin de données. Or ces caractéristiques physiques ne sont connues qu'une fois obtenue la vue physique du coprocesseur (figure 2.8). Il y a donc une dépendance mutuelle entre la phase de synthèse et la phase placement/routage.

- ✓ Existe-t-il une méthode efficace qui permette de tenir compte des contraintes électriques pour la synthèse de haut niveau ?
- ✓ Comment adapter l'ordonnancement à cette méthode ?

2.4.5 Comment interfacer l'outil de synthèse haut niveau avec les outils de synthèse RTL ?

Dans le domaine de la CAO de circuit, il existe de nombreux outils de synthèse performants au niveau RTL (Register Transfer Level). L'objectif de la synthèse de haut niveau n'est pas de les réinventer mais d'être inter-opérable avec eux.

Le résultat de la synthèse de haut niveau est une description synchrone de niveau RTL.
D'autres outils de CAO de circuit sont censés prendre en entrée les résultats de la synthèse haut niveau et il est inutile de prendre des décisions qui seront remises en cause par ces outils. Où doit s'arrêter la synthèse de haut niveau pour laisser la place à des outils de synthèse spécialisés ?

Les outils de synthèse de bas niveau sont avantagés par rapport aux outils de synthèse de haut niveau. Par exemple, la phase de projection structurelle de la synthèse logique connaît les caractéristiques électriques intrinsèques des cellules qu'elle utilise pour réaliser les fonctions booléennes. Ses algorithmes sont ciblés pour cette utilisation et sont très efficaces.

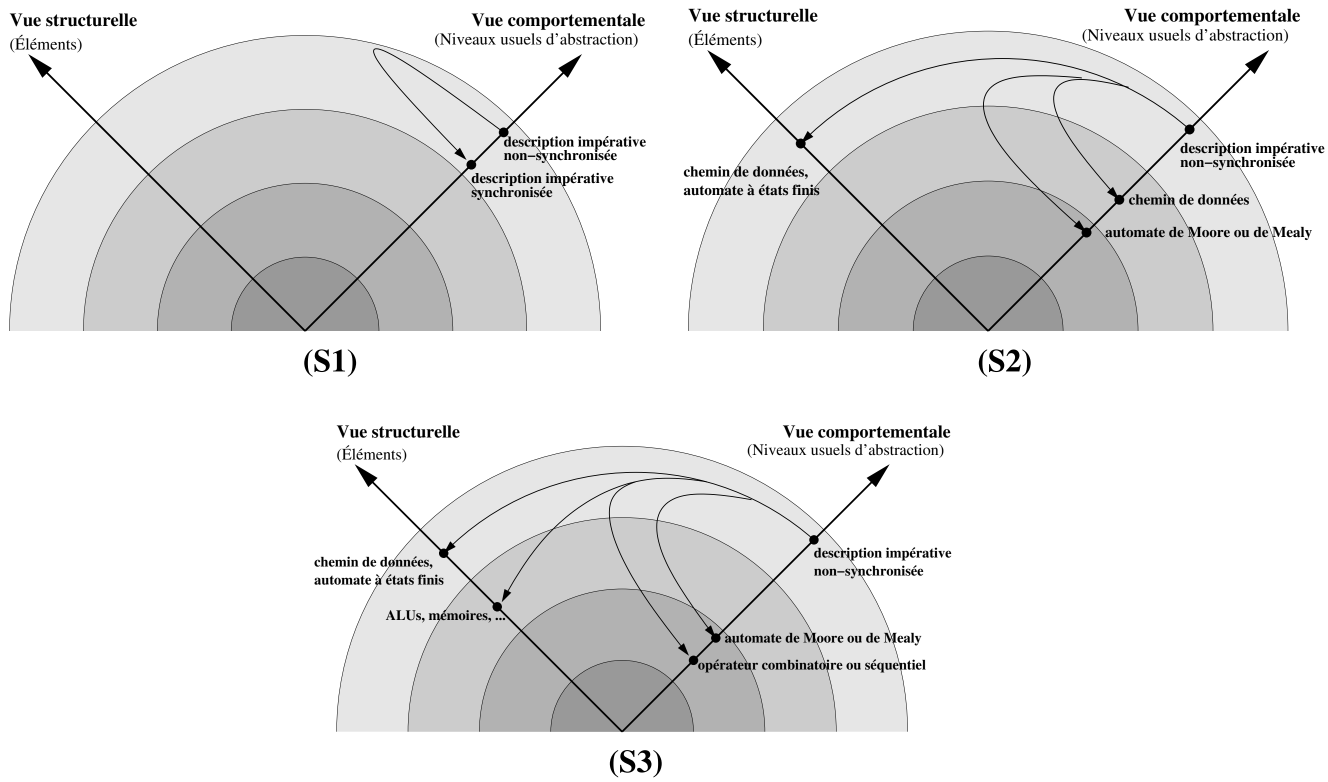


FIG. 2.9 Trois niveaux de sortie pour la synthèse de haut niveau.

Nous considérerons donc plusieurs niveaux de sorties possibles décrites dans la figure 2.9 :

- ⇒ La sortie *S1* est une description comportementale de niveau RTL où la partie opérative n'est pas dissociée de l'automate d'états. La synthèse de haut niveau n'a déterminé que les *fonctions de transition et de génération* de l'automate d'états. L'allocation des ressources est laissée aux autres outils de synthèse.
- ⇒ La sortie *S2* est une vue structurelle constituée de deux descriptions comportementales de niveau RTL (DP et FSM). La partie opérative est donc séparée de l'automate. L'allocation des ressources est laissée aux autres outils de synthèse.
- ⇒ La sortie *S3* est une vue structurelle constituée d'un DP et d'un FSM tous deux de niveau RTL. L'automate est une description comportementale et le chemin de données une vue structurelle composée d'instances d'opérateurs. L'allocation des ressources est faite par la synthèse de haut niveau.

La synthèse haut niveau ne produit pas de vue structurelle de l'automate. C'est une tâche inutile puisque la synthèse d'automate produit un résultat largement satisfaisant.

La différence entre *S1* et *S2* est la séparation du chemin de données de l'automate d'états. La sortie *S1* fusionne les deux comportements pour donner une description synchronisée du coprocesseur. *S1* et *S2* de la figure 2.9 ne font pas d'allocation du chemin de données alors que *S3* va jusqu'à l'affectation des opérateurs (cf. chapitre 2.3.1).

Suivant le type de sortie, le flot de conception et le résultat final ne seront pas les mêmes. Nous avons vu dans le chapitre 2.4.4 qu'il faut faire intervenir les contraintes électriques lors de la synthèse. La forme de la sortie influence-t-elle sur la manière de faire intervenir ces contraintes ?

- ✓ Quel est le niveau de description de la sortie de la synthèse de haut niveau le plus adapté aux autres outils de CAO ?
- ✓ Est-ce que le choix d'un niveau de sortie à des répercussions sur les possibilités de faire intervenir les contraintes électriques ?

2.5 Conclusion

Notre outil de synthèse de haut niveau prend en entrée :

- ➠ la description comportementale non synchronisée d'un coprocesseur.
- ➠ la fréquence du coprocesseur.
- ➠ les registres du coprocesseur.

Notre outil de synthèse de haut niveau génère en sortie :

- ➠ un coprocesseur avec des communications fonctionnelement asynchrones. Il est caractérisé par sa surface et son temps d'exécution global.

L'objectif de cette thèse est de répondre aux problèmes que posent ces conditions :

- ✓ Comment réduire le champ d'investigation de la synthèse de haut niveau ?
  Quelles sont les caractéristiques d'un langage synthétisable ?
  Comment prendre en compte les desiderata du concepteur ?
- ✓ Comment modéliser les communications asynchrones du coprocesseur ?
- ✓ Quelles sont les contraintes de réordonnancement des accès aux registres ?
  Est-ce que des informations supplémentaires peuvent simplifier les algorithmes de la synthèse haut niveau ?
- ✓ Comment tenir compte des caractéristiques électriques ?
- ✓ Comment insérer la synthèse de haut niveau dans le flot de conception ?
  Peut-on toujours tenir compte des contraintes électriques ?

Chapitre 3

État de l'art

Sommaire

Depuis les années 1970, des outils de synthèse de haut niveau ont été développés aussi bien dans les milieux académiques qu'industriels. Nous présentons dans ce chapitre les différentes approches existantes et les techniques sur lesquelles elles s'appuient.

En effet, un outil de synthèse est avant tout un algorithme. Son type est déterminant dans l'efficacité du traitement. Pour simplifier la complexité du problème, il est la plupart du temps scindé en deux parties : une partie pour extraire l'ordonnancement des opérations fonctionnelles, une autre pour déterminer les opérateurs matériels nécessaires à ces opérations.

La description des méthodes générales utilisées pour ces deux aspects de la synthèse de haut niveau correspondent respectivement aux deux premiers sous-chapitres.
Nous verrons ensuite leur mise en application dans les outils de synthèse et l'interaction que ces derniers ont avec le concepteur.

3.1 Les algorithmes d'ordonnancement

L'objectif de l'ordonnancement est de déterminer à quel cycle s'effectue les opérations fonctionnelles contenues dans la description comportementale du circuit. Il détermine donc le parallélisme des opérations et influence fortement la surface du circuit.

Ils existent quatre familles d'algorithmes d'ordonnancement :

1. *algorithmes de programmation linéaire en nombre entiers* : reformulent le problème d'ordonnancement comme un problème de programmation linéaire au moyen d'inéquations dues aux contraintes. Ils fournissent une solution optimale [CWM94].
2. *algorithmes neuronaux* : modélisent le problème d'ordonnancement par un réseau de neurones trouvant une solution par auto-réorganisation [Phi92].
3. *les algorithmes itératifs/constructifs* : répartissent les opérations dans le temps les unes après les autres suivant un critère de priorité sur chacune d'elles.

Pour la synthèse de haut niveau, le nombre d'opérations à ordonnancer est très important, les outils de synthèse ont besoin d'algorithmes rapides. Les *algorithmes de programmation linéaire en nombre entier* et les *algorithmes neuronaux* sont inadaptés à notre problème car ils entraînent des modèles mathématiques très complexes.
Les algorithmes que nous développons dans ce chapitre sont itératifs.

Certains algorithmes se limitent à l'ordonnancement d'un flot de données, d'autres sont plus orientés flot de contrôle. Cependant, ils obéissent tous au mêmes types de contraintes. [WC95] nous présente les différents types de contraintes sur l'ordonnancement :

- non contraint : seules les contraintes de précédence sur les données ordonnent les opérations.
- sous contrainte temporelle : l'ordonnancement doit obéir aux contraintes de précédence et le temps d'exécution de la tâche ne doit pas dépasser la limite temporelle.
- sous contrainte de ressource : l'ordonnancement doit obéir aux contraintes de précédence et ne pas dépasser un nombre maximum d'instanciations pour chaque type d'opérateur matériel.

- sous contrainte temporelle et de ressource : l'ordonnancement doit obéir aux contraintes de précédence, ne pas dépasser la limite temporelle et le nombre maximum d'instanciations d'opérateurs matériels.

Tous les types d'algorithmes d'ordonnancement respectent les contraintes de précédence afin de respecter la fonctionnalité des opérations.

ASAP et ALAP

Ils s'appliquent à des graphes de flots de données.
ASAP (As Soon As Possible) est un ordonnancement des opérations au cycle le plus tôt en ne tenant compte que des contraintes de précédence.
ALAP (As Late As Possible) est un ordonnancement des opérations au cycle le plus tardif sans que cela change le nombre total de cycles par rapport à un ordonnancement ASAP.
Ces algorithmes sont très rapides. Ils donnent un ordonnancement optimal en nombre de cycles mais coûtent cher en surface.

La mobilité d'une opération est égale à $cycle_{ALAP} - cycle_{ASAP}$, où $cycle_{ALAP}$ et $cycle_{ASAP}$ sont les cycles d'ordonnancement de l'opération, pour respectivement, un algorithme ALAP et un algorithme ASAP.

Ordonnancement par liste de priorité

L'ordonnancement par liste, ou *list scheduling*, cherche à minimiser le nombre de cycles. Cet algorithme prend en compte une contrainte sur les ressources.

Dans une suite d'opérations fonctionnelles sans branchement conditionnel, l'algorithme choisit à chaque cycle l'opération la plus prioritaire (tout en respectant les contraintes de précédence) [PG87]. La priorité des opérations peut être évaluée une fois pour toute au début de l'ordonnancement ou réévaluée à chaque cycle.

En général, les priorités utilisées sont la mobilité ou le nombre de successeurs de l'opération.

Ordonnancement basé sur les distributions

Cet algorithme aussi appelé *force directed scheduling*, cherche à minimiser le nombre de ressources. Il peut prendre une contrainte temporelle pour forcer la parallélisation des opérations.

Le principe de l'algorithme est de reporter les opérations les plus mobiles à des cycles nécessitant moins de ressources [PK89].
L'opération fonctionnelle est placée sur le cycle qui minimise le nombre total d'opérateurs.
L'estimation du nombre d'opérateurs est basée sur la mobilité des opérations.

La probabilité d'exécution d'une opération logicielle dans un cycle donné est l'inverse de sa mobilité. Le nombre d'opérateurs matériels est estimé pour chaque type d'opérateur et pour chaque cycle. Pour chaque type d'opérateur matériel, on ne considère que les opérations logicielles pouvant y être affectées.

Le nombre d'opérateurs nécessaires pour un cycle donné est la somme des probabilités d'exécution de chaque opération dans ce cycle.
Le nombre global d'opérateurs est le nombre qui satisfait la contrainte estimée pour chaque cycle.

Cet algorithme est plus lent car sa fonction de coût est complexe. Il reste un bon compromis entre le degré d'optimisation de la surface et le temps de calcul.

Ordonnancement basé sur les chemins d'exécution

Ce type d'algorithme est aussi appelé *path-based scheduling* [Cam91].
Au lieu de considérer une partie de la description, ces algorithmes considèrent l'intégralité du flot de description au moyen de CDFG (graphe mélangeant les flots de données au flot de contrôle).

Le flot de description est alors considéré dans son intégralité.
L'algorithme considère pour cela tous les chemins d'exécution.

Toutes les boucles dans le flot de contrôle sont cassées de manière à obtenir un graphe de contrôle sans cycle. On extrait les différents chemins d'exécution de ce graphe.
Le noeud initial de la description est le départ d'un chemin. Chaque boucle est aussi le départ d'un chemin car il peut y avoir plusieurs itérations sur la boucle. Donc s'il y a *n* boucles, il y a *n+1* départs de chemin.
Chaque branchement conditionnel donne deux chemins différents. Par exemple dans la figure 3.1, il y a un chemin passant par les *noeuds 3 et 4* et un chemin passant par le *noeud 5*.

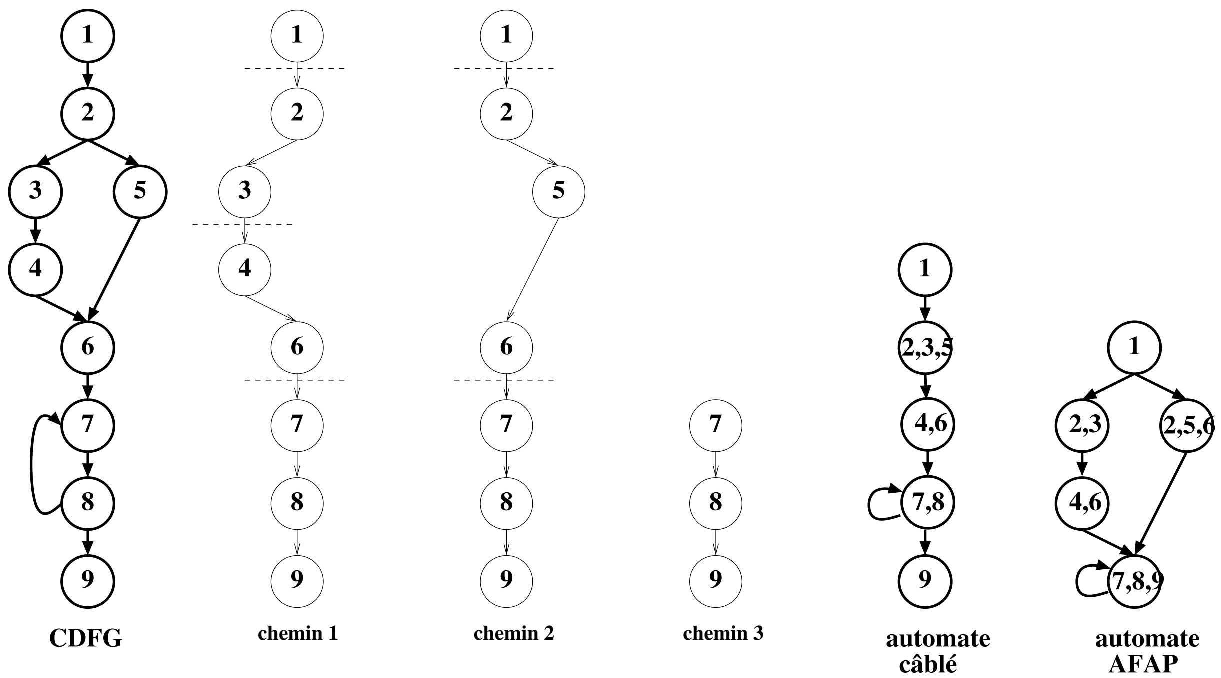


FIG. 3.1 Extraction des chemins du CDFG.

Pour chaque chemin d'exécution, on détermine les groupes d'opérations logicielles pouvant s'effectuer dans le même cycle. Les coupures sont placées en parcourant le chemin séquentiellement.

Un groupe courant est créé. Pour chaque opération du parcours :

- si l’opération n’est pas en conflit avec les autres opérations du groupe courant, on l’ajoute au groupe courant.
- sinon on l’ajoute à un nouveau groupe qui devient le groupe courant.

Ainsi une opération logicielle ne peut être associée qu’à l’opération suivante ou à l’opération précédente.
Les conflits sont déterminés à partir de contraintes de ressources (nombre d’opérateurs combinatoires ou conflit d’écriture sur les registres par exemple).

Tous les chemins sont ensuite réunis pour constituer l’automate d’états.
L’exécution d’opérations n’appartenant pas au même branchement sont mises sous des conditions d’exclusivité (câblage) pour être ordonnancées dans le même cycle. Les opérations intégrées dans une boucle sont dépendantes du test de fin de boucle.
Chaque coupure de chemin est répercutée dans l’automate d’états.

L’AFAP (As Fast As Possible) est un algorithme basé sur les chemins d’exécution qui produit l’automate le plus rapide en temps d’exécution. Mais les états de l’automate sont répliqués. La surface du circuit généré est souvent importante.

Cet algorithme a une complexité non polynomiale et est donc lent sur de gros circuit. En effet, un branchement multiplie le nombre de parcours d’exécution par deux et ceci pour chaque branchement !

➠ Il existe des algorithmes d’ordonnancement très rapides mais qui ne produisent pas de circuits exploitables (surface ou temps de propagation trop grands).

➠ Il existe des algorithmes d’ordonnancement produisant une solution optimale mais leur complexité n’est pas polynomiale.

➠ L’ordonnancement est dissocié de l’affectation des opérations fonctionnelles. Or l’affectation détermine les connexions (multiplexeurs, bus) qui influencent considérablement les temps de propagation des opérateurs. Les algorithmes d’ordonnancement n’en tiennent pas compte lorsqu’ils placent les frontières de cycle.

3.2 Les algorithmes d’affectation

L’objectif de l’affectation des opérations est de partager les ressources afin de minimiser la surface. L’affectation est fortement dépendante de l’ordonnancement puisqu’il détermine l’allocation minimale.

On peut utiliser un algorithme général de type *branch and bound* qui explore l’arbre d’exploration des solutions. Chaque noeud est estimé grâce à une fonction de coût globale. Si ce noeud a un coût supérieur à la limite, on peut supprimer ce noeud et tous ses fils de l’arbre d’exploration.
Ces algorithmes ont une complexité non polynomiale. Ce type d’algorithme est utilisé pour l’allocation de ressources [Pan88] [SMT^{+}92] mais aussi pour l’ordonnancement [OJ92]. La difficulté de ce type d’algorithme est de trouver la bonne fonction de coût qui permettra une heuristique rapide et réaliste.

Pour la réallocation, on peut aussi utiliser la technique du *recuit simulé* inspirée d'un modèle de simulation physique.
Les affectations des opérations sont permutées tour à tour en conservant ou non les solutions intermédiaires. La solution est conservée suivant son coût et une probabilité autoritaire dont l'influence va décroissante. Au début des permutations, la probabilité est prépondérante sur le coût : des solutions intermédiaires très coûteuses peuvent être conservées. Au fur et à mesure des permutations, la probabilité diminue et le critère de coût devient prépondérant dans la sélection d'une solution.
Cette technique permet de sortir de minimums locaux et d'obtenir une bonne approximation de la solution optimale.

3.2.1 Affectation des opérateurs combinatoires

Partitionnement en cliques

Soit $G = (N,A)$ un graphe non orienté avec N un ensemble de noeuds et A un ensemble d'arcs.
Une clique est un ensemble de noeuds qui forme sous-graphe complet de G.

La formulation de l'allocation d'opérateurs matériels pour le problème de partitionnement en cliques est :

- chaque opération fonctionnelle est un noeud
- un arc est présent entre deux noeuds (i.e deux opérations) si
  1. ces deux opérations ne sont pas exécutées dans le même cycle
     et
  2. ces deux opérations peuvent s'exécuter sur le même opérateur matériel.

Un noeud peut appartenir à des cliques différentes : une opération fonctionnelle peut être affectée à des opérateurs matériels différents.
Une solution est trouvée lorsque chaque noeud de G appartient à une seule clique : toutes les opérations sont affectées. Le nombre de clique est alors le nombre d'opérateurs matériels.

L'algorithme peut être utilisé pour n'importe quel type de ressource matérielle. [RS91] présente un algorithme d'affectation des registres qui minimise le nombre de registres et prend en compte le coût en surface des interconnexions.

Le problème du partitionnement est donc de trouver le nombre minimal de cliques tel que chaque noeud de G appartienne à une seule clique. Le but est de minimiser le nombre d'opérateurs matériels. Ce problème a une complexité non polynomiale. [TS86] a donné une heuristique efficace avec un temps polynômial. Il consiste à privilégier les cliques contenant le plus de noeuds : pour chaque noeud, on ne conserve que les arcs le liant à la plus grosse clique.

3.2.2 Affectation des registres

Left edge

Cet algorithme vient du routage canal d'une vue physique. Il est couramment utilisé pour l'*affectation des registres*. L'une des premières utilisations pour la synthèse est dans [KP87].

Il est basé sur un choix arbitraire pour chaque noeud de l'arbre de décision. L'algorithme se comporte comme suit :

1. les valeurs sont triées par ordre d'apparition (naissance) dans le graphe d'ordonnancement. Dans la figure 3.2, la durée de vie des valeurs est modélisée par un *schéma de Gantt*.
2. un registre est alloué.
3. la liste des valeurs est parcourue dans l'ordre chronologique des naissances : une valeur est affectée au registre si elle n'entre pas en conflit de dates avec les autres valeurs déjà affectées à ce registre.
4. Si la liste de valeurs n'est pas vide, il faut repartir à l'*étape 2*.

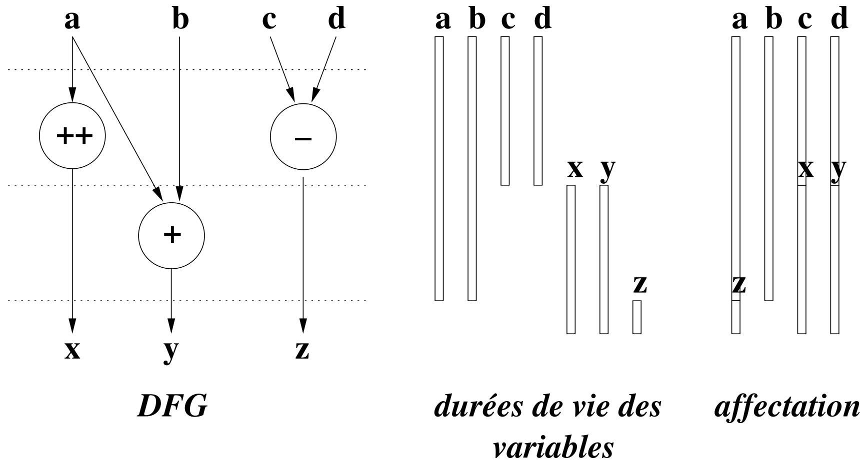


FIG. 3.2 Exemple d'application de l'algorithme pour l'affectation des registres.

L'algorithme garantit l'allocation d'un nombre minimum de registres dans un temps très court mais possède deux désavantages :

- l'algorithme n'est pas adapté aux boucles et aux branchements conditionnels car il n'est pas adapté au CDFG. Les durées de vie des variables dépassent la portée des DFG. De plus dans les branchements, les affectations de registres ne sont pas indépendantes.
- l'algorithme ne prend pas en compte le coût des multiplexeurs.

Comparaison bipartite par poids

[HCLH90] est une variation de l'algorithme du *left edge*, elle permet d'introduire des poids dans les affectations.

L'algorithme se comporte comme suit :

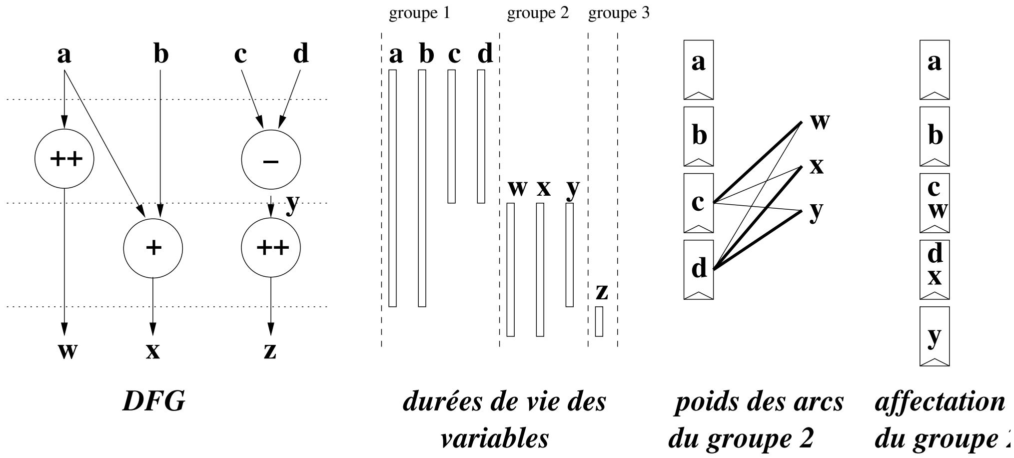


FIG. 3.3 Exemple d'application de l'algorithme pour l'affectation de registres.

1. les valeurs sont triées par ordre d'apparition (naissance) dans le graphe d'ordonnancement (cf. figure 3.3).
2. la liste ordonnée des valeurs est scindée en groupes.
   Un groupe courant est créé. Pour chaque valeur de la liste parcourue :
   - Si la valeur est en conflit de dates avec toutes les valeurs du groupe courant, on ajoute cette valeur au groupe courant.
   - Sinon on ajoute cette valeur à un nouveau groupe qui devient le groupe courant.
3. Pour chaque groupe de la liste ordonnée de valeurs :
   - On établit un graphe biparti tel que les noeuds sont les registres et les valeurs appartenant au groupe et les arcs sont positionnés exclusivement entre une valeur et un registre.
     Un arc existe entre une valeur et un registre si et seulement si cette valeur n'est pas en conflit de date avec les autres valeurs déjà affectées à ce registre. Les arcs possèdent un poids inversement proportionnel à la fonction de coût de l'affectation de la valeur au registre. En général, le poids de l'arc prend en compte les coûts de la connectique (i.e multiplexeurs, bus).
   - L'affectation des valeurs est choisie de sorte à favoriser les arcs de poids les plus forts.

   Remarque : les registres sont alloués au fur et à mesure de l'affectation des valeurs dans les registres. Si une valeur ne peut être affectée dans aucun des registres (conflit de dates), un nouveau registre est alloué.

➠ L'allocation des registres est dissociée de l'allocation des opérateurs combinatoires. Or ces allocations sont très liées puisqu'elles doivent minimiser les multiplexeurs.

➠ Les algorithmes d'allocation sont des algorithmes de type glouton qui ne permettent pas d'avoir une vision globale telle que pourrait l'avoir le concepteur.

➠ L'affectation des registres produit une description de niveau RTL non compréhensible par son concepteur. En effet, les registres ne correspondent pas au variables de la description comportementale. Cela empêche tout ajustement du circuit au niveau RTL.

3.3 Les outils de synthèse haut niveau

Il y a deux types d'outil de synthèse de haut niveau :

1. les outils orientés flot de données.
2. les outils orientés flot de contrôle ou mixtes.

Les outils orientés flot de contrôle sont en fait des outils mixtes car ils sont capables de traiter des flots de données.
Le concepteur a aussi la possibilité d'utiliser plusieurs outils. Dans [BKV+96], Berrebi a utilisé un outil orienté flot de données (*cathedral 2/3* et un outil orienté flot de contrôle (*amical*) pour concevoir un estimateur de mouvement. *amical* a réutilisé le chemin de données généré par *cathedral* pour construire le circuit global. Le circuit généré a été comparé au même circuit conçu "à la main" directement au niveau RTL, le circuit généré de façon automatique est seulement 5 % plus gros.

3.3.1 Outils orientés flot de données

GAUT

GAUT est un outil développé conjointement au Laboratoire d'Analyse des systèmes de Traitement de l'Information (*LASTI*) à Lannion et au Laboratoire d'Etude des Systèmes Temps Réel (*LESTER*).
Cet outil est dédié aux applications de traitement du signal sous contrainte de temps d'exécution [MSDP93]. Il cible principalement la conception sur FPGA (Field Programmable Gate Array) mais peut aussi produire des ASICs (Application Specific Integrated Circuit).
Le circuit généré est un circuit synchrone de niveau RTL (Register Transfer Level) avec une seule horloge.
[**?**] permet de prendre en compte les contraintes de communication. L'ordonnancement tient alors compte de l'ordre des données échangées avec le système ainsi que des dates auquelles sont possibles ces échanges.

Le concepteur fournit une description comportementale en VHDL [Ins87] du circuit, une bibliothèque de modèles caractérisés, une cadence et une fréquence de fonctionnement. Cette description ne comporte pas d'informations sur les tailles des opérations fonctionnelles car pour les applications visées (de type DSP), les opérations ont généralement toutes la même taille.
La taille des données et des opérations est spécifiée dans la bibliothèque de modèles.

Le concepteur doit caractériser les modèles de la bibliothèque. L'outil utilise les caractéristiques physiques de chaque modèle pour calculer leur coût d'instanciation et respecter les contraintes du concepteur. Ces données caractérisent la surface, les temps de propagation et la consommation de chaque modèle.
Le concepteur doit donc procéder à la synthèse et, éventuellement, au placement/routage de chaque modèle afin d'en extraire les caractéristiques physiques.

Le concepteur fournit à l'outil *GAUT* une cadence et une fréquence de fonctionnement. Ces paramètres constituent le *temps d'exécution total* visé par le concepteur : c'est la contrainte temporelle pour l'algorithme d'ordonnancement. Une fois cette contrainte respectée, la surface et la consommation doivent être minimisées.

Le concepteur peut donner d'autres directives à l'outil comme le choix des fonctions de coût et d'heuristiques. Ces directives interviennent sur de multiples critères tels que l'estimation de la surface, l'algorithme d'*affectation des registres*, le nombre d'essais et l'*affectation des opérateurs combinatoires*.

Pour évaluer le nombre de cycles total de la description comportementale, *GAUT* déroule toutes les boucles. Les branchements conditionnels sont cablés, le chemin d'exécution est alors le plus long des deux branchements.
Pour respecter la cadence, l'outil ajoute des opérateurs matériels pour augmenter la parallélisation des opérations fonctionnelles et diminuer le nombre de cycles total d'exécution. L'ordonnancement est basé sur un ASAP avec contrainte temporelle.

Ensuite l'outil détermine l'*affectation des registres* à chaque cycle à partir de l'*ordonnancement*.
La construction de l'automate du circuit découle des choix faits lors de l'*ordonnancement*, de l'*affectation* et de la connectique. C'est un automate d'états très simple car il n'y a aucun branchement.

Pour minimiser les coûts en surface, le circuit généré est basé autour de bus. [JCM99b] détermine le nombre de bus de manière à minimiser les multiplexeurs. Ces bus de données servent à la communication entre les trois unités fonctionnelles décrites au niveau RTL : l'automate d'états, la partie opérative et la partie mémorisation.
Le chemin de données forme les parties opérative et mémorisation du circuit. Il est constitué d'instanciations d'opérateurs tirés de la bibliothèque de modèles.
La synthèse de l'unité de mémorisation est détaillée dans [**?**].

Les boucles non bornées ne sont pas acceptées par *GAUT*. Cela limite le champs d'application de l'outil.
Faire disparaître les branchements conditionnels de l'automate d'états permet la parallélisation des opérations de ces branchements avec les autres opérations. Cependant cela peut être pénalisant si il y a des dépendances fonctionnelles, le temps d'exécution s'aligne alors sur le chemin d'exécution le plus long.

La contrainte de cadence est inadaptée lorsque le traitement consomme plusieurs données (plusieurs lectures successives) et produit plusieurs données (plusieurs écritures successives).
La cadence est garantie entre la consommation de deux jeux d'entrées successifs, mais il n'y a pas de contrainte sur l'ordre des données, ni sur l'intervalle séparant chaque échange de données. L'environnement du circuit doit être adapté au circuit généré pour être logiquement synchrone lors de chaque échange de données.

Il n'y a pas de garantie que le circuit fonctionne à la fréquence visée par le concepteur. Car la caractérisation des modèles de la bibliothèque entraîne des imprécisions : chaque modèle de la bibliothèque a été caractérisé en dehors du contexte global du circuit.
Par exemple, la synthèse RTL peut produire des additionneurs matériels différents suivant les fonctions logiques connectées aux ports de l'instance de l'opérateur et des contraintes globales appliquées au circuit. La vue structurelle d'une instance de modèle à l'intérieur d'un circuit est souvent différente de la vue structurelle du modèle synthétisé hors du circuit.

Si l'on suppose que la fréquence visée est respectée, *GAUT* permet de cibler assez précisément un circuit en surface et en temps d'exécution total, car il minimise la surface après avoir respecté le critère de cadence.

cathedral 2/3

cathedral 2/3 est un outil développé par les laboratoires de l'IMEC et de l'ESAT situés en Belgique. [NCG91] part d'une description en *sillage* [Hil85] d'un circuit de type flot de données (DSP - Digital Signal Processing).
Le concepteur décide de la fréquence de son circuit.

Le circuit généré est constitué d'un chemin de données et d'un contrôleur. Il peut être comparé à un processeur *VLIW* avec contrôleur micro-programmé. Ce circuit est décrit en VHDL au niveau RTL.

L'outil est constitué de deux parties : *cathedral 2* et *cathedral 3*. *cathedral 2* est l'ossature du programme et *cathedral 3* un générateur d'opérateurs.
Dans *cathedral*, il existe deux types d'opérateurs : les *EXU* (Execution Unit) et les *ASU* (Application Specific Unit).

- les *EXU* sont issues de la bibliothèque de l'outil. Il peut y avoir au maximum quatre *EXU* instanciées dans le circuit. Une *EXU* peut être soit une ALU, soit une ROM. Chaque *EXU* possède un banc de registres et un registre d'instructions.
- les *ASU* sont créés par *cathedral 3* sur demande et utilisées comme opérateurs par *cathedral 2* au même titre que les *EXU*. Elles sont instanciées pour des opérations complexes qui se répètent un certain nombre de fois (le flot d'opérations fonctionnelles d'un corps de boucle par exemple). Chaque *ASU* possède un contrôleur local pour éviter la perte d'un cycle au contrôleur global. Chaque délai d'exécution de l'*ASU* est un multiple du cycle.

Il y a au moins une ROM dans l'architecture. Elle contient l'ensemble des micro-instructions à appliquées aux instances des *EXU* et des *ASU*. La largeur d'une instruction peut être de 50 à 200 bits !
Les *ASU* et *ASU* sont interconnectées entre elles grâce à des bus. Les interconnexions à l'intérieur des *EXU* et des *ASU* sont à base de multiplexeurs.

cathedral se sépare en deux étapes : la synthèse comportementale et la synthèse structurelle.

1. Allocation de la mémoire avec pour principal critère la surface. L'objectif est de paralléliser au maximum les opérations : il faut donc allouer suffisamment de mémoire et de ports mémoires pour ne pas diminuer le débit. Une fois cette contrainte respectée, il faut partager l'espace mémoire pour diminuer la surface occupée.
2. Affectation des opérations fonctionnelles sur les *EXU* disponibles. Si une opération peut être affectée à plusieurs *EXU*, des directives utilisateurs sont possibles pour influencer l'affectation.
3. Si besoin, utilisation des ASU générées par *cathedral 3* pour des opérations fonctionnelles particulières.
4. Ordonnancement basé sur un *ASAP*. Traitement des dépendances de données, des branchements conditionnels et des boucles.
5. Terminer l'allocation. Partage des registres internes.
6. Allocation des connexions.
7. Construction du code en micro-instructions.

Des contraintes supplémentaires peuvent être ajoutées pour diriger l'optimisation.

- Option permettant de réaliser le repliement de boucles.
- Insertion de plusieurs étages de pipeline dans une ASU.

3.3.2 Outils mixtes

amical

amical est un outil développé au laboratoire Techniques de l'informatique et de la Micro-électronique pour l'Architecture d'ordinateurs (TIMA) de Grenoble.

L'outil prend en entrée une description comportementale en VHDL et une bibliothèque de modèles. Il produit un circuit constitué d'un automate d'états et d'un chemin de données avec des bus.
La description comportementale est constituée d'un unique processus VHDL. Il contient des instructions séquentielles ainsi que des appels procéduraux.

Les fonctions utilisées dans la description sont des boîtes noires faisant référence aux modèles de la bibliothèque. [KDJ94] et [Ber92] montrent que l'on peut réutiliser des opérateurs déjà synthétisés par *amical* (ou un autre outil) pour effectuer la synthèse d'un circuit plus gros.
Cela permet une démarche de type ascendante qui découpe le problème en sous-problèmes plus simples et permet la réutilisation de circuits déjà synthétisés.
Cependant, le sous-circuit doit être suffisamment bien caractérisé dans la bibliothèque de modèles pour être correctement utilisé par *amical*.
Plus le sous-circuit est complexe, plus l'identification de ses caractéristiques sera malaisée. Les informations sur ces boîtes noires sont, par exemple, les opérations qu'elles sont capables d'exécuter et le protocole qu'elles suivent.

Les différentes étapes de l'outil sont le macro-ordonnancement, l'allocation des unités fonctionnelles, le micro-ordonnancement et la génération de l'architecture.

1. le macro-ordonnancement produit un automate d'états constitué de macro-cycles (i.e il ne tient pas compte des temps de propagation des opérateurs instanciés).
2. l'allocation fait appel à un algorithme de type itératif simple.
3. le micro-ordonnancement prend en compte les temps de propagation et transforme les macro-cycles en cycles réels (opérations multi-cycles).
4. la génération de l'architecture est la construction de l'automate et l'ajout de la connectique.

L'outil permet d'alterner la conception manuelle et automatisée. Le concepteur peut contrôler chaque étape de la synthèse.

La connectique n'est pas un des objectifs majeurs de *amical*. L'outil ne cherche pas à minimiser les multiplexeurs (i.e leurs nombres d'entrées). Or, les multiplexeurs jouent un rôle très important dans les temps de propagation, cela nuit aux performances ciblées, notamment à la fréquence de fonctionnement.
De plus, au niveau physique, les capacités des fils jouent aussi un rôle très important. Il y a encore plus de risques de s'éloigner de l'objectif visé.

camad

camad [PKL88] a été développé au département *Computer and Information Science* de l'Université de Linköping en Suède.

Il transforme la description comportementale VHDL en un réseau de Pétri étendu sur lequel travaille son algorithme d'ordonnancement. Chaque noeud du réseau est une suite d'opérations fonctionnelles sans branchement (i.e un flot de données) de telle sorte que le réseau forme un CDFG.

Le concepteur a la possibilité de poser des contraintes temporelles entre deux opérations. Les contraintes temporelles sont des latences à respecter, elles peuvent être des bornes minimales, maximales ou exactes.
Il peut y avoir plusieurs contraintes temporelles sur un chemin d'exécution (i.e les contraintes peuvent s'entrelacer). *camad* doit donc vérifier leur cohérence [HP95].
Il est possible de donner une latence sur un chemin d'exécution d'un branchement conditionnel sans que l'autre chemin soit contraint. Les contraintes peuvent être à l'intérieur d'un corps de boucle mais ne peuvent pas recouvrir une boucle.

L'ordonnancement se fait sous contraintes temporelles locales.
Chaque DFG issu d'un chemin d'exécution possédant une contrainte temporelle est au départ ordonnancé par un *ASAP*. Cet ordonnancement constitue la parallélisation maximale du circuit.
Le temps de propagation de chaque opération fonctionnelle est calculé pour évaluer le temps total du chemin d'exécution. Le temps total est comparé à la contrainte temporelle :

- Si elle n'est pas respectée, il n'y a pas de solution.
- Si elle est respectée, on tente de répartir les opérations sur des cycles supplémentaires pour mieux partager les opérateurs et ainsi diminuer la surface.

Ces outils acceptent des descriptions mélangeant du flot de données à du flot de contrôle.

behavioral compiler et CoCentric

behavioral compiler et *CoCentric* sont des outils appartenant à *Synopsys*.
CoCentric est l'évolution de *behavioral compiler* afin que l'outil de synthèse de haut niveau accepte un sous-ensemble de *SystemC* [Sys] [Kar99] comme langage de description comportementale. La description peut être de haut niveau ou de niveau RTL [PHCT01].

Outre la description comportementale, la contrainte essentielle de l'outil est la fréquence de fonctionnement visée. L'algorithme d'optimisation est basé sur cette contrainte pour décider quel type d'opérateur est à utiliser (par exemple, un additionneur simple ou par propagation de retenue, etc...) et quelles optimisations logiques sont à privilégier (temporelle ou en surface).

Le résultat est un circuit synchrone au niveau RTL constitué d'un automate d'états et d'un chemin de données.
Les *fonctions de transition et de génération* de l'automate sont décrites par une suite d'expressions booléennes comportementales. Les registres d'états de l'automate sont des instanciations d'un modèle de registre qui n'est pas encore ciblé pour une bibliothèque de cellules.
Le chemin de données est une vue structurelle et comportementale. Des expressions booléennes comportementales effectuent toutes les opérations logiques du chemin de données. Les modèles instanciés sont les multiplexeurs, les opérateurs arithmétiques et les bancs de mémoire. Le chemin de données utilise toujours des multiplexeurs et jamais de bus. Ils sont placés devant chaque opérateur.
Contrairement aux modèles de multiplexeurs qui sont décrits dans le chemin de données, les autres modèles sont issus d'une bibliothèque générique fournie par *Synopsys* : la bibliothèque *Design Ware*.

La forme de ces opérateurs particuliers est déjà déterminée dans cette bibliothèque grâce à des contraintes sous forme de *pragma VHDL*.

Le circuit RTL est destiné à l'outil de synthèse logique de *Synopsys* : *design analyser*. Il est la continuité de la synthèse de haut niveau car il accepte les mêmes modèles d'opérateurs.
La synthèse logique construit la vue structurelle au niveau portes, c'est donc à cette étape de la synthèse que les contraintes temporelles sont les mieux connues. Elle pourrait optimiser certains choix pris par la synthèse RTL.
Cependant, il y a une véritable coupure du flot de conception entre la synthèse de haut niveau et la synthèse logique : la synthèse logique ne peut remettre en cause les états de l'automate (ni l'encodage, ni les valeurs de sortie pour chaque état) et difficilement modifier les opérateurs instanciés.

Plusieurs optimisations de haut niveau des temps de propagation sont possibles :

- l'insertion de pipeline dans les modèles de la bibliothèque *Design Ware* pour décomposer un chemin de propagation.
- l'étalement d'une opération sur plusieurs cycles (i.e opération multi-cycles) pour laisser les signaux se propager.
- le chaînage de plusieurs opérations dans un même cycle, si les temps de propagation de celles-ci le permettent.
  Cependant le concepteur est obligé d'encapsuler les opérations dans une fonction.

L'outil peut dérouler une boucle ou la replier. Toutefois, le repliement est limité en nombre d'opérations dans la boucle.
Tous les branchements conditionnels sont systématiquement câblés pour diminuer la taille de l'automate.

Dans [LKMM95], l'outil utilise des gabarits comportementaux qui permettent de fixer l'ordre relatif d'un ensemble d'opérations. Cette construction simple permet le respect des contraintes temporelles, la modélisation des opérations séquentielles, le chaînage de certaines opérations et l'ordonnancement hiérarchique.
L'étape d'allocation fait appel à des unités génériques de la bibliothèque *Design Ware* construites à l'aide de portes. Cela simplifie l'estimation de la surface et des temps de propagation.

Le concepteur peut donner des contraintes supplémentaires à l'outil de synthèse haut niveau :

- indiquer certaines frontières de cycle.
- indiquer certaines *affectations* d'opérateurs.

Le concepteur donne des contraintes sur l'ordonnancement en imposant des frontières ainsi qu'un nombre minimum, maximum ou exact de cycles pour un processus ou une boucle. L'ordonnanceur possède trois modes de fonctionnement [KMM95] :

1. synchronisé : chaque cycle est déterminé par des barrières de synchronisation sur l'horloge (l'instruction *wait on* en VHDL).
2. macro-cycle : le nombre de cycles est indéterminé entre deux barrières de synchronisation. Ces barrières de synchronisation sont nécessaires pour ordonner les communications externes et décrire le mécanisme de *synchronisation logique*.

3. flottant : Les barrières de synchronisation ne sont pas effectives. La description du protocole de communication n'est pas possible. Il n'y a donc pas de contrainte applicable aux communications du circuit.

Dans le cadre de cette thèse, l'ordonnanceur avec *macro-cycles* est celui qui correspond aux conditions posées dans le chapitre 2. L'ordonnancement est libre entre deux communications.

Après une première synthèse avec le minimum de contraintes, l'outil laisse au concepteur le soin de raffiner sa description en lui donnant la possibilité de choisir son affectation. Pour faciliter ses choix, l'outil établit une évaluation des caractéristiques du circuit comme, par exemple, le nombre de cycles dans chaque boucles ou l'affectation des opérateurs.
C'est donc une synthèse par itérations successives pour explorer le compromis entre la période et l'allocation des ressources. Cela permet une démarche de type descendante qui permet des raffinements successifs.
L'utilisation conjointe de *CoCentric* et de *SystemC* permet une conception de plus en plus raffinée pour obtenir un circuit RTL [DTEB$^+$02].
Les nombreux essais nécessaires pour cibler une solution rendent la conception difficile. D'autant plus que chaque phase de synthèse peut être assez longue.

Il n'y a pas de contraintes sur les communications externes. Le concepteur est responsable de la modélisation correcte des communications et de sa bonne interprétation par l'outil de synthèse.
Les communications nécessitent un protocole cycle précis. La description doit décrire cycle par cycle le protocole de synchronisation. Cette description cohabite difficilement avec la description de haut niveau du circuit à synthétiser.

music

Il existe aussi des outils d'ordonnancement sans contrainte. Ils ne prennent en compte ni les contraintes matérielles ni les contraintes temporelles.
Ils construisent un automate synchrone à macro-cycles censé être modifié par un outil de synthèse d'automate.

A partir d'une description comportementale en VHDL, [Sug00] construit un automate de macro-états contenant des expressions comportementales. Cet outil fait partie d'un ensemble de modules de synthèse appelé *music* [CSSJ99].
Chaque état contient les opérations fonctionnelles sur les données. L'automate est à lui seul un circuit complet décrit au niveau RTL.

L'outil détermine les frontières des macro-cycles à partir des différents critères suivant :

1. barrières de synchronisation sur l'horloge (les instructions *wait* du VHDL) qui peuvent être éventuellement présentes dans le flot des description.
2. boucles : chaque corps de boucle est un flot de données séparé.
3. affectations de registres : un registre ne peut être affecté qu'une seule fois au cours d'un cycle. Deux écritures successives dans un registre ne pourront donc être dans le même macro-cycle.

L'algorithme est basé sur un ordonnancement de type ASAP qui parallélise les opérations au maximum en chaînant les opérations.
Tous les branchements sont câblés.

Il y a deux flots de conception possibles pour les automates générés :

- la synthèse RTL de l'automate.
  Le circuit généré sera utilisable si les deux conditions suivantes sont vérifiées :
  1. les opérations effectuées en parallèle sont suffisamment simples pour ne pas créer un circuit trop gros.
  2. les opérations chaînées sont suffisamment simples pour ne pas créer des temps de propagation trop longs.

  Dans ce cas, [MSS+99] a montré que l'on pouvait se dispenser de la phase d'allocation des opérateurs, la synthèse RTL produit un circuit voisin d'une conception "à la main" au niveau RTL.
- le raffinement de l'automate.
  Il faut modifier l'automate pour respecter les contraintes. Cela reporte donc le choix de l'*ordonnancement* et de l'*affectation* sur un outil de raffinement.

➠ Avec la plupart des outils de synthèse de haut niveau, le concepteur ne peut cibler précisément un circuit car les directives du concepteur sont très générales.

➠ Les outils se basent sur les caractéristiques des modèles contenues dans une bibliothèque d'opérateurs. Mais ces opérateurs n'ont pas les mêmes caractéristiques une fois instanciés dans le circuit. Les temps de propagation, notamment, varient en fonction des connexions aux ports des opérateurs et de l'optimisation logique réalisée par l'outil de synthèse.

➠ Des outils de synthèse de haut niveau ont permis de produire des circuits au niveau physique. Ces circuits étaient soit des flots de données soit des flots de contrôles.
Mais aucun circuit mixte complexe n'a été synthétisé à l'aide d'un seul outil. C'est à dire un circuit comportant des opérateurs arithmétiques dans son chemin de données et des branchements dépendant des données dans son automate.

➠ Les communications ne sont pas prises en compte par les outils de synthèse haut niveau. Au mieux, le concepteur doit transformer la description comportementale de son protocole de communication pour être interprété par l'outil, au pire, c'est l'environnement du circuit généré qui doit se synchroniser avec celui-ci.

3.4 Conclusion

Il y a eu peu de réalisations de circuit jusqu'au niveau des masques physiques avec la synthèse de haut niveau. En voici les premières raisons :

➠ Les temps réels de propagation ne sont pas pris en compte par les algorithmes de synthèse.

➠ La dissociation de la phase d'affectation de l'ordonnancement empêche toute optimisation globale. Seule des directives utilisateurs peuvent empêcher des choix d'ordonnancement qui se révéleront mauvais lors de l'affectation.

➠ Les outils de synthèse haut niveau existant ne peuvent cibler précisément une solution.

➠ Les outils ne permettent pas de définir clairement les communications externes du circuit.

Mais la principale est que les outils actuels ne supportent pas des descriptions mélangeant tous les aspects suivant :

- des séquences d'opérations arithmétiques dont la parallélisation systématique serait trop coûteuse.
- des boucles non bornées dont le déroulement serait impossible.
- des branchements conditionnels avec des chemins d'exécution très longs dont le câblage serait trop coûteux en surface.
- des profondeurs d'imbrications de boucles et de branchements importantes qui rendent difficile l'analyse globale.

Ceci interdit donc leur utilisation dans l'industrie sur des applications concrètes comme la vidéo.

Chapitre 4

Méthode logicielle pour tenir compte des contraintes électriques

Sommaire

La synthèse de haut niveau doit prendre en compte les caractéristiques électriques du circuit (cf. chapitre 2.4.4). Si elle n'en tient pas compte, la fréquence de fonctionnement du circuit ne sera pas celle visée par le concepteur. Ce qui sera rédhibitoire pour l'utilisation du circuit.
Les caractéristiques électriques sont issues des propriétés physiques du circuit une fois placé et routé. Il y a donc une dépendance mutuelle entre l'étape de synthèse et l'étape où l'on connaît les contraintes électriques !

La cible de la synthèse de haut niveau est un coprocesseur avec des communications asynchrones (cf. chapitre 2.2.2). Elle produit une vue structurelle avec deux éléments distincts : un chemin de données et un automate à états finis. Le chemin de données constitue la partie opérative et l'automate modélise la séquentialité des opérations.

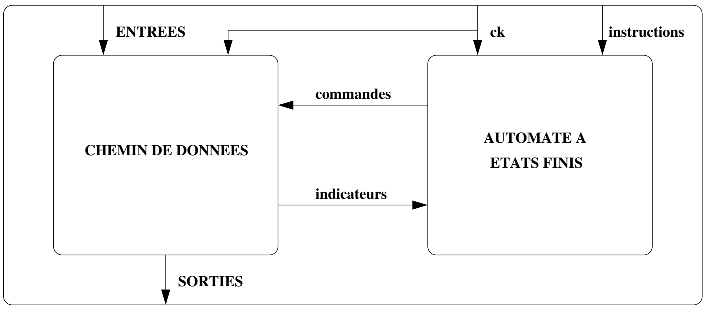


FIG. 4.1 Cible de la synthèse de haut niveau.

La figure 4.1 représente un automate dirigeant un chemin de données via des signaux de *commandes*. Le chemin de données envoie en retour des *indicateurs* à l'automate.
Les signaux *instructions* représentent, de manière générale, les signaux de communication directes entre deux circuits tels que les signaux d'interruptions, d'attentes, d'initialisations, etc...
Tous les autres ports d'*entrées* et de *sorties* sont des données classiques à traiter par le chemin de données. Les signaux *indicateurs* servent à l'évaluation de la *fonction de transition* de l'automate. Les signaux de commandes sont directement issus de la *fonction de génération* de l'automate.

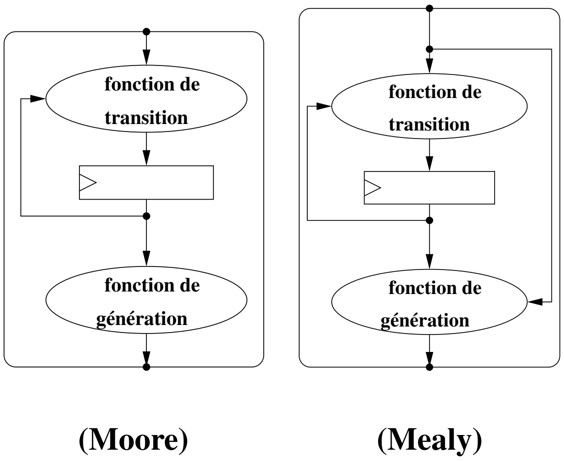


FIG. 4.2 Les deux types d'automates à états finis.

Il existe deux types d'automates à états finis (figure 4.2) :

soient
$\mathcal{E}$: l'ensemble fini des états de l'automate,
$\mathcal{I}$: l'ensemble fini des entrées de l'automate,
$\mathcal{S}$: l'ensemble fini des sorties de l'automate.
On définit la *fonction de transition* de l'automate telle que $\mathcal{E} \otimes \mathcal{I} \longmapsto \mathcal{E}$.
On définit la *fonction de génération* de l'automate de Moore telle que $\mathcal{E} \longmapsto \mathcal{S}$.
On définit la *fonction de génération* de l'automate de Mealy telle que $\mathcal{E} \otimes \mathcal{I} \longmapsto \mathcal{S}$.

Dans le cadre de la synthèse ciblée, nous nous intéresserons uniquement à l'automate de Moore. Même si l'automate de Mealy a l'avantage de pouvoir répondre dans le même cycle aux stimuli, il présente l'inconvénient de posséder un chemin purement combinatoire entre ses connecteurs externes. Cela risque de donner des chemins combinatoires très longs.
Toutes les fonctions purement combinatoires dépendant directement des entrées de l'automate sont transposées dans le chemin de données, de sorte à toujours avoir un automate de Moore. L'automate de Moore est destiné à fournir les commandes du chemin de données à chaque cycle.

Dans un premier temps, il faut identifier les contraintes électriques du circuit et leurs problèmes. Nous verrons les stratégies déjà utilisées pour tenir compte de ces contraintes. Nous examinerons quels sont leurs avantages et inconvénients.
Nous verrons dans ce chapitre que la séparation de l'automate du chemin de données, nous apporte des possibilités dans la résolution des contraintes électriques. Nous exposerons notre méthode logicielle pour respecter les contraintes électriques et les garanties qu'elle apporte.

4.1 Les contraintes électriques

Les caractéristiques électriques sont les impédances et les capacités des fils de connexion dans le circuit physique. Ces caractéristiques constituent le réseau *RC* du circuit. *R* est l'impédance et *C* est la capacité des fils. Elles dépendent de la technologie utilisée par le fondeur pour concevoir le circuit (largeur et hauteur des fils de polysilicium par exemple), mais aussi de la façon dont sont routés les fils. Le routage des fils dépend du placement des cellules et qui dépend du résultat de la synthèse.
La synthèse intervient en amont du placement/routage en donnant le nombre et le type des cellules ainsi que leurs interconnexions. La figure 4.3 montre l'influence de toute la chaîne de CAO sur les caractéristiques électriques du circuit.

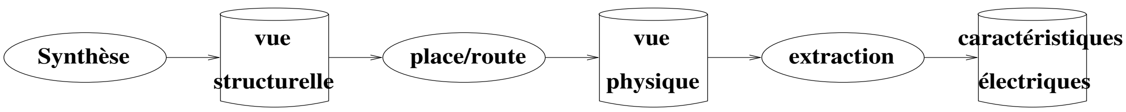


FIG. 4.3 Influence des outils de CAO sur les caractéristiques du circuit.

L'information la plus importante que nous pouvons déduire du réseau *RC* est le temps de propagation de chaque fil.
La formule $V_{out} = (1 - \exp^{\frac{-t_i}{R_i \times C_i}}) V_{in}$ [RCN03] met en rapport la tension avec le temps de propagation. Le temps de propagation d'un signal dans un fil peut être calculé grâce à l'approximation au premier degré $t_i = \alpha(R_i \times C_i)$ (α est une constante liée à la tension de commutation). Le temps de propagation

entre plusieurs fils interconnectés peut être évalué comme la somme des temps de propagation pour chaque fil ($T = \alpha(\Sigma R_i \times C_i)$).

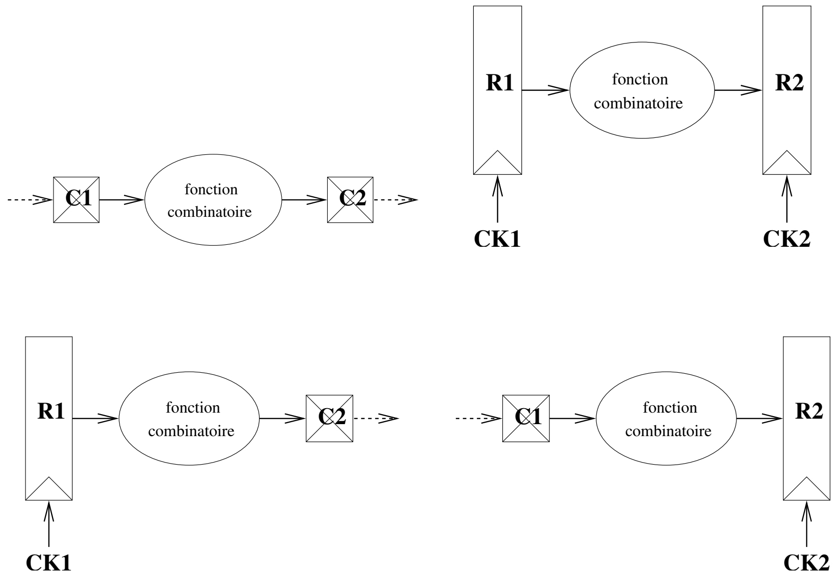


FIG. 4.4 Les différents chemins électriques.

Les temps de propagation sont considérés de registre à registre ou d'une entrée du circuit à un registre ou d'un registre à une sortie du circuit ou bien encore d'une entrée à une sortie du circuit.
La figure 4.4 représente ces différents chemins électriques. *R1* et *R2* sont des registres, *C1* et *C2* sont respectivement des connecteurs d'entrée et de sortie du circuit, *CK1* et *CK2* des horloges. Ces différents chemins peuvent être réduits à un seul. Nous pouvons toujours nous ramener à un chemin entre deux registres car il y a toujours un registre en aval du connecteur *C1* et en amont du connecteur *C2*.

L'étude des caractéristiques physiques de tous les chemins nous donnent les temps de propagation dans le circuit. Les temps de propagation sont des informations importantes car ils peuvent entraîner des problèmes de chaîne longue et de chaîne courte sur le circuit.

4.1.1 Les chaînes longues : violation du temps de préétablissement

Une chaîne longue du circuit est un chemin électrique fonctionnel où le temps nécessaire pour la transmission du signal est le plus long de tout le circuit. Ce temps détermine la période minimale de fonctionnement du coprocesseur. La fréquence maximale possible est l'inverse de cette période.

Dans la figure 4.5, plusieurs temps sont représentés :

- $t_{propagation}$ est le temps que met le signal pour parcourir la fonction combinatoire. Il se décline en un temps de propagation maximal ($t^{max}_{propagation}$) et un temps de propagation minimal ($t^{min}_{propagation}$). Ils représentent respectivement le temps de parcourt du signal le plus long et le plus court dans la fonction combinatoire. Entre ces deux temps, la valeur à la sortie de la fonction combinatoire est indéfinie.

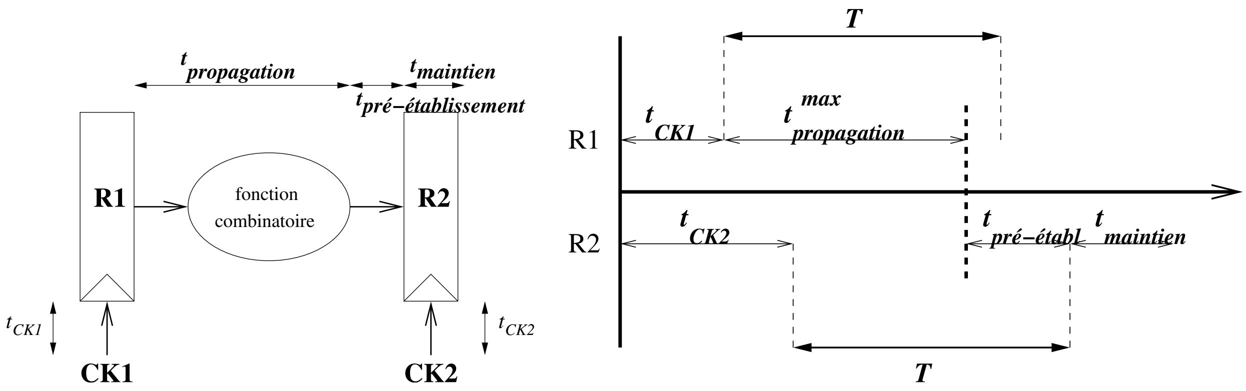


FIG. 4.5 Le temps de cycle ***T***.

- t_{CK1} et t_{CK2} sont les temps d'arrivée au registre des fronts des horloges *CK1* et *CK2*. Ces temps sont influencés par les fonctions logiques présentes sur les horloges qui augmentent alors la capacité du fil.
- $t_{preetablissement}$ est le temps nécessaire de stabilité du signal **avant** le front d'horloge de *CK2* pour que le registre *R2* mémorise la valeur.
- $t_{maintien}$ est le temps nécessaire de stabilité du signal **après** le front d'horloge de *CK2* pour que le registre *R2* mémorise la valeur.

Si nous prenons les temps exprimés dans la figure 4.5, la période maximale de fonctionnement est calculée sur la chaîne longue avec la formule $T = t_{preetablissement} + t^{max}_{propagation} + t_{CK1} - t_{CK2}$.

La synthèse doit connaître les temps de propagation pour optimiser la chaîne longue afin de respecter la fréquence donnée en entrée de la synthèse (cf. figure 4.3).

4.1.2 Les chaînes courtes : violation du temps de maintien

Autant un problème de chaîne longue affecte les performances du circuit, autant la présence d'une chaîne courte rend le circuit inutilisable si elle n'a pas été voulue. Son effet est de rendre le registre destination (*R2*) "transparent" au front de cycle, son comportement équivaut alors à un latch (bascule sur état). Le registre n'a pas le temps de mémoriser la valeur courante à son entrée durant le temps de maintien, elle est aussitôt remplacée par une autre qui sera mémorisée à sa place.

Un chaîne courte est un temps de propagation qui ne respecte pas le temps de *maintien* du registre destination. En référence au chronogramme de la figure 4.6, la formule $t_{maintien} + t_{CK2} - t_{CK1} < t^{min}_{propagation}$ garantit le respect du temps de *maintien* du registre *R2*.

Les problèmes de chaîne courte apparaissent généralement lorsque l'on positionne de la logique sur l'horloge. Si $t_{CK2} = t_{CK1}$, les temps de propagation sont suffisamment grands par rapport au temps de *maintien* des registres pour éviter l'apparition des chaînes courtes. La phase de synthèse doit s'attacher tout particulièrement à la synchronisation des signaux d'horloge pour les éviter.

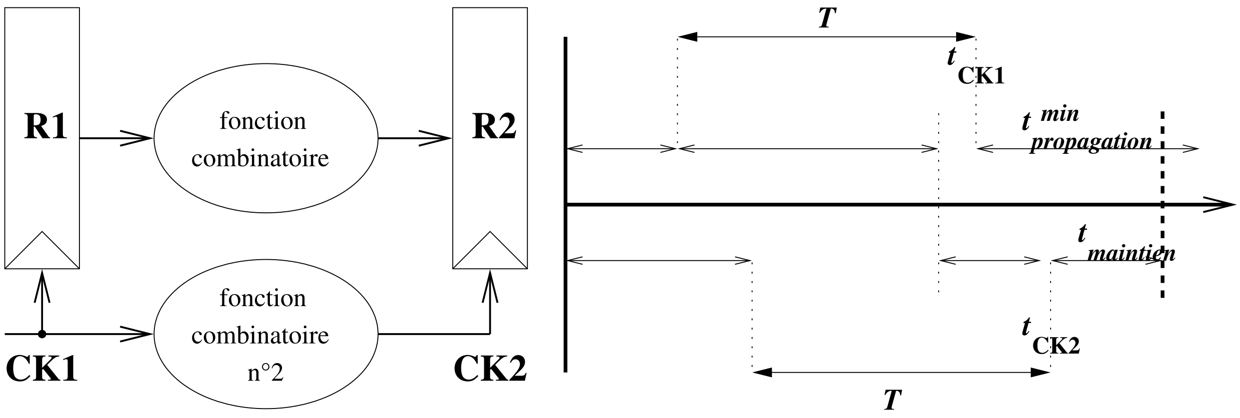


FIG. 4.6 La chaîne courte.

- ➠ Les contraintes électriques se matérialisent par des problèmes de chaînes longues et de chaînes courtes dans le circuit.
- ➠ La synthèse de haut niveau doit prévenir l'apparition de chaînes longues et courtes pour garantir le bon fonctionnement du circuit.

4.2 La prise en compte des contraintes électriques par la synthèse classique

Pour tenir compte des contraintes électriques, il existe des outils qui modifient la vue structurelle au niveau portes logiques afin de respecter les contraintes électriques. Ces outils utilisent des méthodes d'évaluation heuristiques des caractéristiques électriques. Ils doivent posséder un modèle temporel des cellules car ils interviennent avant le placement et le routage physique. Malheureusement, avec les technologies actuelles (de $0,18$ à $0,13 \mu m$), les résistances électriques dans les fils de routage augmentent et constituent le principal facteur temporel. Cela rend tout les modèles de prédiction temporels peu fiables.

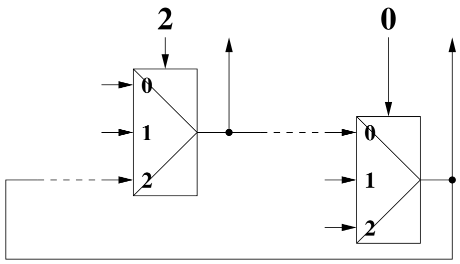


FIG. 4.7 Boucle combinatoire non fonctionnelle.

Les outils essaient d'optimiser tous les chemins combinatoires pour qu'ils respectent les contraintes de temps. Or certains chemins ne sont pas fonctionnels (ce sont des "faux chemins"), c'est à dire qu'ils ne seront jamais actifs dans leur intégralité lors du fonctionnement normal du circuit. La propagation du signal portant l'information ne sera utile que sur une partie de ce chemin.

Les commandes des multiplexeurs sont ignorées et tous leurs branchements sont envisagés. Les outils actuels ne sont pas capables de rapprocher les valeurs des commandes de l'automate d'états au chemin de données et donc de corréler les sélections respectives de deux multiplexeurs sur un même chemin combinatoire. Une boucle combinatoire non fonctionnelle peut même apparaître et peut empêcher l'outil d'optimiser les "vrais chemins" !
Les outils classiques ne sont pas capables de distinguer les "faux chemins", ils utilisent donc de la surface inutilement pour les optimiser.
Dans l'exemple de la figure 4.7, la *commande 2* sur le premier multiplexeur et la *commande 0* sur le deuxième multiplexeur ne sont jamais simultanées.

Les méthodes d'optimisation des chaînes longues et courtes se basent sur des estimations temporelles.

4.2.1 Les chaînes longues

Les outils classiques cherchent à insérer des registres (pipeline) ou à les déplacer (retiming [MSBSV91]) pour respecter les contraintes électriques estimées. Ils peuvent aussi dupliquer du matériel pour faire de la procrastination.

La diminution de la fréquence

Si le circuit possède des chemins dont les temps de propagation sont plus longs que le temps de cycle, il faut adapter la fréquence de fonctionnement. Diminuer la fréquence permet de respecter les contraintes électriques du circuit.
Dans la synthèse ciblée de haut niveau, la fréquence est une contrainte d'entrée. On ne peut donc la diminuer pour faire fonctionner le circuit.

L'insertion de registres

Le fait d'insérer un registre *R3* au milieu de la fonction combinatoire dans la figure 4.8 divise la fonction en sous-fonctions (n^o2 et n^o3) avec des temps de propagation plus petits (*T'* et *T"*) que la période initiale *T*. Cela diminue la période globale de fonctionnement qui devient le maximum entre *T'* et *T"*.

L'insertion d'un registre d'un bit n'est pas coûteux en surface. Cependant, le signal fait souvent partie d'un vecteur, on est alors obligé d'ajouter un registre sur chacun des bits du vecteur pour conserver la synchronisation de la donnée.
D'autre part, à période constante, cela augmente le temps d'exécution du circuit d'un cycle. L'augmentation du nombre de cycles peut constituer une perte de vitesse importante si la modification se situe dans une boucle. Cette perte peut tout aussi bien être négligeable si elle se situe dans un pipeline puisque l'on n'augmente que la latence des données en ajoutant des étages de pipeline.

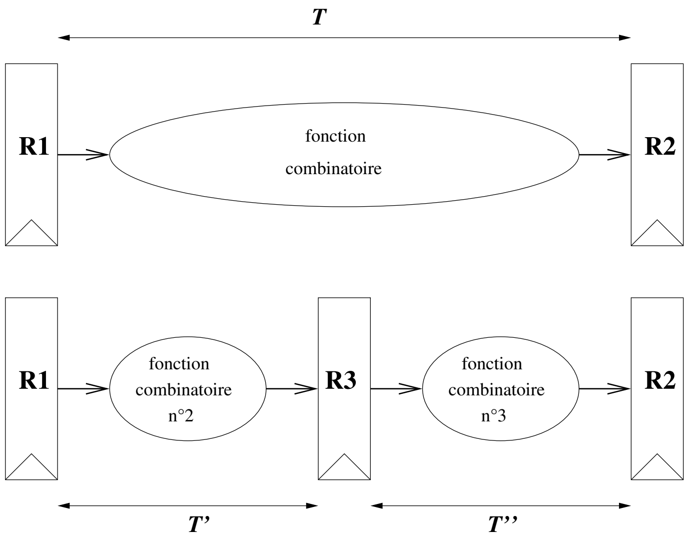


FIG. 4.8 Insertion d'un étage de pipeline.

Le déplacement de registres

S'il existe une grande différence dans les périodes estimées des fonctions logiques situées avant et après un registre (dans la figure 4.9, le registre *R3*), l'outil peut déplacer ce registre dans l'une des fonctions logiques : le registre *R3'* divise la fonction combinatoire n^o3 en deux autres fonctions n^o4 et n^o5 avec des temps de propagation moindre. La période minimale est passée de T à $\frac{3}{4}T$.
Contrairement à l'insertion de registre, le temps d'exécution n'est pas modifié car aucune coupure séquentielle n'a été rajoutée dans les chemins modifiés.

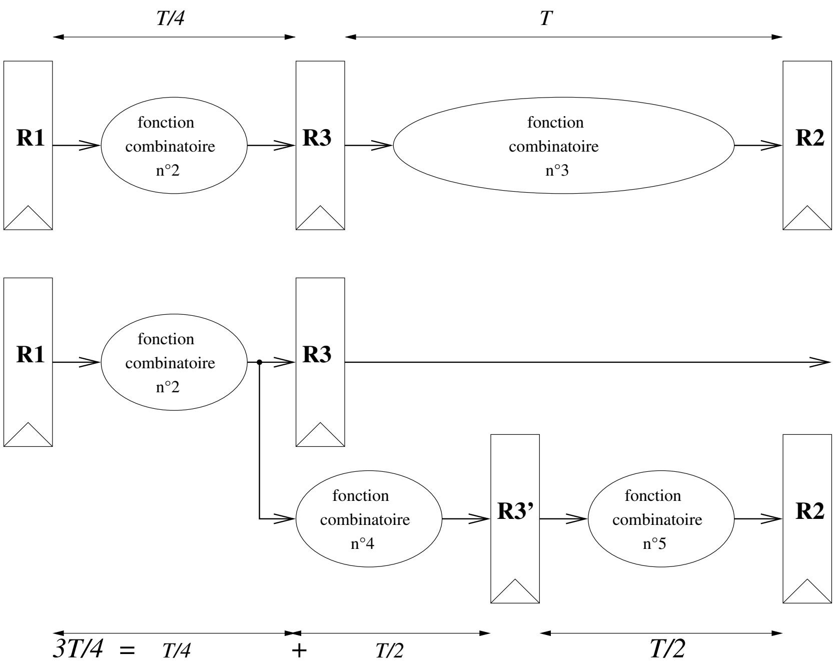


FIG. 4.9 Déplacement d'un registre.

Le déplacement de registre implique souvent une duplication de registre car comme le registre *R3* de la figure 4.9, il représente rarement une variable temporaire (cas des étages de pipeline).
Le principal inconvénient de cette méthode est que, en pratique, la fonction combinatoire est quasiment jamais sécable en un point sur un seul et unique bit. Le plus souvent, plusieurs bits doivent être mémorisés ce qui représente la création d'autant de nouveaux registres dans le circuit. Le fort accroissement en surface est donc le point faible du déplacement de registre.

La procrastination

Le principe de la procrastination est de "remettre au lendemain ce que l'on peut faire tout de suite". La figure 4.10 montre une duplication de la fonction n^o2 pour évaluer sans attendre le résultat de la fonction n^o1. L'évaluation de la fonction n^o2 devient complètement transparente.
La duplication de matériel est couramment utilisée pour créer des opérateurs arithmétiques performants (il n'y a plus de propagation de retenue dans un additionneur par exemple).

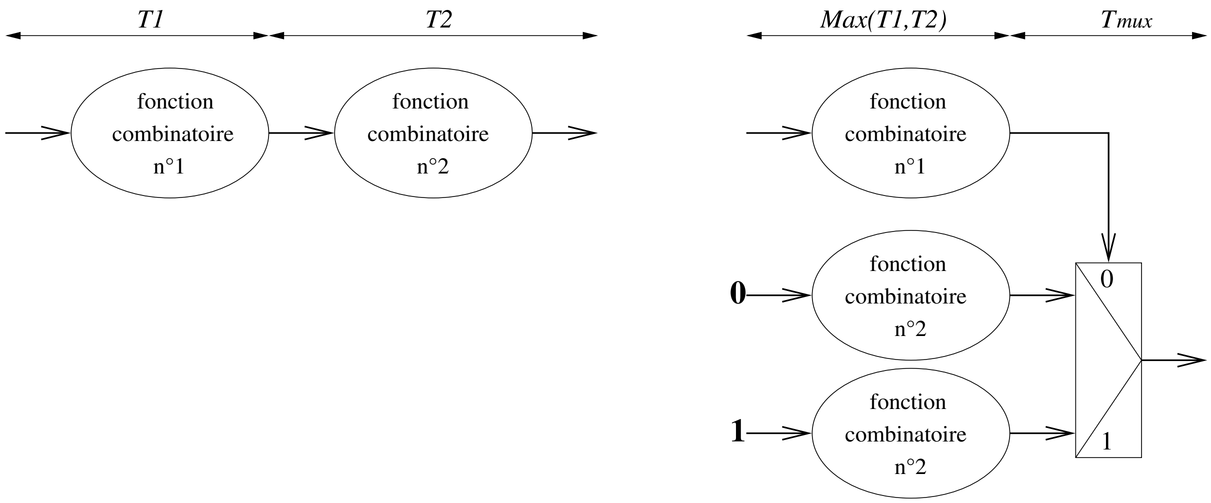


FIG. 4.10 Duplication de la fonction combinatoire n^o2 pour la procrastination.

La surface augmente géométriquement avec l'optimisation que l'on veut faire sur le signal.
L'ajout d'un multiplexeur dans le chemin pondère l'optimisation temporelle car le multiplexeur possède son propre temps de propagation.

Le gel de l'horloge de l'automate

Si pour un état de l'automate, une opération ne tient pas dans la période visée, [RD95] retarde le front d'horloge de l'automate. Ainsi le changement d'état est retardé pour laisser les signaux se propager dans le chemin de données.

Le circuit est synthétisé au niveau portes logiques. Les états de l'automate ne pouvant tenir dans le cycle sont repérés par simulation temporelle. On en déduit le nombre de cycles nécessaires à chaque état.
Un compteur et des comparateurs sont alors ajoutés au circuit. Chaque état émet un signal pouvant

mettre à zéro le compteur et enclencher le blocage de l'horloge de l'automate. L'horloge est libérée suivant l'état de ce compteur.

L'intérêt de cette méthode est d'ajouter des cycles uniquement là où c'est nécessaire. Il ne modifie pas le chemin de données. Les faux chemins sont ignorés car on ne considère que les états codants de l'automate.
Le coût supplémentaire de la méthode est uniquement une augmentation de la surface. Il faut instancier un incrémenteur et un registre accumulateur ainsi qu'une fonction logique de comparaison des valeurs. Cependant, il faut modifier l'automate pour que celui-ci émette les signaux de contrôle pour le compteur. Ils doivent porter certaines informations telles que le nombre de cycles d'attente.
La modification de l'automate peut se faire à deux niveaux :

- modification au niveau structurel : l'opération est complexe car il faut modifier les connexions et insérer des portes logiques.
- modification au niveau comportemental : il faut refaire la synthèse de l'automate.

Dans les deux cas, ces méthodes modifient la structure de l'automate et, le plus souvent, rallonge la *fonction de génération*.
Si les temps de propagation sont rallongés, rien ne garantit que le circuit muni de son compteur fonctionne à la fréquence visée.

4.2.2 Les chaînes courtes

Si le temps de propagation du signal entre deux registres est trop court par rapport au décalage d'horloge pour être mémorisé, il y a une chaîne courte entre ces deux registres.
En référence à la figure 4.6, la *fonction combinatoire* n^o2 possède un temps de propagation plus long que la *fonction combinatoire* de registre à registre. Il s'agit alors d'allonger le temps de la *fonction combinatoire* de registre à registre en ajoutant des buffers ou des portes logiques plus lentes. Si le temps de propagation de cette fonction devient plus long que celui de la *fonction combinatoire* n^o2, la condition $t_{maintien} + t_{CK2} - t_{CK1} < t_{propagation}^{min}$ est respectée et la chaîne courte disparaît.

L'allongement du temps de propagation pour éliminer une chaîne courte doit respecter les contraintes de chaîne longue et ne pas dépasser la période de fonctionnement du circuit. De plus, cette méthode tend à augmenter la surface du circuit.

4.2.3 Les limitations

Les méthodes de prise en compte des contraintes électriques par les outils de synthèse classiques ont toutes des limitations :

- La diminution de la fréquence permet de faire fonctionner un circuit en respectant les chaînes longues. Le concepteur n'a pas de contrôle sur la fréquence du circuit.
- L'insertion de registres permet d'éliminer une chaîne longue. Cependant elle ralentit la vitesse d'exécution et augmente considérablement la surface du circuit.
- Le déplacement de registres permet d'éliminer une chaîne longue mais elle augmente considérablement la surface du circuit.

- La procrastination permet d'éliminer une chaîne longue. Cependant la surface du circuit croît de façon géométrique avec l'optimisation.

- Le gel de l'horloge de l'automate permet d'insérer des états d'attente. Cela modifie les caractéristiques électriques de l'automate d'états.

- L'insertion de buffers permet d'éliminer une chaîne courte mais elle augmente la surface.

➠ La synthèse classique peut produire un circuit avec une faible surface mais il n'y a aucune garantie sur le respect des contraintes électriques.
➠ La synthèse classique peut prendre des gardes très grandes dans ses modèles temporels pour essayer de garantir le respect des contraintes électriques (sans garantie absolue toutefois). Malheureusement en prenant ces gardes, le circuit occupe une surface très importante.
➠ La synthèse classique ne garantit jamais le bon fonctionnement du circuit après obtention de la vue physique. Cela oblige le concepteur à faire plusieurs tentatives, ce qui est en opposition avec l'approche linéaire de la conception de circuit.

4.3 La prise en compte des caractéristiques électriques dans la synthèse ciblée

Afin de briser la dépendance mutuelle entre la synthèse et le placement/routage évoquée dans le chapitre 4.1, nous cherchons à décomposer la synthèse de haut niveau en plusieurs étapes.

4.3.1 Génération des contraintes électriques

Dans un premier temps, nous choisissons une démarche plutôt ”naïve” pour avoir un résultat sur lequel se baser. Il s'agit de produire de façon linéaire un chemin de données et un automate d'états *non contraint* en réduisant les contraintes à un classement très simple : le multiplieur est deux fois plus lent qu'une ALU ou un décaleur, les temps de propagation des opérateurs logiques sont considérés comme nuls.
Ce classement relatif est indépendant du circuit et des caractéristiques de chaque opérateur instancié dans le circuit. Cette simplification revient à supposer que les temps de propagation des opérateurs sont infiniment petits par rapport à la fréquence. La fréquence normalement fournie en entrée de la synthèse peut donc être ignorée pour cette étape.

Cette phase d'ordonnancement extrait la séquentialité des commandes de la description du coprocesseur donnée en entrée. Les algorithmes mis en oeuvre sont décrits dans le chapitre 6. Ils sont basés sur des heuristiques d'allocation et des modèles temporels extrêmement simples et rapides.

4.3.2 Extraction des contraintes électriques du chemin de données

Nous considérons maintenant le chemin de données obtenu en ignorant les contraintes électriques. Grâce à la synthèse au niveau portes et au placement/routage, nous obtenons une vue physique de notre chemin de données. Les caractéristiques électriques du chemin de données sont alors extraites de la vue physique.

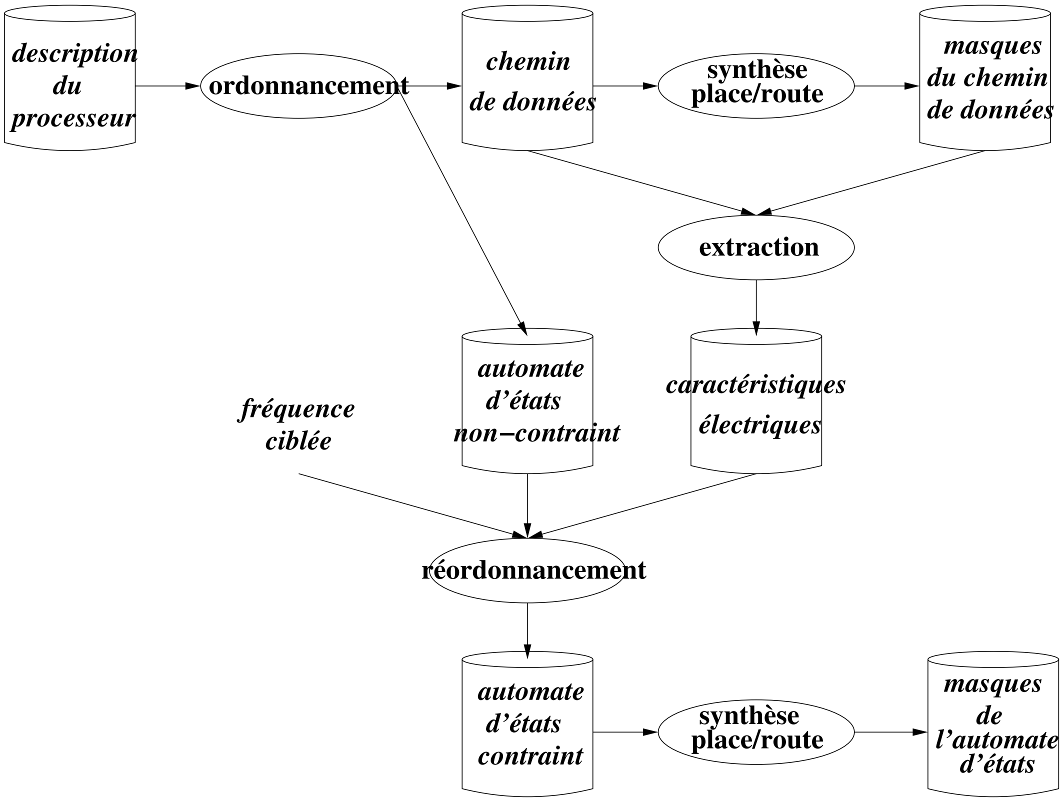


FIG. 4.11 Méthode logicielle.

La figure 4.11 montre l'enchaînement des phases qui permet d'extraire les contraintes électriques du chemin de données.

4.3.3 Raffiner l'automate en fonction des contraintes électriques

Les contraintes électriques du chemin de données servent de base à la transformation de l'automate d'états. L'automate *non contraint* doit être modifié car il a été produit avec une hypothèse irréaliste qui suppose que toutes les opérations tiennent en un cycle. L'automate à états *non contraint* doit subir une nouvelle phase d'ordonnancement (appelée *réordonnancement* dans la figure 4.11) tenant compte des contraintes électriques et de la fréquence passée en entrée de la synthèse. Seul l'automate est modifié lors de cette étape, le chemin de données reste, lui, inchangé. La démarche pour respecter ces contraintes est détaillée dans le chapitre 4.4.

Les chemins combinatoires considérés sont toujours des chemins fonctionnels. Pour chaque état de l'automate, nous identifions les chemins fonctionnels du chemin de données. L'identification de la

sélection d'un multiplexeur est aisée si l'on connaît l'état de l'automate qui active la commande. C'est uniquement sur ces chemins sensibles que nous tentons de respecter les contraintes électriques. Le contrôle du respect des contraintes électriques est fait pour tous les états de l'automate.
De cette façon, nous écartons les "faux chemins" et nous évitons d'utiliser de la surface pour les optimiser.

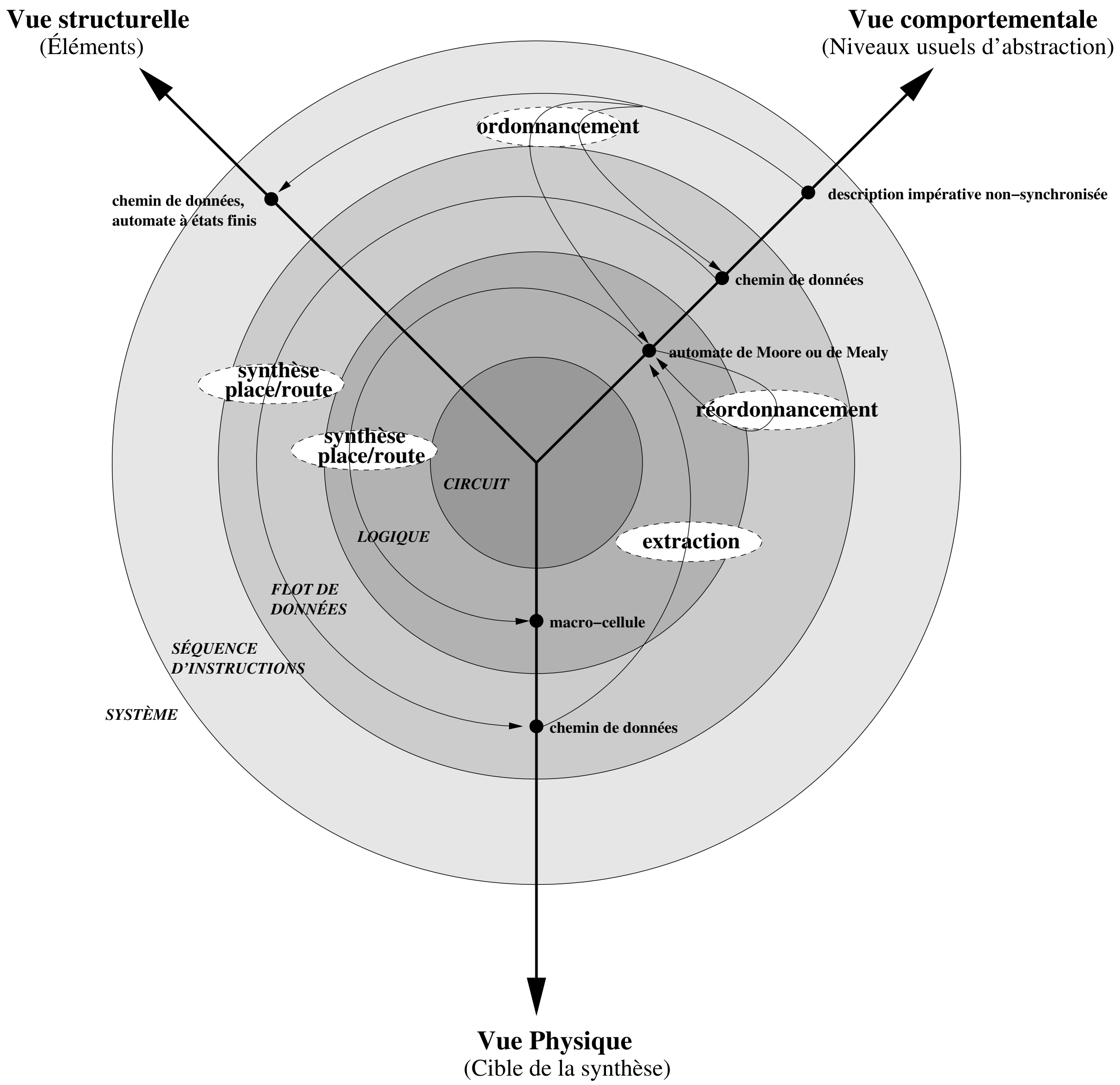


FIG. 4.12 Représentation des étapes de conception.

La figure 4.12 montre les différentes étapes du flot de conception dans le *schéma en Y de Gajski* [RKG$^+$92]. Les arcs de la figure représente uniquement les étapes de conception évoquées dans la figure 4.11.

✓ Dissociation de deux étapes dans l'ordonnancement pour la synthèse de haut niveau :

1. La recherche du parallélisme.
2. Le respect des contraintes électriques.

Le chemin de données est construit définitivement lors de la première étape et l'automate de commandes est affiné lors de la seconde étape de la synthèse.

✓ La taille du chemin de données est déterminée par l'ordonnancement initial sans prise en compte de la fréquence ciblée. Cette phase ne s'occupe que de la parallélisation des opérations et est déchargée des contraintes électriques.

4.4 Le respect des contraintes électriques dans la synthèse ciblée

Notre synthèse de haut niveau vise des coprocesseurs avec des communications asynchrones. Le temps d'exécution du coprocesseur synthétisé n'a donc aucun impact sur le bon fonctionnement des communications externes. Nous pouvons donc nous autoriser à changer la forme de l'automate d'états *non contraint* commandant la partie opérative du coprocesseur pour résoudre les problèmes électriques. Une fois les caractéristiques électriques connues, nous pouvons déterminer les chaînes longues et les chaînes courtes (cf. chapitre 4.1).

4.4.1 Respect des contraintes électriques dans le chemin de données

Nous disposons des contraintes électriques du chemin de données. Ces contraintes sont issues de l'extraction des capacités et impédances du chemin de données placé et routé. Il n'est plus question de modifier ce chemin de données pour respecter la fréquence. Il s'agit d'adapter l'automate d'états pour qu'il rende compatible les caractéristiques du chemin de données avec la fréquence ciblée.

Les chaînes longues

La figure 4.13 présente un exemple de chaîne longue ne respectant pas le temps de cycle : le temps de propagation de la *fonction combinatoire* est plus long que le temps de cycle. Les *commandes séquentielles* représentent les commandes d'écriture dans les registres et les *commandes combinatoires* représentent les commandes des multiplexeurs.
Pour résoudre la chaîne longue, il faut ajouter un cycle pour évaluer la *fonction combinatoire*. Cela revient à ajouter un état dans l'automate (l'*état** sur la figure). Cet état supplémentaire se consacre à la poursuite de l'évaluation de la fonction combinatoire en laissant les *commandes combinatoires* actives sur deux cycles. Cette fonction combinatoire devient une opération multi-cycles.
Les écritures dans les registres doivent être reportées au deuxième cycle (l'*état**) et ne pas être effectives sur les deux cycles. Ceci afin de ne pas écraser la valeur du premier registre lorsqu'il y a une boucle entre les deux registres.

Le fait de rajouter un cycle ralentit le temps d'exécution mais ne coûte quasiment rien en surface. En effet, les changements des *fonctions de transition et de génération* de l'automate ne provoquent

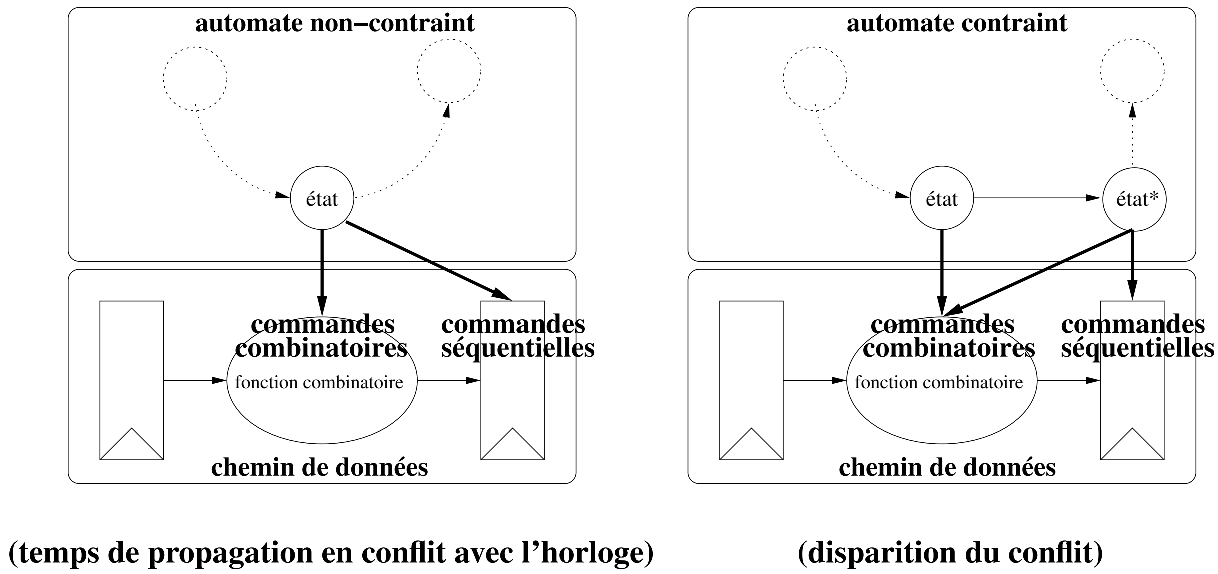


FIG. 4.13 Ajout d'un cycle pour l'évaluation de la chaîne longue.

quasiment pas de changement de taille. De plus, ajouter un état n'implique que très rarement l'agrandissement du registre d'états (suivant le type d'encodage utilisé et le nombre total d'états). Toutefois, si le registre d'états s'agrandit, il ne le sera que d'un bit. Ce résultat est à comparer aux techniques de pipeline et de déplacement de registres qui multiplient le nombre de registres par la taille des données manipulées.

Par la méthode de réordonnancement, chaque état peut donc être décomposé en autant d'états que nécessaire pour respecter le temps de cycle.

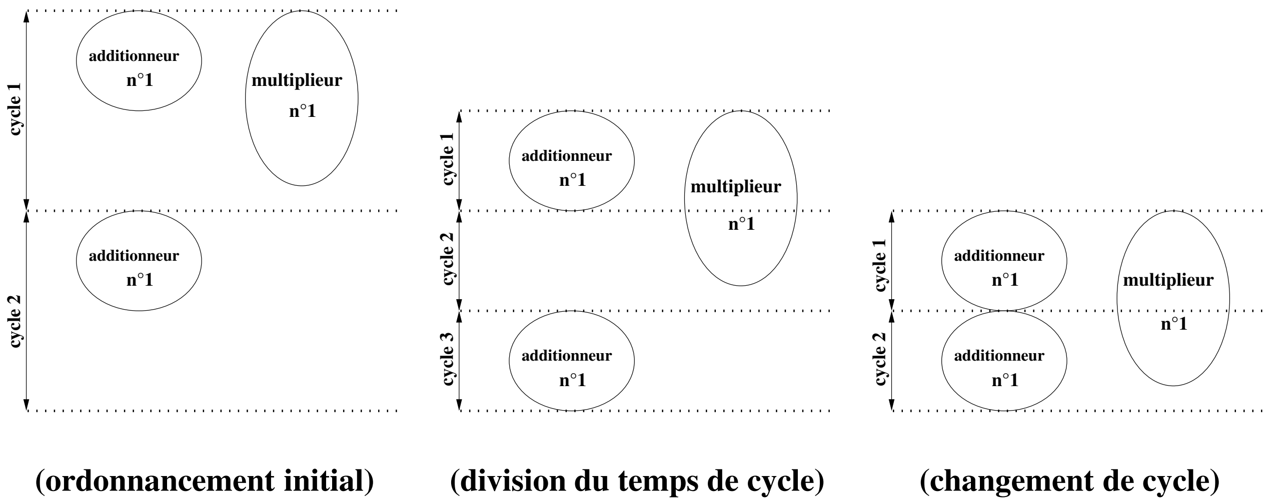


FIG. 4.14 Division du temps de cycle et changement de cycle pour l'additionneur n^o 1.

Nous pouvons diviser le temps de cycle comme nous le désirons et optimiser l'ordonnancement. Par exemple, la figure 4.14 nous montre les bénéfices possibles qu'entraîne une division du temps de cycle. Ce gain ne s'applique pas qu'à la chaîne longue mais à tous les chemins combinatoires.

La diminution du temps cycle augmente la finesse de l'ordonnancement. L'*opérateur* n^o1 peut être

réutilisé plus rapidement. D'une manière générale, la diminution du temps de cycle optimise le partage des chemins combinatoires puisqu'ils sont libérés plus rapidement pour une autre tâche.
Le coût en temps d'exécution est pondéré par l'optimisation de l'ordonnancement qui est faite après l'ajout d'un état.

Le nombre de cycles obtenus est influencé par l'ordonnancement initial. Si dans l'exemple de la figure 4.14, le *multiplieur* n^o1 était réutilisé dans le *cycle* n^o2 de l'ordonnancement initial à l'instar de l'*additionneur* n^o1, la division du temps de cycle n'aurait amené aucune économie en temps d'exécution.

En conclusion, le chemin de données peut supporter n'importe quelle fréquence de fonctionnement. La technique de réordonnancement a un faible coût en surface, l'ajout d'états dans l'automate est très peu pénalisant. Cependant, cette technique est fortement dépendante de la fiabilité des caractéristiques électriques extraites du chemin de données. L'efficacité du réordonnancement est aussi dépendante de l'allocation et l'affectation des ressources qui sont faites par la première phase d'ordonnancement.

Les chaînes courtes

Les chaînes courtes ne sont pas résolvables comme les chaînes longues. La résolution des chaînes longues consiste à augmenter le nombre de cycles pour satisfaire les contraintes électriques. Cependant cette méthode ne modifie pas les temps de propagation des fonctions combinatoires. C'est pourtant le différentiel entre les temps de propagation des données et de l'horloge qui influent sur les chaînes courtes.

En premier lieu, notre outil de synthèse de haut niveau détecte les chaînes courtes. La stratégie est de ne pas faire des écritures simultanées dans deux registres reliés par une chaîne courte. La chaîne courte entraîne la "transparence" du registre destination. La valeur présente à l'entrée de ce registre est écrasée par une autre valeur avant d'avoir pu être mémorisée. Si l'on sépare d'un cycle les deux écritures simultanées, il ne peut y avoir écrasement de cette valeur à l'entrée du registre destination.

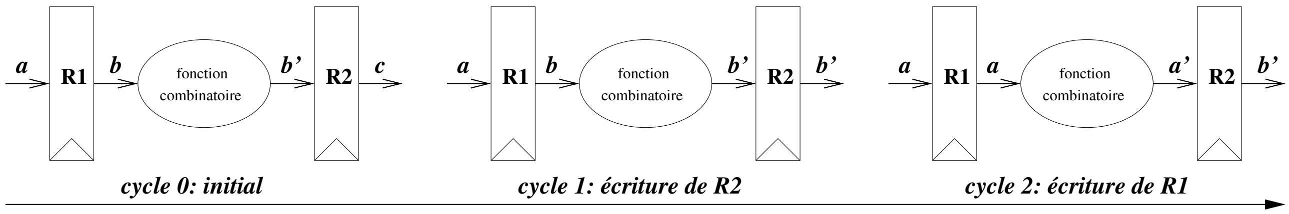


FIG. 4.15 Résolution de la chaîne courte en trois étapes.

La figure 4.15, nous montre la possibilité de faire fonctionner malgré tout une chaîne courte (la *fonction combinatoire*) comme un chemin normal. La valeur *b'* risque d'être écrasée par la valeur *a'* si on écrit simultanément dans les registres *R1* et *R2*.
La méthode de résolution utilise les permissions d'écritures sur les registres. On autorise dans le *cycle 1* l'écriture sur *R2* de manière à sauvegarder la valeur *b'*. Au *cycle 2* seulement, on procède à l'écriture sur *R1*.

Le coût de cette résolution est de un cycle. Le fait d'empêcher l'écriture simultanée interdit la résolution de chaîne courte dans les pipelines.

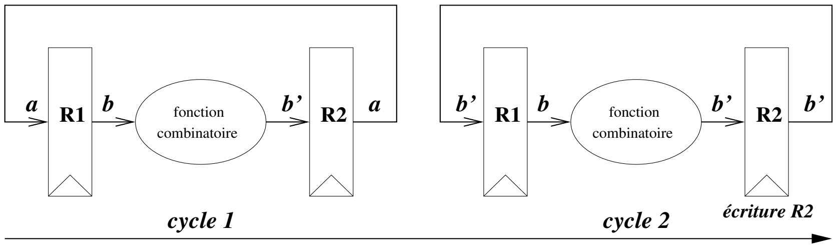


FIG. 4.16 Disparition de a dans la chaîne courte avec cycle.

Les chaînes courtes cycliques (figure 4.16) ne sont pas résolues par cette méthode puisque que l'on joue sur la simultanéité des écritures pour effectuer une permutation. Dans cet exemple, on perd alors la valeur *a* avant d'avoir pu la mémoriser.
L'outil de synthèse de haut niveau signale à l'utilisateur cette chaîne courte cyclique mais ne peut la résoudre. On pourrait imaginer une simple insertion de buffer dans la chaîne courte pour augmenter son temps de propagation et la faire disparaître.

4.4.2 Respect des contraintes électriques dans l'automate d'états

Il faut aussi prendre en compte les problèmes électriques provenant de l'automate d'états. Le problème majeur est que nous ne connaissons pas ses caractéristiques électriques. Les chaînes longues et courtes sont donc inconnues. Il y a chemin combinatoire entre le registre d'états et lui-même passant par la *fonction de transition*. C'est le seul chemin combinatoire entièrement interne de l'automate où il peut y avoir des contraintes électriques.

Les chaînes longues

Nous ignorons le temps de propagation de la *fonction de transition* lors de l'étape de *réordonnancement*. Si ce temps est plus grand que la période d'horloge imposée par le concepteur, il est impossible de produire un automate fonctionnant à la fréquence ciblée. Notre méthode repose ici complètement sur la puissance de la synthèse d'automate.

Notre outil, tout comme ceux de la synthèse classique, ne peut atteindre des objectifs physiquement irréalisables. Si la fréquence du coprocesseur est trop élevée par rapport à sa complexité, il ne peut y avoir de solution.

Les chaînes courtes

Notre outil de synthèse haut niveau vise des descriptions orientées flux de contrôle et mixtes. A priori, la *fonction de transition* de l'automate est suffisamment importante pour empêcher l'apparition de chaîne courte sur le registre d'états.

De plus, il n'y a pas de combinatoire sur l'horloge des registres d'états. Or d'après le chapitre 4.1.2, la condition $t_{maintien} + t_{CK2} - t_{CK1} < t^{min}_{propagation}$ nous garantit qu'il n'y a pas de chaîne courte. Nous

pouvons donc considérer que $t_{CK2} = t_{CK1} = 0$ pour le chemin du registre d'états vers lui-même. La condition devient donc $t_{maintien} < t^{min}_{propagation}$. Dès l'existence de deux états dans l'automate, le temps de propagation minimal ($t^{min}_{propagation}$) de la *fonction de transition* devient très grand par rapport au temps de *maintien* du registre d'états.
Il ne peut donc y avoir de chaîne courte sur le registre d'états.

4.4.3 Respect des contraintes électriques sur les connecteurs externes du circuit

Si nous nous référons au circuit schématisé par la figure 4.1, les entrées/sorties des données ne sont reliées qu'au chemin de données, seul les connecteurs d'*instructions* sont liés à l'automate d'états. Il n'y a pas de chemin combinatoire direct reliant une entrée du coprocesseur à une sortie. Les ports d'entrées/sorties du circuit sont donc tous reliés à des registres internes du chemin de données. Les signaux d'*instructions* sont quand à eux reliés au registre d'états de l'automate.

Sur tout ces chemins, il y a priori des contraintes électriques.
Cependant pour la synthèse de haut niveau, l'environnement externe du coprocesseur est caractérisé. Les contraintes électriques des signaux externes sont donc connues.

Les chaînes longues sur les ports de données

Les ports de données sont connectés à des registres du chemin de données. Les caractéristiques électriques des chemins combinatoires internes au circuit les reliant sont connues. Le signal externe est lui aussi caractérisé car l'environnement du coprocesseur est connu. Un signal d'entrée a pour source un registre externe au circuit. Un signal de sortie a pour destination un registre externe au circuit.
Nous nous ramenons donc à des chemin combinatoire entre deux registres avec des caractéristiques électriques connues. Le respect des contraintes de chaînes longues sur les ports de données se ramène à la méthode de résolution des contraintes électriques à l'intérieur du chemin de données du chapitre 4.4.1.

Les chaînes longues sur les ports d'instructions

Si les signaux d'*instructions* de l'automate ne sont pas suffisamment optimisés, leurs temps de propagation à l'extérieur du circuit s'ajoutent à ceux de la *fonction de transition*. Pour cette raison, les outils de synthèse d'automate doivent produire un signal d'*instruction* le plus proche possible des registres (cf. le signal d'*initialisation* de la figure 4.17). Le temps de propagation du signal d'*instruction* sera donc minimal à l'intérieur de l'automate, il pourra compenser un temps de propagation induit par l'environnement extérieur du circuit.

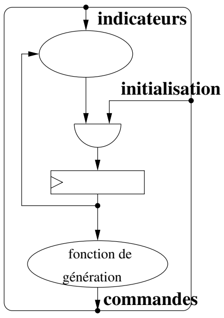


FIG. 4.17 Automate d'états avec initialisation synchrone.

Les chaînes courtes sur les ports de données

Nous pouvons nous affranchir des caractéristiques de l'environnement externe en étudiant uniquement le pire cas possible de chaîne courte sur un port d'entrée/sortie de données.

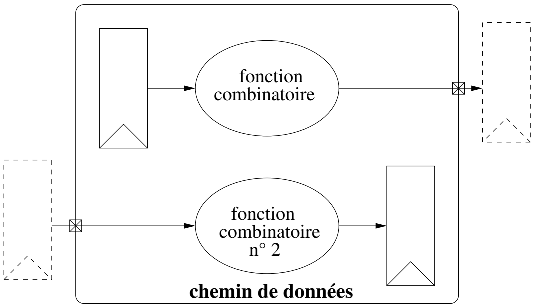


FIG. 4.18 Registres virtuels insérés juste après le connecteur.

On suppose qu'il existe un registre juste après le connecteur de sortie du chemin de données ou avant le connecteur d'entrée externe comme montré sur la figure 4.18. On suppose que ce registre possède les mêmes propriétés de *préétablissement* et de *maintien* que les registres du chemin de données et obéit à la même horloge.

Le fait qu'il y ait une connexion directe entre le registre externe et le connecteur représente le pire cas pour une chaîne courte avec l'extérieur. En effet, s'il existait une fonction logique entre le connecteur et le registre à l'extérieur du circuit, le temps de propagation en serait allongé. La condition $t_{maintien} + t_{CK2} - t_{CK1} < t_{propagation}^{min}$ (cf. chapitre 4.1.2) serait alors plus facile à respecter.

Pour l'étude du pire cas, il suffit de vérifier que la connexion sur l'un des ports du circuit d'un registre externe ne peut créer de chaîne courte.

L'étude des chemins entre chaque registre interne et chaque registre externe supposé être directement connecté à un port du chemin de données se ramène à une étude des chaînes courtes du chemin de données. La méthode employée est donc celle du chapitre 4.4.1.

Les chaînes courtes sur les ports d'instructions

Le signal d'horloge est critique dans le système, aucune logique n'est ajoutée sur ce signal pour éviter une désynchronisation des éléments du système. Le registre externe commandant l'*instruction* fonctionne avec la même horloge que le registre d'états.
D'après le chapitre 4.1.2, la condition $t_{maintien} + t_{CK2} - t_{CK1} < t^{min}_{propagation}$ devient $t_{maintien} < t^{min}_{propagation}$. Dans les technologies actuelles, le temps de propagation d'opérateur logique est très grand en comparaison du temps de *maintien*.
Il ne peut pas y avoir de chaîne courte entre le registre d'états et les connecteurs externes du circuit.

4.4.4 Respect des contraintes électriques sur la totalité du circuit

Du fait de l'approche basée sur l'évaluation des caractéristiques électriques du chemin de données, notre méthode ne prend pas en compte les temps de propagation entre le chemin de données et l'automate d'états (figure 4.19). En effet, à l'intérieur du circuit, il existe des chemins combinatoires reliant l'automate au chemin de données. Leurs temps de propagation respectifs peuvent à priori s'accumuler. Ces délais globaux sont à prendre en compte pour respecter la fréquence d'horloge.

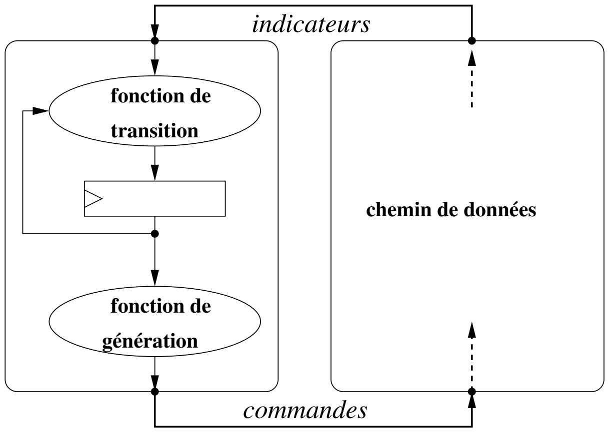


FIG. 4.19 Cumul des temps de propagation du chemin de données avec ceux de l'automate.

Contraintes électriques des interconnexions

Nous ignorons les caractéristiques électriques des interconnexions du chemin et de l'automate d'états. Comme les fils d'interconnexions relient deux blocs, ils sont probablement très longs et peuvent avoir une forte capacité. Pour obtenir une puissance suffisante, il faut insérer des buffers par défaut sur les fils d'interconnexions. Nous pouvons donc négliger les caractéristiques électriques des fils reliant le chemin de données à l'automate.

Les chaînes longues

D'après la figure 4.19, il existe un chemin combinatoire entre la *fonction de génération* et la *fonction de transition* passant par le chemin de données.
Dans le cas le plus problématique, les temps de propagations des *fonctions de génération et de transition* et du chemin de données sont cumulés (i.e dans le même cycle, l'automate change les valeurs des commandes par sa *fonction de génération*, des données sont manipulées dans le chemin de données et l'automate récupère les valeurs des *indicateurs* pour décider du prochain état grâce à sa *fonction de transition*).

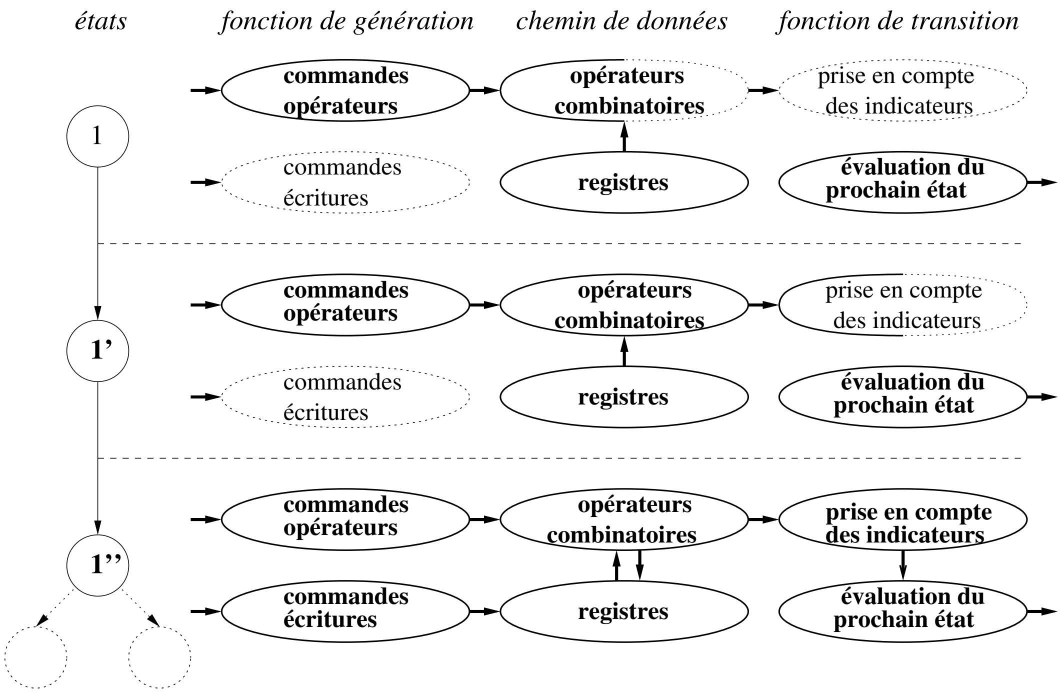


FIG. 4.20 Trois cycles pour les temps de propagation cumulés du chemin de données et de l'automate.

Dans l'exemple donné par la figure 4.20, deux cycles supplémentaires (l'*état 1'* et *état 1"*) ont été rajoutés à l'*état 1* initial de l'automate. La figure représente trois cycles matérialisés par trois états différents de l'automate. L'*état 1"* effectue un branchement.
La figure fait correspondre ces états aux parcours des signaux dans les chemins combinatoires entre la *fonction de génération*, le chemin de données et la *fonction de transition*. Les motifs en gras sont les éléments du chemin de données ou de l'automate actifs à l'état courant.

- La *fonction de génération* produit les commandes des opérateurs combinatoires et les commandes d'écritures dans les registres.
- Le chemin de données est constitué d'opérateurs combinatoires et de registres. Ils sont commandés respectivement par les commandes combinatoires et les commandes d'écritures de la *fonction de génération.*
- La fonction de transition est une fonction d'évaluation du prochain état en fonction de l'état courant. Elle peut éventuellement prendre en compte les signaux *indicateurs* issus du chemin

de données lorsqu'il y a un branchement entre deux états.

Dans cette figure 4.20, les commandes des opérateurs combinatoires restent identiques durant les trois cycles. Durant les deux premiers cycles, il n'y a pas de branchement (changement inconditionnel d'état) et donc aucune nécessité de prendre en compte les signaux *indicateurs* à ces cycles. En plus du troisième cycle, les signaux *indicateurs* ont deux cycles entiers pour se propager de la fonction de génération (i.e les commandes des opérateurs) vers le chemin de données (i.e les opérateurs combinatoires pour le test).

Nous constatons dans l'exemple de la figure 4.20 que la multiplication du nombre d'états a étalé le cumul des temps de propagation du chemin de données et de l'automate sur plusieurs cycles. Seules les commandes d'écritures et la mémorisation des données doivent se faire en un cycle (l'*état 1"*).

Les temps de propagation dans l'automate d'états ne sont pas connus. Le concepteur doit donc définir les temps de propagation de l'automate de son circuit. L'utilisateur a la possibilité de fournir ces temps à l'outil de synthèse de haut niveau.
Si les temps de propagation des *fonctions de transition et de génération* sont connus, ils sont traités comme des retards sur les chemins combinatoires provenant de l'automate et allant vers l'automate. Ainsi, le *réordonnancement* peut satisfaire les contraintes issues du cumul des temps de propagation entre l'automate et le chemin de données. La fréquence de fonctionnement est garantie, si et seulement si, le concepteur ne fournit pas des temps de propagation des *fonctions de transition et de génération* trop petits.

Les chaînes courtes

Lorsque nous avons étudié les chaînes courtes sur les connecteurs externes du circuit (chapitre 4.4.3), nous avons étudié le pire cas possible en insérant un registre virtuel derrière ces connecteurs. L'étude et la prévention du pire cas nous garantissait la non apparition de chaîne courte sur les connecteurs externes du circuit.

Il suffit de procéder de même avec les connecteurs du chemin de données liés à l'automate (i.e les signaux de *contrôle* et de *commande*) pour éviter l'apparition de chaîne courte entre ces deux blocs.

- ✓ Seules les caractéristiques électriques du chemin de données sont nécessaires pour respecter les contraintes électriques du circuit. Elles doivent être les plus précises possible.
- ✓ Les contraintes électriques du circuit sont respectées en ajoutant des états dans l'automate.
- ✓ La méthode logicielle de synthèse ciblée de haut niveau ne peut produire de circuit valide si :
  1. La *fonction de transition* est trop complexe pour la fréquence visée.
  2. Une chaîne courte est cyclique. Elle est détectée mais non résolue.

4.5 Conclusion

L'outil de synthèse de haut niveau doit tenir compte des contraintes électriques du circuit :

- La synthèse doit produire un circuit fonctionnant à la fréquence visée par le concepteur. La fréquence maximale de fonctionnement du circuit est connue une fois le circuit placé et routé.
- Les méthodes de résolution classiques telle que la procrastination ou l'insertion de registres n'offrent pas suffisamment de garanties pour le respect des contraintes électriques.

Notre approche consiste à ajouter des états supplémentaires à l'automate pour garantir la propagation des signaux dans le circuit.

✓ La phase initiale d'ordonnancement prend en charge la parallélisation des opérations sans s'occuper des contraintes électriques.

✓ Notre approche garantit que le circuit fonctionne à la fréquence visée par le concepteur. Seule une *fonction de transition* trop complexe empêcher de concevoir un circuit fonctionnant à une fréquence donnée. Les chaînes courtes cycliques ne sont pas résolues.

✓ La surface occupée est minimisée par cette approche mais le temps d'exécution peut éventuellement s'allonger.

Chapitre 5

Cibler une solution

Sommaire

Les outils de synthèse de haut niveau classiques génèrent plusieurs circuits physiques à partir d'une description comportementale. Ces circuits ne correspondent pas forcément à des solutions au problème du concepteur (le temps d'exécution, la surface ou la fréquence ciblées ne sont pas respectées).
Dans ce cas, il faut fournir à l'outil des directives supplémentaires afin de raffiner l'un des circuits générés dans la première passe.
En général, les outils nécessitent plusieurs passes successives pour obtenir une solution. Chaque passe requiert de la part du concepteur, une analyse ardue des résultats.

L'objectif de ce chapitre est de déterminer quelles directives sont vraiment nécessaires pour permettre à l'outil de synthèse de haut niveau de générer la solution visée en une seule passe.

5.1 L'espace des solutions

Nous avons vu dans le chapitre 2.3.2 que l'on peut décrire l'espace des solutions avec 4 critères :

- la fréquence d'horloge.
- le temps d'exécution global.
- la surface.
- la consommation.

Ces caractéristiques sont interdépendantes. Nous faisons l'hypothèse que la consommation peut se déterminer en fonction des autres critères. Par exemple, une fréquence plus grande augmente la consommation du circuit car l'oscillation de l'horloge est le principal consommateur de courant.

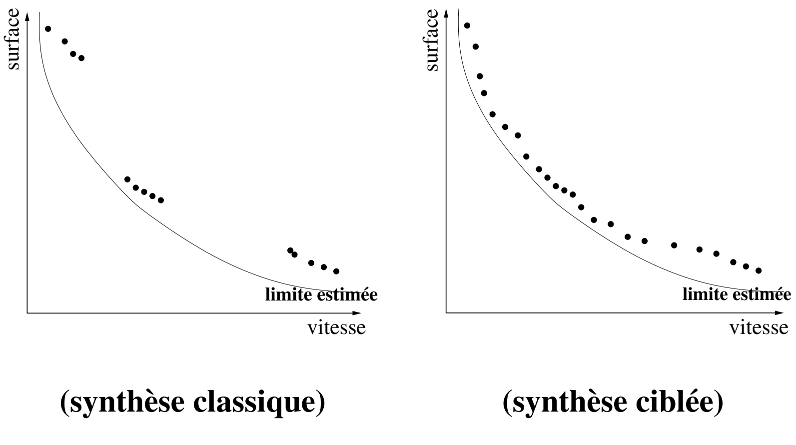


FIG. 5.1 Les *points de Pareto* atteignables par les outils de synthèse haut niveau.

Nous posons la fréquence comme une entrée de la synthèse. L'espace des solutions possibles n'est plus décrit que par un plan. Il s'agit alors de résoudre l'antagonisme entre ces deux critères :

- le temps d'exécution global.
- la surface.

La figure 5.1 décrit l'espace des solutions réduit aux caractéristiques de surface et de temps d'exécution du circuit. Un *point de Pareto* est un point pour lequel il n'existe aucune solution expérimentale meilleure sur l'un des axes de l'espace. Il y a une infinité de *point de Pareto* dans cet espace, chaque point bénéficiant de caractéristiques qui lui sont propres.
La synthèse classique ne peut pas couvrir l'intégralité de l'espace des solutions. Le concepteur doit, par des directives à la synthèse ciblée, pouvoir obtenir tous les *points de Pareto* de l'espace des solutions.

5.1.1 Les choix architecturaux

Certaines décisions architecturales ont un impact direct sur les performances du coprocesseur. Les choix principaux sont la surface, le temps d'exécution et les communications.

Les choix d'optimisations de la surface

Le nombre et le type des opérateurs matériels : Le choix des opérateurs du chemin de données est important pour cibler une surface. L'automate d'états influe très peu sur la surface totale du coprocesseur car la logique booléenne issue des *fonctions de transition et de génération* est très souvent négligeable par rapport au chemin de données. Le registre d'états a un faible coût en surface car un simple registre de 10 bits suffit à coder 2047 états différents. En comparaison, les données manipulées sont souvent de 32 bits, ce qui implique des registres de la même taille pour manipuler les valeurs dans le chemin de données.

Les opérateurs matériels du chemin de données sont de deux types :

- opérateurs séquentiels : registres, banc de registres...
- opérateurs combinatoires : additionneurs, multiplieurs, décaleurs...

Le nombre d'opérateurs matériels influe de façon importante sur la surface occupée par le chemin de données. L'*affectation des opérateurs combinatoires* est déterminée en partie par le type des opérations logicielles de la description comportementale.

Si nous considérons l'opération $x + y$ d'une description comportementale, l'opérateur matériel doit supporter l'addition. Cette opération peut être réalisée avec un additionneur simple ou avec une unité arithmétique et logique (ALU). L'additionneur occupe moins de surface mais l'ALU permet de faire d'autres opérations telles que la soustraction ou le "et logique".
Si la description comportementale contient aussi les opérations $x - y$, $y - x$, $x\ and\ y$, il est peut-être avantageux d'instancier une seule ALU pour minimiser la surface plutôt que des opérateurs simples.

Les variables de la description comportementales sont soit des signaux internes, soit des registres pour stocker des valeurs intermédiaires à la frontière de cycle.

La taille des opérateurs matériels : La taille des connecteurs externes est une caractéristique de l'interface du circuit ciblé, le concepteur doit impérativement la fournir à l'outil de synthèse.

La taille d'un opérateur matériel découle de la taille maximale de toutes les opérations fonctionnelles qui lui sont affectées. Les tailles des opérations fonctionnelles sont souvent définies par la description comportementale et la norme du langage de description utilisé.

On peut prendre les tailles utilisées lors de la compilation logicielle (généralement de 32 bits) pour déterminer les tailles des opérateurs. Malheureusement, ce qui n'avait aucun coût en logiciel se révèle être très coûteux en surface lorsqu'il s'agit de concevoir du matériel. La synthèse ne peut se contenter de la taille maximale proposée pour le logiciel.

On peut déterminer la taille de chaque opération fonctionnelle par une analyse du code de la description comportementale. Dans la plupart des cas, la taille peut être déduite de proche en proche uniquement à partir de la taille des connecteurs externes.
Par exemple, en *SystemC* ([Sys]) la taille t_{x+y} du résultat de l'addition $x + y$ est égale à $t_{x+y} = Max(t_x, t_y) + 1$. La taille du résultat est supérieure à la taille des opérandes pour ne pas perdre la retenue éventuelle issue de l'opération.
Si la taille du résultat est connue, elle peut borner la taille de l'opération. Par exemple, dans l'assignation $z = x + y$, la taille de l'opération d'addition est définie par $t_{x+y} = Min(t_z, Max(t_x, t_y) + 1)$.
Nous pouvons aussi déduire à rebours des propriétés sur la taille des opérandes : $t_x = t_y = Min(t_z, Max(t_x, t_y))$ (cf. [MRS$^+$01]).
La norme du langage de description doit absolument définir la taille de chaque opération fonctionnelle à partir des opérandes.

La taille des opérations flottantes obéit à la norme IEEE [Ins85]. La synthèse de haut niveau, à moins d'intégrer un opérateur matériel en arithmétique flottante, ne peut intégrer d'opérations sur les flottants. La granularité de description est inadaptée pour de tels opérateurs nécessitant des optimisations poussées.
Cependant, le concepteur peut se ramener dans certains cas à des opérations en virgule fixe. La translation n'est pas triviale car il doit évaluer la perte d'information par rapport aux données en entrée. Si l'opération s'effectue en virgule fixe, la détermination de sa taille se ramène à une détermination classique d'une opération entière. En reprenant l'exemple de $x + y$, la taille des opérandes en virgule flottante est calculée comme suit : $t_x = t_x^{entier} + Max(t_x^{decimal}, t_y^{decimal})$.
La partie décimale du résultat est $t_{x+y}^{decimal} = Max(t_x^{decimal}, t_y^{decimal})$.

Le calcul de la taille des opérations de proche en proche a ses limites car il y a des majorations pour éviter de perdre des données. Par exemple, les branchements conditionnels et les boucles rendent l'analyse difficile.
La taille évaluée a tendance à augmenter à chaque opération traversée. Or la plupart du temps seuls certains bits du résultat nous intéresse (généralement les bits de poids faible). La taille des opérations est souvent trop grande.

Le concepteur doit donc borner régulièrement les opérations de la description comportementale. Le moyen le plus simple est de donner la taille minimale des variables.

La connectique : La connectique ne doit pas être négligée, si nous voulons minimiser la surface. Elle est aussi bien présente au niveau logique (bus, multiplexeurs) qu'au niveau physique (fils de routage).
Si la description comportementale comporte deux opérations $x = a + b$ et $y = c - d$ affectées à un additionneur/soustracteur, cet opérateur matériel est connecté à deux multiplexeurs à deux entrées. Il peut être moins coûteux en surface d'instancier un additionneur et un soustracteur sans multiplexeur.

Les choix d'optimisations du temps d'exécution

Le temps d'exécution est la multiplication de la période d'horloge du circuit par le nombre de cycles d'exécution pour exécuter intégralement la tâche.
La fréquence est une contrainte d'entrée de la synthèse (cf. l'hypothèse du chapitre 2.2.2), elle ne peut donc être un critère sur lequel nous pouvons jouer. Seul le nombre de cycles dans les parties les plus exécutées influence le temps d'exécution de notre coprocesseur.
A moins de faire de la vérification formelle et/ou de la simulation (le comportement peut dépendre des données), il est impossible d'évaluer le nombre de cycles séparant deux communications externes. Le nombre de cycles ne peut être considéré que localement sans vision globale.

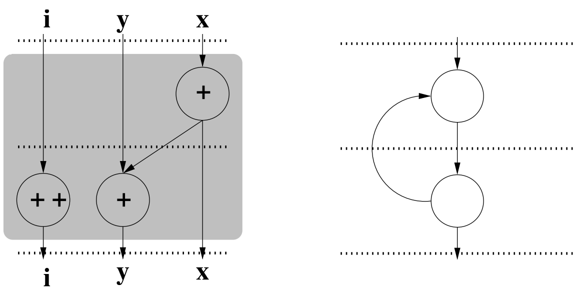
La parallélisation des opérations : La diminution du nombre de cycles passe par la parallélisation des opérations fonctionnelles. Cela implique une augmentation du nombre de ressources matérielles. En effet, des opérations simultanées demandent plus d'opérateurs combinatoires. Cette optimisation entraîne un accroissement de la surface (d'où l'antagonisme entre le temps d'exécution et la surface), nous devons donc évaluer son gain pour rendre la parallélisation effective ou non.

A priori, l'optimisation locale du nombre de cycles ne peut que diminuer le temps d'exécution global. Seul le niveau d'imbrication des boucles dans lequel l'optimisation est faite peut donner un facteur d'impact. Cependant nous nous concentrons sur des descriptions avec des branchements dépendants des données. L'impact des optimisations est le plus souvent imprédictible.

Outre la parallélisation des opérations fonctionnelles, il existe trois actions sur les opérations de branchement ("while", "for" et "if") pour modifier le nombre de cycles. Il s'agit du déroulement de boucle, du repliement de boucle et du câblage des branchements conditionnels. Ils permettent d'accroître le parallélisme des opérations fonctionnelles.

Le déroulement de boucle : La figure 5.2 représente l'ordonnancement d'un corps de boucle sans optimisation de sa latence. La boucle et le corps de boucle nécessitent un additionneur utilisé durant deux cycles et un incrémenteur. La réalisation totale de la boucle est $4 * 2 = 8$ cycles. La boucle constitue une frontière pour la parallélisation des opérations.

```
for ( i= 0; i < 4; i++ )
{
  x = x + 3;
  y = y + x;
}
```

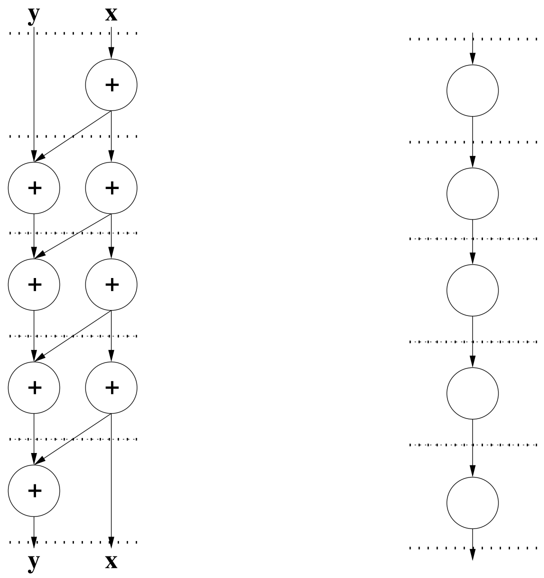


(description comportementale) **(S1 : corps de boucle)** **(automate de la boucle)**

FIG. 5.2 Ordonnancement d'un corps de boucle.

La figure 5.3 montre le déroulement de la boucle sur l'ordonnancement initial de la figure 5.2. L'incrémenteur disparaît. L'économie de l'incrémenteur est pondérée par l'augmentation du nombre d'états. Ce type de modification n'est avantageuse que si la boucle possède un petit nombre d'itérations.

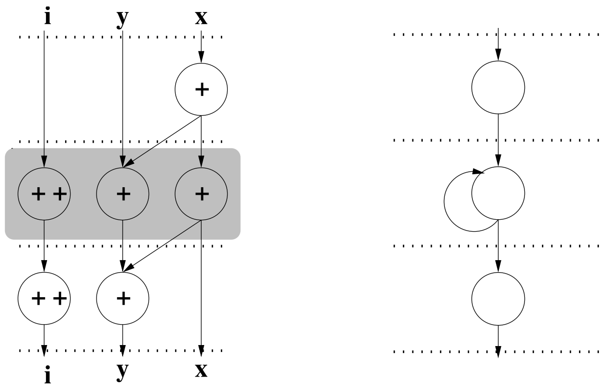
Le déroulement de boucle nous permet d'optimiser l'ordonnancement en parallélisant l'utilisation des ressources en vue d'une minimisation du nombre de cycles. Dans le schéma *S2*, le nombre de cycles n'est plus que de $4 * 1 + 1 = 5$ cycles. De plus, la suppression des frontières du corps de boucle favorise la parallélisation avec les opérations fonctionnelles précédant et suivant la boucle. L'optimisation en nombre de cycles se paie en surface puisqu'il faut deux additionneurs. Il ne faut que 5 états dans l'automate pour représenter la boucle.



(S2 : boucle déroulée) **(automate de la boucle déroulée)**

FIG. 5.3 Déroulement de boucle.

Le repliement de boucle : Le repliement de boucle ([CS97]) est nécessaire si la boucle possède trop d'itérations ou si l'on ne connaît pas le nombre d'itérations.

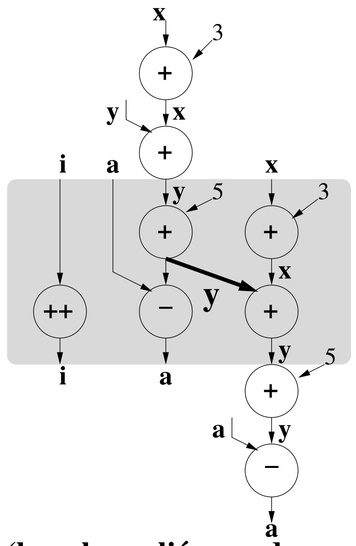


(S3 : boucle repliée) **(automate de la boucle repliée)**

FIG. 5.4 Repliement de boucle.

Si l'on peut ajouter un additionneur, l'ajout d'un simple incrémenteur permet de recréer un corps de boucle optimisé (la boucle repliée du schéma *S3* de la figure 5.4).
Cet ordonnancement de boucle nécessite 3 états de l'automate, deux additionneurs et un incrémenteur. Il faut $3 * 1 + 2 = 5$ cycles pour exécuter entièrement la boucle.
Le repliement peut se faire sur autant d'états que le permettent les dépendances de données. Le nombre de ressources augmente en conséquence.
Dans la figure 5.5, le repliement est bloqué par une dépendance de donnée sur la variable *y*.

```
for ( i= 0; i < 100; i++ )
{
  x = x + 3;
  y = y + x;
  y = y + 5;
  a = a - y;
}
```

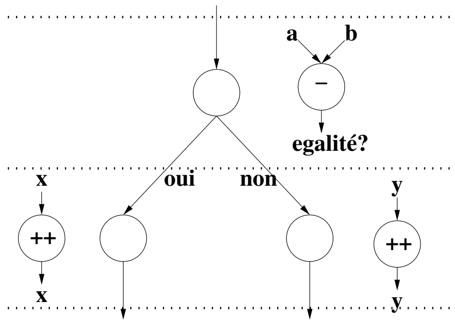


(description comportementale) **(boucle repliée sur deux cycles)**

FIG. 5.5 Repliement de boucle avec dépendance de données sur y.

Le câblage des branchements conditionnels : Sur l'exemple de la figure 5.6, l'ordonnancement donne trois états différenciés dans l'automate. Un seul incrémenteur est nécessaire. Il faut deux cycles pour effectuer ce branchement (un pour le test effectif et un autre pour exécuter l'incrémentation).

```
if ( a == b ) x++;
else          y++;
```



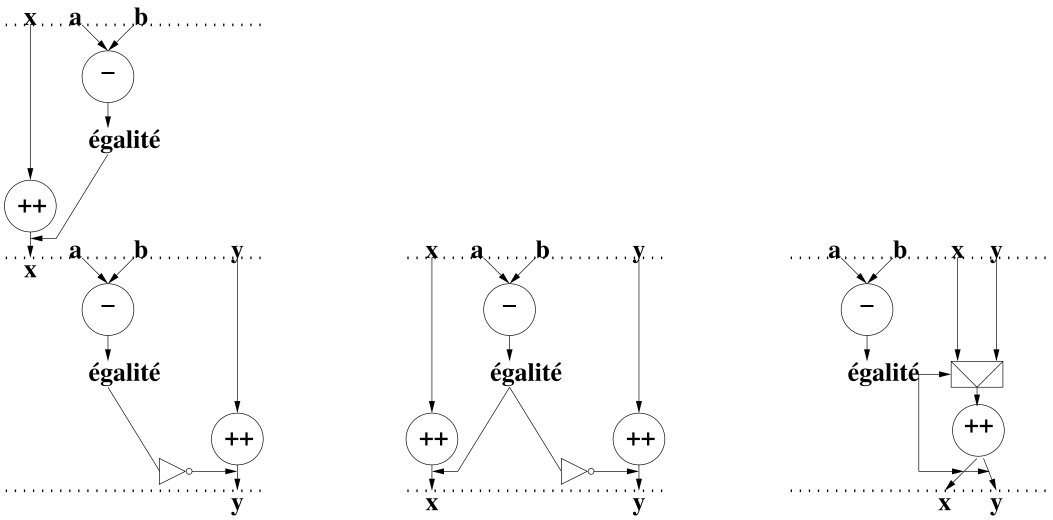
(description comportementale) **(CDFG sans câblage)**

FIG. 5.6 Branchement conditionnel non câblé.

Les branchements conditionnels constituent des frontières pour la parallélisation des opérations fonctionnelles.

La figure 5.7 représente plusieurs ordonnancements d’un branchement câblé. Le test de la condition de branchement s’effectue alors dans le même cycle que l’opération sur les données. L’exécution des opérations fonctionnelles devient indépendante de l’automate.

- Dans l’ordonnancement *O1*, un seul incrémenteur est instancié. Les opérations sont sur deux cycles.
  Il n’y pas de diminution du nombre de cycles d’exécution et pas de surcoût.
- Dans l’ordonnancement *O2*, deux incrémenteurs sont instanciés. Les opérations sont sur un cycle.
  Il y une diminution du nombre de cycles d’exécution et un surcoût en surface.
- Dans l’ordonnancement *O3*, un seul incrémenteur est instancié. Les opérations sont sur un cycle. Le temps de propagation est allongé par la présence d’un multiplexeur.
  Il y une diminution du nombre de cycles d’exécution et un surcoût en temps de propagation.



(O1 : sur des cycles différents) (O2 : sur le même cycle) (O3 : avec un multiplexeur)

FIG. 5.7 Ordonnancements d’un branchement câblé.

Le câblage des branchements conditionnels entraînent une augmentation du nombre d’opérateurs matériels et rallongent les temps de propagation. Les branchements ne peuvent être systématiquement câblés. Cependant, le câblage ne fait jamais perdre de cycle.

Les choix de protocoles de communication

Le coprocesseur généré va être inséré dans un système avec ses propres contraintes de communication. Les données ne pourront être échangées avec le coprocesseur que sur des cycles précis. Le fonctionnement du coprocesseur devra être synchrone avec le système.
Si le fonctionnement du coprocesseur n’est pas complètement synchrone avec le système, il faut un protocole de communication pour que le coprocesseur soit *logiquement synchrone* avec le système.

Il existe une multitude de protocoles de communication. Ces communications s'effectuent toujours entre un maître et un esclave, le maître décidant de leur établissement. Outre les signaux véhiculant les données, le maître et l'esclave possèdent des signaux de contrôle.
D'une manière générale, le maître sélectionne et prévient l'esclave grâce à un signal de sélection (appelé *STROBE* ou *SELECT*). L'esclave sélectionné répond qu'il est prêt à communiquer par un signal d'acquittement (*ACKNOWLEDGE*).

Si il y a plusieurs destinations dans l'esclave, le signal de sélection n'est pas suffisant pour distinguer le registre interne. Un signal d'adresse est associé au signal de sélection pour distinguer les registres internes (figure 5.8).

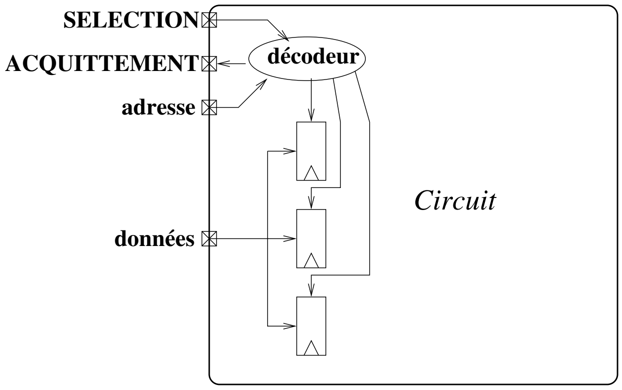


FIG. 5.8 Exemple d'interface esclave avec adresse de registre interne.

Les fils de communication peuvent être soit directs et exclusifs, soit partagés entre plusieurs composants tels les bus, soit mixtes telles les interruptions matérielles. Les connections par bus nécessitent des signaux pour la réservation de bus, de traitement du mode rafale, etc... Les interruptions nécessitent des signaux dédiés pour chaque type d'interruption matérielle.

Les signaux de contrôle peuvent posséder une signification sur front ou sur état. Il peut y avoir des contraintes de réponse dans le cycle, voire dans le demi-cycle.

5.1.2 Obtentions usuelles des choix architecturaux

Définition du rapport surface/temps d'exécution

Les outils de synthèse de haut niveau classiques choisissent généralement les solutions extrêmes : optimisation totale soit de la surface, soit du temps d'exécution. Cette optimisation se fait toujours au dépend de l'autre caractéristique, car le temps d'exécution et la surface sont en opposition. Il est impossible pour le concepteur de viser une solution intermédiaire précise.

Voici les différentes méthodes pour cibler une solution par la synthèse de haut niveau classique :

- Optimisation complète en surface obtenue en minimisant le nombre de registres et d'unités fonctionnelles. On reporte les opérations les plus mobiles à des cycles nécessitant moins de ressources (*force directed scheduling* [PK89]).

- Optimisation complète en temps d'exécution en parallélisant au maximum les opérations et en câblant tous les branchements conditionnels (*path-based scheduling* [Cam91]). Cependant le gain en vitesse globale n'est pas assuré par rapport à une optimisation moindre car les impacts des optimisations en temps d'exécution sont difficilement prédictibles.
- Optimisation intermédiaire entre la surface et le temps d'exécution en essayant de minimiser le nombre d'unités fonctionnelles et en câblant les branchements conditionnels.
- Ajout d'une contrainte de nombre de cycles ou un temps d'exécution comme pour l'outil *camad* [PKL88] [HP95]. Il ne s'agit pas de délimiter les frontières de chaque cycle dans la description comportementale mais plutôt de déterminer un nombre de cycles pour chaque bloc d'actions séquentielles de la description.

Ces approches peuvent être combinées pour produire un ou plusieurs circuits avec des optimisations différentes.

Définition des communications

Les langages de description apportent des informations sur la taille des connecteurs externes ainsi que leur sens. Ils sont définis par des types particuliers les différenciant des variables. Cependant, la description comportementale ne précise, ni ne contraint le protocole de communication utilisé sur ces ports.

Les outils classiques de synthèse de haut niveau donnent la possibilité au concepteur de décrire les communications de manière très fine. Ils imposent souvent des restrictions d'ordre syntaxique ou même fonctionnelle, car l'interprétation des communications est difficile pour l'outil de synthèse.
Par exemple, dans le domaine de la synthèse de haut niveau, les frontières de cycle ne sont pas positionnées clairement. Or les frontières de cycle sont critiques pour la synchronisation avec l'environnement extérieur du coprocesseur.

5.1.3 La difficulté des choix architecturaux

Le rapport surface/temps d'exécution

La description comportementale de haut niveau peut fournir des informations sur la taille des opérations logicielles. Les langages de description communément utilisés tels que le *VHDL* ou le *Verilog* intègrent la taille des variables.
Mais la taille des opérations logicielles ne permet pas de cibler précisément la surface d'un circuit. Il n'y a aucune information sur le nombre d'opérateurs matériels, ni sur leurs types.

Les frontières de cycle sont absentes d'une description comportementale de haut niveau. Les états de l'automate du circuit ne sont pas définis.

Il est impossible de déterminer le rapport entre la surface et le temps d'exécution du coprocesseur avec uniquement une description comportementale.

La définition des communications

Le respect du protocole de communication par la synthèse est indispensable pour garantir le bon fonctionnement du coprocesseur. Le concepteur peut décrire d'une façon très fine les communications du coprocesseur. Cependant l'outil de synthèse reste limité dans l'interprétation de la description. Une granularité fine est en opposition avec le flot de description de la synthèse de haut niveau. Ces communications sont noyées dans le flot de description comportementale et sont donc difficilement détectables et cernables.

Une fois le protocole de communication compris par l'outil, il faut qu'il sache le maîtriser : le protocole PI-bus, par exemple, est très complexe en nombre de signaux à gérer.

Le fait que l'entrée comportementale ne possède théoriquement pas d'information sur les frontières de cycle est un obstacle de plus. Certaines communications pouvant s'effectuer sur un ou deux cycles.

Il existe une multitude de protocoles de communication et un nombre encore plus grand de les exprimés. L'interface externe est difficile à déterminer et leur interprétation par la synthèse de haut niveau est encore moins évidente.

- ➠ La description comportementale n'est pas suffisante pour cibler le rapport surface/temps d'exécution. Elle contient, au mieux, la taille des opérations logicielles.
  La synthèse de haut niveau a besoin de directives supplémentaires.
- ➠ Les communications doivent être définies de façon stricte.

5.2 Les directives nécessaires pour cibler une solution

5.2.1 Définition des communications

Les communications sont asynchrones et dépendent des données traitées. Par conséquent, le type de protocole acceptable par le coprocesseur est uniquement celui du maître car lui seul décide du moment où il est prêt à communiquer.

Découpler les communications de la synthèse de haut niveau

Le protocole de communication doit comporter un mécanisme de synchronisation. La synthèse de haut niveau ne peut réaliser correctement la synthèse des communications vu l'étendue des possibilités (cf. chapitre 5.1.3). Nous réduisons le protocole à des instructions intégrant et masquant tout le mécanisme de communication.
De manière concrète, les instructions de communication autorisées se résument à une opération d'écriture et une de lecture.

Le coprocesseur synthétisé prend toujours l'initiative des communications. Il est donc maître vis-à-vis de l'extérieur. Il ne peut être esclave car il n'a qu'un seul automate. Il ne peut donc s'occuper à la fois des communications sur ses ports et du traitement interne des données.

Nous définissons un protocole de base systématique pour notre coprocesseur.
Si les communications de l'environnement du coprocesseur obéissent à un autre protocole, il faut insérer un module matériel de translation. La conception de ce module de translation est à la charge du concepteur.
Le protocole doit être suffisamment générique pour pouvoir être interfacé avec n'importe quel autre protocole et ne pas rajouter de la complexité au module de translation.
Le coprocesseur synthétisé, bien que maître sur ses ports, peut apparaître en temps qu'esclave vis-à-vis de l'environnement global si le module de translation est esclave des deux côtés.
Le module de translation de protocole faisant le lien avec l'environnement extérieur est conçu soit avec des outils dédiés (synthèse d'automate) soit "à la main" au niveau RTL.

En découplant les communications du traitement des données, le concepteur peut se consacrer entièrement à la synthèse de haut niveau, puisque la description comportementale des communications peut être réduites à la simple détermination de la source ou de la destination.
Ainsi, **le coprocesseur peut être conçu indépendamment du protocole de communication**. **Cela augmente la réutilisabilité du circuit**.

Simplicité du protocole

Tous les protocoles de synchronisation se ramènent dans leur plus simple expression à un signal de sélection et un signal d'acquittement accompagnant la donnée.

La figure 5.9 représente les états de synchronisation de l'automate et les interfaces du circuit pour les communications.
Les signaux *READ* et *WRITE* sont équivalents à des signaux de sélection, vus du maître.
Les signaux *READ OK* et *WRITE OK* sont équivalents à des signaux d'acquittement, vus du maître.

état de lecture de l'automate — *interface du port de lecture* — *état d'écriture de l'automate* — *interface du port d'écriture*

lecture(**ENTREE**, registre) — écriture(**SORTIE**, registre)

FIG. 5.9 lecture et écriture.

Les opérations de lecture et d'écriture dans la description comportementale sont de simples appels de fonction : *lecture(entrée, destination)* et *écriture(sortie, source)* où *source* et *destination* sont des registres internes au circuit, *entrée* et *sortie* sont, respectivement, des ports d'entrée et de sortie du circuit.
Les ports ne peuvent être bi-directionnels.

Le protocole de communication choisi pour le coprocesseur est suffisamment simple et générique pour permettre de l'interfacer avec n'importe quel autre protocole via un module de translation.

5.2.2 Les différentes directives architecturales

Les différentes directives que peut donner l'utilisateur à l'outil pour résoudre le problème d'antagonisme entre la surface et le temps d'exécution, sont locales ou globales à la description comportementale. Les directives architecturales sont locales si elles sont rattachées à des opérations fonctionnelles de la description comportementale (par exemple, faire un branchement câblé). Les directives architecturales sont globales si elles s'appliquent à l'intégralité de la description (le nombre d'opérateurs du circuit par exemple).

	désimplication du concepteur	**interprétabilité par la HLS**	**prédictibilité du temps d'exécution**	**prédictibilité de la surface**
nombre de cycles par bloc séquentiel	$-$	$+$	$++$	$-$
tests à câbler	$+$	$++$	$-$	$-$
boucles à dérouler, à replier	$+$	$++$	$-$	$-$
nombre de registres	$-$	$+$	$--$	$--$
affectation des registres	$-$	$++$	$+$	$-$
nombre d'opérateurs combinatoires	$+$	$+$	$--$	$+$
affectation des opérateurs combinatoires	$--$	$++$	$+$	$++$
taille des variables	$-$	$++$	$--$	$+$

FIG. 5.10 Les directives du concepteur pouvant être ajoutées à la description comportementale.

La figure 5.10 présente les caractéristiques des différents types de directives supplémentaires que l'on peut fournir à l'outil de synthèse. Le signe $+$ signale un point fort de la directive utilisée, tandis que le signe $-$ signale sa faiblesse.
Nous caractérisons les qualités des directives :

- La *désimplication du concepteur* montre la facilité avec laquelle le concepteur peut fournir les informations sans entrer, lui-même, dans une phase conception dite "à la main".

- L'*interprétabilité pour la HLS* nous renseigne sur les possibilités d'interprétation et d'utilisation de l'information par l'outil de synthèse haut niveau pour optimiser le résultat de la synthèse.
- La *prédictibilité du temps d'exécution* est la certitude pour le concepteur d'obtenir une vitesse d'exécution précise pour son circuit. C'est la capacité à **cibler la vitesse** du circuit.
- La *prédictibilité de la surface* est la certitude pour le concepteur d'obtenir une surface précise pour son circuit. C'est la capacité à **cibler la surface** du circuit.

La pertinence des directives diffère suivant leur type :

⇒ *Le nombre de cycles par bloc séquentiel* :
Pour chaque groupe d'assignations sans branchement dans la description comportementale, le concepteur définit un nombre de cycles.
Cette directive est suffisante pour cibler une solution car elle optimise le nombre de ressources en fonction du nombre de cycles à tenir. Elle laisse quand même un choix à l'outil dans l'affectation des ressources matérielles.

⇒ *Les tests à câbler* :
Cette directive spécifie de manière locale quels sont les branchements de la description à câbler et quels sont ceux qu'il ne faut pas câbler (cf. chapitre 5.1.1).
Les tests câblés permettent de supprimer un état dans l'automate mais augmentent la surface du chemin de données.
Si la surface ou bien le temps d'exécution ont été complètement caractérisés, le câblage des branchements peut se déduire, respectivement, des ressources disponibles ou du nombre de cycles requis.

⇒ *Les boucles à dérouler ou à replier* :
Cette directive spécifie de manière locale quelles sont les boucles de la description à dérouler ou à replier (cf. chapitre 5.1.1). Les boucles repliées nécessitent des opérateurs supplémentaires pour optimiser la latence des boucles. Le concepteur doit obligatoirement signaler les boucles à dérouler ou à replier car leurs gains sont difficilement prédictibles.

⇒ *Le nombre de registres* :
Il spécifie le nombre maximal d'éléments mémorisants dans le chemin de données.
Le nombre d'éléments mémorisants nous donne le nombre de valeurs différentes pouvant être mémorisées sur un front d'horloge. Cependant il ne donne aucune information sur le nombre d'opérateurs combinatoires et le partage des registres.

⇒ *L'allocation et affection des registres* :
Cette directive lie chaque registre à un ou plusieurs résultats d'opérations fonctionnelles de la description comportementale.
L'affectation des registres permet de déduire les fronts d'horloge dans le graphe de dépendance de données (DFG). Il n'apporte aucune information sur la parallélisation de deux opérations sans dépendances de données entre elles. L'outil peut donc soit paralléliser au maximum ces opérations indépendantes pour optimiser la vitesse, soit les séquentialiser pour optimiser la surface.

⇒ *Le nombre d'opérateurs combinatoires* :
Il spécifie le nombre maximal d'opérateurs par type (ALU, décaleur, multiplieur, etc...) utilisables dans le chemin de données.

Il n'y a pas d'information sur le partage des opérateurs combinatoires, le degré de parallélisation n'est pas connu. Les multiplexeurs ne sont pas pris en compte. La surface est mal définie et les temps de d'exécution sont difficilement prédictibles.

⇒ *L'affectation des opérateurs combinatoires* :
Cette directive lie les opérateurs matériels aux opérations logicielles contenues dans la description comportementale.
Pour avoir une affectation des opérateurs combinatoires qui minimise le nombre de multiplexeurs, le concepteur doit penser à l'*affectation des registres*. Établir l'affectation des opérateurs combinatoires revient à établir l'affectation de tous les opérateurs du chemin de données ! Le chemin de données est donc complètement caractérisé. La parallélisation des opérations est déterminée par l'utilisation maximale des ressources.

⇒ *taille des variables* :
La taille des variables permet de déterminer la taille des opérations à affecter sur les opérateurs matériels (cf. chapitre 5.1.1). Mais le nombre d'opérateurs du chemin de données est indéterminé.
Le concepteur doit trouver la taille minimale des variables pour minimiser la surface. La conversion d'algorithme flottant en virgule fixe n'est pas évidente.

Caractériser complètement un critère revient à définir l'autre grâce à leur antagonisme. Ainsi si nous définissons les ressources matérielles, le temps d'exécution est défini par l'optimisation maximale de ces ressources. Et inversement, connaissant le nombre de cycles pour une séquence d'opérations, nous pouvons déterminer le nombre minimal d'opérateurs matériels nécessaires.

Le *nombre de cycles par bloc séquentiel* caractérise complètement le temps d'exécution. L'*affectation des opérateurs combinatoires et des registres* caractérise complètement la surface.
Ces deux directives sont donc très précises pour cibler un circuit. Elles ont malheureusement des inconvénients majeurs :

⇒ *L'affectation des opérateurs combinatoires et des registres* est la meilleure caractérisation que le concepteur puisse faire.
Cette directive force le concepteur à penser complètement l'architecture du chemin de données du circuit. Cette démarche demande trop de travail pour le concepteur.

⇒ Le *nombre de cycles par bloc séquentiel* est une information suffisante pour cibler une solution car elle optimise le nombre de ressources en fonction du nombre de cycles à tenir.
Mais elle ne correspond pas à la méthode de résolution des contraintes électriques. Dans le chapitre 4.4.1, nous montrons comment respecter ces contraintes en augmentant le nombre de cycles dans l'automate.
Avec cette méthode, nous ne pouvons pas garantir le nombre de cycles de l'automate après *réordonnancement*. La directive du concepteur ne sera sans doute pas respectée.

Ni l'*affectation des opérateurs combinatoires et des registres* et ni le *nombre de cycles par bloc séquentiel* ne sont des directives utilisables pour la synthèse ciblée de haut niveau.
Considérées séparément, les autres directives ne sont pas suffisantes pour cibler le rapport entre la surface et le temps d'exécution. Nous devons donc combiner plusieurs de ces directives pour cibler efficacement une solution.

5.2.3 Les directives architecturales nécessaires

Nous choisissons cinq informations essentielles pour décrire le coprocesseur ciblé :

1. la *description comportementale*
2. la *taille des variables*
3. les *boucles à dérouler ou à replier*
4. l'*affectation des registres*
5. le *nombre d'opérateurs combinatoires*

D'après le chapitre 5.1.1, des choix architecturaux ne sont déductibles automatiquement, deux directives doivent être obligatoirement fournies par le concepteur à l'outil :

- la *taille des variables*
  La détermination de la taille de proche en proche à partir de la taille des ports de l'interface du circuit ne permet pas d'obtenir la taille minimale des variables.
- les *boucles à dérouler ou à replier*
  Le gain de l'optimisation d'une boucle en nombre de cycles est difficilement évaluable. Le nombre d'itérations est dépendant des données.
  Comme l'optimisation de la boucle possède un coût important en surface (parallélisation des opérations), l'outil de synthèse laisse ce choix à l'utilisateur.

Ces informations ne sont pas suffisantes pour caractériser le rapport surface/temps d'exécution du coprocesseur.

- l'*affectation des registres*
  Elle fournit les frontières de cycles entre des opérations ayant une dépendance de données.
- Le *nombre d'opérateurs combinatoires*
  Il fournit le degré de parallélisation des opérations indépendantes.

La parallélisation est déterminée par l'utilisation maximale de ces ressources. Les branchements sont câblés s'il y a suffisamment de ressources disponibles pour paralléliser les deux actions.
Le circuit est suffisamment bien ciblé.

L'affectation des registres

Pour chaque opération fonctionnelle de la description, le concepteur doit déterminer si son résultat est stocké dans un registre et quel est ce registre. Si aucun registre n'est désigné, le résultat de l'opération logicielle est un signal intermédiaire.

Le concepteur peut allouer plus de registres que nécessaire. L'affectation des registres décrit par l'utilisateur peut aussi être incompatible avec la description comportementale : il faut prendre en compte la durée de vie des variables.

Dans l'exemple de la figure 5.11, le concepteur a lié le registre *R1* à deux opérations différentes. Ce registre est réutilisé à mauvais escient car la valeur *2* qu'il contenait a été écrasée par la valeur *10* avant d'avoir pu être utilisée.

$a = 2$	$\mathbf{R1} \longleftarrow 2$
$b = a + 1$	$\mathrm{R2} \longleftarrow 3$
$c = 10$	$\mathbf{R1} \longleftarrow 10$!!!
$d = a + 1$	$\mathrm{R3} \longleftarrow \mathbf{R1} + 1 = \mathbf{10} + 1 \quad (\mathbf{R1} \neq \mathbf{2}$!!!)

FIG. 5.11 Mauvais partage du registre **R1**.

Pour simplifier l'affectation des registres, nous décidons d'attacher à chaque variable un registre dédié unique. Une variable de la description comportementale est donc soit un registre, soit un signal interne. L'information de l'affectation des registres est alors portée par la description comportementale elle-même.
Les registres synthétisés possèdent le même nom que les variables de la description comportementale. Le concepteur retrouve ses variables, cela facilite sa compréhension de la description du circuit générée au niveau RTL. Cela n'est souvent pas le cas avec les outils de synthèse classiques dont les registres ne correspondent à aucune variable de la description.

Les choix restant à l'outil de synthèse haut niveau

L'*affectation des registres* associée au *nombre d'opérateurs combinatoires* laisse peu de choix à l'outil de synthèse.
L'outil doit faire un choix entre :

- minimisation de la surface :
  La taille des opérateurs matériels et le nombre d'entrées des multiplexeurs sont minimisés en affectant les opérations de même type sur les mêmes opérateurs matériels.
- minimisation du temps d'exécution :
  Les opérations logicielles peuvent être parallélisées en utilisant au maximum les opérateurs matériels.

Le but principal de l'outil est de minimiser le temps d'exécution . L'outil doit aussi veiller à ne pas trop augmenter la surface du circuit.

Les conflits avec la description comportementale

Les directives peuvent entraîner des conflits avec la description comportementale du circuit.
Dans ce cas, l'outil de synthèse de haut niveau doit informer le concepteur qu'il ne peut respecter les contraintes. La synthèse est stoppée car le conflit peut mettre en valeur un problème dans les choix architecturaux.

Il y a un conflit si le nombre d'opérations logicielles comprises entre deux registres excède le nombre d'opérateurs.
Il y a un conflit lorsque la valeur d'une variable de la description comportementale non affectée à un registre possède une durée de vie ne correspondant pas à un signal (par exemple, la variable d'un compteur de boucle doit être systématiquement affectée à un registre).

La flexibilité des directives

Afin de faciliter l'exploration architecturale, le concepteur doit pouvoir modifier ses directives architecturales sans avoir à changer la description comportementale. Pour rendre plus indépendante la description comportementale des directives du concepteur, l'outil de synthèse ciblée de haut niveau doit permettre une certaine flexibilité des contraintes architecturales.
Si au contraire, le concepteur veut décrire très précisément un circuit, il doit pouvoir augmenter les contraintes architecturales.

Grâce à un paramétrage de l'outil de synthèse ciblée de haut niveau, le concepteur peut rendre les directives architecturales plus précises (**+**) ou moins précises (**-**). La flexibilité se joue sur 4 points :

- Le *nombre d'opérateurs combinatoires* peut être moins précis.
L'outil peut ajouter des opérateurs logiques sans que ceux-ci soient énumérés par le concepteur car les opérateurs logiques ont un faible coût en surface. La synthèse logique optimise très bien l'utilisation de ces opérateurs.
Cela libère le concepteur de l'énumération de toutes les petites opérations logiques de la description comportementale.

- L'*affectation des registres* peut être moins précise.
L'outil peut s'autoriser à faire des by-pass de registre dans certains cycles si le *nombre d'opérateurs combinatoires* le permet.
L'outil peut aussi introduire un accumulateur si le *nombre d'opérateurs combinatoires* ne permet pas de faire une suite d'opérations en un seul cycle.

+ L'*affectation des opérateurs combinatoires* peut être précisée.
Elle peut être partielle ou sur la totalité du chemin de données. Le concepteur est ainsi sûr qu'un chemin particulièrement critique du circuit obtiendra les caractéristiques ciblées.

+ Le *câblage des branchements* peut être précisé.
Le concepteur peut forcer ou interdire à l'outil de câbler les branchements conditionnels grâce à des directives locales dans la description comportementale.

✓ La synthèse des communications est dissociée de la synthèse de haut niveau. Le protocole de communication est prédéfini par l'outil de synthèse ciblée.

✓ La synthèse ciblée de haut niveau nécessite des directives architecturales en plus de la description comportementale :
- la *taille des variables*
- les *boucles à dérouler ou à replier*
- l'*affectation des registres*
- le *nombre d'opérateurs combinatoires*

✓ Les directives permettent de cibler précisément un circuit. Dans une certaine mesure, le résultat de la synthèse ciblée est prédictible.

✓ Les directives sont flexibles pour pouvoir s'adapter à des contraintes plus ou moins fortes. La précision de la synthèse ciblée s'adapte au temps dont dispose le concepteur pour la description des contraintes architecturales.

5.3 Exemples de directives dans la synthèse ciblée

L'entrée de la synthèse haut niveau contient au minimum une description comportementale.

Prenons la description de la figure 5.12. Cette description comportementale est écrite en **C**. Elle calcule le plus grand diviseur commun (pgcd) deux par deux sur 16 données. Les 8 pgcd sont additionnés. La lecture des données est constituée de deux appels successifs de la fonction *read* sur le même port d'entrée. Le résultat du calcul est envoyé sur le port de sortie grâce à la fonction *write*.

```
int Var1;   // 16 bits
int Var2;   // 16 bits
int Var3;   // 32 bits
int i;      // 5 bits
int Port1;  // entrée 16 bits
int Port2;  // sortie 32 bits

main()
{
  //processus
  while( 1 )
  {
    Var3 = 0;

    for ( i = 0; i <= 16; i++ )
    {
      read( port1, Var1 );
      read( port1, Var2 );

      //plus grand diviseur commun
      while( Var1 != Var2 )
      {
        if ( Var1 - Var2 < 0 )
             Var2 = Var2 - Var1;
        else Var1 = Var1 - Var2;
      }

      Var3 = Var3 + Var1;
    }

    //somme des pgcd
    write( port2, Var3 );
  }
}
```

FIG. 5.12 Exemple de description comportementale.

La taille des variables est en commentaire en face de chaque déclaration de variable dans la description. Nous pouvons donc déterminer les tailles des opérations.
Les boucles ne sont ni déroulées, ni repliées car il n'y a pas de directives présentes dans la description.

Nous allons voir trois exemples de directives architecturales appliquées à cette description comportementale.

5.3.1 Exemple de directives minimales d'affectation

Les directives minimales comprennent à priori la description comportementale (cf. figure 5.12).
Le concepteur donne les directives d'*affectation des registres* et le *nombre d'opérateurs combinatoires* représentées symboliquement par la figure 5.13 pour l'exemple de la figure 5.12.

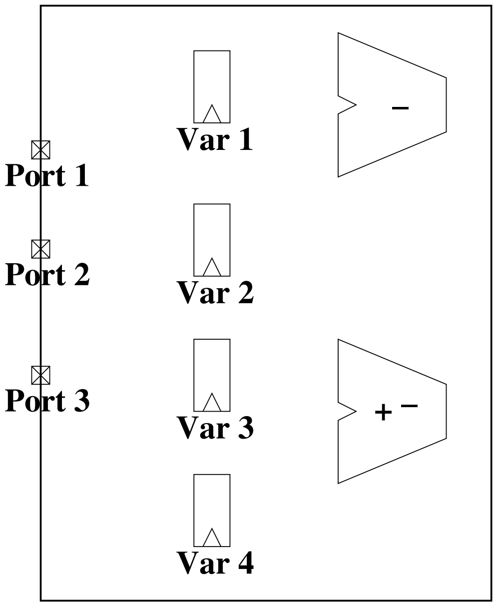


FIG. 5.13 Directives d'affectation minimales.

Impact sur l'affectation des opérateurs

Les directives de la figure 5.13 implique que toutes les variables de la description comportementale soient instanciées en tant que registres. La taille des registres est égale à la taille des variables auxquelles ils se substituent.
La figure 5.13 instancie deux opérateurs combinatoires : un soustracteur simple et un additionneur/soustracteur. La taille de ces opérateurs est déterminée par leur affectation.

L'*allocation des opérateurs combinatoires* permet de contrôler le nombre de cycles.
Les deux additions de la description comportementale (cf. figure 5.12) ne peuvent se faire que sur l'additionneur/soustracteur matériel. Cet opérateur combinatoire aura donc une taille maximale de 32 bits car la variable *Var3* est de 32 bits.

Impact sur le câblage des branchements

```
                .
                .
                .
if ( Var1 - Var2 < 0 )
     Var2 = Var2 - Var1;
else Var1 = Var1 - Var2;
                .
                .
                .
```

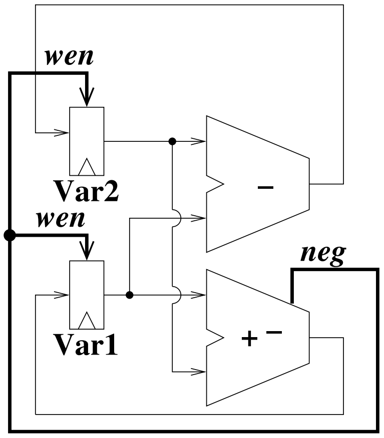


FIG. 5.14 Extrait du chemin de données généré pour le branchement câblé.

Les opérations de comparaison de la description nécessitent un soustracteur pour pouvoir tester les *drapeaux négatif ou zéro*.
Cependant le branchement conditionnel peut être câblé (cf. figure 5.14) car le test procède à la même opération que l'écriture du registre *Var1*. Pour câbler ce branchement conditionnel, nous n'avons besoin que de deux soustracteurs. Le branchement est donc câblé.

5.3.2 Exemple de directives maximales d'affectation

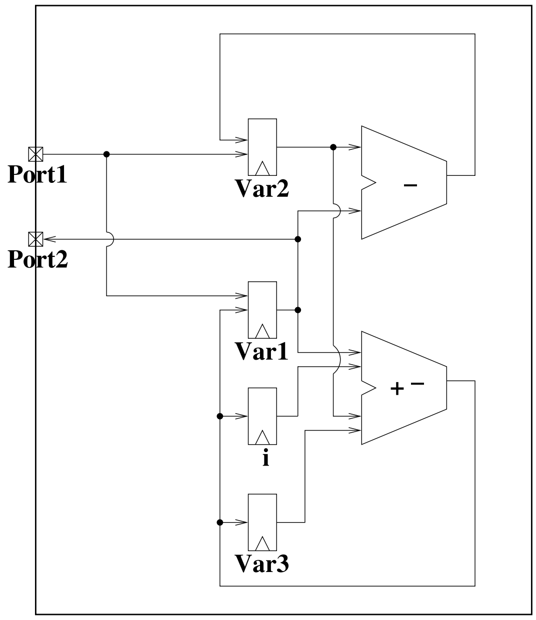


FIG. 5.15 Directives maximales d'affectation.

Pour être sûr de l'affectation et donc contrôler les multiplexeurs , le concepteur doit fournir l'*affectation des opérateurs combinatoires*.
La figure 5.15 montre les directives d'affectation ajoutées par le concepteur.

Impact sur l'affectation des opérateurs

Les ports des soustracteurs ne sont pas commutatifs.
Contrairement au chapitre 5.3.1, les variables *Var1* et *Var2* sont connectées aux opérateurs matériels. Le soustracteur peut faire l'opération *Var2 - Var1* mais pas *Var1 - Var2*. L'additionneur/soustracteur ne peut faire que l'opération *Var1 - Var2*.

L'outil de synthèse ne peut pas choisir l'opérateur matériel. Ainsi, le concepteur ne peut pas manquer la cible qu'il s'était fixé.
Dans cet exemple, il décide d'équilibrer le nombre d'entrées des multiplexeurs.

5.3.3 Exemple de directives sous-optimales d'affectation

Le respect des directives du concepteur est prioritaire sur l'obtention d'un *point de Pareto* (solution optimale).
Dans l'exemple de la figure 5.16, le concepteur a changé son affectation des opérateurs combinatoires par rapport à celui de la figure 5.15.

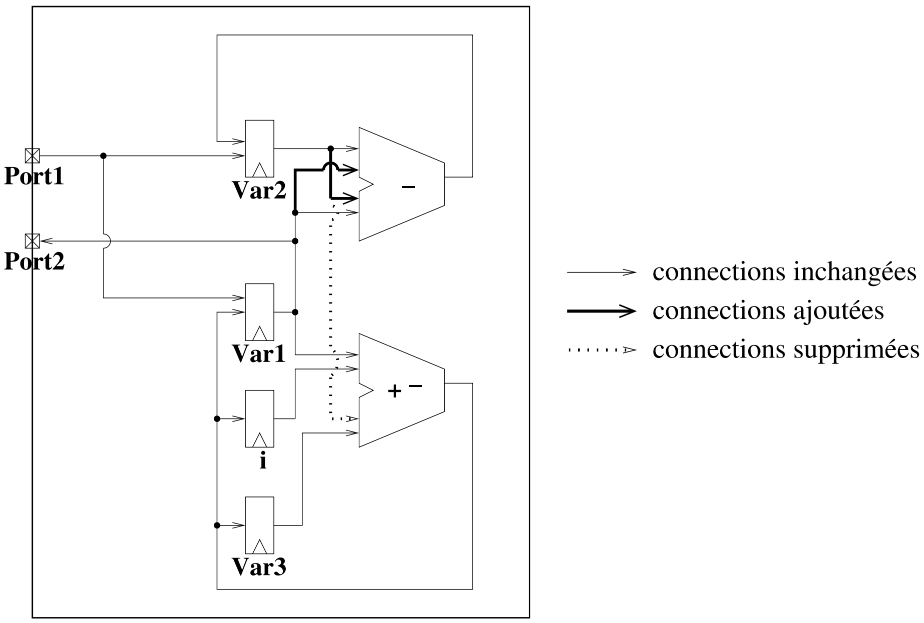


FIG. 5.16 Exemple de mauvaises contraintes d'affectation données par le concepteur.

Impact sur le câblage des branchements

Le concepteur a décidé de faire les opérations *Var2 - Var1* et *Var1 - Var2* uniquement sur le soustracteur. Ce qui peut paraître, à priori, un bon choix car cela libère l'additionneur/soustracteur.

Or pour faire le branchement câblé, les deux opérateurs matériels sont nécessaires. D'après les directives, seul le soustracteur est utilisable pour faire des opérations sur *Var1* et *Var2*.
Le branchement conditionnel ne pourra pas être câblé.

Et pourtant, le concepteur n'a pas diminué la surface puisqu'il y a même d'avantage de multiplexeurs. Il obtient donc un circuit plus gros et plus lent.

Grâce aux directives architecturales, le concepteur peut aller jusqu'à cibler une solution sous-optimale !

- ✓ Les directives architecturales permettent de cibler des circuits atypiques. Le concepteur peut même forcer l'outil à choisir une solution moins performante.
- ✓ Peu de modifications dans les directives architecturales sont nécessaires pour obtenir un circuit différent.

5.4 Conclusion

La caractérisation d'un circuit synthétisé se fait sur de nombreux critères. Or certaines caractéristiques ne peuvent être déduites de façon automatique par un outil de synthèse de haut niveau :

- ➠ Les protocoles de communication sont de trop bas niveau pour être synthétisés par la synthèse de haut niveau.
- ➠ L'outil nécessite des directives supplémentaires pour choisir le type d'optimisation (en surface ou en temps d'exécution) pour le circuit.

Le concepteur doit pouvoir cibler très précisément le circuit qu'il veut obtenir. La méthode utilisée est de limiter le nombre de choix pour l'outil de synthèse :

- ✓ Les communications et leurs latences sont indépendantes de la synthèse de haut niveau. La synthèse de haut niveau se contente d'instancier des protocoles prédéfinis au moyen d'instructions simples dans la description comportementale. Le protocole de communication utilisé par le coprocesseur synthétisé utilise seulement deux signaux de synchronisation par port de données.
- ✓ Pour cibler une solution, le concepteur doit définir l'affectation des registres, le nombre d'opérateurs combinatoires, les tailles des variables et la méthode de traitement de chaque boucle.
- ✓ L'ordonnancement et l'affectation des opérateurs combinatoires restent à déterminer. L'outil a la charge de maximiser le parallélisme. Ces problèmes sont développés dans le chapitre 6.

Chapitre 6

Obtention rapide d'une solution

Sommaire

L'obtention d'une solution passe par la détermination des opérateurs matériels mis en jeu et du moment de leur activation. Cette phase importante de la synthèse qui décide des performances du circuit se nomme l'ordonnancement.

Dans une description comportementale complètement synchronisée, l'ordonnancement est déjà fait. En effet, dans les descriptions synchronisées, l'allocation minimale est directement déductible du nombre d'opérateurs utilisés dans chaque cycle.
La figure 6.1 montre l'évidence d'une allocation lorsqu'un cycle est bien délimité par deux instructions de synchronisation sur le cycle d'horloge (i.e les instructions *wait*). L'allocation minimale est le nombre d'opérateurs nécessaires pour satisfaire les contraintes de chaque cycle.

```
wait on ck until ck=1 ;
   X := A + B ;
   Y := X * C ;
   Z := D + 1 ;
wait on ck until ck=1 ;
```

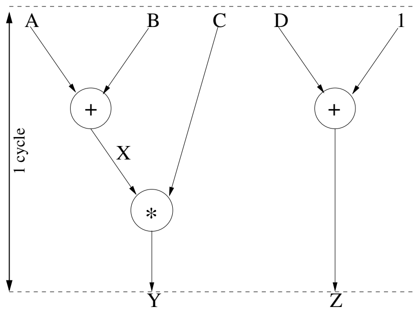


FIG. 6.1 Deux additionneurs et un multiplieur sont nécessaires pour le cycle délimité par deux *wait*.

Cela n'est pas le cas dans une description dépourvue de synchronisation qu'est censée accepter la synthèse de haut niveau. La synthèse de haut niveau doit faire un ordonnancement.
L'entrée comportementale écrite en langage impératif constitue des blocs d'instructions séquentielles reliées par des branchements conditionnels. Reprendre les branchements et la séquentialité de la description comportementale serait sous-optimal pour une architecture visée. En effet, la synthèse peut modifier les branchements et paralléliser certaines instructions pour diminuer la latence globale du circuit.

La vitesse et l'efficacité de la synthèse de haut niveau dépendent de la qualité de son ordonnancement. Nous verrons deux types d'ordonnancement : l'ordonnancement matériel et l'ordonnancement logiciel. Quels sont les liens entre ces deux types d'ordonnancement et notre synthèse ciblée ?

La synthèse ciblée de haut niveau nécessite des directives architecturales. Nous verrons comment nous pouvons simplifier la complexité de l'algorithme de synthèse grâce à ces contraintes.
Quels objectifs reste-t-il à l'algorithme de synthèse ?

6.1 L'ordonnancement dans la synthèse classique

La synthèse de haut niveau est constituée de trois phases : l'élaboration du CDFG (graphe de flot de contrôle et de flots de données), l'ordonnancement et l'allocation.
L'élaboration du CDFG détermine l'optimisation des branchements et des boucles. L'ordonnancement évalue à quel cycle les opérations fonctionnelles sont effectuées. L'allocation détermine l'affectation des opérations fonctionnelles et le nombre d'opérateurs matériels.

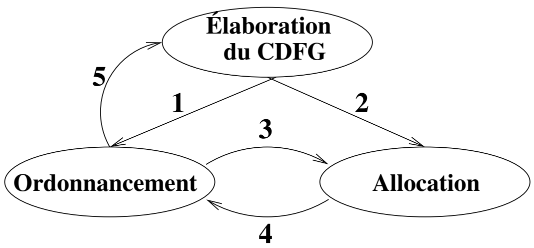


FIG. 6.2 Cercles vicieux de la synthèse de haut niveau.

La figure 6.2 montre la complexité de la synthèse de haut niveau. Ce sont des phases qui paraissent chronologiques mais qui sont en fait très liées. L'ordonnancement est au centre des cycles de dépendance.
Chaque arc, numéroté de 1 à 5, représente une dépendance entre les phases :

Arc 1 L'*ordonnancement* se base sur les DFG extraits du CDFG (control data flow graph). Chaque DFG va être ordonnancé séparément.

Arc 2 Le CDFG influence le nombre minimal d'opérateurs en optimisant les boucles et les tests.

Arc 3 L'allocation se base sur la parallélisation des opérations fonctionnelles pour déterminer le nombre de ressources matérielles nécessaires.

Arc 4 Le multiplexage d'un opérateur modifie ses caractéristiques temporelles. Il faut remettre en cause les frontières de cycle.

Arc 5 Les contraintes temporelles et de surface révèlent que l'on ne peut optimiser efficacement le DFG. Il faut remettre en cause le CDFG.

L'ordonnancement est indissociable des deux autre phases de la synthèse, la construction du CDFG et l'allocation. Cela forme un cercle vicieux dans les décisions d'optimisation. L'algorithme ne peut être rapide et en même temps intégrer les cycles de remise en cause de ses décisions.

6.1.1 Construction du CDFG

Le CDFG peut prendre plusieurs formes suivant les décisions d'optimisation des branchements tels que le déroulement de boucles (cf. chapitre 5.1.1) ou bien les branchements câblés.
Des opérations peuvent être déplacées ou dupliquées lors de ces optimisations.

La figure 6.3 présente un branchement conditionnel. Il existe au moins deux choix possibles pour insérer ce branchement dans le CDFG. Le CDFG avec le *branchement non câblé* représente un squelette d'automate à états avec deux branches et des états dissociés. Le CDFG avec le *branchement câblé* représente un squelette d'automate avec une seule branche, les tests sont reportés dans la partie opérative du chemin de données.

Les branchements du CDFG restant nécessitent une frontière de cycle pour évaluer le résultat du test. Les blocs séquentiels du CDFG constituent des DFG dans lesquels il faut déterminer les frontières de cycles suivant les temps de propagation des opérateurs matériels et la fréquence d'horloge.

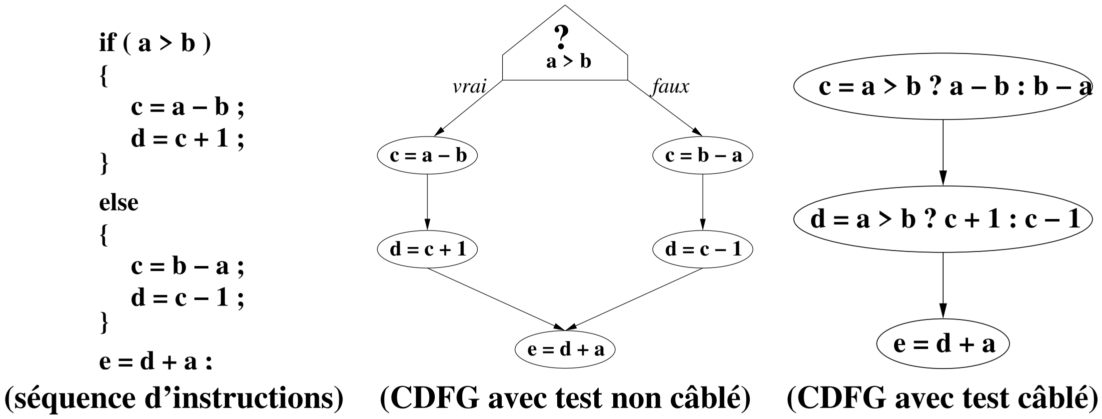


FIG. 6.3 Exemple de topologies différentes du CDFG.

6.1.2 Les frontières de cycle

Les frontières de cycles sont déterminées par un ordonnancement (ASAP, etc...) sur les DFG (cf. figure 6.4). Elles doivent tenir compte des temps de propagation des opérateurs matériels. Ces temps sont souvent évalués par des heuristiques suivant le poids de l'opération fonctionnelle. Par exemple, une multiplication sera estimée deux à trois fois plus lente qu'une addition.

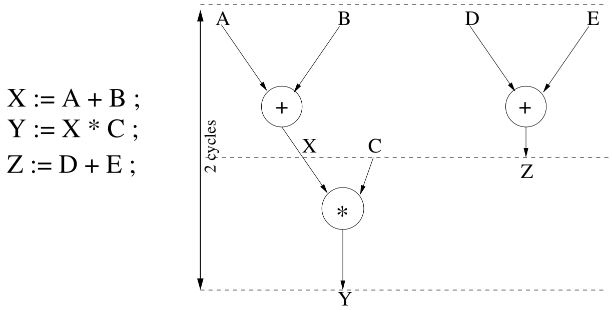


FIG. 6.4 ASAP sur une séquence d'opérations.

Les opérations fonctionnelles indépendantes sont parallélisées pour diminuer le nombre de cycles. Le degré de parallélisation détermine, en partie, la taille du circuit.

Les dépendances des opérations du DFG sont purement fonctionnelles. Chaque noeud représente une seule opération fonctionnelle et chaque arc une dépendance fonctionnelle. Il n'y a pas d'idée de registres et les arcs ne représentent pas de frontières temporelles. Ce n'est qu'en corrélant les temps de propagation estimés des opérations avec la fréquence d'horloge que l'on peut positionner les frontières de cycle sur le DFG.

6.1.3 L'allocation

L'allocation consiste en la détermination du nombre de registres et d'opérateurs combinatoires et de leurs affectations respectives [GDWL92]. La connectique (les multiplexeurs) est ajoutée en fonction de l'affectation.

L'affectation des registres et d'opérateurs combinatoires constitue la phase finale de la synthèse. Le nombre minimal de ressources est déterminé par l'ordonnancement.

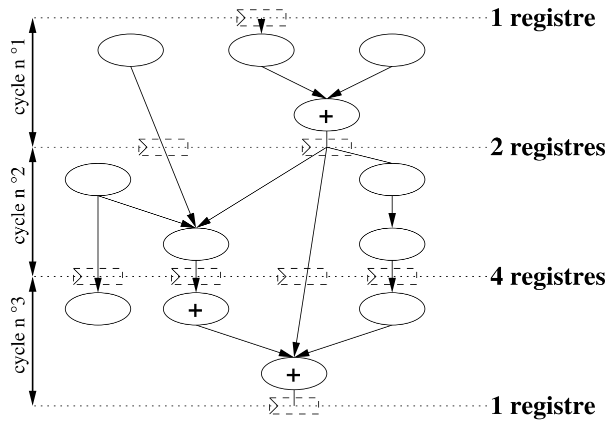


FIG. 6.5 Il faut 4 registres pour cet ordonnancement.

Dans la figure 6.5, l'ordonnancement est fait sur un DFG en 3 cycles. Il y a 4 valeurs différentes à stocker entre le *cycle 2* et le *cycle 3*. C'est le nombre maximal de valeurs temporaires à stocker entre deux cycles. Il faut au minimum 4 registres pour faire cet ordonnancement.
Si nous observons le nombre d'opérateurs $+$ dans la figure, nous constatons qu'il faut au minimum deux additionneurs pour satisfaire les besoins matériels de tous les cycles et en particulier ceux du *cycle 3*.

Le but de l'affectation est de minimiser le nombre d'opérateurs matériels et les insertions de multiplexeurs pour minimiser, respectivement, la surface et les temps de propagation.
Par exemple, l'affectation des registres détermine quel registre sera utilisé pour stocker la valeur entre deux cycles. Ce choix est fait en fonction de la durée de vie de cette valeur (pour éviter les conflits d'écriture sur les registres) et des multiplexeurs créés (pour minimiser leurs tailles).

- Les trois phases de la synthèse (l'élaboration du CDFG, l'ordonnancement et l'allocation) sont très liées.
- Les algorithmes classiques de la synthèse de haut niveau ne sont pas suffisamment simples pour être rapides.
- L'ordonnancement des opérations repose uniquement sur les dépendances fonctionnelles. L'ordonnancement n'a pas d'indication sur le degré de parallélisation. L'objectif du concepteur est donc mal ciblé.

6.2 L'ordonnancement dans la compilation logicielle

L'objectif de la compilation logicielle sur des architectures dédiées est de produire la séquence d'instructions la plus rapide possible pour l'architecture cible. Il s'agit d'utiliser au maximum les capacités du circuit.
L'optimisation de l'ordonnancement se résume à la parallélisation de micro-instructions dans un pipeline (cf. [HD01]) ou d'instructions dans des circuits superscalaires.

C'est la compilation pour processeur VLIW (Very Large Instruction Word) qui se rapproche le plus de la synthèse de haut niveau. Pour les processeurs VLIW, un mot d'instruction contient plusieurs opérations indépendantes.

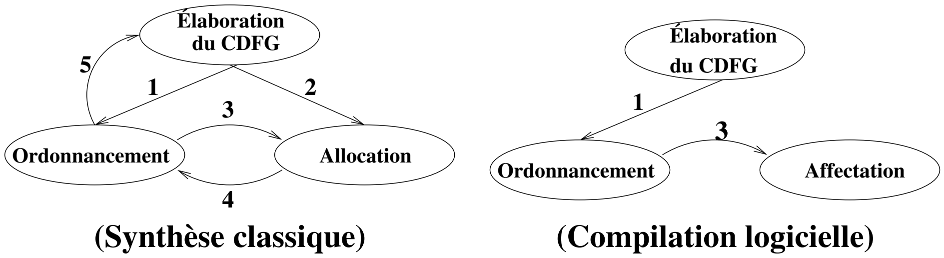


FIG. 6.6 Phases de la compilation logicielle.

Le nombre d'opérateurs matériels est déjà déterminé par l'architecture du circuit, l'algorithme d'optimisation en est donc simplifié (cf. figure 6.6). Il ne reste de la phase d'allocation que l'affectation des registres et des opérateurs combinatoires.

Les choix d'optimisation du CDFG sont presque les mêmes que pour la synthèse de haut niveau classique. L'objectif est de diminuer le nombre de cycles. Par contre, il n'y a pas d'optimisation de la surface puisque celle-ci est bornée. Les branchements conditionnels ne peuvent être câblés (cf. chapitre 5.1.1). Il s'agit d'optimiser le nombre de cycles des boucles (déroulement de boucles, pipeline du corps de boucle, projection de boucles imbriquées en les fusionnant). L'objectif est de paralléliser au maximum les instructions. C'est donc un ordonnancement sous contraintes de ressources.
Les temps d'exécution en nombre de cycles des instructions peut varier d'une instruction à l'autre. Cependant ces temps sont parfaitement connus. Cela facilite la détermination des cycle lors de l'ordonnancement.

L'ordonnancement de la compilation logicielle s'effectue sur les DFG constituant le CDFG. Il consiste à paralléliser le plus possible d'opérations et ordonnancer en priorité les opérations de mobilités les plus faibles.

Il n'y a plus que deux dépendances entre les phases de la compilation dans la figure 6.6. Ces dépendances sont à comparer avec celles de la synthèse :

Arc 1 Les choix d'optimisation du CDFG influencent le nombre de cycles.

Arc 2 Cette dépendance a disparu car le choix du CDFG ne peut plus influencer le nombre de ressources matérielles.

Arc 3 A chaque cycle, on détermine quel registre va stocker le résultat.

Arc 4 Cette dépendance a disparu car l'affectation d'un registre ou d'un opérateur combinatoire ne peut modifier ses caractéristiques temporelles.

Arc 5 Cette dépendance a disparu car l'optimisation des boucles est systématique et n'est pas remise en cause. Il n'y a pas de temps de propagation ni de surface en jeu.

- Les phases de la compilation logicielle sont les mêmes que celles de la synthèse de haut niveau.
- Il y a peu de modifications possibles du CDFG.
- La phase d'allocation se limite à l'affectation des ressources.
- Les phases de la compilation peuvent être complètement dissociées car le nombre de choix est limité. L'algorithme d'ordonnancement est donc plus simple car il n'y a pas de remise en cause des choix.

6.3 L'ordonnancement avec l'affectation fixée des registres

L'*affectation des registres* dans la synthèse de haut niveau ou bien dans la compilation logicielle lie des registres à des variables de la description comportementale.
Pour obtenir l'*affectation des registres*, il faut procéder à la phase d'*ordonnancement* des opérations fonctionnelles. La figure 6.7 illustre l'obtention d'une description comportementale après la phase d'*affectation des registres*.
Des contraintes matérielles issues de l'*affectation des registres* apparaissent.

Le **RFG** (Register transfer Flow Graph) issu d'un DFG (Data Flow Graph) après l'*affectation des registres* est un graphe mêlant dépendance de données et contraintes matérielles. Les noeuds du graphe expriment une écriture dans un registre et les arcs des contraintes de précédence entre les noeuds. [CG96], par exemple, l'utilise pour optimiser l'*affectation des opérateurs* et diminuer ainsi le nombre d'entrées des multiplexeurs.
Les RFG peuvent être agrégés dans un **CRFG** (Control et Register transfer Flow Graph) global à la description. Ce CRFG global est un graphe ou l'affectation des registres est fixée mais où l'on peut remettre en cause l'ordonnancement des opérations.

Le RFG autorise une certaine liberté de réordonnancement sur ses noeuds. Cette description comportementale est utilisable pour effectuer un nouvel ordonnancement par un outil de synthèse.
Les contraintes sont soit fonctionnelles, soit matérielles.

Les contraintes d'ordonnancement dans le CRFG après l'affectation des registres sont de trois types : lecture après écriture (*Read after Write*), écriture après lecture (*Write after Read*) et écriture après écriture (*Write after Write*).

6.3.1 La dépendance fonctionnelle : *lecture après écriture*

La lecture d'une valeur dans un registre n'est possible qu'aux cycles suivant l'écriture de cette valeur. Dans la figure 6.7, il existe un arc *RaW* (*Read after Write*) entre les opérations fonctionnelles *(a)* et *(b)*. La nouvelle valeur écrite dans *R1* lors de l'opération *(a)* n'est lisible qu'au cycle suivant.

L'arc *RaW* symbolise au minimum une frontière de cycle dans le CRFG. L'opération *(b)* est ordonnancée, au plus tôt, au cycle suivant celui de l'opération *(a)*.

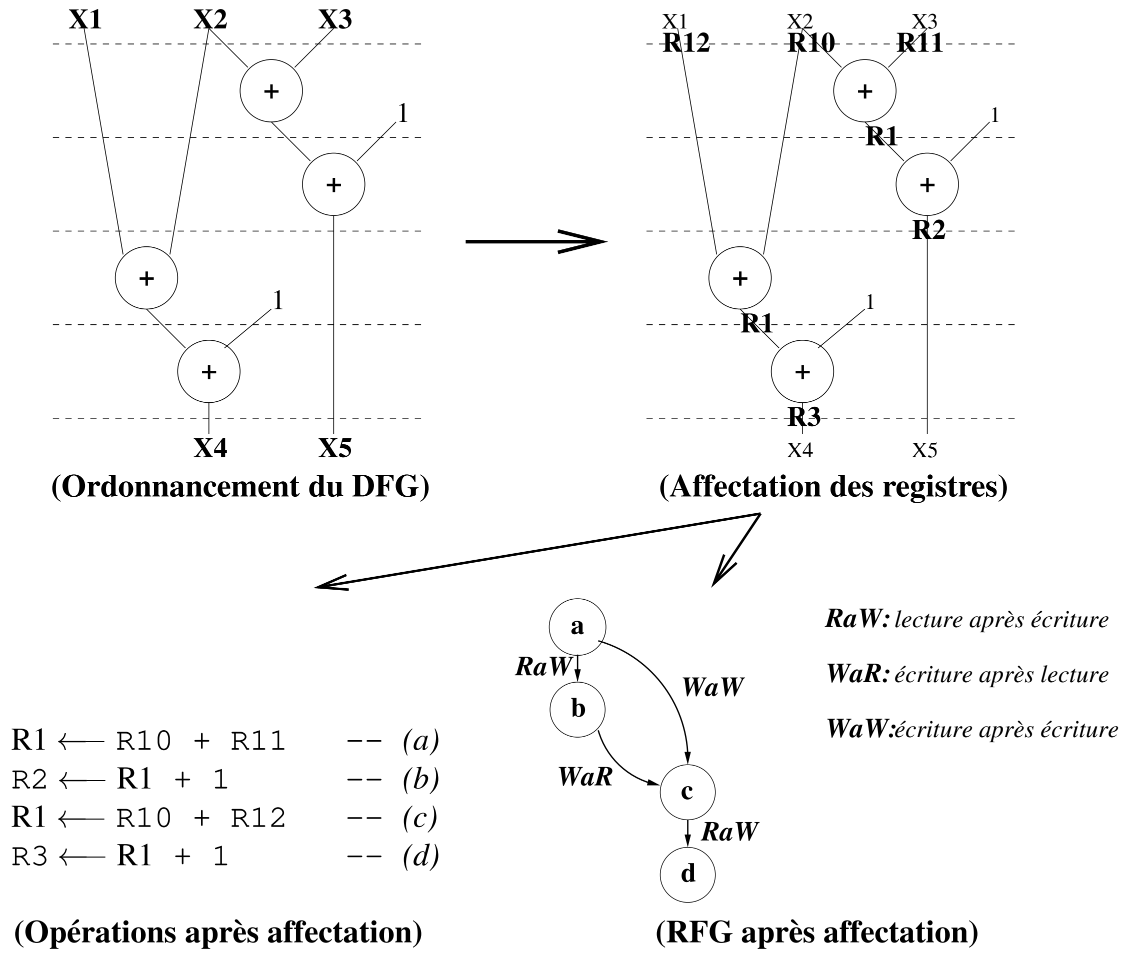


FIG. 6.7 L'apparition des contraintes d'ordonnancement.

6.3.2 La dépendance matérielle : *écriture après lecture*

Le registre ne peut contenir qu'une seule valeur. Toute écriture écrase l'ancienne valeur.
Pour garder la cohérence de la description initiale dans l'exemple de la figure 6.7, on ne peut pas ordonnancer l'opération *(c)* avant l'opération *(b)*. La valeur contenue dans *R1* serait alors remplacée avant d'avoir pu être lue par l'opération *(b)*.

L'arc *WaR* (*Write after Read*) symbolise au minimum un délai infinitésimal en comparaison de la période d'horloge. L'opération *(c)* est ordonnancée, au plus tôt, dans le même cycle que l'opération *(b)*.
S'il n'y a pas de chaîne courte (cf. chapitre 4.1.2), les temps de propagation permettent de faire une lecture et une écriture sur un registre dans le même cycle.

6.3.3 La dépendance matérielle : *écriture après écriture*

Deux écritures simultanées (dans le même cycle) sur le même registre entraînent des conflits électriques.

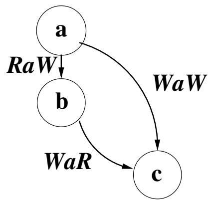
Dans la figure 6.7, les opérations *(a)* et *(c)* ne peuvent être effectuées dans le même cycle car elles écrivent dans le même registre *R1*.

L'arc *WaW* (*Write after Write*) symbolise au minimum une frontière de cycle dans le CRFG. L'opération *(c)* est ordonnancée, au plus tôt, au cycle suivant celui de l'opération *(a)*.

6.3.4 L'élimination de la dépendance *écriture après écriture*

R1 ⟵ `R10 + R11` -- *(a)*
R2 ⟵ `R1 + 1` -- *(b)*
R1 ⟵ `R10 + R12` -- *(c)*

(Opérations)



RaW: *lecture après écriture*
WaR: *écriture après lecture*
WaW:*écriture après écriture*

(RFG)

FIG. 6.8 La redondance du *WaW*.

Dans le RFG de la figure 6.8, nous apercevons que l'*arc (a,c)* de type *WaW* est superflu. En effet, l'*arc (b,c)* de type *WaR* constitue une contrainte plus forte.

Nous montrons que les contraintes *WaW* sont toujours superflues :
Soient deux *noeuds a et c* d'un CRFG possédant un *arc (a,c)* de type *WaW* allant de *a* vers *c* (i.e ces noeuds écrivent sur le même registre).

- S'il existe un *noeud b* du CRFG tel que l'opération *b* utilise la valeur du registre destination de *a*,
  - alors il existe un *arc (a,b)* de type *RaW* puisqu'il y a une dépendance de données entre *a* et *b*.
  - alors il existe un *arc (b,c)* de type *WaR* puisque la valeur doit être lue avant d'être écrasée par l'opération *c*.

  Alors il existe un chemin orienté allant de *a* vers *c* passant par les *arcs (a,b) et (b,c)*. Il doit y avoir au minimum un cycle entre l'opération *a* et l'opération *c*.
  L'*arc (a,c)* de type *WaW* n'ajoute pas de contraintes supplémentaires, il peut être supprimé du CRFG.
- S'il n'existe aucun *noeud b* du CRFG tel que l'opération *b* utilise la valeur du registre destination de *a*,
  alors le *noeud a* peut être supprimé du CRFG sans affecter la fonctionnalité.
  Il ne peut pas y avoir d'*arc (a,c)* de type *WaW*.

Nous pouvons supprimer toutes les contraintes *écriture après écriture* car la cohérence est garantie par les contraintes *écriture après lecture* et *lecture après écriture*.

➠ L'*ordonnancement* des opérations d'un CRFG est possible.

➠ Cet *ordonnancement* obéit à des contraintes de dépendances :

- *lecture après écriture* appelée contrainte de type *RaW*.
- *écriture après lecture* appelée contrainte de type *WaR*.

6.4 L'ordonnancement dans la synthèse ciblée

La synthèse ciblée nécessite l'*affectation des registres* et le *nombre d'opérateurs combinatoires*. Elle se base donc sur un CRFG.
L'algorithme de synthèse est simplifié : la phase d'allocation se réduit à l'affectation des opérateurs combinatoires (cf. figure 6.9).

6.4.1 La linéarité de l'algorithme

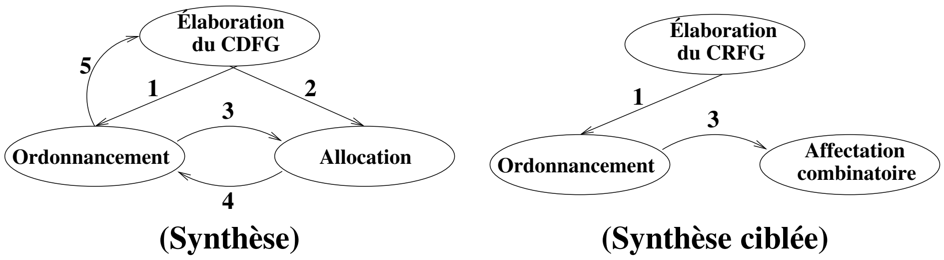


FIG. 6.9 Séquence de la synthèse ciblée de haut niveau.

Les dépendances de la figure 6.9 sont à comparer avec celles de la synthèse dans la figure 6.2 :

Arc 1 Les choix d'optimisation du CRFG influencent le nombre de cycles.

Arc 2 Cette dépendance a disparu car le choix du CRFG ne peut plus influencer le nombre de ressources matérielles.

Arc 3 A chaque cycle, on détermine quels opérateurs matériels combinatoires vont être utilisés.

Arc 4 Cette dépendance a disparu car l'ordonnancement est indépendant des temps de propagation des opérateurs matériels. Dans le chapitre 4, nous reportons la contrainte de fréquence à une phase de *réordonnancement* postérieure.

Arc 5 Cette dépendance a disparu car l'optimisation des boucles en surface ou en latence est contrôlée par le concepteur. Les branchements sont câblés en fonction du nombre de ressources matérielles.

La synthèse de haut niveau est devenue une succession linéaire de phases comme la compilation logicielle : l'*élaboration du CRFG* puis l'*ordonnancement* et enfin l'*affectation du matériel combinatoire*.

6.4.2 L'ordonnancement avec l'affectation des registres

L'*ordonnancement* représenté dans le schéma de la *synthèse ciblée* de la figure 6.9 s'effectue sur tous les RFG du CRFG.

Avec l'*affectation des registres* déjà déterminée, les opérations fonctionnelles de la description comportementale se ramènent donc à des lectures et des écritures dans des registres matériels. Les opérations sont des instructions de transfert de registres telles que le montre la figure 6.7.
Les noeuds du RFG correspondent aux opérations de la description comportementale et les registres sont les variables de la description.

Les dépendances du RFG sont la *lecture après écriture* (*RaW*) et l'*écriture après lecture* (*WaR*).

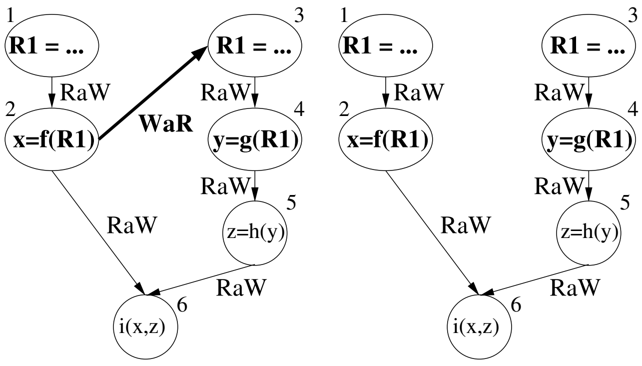
Objectif de l'ordonnancement

Nous avons estimé au chapitre 5 que donner l'affectation des registres et le nombre d'opérateurs combinatoires était suffisant pour cibler une solution en surface. Les variations de surface dues aux multiplexeurs sont des changements acceptables pour le concepteur.
La surface n'étant plus le critère prépondérant, l'optimisation en nombre de cycles doit être maximale.

L'objectif de l'ordonnancement est d'obtenir le minimum de nombre de cycles en utilisant les ressources données par l'utilisateur. L'algorithme privilégie donc toujours l'ordonnancement qui donne le temps d'exécution global le plus court. La solution la plus rapide en temps d'exécution est toujours choisie même si une autre solution bénéficie d'une surface moindre.
C'est donc un algorithme de type *list scheduling* [PG87] qui a été utilisé car il permet l'optimisation du nombre de cycles sous contraintes de ressources.

Assouplissement de la contrainte *écriture après lecture*

```
R1 = ...
x  = f( R1 )
R1 = ...
y  = g( R1 )
z  = h( y )
out= i( x, z )
```



(séquence d'instructions) **(RFG avec WaR)** **(RFG sans WaR)**

FIG. 6.10 Allègement des dépendances.

Dans l'ordonnancement classique d'un RFG, les contraintes d'écritures (*WaR*) et de lectures (*RaW*) sont positionnées dans le graphe.
Nous proposons de faire l'ordonnancement d'un RFG ne contenant que les contraintes de *lecture*

après écriture (*RaW*). Les contraintes d'*écriture après lecture* (*WaR*) sont positionnées au fur et à mesure de l'ordonnancement.
Les choix d'ordonnancement sont plus nombreux car les contraintes de précédence sont moins fortes.

Algorithme d'ordonnancement

L'algorithme, présenté ci-après, ordonnance les noeuds d'un RFG en respectant les contraintes de dépendance.
Pour chaque cycle, la fonction *C_mon_Choix()* choisit le meilleur noeud parmi les noeuds respectant les contraintes de précédence. Le critère de sélection est de favoriser le noeud le moins mobile dans le RFG ou celui qui a plus de successeurs.

Puis, le noeud sélectionné est ordonnancé dans le cycle courant. Des nouvelles contraintes de dépendance (de type *WaR*) issues de l'ordonnancement du noeud sont ajoutées au RFG.
Lors de la sélection de l'écrivain d'un registre *R1*, tous les lecteurs dépendants de la donnée écrite dans *R1* sont prioritaires sur les autres écrivains du registre *R1* (ajouts d'arcs *WaR*).
Ainsi, les données ne peuvent pas être écrasées avant d'être lues.

```
// construction de l'ordonnancement pour tous les cycles
Soient les paramètres,
       𝒩: l'ensemble des noeuds du CRFG
       ℛ𝒜𝒲: l'ensemble des arcs de type RaW (dépendance de données)
et le résultat:
       𝒮: l'ensemble ordonné des noeuds ordonnancés

Ordonnancement( 𝒩, ℛ𝒜𝒲 )
{

Soient 𝒮_c: l'ensemble des noeuds ordonnancés au cycle c
       𝒲𝒜ℛ: l'ensemble des arcs de type WaR (dépendance matérielle)
       cycle: le cycle courant de l'ordonnancement
       choix: le noeud courant de l'ordonnancement
       noeud: un noeud quelconque
       successeur: un noeud successeur
       ecrivain: un noeud écrivant dans un registre
       arc_(n1,n2): un arc allant du noeud n1 vers le noeud n2

       cycle = 0

       tant que 𝒩 ≠ ∅
       {
          // choisir le meilleur noeud parmi les noeuds ne créant pas de conflit
          choix = C_mon_Choix( { noeud ∈ 𝒩 / ∀ ecrivain ∈ (𝒮_cycle ∪ 𝒩) ,
                                 arc(ecrivain,noeud) ∉ (ℛ𝒜𝒲 ∪ 𝒲𝒜ℛ) } )

          // Est-ce qu'un noeud a été choisi pour le cycle?
          si choix ≠ {∅} alors
          {
            // ajouter le noeud courant à la solution
            𝒮_cycle = 𝒮_cycle ∪ {choix}
            𝒩 = 𝒩 \ {choix}

            // ajouter les contraintes WaR
            pour tout ecrivain ∈ 𝒩 / (ecrivain possédant le même registre
                                      destination que le noeud choix)
            {
              pour tout successeur ∈ 𝒩 / (successeur ≠ ecrivain et arc_(choix,successeur) ∈ ℛ𝒜𝒲)
              {
                𝒲𝒜ℛ = 𝒲𝒜ℛ ∪ {arc_(successeur,ecrivain)}
              }
            }

          }
          sinon // pas de noeud éligible pour ce cycle
          {
            cycle = cycle + 1
          }


       }

retourne 𝒮
}
```

Application de l'algorithme

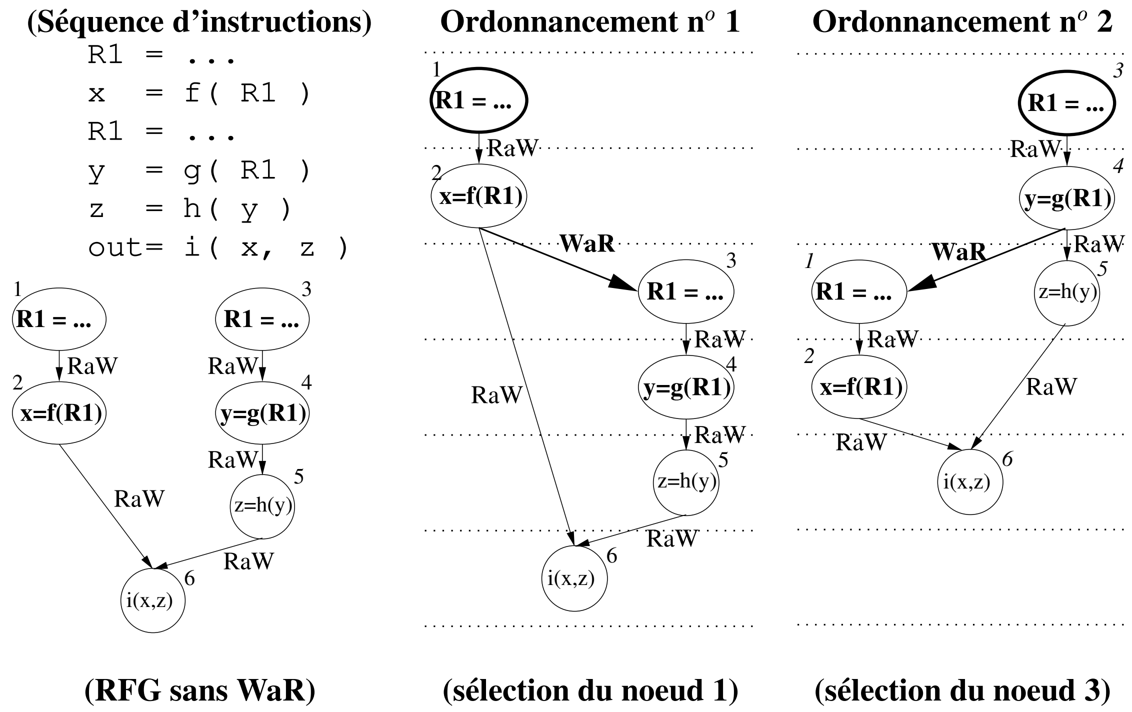


FIG. 6.11 Deux ordonnancements sont possibles grâce à l'absence de *WaR* dans le RFG.

Dans l'exemple de la figure 6.11, l'algorithme d'ordonnancement débute sur un RFG sans contrainte de type *WaR*. La fonction *C_mon_Choix()* sélectionne un noeud dans le RFG :

1. Si le *noeud 1* est sélectionné (*ordonnancement n° 1*), l'*arc (2,3)* de type *WaR* est positionné dans le RFG pour empêcher la sélection du *noeud 3* avant le *noeud 2*.

2. Si le *noeud 3* est sélectionné (*ordonnancement n° 2*), l'*arc (4,1)* de type *WaR* est positionné dans le RFG pour empêcher la sélection du *noeud 1* avant le *noeud 4*.

Il y a donc deux ordonnancements de possible avec notre algorithme.
Dans un algorithme classique, le RFG contient dès le départ les contraintes *WaR*, l'*arc (2,3)* serait positionné avant toute sélection de noeud. Seul l'*ordonnancement n°1* est alors possible.

L'avantage de positionner les contraintes *WaR* à chaque sélection de noeud est de disposer d'un plus grands choix de noeuds. Ainsi on peut faire apparaître un ordonnancement optimisé.
Dans l'exemple de la figure 6.11, l'*ordonnancement n°2* fait gagner un cycle.

La détermination de l'ordonnancement optimal se fait en sélectionnant systématiquement le noeud le moins mobile.
Le *noeud 1* possède une mobilité de 1 dans le RFG, alors que le *noeud 3* possède une mobilité nulle. C'est donc l'*ordonnancement n°2* qui sera préféré.

6.4.3 L'affectation des opérateurs combinatoires

L'*ordonnancement* vise à minimiser le temps d'exécution global. La phase d'*affectation des opérateurs combinatoires* a pour objectif de minimiser la surface.

Objectifs de l'affectation

La phase d'*ordonnancement* recherche la plus grande parallélisation des opérations fonctionnelles pour chaque cycle pour diminuer le temps d'exécution. La parallélisation est bornée par le nombre d'opérateurs donné par l'utilisateur.

Dans la synthèse ciblée, le nombre d'opérateurs et leurs types sont déjà déterminés par l'utilisateur. La phase d'*affectation des opérateurs combinatoires* de la figure 6.9 correspond à la minimisation du nombre d'entrées des multiplexeurs et de la taille des opérateurs arithmétiques. C'est les seules caractéristiques qui permettent de diminuer la surface.

Minimiser la taille des multiplexeurs et des opérateurs a aussi pour intérêt de diminuer les temps de propagation de ces opérateurs. Réduire les temps de propagation locaux peut diminuer la chaîne longue globale du circuit. Cela facilite l'étape de *réordonnancement* qui vise à respecter la fréquence d'horloge donnée par l'utilisateur en insérant des cycles d'attente (cf. chapitre 4).
Le nombre de cycles d'attentes pour la propagation des signaux insérés est donc moins important si les temps de propagation sont plus courts. Nous rejoignons donc l'objectif de l'*ordonnancement* qui est de minimiser le nombre de cycles.

Les deux objectifs de l'*affectation* sont donc la minimisation de la surface et du temps d'exécution global.

La minimisation des multiplexeurs

Si l'on suppose que devant chaque entrée d'opérateur combinatoire, il y a un multiplexeur, l'optimisation de la phase d'*affectation des opérateurs combinatoires* se résume à minimiser la taille des multiplexeurs. Chaque entrée est connectée à un multiplexeur virtuel possédant au moins une entrée.

La phase d'*affectation des opérateurs combinatoires* choisit pour chaque cycle l'affectation qui minimise donc la taille des multiplexeurs.
La fonction de coût d'un multiplexeur est la suivante : $nombre\_entrees * nombre\_bits$. Ces deux paramètres augmentent la surface du multiplexeur. Nous avons juste besoin d'une fonction "naïve" car nous ne mesurons que des différences entre des solutions.
Le coût d'une solution est la somme des coûts des multiplexeurs qu'elle instancie : $\Sigma_m^{chemin\_donnees} nombre\_entrees_m * nombre\_bits_m$.

[CG96] a montré que le simple fait d'intervertir les sources des opérateurs combinatoires permettait de diminuer de 30 % le nombre d'entrées des multiplexeurs.
Pour minimiser les multiplexeurs, nous procédons donc à la permutation des opérateurs combinatoires et de leurs entrées. Ces permutations sont effectuées dans la mesure où elles sont autorisées par le concepteur. Il faut que ses directives architecturales le permettent (cf. chapitre 5.2.2).

Chaque solution est évaluée et est conservée si elle est meilleure.

L'algorithme de permutation est de type glouton, il peut tomber dans un minimum local.
Cependant, nous tentons de nous approcher de la solution optimale au moyen d'une boucle à relaxation. C'est à dire que l'algorithme réessaie toutes les permutations possibles à chaque fois qu'une permutation donne une solution meilleure.

- ✓ L'algorithme d'ordonnancement de la synthèse ciblée est linéaire. Les phases de la synthèse sont rendues indépendantes grâce aux contraintes architecturales du concepteur.
- ✓ L'ordonnancement de la synthèse ciblée est effectué sur des RFG.
- ✓ Les contraintes d'*écriture après lecture* (*WaR*) sont évaluées lors de l'ordonnancement. L'allégement des contraintes de précédence permet d'obtenir un meilleur ordonnancement.
- ✓ L'objectif de la phase d'ordonnancement est la minimisation du temps d'exécution.
- ✓ Les objectifs de la phase d'*affectation des opérateurs combinatoires* sont la minimisation de la surface et la minimisation du temps d'exécution. Le nombre de bits et le nombre d'entrées des multiplexeurs sont les paramètres d'optimisation.

6.5 Les points clefs de l'algorithme d'ordonnancement

Nous avons vu dans le chapitre précédent que l'algorithme d'ordonnancement ajoutait des arcs de dépendances de type *WaR* dans le RFG.
L'ajout de ces arcs garantissent que la valeur enregistrée dans un registre ne sera pas écrasée avant d'être lue.

L'ordonnancement se fait sur le RFG noeud après noeud. Une étape d'ordonnancement correspond à la sélection d'un noeud, l'ajout de nouvelles contraintes dans le graphe et la suppression du noeud ainsi que des arcs qui lui sont attachés.
Un noeud est sélectionnable si et seulement s'il ne possède plus aucun arc entrant.
Ces arcs *WaR* protègent de l'écrasement de la valeur par une autre écriture (cf. chapitre 6.3.2). Ils garantissent donc le respect des dépendance de données.

6.5.1 Apparition d'interblocages

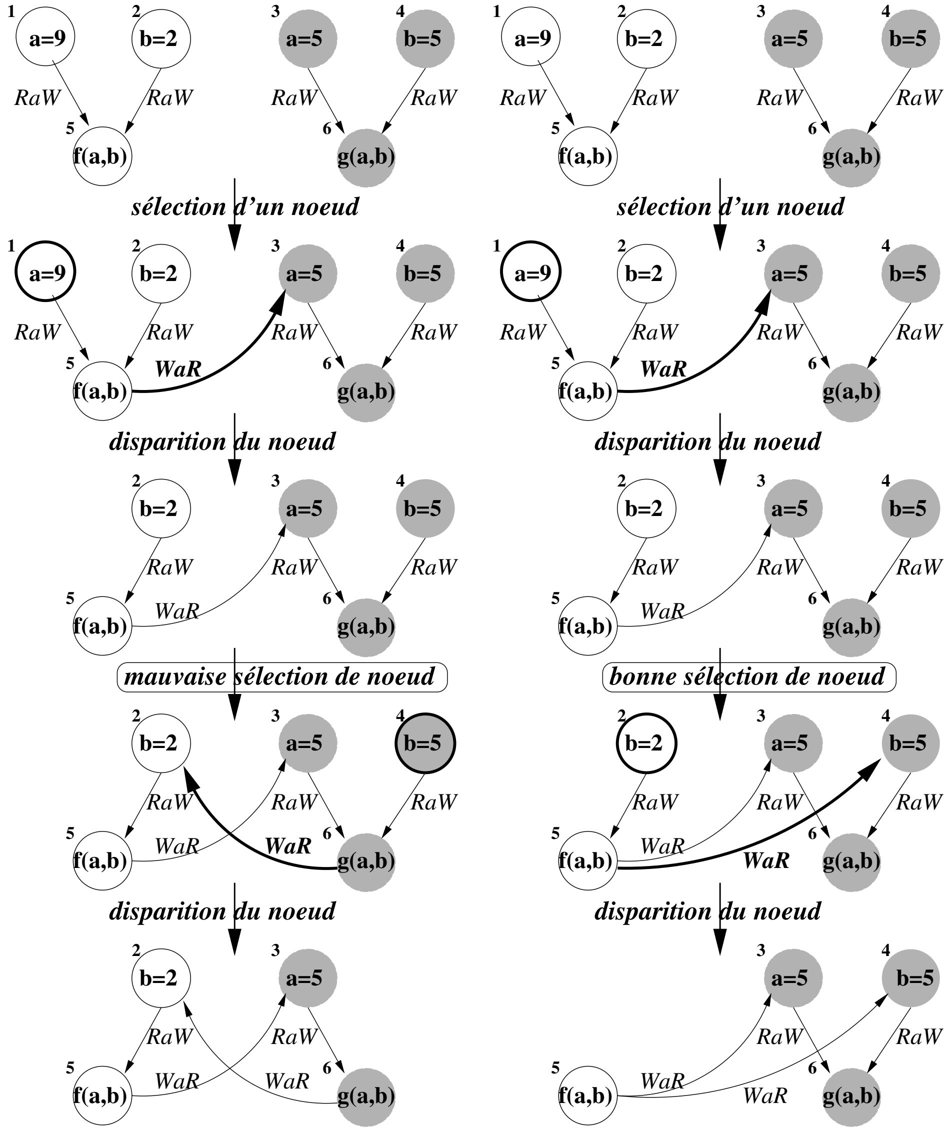
Ajouter des arcs dans le RFG à chaque étape de l'ordonnancement peut entraîner des cycles dans le graphe. Ce cycle traduit un conflit entre une lecture et une écriture. L'ordonnancement est incompatible avec l'*affectation des registres*.

Premier exemple d'interblocage

L'exemple de la figure 6.12, nous montre la possibilité d'obtenir un interblocage.

Le *noeud 1* est sélectionné pour être ordonnancé en premier. Cela crée un arc du *noeud 5* au *noeud 3* pour protéger la lecture du registre *a*. Le *noeud 1* disparaît du RFG pour ne pas être de nouveau

```
a = 9
b = 2
f( a, b )
...
...
...
a = 5
b = 5
g( a, b )
```



(séquence d'instructions) **(mauvais enchaînement)** **(bon enchaînement)**

FIG. 6.12 Possibilité d'apparition d'un interblocage.

ordonnancé.
Le *noeud 4* est sélectionné. C'est un mauvais choix car l'apparition de l'*arc (6,2)* (i.e du *noeud 6* au *noeud 2*) crée un cycle passant par les *noeuds 2,5,3,6*.

Il y avait pourtant une alternative ne créant pas de cycle : c'est la sélection du *noeud 2* à la place du *noeud 4*. La sélection de ce noeud aurait créé un *arc (5,4)* pour protéger la lecture du registre *b*.
Le *noeud 5* n'ayant plus de prédécesseurs serait à son tour sélectionné Sa sélection ne crée pas d'arc supplémentaire car il n'y a pas de lecture à protéger. Les ordonnancements suivants ne créeront pas non plus d'arcs, il n'y a donc plus d'apparition de cycle possible dans le graphe.

L'ordonnancement *noeud 1*, *noeud 2*, *noeud 5*, *noeud 3*, *noeud 4*, *noeud 6* est correct car il ne crée pas de cycle.

Les ordonnancements commençant par la sélection du *noeud 1* puis du *noeud 4* ou du *noeud 4* puis du *noeud 1* créent un cycle. Ce ne sont pas les seuls, les ordonnancements commençant par la sélection des *noeuds 2 et 3* créent aussi un cycle.
Il y a donc des contraintes non exprimées dans le graphe. Ces contraintes devraient apparaître après la sélection du premier noeud de ce graphe. Par exemple, il devrait y avoir une interdiction de sélection du *noeud 4*, si le *noeud 1* a été sélectionné en premier et ce tant que le *noeud 5* n'a pas été sélectionné.

Les contraintes devraient empêcher la sélection du noeud formant le cycle. Seuls des arcs supplémentaires peuvent expliciter ces contraintes. Dans notre exemple, si un *arc (5,4)* s'était formé après la sélection du *noeud 1*, il aurait empêché la sélection du *noeud 4* et la formation d'un cycle.

Deuxième exemple d'interblocage

Dans l'exemple de la figure 6.13, la seule sélection du *noeud 2* en premier provoque inévitablement un cycle.

La sélection du *noeud 2* crée l'*arc (5,4)*.
La sélection du *noeud 1* crée l'*arc (10,6)*.
La sélection du *noeud 3* crée l'*arc (8,7)*.
Le cycle est créé entre les *noeuds 10,6,8,7,9*.

Ce cycle est inévitable dès la sélection du *noeud 2* en premier. Les *noeuds 1 et 3* sont les seuls noeuds libres (i.e sans prédécesseur). Ils doivent tous les deux être ordonnancés pour que l'on puisse libérer d'autres noeuds.

Il doit y a voir une contrainte non exprimée dans le graphe. Elle doit empêcher la sélection du *noeud 2* avant le *noeud 4* qui permet de libérer les ressources.

Résumé du problème

Un interblocage correspond à l'apparition d'un cycle dans le RFG. Un interblocage apparaît lorsqu'un registre *b* possède une valeur dont la lecture est conditionnée par l'écriture d'une autre valeur sur ce registre *b* (cf. le cycle de la figure 6.14 issu de l'exemple de la figure 6.12). Le registre ne pouvant contenir deux valeurs à la fois, un interblocage apparaît.

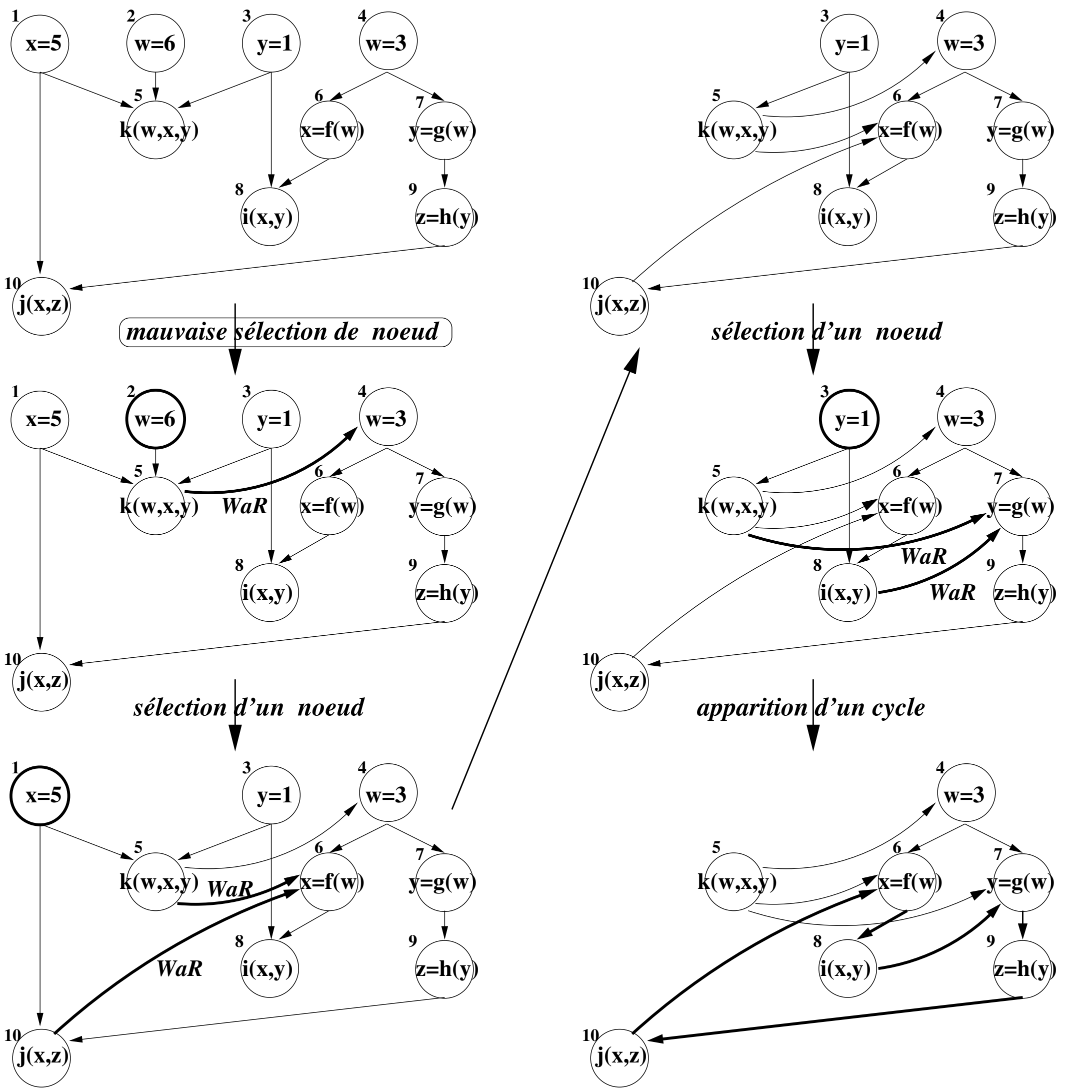
Le RFG est un graphe acyclique comportant des **chemins orientés** (cf. figure 6.14). Un *chemin orienté* allant d'un noeud A (*noeud 2* dans la figure) à un noeud B (*noeud 6*) est un parcours du RFG du noeud A au noeud B circulant de noeud en noeud en passant par les arcs et en respectant leur sens. Un arc est donc, à lui seul, un chemin orienté.

Il y a fondamentalement deux types de contraintes à mettre en évidence pour ne pas créer de cycle dans le RFG lors de l'ordonnancement. Ils correspondent respectivement aux deux exemples d'interblocages que nous avons donnés plus haut :

1. Le noeud sélectionné dans la figure 6.12 (*noeud 4*) crée immédiatement un cycle à cause des nouveaux arcs qui apparaissent.
2. Le noeud sélectionné dans la figure 6.13 (*noeud 2*) ne crée pas immédiatement un cycle.
   L'arc *WaR* que cette sélection crée ne participe pas à la construction d'un cycle. Mais il bloque

```
w = 3
x = f( w )
y = g( w )
z = h( y )
y = 1
i( x, y )
x = 5
j( x, z )
w = 6
k( w, x, y )
```

(séquence d’instructions)



(Sélections des noeuds)

FIG. 6.13 Mauvais ordonnancement entraînant un interblocage.

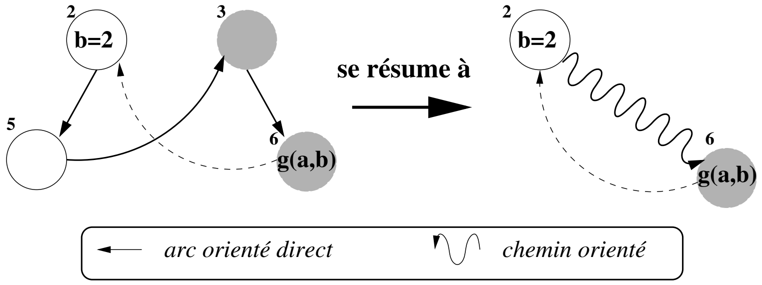


FIG. 6.14 La lecture du registre *b* dépend d'une autre écriture sur *b*.

une ressource (*noeud 4*) qui est cruciale dans le graphe. La sélection de ce noeud produit de façon inévitable un cycle lors des prochaines sélections.

Il faudrait identifier ces différents types de contraintes et les expliciter dans le RFG afin de ne jamais produire de cycle lors de l'ordonnancement.
Par la suite, nous distinguerons deux types de contraintes :

1. **les dépendances implicites sur l'utilisation d'un registre**.
2. **les dépendances implicites entre registres**.

6.5.2 Les dépendances implicites sur un registre

Nous allons étudier tous les cas possibles où la sélection d'un noeud dans un RFG peut entraîner immédiatement un cycle. Pour cela il faut une dépendance de données sur un registre *R* constituée d'un noeud écrivain (noeud à sélectionner) et d'un noeud lecteur. Il faut aussi un autre noeud écrivain sur le registre *R* fonctionnelement étranger à la dépendance de données pour matérialiser l'arc *WaR*.

Les différentes configurations de ces trois noeuds constituant un sous-graphe du RFG sont au nombre de six.
Elles sont présentées dans la figure 6.15. Le noeud écrivain est le *noeud 1*. Le noeud lecteur est le *noeud 2*. Le noeud écrivain sans dépendance de données est le *noeud 3*.
Les noeuds possédant une dépendance de donnée sont représentés avec la même couleur dans la figure. Cette dépendance de donnée est matérialisée par un arc de type *RaW* allant du *noeud 1* au *noeud 2*.

Les sous-graphes *(G1)* à *(G6)* de la figure 6.15 correspondent aux différentes configurations possibles lors de la sélection d'un *noeud 1* quelconque. Ils représentent tous les chemins possibles entre ce *noeud 1* et les autres noeuds accédant au même registre *R* :

chemin du noeud 1 au noeud 2 : Il existe déjà un chemin entre ces deux noeuds, il est constitué par l'arc de type *lecture après écriture* (*RaW*) issu de la dépendance de donnée entre ces deux noeuds.

chemin du noeud 2 au noeud 1 : Ce chemin ne peut pas exister car il constituerait un cycle avec l'*arc (1,2)* dans le RFG.

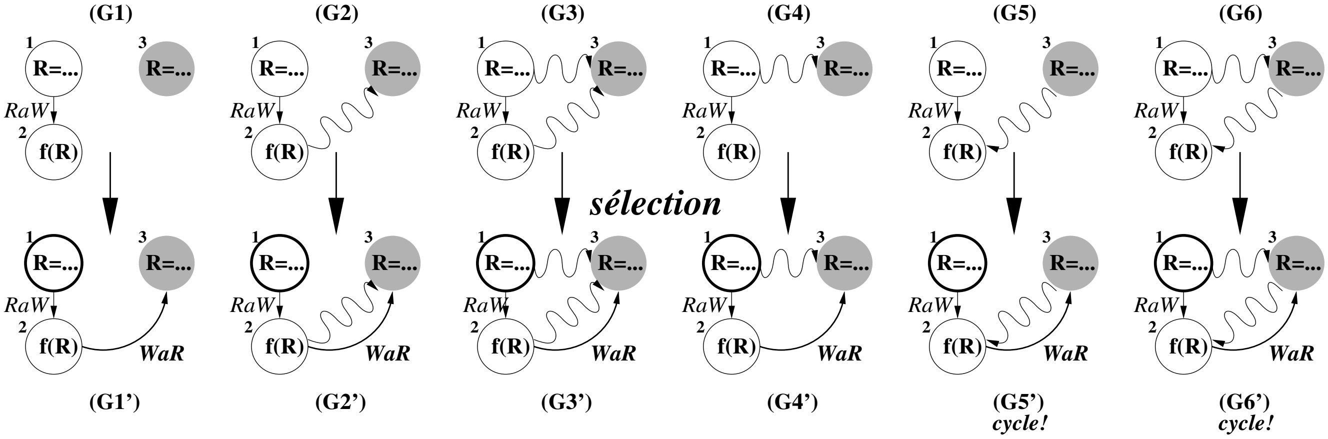


FIG. 6.15 La sélection du *noeud 1* peut se produire dans 6 configurations différentes.

chemin du noeud 3 au noeud 1 : Ce chemin ne peut pas exister car le *noeud 1* ne serait pas sélectionnable avant le *noeud 3*. Ce qui reviendrait à étudier les conséquences de la sélection du *noeud 3* à la place du *noeud 1*.

Tous les autres chemins sont possibles : il y a donc bien 6 possibilités.

La sélection du *noeud 1* dans les différents sous-graphes *(G1)* à *(G6)* de la figure 6.15 entraîne leur transformation en sous-graphes *(G1')* à *(G6')* respectivement.
Nous constatons dans les sous-graphes *(G1')* à *(G6')*, l'apparition de l'arc *WaR* (en **gras** dans la figure) du *noeud 2* vers le *noeud 3* et issu de la sélection du *noeud 1*.

- Dans *(G1')* et *(G4')*, un nouveau chemin est créé allant du *noeud 2* au *noeud 3*.
- Dans *(G2')* et *(G3')*, aucun nouveau chemin est créé.
- Dans *(G5')* et *(G6')*, un cycle est apparu !
  Dans *(G5)*, le *noeud 1* n'aurait pas du être sélectionné. Dans *(G6)*, il n'y a pas d'autre solution possible, c'est donc une impasse. Le sous-graphe *(G6)* ne devrait jamais apparaître dans le RFG.

Explicitation des dépendances implicites

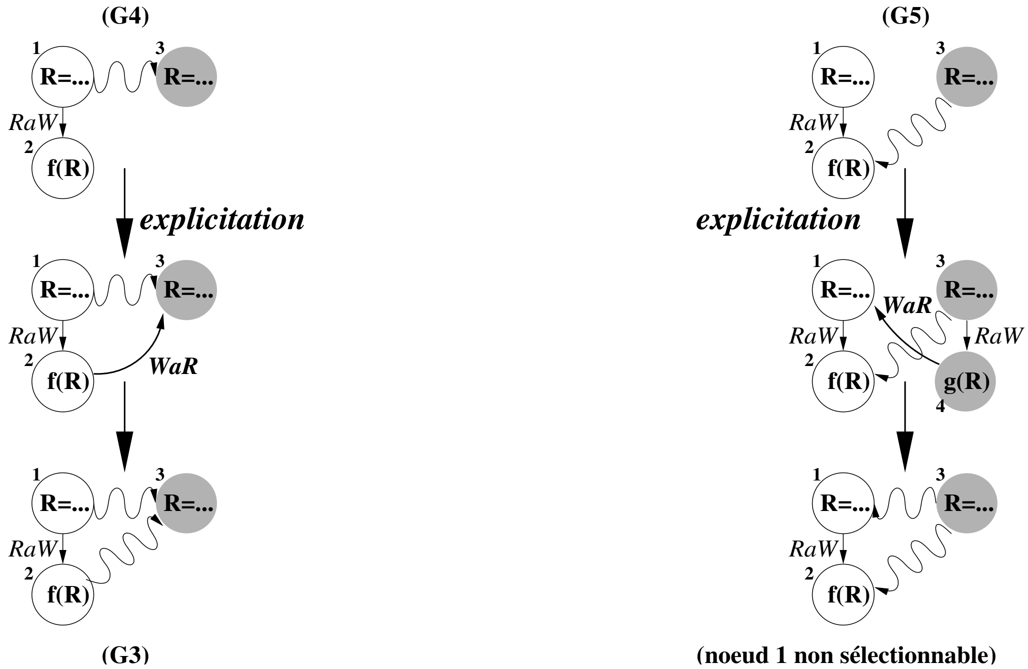


FIG. 6.16 Explicitation des dépendances implicites.

⇒ Si l'on observe l'apparition du chemin dans *(G4')* de la figure 6.15 allant du *noeud 2* au *noeud 3*. L'apparition de ce chemin est inévitable quel que soit l'ordre de la sélection des noeuds car le *noeud 3* ne peut être sélectionné avant le *noeud 1*.
Dans la figure 6.16, nous faisons apparaître ce chemin avant même la sélection du *noeud 1*, en créant l'*arc WaR* du *noeud 2* au *noeud 3*. Le sous-graphe *(G4)* se transforme donc en sous-graphe *(G3)* (cf. figure 6.15). L'ajout de ce chemin n'est pas une limitation du choix de l'ordonnancement, il met juste en évidence la contrainte implicite sur *(G4)*.

⇒ Si l'on observe l'apparition du cycle en *(G5')* de la figure 6.15, nous nous apercevons que le *noeud 1* n'aurait pas du être sélectionné avant le *noeud 3*.
Dans la figure 6.16, nous faisons apparaître un chemin allant du *noeud 3* vers le *noeud 1*, en créant l'*arc WaR* du *noeud 4* au *noeud 1*. Le chemin est créé par l'ajout d'un arc *WaR* allant du noeud constituant la dépendance de donnée du *noeud 3* vers le *noeud 1*. La construction de ce chemin n'est pas une limitation du choix de l'ordonnancement, il met juste en évidence la contrainte implicite sur *(G5)*.

L'explicitation des dépendances implicites consistent à **transformer *(G4)* en *(G3)*** et à **supprimer *(G5)*** pour tous les noeuds du RFG. A moins de créer des cycles, le sous-graphe ***(G6)* ne peut plus apparaître** puisqu'il n'existe plus de *(G4)* ni de *(G5)* dans le RFG.

Algorithme d'explicitation

Pour chaque nouvel arc créé par la sélection d'un noeud :

1. Identifier tous les couples de noeuds *[i,j]* pour lesquels le nouvel arc a créé un nouveau chemin entre *i* et *j*.
2. Enlever les couples de noeuds qui ont déjà été examinés lors de l'*étape 3*.
3. Pour chaque couple de noeuds *[i,j]* :
   - Si *i* et *j* possède le même registre destination, alors créer un arc du noeud *i* vers le noeud *j* de type *WaR* (i.e transformation d'un *(G4)* en *(G3)*).
   - Si *i* possède un registre destination appartenant au support de *j*, alors créer des arcs de type *WaR* allant de tous les noeuds successeurs de *i* vers un noeud prédécesseur de *j* (i.e disparition du *(G5)*).
     Ces noeuds successeurs de *i* doivent avoir une dépendance de données avec le registre destination de *i*.
     Le noeud prédécesseur de *j* doit avoir le même registre destination que *i*.
4. Pour chaque nouvel arc créé par l'*étape 3*, recommencer à partir de l'*étape 1*.

Mise en application sur un exemple

Si nous reprenons l'exemple d'interblocage donné par la figure 6.12, nous arrivons à empêcher l'apparition d'un cycle grâce à l'explicitation de la contrainte entre le *noeud 2* et le *noeud 4*.
La figure 6.17 présente l'ordonnancement qui découle de la mise en évidence de cette contrainte. La sélection du *noeud 4* est empêchée par un arc *WaR* issu de l'explicitation des dépendances implicites. Ces dépendances implicites sont apparues après la sélection du *noeud 1*.

6.5.3 Les dépendances implicites entre registres

La propagation des dépendances

L'explicitation des dépendances se propage dans le RFG car elle crée de nouveaux chemins qui entraînent souvent de nouvelles dépendances à expliciter.

Ainsi l'apparition d'une contrainte sur le registre *a* se propage-t-elle sur le registre *b* dans l'exemple de la figure 6.18.
Le sous-graphe concernant le registre *a* est de type *(G4)*. Il est transformé en *(G3)* créant un nouveau chemin allant du *noeud 4* au *noeud 6*. Le sous-graphe concernant le registre *b* devient donc un *(G5)* qui doit être à son tour transformé pour ne pas tomber dans un cas de type*(G6)*.

Limitation de l'explicitation des dépendances

La méthode d'explicitation des dépendances est inefficace sur l'exemple de la figure 6.13. Dans cet exemple, il faut faire apparaître l'*arc (4,2)* pour empêcher la sélection du *noeud 2* avant le *noeud 4*

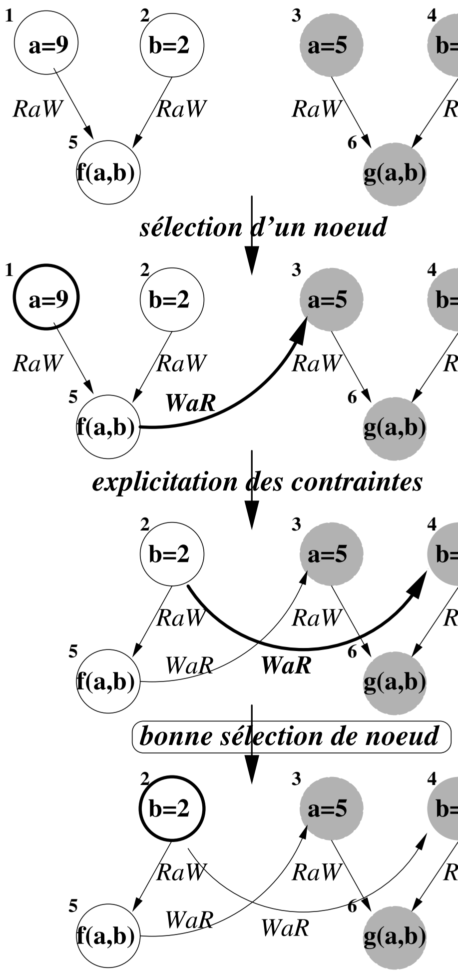


FIG. 6.17 Exemple d'explicitation des dépendances implicites.

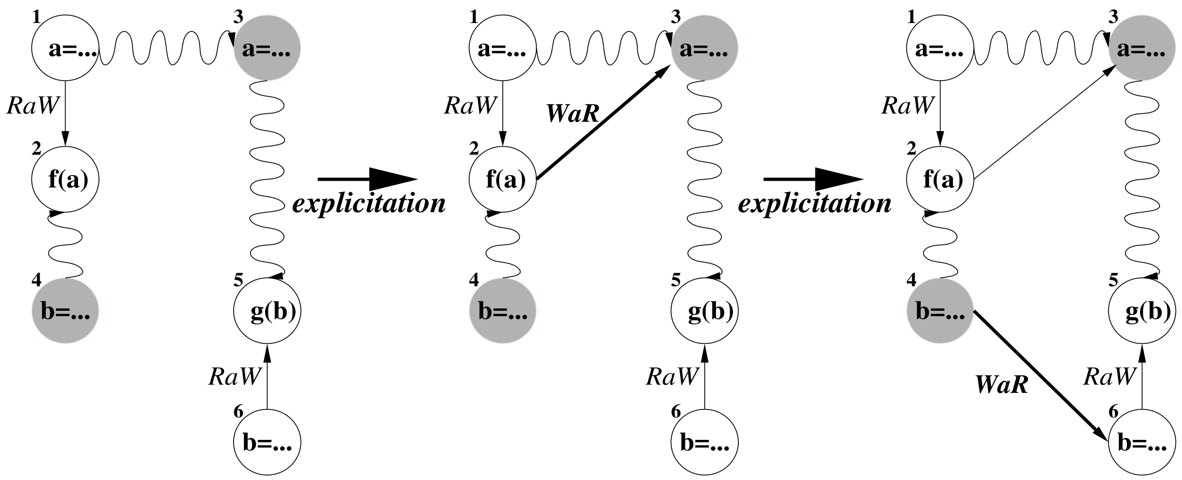


FIG. 6.18 Exemple de propagation de proche en proche.

qui mène à un cycle. Or la méthode d'explicitation des dépendances ne fait apparaître aucun arc dans le graphe initial.

Le *noeud 2* peut donc être sélectionné. L'explicitation des dépendances lors des sélections de noeud suivantes ne fait qu'apparaître le cycle plus tôt (cf. figure 6.19).
Une fois le *noeud 2* sélectionné et l'*arc (5,4)* construit, il existe un chemin entre le *noeud 3* et le *noeud 7* concernant la ressource *y*. La propagation des dépendances met en évidence l'*arc (8,9)*.
Il y a un nouveau chemin entre le *noeud 8* et le *noeud 10* concernant la ressource *x*. La propagation des dépendances met en évidence l'*arc (8,1)*.
Un cycle apparaît alors entre les *noeuds 1, 5, 4, 6, 8*. Il n'y a pas besoin de sélectionner les *noeuds 1 et 3* pour s'apercevoir que nous sommes dans un interblocage.

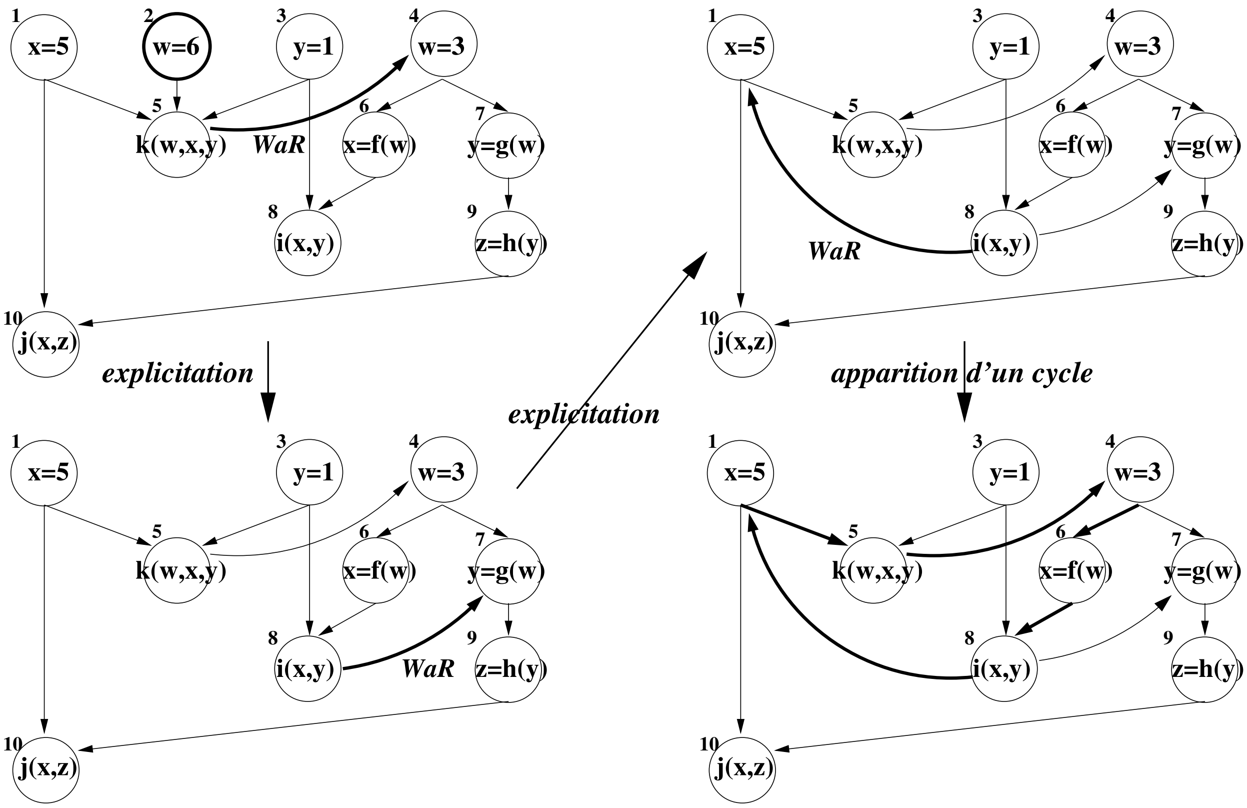


FIG. 6.19 Apparition du cycle malgré l'explicitation des dépendances pour chaque registre.

Le cycle qui est apparu par propagation des dépendances montre qu'il y a un interblocage entre les registres *w, x et y*. L'algorithme d'ordonnancement ne peut parvenir à une solution, une fois le *noeud 2* sélectionné.

La mise en évidence des dépendances implicites sur chaque registre et la propagation des contraintes de proche en proche n'évite pas les interblocages.
En effet, il peut y avoir des dépendances entre plusieurs registres, chacun dépendant de la valeur de l'autre.

Si la sélection d'un noeud *A* mène forcément à un interblocage, la propagation des contraintes permet de le révéler plus tôt grâce à la détection de cycle dans le RFG.
Mais rien ne garantit que l'interblocage soit détecté immédiatement après la sélection du noeud *A* !

Le cycle peut se former après plusieurs sélections postérieures à la sélection du noeud A.
Cependant plus le nombre de sélections postérieures au noeud A augmente, plus la probabilité de former le cycle augmente.

6.5.4 Mise en application dans l'algorithme d'ordonnancement

Il faut modifier l'algorithme pour éviter les interblocages. Il faut ajouter trois mécanismes :

- la mise en évidence des dépendances implicites sur chaque registre.
- la propagation des contraintes de proche en proche.
- la détection de cycle dans le RFG.
- la remise en cause de l'ordonnancement si un cycle apparaît.

Le coût de la remise en cause de l'ordonnancement peut sembler prohibitif en vitesse algorithmique. Cependant, seul un petit nombre de noeuds doivent être désélectionnés. Car la probabilité de tomber sur un cycle lors de l'exploration d'une branche des ordonnancements menant à interblocage augmente au fur et à mesure de l'exploration.
En pratique, la remise en cause de l'ordonnancement est très rare. La méthode d'explicitation et de propagation des contraintes est suffisamment efficace pour prévenir les interblocages dans les descriptions comportementales classiques.
La faible probabilité d'avoir un interblocage est du aussi au nombre restreint de chemins créés à chaque nouvelle sélection de noeud. En effet,la fonction *C_mon_Choix()* choisit en priorité ceux qui possèdent la plus grande hauteur dans le RFG (i.e ceux qui sont les moins mobiles).

> ✓ Des interblocages peuvent apparaître lors de l'*ordonnancement*. Ils sont dus à des dépendances implicites dans le RFG. Les contraintes matérielles qui ont été relâchées pour l'*ordonnancement* empêchaient ces interblocages.
>
> ✓ Tous les interblocages sont, par défaut, résolus par une remise en cause récursive des choix de l'ordonnanceur.
>
> ✓ Le risque d'interblocage est minimisé par une méthode de propagation des dépendances qui tente de maximiser les contraintes implicites.
> Les interblocages non détectés par la propagation sont peu probables car ils impliquent des relations complexes entre les noeuds.

6.6 Conclusion

L'algorithme de synthèse de haut niveau classique est très complexe, il ne permet pas d'obtenir un résultat de façon rapide :

- ➠ Les différentes phases de la synthèse de haut niveau sont très liées.
- ➠ L'algorithme ne permet pas de cibler une solution.

L'algorithme de la compilation logicielle est une simplification du problème de la synthèse :

- ➠ Les différentes phases de la compilation peuvent être dissociées.
- ➠ La phase d'allocation se limite à l'affectation des ressources et le CDFG est peu modifié.

Les hypothèses que nous avons posées dans les chapitres précédents, nous permettent de simplifier le problème de la synthèse :

- ➠ Les différentes phases de la synthèse ciblée s'enchaînent de façon linéaire.
- ➠ L'ordonnancement est dissocié de l'*affectation des registres*.
- ➠ L'ordonnancement se base sur un CRFG.

Le problème de la synthèse ciblé est réduit à un algorithme d'ordonnancement :

- ✓ La synthèse ciblée de haut niveau possède un algorithme d'*ordonnancement* linéaire et rapide.
- ✓ L'ordonnancement minimise le temps global d'exécution. La phase d'allocation minimise ensuite la taille des multiplexeurs.
- ✓ L'*ordonnancement* après l'*affectation des registres* peut remettre en cause certaines contraintes de précédence. Le risque d'interblocage est théoriquement possible si l'ordonnanceur fait un mauvais choix. Il doit alors changer son ordonnancement. Cependant, en pratique, les interblocages n'arrivent jamais sur une description comportementale.

Chapitre 7

Résultats et performances

Sommaire

Nous analysons dans ce chapitre les performances de l'outil *ugh*. Il s'agit d'évaluer les caractéristiques des circuits synthétisés par notre outil de synthèse de haut niveau par rapport aux autres outils existants. Mais aussi de montrer les variations des caractéristiques du circuit sur une même description comportementale rendues possibles par l'outil *ugh*. Nous verrons que l'outil offre plusieurs intégrations intéressantes suivant l'objectif du concepteur et le temps qu'il veut consacrer à la conception.

7.1 Les réalisations avec la synthèse ciblée

L'outil *ugh* nous a permis de réaliser différentes applications au niveau masques physiques. Elles sont assez représentatives de la complexité des applications que l'industrie cherche à concevoir.
Ces applications sont divisées en tâches indépendantes qui sont synthétisées séparément. Les différents types de tâches synthétisées couvrent la diversité des descriptions de modèle : flot de données, flot de contrôle et mixtes.

Les descriptions comportementales de ces tâches sont écrites en *C* [WM90]. Ces descriptions peuvent être compilées et exécutées de manière logicielle pour tester leur fonctionnalité.

7.1.1 Présentation des tests

Nous avons synthétisé des modules issus de quatre applications différentes.

Le VLD du Mpeg

Nous avons testé la synthèse ciblée sur le VLD (Variable Length Decoder) du Mpeg2, la couche vidéo de la norme de compression Mpeg. Le VLD est un module de décompression vidéo.
Le modèle comportemental utilisé pour le test est aussi capable de traiter du codage de type Mpeg1 (autre version du codage vidéo).
Le VLD prend en entrées des blocs de données et les décode en utilisant des tables de Huffmann. Ces tables contiennent des valeurs prédéfinies stockées dans des ROMs de 128 mots de 12 bits et 512 mots de 13 bits. Les données sont décompressées grâce à ces tables de correspondances. Les données sont décalées et concaténées en fonction des valeurs de ces tables.
Chaque pixel entrant codé au maximum sur 16 bits ressort en sortie du VLD sur 32 bits. Le passage des données dans les tables de Huffmann se fait sans perte de données.

Les éléments matériels indispensables au VLD sont un décaleur de 32 bits, plusieurs additionneurs/soustracteurs (8 de largeurs maximales de 16 bits) , les ROMs, des registres.
La complexité de la description comportementale est surtout composée d'une série de branchements et de comparaisons (chapitre A.1 montre la description comportementale du VLD). Cela implique soit un automate d'états complexes, soit un nombre de comparateurs importants pour câbler les branchements dans le chemin de données.

Pour tester le circuit généré, nous donnons des données issues du décodage réel d'une séquence vidéo. Les données traitées par le *VLD* sont constituées de 270.845 mots de 32 bits regroupés en 12.513 blocs

Mpeg. Pour donner un ordre d'idée, le traitement intégral de ces données par la tâche logicielle du VLD est effectué en environ 3 secondes sur un Pentium II cadencé à 200 MHz. La tâche logicielle est obtenue par la compilation de la description comportementale en C du VLD.

Le multi-Jpeg

Nous avons testé la décompression multi-Jpeg, norme pour le Jpeg animé. Le Jpeg animé est un flot constitué d'une série d'images fixes de types Jpeg.
On s'intéresse dans ce test à la globalité de la décompression. Les différentes tâches de l'application sont DEMUX, VLD, IQ, ZZ, IDCT, LIBU (cf. figure 7.1).
A l'exception du module DEMUX, les synthèses des autres modules ont été réalisées [Fau02].

- *DEMUX* (Demultiplexer) : sépare les informations et envoie les données aux différents modules.
- *VLD* (Variable Length Decoder) : première phase de décodage. C'est un décodage de Huffmann par bloc grâce une table dynamique reçue avec les données. Les valeurs de ces tables sont chargées au début du décodage de chaque image (le VLD du multi-Jpeg est donc différent de celui du Mpeg). Les tables du Huffmann sont stockées dans deux RAMs de 162 mots de 8 bits et deux RAMs de 14 mots de 8 bits.
- *IQ* (Inverse Quantification) : chargement d'une table dynamique (envoyée par le DEMUX) et multiplication des données reçue du VLD.
- *ZZ* (ZigZag) : inversion de l'ordre des données.
- *IDCT* (Inverse Discret Cosinus Transform) : La transformation consiste en deux multiplications et une série d'additions et de soustractions avec une table de cosinus. Les données entrantes codées sur 14 bits ressortent sur 8 bits de l'IDCT (codage d'un pixel).
- *LIBU* (Line Builder) : construit les lignes d'images à partir de la taille d'image envoyée par le DEMUX et des pixels envoyés par l'IDCT.

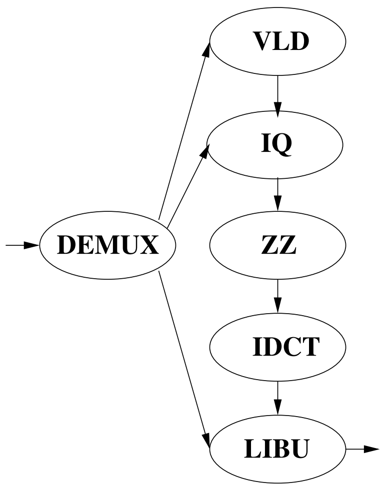


FIG. 7.1 Les tâches du décodage du multi-Jpeg.

Les éléments matériels indispensables de chaque tâche sont :

- *VLD* : des additionneurs/soustracteurs, 8 RAMs de 64 mots de 8 et 32 bits et des registres.
- *IQ* : un multiplieur de 32 bits, un incrémenteur de boucle et une RAM de 16 mots de 32 bits.
- *ZZ* : deux RAMs de 64 mots de 32 bits, un compteur de boucles et registre compteur.
- *IDCT* : deux multiplieurs et 8 additionneurs/soustracteurs 32 bits pour paralléliser les calculs. La table de cosinus est remplacée par des constantes propagées dans la description. Il faut aussi un banc de registres de 64 mots de 8 bits pour les données en entrée et un autre de 64 mots de 14 bits pour la sortie.
- *LIBU* : Une RAM de 64 mots de pixels "bruts" codés sur 8 bits. Une RAM de 512 mots de 8 bits représentant une ligne de l'image de 512 pixels pour effectuer l'ordonnancement des pixels. Pour cela il faut un multiplieur et des additionneurs 32 bits permettant de changer les index dans les RAMs.

Le contrôleur DMA

Le contrôleur DMA (Direct Memory Access) peut être utilisé en émission ou en réception des données. Il est constitué de 9 modules : DMAR, DMAW, TIMER, REG, PB, PX, TX, RX, WD (figure 7.2.
Seuls les trois derniers modules TX, RX et WD ont été synthétisés grâce à *ugh* [DT03]. Les autres ont été construits directement en VHDL au niveau RTL car ils étaient très simples à réaliser.

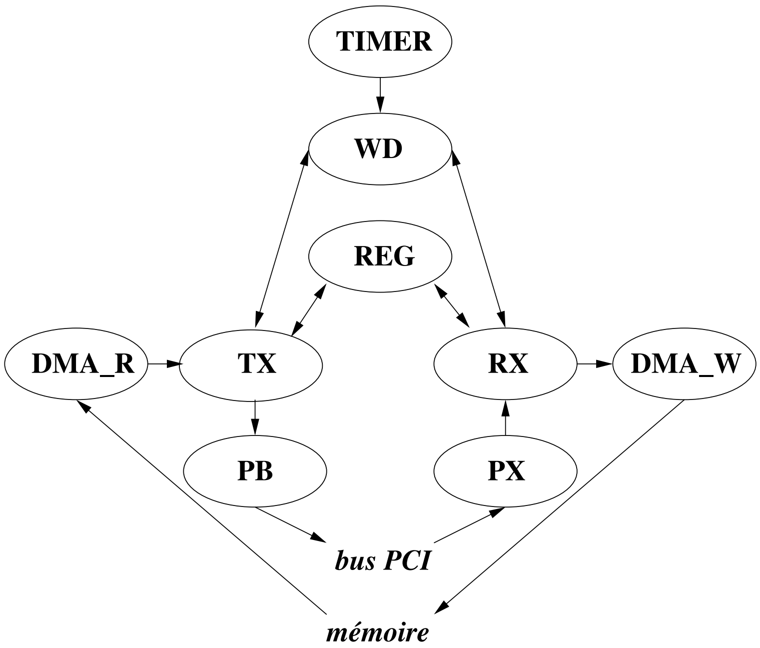


FIG. 7.2 Les tâches du contrôleur DMA.

1. *DMAR* (Direct Memory Access Read) : module de lecture à partir d'une zone mémoire de l'ordinateur.

2. *DMAW* (Direct Memory Access Write) : module d'écriture dans une zone mémoire de l'ordinateur.
3. *TIMER* : horloge de time out.
4. WD (Watch Dog) : gestion des drapeaux d'interruption et d'attente.
5. *REG* (Register) : registres de configuration du contrôleur DMA.
6. *TX* : automate d'envoi des données.
7. *RX* : automate de réception des données.
8. *PB* (Paquet Builder) : génération des paquets et insertion de bits de contrôle.
9. *PX* (Paquet eXtractor) : extraction des paquets et contrôle des erreurs.

Le contrôleur ainsi synthétisé est destiné à un FPGA pouvant représenter une vue structurelle d'au maximum 400.000 portes. Le contrôleur DMA synthétisé sur FPGA fonctionne à 33 MHz.

RX, TX et WD ne comportent pas d'opérateurs arithmétiques, ils font des branchements suivant les paquets de contrôle reçus.
RX et TX comportent en plus un banc registres de 16 mots de 32 bits et un compteur. Le banc de registres sert à compter le nombre de paquets reçus ou envoyés pour contrôler les pertes de paquets. Les 16 mots correspondent à 16 sources ou destinations différentes possibles.

Un flot de données

Le flot de données utilisé pour être synthétisé par *ugh* est un modèle uniquement construit dans le but de tester un flot de données sans branchement.

Il est constitué d'une série d'additions et de multiplications. Des registres servent de registres temporaires pour stocker les valeurs intermédiaires. Les accès aux ports se font uniquement au début et à la fin du traitement arithmétique.

Nous avons testé plusieurs versions de ce flot de données. Le nombre d'opérateurs varient suivant les versions du flot de données.

1. *FD 1* (flot de données n^{o}1) : 5 multiplieurs 32 bits, 5 additionneurs 32 bits.
2. *FD 2* (flot de données n^{o}2) : 8 multiplieurs 32 bits, 7 additionneurs 32 bits.
3. *FD 3* (flot de données n^{o}3) : 4 multiplieurs 16 bits, 3 multiplieurs de 12 bits, 7 additionneurs 8 bits.

7.1.2 Les méthodes de synthèse ciblée

La méthodologie de *ugh* consiste à donner une fréquence (cf. chapitre 4) à l'outil. Pour cela l'outil se scinde en deux phases appelées *ordonnancement* et *réordonnancement*. La phase d'*ordonnancement* construit un modèle RTL avec un automate d'états à partir de la description comportementale. La phase de *réordonnancement* modifie l'automate pour respecter la contrainte de fréquence.

Synthèse ciblée avec phase de réordonnancement

La première phase de la synthèse ciblée génère un chemin de données qui est ensuite synthétisé au niveau portes logiques (cf. figure 7.3). Elle génère aussi un automate d'états, cependant les états ne tiennent pas compte des temps de propagation du chemin de données.
Le chemin de données est synthétisé. Les temps de propagation sont extraits de la vue structurelle en portes de ce dernier. Le *réordonnanceur* adapte l'automate (en insérant des cycles) pour respecter la fréquence ciblée. Le nouvel automate d'états est synthétisé au niveau portes logiques.

L'ensemble automate et chemin de données sont placés et routés ensemble pour obtenir un circuit au niveau physique.
La simulation temporelle VHDL au niveau portes logiques est utilisée pour valider le résultat.

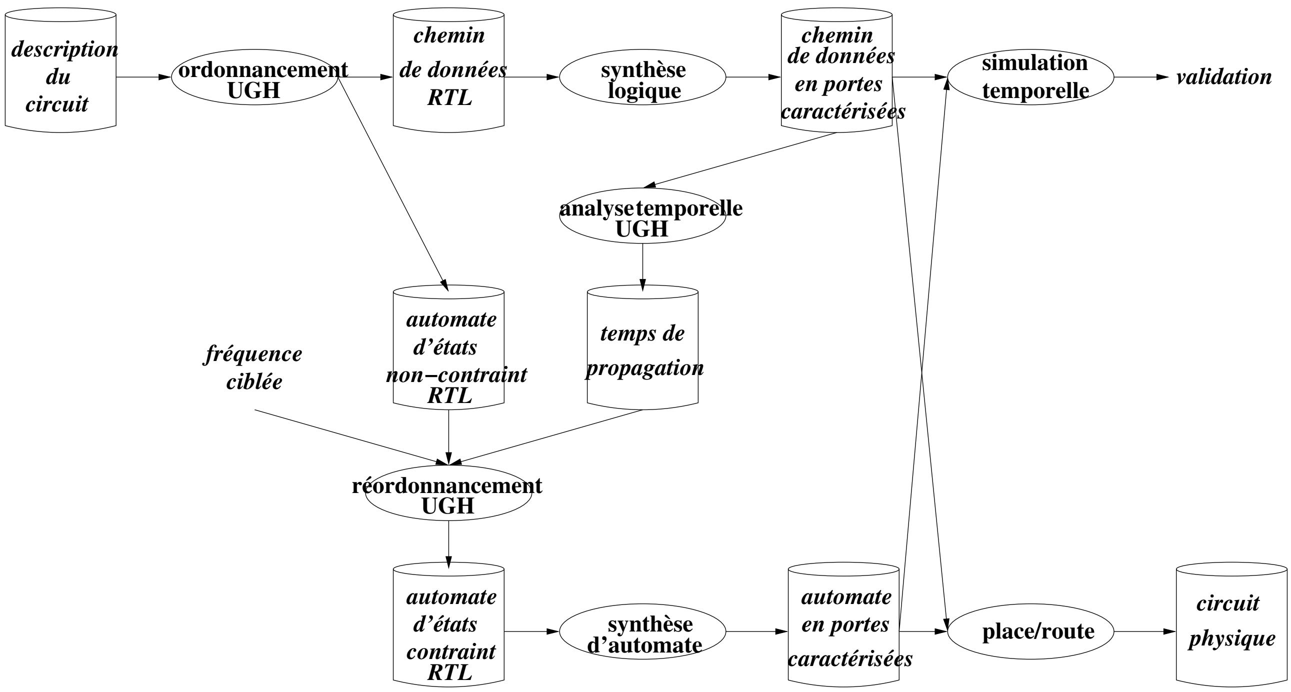


FIG. 7.3 Le flot logiciel avec la phase de réordonnancement.

La synthèse logique et la synthèse d'automate ont été réalisées avec l'outil *Design Analyser* de *Synopsys*. Le placement/routage a été réalisé avec l'outil *Silicon Ensemble* de *Cadence*.

La bibliothèque de cellules utilisée pour faire les tests est une bibliothèque de cellules logiques caractérisées. Cette bibliothèque nommée *sxlib* a été conçue au département *ASIM* du *LIP6*. Elle est distribuée avec la chaîne de CAO *Alliance*.
La technologie utilisée est une technologie de $0,35\mu m$. Cette technologie possède l'avantage de ne pas créer des fils trop résistifs et de ne pas trop influencer les temps de propagation.

Synthèse ciblée sans phase de réordonnancement

Si le concepteur est sûr de respecter la fréquence, il peut se passer de l'étape de *réordonnancement*. Il synthétise l'ensemble constitué de l'automate et du chemin de données au niveau portes logiques (cf. figure 7.4). Le résultat est placé et routé.
Pour vérifier le respect de la fréquence, le concepteur doit faire une analyse temporelle du résultat.

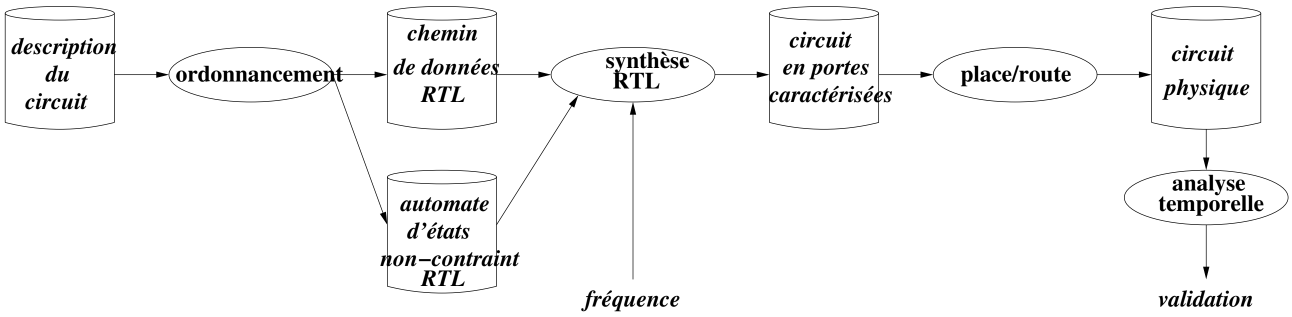


FIG. 7.4 Le flot logiciel sans la phase de réordonnancement.

La synthèse sur FPGA a été faite à l'aide de *Quartus*. Cet outil procède à la synthèse logique, au placement/routage et à l'évaluation des performances. Cet outil a été utilisé pour concevoir le contrôleur DMA.
La fréquence d'horloge de 33 MHz (vitesse de fonctionnement du bus) de ce contrôleur est facile à respecter car il y a très peu d'opérations arithmétiques dans les modules synthétisés.

Les autres circuits étant destinés à être intégrés dans un système sur puce, nous utilisons un flot de conception classique. La synthèse RTL a été réalisée avec l'outil *Design Analyser* de *Synopsys*. Le placement/routage a été réalisé avec l'outil *Silicon Ensemble* de *Cadence*. La bibliothèque de cellules utilisée est celle du *LIP6* en $0,35\mu m$.

7.1.3 Résultats

L'outil de synthèse ciblée *ugh* nous a permis de synthétiser avec succès tous les modules de nos applications présentées ci-dessus.
La simulation temporelle sous Synopsys des modèles structurels au niveau portes logiques fonctionnent pour les modules du Mpeg, multi-Jpeg et FD.
L'essai sur FPGA du contrôleur DMA fonctionne à 33 MHz.

Le tableau de la figure 7.5 présente les résultats obtenus regroupés suivant les quatre applications utilisées.

La description des résultats est la suivante :

- le *module* constitue une tâche ou une version de l'application.
- la *complexité* est le nombre de lignes de code de la description comportementale du module.
- la *taille* est la surface du module une fois celui-ci synthétisé au niveau portes logiques. Elle est exprimée en nombre d'inverseurs équivalents.

	module	complexité (nombre de lignes)	taille (nombre inverseurs)	nombre d'états	flot de données	temps		
						total	ugh	Synopsys /Quartus
Mpeg	VLD	704	10.060	109	non	1h18	0h38	0h40
multi-Jpeg	VLD	513	182.102	167	non	3h01	1h04	1h57
	IQ	93	12.520	38	oui	1h11	0h11	1h00
	ZZ	44	49.645	14	oui	1h55	0h24	1h31
	IDCT	186	73.776	113	oui	3h46	2h38	1h08
	LIBU	170	47.331	43	mixte	1h37	1h15	0h22
FD	FD 1	144	47.644	32	oui	2h35	1h40	0h55
	FD 2	144	51.826	32	oui	2h47	1h57	0h50
	FD 3	144	9.371	32	oui	0h15	0h02	0h13
DMA	TX	346	48.000	442	non	0h16	0h01	0h15
	RX	287	40.000	111	non	0h23	0h02	0h21
	WD	212	16.000	43	non	0h19	0h01	0h18

FIG. 7.5 Résultats obtenus grâce à la synthèse ciblée.

- le *nombre d'états* est le nombre d'états de l'automate du circuit synthétisé.
- le type *flot de données* est positionné à *oui* si l'automate est très peu complexe : il contient très peu de branchements par rapport à la complexité du chemin de données.
  Certains modules peuvent être *mixtes* en mélangeant des branchements et beaucoup de traitement de données.
- Le *temps total* est le temps pour obtenir un circuit au niveau portes logiques.
- Le *temps ugh* est le temps qu'a mis la synthèse ciblée pour générer un circuit au niveau RTL.
- Le *temps Synopsys/Quartus* est le temps qu'a mis la synthèse logique pour générer un circuit au niveau portes à partir du niveau RTL.

Nous pouvons déduire de ces résultats que l'outil *ugh* peut générer des circuits de plus de 100.000 portes. L'outil n'est pas limité par la taille des descriptions.
Les modules synthétisés sont soit des flots de contrôle tels les modules du contrôleur DMA, soit des flots de données tels les exemples *FD 1*, *FD 2*, *FD 3*, soit mixte tel le *LIBU*. L'outil *ugh* peut prendre n'importe quel type de description.

Quel que soit le type de description et la taille du circuit généré, la synthèse est parvenue à un résultat de façon rapide. Le nombre de chemins à évaluer évoluent de façon exponentielle suivant la taille des multiplieurs. C'est pourquoi les flots de données possédant des multiplieurs 32 bits comme *FD 1* et *FD 2* prennent plus d'une heure pour *ugh*. C'est l'extraction des temps de propagation qui se trouve le plus affectée par l'évolution exponentielle du nombre de chemins.

✓ La synthèse ciblée accepte aussi bien les descriptions dominées par des branchements conditionnels, que les flots de données ou mixtes.
✓ Les circuits de grosse taille ne sont pas bloquants pour la synthèse ciblée.
✓ La synthèse ciblée produit un résultat rapidement par rapport au reste du flot de conception.

7.2 La comparaison avec d'autres outils de synthèse

Les résultats de l'outil *ugh* ont été comparés à ceux d'autres outils de synthèse de haut niveau.
Les différentes caractéristiques étudiées dans les tableaux 7.7, 7.8, 7.9, 7.13, 7.14, 7.15, 7.16, 7.20, 7.21 sont :

- La *période ciblée* par l'outil *ugh* pour synthétiser le circuit.
  Pour les autres outils, ce temps est la fréquence maximale de fonctionnement du circuit synthétisé.
  Ce temps est évalué par simulation temporelle du circuit au niveau portes logiques grâce au simulateur VHDL de *Synopsys* et vérifié par l'analyseur de temps *tas* d'*Avertec* une fois le circuit placé et routé.
- Le *nombre d'états* de l'automate du coprocesseur.
  Plus la fréquence et/ou les temps de propagation sont élevés, plus le nombre d'états augmentent.
- Le *nombre de cycles d'exécution* du coprocesseur matériel pour traiter intégralement les données.
  Le nombre de cycles est obtenu par simulation du modèle VHDL au niveau portes logiques.
- Le *temps d'exécution* du coprocesseur matériel pour traiter intégralement les données.
  C'est le produit de la période du circuit par le nombre total de cycles d'exécution. Il donne la durée globale du traitement brut sans interférence avec le système dans lequel le coprocesseur est intégré.
- La *surface* du circuit après placement et routage avec une technologie $0,35\mu m$.
  Elle est obtenue à la fin du placement/routage du circuit par *Silicon Ensemble* de *Cadence*.
- Le nombre d'*inverseurs* équivalents pour donner une idée de la complexité du circuit.
  Ce nombre est obtenu à la fin de la synthèse logique par l'estimateur de *Design Analyser* de *Synopsys*.

Les outils de synthèse de haut niveau avec lesquels nous effectuons des comparaison sont *CoCentric* et *GAUT*.

Ces résultats sont comparés avec ceux de *ugh* obtenus avec le minimum de directives utilisateurs (i.e l'*affectation des registres*, le nombre et le type des opérateurs combinatoires).

CoCentric

CoCentric est un outil commercial développé par *Synopsys* et appartenant à l'ensemble *CoCentric*.
Cet outil demande uniquement une fréquence cible avec la description comportementale en *SystemC*.
CoCentric peut prendre en compte des informations supplémentaires sur le circuit pour raffiner une solution déjà obtenue. Cependant, nous décidons de l'utiliser avec le minimum de directives utilisateur.
CoCentric est utilisé dans le mode *macro-cycle* (cf. chapitre 3.3.2) : l'ordonnancement est libre entre les communications.

Les modèles *SystemC* pour *CoCentric* sont les mêmes que ceux donnés à *ugh*.
Seuls les types des variables et des ports ont changé pour respecter l'interface de *SystemC*. Les accès

aux ports ont été encapsulés dans des boucles d’attentes pour obtenir exactement le même comportement de type fifo sur les ports que les modèles *ugh*.
Mis à part ces différences concernant l’interface, la description des fonctions des modèles comportementaux est rigoureusement la même.
Le résultat de la synthèse est une vue structurelle de portes logiques. La bibliothèque de cellules utilisée par *CoCentric* est la même que celle utilisée par le flot de conception de l’outil *ugh*.

GAUT

GAUT (Génération Automatique d’Unités de Traitement) est un outil universitaire développé conjointement aux laboratoires du *Lester* (Université de Bretagne) et du *Lasti* (Université de Rennes). Il est destiné exclusivement à des descriptions avec un flot de données régulier. Selon [MSDP93] et [SDPC95], l’outil déroule les boucles pour optimiser l’*ordonnancement*. Il ne peut donc s’appliquer au *VLD* par exemple. L’outil demande une cadence et une fréquence cible avec la description comportementale en *VHDL*. *GAUT* peut être utilisé sur l’IDCT du multi-Jpeg.

opérateur 32 bits	temps (ns)	surface en inverseurs
registre	0,54	219
multiplieur	13,19	7.599
additionneur	3,68	676
soustracteur	3,68	841
AND bit à bit	0,46	52
tristate	0,155	350

FIG. 7.6 Échantillon de la caractérisation des opérateurs de *GAUT*.

Les modèles *VHDL* pour *GAUT* diffèrent légèrement des modèles *ugh* car ils sont de plus haut niveau. Les fonctions ont été supprimées et les tailles des variables ne sont pas exprimées.
De plus, *GAUT* requiert une bibliothèque d’opérateurs caractérisés. Ce sont des opérateurs arithmétiques, logiques et séquentiels. Nous avons donc synthétisé séparément les comportements *VHDL* des opérateurs avec *Synopsys* (cf. figure 7.6). Les caractéristiques extraites de la vue structurelle en portes logiques de chacun d’eux ont été ajoutées à la bibliothèque d’opérateurs de *GAUT*.
Il y a donc une imprécision dans le calcul de délai du à l’imprécision intrinsèque de la bibliothèque d’opérateurs. [JCM99a] montre que plus l’architecture du circuit est complexe, plus la surface prévue par *GAUT* s’écarte de la surface du circuit réel.
Pour diminuer les variations, l’outil essaie de minimiser les connexions (qui sont l’un des principaux facteurs d’imprédictibilité) grâce à sa stratégie de partage des opérateurs (cf. [JCM01] et [JCM99b]).

Le résultat de la synthèse de *GAUT* est une vue structurelle dont les feuilles sont les opérateurs de sa bibliothèque. Nous utilisons ensuite *Design Analyser* de *Synopsys* pour avoir une vue structurelle en portes du circuit.

7.2.1 Comparaison avec le VLD du Mpeg

Le VLD n'a pu être synthétisé avec *GAUT*. Nous comparerons les résultats uniquement avec *CoCentric*.
CoCentric, associé à *Design Analyser*, a mis 6h10 pour générer le VLD au niveau portes logiques.

	période ciblée (ns)	**nombre de cycles de l'automate**	**nombre de cycles d'exécution**	**temps d'exécution** (ms)	**surface** (mm^2)	**surface en nombre d'inverseurs**
CoCentric	9,61	120	7.186.411	69	2,2	17.214
ugh	10	90	5.232.247	**52,3**	**1,1**	9.842

FIG. 7.7 Comparaison des outils de synthèse de haut niveau sur le VLD du Mpeg.

L'outil *CoCentric* produit un circuit avec une fréquence maximale de $9,61ns$. Il ne peut produire un circuit avec une fréquence plus élevée.

Pour produire un résultat équivalent avec *ugh*, nous ciblons une fréquence de $10ns$.
L'outil *ugh* associé à la synthèse logique de *Synopsys* (*Design Analyser*) a produit une vue structurelle au niveau portes logiques en 1h18.
La synthèse ciblée est 4 fois plus rapide pour produire un circuit au niveau portes.

A une fréquence équivalente, l'outil de synthèse ciblée *ugh* est en tout point plus performant que l'outil *CoCentric*. La surface du circuit de *ugh* est **200 % plus petite** et la **vitesse d'exécution est 30 % plus importante**.
Le circuit de *CoCentric* est plus gros car il partage très peu les opérateurs. Il y a donc beaucoup de duplications d'opérateurs dans le chemin de données.

7.2.2 Comparaison avec l'IDCT du multi-Jpeg

L'IDCT a été synthétisé avec *CoCentric* et *GAUT*.
Nous supposons pour *GAUT* que l'environnement est synchrone avec le coprocesseur synthétisé. La description comportementale ne contient donc pas de synchronisation.
Ces résultats sont comparés avec ceux de *ugh* obtenus avec le minimum de directives utilisateurs.

L'*IDCT* attend 64 données en entrée avant de procéder à leur traitement puis produit 64 données en sortie. Quelle que soit la valeur des données, la vitesse d'exécution reste la même.
Nous mesurons donc le nombre de cycles d'exécution du circuit pour produire 64 données en sortie.
Il y a donc au minimum 128 cycles de lectures et d'écritures.

Nous avons par ailleurs obtenu les caractéristiques d'une DCT pour le décodage vidéo [CDM00]. Elle est le résultat d'une conception "à la main" et à été optimisée pour obtenir la meilleure performance en latence. Elle utilise le principe de l'arithmétique redondante qui optimise les temps de propagation des opérations arithmétiques mais est très gourmande en surface [Mul97]. Elle est constituée de 6 multiplieurs et de 10 additionneurs/soustracteurs contrôlés par un séquenceur.
Dans la figure 7.8 et la figure 7.9, ce circuit est appelé conception *à la main*, il constitue la référence pour les circuits synthétisés.

Malheureusement, le nombre de cycles de l'automate et la surface de ce circuit ne sont pas connus (i.e *nc* dans les tableaux).

CoCentric

L'architecture ciblée par *ugh* est une IDCT contenant 2 multiplieurs, 7 additionneurs et 5 soustracteurs. Ce sont des opérateurs 32 bits. De plus, elle contient 2 bancs de registres (de 32 et 8 bits) de 64 mots chacun.

	période ciblée (ns)	**nombre de cycles de l'automate**	**nombre de cycles d'exécution**	**temps d'exécution** (μs)	**surface** (mm^2)	**surface en nombre d'inverseurs**
à la main	10,41	*nc*	118	1,228	*nc*	242.120
CoCentric	21	86	1.645	34,545	19,94	165.660
ugh	17	90	1.466	**24,922**	**10,9**	70.885

FIG. 7.8 Comparaison des outils de synthèse de haut niveau sur l'IDCT du Jpeg.

La figure 7.8 nous donne les caractéristiques des circuits obtenus avec les différents outils de synthèse de haut niveau. L'outil *CoCentric* produit un circuit avec une fréquence maximale de 47 MHz (soit une période de $21ns$).
CoCentric, associé à *Design Analyser*, a synthétisé le circuit en 4h34.

La période ciblée par *ugh* est de $17ns$ pour obtenir des résultats comparables avec *GAUT* (cf. figure 7.9). L'outil *ugh*, associé à *Design Analyser*, a produit une vue structurelle au niveau portes en 3h04.
ugh est toujours plus rapide pour produire un résultat par rapport à *CoCentric*, mais il n'est que 30% plus rapide. Il y a une différence de vitesse de synthèse entre le VLD et l'IDCT. Ce sont les multiplieurs de l'IDCT qui rendent plus longue l'extraction des temps, nécessaire pour la phase de *réordonnancement* (cf. figure 7.3).

Le circuit généré par *ugh* est **200 % plus petit** pour une **vitesse d'exécution 50% plus rapide**.

GAUT

L'architecture ciblée par *ugh* est une IDCT contenant 2 multiplieurs, 7 additionneurs et 5 soustracteurs. Ce sont des opérateurs 32 bits.

GAUT a produit en deux minutes une IDCT au niveau RTL.
La multiplication a été caractérisée avec un temps de propagation de $13,19ns$ et la période visée est de $10ns$, elle prend donc deux cycles. Il y a 224 multiplications dans le flot d'opérations, soit deux fois plus de cycles (448 cycles). Nous visons donc une cadence de $400 * 10ns$, pour forcer *GAUT* à utiliser deux multiplieurs.

	période ciblée (ns)	**nombre de cycles de l'automate**	**nombre de cycles d'exécution**	**temps d'exécution** (μs)	**surface** (mm^2)	**surface en nombre d'inverseurs**
à la main	10,41	*nc*	118	1,228	*nc*	242.120
GAUT	17,499	398	$398 + 128$	**9,2**	19	123.568
ugh	17	90	1466	24,922	**10,9**	70.885

FIG. 7.9 Comparaison des outils de synthèse de haut niveau sur l'IDCT du Jpeg.

GAUT réussit à tenir la contrainte du nombre de cycles d'exécution, soit 398 cycles par bloc de 64 données. L'IDCT généré comporte 2 multiplieurs, 4 additionneurs, 3 soustracteurs, 195 registres de données et 50 bus (tous les opérateurs sont 32 bits).

La version de *GAUT* utilisée n'intègre pas les récents travaux réalisés par [**?**] sur la synthèse des communications. Cette version de *GAUT* ne tient pas compte de l'ordre des entrées/sorties et de la synchronisation avec le système. Les communications sont constituées de 64 lectures et 64 écritures soient 128 cycles.
La latence totale du circuit est de $398 + 128 = 526$ cycles.

La période d'horloge visée était de $10ns$, cependant le circuit synthétisé ne fonctionne pas à cette fréquence. La fréquence maximale de fonctionnement du circuit est de $17ns$, cela est du à l'imprécision de la caractérisation des opérateurs de la bibliothèque utilisée par *GAUT* (cf. figure 7.6). Il est difficile, aussi, de prévoir l'influence réelle des connexions dans les temps de propagation.
ugh obtient un circuit **270 % plus lent** pour une surface **200 % plus petite**.

GAUT obtient de meilleur résultat en vitesse d'exécution car il est adapté aux descriptions de type flot de données telle que l'IDCT.
ugh n'a pas le choix des registres car c'est le concepteur qui fournit l'*affectation des registres*. Chaque assignation de variable représentant un registre constitue une frontière de cycle dans le circuit. Il y a donc beaucoup de cycles imposés par le choix de l'*affectation des registres*.

Faire correspondre à chaque registre, un nom de variable de la description comportementale n'est pas adapté aux descriptions de type flot de données. Le partage des registres n'est alors pas possible. Cela augmente de façon problématique le nombre d'entrées des multiplexeurs des opérateurs combinatoires.
Par conséquent, les temps de propagation sont augmentés et des cycles d'attente supplémentaires sont ajoutés pour satisfaire la contrainte de fréquence.

Modification des directives

Pour obtenir une vitesse d'exécution plus grande de l'IDCT avec *ugh*, il faut modifier les directives.
En particulier, le concepteur doit repenser complètement l'utilisation des variables de la description comportementale. Il doit réutiliser les variables plusieurs fois dans la description pour sauvegarder des valeurs intermédiaires différentes tout en pensant à l'*affectation des registres* ciblée.
Pour cela, il faut qu'il développe lui-même "à la main" le flot de données pour trouver une affectation plus efficace. Cette démarche est très contraignante et pas forcément évidente.

Nous décidons de modifier les directives sur l'exemple de l'IDCT.
Le nombre d'opérateurs combinatoires est inchangé : l'architecture ciblée par *ugh* est une IDCT contenant 2 multiplieurs, 7 additionneurs et 5 soustracteurs.
Nous avons effectué quatre types de modifications dans les directives :

1. Les boucles ont été déroulées.

2. Les branchements ont été câblés.

3. Les deux bancs de registres (cf. l'architecture décrite au chapitre 7.1.1) ont été remplacés par 64 registres indépendants représentant les 64 données.

4. Les huit registres pour les calculs intermédiaires ont été changés. Ils sont placés de sorte à minimiser les coupures de cycle dans le flot de données. Il y a une barrière de registres toutes les deux multiplications logicielles car il n'existe que deux multiplieurs matériels.

	période ciblée (ns)	**nombre de cycles de l'automate**	**nombre de cycles d'exécution**	**temps d'exécution** (μs)	**surface** (mm^2)	**surface en nombre d'inverseurs**
à la main	10,41	*nc*	118	1,228	*nc*	242.120
GAUT	17,499	398	$398 + 128$	9,2	19	123.568
ugh optimisé en surface	17	90	1466	24,922	**10,9**	70.885
ugh optimisé en vitesse	17	460	460	**7,82**	18,4	119.346

FIG. 7.10 Optimisation de la vitesse de l'IDCT du Jpeg.

Ainsi, la figure 7.10 présente l'effet de la modification des directives sur la vitesse d'exécution. Par rapport au circuit de *GAUT*, le circuit généré par *ugh* pour optimiser la vitesse est **15% plus rapide** et **5% plus petit**.

L'outil *ugh* s'adresse aux concepteurs. Si les directives sont mal définies, les caractéristiques du circuit ne seront pas optimales.
Or pour les flots de données, l'*affectation des registres* est faisable automatiquement. Dans ce cas, les directives introduites par l'utilisateur sont un risque de limitation des capacités de l'outil de synthèse.

✓ L'outil *ugh* produit des circuits performants en vitesse d'exécution et en surface.

✓ L'*affectation des registres* par le concepteur n'est pas adaptée pour les descriptions comportementales de type flot de données.
Un outil orienté flot de données génère de façon automatique une *affectation des registres* presque aussi efficace.

✓ Les outils classiques de synthèse de haut niveau ne garantissent pas la fréquence de fonctionnement visée par le concepteur.

7.3 La couverture de l'espace des solutions

Nous voulons juger de la capacité de la synthèse ciblée à couvrir l'intégralité de l'espace des solutions en surface, vitesse d'exécution et fréquence. La figure 7.11 est un exemple imagé d'une bonne couverture sur deux dimensions de l'espace des solutions.

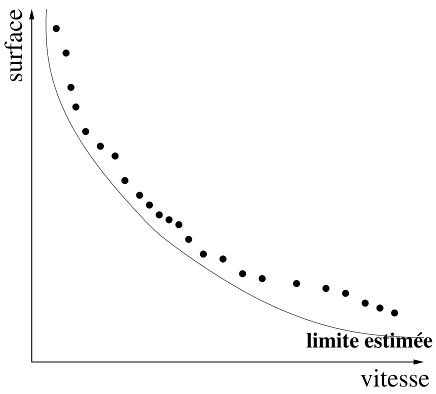


FIG. 7.11 Exemple imagé de couverture de l'espace des solutions

7.3.1 La flexibilité des critères du concepteur

Le concepteur doit fournir des informations à l'outil sur le chemin de données (cf. chapitre 5). Le concepteur peut choisir une description avec des contraintes plus ou moins fortes pour l'outil de synthèse *ugh* (User Guided High level synthesis). Il peut donner son choix d'*affectation des opérateurs combinatoires*, changer un type d'opérateur matériel et choisir les branchements à câbler.

Atteindre une fréquence

La technique du *réordonnancement* permet théoriquement d'atteindre n'importe quelle fréquence de fonctionnement pour un circuit. L'ajout de cycles d'attente n'augmente pas sa surface.
La figure 7.13 met en évidence l'efficacité du *réordonnancement*.

L'affectation des opérateurs combinatoires

L'outil de synthèse ciblée *ugh* demande une description grossière du chemin de données (cf. les directives architecturales du chapitre 5.2.2). Il permet une certaine flexibilité dans la description de ce chemin de données.
L'outil demande obligatoirement l'*affectation des registres* et le *nombre d'opérateurs arithmétiques* au concepteur. L'utilisateur peut choisir de descendre plus profondément dans la description du chemin de données suivant les aspects critiques du circuit.

1. **affectation complète :** Le concepteur peut décrire la totalité des connexions du chemin de données. Il fournit alors l'*affectation des opérateurs combinatoires*.
   Cela permet de minimiser la surface ou les temps de propagation puisque le concepteur agit sur les multiplexeurs situés aux entrées des opérateurs du chemin de données.
2. **affectation partielle :** Le concepteur peut décrire partiellement les connexions du chemin de données. Seule l'affectation de certains opérateurs combinatoires (les opérateurs critiques du circuit par exemple) est fournie à l'outil. L'outil est alors chargé de l'affectation des autres opérateurs combinatoires.
   En outre, le concepteur peut fournir plusieurs possibilités d'affectations sur les opérateurs combinatoires. Cela consiste à réduire les choix de l'outil à un ensemble déterminé et donc à forcer l'outil à choisir qu'entre certaines possibilités.
3. **affectation absente :** Le concepteur peut s'en tenir au strict minimum et ne pas donner l'*affectation des opérateurs combinatoires*.
   Seule l'*affectation des registres* est donnée à l'outil. L'outil est chargé de décider de l'affectation de tous les opérateurs combinatoires.

Cette contrainte donnée par l'utilisateur permet de couvrir tous les degrés de description de l'affectation du chemin de données.
La méthode d'*affectation des opérateurs combinatoires* du concepteur et celle, automatique, de l'outil ont des approches différentes.
L'outil cherche a minimiser la taille et les temps de propagation de façon homogène. Il cherche à équilibrer les multiplexeurs pour que les temps de propagation soient identiques d'un chemin à l'autre. Tandis que le concepteur optimise l'affectation sur une partie qu'il sait critique du coprocesseur.

Le concepteur peut alors choisir de fournir à l'outil uniquement l'*affectation des opérateurs combinatoires critiques*. Les opérateurs critiques sont des opérateurs souvent utilisés par la tâche (boucle, partage de l'opérateur entre plusieurs opérations). Si l'opérateur critique ne satisfait pas la fréquence de fonctionnement, le nombre de cycles d'attente insérés sera important.
Fournir l'*affectation des opérateurs combinatoires critiques*, c'est privilégier ces opérateurs matériels en minimisant les multiplexeurs à leurs entrées et les dédier exclusivement aux opérations logicielles les plus critiques. En minimisant les multiplexeurs, nous réduisons les temps de propagation des opérateurs critiques et donc le nombre de cycles d'attente.

A moins de faire de la preuve formelle ou de la simulation, la détection des opérations critiques n'est pas automatisable. L'outil de synthèse *ugh* ne favorise pas une opération aux dépends des autres car il n'est pas capable de détecter les opérations critiques. C'est donc au concepteur de fournir cette information (cf. les résultats obtenus dans la figure 7.14).

Le type des opérateurs du chemin de données

Le concepteur doit donner le nombre et le type des opérateurs (combinatoires et séquentiels) constituants le chemin de données (cf. chapitre 5.1.1).
Le nombre d'opérateurs influence directement les caractéristiques du circuit (surface et vitesse d'exécution). En augmentant le nombre d'opérateurs, on facilite la parallélisation des opérations.

Le concepteur peut aussi modifier le type de chaque opérateur. Le type sert à différencier les opérations acceptées par l'opérateur. Certains types permettent des opérations différentes telle l'ALU.

Les différents types d'opérateurs sont :

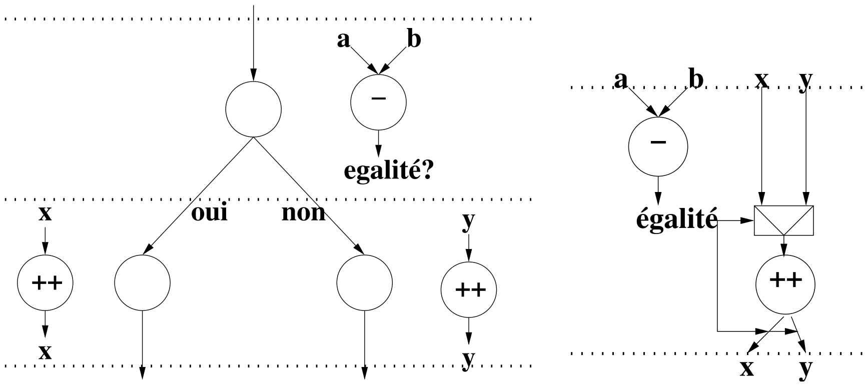
- l'additionneur.
- le soustracteur.
- l'additionneur/soustracteur.
- l'ALU (opérateur arithmétique et logique).
- le registre.
- le banc de registre à deux accès en lecture.
- la RAM statique à un accès en lecture.

L'ALU peut être remplacée par un soustracteur, un additionneur et des portes logiques.
Le banc de registre peut être remplacé par une RAM. L'unique différence entre le banc de registre et la RAM est que le banc de registre peut avoir deux accès simultanés en lecture à des index différents. La RAM possède un seul bus d'accès en lecture.
Nous allons tester l'impact des changements de type d'opérateur dans le chemin de données (cf. chapitre 7.15).

Le câblage des branchements conditionnels

Les tests des branchements conditionnels peuvent être câblés (cf. chapitre 5.1.1).
Le câblage diminue le nombre d'états de l'automate mais allonge les temps de propagation. En effet, il ajoute de la logique de décodage et, le cas échéant, du multiplexage. Dans la figure 7.12, un multiplexeur est inséré sur le chemin de propagation du signal.

```
if ( a == b ) x--;
else          y--;
```



(description comportementale) **(CDFG sans câblage)** **(CDFG avec câblage)**

FIG. 7.12 Câblage de branchement conditionnel.

La synthèse ciblée permet au concepteur de décider de câbler ou non un branchement. Ce qui donne des vitesses d'exécution très différentes.
Nous testons l'effet du câblage en fonction de la fréquence ciblée (cf. figure 7.16).

7.3.2 Résultats

Nous procédons aux variations des directives utilisateurs décrites plus haut sur le VLD du Mpeg et l'IDCT du multi-Jpeg.

Atteindre une fréquence

L'efficacité du *réordonnancement* est évaluée sur le VLD du Mpeg. Nous générons deux circuits avec et sans *réordonnancement* au niveau portes en utilisant la synthèse RTL. Les fréquences maximales de fonctionnement des circuits sont vérifiées par simulation temporelle.

	période ciblée (ns)	**nombre de cycles de l'automate**	**nombre de cycles d'exécution**	**temps d'exécution** (ms)	**surface** (mm^2)	**surface en nombre d'inverseurs**
avant réordonnancement	7,9	90	4.976.322	39,3	**1,12**	10.052
après réordonnancement	5	109	6.530.912	**32,6**	1,13	10.060

FIG. 7.13 Réordonnancement du VLD.

La figure 7.13 présente les caractéristiques du VLD synthétisé avec et sans *réordonnancement*.
Sans le *réordonnancement*, le circuit ne peut atteindre la fréquence de $5ns$. La synthèse RTL ne peut optimiser les chemins combinatoires du circuit au-delà de $7,9ns$.

Pour obtenir une période de $5ns$, le *réordonnancement* a ajouté une vingtaine d'états dans l'automate. La **surface a augmenté d'à peine 1 %** tandis que le **temps d'exécution a diminué de 28 %**. L'adaptation à la fréquence du système par le *réordonnancement* a un coût très faible.
Le *réordonnancement* nous permet de couvrir l'espace des fréquences.

L'affectation des opérateurs combinatoires sur le VLD du Mpeg

Nous allons tester l'efficacité et la pertinence de l'*affectation des opérateurs combinatoires* du concepteur pour l'outil *ugh*. La tâche choisie pour être synthétisée est le VLD du Mpeg.

Il existe plusieurs opérations logicielles critiques dans le VLD. Nous choisissons de favoriser la plus critique de toutes dans l'affectation partielle fournie à l'outil. Il s'agit d'un décalage suivi d'une addition ou d'une soustraction. Cette opération est exécutée un peu plus souvent que les autres, ce qui la rend critique.
L'*affectation des opérateurs critiques* est appelée *affectation partielle* dans la figure 7.14.

La figure 7.14 donne les caractéristiques des circuits du VLD suivant la précision du chemin de données fourni en entrée de l'outil *ugh*. Les résultats obtenus sont globalement assez semblables pour une fréquence d'horloge de 200 MHz (soit une période de $5ns$).

	période ciblée (ns)	nombre de cycles de l'automate	nombre de cycles d'exécution	temps d'exécution (ms)	surface (mm^2)	surface en nombre d'inverseurs
affectation complète	5	109	6.530.912	**32,6**	**1,13**	10.060
affectation partielle	5	112	6.905.663	34,5	1,14	10.168
affectation absente	5	115	6.936.683	34,6	1,14	10.134

FIG. 7.14 Influence de l'affectation des opérateurs combinatoires.

Par rapport à l'*affectation complète* fournie par le concepteur, l'affectation automatique (i.e l'*affectation absente* du concepteur) donne des résultats moins bons. Ils divergent de **5% en temps d'exécution** et de **1% en surface** en faveur d'un circuit dont l'*affectation* est effectuée "à la main".

En fournissant l'*affectation des opérateurs critiques* (i.e l'*affectation partielle* du concepteur), nous avons choisis de favoriser l'opération la plus utilisée dans la description.
Cette directive d'affectation nous permet de réduire la différence entre un circuit avec l'affectation du concepteur et celui avec l'affectation automatique. Cependant le gain est très faible par rapport au gain de **5 % en nombre d'états**.
Nous avons favorisé une opération critique au détriment des autres opérations logicielles ! Il y a des multiplexeurs plus importants sur les opérateurs matériels des autres opérations logicielles, ce qui augmente leurs temps de propagation. Ces opérations font perdre des cycles à des endroits importants du traitement. Cette perte compense le gain sur l'opération critique.

Sur l'exemple du VLD du Mpeg, nous pouvons dire que l'affectation automatique de l'outil est suffisante pour caractériser le circuit. Elle est suffisamment efficace pour produire un résultat proche de celui du concepteur.

Le type des opérateurs sur l'IDCT du Jpeg

Nous faisons varier les types des opérateurs que le concepteur fournit à *ugh* :

1. 1 banc de registres de 64 mots de 32 bits à deux entrées en lecture et 8 ALUs de 32 bits.
2. 1 banc de registres de 64 mots de 32 bits à deux entrées en lecture, 7 additionneurs simples de 32 bits, 5 soustracteurs simples de 32 bits.
3. 1 banc mémoire de 64 mots de 32 bits à une seule entrée en lecture et 8 ALUs de 32 bits.

Le banc de registres prend plus de surface que le banc de mémoire mais il permet deux lectures simultanées. La description comportementale de l'IDCT permet des lectures simultanées, cela permet de gagner des cycles.
Les 8 ALUs sont remplacées par 8 additionneurs et 8 soustracteurs simples, cependant l'outil n'utilise que 7 des additionneurs et 5 des soustracteurs. Les 12 opérateurs simples occupent quasiment la même surface que les 8 ALUs mais ils permettent surtout de minimiser le nombre de multiplexeurs en distribuant les opérations.

	période ciblée (ns)	nombre de cycles de l'automate	nombre de cycles d'exécution	temps d'exécution (μs)	surface (mm^2)	surface en nombre d'inverseurs
banc de registres +8 ALUs	50	61	1.234	61,7	13,76	74.501
banc de registres +7 additionneurs +5 soustracteurs	50	61	1.234	**61,7**	10,89	70.761
banc mémoire +8 ALUs	50	67	1.282	64,1	**9,41**	66.355

FIG. 7.15 Influence des types d'opérateurs.

La figure 7.15 présente les caractéristiques des circuits de l'IDCT obtenus suivant les modèles d'opérateurs utilisés dans les directives du concepteur. La fréquence ciblée de 20 Mhz (soit une période de $50ns$) est la même pour tout les circuits afin de ne pas interférer avec les variations de vitesse et de surface.

Le banc de mémoire, s'il entraîne des cycles supplémentaires pour la lecture, est plus petit que le banc de registres, le banc de mémoire permet de **réduire de 30% la surface**.
Si l'on considère le nombre d'opérateurs, la première architecture compte moins d'opérateurs (8) que la deuxième (7+5), pourtant, la deuxième architecture est **plus petite de 21 %**. Une ALU est plus grosse de 20 % par rapport à un additionneur et un soustracteur simples. En outre, avec plus d'opérateurs, l'outil peut mieux équilibrer les multiplexeurs. La minimisation des multiplexeurs compense les 4 opérateurs de différences !

Les branchements câblés sur le VLD du Mpeg

Nous allons tester les effets du câblage des branchements conditionnels sur le VLD. Nous faisons deux types de description du VLD où dans l'une, nous interdisons tout câblage, et dans l'autre, nous forçons la majeure partie des branchements à être câblés.
A priori, l'effet du câblage doit donc diminuer le temps d'exécution et augmenter la surface du circuit.

	période ciblée (ns)	nombre de cycles de l'automate	nombre de cycles d'exécution	temps d'exécution (ms)	surface (mm^2)	surface en nombre d'inverseurs
empêcher le câblage	5	177	6.248.232	**31,24**	**1,19**	10.652
imposer le câblage	5	113	6.917.039	34,5	1,23	10.986
empêcher le câblage	20	147	5.295.909	105,9	1,17	10.508
imposer le câblage	20	88	5.013.194	**100,2**	**1,14**	10.161

FIG. 7.16 Influence des branchements câblés.

La figure 7.16 compare l'effet du câblage sur le VLD avec deux fréquences d'horloge différentes : 200 Mhz (soit une période de $5ns$) et 50 Mhz (soit une période de $20ns$).

Le câblage donne de meilleurs résultats lorsque la fréquence est faible.
Le câblage rajoute de la logique dans le chemin de données, il allonge donc les temps de propagation et fait disparaître les branchements dans l'automate. Or la phase de *réordonnancement* doit ajouter des cycles d'attente pour compenser l'allongement des temps de propagation et tenir la contrainte d'une haute fréquence.
Dans le VLD, les opérateurs constituant la chaîne longue effectuent les opérations logicielles critiques. L'ajout de cycles d'attente, même peu nombreux, entraîne un ralentissement notable de la vitesse d'exécution !

- ✓ Toutes les fréquences sont atteignables quel que soit le circuit.
- ✓ Les différentes directives que le concepteur peut fournir à l'outil permettent de couvrir l'ensemble de l'espace des solutions de manière très précise.
- ✓ L'*affectation des registres* est suffisamment forte pour que le concepteur contrôle les caractéristiques de son circuit.
- ✓ L'outil *ugh* est capable de prendre des décisions si les contraintes sont absentes. Cela laisse au concepteur le choix de la finesse des contraintes qu'il impose à l'outil et permet de diminuer le temps de conception.
  Avec les directives minimum, l'outil prend de bonnes décisions dans la plupart des cas. La solution n'est pas alors un *Point de Pareto* mais elle est souvent satisfaisante.
- ✓ Cependant le concepteur doit donner des directives plus précises si le circuit visé possède des caractéristiques singulières. Le concepteur obtiendra alors le *Point de Pareto* de l'outil pour ce circuit.

7.4 L'intégration de la synthèse ciblée dans le flot de conception

Pour pouvoir effectuer le placement et le routage du circuit, le résultat final de la synthèse doit être une vue structurelle au niveau portes logiques. Il existe de nombreux outils de synthèse et d'optimisation permettant de parvenir à cette vue structurelle. La plupart d'entre eux travaillent sur des descriptions de niveau RTL (Register Transfer Level). Une description de niveau RTL est une description où les registres sont identifiés et où leurs opérations sont connues pour chaque cycle du circuit.
Nous identifions quatre types d'outils modifiant une description de niveau RTL :

- la synthèse d'automate d'états de commandes.
  Chaque état comporte les signaux de contrôle du chemin de données.
- la synthèse d'automate d'états fonctionnels.
  Chaque état comporte la description comportementale des opérations du chemin de données.
- la synthèse de chemin de données (*optimisation du chemin de données à la fréquence*).
- le *réordonnancement* d'automate d'états (*optimisation de l'automate d'états à la fréquence* et au temps de propagation du chemin de données cf. chapitre 4).

Les transformations effectuées par ces quatre outils sont représentées sur la figure 7.17. Les arcs en pointillés représentent les transformations et les optimisations de la synthèse logique. Elle est souvent intégrée aux outils RTL énoncés plus haut.

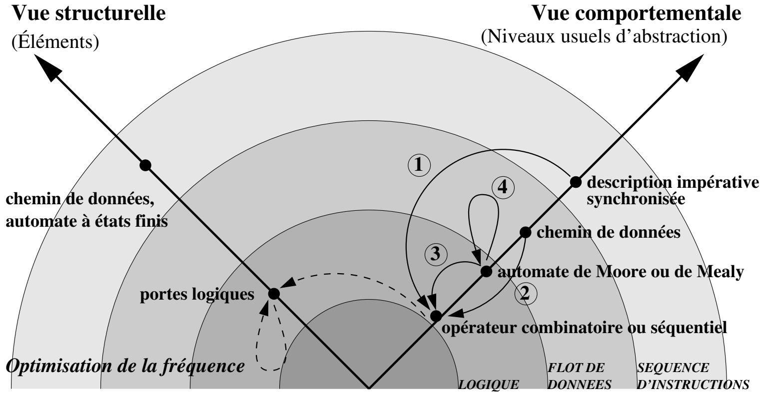


FIG. 7.17 Transformations existantes pour obtenir une vue structurelle en portes.

Grâce aux différents outils RTL, la synthèse ciblée de haut niveau a plusieurs possibilités pour s'intégrer dans le flot de conception.

7.4.1 Les différentes intégrations

La synthèse ciblée peut s'intégrer de trois manières différentes dans le flot de conception. Les trois schémas de la figure 7.18 représentent les trois intégrations possibles. Les arcs pleins symbolisent les transformations effectuées par la synthèse de haut niveau et les arcs en pointillés les transformations de la synthèse RTL.

1. Dans l'*intégration* $n^o 1$,
   - **objectifs de la synthèse ciblée de haut niveau :** ordonnancement.
   - **résultats de la synthèse ciblée :** automate fonctionnel et chemin de données comportemental. Le chemin de données et l'automate d'états sont fusionnés (cf. l'exemple du chapitre C.1).
   - **objectifs de la synthèse RTL :** *affectation des opérateurs combinatoires* du chemin de données et *optimisation de la fréquence en modifiant les chemins combinatoires*.
   - **outils RTL utilisés :** synthèse d'automate d'états fonctionnels.
2. Dans l'*intégration* $n^o 2$,
   - **objectifs de la synthèse ciblée de haut niveau :** ordonnancement.
   - **résultats de la synthèse ciblée :** automate de commandes et chemin de données comportemental. Le chemin de données et l'automate d'états sont séparés (cf. l'exemple du chapitre C.2).
   - **objectifs de la synthèse RTL :** *affectation des opérateurs combinatoires* du chemin de données et *optimisation de la fréquence en modifiant les chemins combinatoires du chemin de données*.

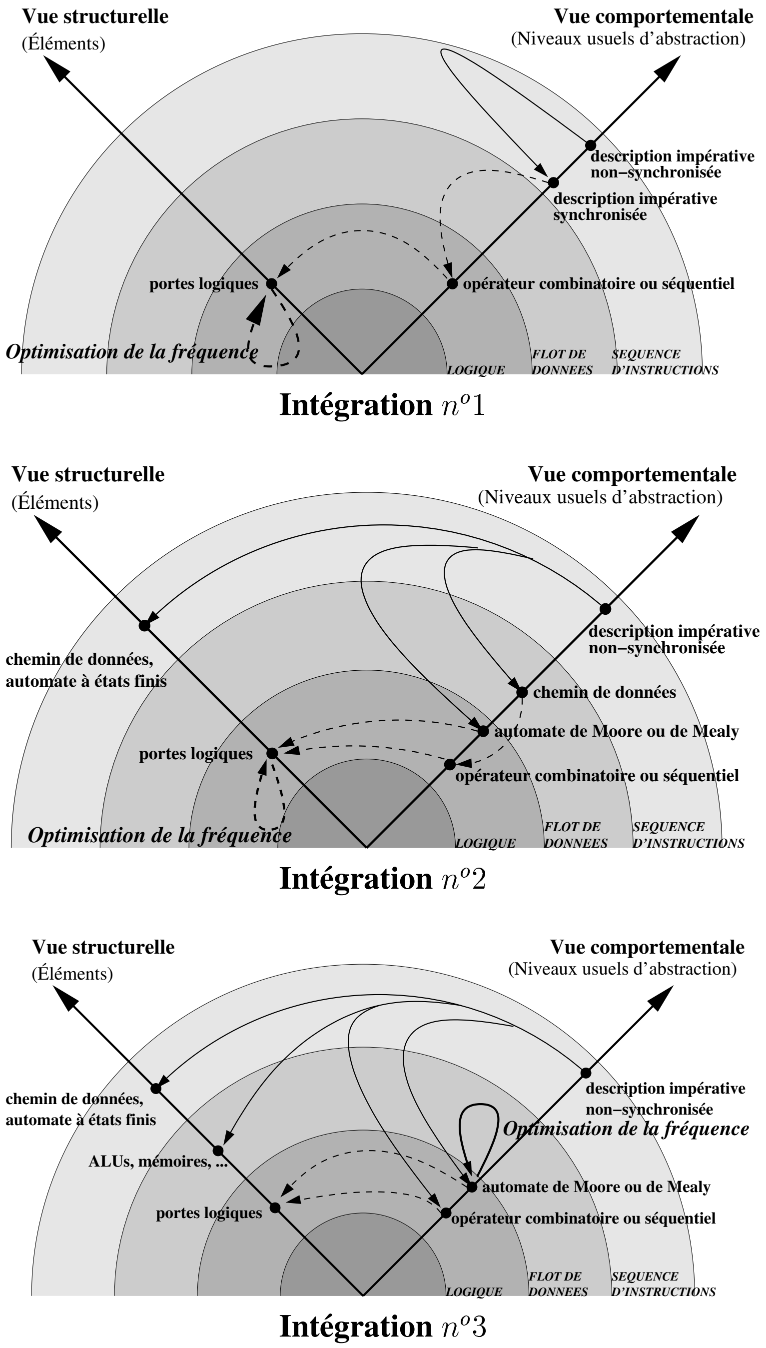


FIG. 7.18 Les trois intégrations de la synthèse ciblée.

- **outils RTL utilisés :** synthèse de chemin de données et synthèse d'automate de commandes.

3. Dans l'*intégration* n^o3,
   Cette intégration correspond à la méthode de l'outil de synthèse ciblée *ugh* telle qu'elle a été présentée dans le chapitre 4.

   - **objectifs de la synthèse ciblée de haut niveau :** ordonnancement et *affectation des* *opérateurs combinatoires* du chemin de données.

- **résultats de la synthèse ciblée :** Le chemin de données est une vue structurelle d'opérateurs (cf. l'exemple du chapitre C.3). L'automate de commandes est une description comportementale.
- **objectifs de la synthèse RTL :** *optimisation de la fréquence en modifiant l'automate de commandes*. Chaque opérateur du chemin de données est synthétisé séparément des autres opérateurs.
- **outils RTL utilisés :** *réordonnancement* et synthèse d'automate de commandes.

L'efficacité des trois intégrations dépend essentiellement de l'efficacité de l'outil utilisé pour effectuer l'*affectation des opérateurs combinatoires* : la *synthèse ciblée de haut niveau* ou la *synthèse d'automate fonctionnel* ou la *synthèse de chemin de données*.
Cependant, les intégrations possèdent aussi des propriétés inhérentes à la méthode utilisée.

7.4.2 Avantages et inconvénients des méthodes d'intégration

Possibilités de réordonnancement

La seule condition pour être théoriquement compatible avec le *réordonnancement* de la synthèse ciblée (l'outil $n^o 4$ de la figure 7.17), c'est d'isoler la *fonction de transition de l'automate d'états* du reste du circuit et de l'adapter une fois les caractéristiques électriques du chemin de données connues. En pratique, la *fonction de génération* est rarement séparée de la *fonction de transition* car cela décorrèle l'encodage des états, des valeurs de commande envoyées au chemin de données.

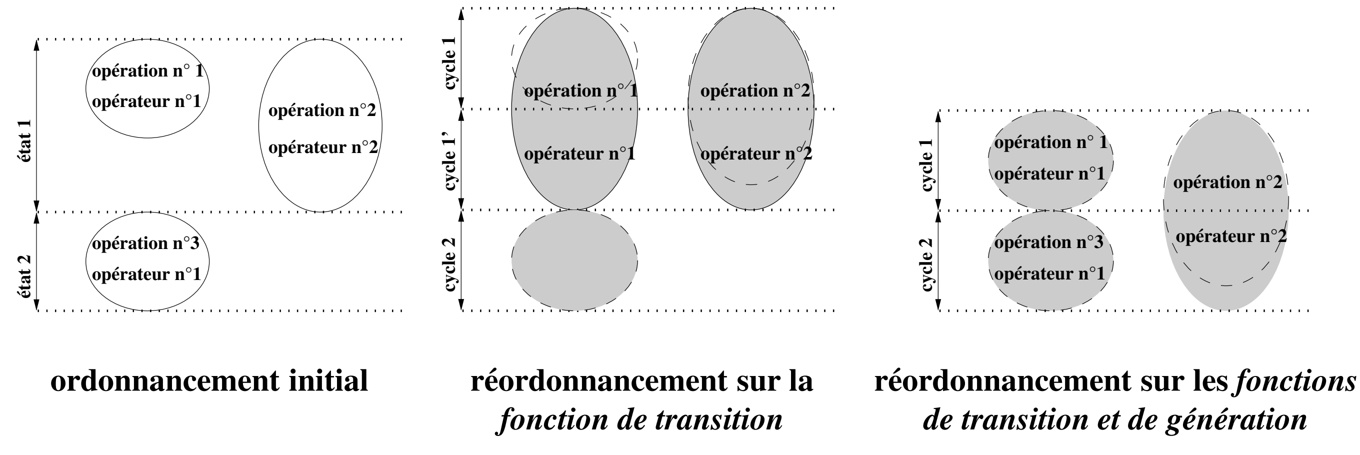


FIG. 7.19 Réordonnancements de l'*opération* $n^o 2$.

- Si l'outil de *réordonnancement* ne s'applique qu'à la *fonction de transition* de l'automate, il peut insérer des cycles supplémentaires dans l'automate, mais il ne peut pas déplacer une opération indépendamment des autres opérations du cycle.
- Si l'outil de *réordonnancement* s'applique aux *fonctions de transition et de génération* de l'automate, il peut insérer des cycles supplémentaires et déplacer des opérations d'un cycle à un autre.

Remarque : La *fonction de génération* n'est pas intégralement séparée de la *fonction de transition*. La *fonction de transition* définit l'état de l'automate et la *fonction de génération* définit les commandes du chemin de données en rapport avec cet état.

Dans l'exemple de la figure 7.19, l'*opération 2* ne tient pas dans le cycle.

1. L'*ordonnancement initial* présente les temps de propagation de chaque opération.
2. Le *réordonnancement sur la fonction de transition* a inséré un cycle supplémentaire (*cycle 1'*) où les *opérations* n^o1 *et* n^o2 continuent à s'exécuter.
   La *fonction de génération* est inchangée. En effet, les *cycles 1 et 1'* correspondent à l'*état 1* et le *cycle 2* correspond à l'*état 2*.
3. Le *réordonnancement sur les fonctions de transition et de génération* a copié l'*opération* n^o2 dans le *cycle 2*, elle a donc maintenant deux cycles pour se propager.
   La *fonction de génération* est modifiée. En effet, le *cycle 1* correspond l'*état 1* mais le *cycle 2* ne correspond pas l'*état 2*.
   Bien sûr, il ne doit y avoir ni dépendance de données, ni dépendance matérielle entre l'*opération* n^o2 et l'*opération* n^o3.

Le *réordonnancement* est plus efficace lorsqu'il s'applique à un automate à états contenant la *fonction de génération*.
Cela permet de remanier les états de l'automate. Ainsi, le *réordonnancement* des opérations est au plus juste.

Applications du réordonnancement aux trois intégrations

1. *Intégration* n^o1 : Aucun cycle d'attente ne peut être inséré. Car la *fonction de transition* est indissociable du chemin de données.
2. *Intégration* n^o2 : Des cycles d'attente peuvent être insérés. Car la *fonction de transition* est dissociée du chemin de données.
3. *Intégration* n^o3 : Des cycles d'attente peuvent être insérés et les opérations peuvent être déplacées de façon indépendante. Car les *fonctions de transition et de génération* sont dissociées du chemin de données.

Malheureusement, il existe aujourd'hui des contraintes techniques sur les outils. Le *réordonnanceur* de *ugh* nécessite que l'*affectation des opérateurs combinatoires* du chemin de données soit effectuée par la synthèse de haut niveau. Seule l'intégration n^o3 permet le *réordonnancement*.

Limites des intégrations

Les descriptions comportementales du chemin de données au niveau RTL ne comportent pas d'*affectation des opérateurs combinatoires*.
Les *intégrations* n^o1 *et* n^o2 remettent en cause un des principes de la synthèse ciblée : le concepteur ne peut plus cibler une *affectation des opérateurs combinatoires* ni même le nombre d'opérateurs combinatoires !

7.4.3 Résultats sur le VLD du Mpeg

	période ciblée (ns)	**nombre de cycles de l'automate**	**nombre de cycles d'exécution**	**temps d'exécution** (ms)	**surface** (mm^2)	**surface en nombre d'inverseurs**
Intégration n^o1	7,738	90	4.976.322	**38,5**	1,83	14.010
Intégration n^o2	8,838	90	4.976.322	43,9	1,64	12.569
Intégration n^o3	7	98	5.847.475	40,9	**1,1**	9.853

FIG. 7.20 Efficacité de l'affectation VLD.

La figure 7.20 évalue les circuits VLD générés par les trois intégrations du flot de conception.
L'*optimisation maximale du chemin de données* (*intégrations* n^o1 *et* n^o2) ne peut faire descendre la fréquence en dessous des $7ns$, c'est donc la fréquence cible pour le *réordonnanceur* de *ugh* (*intégration* n^o3).
Pour obtenir la fréquence de $7ns$, 8 cycles ont été insérés par le *réordonnanceur* dans l'automate. Cela se paie en temps d'exécution car les cycles ont été ajoutés à des endroits critiques de l'automate (i.e à l'intérieur de boucle). Le circuit généré par l'*intégration* n^o3 est **6 % plus lent en temps d'exécution** que celui de l'*intégration* n^o1.

L'*intégration* n^o3 de la synthèse ciblée a produit un circuit **40 % plus petit** que les autres intégrations.
L'*optimisation du chemin de données* produit un circuit plus gros que l'*optimisation de l'automate d'états*.
Pour optimiser les temps de propagation de la description comportementale du chemin de données et faire respecter la fréquence, l'*optimisation du chemin de données* ajoute des opérateurs combinatoires (méthode de procrastination détaillée dans le chapitre 4.2.1).
La surface du circuit augmente donc. De plus, l'*optimisation des temps de propagation du chemin de données* optimise aussi les faux chemins.
Alors que le *réordonnancement optimise l'automate* pour respecter la fréquence. Il intervient en insérant des cycles uniquement là où les opérations fonctionnelles dépassent le temps de cycle.

Remarque : Une description avec le chemin de données et l'automate d'états séparés (*intégration* n^o2) est moins performante qu'une description associant l'automate et le chemin de données (*intégration* n^o2).
Dans l'*intégration* n^o2, la *fonction de génération* est indissociable du chemin de données. L'encodage des états est donc décorrélé de la *fonction de génération*, il est donc moins adapté et génère de la logique supplémentaire sur les commandes du chemin de données.
Le circuit généré est donc plus gros et plus lent que ceux généré à partir d'un automate contenant la *fonction de génération*.

7.4.4 Résultats sur l'IDCT du multi-Jpeg

La figure 7.21 évalue les circuits IDCT générés par les trois intégrations du flot de conception.
Pour les mêmes raisons que sur le VLD, l'intégration n^o2 n'est pas intéressante. La séparation de la *fonction de génération* de la *fonction de transition* est trop pénalisante.

	période ciblée (ns)	**nombre de cycles de l'automate**	**nombre de cycles d'exécution**	**temps d'exécution** (ms)	**surface** (mm^2)	**surface en nombre d'inverseurs**
Intégration n^o1	30,235	63	1.282	**38,761**	42,12	205.310
Intégration n^o2	34,511	63	1.282	44,243	36,26	196.220
Intégration n^o3	30	78	1.370	41,1	**9,35**	65.977

FIG. 7.21 Efficacité de l'affectation sur l'IDCT.

L'*optimisation maximale du chemin de données* (*intégrations* n^o1 *et* n^o2) ne peut faire descendre la fréquence en dessous des $30ns$, c'est donc la fréquence cible pour le *réordonnanceur de ugh* (*intégration* n^o3).
A partir du même automate, l'*ordonnanceur de ugh* a estimé devoir insérer des cycles supplémentaires pour respecter la fréquence visée. Le circuit généré par l'*intégration* n^o3 par *optimisation de l'automate* est donc **moins rapide de 6 %**. Mais les autres intégrations ont du créer un circuit **500 % plus gros** pour optimiser la fréquence !
On atteint donc les limites de l'*optimisation du chemin de données* pour respecter la fréquence.

La méthode d'allocation par procrastination des opérateurs arithmétiques de la synthèse RTL ne marche plus lorsqu'il s'agit de description faisant des opérations arithmétiques complexes et nombreuses (des multiplications et des additions dans l'exemple de l'IDCT). Le nombre d'opérateurs ajoutés produit un circuit très gros en surface. Cela pose des problèmes pour obtenir une vue physique correcte (sans conflit de routage). De plus, la distance entre les cellules augmente les temps de propagation et cela dégrade les performances du circuit.

✓ Il existe deux façons intéressantes d'intégrer la synthèse ciblée dans le flot de conception :
  - *optimisation de l'automate d'états* (*intégration* n^o3).
  - *optimisation du chemin de données* (*intégration* n^o1).

✓ L'*optimisation du chemin de données* produit un circuit plus rapide.

✓ L'*optimisation de l'automate d'états* produit un circuit plus petit.

✓ Le choix entre les deux intégrations dépend du circuit et de la fréquence ciblée :
  - Si la période visée est très inférieure à la chaîne longue du circuit, il faut utiliser l'*optimisation de l'automate d'états*.
  - Dans les autres cas, il n'y a pas de méthode systématiquement plus efficace, il faut essayer les deux types d'intégration.

✓ Pour tous les circuits et toutes les fréquences, l'*optimisation de l'automate d'états* (*intégration* n^o3) donne un résultat acceptable.

7.5 Les perspectives d'amélioration

Des améliorations de la méthode logicielle de *ugh* sont possibles. Elles peuvent faire gagner en temps d'exécution et en surface sur les circuits générés.

Nous avons vu dans le chapitre 7.4, que l'*optimisation de l'automate d'états* produit généralement des circuits plus lents que l'*optimisation du chemin de données* par la synthèse RTL (même si celle-ci produit des circuits inutilisables car trop gros).
Cet écart de vitesse d'exécution peut être diminué par l'amélioration de deux aspects constituant la méthode logicielle de l'outil *ugh* : l'amélioration de la synthèse logique des opérateurs et l'amélioration de l'extraction des temps du chemin de données.

Le chapitre 7.2 met en évidence que les algorithmes d'allocation automatique sont plus efficaces sur des descriptions régulières. La cohabitation de plusieurs algorithmes de synthèse et le remaniement des boucles peuvent améliorer l'efficacité de *ugh*.

7.5.1 Synthèse logique des opérateurs du chemin de données

La synthèse logique calcule les temps de propagation grâce au réseau RC (Résistance * Capacité), c'est sa principale fonction de coût pour l'optimisation.
Or dans la méthode de la synthèse ciblée, chaque opérateur est synthétisé séparément au niveau portes logiques. La résistance et la capacité aux ports des opérateurs ne sont donc pas connus. La synthèse logique ne peut donc pas évaluer les chemins combinatoires, elle reste à une vision locale à chaque opérateur.
La fonction de coût de la synthèse est biaisée. Les optimisations de la synthèse logique pour diminuer les temps de charge des fils sont inopérantes.

De plus, la synthèse logique ne peut simplifier les chemins combinatoires entre deux opérateurs puisqu'ils sont synthétisés séparément. Cela engendre plus de logique combinatoire et donc plus de temps de propagation.
Le chemin de données doit être synthétisé globalement pour avoir une optimisation logique optimale.

7.5.2 Analyse temporelle du circuit

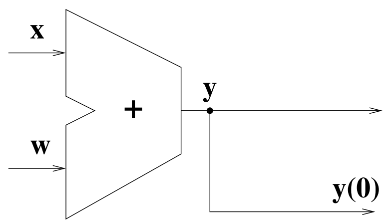


FIG. 7.22 Utilisation du bit de poids faible de l'additionneur.

Les temps des opérateurs sont extraits à partir de la vue structurelle en portes logiques. Ce temps est égal au pire cas des chemins combinatoires de l'opérateur.
L'inconvénient de cette simplification est qu'il n'y a pas de distinction entre les bits de sortie de l'opérateur.
Dans l'exemple de la figure 7.22, le bit de poids faible de l'additionneur *y(0)* a le même temps de propagation par rapport aux entrées que le bit de poids fort.

C'est une approximation très pessimiste. Les temps de propagation extraits sont plus longs que les temps réels. Il y aura donc des cycles d'attente insérés inutilement.

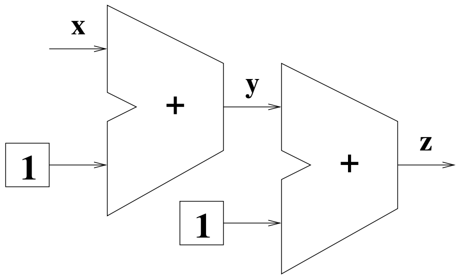


FIG. 7.23 Cumul de la majoration du temps de propagation.

Le fait de prendre le pire cas majore les temps de propagation. Dans l'exemple de la figure 7.23, la majoration se cumule sur deux additionneurs : $t^{max}_{x\to z} = t^{max}_{x\to y} + t^{max}_{y\to z}$. C'est une évaluation très pessimiste car le temps de propagation du bit de poids fort sortant du premier opérateur n'est pas l'entrée des bits de poids faible du deuxième opérateur.

Il faut conserver l'évaluation bit à bit des temps propagation des chemins combinatoires à l'intérieur des opérateurs. Les temps de propagation du chemin de données seront analysés plus précisément. Ainsi, on évitera d'insérer des cycles d'attente inutiles.

7.5.3 Cohabitation de plusieurs algorithmes de synthèse

L'*allocation des registres* par le concepteur permet de contrôler le résultat de la synthèse. Les registres sont liés à des variables de la description et ne peuvent pas échapper à cette affectation.
Cependant lorsqu'il s'agit de description de type flot de données, les registres peuvent être réutilisés de façon avantageuse dans de nombreuses opérations logicielles. Les registres ne représentent plus des variables mais des valeurs intermédiaires en vue d'une prochaine opération. Les registres sont donc liés à des opérateurs combinatoires. Cela permet de minimiser le nombre de connexions (i.e les multiplexeurs ou les bus) entre les registres et les opérateurs combinatoires.
Plus il y a des connexions différentes, plus un multiplexeur est gros et plus le nombre de couches logiques est important. La minimisation de ces connexions diminue donc les temps de propagation et la surface du circuit.

Pour être plus efficace sur les descriptions de type flot de données, il faut un algorithme d'*allocation automatique des registres* pour identifier les registres nécessaires dans le DFG tel que le *left edge* [KP87] qui est simple et rapide.

7.5.4 Remaniement des boucles

Le déroulement et le repliement de boucle détaillés dans le chapitre 5.1.1 sont décidés par le concepteur car leur impact est imprédictible.
Cependant, pour les descriptions régulières de type flots de données, ces optimisations deviennent automatisables car elles sont souvent très avantageuses. La latence des boucles *for* peut être évaluée.

➠ Le chemin de données doit être synthétisé globalement et non pas opérateur par opérateur.

➠ Les temps de propagation doivent être analysés bit à bit et non pas uniquement sur le pire cas.

➠ Les descriptions de type flot de données doivent être traitées avec des algorithmes d'allocation automatique.

7.6 Conclusion

✓ L'outil de synthèse ciblée *ugh* produit rapidement des circuits performants quel que soit le type de circuit et sa taille.

✓ L'outil de synthèse ciblée *ugh* supporte des directives du concepteur de précisions différentes. Cela permet de cibler précisément un circuit ou bien de concevoir un circuit rapidement.

✓ L'outil de synthèse ciblée *ugh* supporte différentes intégrations dans le flot de conception.

✓ L'*affectation des registres* par le concepteur n'est pas optimale pour les descriptions de type flot de données.

Chapitre 8

Conclusions et perspectives

Sommaire

Dans ce chapitre, nous dressons tout d'abord le bilan de la méthode proposée dans le cadre de cette thèse. Nous donnons ensuite quelques perspectives de recherches qui peuvent être entreprises à la suite de nos travaux.

Dans la problématique sur la synthèse de haut niveau dans le domaine des systèmes intégrés, nous nous sommes intéressés à la synthèse des coprocesseurs dont le contrôle dépend des données.
Un coprocesseur est destiné à intégrer un système déjà défini. Il en résulte des contraintes fortes sur les caractéristiques du coprocesseur généré. Elles constituent les premiers objectifs de notre synthèse ciblée. Une fois ces contraintes respectées, la synthèse doit produire la solution la plus efficace.

8.1 Les objectifs atteints par la synthèse ciblée de haut niveau

8.1.1 Contraintes sur les caractéristiques du circuit

Les contraintes les plus contraignantes proviennent des communications entre le circuit à synthétisé et le système (cf. chapitre 2).
Les contraintes de communication sont des contraintes de synchronisation :

1. la *synchronisation logique* : le circuit doit communiquer sur les mêmes cycles que le système. Le circuit doit obéir à un protocole de communication.

2. la *synchronisation physique* : le circuit doit avoir une fréquence de fonctionnement adaptée au système. La fréquence d'horloge du circuit doit pouvoir être fixée.

Le deuxième type de contraintes sont des contraintes de performances du circuit. Le concepteur veut obtenir le meilleur rapport qualité/prix pour les critères fixés par le cahier des charges. Dans le cadre de notre thèse nous avons limité le nombre de critères qualifiant la qualité d'un circuit :

1. la surface du circuit.

2. le nombre de cycles du circuit pour l'exécution totale d'une tâche.

8.1.2 Le protocole de communication

Les protocoles de communication sont très nombreux. La description de ces protocoles est d'une granularité trop fine pour une description comportementale de haut niveau.

Le parti pris de notre méthode de synthèse est de ne pas respecter le protocole fixé par le système ! Le protocole automatiquement choisi par l'outil de synthèse ciblée se résume à l'attente d'un acquittement. Ce protocole est le plus simple possible.
L'utilisateur a la charge de concevoir un module d'interface pour les communications avec le système. Cette conception est simplifiée par la simplicité du protocole de communication du côté du circuit généré.

8.1.3 Respect de la fréquence d'horloge

Dans le flot de conception classique, c'est la synthèse RTL qui est tenue de respecter cette contrainte. Or cette dernière ne possède pas certains atouts pour respecter la fréquence.

- Si la synthèse RTL arrive à respecter la fréquence,
  - Si l'*allocation des opérateurs combinatoires* du chemin de données est satisfaisante. Le circuit est suffisamment simple pour être synthétisé efficacement en une phase d'*ordonnancement*.
  - Si l'*allocation des opérateurs combinatoires* du chemin de données n'est pas satisfaisante, alors il faut faire un *réordonnancement* (cf. chapitre 4).
- Si la synthèse RTL n'arrive pas à respecter la fréquence, alors il faut faire un *réordonnancement* (cf. chapitre 4).

Le chemin de données du circuit est synthétisé séparément de l'automate d'états. Les temps de propagation des signaux sont extraits du chemin de données.
Dans un environnement fonctionnelement asynchrone, des cycles d'attente peuvent être insérés dans l'automate sans en affecter sa fonctionnalité. La phase de *réordonnancement* utilise ces cycles pour laisser le temps aux signaux de se propager dans les chemins combinatoires du circuit.
L'automate modifié est ensuite synthétisé au niveau RTL puis placé et routé.

La synthèse expérimentale des différents circuits présentée dans le chapitre 7.5 a montré que, quel que soit le circuit synthétisé, il fonctionne toujours à la fréquence visée par le concepteur.
Le *réordonnancement* n'est pas imposé par notre outil. Si le concepteur le juge, il peut lui-même changer l'implication de la synthèse ciblée dans le flot de conception du circuit.

8.1.4 Couverture de l'espace des solutions

Un circuit possède un coût de production proportionnel à la surface de silicium qu'il occupe. Plus on augmente la vitesse d'exécution du circuit, plus sa surface augmente. En effet, l'augmentation de la vitesse passe par une parallélisation des opérations logicielles et donc un ajout d'opérateurs matériels. Le concepteur veut pouvoir choisir son rapport entre le coût et la performance. L'objectif de la synthèse est de laisser au concepteur le pouvoir de cibler une solution.

Dans le chapitre 5, nous avons vu qu'un moyen efficace pour cibler un circuit est de donner l'*affectation des registres* et *le nombre d'opérateurs combinatoires* du circuit. Ces directives utilisateurs peuvent être enrichies pour cibler encore plus précisément un circuit.

Ainsi le concepteur est capable de choisir les caractéristiques de son circuit. Il n'est concerné que par la partie "noble" de la conception, c'est à dire l'exploration architecturale.

8.1.5 Efficacité de la synthèse ciblée

Une fois que le concepteur a ciblé son circuit, il reste à l'outil de synthèse ciblée à produire la meilleure solution dans l'intervalle des solutions délimité par le concepteur.

Le concepteur a défini l'*affectation des registres* et le *nombre d'opérateurs combinatoires*, l'outil de synthèse se résume à la parallélisation maximale des opérations logicielles et à l'*affectation des opérateurs combinatoires*.

L'*affectation des registres* n'est pas remise en cause par la synthèse RTL. Cela implique des contraintes matérielles issues des registres sur l'ordonnancement des opérations logicielles.
L'algorithme présenté dans le chapitre 6 permet d'obtenir un *ordonnancement* efficace malgré les contraintes supplémentaires de précédence.

Les circuits générés par *ugh*, présentés dans le chapitre 7.2, possèdent des surfaces bien plus petites comparés aux circuits générés par *CoWare* et *gaut*.

Cependant, l'*affectation des registres* par le concepteur montre ses limites lorsqu'il s'agit d'effectuer la synthèse d'un circuit de type flot de données.
L'algorithme de la synthèse ciblée est rapide car l'*affectation des registres* est fournie par le concepteur.

La synthèse ciblée produit des résultats intéressants face à d'autres outils de synthèse de haut niveau généralistes ou spécialisés.

L'outil *ugh* donne une réponse efficace et rapide au problème de l'exploration d'architecture.
ugh a atteint les objectifs de la problématique :

- ✓ Les *points de Pareto* de l'outil sont proches de la limite théorique ("à la main").
- ✓ *ugh* couvre l'espace des solutions.
- ✓ Les circuits synthétisés s'intègrent facilement dans le système cible. *ugh* s'affranchit des contraintes de communication et de fréquence.
- ✓ *ugh* s'intègre bien et de différentes façons dans le flot de conception.
- ✓ *ugh* est rapide et supporte les gros modèles de circuit qu'ils soient de type *flot de données* ou bien mixtes.

8.2 Les perspectives

ugh se découpe en deux étapes successives : *l'ordonnancement* et le *réordonnancement*.

8.2.1 L'ordonnancement

Nous avons vu au chapitre 7.2.2, que la synthèse de flot de données produisait un circuit comprenant un nombre de cycles d'exécution plus faible (latence moins longue) que *ugh* lorsqu'il s'agissait d'un circuit de type flot de données. C'est l'*affectation des registres* qui impose à *ugh* des frontières de cycle entre certaines opérations qui auraient pu être parallélisées. Il faut que *ugh* puisse avoir un algorithme différent suivant les parties de la description comportementale du circuit qu'il est en train de traiter :

1. la partie flot de données doit être traitée avec un algorithme d'*affectation automatique des registres* tenant compte du *nombre de ressources* qu'a indiquées le concepteur.

2. le reste de la description doit être traité avec l'algorithme classique de *ugh* où les registres sont affectés et l'architecture ciblée par le concepteur.

Il y a plusieurs difficultés dans cette démarche :

- ➠ Détecter les parties de la description comportementale de type flot de données.
- ➠ Faire cohabiter les contraintes du concepteur avec celles plus floues de l'*affectation automatique des registres*.
- ➠ Maximiser l'utilisation des opérateurs pour paralléliser et gagner en vitesse d'exécution. Il faut alors déplacer les opérations d'un noeud du CDFG vers un autre, voire modifier la structure du CDFG.

8.2.2 Le réordonnancement

Le chapitre 7.4 donne deux types d'optimisation intéressants dans le flot de conception.

1. Soit la synthèse RTL transforme le chemin de données pour respecter la fréquence visée.
2. Soit le *réordonnancement* de *ugh* transforme l'automate pour respecter la fréquence visée.

De meilleurs résultats pourraient être obtenus en combinant ces deux optimisations (cf. figure 8.1). Les arcs pleins sont les transformations effectuées par la synthèse ciblée de haut niveau. Les arcs en pointillés sont les transformations effectuées par la synthèse logique.
Le chemin de données généré par la synthèse ciblée est une vue structurelle d'opérateurs car elle permet au *réordonnanceur* de changer les opérations de cycle.

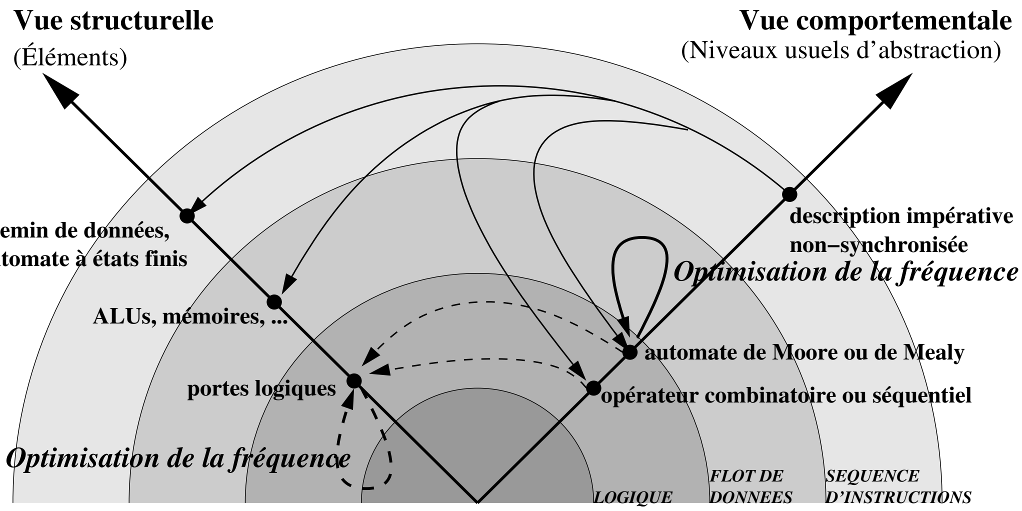


FIG. 8.1 Transformation de l'automate et du chemin de données.

Le choix de la technique d'optimisation pour respecter la fréquence se fait selon la criticité de l'opérateur.

- Si l'opérateur est critique, l'optimisation doit être faite sur la fonction combinatoire du chemin de données. L'insertion de cycles d'attente dans l'automate serait trop coûteuse en temps d'exécution. L'insertion d'un seul cycle dans l'automate entraîne une augmentation de n cycles

si l'opération est calculée n fois lors de l'exécution.
La synthèse logique doit ajouter de la logique pour optimiser les opérations critiques.

- Les opérateurs non critiques concernent des états de l'automate moins problématiques. Ils sont laissés au *réordonnanceur* de la synthèse ciblée de haut niveau qui analysera, après la synthèse logique, le chemin de données pour insérer des cycles d'attente dans l'automate.
  Il peut agir sur tous les états où la fréquence n'est pas respectée :
  - soit parce que la synthèse n'est pas arrivée à optimiser suffisamment le chemin de données,
  - soit parce que ce n'était pas une partie critique du circuit.

Les principales difficultés pour combiner les deux types d'optimisation de la fréquence sont :

- ➠ Pouvoir déterminer les opérateurs critiques du chemin de données. Il faut analyser le nombre d'états de l'automate utilisant l'opérateur et tenir compte des boucles.
- ➠ Déterminer le degré d'optimisation du chemin de données par la synthèse logique suivant la criticité des chemins combinatoires fonctionnels afin de ne pas trop augmenter la surface du circuit.
- ➠ Prendre en compte le degré d'optimisation du chemin de données souhaité par le concepteur.
- ➠ Transmettre les contraintes d'optimisation à la synthèse logique.

Table des figures

Glossaire

Affectation : association d'un opérateur à une opération fonctionnelle.

ALAP : (As Late As Possible) ordonnancement des opérations au cycle le plus tardif sans que cela change le nombre total de cycles par rapport à un ordonnancement ASAP.

Allocation : détermination du nombre d'opérateurs.

ALU : (Arithmetic and Logic Unit) opérateur arithmétique et logique.

ASAP : (As Soon As Possible) ordonnancement des opérations au cycle le plus tôt en ne tenant compte que des contraintes de précédence.

ASIC : (Application Specific Integrated Circuit) circuits dédiés à une application.

Assignation : cf. affectation.

Basic block : série d'opérations logicielles sans branchement conditionnel.

Binding : cf. affectation.

Bottom-up : approche de conception s'appuyant sur des circuits déjà existant pour construire un circuit plus complexe (à l'inverse cf. Top-down).

Branchement dépendant des données : branchement de l'automate dont le test dépend des valeurs des données.

Branchement cablé : branchement matérialisé par un seul état dans l'automate du circuit et des conditions supplémentaires mutuellement exclusives sur les signaux du chemin de données.

CAD : (Computer Aided Design) conception assistée par ordinateur.

CAO : Conception Assitée par Ordinateur.

CASS : (Cycle Accurate System Simulator) outil de simulation cycle précis.

CDFG : (Control Data Flow Graph) graphe exprimant les dépendances de données entre les opérations et les branchements conditionnels.

CFG : (Control Flow Graph) graphe exprimant les branchements conditionnels.

Chaîne longue : chemin combinatoire fonctionnel dont le temps de propagation du signal est le plus long du circuit.

Chemin combinatoire : parcours d'un signal électrique partant d'un registre ou d'un connecteur externe et allant à un registre ou un connecteur externe.

CISC : (Complex Instruction Set Component) jeu d'instructions complexes pouvant comprendre plusieurs opérations.

Codesign : cf. exploration architecturale.

Control dominated circuit : circuit où la complexité est dans l'automate.

Data dominated circuit : circuit où la complexité est dans le chemin de données.

Déroulement de boucle : (loop unrolling) chaque itération de la boucle est matérialisée par au moins un état distinct dans l'automate du circuit.

Design Reuse : réutilisation d'une partie ou de l'intégralité d'un circuit pour concevoir un autre circuit.

DFG : (Data Flow Graph) graphe exprimant les dépendances de données entre les opérations.

DP : (Data-Path) chemin de données du circuit.

Exploration architecturale : découpage en tâches parallélisables d'une application et détermination des tâches matérielles à synthétiser par le concepteur.

Faux chemin : chemin combinatoire non fonctionnel du circuit.

FPGA : (Field Programmable Gate Array) circuit programmable.

FSM : (Finite State Machine) automate d'états du circuit.

HLS : (High Level Synthesis) synthèse de haut niveau.

IP reuse : (Intelectual Property reuse) réutilisation d'un circuit existant pour une nouvelle application (cf. Bottom-up).

Langage impératif : langage d'assignations contenant des instructions de branchement conditionnel.

Latence : temps d'exécution du circuit nécessaire entre la réception de la première donnée et l'émission de la dernière donnée d'un traitement.

Fonctionnelement asynchrone : électriquement synchrone (même horloge) mais dont les communications ne s'effectuent pas obligatoirement dans les mêmes cycles.

Mobilité d'un opération : $cycle_{ALAP} - cycle_{ASAP}$ où $cycle_{ALAP}$ et $cycle_{ASAP}$ sont les cycles d'ordonnancement de l'opération.

Opération multi-cycles : (multi-cycling) fonction combinatoire exécutée sur plusieurs cycles.

Opérations chaînées : (chaining) opérations possédant des dépendances entre elles et s'effectuant au même cycle.

Opérations concourantes : opérations s'effectuant simultanément.

Opérations séquentielles : opérations s'effectuant consécutivement.

Ordonnancement : détermination du cycle auquel s'exécute une opération logicielle.

Pipeline : processus permettant de scinder un traitement en plusieurs étapes en insérant des registres intermédiaires.

Pipeline de boucle : cf. repliement de boucle.

Pipeline de données : insertion de registres dans le chemin de données.

Pipeline du contrôle : insertion de registres entre l'automate d'états et le chemin de données.

Point de Pareto : meilleure solution qu'un outil puisse produire sur les axes du référentiel.

Procrastination : technique consistant à "remettre à demain ce que l'on peut faire aujourd'hui même".

Repliement de boucle : (loop folding ou loop pipelining) certaines opérations de l'itération $n + 1$ sont effectuées en parallèle avec des opérations de l'itération n.

Retiming : déplacement des registres sans changer le comportement afin de diminuer les temps de propagation et la taille des registres.

RFG : (Register Flow Graph) graphe exprimant les dépendances de données et les dépendances matérielles de lecture/écriture sur les registres.

RISC : (Reduced Instruction Set Component) jeu d'instructions simples simplifiant les optimisations de la vitesse d'exécution.

RTL : (Register Transfer Level) niveau de description où l'affectation des registres est faite.

Scheduling : cf. ordonnancement.

Synthèse : transformation d'une description comportementale en une vue structurelle.

Synthèse architecturale : (architectural synthesis) synthèse d'un flot de données où la latence est constante car le traitement est indépendant des données.

Synthèse comportementale : (behavioral synthesis) synthèse de toute description comportementale mais souvent utilisée pour désigner la synthèse architecturale.

Synthèse logique : synthèse d'une vue structurelle en portes logiques à partir d'expressions booléennes et d'éléments mémorisants.

Synthèse de chemin de données : (data-path synthesis) cf. synthèse architecturale.

Synthèse de haut niveau : synthèse à partir d'une description comportementale non synchronisée. Le rôle de la synthèse de haut niveau est d'affecter les registres et d'extraire le parallélisme.

Synthèse RTL : synthèse à partir d'une description comportementale de niveau RTL.

Synthèse système : cf. exploration architecturale.

SoC : (System on Chip) système intégré.

Temps de cycle : temps maximal de propagation des signaux sur tous les chemins combinatoires fonctionnels du circuit.

Temps de propagation : temps nécessaire à un signal pour traverser un chemin combinatoire.

Top-down : approche de conception partant d'un niveau élevé d'abstraction et arrivant au niveau physique par étapes successives de raffinement (à l'inverse cf. Bottom-up).

UGH : (User Guided High level synthesis) outil de synthèse ciblée de haut niveau utilisé dans cette thèse pour obtenir des résultats expérimentaux.

VHDL : (Very high speed integrated circuit Hardware Description Language) langage de modélisation de circuit.

VLIW : (Very Large Instruction Word) jeu d'instructions consistant à faire exécuter plusieurs opérations en même temps sur un processeur superscalaire.

VLSI : (Very Large Scale Integration) circuits complexes comprenant un nombre important de transistors.

Bibliographie

[ADG+02] Ivan Augé, François Donnet, Pascal Gomez, Denis Hommais, and Frédéric Pétrot. Disydent : A pragmatic approach to the design of embedded systems. In *DATE02 Designers' Forum, University Booth Demonstrations*, page 255, `http://www-asim.lip6.fr/disydent`, mars 2002.

[Ber92] Elisabeth Berrebi. *Méthodologie pour l'application industrielle de la synthèse comportementale*. PhD thesis, Institut National Polytechnique de Grenoble, mars 1992.

[BKV+96] E. Berrebi, P. Kission, S. Vernalde, S. De Troch, J.C. Herluison, J. Fréhel, A.A. Jerraya, and I. Bolsens. Combined control flow dominated and data flow dominated high-level synthesis. In *33rd Design Automation Conference - DAC 96*, 1996.

[Cam91] R. Camposano. Path-based scheduling for synthesis. *IEEE Transactions on Computer Aided Design*, 10(1) :85–93, janvier 1991.

[Cam96] Raul Camposano. Behavioral synthesis. In *33rd Design Automation Conference*, 1996.

[CDM00] Roselyne Chotin, Yannick Dumonteix, and Habib Mehrez. Use of redundant arithmetic on architecture and design of a high performance dct macro-bloc generator. In *15th Design of Circuit and Integrated Systems Conference (DCIS)*, pages 428–433, Novembre 2000.

[CG96] En-Shou Chang and Daniel D. Gajski. A connection-oriented binding model for binding algorithms. Technical Report ICS-TR-96-49, 1996.

[CS97] Liang-Fang Chao and E. Hsing-Mean Sha. Scheduling data-flow graphs via retiming and unfolding. *IEEE Transactions on Parallel and Distributed Systems*, 8(12) :1259–1267, 1997.

[CSSJ99] Wander O. Cesário, Zoltan Sugár, Rodolphe Suescun, and Ahmed Amine Jerraya. Overlap and frontiers between behavioral and rtl synthesis. In *DATE'99 User's Forum*, 1999.

[CWM94] S. Chaudhuri, R. A. Walker, and J. E. Mitchell. Analysing and exploiting the structure of the constraints in the ilp approach to the scheduling problem. In *IEEE Transactions on VLSI*, pages 456–471, 1994.

[DT03] Mouhamadou Diaby and Mathieu Tuna. Réalisation d'un contrôleur réseau sur fpga. Technical report, septembre 2003.

[DTEB+02] H. Dawid, S. Thiel, H. Elders-Boll, M. Vaupel, E. Geesmann, M. Vellachi, and M. Antweiler. Efficient design flow from system level to hardware in cocentric system studio. In *Design, Automation and Test in Europe - DATE'02 Designers' Forum*, pages 3–7, 2002.

[Fau02] Etienne Faure. Jpeg : des spécifications comportementales au layout. Technical report, 2002.

[GDWL92] Daniel Gajski, Nikil Dutt, Allen Wu, and Steve Lin. *Introduction to Chip and System Design*, chapter 8, page 259. Kluwer Academic Publishers, 1992.

[HCLH90] C-Y. Huang, Y-S. Chen, Y-L. Lin, and Y-C. Hsu. Data path allocation based on bipartite weighted matching. In *27th Design Automation Conference*, pages 499–504, 1990.

[HD01] Ing-Jer Huang and Alvin Despain. Co-synthesis of pipeline structures and instruction reordering constraints for instruction set processors. In *ACM Transactions on Design Automation of Electronic Systems*, volume 6 of *1*, janvier 2001.

[HG85] Steven T. Healey and Daniel D. Gajski. Decomposition of logic networks into silicon. In *Proceedings of the 22nd ACM/IEEE conference on Design automation*, pages 162–168. ACM Press, 1985.

[Hil85] P.N. Hilfinger. A high-level language and silicon compiler for digital signal processing. In *IEEE Custom Integrated Circuits Conf.*, mai 1985.

[HP95] J. Hallberg and Z. Peng. Synthesis under local timing constraints in the camad high-level synthesis system. In *21st EUROMICRO Conference*, pages 650–656, 1995.

[Ins85] Institute of Electrical and Electronics Engineers. *IEEE Standard for Binary Floating-Point Arithmetic*, ieee std 754-1985 edition, 1985.

[Ins87] Institute of Electrical and Electronics Engineers. *IEEE Standard VHDL Language Reference Manual*, ansi/ieee std 1076-1987 edition, mars 1987.

[JCM99a] Christophe Jego, Emmanuel Casseau, and Eric Martin. Architectural synthesis of a complex application : the viterbi algorithm. In *Design, Automation and Test in Europe Conference, Date 99*, pages 69–73, Mars 1999.

[JCM99b] Christophe Jego, Emmanuel Casseau, and Eric Martin. *Architectural Synthesis with Interconnection Cost Control*, chapter VLSI : System on chip, pages ”509–520”. Kluwer Academic Publishers, Décembre 1999.

[JCM01] Christophe Jego, Emmanuel Casseau, and Eric Martin. Architectural synthesis of digital signal processing applications dedicated to submicron technologies. In *IEEE International Conference on Electronics Circuits and Systems, ICECS 01*, pages ”533–536”, Septembre 2001.

[Kar99] Karen Bartleson Karen. A new standard for system-level design. Open systemc white paper, Synopsys Inc., Septembre 1999.

[KDJ94] Polen Kission, Hong Ding, and Ahmed A. Jerraya. Structured design methodology for high-level design. In *31st ACM/IEEE Design Automation Conference*, 1994.

[KMM95] D. Knapp, D. Mac Millen, and R. Miller. Behavioral synthesis methodology for hdl-based specification and validation. In *32th ACM/IEEE Design Automation Conference - DAC'95*, juin 1995.

[KP87] F. J. Kurdahi and A. C. Parker. Real : A program for register allocation. In *Design Automation Conference DAC'87*, pages ”210–215”, juin 1987.

[LKMM95] T. Ly, D. Knapp, R. Miller, and D. Mac Millen. Scheduling using behavioral templates. In *32th ACM/IEEE Design Automation Conference - DAC'95*, juin 1995.

[Mic94a] Giovanni De Micheli. *Synthesis and Optimisation of Digital Circuit*, chapter 1, page 23. McGraw-Hill, 1994.

[Mic94b] Giovanni De Micheli. *Synthesis and Optimisation of Digital Circuit*, chapter 5, page 185. McGraw-Hill, 1994.

[MRS+01] Scott Mahlke, Rajiv Ravindran, Michael Schlansker, Robert Schreiber, and Timothy Sherwood. Bitwidth cognizant architecture synthesis of custom hardware accelerators. mars 2001.

[MSBSV91] S. Malik, E.M. Sentovich, R.K. Brayton, and A. Sangiovanni-Vincentelli. Retiming and resynthesis : optimizing sequential networks with combinational techniques. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 74–84, Janvier 1991.

[MSDP93] E. Martin, O. Sentieys, H. Dubois, and J.-L. Philippe. an architectural synthesis tool for dedicated signal processors. In *Proceedings EURO-DAC 93*, pages 14–19, 1993.

[MSS+99] I. Moussa, Z. Sugár, R. Suescun, A.a. Jerraya, M. Diaz-Nava, M. Pavesi, S. Crudo, and L. Gazzi. Comparing rtl and behavioral design methodologies in the case of a 2m transistors atm shaper. In *Design Automation Conference - DAC'99*, 1999.

[Mul97] J-M Muller. *Elementary Functions, Algorithms and Implementation*, chapter 2, page 13. Birkhauser Boston, 1997.

[NCG91] S. Note, F. Catthoor, and G. Goosens. Cathedral-iii architecture-driven high-level synthesis for high throughput dsp applications. In *28th ACM/IEEE Design Automation Conference - DAC 91*, 1991.

[OJ92] Seong Yong Ohm and Chu Shik Jhon. A branch-and-bound method for the optimal scheduling. In *CICC'92*, mai 1992.

[Pan88] B. Pangrle. Splicer : A heuristic approach to connectivity binding. In *25th ACM/IEEE DAC*, pages 536–541, 1988.

[PG87] Barry Michael Pangrle and Daniel D. Gajski. Design tools for intelligent silicon compilation. *IEEE Transactions on Computer Aided Design*, 6(6) :1098–1112, novembre 1987.

[PHCT01] Gilles Pelissier, Richard Hersemeule, Gaston Cambon, and Lionel Torres. How to accelerate c to vhdl design flows. In *Designer's Forum, DATE'01*, pages 114–116, March 2001.

[Phi92] W. J. M. Philipsen. Data flow scheduling using potts neurons. In *Workshop on Circuits, Systems and Signal Processing - CSSP'92*, 1992.

[PK89] Pierre G. Paulin and John P. Knight. Algorithms for high-level synthesis. *IEEE Design and Test*, pages 18–31, décembre 1989.

[PKL88] Z. Peng, K. Kuchcinski, and B. Lyles. *CAMAD : A Unified Data Path/Control Synthesis Environment*, pages 53–67. North-Holland, 1988.

[RCN03] Jan Rabaey, Anantha Chandrakasan, and Borivoje Nikilić. *Digital Integrated Circuits, A design perspective 2nd edition*, chapter Chapitre 4 - The Wire. Prentice hall, 2003.

[RD95] B. Rouzeyre and D. Dupont. *Behavioral synthesis : control schemes in question*, pages 223–229. Chapman and Hall, 1995.

[RKG+92] C. Ramachandran, F. J. Kurdahi, D. D. Gajski, A. C.-H. Wu, and V. Chaiyakul. Accurate layout area and delay modeling for system level design. In *Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, pages 355–361. IEEE Computer Society Press, 1992.

[RS91] B. Rouzeyre and G. Sagnes. A new method for the minimization of memory area in high level synthesis. In *EURO-ASIC 91*, pages 184–189, 1991.

[SDPC95] O. Sentieys, J.Ph. Diguet, J.-L. Philippe, and D. Chillet. Gaut : A high level synthesis tool, dedicated to real time signal processing application. In *EURO-DAC 95*, Septembre 1995.

[SMT+92] J. Septién, D. Mozos, F. Tirado, R. Hermida, and M. Fernández. Heuristics for branch-and-bound global allocation. In *European Design Automation Conference*, pages 334–340, 1992.

[Sug00] Zoltán Sugár. *Synthèse comportementale basée sur l'ordonnancement*. PhD thesis, Institut National Polytechnique de Grenoble, mai 2000. pages 75-77.

[Sys] SystemC Open Source. SystemC Version 2.0.1 User's Guide.

[TS86] C. J. Tseng and D. P. Sieworek. Automatic synthesis of data path on digital system. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 379–395, juillet 1986.

[WC95] R. A. Walker and S. Chaudhuri. High level synthesis : Introduction to the scheduling problem. In *IEEE Design and Test of Computers*, pages 60–69, 1995.

[WM90] Kernighan Brian W. and Ritchie Dennis M. *Le langage C*. Masson, 1990.

Annexe A

Langages de description comportementale utilisés

Plusieurs outils de synthèse de haut niveau ont été utilisés dans cette thèse afin de comparer les résultats expérimentaux. Ces outils possèdent des langages de description comportementale différents. Cependant les sous-ensembles du *C*, du *C++* et du *VHDL* acceptés par ces outils sont assez proches sémantiquement.

Le but de ce chapitre est de présenter les descriptions comportementales utilisées pour l'évaluation chiffrée des outils de synthèse. Ces descriptions montrent la complexité des applications visées par la synthèse de haut niveau.
Hormis l'aspect syntaxique propre à chaque langage, nous nous sommes efforcés de donner la même description comportementale pour chaque outil. Le haut niveau de description de chaque langage rend souvent possible la traduction littérale.

A.1 Le langage utilisé par *ugh*

Bien que notre outil de synthèse ciblée *ugh* accepte aussi du *VHDL* en entrée, le langage qui a été utilisé pour les mesures expérimentales, est un sous-ensemble du langage *C*.

Le fichier *ugh.h*, inclus dans chaque description comportementale, définit les types et les fonctions synthétisables :

- les types des ports du circuit (le nombre de bits et le sens).
- les types des variables (le nombre de bits et la présence du signe).
- les fonctions d'accès au ports.
- les fonctions arithmétiques spécifiques.

Le sous-ensemble *C* ne comprend pas, entre autres :

- la récursivité.
- l'allocation dynamique de la mémoire.

- les opérations arithmétiques complexes : division, puissance, racine, ...
- l'effet de bord dans une expression booléenne (exemple : *a && (b=1)*).

Description comportementale *C* du vld du Mpeg

```
#include "ugh.h"

#define BUF_SZ          32
#define VLD_EOB         0xFF
#define DECODE_ERROR    0x0

/* Defines for the function to perform (fct) */
#define UGH_getmpegintramblock 0
#define UGH_getmpegintermblock 1
#define UGH_getmpg2intramblock 2
#define UGH_getmpg2intermblock 3

/* some cut-short to control register */
#define intra() (control & 0x1)
#define func()  ((control>>7) & 0x3)

/* The bits returned by CAT_RIGHT are
   in the higest part of the word */
#define buf_showbits(nb)  \
     ((ugh_catshr(32,32,buf_r_l, buf_r_th, buf_r_h) \
      >> (BUF_SZ - (nb)))&((1<<nb)-1))

/*** tasks communication ***/
extern ugh_inChannel32  data_dn_In;
extern ugh_inChannel32  control_in;
extern ugh_outChannel32  data_out;
extern ugh_outChannel32  control_out;


/**** registres globaux ***/
uint9 control;  /*i.e ctl_ctl*/
uint14 skipcount;
uint7 dec_cbp;

uint16 ctlNbIn;
uint16 ctlNbDataWordRead;
uint16 ctlNbOut;

uint6  buf_r_th, buf_r_tl;
uint32 buf_r_h, buf_r_l;
/* Parameter of the flushbits and buf_valid functions */
uint6  buf_out_sz;


typedef struct {
  uint7 run;
  uint6 level;
  uint6 len;
} DCTtab;

DCTtab dec_value;
uint4 dec_comp;
uint1 dec_fault;
uint1 dec_Ieq0;


static const DCTtab dec_rom[512] = {
/*    0 0 00xx xxxx :  tab0a : 256  252  :      9      :     0*/
  {0,0,0}, {0,0,0}, {0, 0,0}, {0, 0,0}, /*added blanks*/
  {65,0,6}, {65,0,6}, {65,0,6}, {65,0,6}, /* Escape */
  {7,1,7}, {7,1,7}, {8,1,7}, {8,1,7},
  {6,1,7}, {6,1,7}, {2,2,7}, {2,2,7},
  {0,7,6}, {0,7,6}, {0,7,6}, {0,7,6},
  {0,6,6}, {0,6,6}, {0,6,6}, {0,6,6},
  {4,1,6}, {4,1,6}, {4,1,6}, {4,1,6},
  {5,1,6}, {5,1,6}, {5,1,6}, {5,1,6},
  {1,5,8}, {11,1,8}, {0,11,8}, {0,10,8},
  {13,1,8}, {12,1,8}, {3,2,8}, {1,4,8},
  {2,1,5}, {2,1,5}, {2,1,5}, {2,1,5},
  {2,1,5}, {2,1,5}, {2,1,5}, {2,1,5},
  {1,2,5}, {1,2,5}, {1,2,5}, {1,2,5},
  {1,2,5}, {1,2,5}, {1,2,5}, {1,2,5},
  {3,1,5}, {3,1,5}, {3,1,5}, {3,1,5},
  {3,1,5}, {3,1,5}, {3,1,5}, {3,1,5},
  {1,1,3}, {1,1,3}, {1,1,3}, {1,1,3},
  {1,1,3}, {1,1,3}, {1,1,3}, {1,1,3},
  {1,1,3}, {1,1,3}, {1,1,3}, {1,1,3},
  {1,1,3}, {1,1,3}, {1,1,3}, {1,1,3},
  {1,1,3}, {1,1,3}, {1,1,3}, {1,1,3},
  {1,1,3}, {1,1,3}, {1,1,3}, {1,1,3},
  {1,1,3}, {1,1,3}, {1,1,3}, {1,1,3},
  {1,1,3}, {1,1,3}, {1,1,3}, {1,1,3},
  {64,0,4}, {64,0,4}, {64,0,4}, {64,0,4}, /* EOB */
  {64,0,4}, {64,0,4}, {64,0,4}, {64,0,4},
  {64,0,4}, {64,0,4}, {64,0,4}, {64,0,4},
  {64,0,4}, {64,0,4}, {64,0,4}, {64,0,4},
  {0,3,4}, {0,3,4}, {0,3,4}, {0,3,4},
  {0,3,4}, {0,3,4}, {0,3,4}, {0,3,4},
  {0,3,4}, {0,3,4}, {0,3,4}, {0,3,4},
  {0,3,4}, {0,3,4}, {0,3,4}, {0,3,4},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,2,3}, {0,2,3}, {0,2,3}, {0,2,3},
  {0,2,3}, {0,2,3}, {0,2,3}, {0,2,3},
  {0,2,3}, {0,2,3}, {0,2,3}, {0,2,3},
  {0,2,3}, {0,2,3}, {0,2,3}, {0,2,3},
  {0,2,3}, {0,2,3}, {0,2,3}, {0,2,3},
  {0,2,3}, {0,2,3}, {0,2,3}, {0,2,3},
  {0,2,3}, {0,2,3}, {0,2,3}, {0,2,3},
  {0,2,3}, {0,2,3}, {0,2,3}, {0,2,3},
  {0,4,5}, {0,4,5}, {0,4,5}, {0,4,5},
  {0,4,5}, {0,4,5}, {0,4,5}, {0,4,5},
  {0,5,5}, {0,5,5}, {0,5,5}, {0,5,5},
  {0,5,5}, {0,5,5}, {0,5,5}, {0,5,5},
  {9,1,7}, {9,1,7}, {1,3,7}, {1,3,7},
  {10,1,7}, {10,1,7}, {0,8,7}, {0,8,7},
  {0,9,7}, {0,9,7}, {0,12,8}, {0,13,8},
  {2,3,8}, {4,2,8}, {0,14,8}, {0,15,8},
  /*4*63 + 4 = 256 */
/*tab0:  256 1 00xx xxxx :  tab0  :  64   60  :    1   :       0*/
  {0,0,0}, {0,0,0}, {0, 0,0}, {0, 0,0}, /*added blanks*/
  {65,0,6}, {65,0,6}, {65,0,6}, {65,0,6}, /* Escape */
  {2,2,7}, {2,2,7}, {9,1,7}, {9,1,7},
  {0,4,7}, {0,4,7}, {8,1,7}, {8,1,7},
  {7,1,6}, {7,1,6}, {7,1,6}, {7,1,6},
  {6,1,6}, {6,1,6}, {6,1,6}, {6,1,6},
  {1,2,6}, {1,2,6}, {1,2,6}, {1,2,6},
  {5,1,6}, {5,1,6}, {5,1,6}, {5,1,6},
  {13,1,8}, {0,6,8}, {12,1,8}, {11,1,8},
  {3,2,8}, {1,3,8}, {0,5,8}, {10,1,8},
  {0,3,5}, {0,3,5}, {0,3,5}, {0,3,5},
  {0,3,5}, {0,3,5}, {0,3,5}, {0,3,5},
  {4,1,5}, {4,1,5}, {4,1,5}, {4,1,5},
  {4,1,5}, {4,1,5}, {4,1,5}, {4,1,5},
  {3,1,5}, {3,1,5}, {3,1,5}, {3,1,5},
  {3,1,5}, {3,1,5}, {3,1,5}, {3,1,5},
  /* 256 +15 * 4 + 4 = 320*/
 /*  320 1 0100 xxxx :  tabN  :  16   12  :      0     :     0*/
  {0,0,0}, {0,0,0}, {0, 0,0}, {0, 0,0}, /*added blanks*/
  {0,2,4},  {2,1,4},  {1,1,3},  {1,1,3},
  {64,0,2}, {64,0,2}, {64,0,2}, {64,0,2}, /* EOB */
  {0,1,2},  {0,1,2},  {0,1,2},  {0,1,2},
  /* 320 + 3 *4 +4 = 336 */
 /*  336 1 0101 0xxx :  tab1a :   8    8  :      8     :     0 */
  {5,2,9}, {5,2,9}, {14,1,9}, {14,1,9},
  {2,4,10}, {16,1,10}, {15,1,9}, {15,1,9},

  /* 336 + 2*4 + 0 =344 */
 /*  344 1 0101 1xxx :  tab1  :   8    8  :      2     :     0 */
  {16,1,10}, {5,2,10}, {0,7,10}, {2,3,10},
  {1,4,10}, {15,1,10}, {14,1,10}, {4,2,10},
  /* 344 + 2*4 + 0 =352 */
 /*  352 1 0110 xxxx :  tab2  :  16   16  :      3     :     0 */
  {0,11,12}, {8,2,12}, {4,3,12}, {0,10,12},
  {2,4,12}, {7,2,12}, {21,1,12}, {20,1,12},
  {0,9,12}, {19,1,12}, {18,1,12}, {1,5,12},
  {3,3,12}, {0,8,12}, {6,2,12}, {17,1,12},
  /* 352 + 16 + 0 =368 */
 /*  368 1 0111 xxxx :  tab3  :  16   16  :      4     :     0 */
  {10,2,13}, {9,2,13}, {5,3,13}, {3,4,13},
```

```
  {2,5,13}, {1,7,13}, {1,6,13}, {0,15,13},
  {0,14,13}, {0,13,13}, {0,12,13}, {26,1,13},
  {25,1,13}, {24,1,13}, {23,1,13}, {22,1,13},
/* 368 +16=384*/
 /*  384 1 1000 xxxx :  tab4  :  16   16  :      5     :     0 */
  {0,31,14}, {0,30,14}, {0,29,14}, {0,28,14},
  {0,27,14}, {0,26,14}, {0,25,14}, {0,24,14},
  {0,23,14}, {0,22,14}, {0,21,14}, {0,20,14},
  {0,19,14}, {0,18,14}, {0,17,14}, {0,16,14},
 /*  400 1 1001 xxxx :  tab5  :  16   16  :      6     :     0 */
  {0,40,15}, {0,39,15}, {0,38,15}, {0,37,15},
  {0,36,15}, {0,35,15}, {0,34,15}, {0,33,15},
  {0,32,15}, {1,14,15}, {1,13,15}, {1,12,15},
  {1,11,15}, {1,10,15}, {1,9,15}, {1,8,15},
 /*  416 1 1010 xxxx :  tab6  :  16   16  :      7     :     0 */
  {1,18,16}, {1,17,16}, {1,16,16}, {1,15,16},
  {6,3,16}, {16,2,16}, {15,2,16}, {14,2,16},
  {13,2,16}, {12,2,16}, {11,2,16}, {31,1,16},
  {30,1,16}, {29,1,16}, {28,1,16}, {27,1,16},
 /*  432 1 1011 xxxx :  tabF  :  16   12  :     10     :     0 */
  {0,0,0}, {0,0,0}, {0, 0,0}, {0, 0,0}, /*added blanks*/
  {0,2,4}, {2,1,4}, {1,1,3}, {1,1,3},
  {0,1,1}, {0,1,1}, {0,1,1}, {0,1,1},
  {0,1,1}, {0,1,1}, {0,1,1}, {0,1,1}
 /*  448 1           :        :           :    XX      :     1 */
};


typedef struct {
  int8 val;
  uint4 len;
} VLCtab;

uint1 lc_state;

const VLCtab lc_rom[128] = {
/* Table B-12, dct_dc_size_luminance,
   codes 00xxx ... 11110 */
/*static VLCtab DClumtab0[32] =*/
  {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2},
  {2, 2}, {2, 2}, {2, 2}, {2, 2}, {2, 2}, {2, 2}, {2, 2}, {2, 2},
  {0, 3}, {0, 3}, {0, 3}, {0, 3}, {3, 3}, {3, 3}, {3, 3}, {3, 3},
  {4, 3}, {4, 3}, {4, 3}, {4, 3}, {5, 4}, {5, 4}, {6, 5},
  {DECODE_ERROR, 0},


/* Table B-12, dct_dc_size_luminance,
   codes 111110xxx ... 111111111 */
/*static VLCtab DClumtab1[16] =*/
  {7, 6}, {7, 6}, {7, 6}, {7, 6}, {7, 6}, {7, 6}, {7, 6}, {7, 6},
  {8, 7}, {8, 7}, {8, 7}, {8, 7}, {9, 8}, {9, 8}, {10,9}, {11,9},
  {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0},
  {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0},

/* Table B-13, dct_dc_size_chrominance,
   codes 00xxx ... 11110 */
/*static VLCtab DCchromtab0[32] =*/
  {0, 2}, {0, 2}, {0, 2}, {0, 2}, {0, 2}, {0, 2}, {0, 2}, {0, 2},
  {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2},
  {2, 2}, {2, 2}, {2, 2}, {2, 2}, {2, 2}, {2, 2}, {2, 2}, {2, 2},
  {3, 3}, {3, 3}, {3, 3}, {3, 3}, {4, 4}, {4, 4}, {5, 5},
  {DECODE_ERROR, 0},

/* Table B-13, dct_dc_size_chrominance,
   codes 111110xxxx ... 1111111111 */
/*static VLCtab DCchromtab1[32] =*/
  {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6},
  {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6},
  {7, 7}, {7, 7}, {7, 7}, {7, 7}, {7, 7}, {7, 7}, {7, 7}, {7, 7},
  {8, 8}, {8, 8}, {8, 8}, {8, 8}, {9, 9}, {9, 9}, {10,10}, {11,10}
};


inline void buf_init(void)
{
   buf_r_th = 0;
   buf_r_tl = 0;
}

inline uint1 buf_valid(void)
{
   return buf_out_sz < (buf_r_th + buf_r_tl);
}

inline void buf_load(void)
{
   buf_r_tl = BUF_SZ;
   buf_r_h  = buf_r_l;
   ugh_read(data_in, &buf_r_l);
   ctlNbDataWordRead += 1;
}

inline void buf_flushbits(void)
{
   if (buf_r_th <= buf_out_sz ) {
      buf_r_th = buf_r_th + buf_r_tl - buf_out_sz;
      buf_r_tl = 0;
      buf_r_h  = buf_r_l;
   }
   else {
      buf_r_th = buf_r_th - buf_out_sz;
   }

   ctlNbIn += buf_out_sz;
}



inline void dec_calcaddr(void)
{
   uint1  error=0;
   uint9  addr;
   /* SYNTHESIS: dec_in vanishes */
   uint12 dec_in = buf_showbits(12);


   if ((dec_in & 0xFFF) == 0 ) { /* EQUIVALENT: dec_in <= 0xF */
      error = 1;
   } else if ((dec_in & 0xFFE) == 0) {
      error = 0;
    addr = 416 | (buf_showbits(16) & 0xF); /* Tab6 */
   } else if ((dec_in & 0xFFC) == 0) {
      error = 0;
      addr = 400 | (buf_showbits(15) & 0xF); /* Tab5 */
   } else if ((dec_in & 0xFF8) == 0) {
      error = 0;
      addr = 384 | (buf_showbits(14) & 0xF); /* Tab4 */
   } else if ((dec_in & 0xFF0) == 0) {
      error = 0;
      addr = 368 | (buf_showbits(13) & 0xF); /* Tab3 */
   } else if ((dec_in & 0xFE0) == 0) {
      error = 0;
      addr = 352 | (buf_showbits(12) & 0xF); /* Tab2 */
   } else if ((dec_in & 0xFC0) == 0) {
      if (intra()==0) {
         addr = 344 | (buf_showbits(10) & 0x7); /* Tab1 */
      } else {
         addr = 336 | (buf_showbits(10) & 0x7); /* Tab1a */
      }
   } else if ((dec_in & 0xC00) == 0) {
      if (intra()==0) {
         addr = 256 | (buf_showbits(8) & 0x3F); /* Tab0 */
      } else {
         addr = (buf_showbits(8) & 0xFF); /* Tab0a */
      }
   } else {
    if (func()==UGH_getmpg2intramblock) {
      if (intra()!=0) {
           addr = (buf_showbits(8) & 0xFF); /* Tab0a */
         } else {
        addr = (0x140 | (dec_in >> 8)); /* TabN */
           /* EQUIVALENT:
              addr = 320 | (dec_in >> 8));
              addr = 320 | (buf_showbits(4));
              addr = 320 + (dec_in >> 8));
              addr = 320 + (buf_showbits(4));
           */
         }
    } else {
      if (dec_Ieq0!=0) {
           addr = (432 | (dec_in >> 8)); /* TabF */
         } else {
           addr = (0x140 | (dec_in >> 8)); /* TabN */
           /* EQUIVALENT:
              addr = 320 | (dec_in >> 8));
              addr = 320 | (buf_showbits(4));
              addr = 320 + (dec_in >> 8));
              addr = 320 + (buf_showbits(4));
           */
         }
    }
   }

   dec_value = dec_rom[addr];
   dec_fault = error;
   dec_Ieq0  = 0;
}


/**** functions:                                                  ****/
/**** - set_lum():                                                ****/
/****   pre :                                                     ****/
/****   post: configue the bloc to write luminence.               ****/
/**** - set_chrom():                                              ****/
```

```
/****   pre :                                               ****/
/****   post: configue the bloc to write luminence.         ****/
/**** - write_lc():                                        ****/
/****   pre : need 26 bits in input buffer.                 ****/
/****   post: write either luminence or chrominence depending ****/
/****       on bloc state.                                  ****/

inline void lc_setlum()
{
  lc_state = 0;
}

inline void lc_setchrom()
{
  lc_state = 1;
}

inline void lc_write()
{
  uint5 code5;
  uint7 code7;
  uint8 size;
  int32 val;

  code5 = buf_showbits(5);

  if ( code5 < 31 ) {
    code7 = code5 | (lc_state<<6);
  } else if ( lc_state == 1 ) { /*chrom*/
    /* EQUIVALENT: code7 = (1<<6) | (buf_showbits(10) & 0x1e) */
    code7 = ((1<<6) | (buf_showbits(10) - 0x3e0)) + 32;
  } else  { /*lum*/
    /* EQUIVALENT: code7 = buf_showbits(9) & 0x1e0 */
    code7 =  buf_showbits(9) - 0x1f0 + 32;
  }

  size = lc_rom[code7].val;

  /*flush des n premiers bits*/
  buf_out_sz = lc_rom[code7].len;
  buf_flushbits();

  if ( size == 0 ) {
    val = 0;
  } else {
    val = (buf_showbits(11) << 5) & 0xffe0;
    val = (size<<16) | val;
    /*flush de n bits*/
    buf_out_sz = size;
    buf_flushbits();
  }

  /* write a data word and count the number of words sent */
  ugh_write(data_out,&val);
  ctlNbOut++;
}



/*** saute les bits invalides dans la fifo data_in ***/

inline static void Tblock_skip()
{
   uint32 tmp;

   ugh_read(control_in, &tmp);

   skipcount = ((tmp  >> 16) & 0x7FFF) * 32 + (tmp & 0xFFFF);

   buf_out_sz = 32;
   while (skipcount >= 32) {
    if (buf_valid()==0) {
      buf_load();
    }

    buf_flushbits();
    skipcount -= 32;
   }

   buf_out_sz = skipcount;

   if (buf_valid()==0) {
      buf_load();
   }

   buf_flushbits();
}

/*rom*/
int1 dec_comp_sup( uint4 key )
{
  switch (key)
  {
    case 4: case 5: return 1;
    default: return 0;
  }
}

/*rom*/
int1 dec_comp_rom( uint4 key )
{
  switch (key)
  {
    case 0: case 2: case 3: case 1: return 1;
    default: return 0;
  }
}

/*** entry point of vld task ***/
void *ugh_main(void *x)
{
  uint32 tmp;
  uint1 tmp_wired_8;

  /*init  */
  buf_init();

  while(1) {
    ctlNbIn = 0;
    ctlNbDataWordRead=0;
    ctlNbOut = 0;

    /*read fifo control*/
    ugh_read(control_in, &tmp);

    dec_comp  = 0;
    dec_fault = 0;
    control   = tmp;
    dec_cbp   = (control>>1) & 0x3f;

    /*saute les bits invalides dans la fifo data_in*/
    Tblock_skip();

    /*decodage*/
    while (!( dec_comp == 6 || dec_fault == 1 )) {
      buf_out_sz = 17;

      /* MPEG or MPEG 2 */
      if ( func()==UGH_getmpegintermblock
        || func()==UGH_getmpg2intermblock ) {
        ++dec_comp;
        dec_cbp *= 2;
        /* dec_bloc_end = 0; */
        dec_Ieq0 = 1;
      }
      else if ( ( func()==UGH_getmpegintramblock ||
                  func()==UGH_getmpg2intramblock )
             && buf_valid()!=0  && dec_comp_rom(dec_comp) ) {
        /* && data_out non pleine ssi write remplace par des write*/
        lc_setlum();
        lc_write();
        ++dec_comp;
        dec_cbp *= 2;
        /* dec_bloc_end = 0; */
        dec_Ieq0 = 1;
      }
      else if ( ( func() == UGH_getmpegintramblock ||
                  func() == UGH_getmpg2intramblock )
         && buf_valid()!=0 && dec_comp_sup(dec_comp) ) {
        /* && data_out non pleine ssi write remplace par des write*/
        lc_setchrom();
        lc_write();
        ++dec_comp;
        dec_cbp *= 2;
        /* dec_bloc_end = 0; */
        dec_Ieq0 = 1;
      }
      else {
        buf_load();
        continue;
      }

      /* d_coef_loop */
      while (dec_fault==0) {
        uint1 cbp_do= (dec_cbp>>6) & 1;
        buf_out_sz = 17;

        if ( cbp_do == 0 ) {
          break;
        }
        else if ( buf_valid()!=0 ) {
          /*load dec_value, dec_fault */
          dec_calcaddr();
```

```
    if ( dec_fault == 1 ) break;

    buf_out_sz = dec_value.len;
    buf_flushbits();

    if ( dec_value.run == 64 ) {
      tmp =  (uint32) (VLD_EOB << 16);
      /* write a data word
         and count the number of words sent */
      ugh_write(data_out,&tmp);
      ctlNbOut++;
      break;
    }
    else if ( dec_value.run == 65) {
      uint18 val;
      buf_out_sz = 18;
      if (buf_valid()==0) {
        buf_load();
      }

      val = buf_showbits(18);

      buf_out_sz = 18;
      buf_flushbits();


      if ( (val&0xFFF) ==0 ) {
        dec_fault = 1;
        break;
      }

      tmp= (val&0xfff) | ((val&0x800)<<1) | ((val&0x800)<<2) |
         ((val&0x800)<<3) |((val&0x800)<<4)  |
         (( (val&0x3f000)>>12 )<<16);

      /* write a data word
         and count the number of words sent */
          ugh_write(data_out,&tmp);
          ctlNbOut++;
        }
        else {
          if (buf_showbits(1) == 1 ) {
            tmp= (dec_value.run << 16)
                 | ((0-dec_value.level)&0xffff);
          }
          else {
            tmp= (dec_value.run << 16) | (dec_value.level&0xffff);
          }

          /* write a data word and count the number of words sent */
          ugh_write(data_out,&tmp);
          ctlNbOut++;

          buf_out_sz = 1;
          buf_flushbits();
        }
      }
      else if ( buf_valid()==0 ) {
        buf_load();
      }
    }

  }/*fin decodage*/

  tmp = (dec_fault<<31) | ((ctlNbIn&0x7fff)<<16)
                        | (ctlNbDataWordRead*32);
  ugh_write(control_out, &tmp);

  tmp = (dec_fault<<31) | ((ctlNbIn&0x7fff)<<16) | ctlNbOut;
  ugh_write(control_out, &tmp);

 }  /*while(1)*/

}
```

A.2 Le langage utilisé par *CoCentric/CoWare*

Le langage accepté par *CoWare* est un sous-ensemble de *SystemC*. C'est un langage objet qui redéfinit tous les opérateurs et les variables des expressions du *C++*.

Le fichier *systemc.h*, inclus dans chaque description, définit des types et des fonctions :

- le type modèle pour tous les circuits.
- les types des ports du circuit (le nombre de bits et le sens).
- les types des variables (le nombre de bits et la présence du signe).
- les fonctions d'accès au ports.
- les opérations arithmétiques et logiques.

Le sous-ensemble de *SystemC* possède les mêmes restrictions que le sous-ensemble du langage *C* de *ugh*. Il limite aussi le nombre de types synthétisables et n'autorise pas l'héritage.

Description comportementale *SystemC* du vld du Mpeg

```
#include <systemc.h>

#define BUF_SZ          32
#define ERROR           0
#define VLD_EOB         0xFF
/* Defines for the function to perform (fct) */
#define UGH_getmpegintramblock 0
#define UGH_getmpegintermblock 1
#define UGH_getmpg2intramblock 2
#define UGH_getmpg2intermblock 3

/* The bits returned by CAT_RIGHT are
   in the higest part of the word */
#define buf_showbits(nb) \
   (((sc_uint<32>)(ugh_catshr(32,32,buf_r_l, buf_r_th, buf_r_h) \
    >> ((sc_uint<32>) (BUF_SZ - (nb)))))&((1<<nb)-1))

#define ugh_catshr(Isz, SINsz, i, sh, sin) \
   ((sc_uint<32>)(sin << ((sc_uint<32>)(32 - sh))) \
    | ((sc_uint<32>)i >> sh))

/* some cut-short to control register */
#define intra() (control & 0x1)
#define cbp()   ((control>>1) & 0x3f)
#define func()  ((control>>7) & 0x3)

/* write a data word and count the number of words sent */
#define DataWrite(x,y,z) {ugh_write(x,y); ++ctlNbOut;}
```

```
#define ugh_read( fifo, var ) \
   { \
      do \
      { \
         fifo##_read.write( true ); \
         (var) = fifo##.read(); \
      } \
      while ( fifo##_rok.read() == false ); \
      fifo##_read.write( false ); \
      wait(); \
   }

#define ugh_write( fifo, var ) \
   { \
      do \
      {  \
         fifo##_write.write( true ); \
         fifo##.write( (sc_uint<32>)(var) );  \
      } \
      while ( fifo##_wok.read() == false ); \
      fifo##_write.write( false ); \
      wait(); \
   }

typedef struct {
  sc_uint<7> run;
  sc_uint<6> level;
  sc_uint<6> len;
} DCTtab;

typedef struct {
  sc_uint<8> val;
  sc_uint<4> len;
} VLCtab;



SC_MODULE(tblock) {
  sc_in< bool >         abort;
  sc_in< bool >         start;
  sc_in< bool >         reset;
  sc_out< bool >        cmd_dn_In_read;
  sc_in< bool >         cmd_dn_In_rok;
  sc_in< sc_uint<32> >  cmd_dn_In;
  sc_out< bool >        data_dn_In_read;
  sc_in< bool >         data_dn_In_rok;
  sc_in< sc_uint<32> >  data_dn_In;
  sc_out< bool >        data_up_Out_write;
  sc_in< bool >         data_up_Out_wok;
  sc_out< sc_uint<32> > data_up_Out;
  sc_out< bool >        cmd_up_Out_write;
  sc_in< bool >         cmd_up_Out_wok;
  sc_out< sc_uint<32> > cmd_up_Out;
  sc_in_clk ck;

  // Declare internal variables and signals
  /*i.e ctl_ctl*/
  sc_uint<9> control;
  sc_uint<14> skipcount;
  sc_uint<7> dec_cbp;

  sc_uint<16> ctlNbIn;
  sc_uint<16> ctlNbDataWordRead;
  sc_uint<16> ctlNbOut;

  sc_uint<6>  buf_r_th, buf_r_tl;
  sc_uint<32> buf_r_h, buf_r_l;
  /* Parameter of the flushbits and buf_valid functions */
  sc_uint<6>  buf_out_sz;

  DCTtab dec_value;
  sc_uint<4> dec_comp;
  sc_uint<1> dec_fault;
  sc_uint<1> dec_Ieq0;

  sc_uint<1> lc_state;


  // Declare processes in the module
  void ugh_main();

  //other func
  inline void buf_init();
  inline void buf_load();
  inline void buf_flushbits();
  inline void dec_calcaddr();
  inline void lc_write();
  inline void lc_setlum();
  inline void lc_setchrom();
  inline void Tblock_skip();
  inline sc_int<1> dec_comp_sup( sc_uint<4> param );
  inline sc_int<1> dec_comp_rom( sc_uint<4> param );
  inline sc_uint<1> buf_valid(void);


  // Constructor
  SC_CTOR( tblock )
  {
    // register process and define
    // the active clock edge: up
    SC_CTHREAD( ugh_main, ck.pos());
    watching(reset.delayed() == true );
  }
};



inline void tblock::buf_init(void)
{
   buf_r_th = 0;
   buf_r_tl = 0;
}

inline void tblock::buf_load(void)
{
   buf_r_tl = BUF_SZ;
   buf_r_h  = buf_r_l;
   ugh_read(data_dn_In, buf_r_l);
   ctlNbDataWordRead += 1;
}

inline void tblock::buf_flushbits(void)
{
   if (buf_r_th <= buf_out_sz ) {
      buf_r_th = buf_r_th + buf_r_tl - buf_out_sz;
      buf_r_tl = 0;
      buf_r_h  = buf_r_l;
   } else {
      buf_r_th = buf_r_th - buf_out_sz;
   }
   ctlNbIn += buf_out_sz;
}



inline void tblock::dec_calcaddr(void)
{
  sc_uint<1> error=0;
  sc_uint<9> addr;
  /* SYNTHESIS: dec_in vanishes */
  sc_uint<12> dec_in = buf_showbits(12);

  const DCTtab dec_rom[512] = {
/*    0 0 00xx xxxx :  tab0a : 256  252  :      9       :     0*/
  {0,0,0}, {0,0,0}, {0, 0,0}, {0, 0,0}, /*added blanks*/
  {65,0,6}, {65,0,6}, {65,0,6}, {65,0,6}, /* Escape */
  {7,1,7}, {7,1,7}, {8,1,7}, {8,1,7},
  {6,1,7}, {6,1,7}, {2,2,7}, {2,2,7},
  {0,7,6}, {0,7,6}, {0,7,6}, {0,7,6},
  {0,6,6}, {0,6,6}, {0,6,6}, {0,6,6},
  {4,1,6}, {4,1,6}, {4,1,6}, {4,1,6},
  {5,1,6}, {5,1,6}, {5,1,6}, {5,1,6},
  {1,5,8}, {11,1,8}, {0,11,8}, {0,10,8},
  {13,1,8}, {12,1,8}, {3,2,8}, {1,4,8},
  {2,1,5}, {2,1,5}, {2,1,5}, {2,1,5},
  {2,1,5}, {2,1,5}, {2,1,5}, {2,1,5},
  {1,2,5}, {1,2,5}, {1,2,5}, {1,2,5},
  {1,2,5}, {1,2,5}, {1,2,5}, {1,2,5},
  {3,1,5}, {3,1,5}, {3,1,5}, {3,1,5},
  {3,1,5}, {3,1,5}, {3,1,5}, {3,1,5},
  {1,1,3}, {1,1,3}, {1,1,3}, {1,1,3},
  {1,1,3}, {1,1,3}, {1,1,3}, {1,1,3},
  {1,1,3}, {1,1,3}, {1,1,3}, {1,1,3},
  {1,1,3}, {1,1,3}, {1,1,3}, {1,1,3},
  {1,1,3}, {1,1,3}, {1,1,3}, {1,1,3},
  {1,1,3}, {1,1,3}, {1,1,3}, {1,1,3},
  {1,1,3}, {1,1,3}, {1,1,3}, {1,1,3},
  {1,1,3}, {1,1,3}, {1,1,3}, {1,1,3},
  {64,0,4}, {64,0,4}, {64,0,4}, {64,0,4}, /* EOB */
  {64,0,4}, {64,0,4}, {64,0,4}, {64,0,4},
  {64,0,4}, {64,0,4}, {64,0,4}, {64,0,4},
  {64,0,4}, {64,0,4}, {64,0,4}, {64,0,4},
  {0,3,4}, {0,3,4}, {0,3,4}, {0,3,4},
  {0,3,4}, {0,3,4}, {0,3,4}, {0,3,4},
  {0,3,4}, {0,3,4}, {0,3,4}, {0,3,4},
  {0,3,4}, {0,3,4}, {0,3,4}, {0,3,4},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
```

```
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,1,2}, {0,1,2}, {0,1,2}, {0,1,2},
  {0,2,3}, {0,2,3}, {0,2,3}, {0,2,3},
  {0,2,3}, {0,2,3}, {0,2,3}, {0,2,3},
  {0,2,3}, {0,2,3}, {0,2,3}, {0,2,3},
  {0,2,3}, {0,2,3}, {0,2,3}, {0,2,3},
  {0,2,3}, {0,2,3}, {0,2,3}, {0,2,3},
  {0,2,3}, {0,2,3}, {0,2,3}, {0,2,3},
  {0,2,3}, {0,2,3}, {0,2,3}, {0,2,3},
  {0,2,3}, {0,2,3}, {0,2,3}, {0,2,3},
  {0,4,5}, {0,4,5}, {0,4,5}, {0,4,5},
  {0,4,5}, {0,4,5}, {0,4,5}, {0,4,5},
  {0,5,5}, {0,5,5}, {0,5,5}, {0,5,5},
  {0,5,5}, {0,5,5}, {0,5,5}, {0,5,5},
  {9,1,7}, {9,1,7}, {1,3,7}, {1,3,7},
  {10,1,7}, {10,1,7}, {0,8,7}, {0,8,7},
  {0,9,7}, {0,9,7}, {0,12,8}, {0,13,8},
  {2,3,8}, {4,2,8}, {0,14,8}, {0,15,8},
  /*4*63 + 4 = 256 */
/*tab0:  256 1 00xx xxxx :  tab0   :  64   60  :       1     :    0*/
  {0,0,0}, {0,0,0}, {0, 0,0}, {0, 0,0}, /*added blanks*/
  {65,0,6}, {65,0,6}, {65,0,6}, {65,0,6}, /* Escape */
  {2,2,7}, {2,2,7}, {9,1,7}, {9,1,7},
  {0,4,7}, {0,4,7}, {8,1,7}, {8,1,7},
  {7,1,6}, {7,1,6}, {7,1,6}, {7,1,6},
  {6,1,6}, {6,1,6}, {6,1,6}, {6,1,6},
  {1,2,6}, {1,2,6}, {1,2,6}, {1,2,6},
  {5,1,6}, {5,1,6}, {5,1,6}, {5,1,6},
  {13,1,8}, {0,6,8}, {12,1,8}, {11,1,8},
  {3,2,8}, {1,3,8}, {0,5,8}, {10,1,8},
  {0,3,5}, {0,3,5}, {0,3,5}, {0,3,5},
  {0,3,5}, {0,3,5}, {0,3,5}, {0,3,5},
  {4,1,5}, {4,1,5}, {4,1,5}, {4,1,5},
  {4,1,5}, {4,1,5}, {4,1,5}, {4,1,5},
  {3,1,5}, {3,1,5}, {3,1,5}, {3,1,5},
  {3,1,5}, {3,1,5}, {3,1,5}, {3,1,5},
  /* 256 +15 * 4 + 4 = 320*/
 /*  320 1 0100 xxxx :  tabN  :  16   12  :       0        :    0*/
  {0,0,0}, {0,0,0}, {0, 0,0}, {0, 0,0}, /*added blanks*/
  {0,2,4},  {2,1,4},  {1,1,3},  {1,1,3},
  {64,0,2}, {64,0,2}, {64,0,2}, {64,0,2}, /* EOB */
  {0,1,2},  {0,1,2},  {0,1,2},  {0,1,2},
  /* 320 + 3 *4 +4 = 336 */
 /*  336 1 0101 0xxx :  tab1a :   8    8  :       8        :    0 */
  {5,2,9}, {5,2,9}, {14,1,9}, {14,1,9},
  {2,4,10}, {16,1,10}, {15,1,9}, {15,1,9},

  /* 336 + 2*4 + 0 =344 */
 /*  344 1 0101 1xxx :  tab1  :   8    8  :       2        :    0 */
  {16,1,10}, {5,2,10}, {0,7,10}, {2,3,10},
  {1,4,10}, {15,1,10}, {14,1,10}, {4,2,10},
  /* 344 + 2*4 + 0 =352 */
 /*  352 1 0110 xxxx :  tab2  :  16   16  :       3        :    0 */
  {0,11,12}, {8,2,12}, {4,3,12}, {0,10,12},
  {2,4,12}, {7,2,12}, {21,1,12}, {20,1,12},
  {0,9,12}, {19,1,12}, {18,1,12}, {1,5,12},
  {3,3,12}, {0,8,12}, {6,2,12}, {17,1,12},
  /* 352 + 16 + 0 =368 */
 /*  368 1 0111 xxxx :  tab3  :  16   16  :       4        :    0 */
  {10,2,13}, {9,2,13}, {5,3,13}, {3,4,13},
  {2,5,13}, {1,7,13}, {1,6,13}, {0,15,13},
  {0,14,13}, {0,13,13}, {0,12,13}, {26,1,13},
  {25,1,13}, {24,1,13}, {23,1,13}, {22,1,13},
/* 368 +16=384*/
 /*  384 1 1000 xxxx :  tab4  :  16   16  :       5        :    0 */
  {0,31,14}, {0,30,14}, {0,29,14}, {0,28,14},
  {0,27,14}, {0,26,14}, {0,25,14}, {0,24,14},
  {0,23,14}, {0,22,14}, {0,21,14}, {0,20,14},
  {0,19,14}, {0,18,14}, {0,17,14}, {0,16,14},
 /*  400 1 1001 xxxx :  tab5  :  16   16  :       6        :    0 */
  {0,40,15}, {0,39,15}, {0,38,15}, {0,37,15},
  {0,36,15}, {0,35,15}, {0,34,15}, {0,33,15},
  {0,32,15}, {1,14,15}, {1,13,15}, {1,12,15},
  {1,11,15}, {1,10,15}, {1,9,15}, {1,8,15},
 /*  416 1 1010 xxxx :  tab6  :  16   16  :       7        :    0 */
  {1,18,16}, {1,17,16}, {1,16,16}, {1,15,16},
  {6,3,16}, {16,2,16}, {15,2,16}, {14,2,16},
  {13,2,16}, {12,2,16}, {11,2,16}, {31,1,16},
  {30,1,16}, {29,1,16}, {28,1,16}, {27,1,16},
 /*  432 1 1011 xxxx :  tabF  :  16   12  :      10        :    0 */
  {0,0,0}, {0,0,0}, {0, 0,0}, {0, 0,0}, /*added blanks*/
  {0,2,4}, {2,1,4}, {1,1,3}, {1,1,3},
  {0,1,1}, {0,1,1}, {0,1,1}, {0,1,1},
  {0,1,1}, {0,1,1}, {0,1,1}, {0,1,1}
 /*  448 1           :        :           :     XX        :    1 */
};


   if ((dec_in & 0xFFF) == 0 ) { /* EQUIVALENT: dec_in <= 0xF */
      error = 1;
   } else if ((dec_in & 0xFFE) == 0) {
      error = 0;
    addr = 416 | (buf_showbits(16) & 0xF); /* Tab6 */
   } else if ((dec_in & 0xFFC) == 0) {
      error = 0;
      addr = 400 | (buf_showbits(15) & 0xF); /* Tab5 */
   } else if ((dec_in & 0xFF8) == 0) {
      error = 0;
      addr = 384 | (buf_showbits(14) & 0xF); /* Tab4 */
   } else if ((dec_in & 0xFF0) == 0) {
      error = 0;
      addr = 368 | (buf_showbits(13) & 0xF); /* Tab3 */
   } else if ((dec_in & 0xFE0) == 0) {
      error = 0;
      addr = 352 | (buf_showbits(12) & 0xF); /* Tab2 */
   } else if ((dec_in & 0xFC0) == 0) {
      if (intra()==0) {
         addr = 344 | (buf_showbits(10) & 0x7); /* Tab1 */
      } else {
         addr = 336 | (buf_showbits(10) & 0x7); /* Tab1a */
      }
   } else if ((dec_in & 0xC00) == 0) {
      if (intra()==0) {
         addr = 256 | (buf_showbits(8) & 0x3F); /* Tab0 */
      } else {
         addr = (buf_showbits(8) & 0xFF); /* Tab0a */
      }
   } else {
    if (func()==UGH_getmpg2intramblock) {
      if (intra()!=0) {
            addr = (buf_showbits(8) & 0xFF); /* Tab0a */
          } else {
        addr = (0x140 | (dec_in >> 8)); /* TabN */
            /* EQUIVALENT:
               addr = 320 | (dec_in >> 8));
               addr = 320 | (buf_showbits(4));
               addr = 320 + (dec_in >> 8));
               addr = 320 + (buf_showbits(4));
            */
          }
    } else {
      if (dec_Ieq0!=0) {
            addr = (432 | (dec_in >> 8)); /* TabF */
          } else {
            addr = (0x140 | (dec_in >> 8)); /* TabN */
            /* EQUIVALENT:
               addr = 320 | (dec_in >> 8));
               addr = 320 | (buf_showbits(4));
               addr = 320 + (dec_in >> 8));
               addr = 320 + (buf_showbits(4));
            */
          }
    }
   }

   dec_value = dec_rom[addr];
   dec_fault = error;
   dec_Ieq0  = 0;
}

/**** functions:                                                    ****/
/****  - set_lum():                                                 ****/
/****   pre :                                                       ****/
/****   post: configue the bloc to write luminence.                 ****/
/****  - set_chrom():                                               ****/
/****   pre :                                                       ****/
/****   post: configue the bloc to write luminence.                 ****/
/****  - write_lc():                                                ****/
/****   pre : need 26 bits in input buffer.                         ****/
/****   post: write either luminence or chrominence depending ****/
/****       on bloc state.                                          ****/

inline void tblock::lc_setlum()
{
    lc_state = 0;
}

inline void tblock::lc_setchrom()
{
    lc_state = 1;
}

inline void tblock::lc_write()
{
    sc_uint<5> code5;
```

```
    sc_uint<7> code7;
    sc_uint<8> size;
    sc_uint<32> val;

    const VLCtab lc_rom[128] = {
/* Table B-12, dct_dc_size_luminance, codes 00xxx ... 11110 */
/*static VLCtab DClumtab0[32] =*/
  {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2},
  {2, 2}, {2, 2}, {2, 2}, {2, 2}, {2, 2}, {2, 2}, {2, 2}, {2, 2},
  {0, 3}, {0, 3}, {0, 3}, {0, 3}, {3, 3}, {3, 3}, {3, 3}, {3, 3},
  {4, 3}, {4, 3}, {4, 3}, {4, 3}, {5, 4}, {5, 4}, {6, 5},
  {ERROR, 0},


/* Table B-12, dct_dc_size_luminance,
   codes 111110xxx ... 111111111 */
/*static VLCtab DClumtab1[16] =*/
  {7, 6}, {7, 6}, {7, 6}, {7, 6}, {7, 6}, {7, 6}, {7, 6}, {7, 6},
  {8, 7}, {8, 7}, {8, 7}, {8, 7}, {9, 8}, {9, 8}, {10,9}, {11,9},
  {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0},
  {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0},

/* Table B-13, dct_dc_size_chrominance, codes 00xxx ... 11110 */
/*static VLCtab DCchromtab0[32] =*/
  {0, 2}, {0, 2}, {0, 2}, {0, 2}, {0, 2}, {0, 2}, {0, 2}, {0, 2},
  {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2},
  {2, 2}, {2, 2}, {2, 2}, {2, 2}, {2, 2}, {2, 2}, {2, 2}, {2, 2},
  {3, 3}, {3, 3}, {3, 3}, {3, 3}, {4, 4}, {4, 4}, {5, 5},
  {ERROR, 0},


/* Table B-13, dct_dc_size_chrominance,
   codes 111110xxxx ... 1111111111 */
/*static VLCtab DCchromtab1[32] =*/

  {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6},
  {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6},
  {7, 7}, {7, 7}, {7, 7}, {7, 7}, {7, 7}, {7, 7}, {7, 7}, {7, 7},
  {8, 8}, {8, 8}, {8, 8}, {8, 8}, {9, 9}, {9, 9}, {10,10}, {11,10}
};



  code5 = buf_showbits(5);

  if ( code5 < 31 ) {
    code7 = code5 | (lc_state<<6);
  } else if ( lc_state == 1 ) { /*chrom*/
    /* EQUIVALENT: code7 = (1<<6) | (buf_showbits(10) & 0x1e) */
    code7 = ((1<<6) | (buf_showbits(10) - 0x3e0)) + 32;
  } else  { /*lum*/
    /* EQUIVALENT: code7 = buf_showbits(9) & 0x1e0 */
    code7 =  buf_showbits(9) - 0x1f0 + 32;
  }

  size = lc_rom[code7].val;
  /*flush des n premiers bits*/
  buf_out_sz = lc_rom[code7].len;
  buf_flushbits();

  if ( size == 0 ) {
    val = 0;
  } else {
    val = (buf_showbits(11) << 5) & 0xffe0;
    val = (size<<16) | val;
    /*flush de n bits*/
    buf_out_sz = size;
    buf_flushbits();
  }

  DataWrite(data_up_Out, val, 1);
}

inline sc_uint<1> tblock::buf_valid(void)
{
   sc_uint<32> my_tmp;
   my_tmp = buf_r_th + buf_r_tl;

   return buf_out_sz < (my_tmp);
}

/*** saute les bits invalides dans la fifo data_dn_In ***/

inline void tblock::Tblock_skip()
{
   sc_uint<32> tmp;

   ugh_read(cmd_dn_In, tmp);

   skipcount = ((tmp  >> 16) & 0x7FFF) * 32 + (tmp & 0xFFFF);

   buf_out_sz = 32;

   while (skipcount >= 32) {
      if (buf_valid()==0) buf_load();
      else wait();

      buf_flushbits();
      skipcount -= 32;
   }

   buf_out_sz = skipcount;

   if (buf_valid()==0) buf_load();
   else wait();

   buf_flushbits();
}

/*rom*/
inline sc_int<1> tblock::dec_comp_sup( sc_uint<4> key )
{
  switch (key)
  {
    case 4: case 5: return 1;
    default: return 0;
  }
}

/*rom*/
inline sc_int<1> tblock::dec_comp_rom( sc_uint<4> key )
{
  switch (key)
  {
    case 0: case 2: case 3: case 1: return 1;
    default: return 0;
  }
}

/*** entry point of vld task ***/
void tblock::ugh_main()
{
  sc_uint<32> tmp;

  /*init  */
  buf_init();

  /*init protocol*/
  cmd_dn_In_read = false;
  data_dn_In_read = false;
  data_up_Out_write = false;
  cmd_up_Out_write = false;
  wait();

  while(true) {
    ctlNbIn = 0;
    ctlNbDataWordRead=0;
    ctlNbOut = 0;

    /*read fifo control*/
    ugh_read(cmd_dn_In, tmp);

    dec_comp = 0;
    dec_fault = 0;
    control = tmp;
    dec_cbp = cbp();

    /*saute les bits invalides dans la fifo data_dn_In*/
    Tblock_skip();

    /*decodage*/
    while (!( dec_comp == 6 || dec_fault == 1 )) {
      buf_out_sz = 17;

      if ( func()==UGH_getmpegintermblock
        || func()==UGH_getmpg2intermblock ) {
        ++dec_comp;
        dec_cbp *= 2;
        /* dec_bloc_end = 0; */
        dec_Ieq0 = 1;
        wait();
      }
      else if ( ( func()==UGH_getmpegintramblock ||
                  func()==UGH_getmpg2intramblock )
           && ( buf_valid()!=0  && dec_comp_rom(dec_comp) ) ) {
        /* && data_up_Out non pleine ssi write
           remplace par des write */
        lc_setlum();
        lc_write();
        ++dec_comp;
        dec_cbp *= 2;
        dec_Ieq0 = 1;
      }
      else if ( ( func() == UGH_getmpegintramblock ||
```

```
            func() == UGH_getmpg2intramblock )
        &&   ( buf_valid()!=0 && dec_comp_sup(dec_comp) ) ) {
    /* && data_up_Out non pleine ssi write
       remplace par des write*/
    lc_setchrom();
    lc_write();
    ++dec_comp;
    dec_cbp *= 2;
    dec_Ieq0 = 1;
  }
  else {
    buf_load();
    continue;
  }

  /* d_coef_loop */
  /* ( dec_bloc_end!=1 && dec_fault==0 ) */
  while (dec_fault==0) {
    sc_uint<1> cbp_do= (dec_cbp>>6) & 1;
    buf_out_sz = 17;

    if ( cbp_do == 0 ) break;
    else if ( buf_valid()!=0 ) {
      /*load dec_value, dec_fault */
      dec_calcaddr();

      if ( dec_fault == 1 ) break;

      buf_out_sz = dec_value.len;
      buf_flushbits();

      if ( dec_value.run == 64 ) {
        tmp =  (sc_uint<32>) (VLD_EOB << 16);
        DataWrite(data_up_Out, tmp, 1);
        break;
      }
      else if ( dec_value.run == 65) {
        sc_uint<18> val;
        buf_out_sz = 18;
        if (buf_valid()==0) buf_load();
        else wait();

        val = buf_showbits(18);

        buf_out_sz = 18;
        buf_flushbits();


        if ( (val&0xFFF) ==0 ) {
          dec_fault = 1;
          break;
        }

        tmp= (val&0xfff) | ((val&0x800)<<1)
          | ((val&0x800)<<2) | ((val&0x800)<<3)
          | ((val&0x800)<<4)
          | ((( (sc_uint<32>)(val&0x3f000))>>12 )<<16);

        DataWrite( data_up_Out, tmp, 1);
      }
      else {
        if (buf_showbits(1) == 1) {
          tmp= (dec_value.run << 16) | ((0-dec_value.level)&0xffff);
        }
        else {
          tmp= (dec_value.run << 16) | (dec_value.level&0xffff);
        }

        DataWrite( data_up_Out, tmp, 1);
        buf_out_sz = 1;
        buf_flushbits();
      }
    }
    else if ( buf_valid()==0 ) buf_load();
    else wait();

  } /*while*/

}/*fin decodage*/

tmp = (dec_fault<<31) | ((ctlNbIn&0x7fff)<<16)
    | (ctlNbDataWordRead*32);
ugh_write(cmd_up_Out, tmp);

tmp = (dec_fault<<31) | ((ctlNbIn&0x7fff)<<16) | ctlNbOut;
ugh_write(cmd_up_Out, tmp);

}  /*while(1)*/

}
```

A.3 Le langage utilisé par *gaut*

Le langage de description utilisé par *gaut* est un sous-ensemble du *VHDL*. C'est un processus unique constitué d'instructions d'assignations séquentielles de variables et de ports.

Les fonctions ne correspondant pas à des opérateurs de la bibliothèque ne sont pas autorisées.
Les types sont très restrictifs :

- Il existe une seule taille possible : *integer*. Cette taille est définie par la taille des opérateurs dans la bibliothèque. Les opérations ont donc toutes la même taille.
- Dans la sémantique de *gaut*, chaque port représente une et une seule donnée. Les ports sont donc représentés sous forme de tableaux pour forcer l'outil à faire plusieurs accès au même port.

Description comportementale *VHDL* de l'idct du multi-Jpeg

```
library ieee;
use ieee.std_logic_1164.all;

PACKAGE idct_pkg IS
  SUBTYPE vector1   IS integer; --std_logic;
  SUBTYPE vector7   IS integer; --std_logic_vector( 6 downto 0);
  SUBTYPE vector8   IS integer; --std_logic_vector( 7 downto 0);
  SUBTYPE vector32  IS integer; --std_logic_vector(31 downto 0);

  TYPE vector32_64  IS ARRAY (0 TO 63) OF vector32;
  TYPE vector8_64   IS ARRAY (0 TO 63) OF vector8;
  TYPE vector32_16  IS ARRAY (0 TO 15) OF vector32;

  CONSTANT N : INTEGER := 488 ;
END idct_pkg ;

library work;
use work.idct_pkg.all;
library ieee;
use ieee.std_logic_1164.all;
```

```
entity idct is
port(
  -- fifos
  zz2idct        : in vector32_64;
  idct2libu      : out vector32_64
  );
end;

architecture arch_idct of idct is
begin

process
  constant latence : TIME := N * 10 ns;

  constant COS : vector32_16 := (
      16384, 16069, 15137, 13623, 11585, 9102, 6270, 3196,
      23170, 22725, 21407, 19266, 16384, 12873, 8867, 4520);

  -- Declare internal variables and signals
  variable my_block : vector32_64;
  variable Idct     : vector8_64;
  variable k, l     : integer;     -- vector7;
  variable x        : vector32;
  variable z1_0,z1_1,z1_2,z1_3,z1_4,z1_5,z1_6,z1_7 : vector32;
  variable z2_0,z2_1,z2_2,z2_3,z2_4,z2_5,z2_6,z2_7 : vector32;
  variable z3_0,z3_1,z3_2,z3_3,z3_4,z3_5,z3_6,z3_7 : vector32;
  variable i : integer;           --vector8;
  variable s0, s1, s2, s3, s4, s5, s6, s7 : vector32;
  variable d0, d1, d2, d3, d4, d5, d6, d7 : vector32;

begin

  my_block( 0 )  := zz2idct( 0 );
  my_block( 1 )  := zz2idct( 1 );
  my_block( 2 )  := zz2idct( 2 );
  my_block( 3 )  := zz2idct( 3 );
  my_block( 4 )  := zz2idct( 4 );
  my_block( 5 )  := zz2idct( 5 );
  my_block( 6 )  := zz2idct( 6 );
  my_block( 7 )  := zz2idct( 7 );
  my_block( 8 )  := zz2idct( 8 );
  my_block( 9 )  := zz2idct( 9 );
  my_block( 10 ) := zz2idct( 10 );
  my_block( 11 ) := zz2idct( 11 );
  my_block( 12 ) := zz2idct( 12 );
  my_block( 13 ) := zz2idct( 13 );
  my_block( 14 ) := zz2idct( 14 );
  my_block( 15 ) := zz2idct( 15 );
  my_block( 16 ) := zz2idct( 16 );
  my_block( 17 ) := zz2idct( 17 );
  my_block( 18 ) := zz2idct( 18 );
  my_block( 19 ) := zz2idct( 19 );
  my_block( 20 ) := zz2idct( 20 );
  my_block( 21 ) := zz2idct( 21 );
  my_block( 22 ) := zz2idct( 22 );
  my_block( 23 ) := zz2idct( 23 );
  my_block( 24 ) := zz2idct( 24 );
  my_block( 25 ) := zz2idct( 25 );
  my_block( 26 ) := zz2idct( 26 );
  my_block( 27 ) := zz2idct( 27 );
  my_block( 28 ) := zz2idct( 28 );
  my_block( 29 ) := zz2idct( 29 );
  my_block( 30 ) := zz2idct( 30 );
  my_block( 31 ) := zz2idct( 31 );
  my_block( 32 ) := zz2idct( 32 );
  my_block( 33 ) := zz2idct( 33 );
  my_block( 34 ) := zz2idct( 34 );
  my_block( 35 ) := zz2idct( 35 );
  my_block( 36 ) := zz2idct( 36 );
  my_block( 37 ) := zz2idct( 37 );
  my_block( 38 ) := zz2idct( 38 );
  my_block( 39 ) := zz2idct( 39 );
  my_block( 40 ) := zz2idct( 40 );
  my_block( 41 ) := zz2idct( 41 );
  my_block( 42 ) := zz2idct( 42 );
  my_block( 43 ) := zz2idct( 43 );
  my_block( 44 ) := zz2idct( 44 );
  my_block( 45 ) := zz2idct( 45 );
  my_block( 46 ) := zz2idct( 46 );
  my_block( 47 ) := zz2idct( 47 );
  my_block( 48 ) := zz2idct( 48 );
  my_block( 49 ) := zz2idct( 49 );
  my_block( 50 ) := zz2idct( 50 );
  my_block( 51 ) := zz2idct( 51 );
  my_block( 52 ) := zz2idct( 52 );
  my_block( 53 ) := zz2idct( 53 );
  my_block( 54 ) := zz2idct( 54 );
  my_block( 55 ) := zz2idct( 55 );
  my_block( 56 ) := zz2idct( 56 );
  my_block( 57 ) := zz2idct( 57 );
  my_block( 58 ) := zz2idct( 58 );
  my_block( 59 ) := zz2idct( 59 );
```

```
my_block( 60 ) := zz2idct( 60 );
my_block( 61 ) := zz2idct( 61 );
my_block( 62 ) := zz2idct( 62 );
my_block( 63 ) := zz2idct( 63 );


for k in 0  to 7 loop

   s0   := my_block( (k * 8)+0 ) * 8;
   s1   := my_block( (k * 8)+1 ) * 8;
   s2   := my_block( (k * 8)+2 ) * 8;
   s3   := my_block( (k * 8)+3 ) * 8;
   s4   := my_block( (k * 8)+4 ) * 8;
   s5   := my_block( (k * 8)+5 ) * 8;
   s6   := my_block( (k * 8)+6 ) * 8;
   s7   := my_block( (k * 8)+7 ) * 8;

   z1_0 := s0 + s4;
   z1_1 := s0 - s4;
   z1_2 := ((( COS(14) ) * ( s2 ) + (1 sll (13))) srl 14)
         - ((( COS(10) ) * ( s6 ) + (1 sll (13))) srl 14);
   z1_3 := ((( COS(10) ) * ( s2 ) + (1 sll (13))) srl 14)
         + ((( COS(14) ) * ( s6 ) + (1 sll (13))) srl 14);
   z1_4 :=  s1 - s7;
   z1_5 := (( 23170 ) * ( s3 ) + (1 sll (13))) srl 14;
   z1_6 := (( 23170 ) * ( s5 ) + (1 sll (13))) srl 14;
   z1_7 := s1 + s7;

   z2_0 := z1_0 + z1_3;
   z2_1 := z1_1 + z1_2;
   z2_2 := z1_1 - z1_2;
   z2_3 := z1_0 - z1_3;
   z2_4 := z1_4 + z1_6;
   z2_5 := z1_7 - z1_5;
   z2_6 := z1_4 - z1_6;
   z2_7 := z1_7 + z1_5;

   z3_0 := z2_0;
   z3_1 := z2_1;
   z3_2 := z2_2;
   z3_3 := z2_3;
   z3_4 := ((( COS(3) ) * ( z2_4 ) + (1 sll (13))) srl 14)
         - ((( COS(5) ) * ( z2_7 ) + (1 sll (13))) srl 14);
   z3_7 := ((( COS(5) ) * ( z2_4 ) + (1 sll (13))) srl 14)
         + ((( COS(3) ) * ( z2_7 ) + (1 sll (13))) srl 14);
   z3_5 := ((( COS(1) ) * ( z2_5 ) + (1 sll (13))) srl 14)
         - ((( COS(7) ) * ( z2_6 ) + (1 sll (13))) srl 14);
   z3_6 := ((( COS(7) ) * ( z2_5 ) + (1 sll (13))) srl 14)
         + ((( COS(1) ) * ( z2_6 ) + (1 sll (13))) srl 14);

   d7 := z3_0 - z3_7;
   d0 := z3_0 + z3_7;
   d6 := z3_1 - z3_6;
   d1 := z3_1 + z3_6;
   d5 := z3_2 - z3_5;
   d2 := z3_2 + z3_5;
   d4 := z3_3 - z3_4;
   d3 := z3_3 + z3_4;


   my_block( (k * 8)+0 ) := d0;
   my_block( (k * 8)+1 ) := d1;
   my_block( (k * 8)+2 ) := d2;
   my_block( (k * 8)+3 ) := d3;
   my_block( (k * 8)+4 ) := d4;
   my_block( (k * 8)+5 ) := d5;
   my_block( (k * 8)+6 ) := d6;
   my_block( (k * 8)+7 ) := d7;

end loop;

for l in 0  to 7 loop

  s0 := my_block( (0 * 8)+l );
  s1 := my_block( (1 * 8)+l );
  s2 := my_block( (2 * 8)+l );
  s3 := my_block( (3 * 8)+l );
  s4 := my_block( (4 * 8)+l );
  s5 := my_block( (5 * 8)+l );
  s6 := my_block( (6 * 8)+l );
  s7 := my_block( (7 * 8)+l );


  z1_0 :=  s0 + s4;
  z1_1 :=  s0 - s4;
  z1_2 := ((( COS(14) ) * ( s2 ) + (1 sll (13))) srl 14)
        - ((( COS(10) ) * ( s6 ) + (1 sll (13))) srl 14);
  z1_3 := ((( COS(10) ) * ( s2 ) + (1 sll (13))) srl 14)
        + ((( COS(14) ) * ( s6 ) + (1 sll (13))) srl 14);
  z1_4 :=  s1 - s7;
  z1_5 := ((( 23170 ) * ( s3 ) + (1 sll (13))) srl 14);
```

```
z1_6 := ((( 23170 ) * ( s5 ) + (1 sll (13))) srl 14);
z1_7 :=  s1 + s7;

z2_0 := z1_0 + z1_3;
z2_1 := z1_1 + z1_2;
z2_2 := z1_1 - z1_2;
z2_3 := z1_0 - z1_3;
z2_4 := z1_4 + z1_6;
z2_5 := z1_7 - z1_5;
z2_6 := z1_4 - z1_6;
z2_7 := z1_7 + z1_5;

z3_0 := z2_0;
z3_1 := z2_1;
z3_2 := z2_2;
z3_3 := z2_3;
z3_4 :=  ((( COS(3) ) * ( z2_4 ) + (1 sll (13))) srl 14)
       - ((( COS(5) ) * ( z2_7 ) + (1 sll (13))) srl 14);
z3_7 :=  ((( COS(5) ) * ( z2_4 ) + (1 sll (13))) srl 14)
       + ((( COS(3) ) * ( z2_7 ) + (1 sll (13))) srl 14);
z3_5 :=  ((( COS(1) ) * ( z2_5 ) + (1 sll (13))) srl 14)
       - ((( COS(7) ) * ( z2_6 ) + (1 sll (13))) srl 14);
z3_6 :=  ((( COS(7) ) * ( z2_5 ) + (1 sll (13))) srl 14)
       + ((( COS(1) ) * ( z2_6 ) + (1 sll (13))) srl 14);

d7 := z3_0 - z3_7;
d0 := z3_0 + z3_7;
d6 := z3_1 - z3_6;
d1 := z3_1 + z3_6;
d5 := z3_2 - z3_5;
d2 := z3_2 + z3_5;
d4 := z3_3 - z3_4;
d3 := z3_3 + z3_4;

my_block( (0 * 8)+l ) := d0;
my_block( (1 * 8)+l ) := d1;
my_block( (2 * 8)+l ) := d2;
my_block( (3 * 8)+l ) := d3;
my_block( (4 * 8)+l ) := d4;
my_block( (5 * 8)+l ) := d5;
```

```
    my_block( (6 * 8)+l ) := d6;
    my_block( (7 * 8)+l ) := d7;



    for k in 0  to 7 loop
       x := my_block( (k * 8)+l ) + (1 sll 5);

       if x < 0 then
          x := x - 1;
       end if;

       x := (x srl 6) + 128;

       if x < 0 then x := 0;
       else x := 255;
       end if;

       Idct( (k * 8)+l ) := x;

    end loop;

  end loop;



  for i in 0 to 63 loop

     idct2libu(i) <= Idct( i );

  end loop;


  WAIT FOR latence;

end process;


end; -- architecture
```

Annexe B

Directives d'affectations de la synthèse ciblée

Ce chapitre présente un exemple de directives d'affectation que le concepteur fournit à notre outil de synthèse ciblée *ugh*. Ces directives sont indispensables pour guider l'outil et cibler précisément une solution. Cependant, les directives peuvent être plus ou moins contraignantes suivant le degré de liberté que veut laisser le concepteur à l'outil.

B.1 La description des directives d'affectation

Les directives sont composées de deux parties :

1. *déclaration des opérateurs* combinatoires et séquentiels.
2. *assignation des ports* des opérateurs.

Dans ces directives, l'opérateur matériel (combinatoire ou séquentiel) est désigné par un nom unique. Les variables de la description comportementale sont affectées aux registres de même nom. Les variables non affectées sont des signaux.
Les opérateurs combinatoires ne peuvent être affectés explicitement à une opération de la description comportementale.

La *déclaration des opérateurs* est obligatoire. En effet, elle consiste à donner le nombre et le type des opérateurs combinatoires et l'*affectation des registres* (identifiée par correspondance des noms).
L'*assignation des ports* des opérateurs est optionnelle. Elle représente la connexion entre les opérateurs combinatoires et les registres. L'outil affecte alors un opérateur combinatoire à une opération fonctionnelle si les chemins menant à cet opérateur (décrits grâce aux directives de connexion) sont compatibles avec la description comportementale.

B.2 L'exemple du vld

L'exemple présenté plus bas donne les directives d'affectations pour les registres et les opérateurs du vld du Mpeg. Cet exemple de directives d'affectation est fourni à *ugh* avec la description comportementale associée (cf. chapitre A.1).

Directives d'allocation pour vld du Mpeg

```
MODEL VLD ( outChannel control_out; inChannel control_in; outChannel
data_out; inChannel data_in )
{
  -- registers
  DFF buf_flushbits_tmp;
  DFF tmp;
  DFF ctlnbout;
  DFF ctlnbin ;
  DFF dec_fault;
  DFF dec_calcaddr_addr;
  DFF dec_calcaddr_error;
  DFF dec_calcaddr_dec_in;
  DFF ctlnbdatawordread;
  DFF buf_r_tl;
  DFF buf_r_th;
  DFF buf_r_l;
  DFF buf_r_h;
  DFF buf_valid;
  DFF buf_out_sz;
  DFF dec_value;
  DFF val;
  DFF dec_ieq0;
  DFF control;
  DFF cbp_do;
  DFF dec_cbp;
  DFF dec_comp;
  DFF lc_write_code5;
  DFF lc_write_code7;
  DFF lc_state;
  DFF lc_write_size;
  DFF lc_write_val;
  DFF tblock_skip_tmp;
  DFF skipcount;
  DFF tmp_wired_8;

  -- register files
  ROM  dec_rom;
  ROM  lc_rom;
  ROM  dec_comp_rom;
  ROM  dec_comp_sup;

  ADD  add1;
  ADD  buf_add;
  ADD  add2;
  ADD  add3;
  ALU  alu1;
  ADD  add4;
  SUB  buf_sub1;
  SUB  buf_sub2;
  SUB  buf_sub_wired;
  SUB  sub1;
  SUB  sub3;
  CATSHR   cat_shr1;

  --
  -- Connections du bloc Buffer
  --
  buf_r_h.d      = buf_r_l.q, buf_r_h.q;
  buf_r_l.d      = buf_r_l.q;
  buf_r_tl.d     = buf_r_tl.q;

  cat_shr1.sin   = buf_r_h.q;
  cat_shr1.sh    = buf_r_th.q, buf_sub1.s;
  cat_shr1.i     = buf_r_l.q;

  buf_add.a      = buf_r_th.q, ctlNbIn.q;
  buf_add.b      = buf_r_tl.q, buf_out_sz.q;
  ctlNbIn.d      = buf_add.s;

  buf_sub1.a     = buf_add.s, buf_out_sz.q;
  buf_sub1.b     = buf_add.s, buf_out_sz.q;
  buf_r_th.d     = buf_sub1.s;
  buf_valid.d    = buf_sub1.ng, buf_sub1.ov, buf_sub1.co;

  buf_sub2.a      = buf_out_sz.q;
  buf_sub2.b      = buf_r_th.q, buf_flushbits_tmp.q;
  buf_r_th.d      = buf_sub2.s;

  buf_sub_wired.a     = buf_r_th.q;
  buf_sub_wired.b     = buf_out_sz.q;
  buf_flushbits_tmp.d = buf_sub_wired.ng;
  buf_r_th.d          = buf_sub_wired.s;
  tmp_wired_8.d   = cat_shr1.z;

  --FIFO
  control_out  = tmp.q;
  buf_r_l.d    = data_in;
  data_out     = tmp.q, lc_write_val.q;

  --RAM, ROM
  dec_rom.i    = dec_calcaddr_addr.q;
  lc_rom.i     = lc_write_code7.q ;
  dec_comp_rom.i = dec_comp.q;
  dec_comp_sup.i = dec_comp.q;

  --REG
  lc_write_val.d   = cat_shr1.z, lc_write_val.q, lc_write_size.q ;
  tmp.d            = ctlnbout.q, ctlnbin.q, dec_fault.q,
                     ctlnbdatawordread.q, dec_value.q,
                     val.q, control_in;
  val.d            = cat_shr1.z;
  buf_out_sz.d     = dec_value.q, lc_rom.z, lc_write_size.q,
                     skipcount.q;
  dec_calcaddr_dec_in.d =  cat_shr1.z;
  dec_calcaddr_addr.d   =  dec_calcaddr_dec_in.q, cat_shr1.z ;
  dec_value.d      = dec_rom.z;
  lc_write_code7.d = lc_state.q, cat_shr1.z, lc_write_code5.q;
  lc_write_code5.d = lc_state.q, cat_shr1.z, lc_write_code5.q;

  -- INCREMEMT and decremement 16 bits
  alu1.b      = ctlnbdatawordread.q, ctlnbout.q, skipcount.q;
  ctlnbdatawordread.d  =  alu1.s;
  ctlnbout.d  = alu1.s;
  skipcount.d = alu1.s;

  add4.a = tblock_skip_tmp.q;
  add4.b = tblock_skip_tmp.q;
  skipcount.d = add4.s;

  -- INCREMEMT 4 bits
  add1.a      = dec_comp.q;
  dec_comp.d  = add1.s;

  -- ADD 16
  add2.a      = buf_out_sz.q;
  add2.b      = ctlnbin.q;
  ctlnbin.d   = add2.s;

  -- SUB 6 bits
  sub1.b      = dec_value.q, lc_write_code5.q;
  tmp.d       = sub1.s;

  add3.a      = cat_shr1.z;
  lc_write_code7.d  = add3.s;

  sub3.a      = cat_shr1.z;
  sub3.a      = cat_shr1.z;
  lc_write_code7.d  = sub3.s;
}
```

Annexe C

Génération du VHDL RTL

Dans le chapitre 7.4, nous présentons plusieurs approches pour l'intégration de la synthèse de haut niveau dans le flot de conception. Ce chapitre montre les différents niveaux RTL utilisés pour ces intégrations.

En effet, l'outil de synthèse ciblée *ugh* génère trois formes VHDL du circuit, utilisables par les différents types d'outils travaillant au niveau RTL :

1. Chemin de données comportemental inclus dans l'automate d'états.
2. Chemin de données comportemental séparé de l'automate d'états.
3. Chemin de données structurel séparé de l'automate d'états.

C.1 La description VHDL RTL n^o1

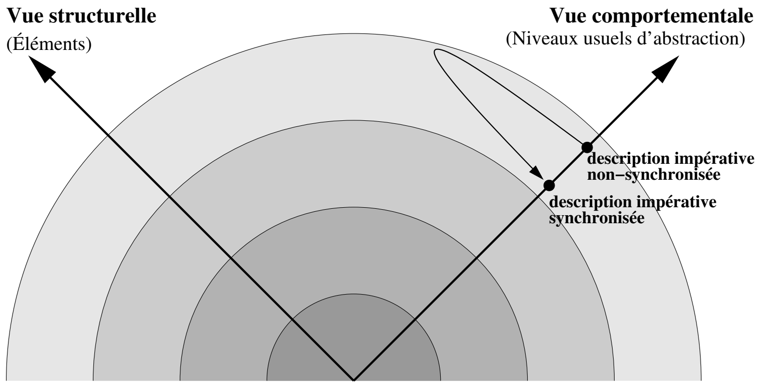


FIG. C.1 Chemin de données comportemental inclus dans l'automate

Le chemin de données est une description comportementale incluse dans l'automate (cf. figure C.1). Chaque état de l'automate contient ses opérations fonctionnelles. C'est la synthèse RTL qui décide de l'allocation et de l'affectation de ces opérations.

La description est constituée de deux processus en VHDL : L'automate est synchrone car la mise à jour de tous les registres se fait dans un processus sensible uniquement à l'horloge.
Pour pouvoir enregistrer une valeur à chaque cycle, un signal intermédiaire est utilisé pour chaque entrée de registre du chemin de donnée. Ce signal est initialisé avec la valeur précédente du registre.

Description RTL de l'automate à états comportementaux

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY vld IS
PORT
(
  abort : IN STD_LOGIC;
  reset : IN STD_LOGIC;
  control_out_write : OUT STD_LOGIC;
  control_out_wok : IN STD_LOGIC;
  control_out : OUT STD_LOGIC_VECTOR(31 DOWNTO 0) ;
  data_out_write : OUT STD_LOGIC;
  data_out_wok : IN STD_LOGIC;
  data_out : OUT STD_LOGIC_VECTOR(31 DOWNTO 0) ;
  data_in_read : OUT STD_LOGIC;
  data_in_rok : IN STD_LOGIC;
  data_in : IN STD_LOGIC_VECTOR(31 DOWNTO 0) ;
  control_in_read : OUT STD_LOGIC;
  control_in_rok : IN STD_LOGIC;
  control_in : IN STD_LOGIC_VECTOR(31 DOWNTO 0) ;
  start : IN STD_LOGIC;
  ck : IN STD_LOGIC
);
END vld;

ARCHITECTURE FSM OF tblock IS

TYPE STATE_TYPE IS
(  beg_func_at_tblock_ughc_260,wait_at_tblock_ughc_260,
   asg_ctlnbout_at_tblock_ughc_125,
   call_ugh_read_at_tblock_ughc_128,
   asg_control_at_tblock_ughc_132,
              .
              .
              .
              .
);

SIGNAL    CURRENT_STATE, NEXT_STATE: STATE_TYPE;

SIGNAL  buf_r_th, buf_r_th_sig :  STD_LOGIC_VECTOR(5 DOWNTO 0) ;
SIGNAL  buf_r_tl, buf_r_tl_sig :  STD_LOGIC_VECTOR(5 DOWNTO 0) ;
SIGNAL  ctlnbin, ctlnbin_sig :  STD_LOGIC_VECTOR(15 DOWNTO 0) ;
              .
              .
              .
              .
              .
BEGIN

--------------------------------------------------------
--------------------------------------------------------
  PROCESS( CURRENT_STATE, abort, reset, control_out_wok,
  data_out_wok, data_in_rok, data_in, control_in_rok,
  control_in, start, ck, buf_r_th,
  buf_r_tl, ctlnbin, ctlnbdatawordread, ctlnbout,
  dec_comp, dec_fault, control, dec_cbp, skipcount,
  buf_out_sz, buf_valid,
  buf_r_h, buf_r_l, dec_ieq0, lc_state, dec_value,
  buf_flushbits_tmp, lc_write_code5, lc_write_code7,
  lc_write_size, lc_write_val, dec_calcaddr_error,
  dec_calcaddr_addr, dec_calcaddr_dec_in, tmp,
  tmp_wired_8, val, tblock_skip_tmp )

VARIABLE  ugh_tmp_unsigned_9 :  STD_LOGIC_VECTOR(31 DOWNTO 0) ;
VARIABLE  ugh_tmp_unsigned_0 :  STD_LOGIC_VECTOR(6 DOWNTO 0) ;
                     .
                     .
                     .
                     .
TYPE dec_rom_type IS ARRAY (0 TO 511)
                  OF STD_LOGIC_VECTOR(18 DOWNTO 0);

CONSTANT  dec_rom :  dec_rom_type := ( "0000000000000000000",
"0000000000000000000", "0000000000000000000", "0000000000000000000",
"1000001000000000110", "1000001000000000110", "1000001000000000110",
"1000001000000000110", "0000111000001000111", "0000111000001000111",
                        .
                        .
                        .

function catshr(I: in STD_LOGIC_VECTOR; SH: in STD_LOGIC_VECTOR;
                SIN: in STD_LOGIC_VECTOR)
                return STD_LOGIC_VECTOR is

constant I_left    : INTEGER := (I'length) - 1;
constant Tmp_left  : INTEGER := (I'length) + (SIN'length) - 1;
variable Tmp       : STD_LOGIC_VECTOR( Tmp_left downto 0 );
begin

    Tmp   := SIN & I;
    Tmp   := SHR( Tmp, SH );
    return Tmp( I_left downto 0 );

end;

  BEGIN

  control_out_write <= '0';
  control_out <= "00000000000000000000000000000000";
  data_out_write <= '0';
  data_out <= "00000000000000000000000000000000";
  data_in_read <= '0';
  control_in_read <= '0';

  buf_r_th_sig <= buf_r_th;
  buf_r_tl_sig <= buf_r_tl;
  ctlnbin_sig <= ctlnbin;
  ctlnbdatawordread_sig <= ctlnbdatawordread;
  ctlnbout_sig <= ctlnbout;
  dec_comp_sig <= dec_comp;
  dec_fault_sig <= dec_fault;
  control_sig <= control;
  dec_cbp_sig <= dec_cbp;
  skipcount_sig <= skipcount;
  buf_out_sz_sig <= buf_out_sz;
  buf_valid_sig <= buf_valid;
  buf_r_h_sig <= buf_r_h;
  buf_r_l_sig <= buf_r_l;
  dec_ieq0_sig <= dec_ieq0;
  lc_state_sig <= lc_state;
  dec_value_sig <= dec_value;
  buf_flushbits_tmp_sig <= buf_flushbits_tmp;
  lc_write_code5_sig <= lc_write_code5;
  lc_write_code7_sig <= lc_write_code7;
  lc_write_size_sig <= lc_write_size;
  lc_write_val_sig <= lc_write_val;
  dec_calcaddr_error_sig <= dec_calcaddr_error;
  dec_calcaddr_addr_sig <= dec_calcaddr_addr;
  dec_calcaddr_dec_in_sig <= dec_calcaddr_dec_in;
  tmp_sig <= tmp;
  tmp_wired_8_sig <= tmp_wired_8;
  val_sig <= val;
  tblock_skip_tmp_sig <= tblock_skip_tmp;

  CASE CURRENT_STATE IS

    ------------------------------------------------
    WHEN asg_buf_r_tl_at_tblock_ughc_115 =>
      NEXT_STATE <= asg_ctlnbout_at_tblock_ughc_125;

      buf_r_tl_sig <= "000000";
      buf_r_th_sig <= "000000";

    ------------------------------------------------
    WHEN asg_ctlnbout_at_tblock_ughc_198 =>
      NEXT_STATE <= while_test_at_tblock_ughc_246;

      ctlnbout_sig <= (ctlnbout(15 downto 0) + "0000000000000001");

    ------------------------------------------------
    WHEN asg_dec_comp_at_tblock_ughc_154 =>
      NEXT_STATE <= while_test_at_tblock_ughc_244;

      dec_comp_sig <= (dec_comp(3 downto 0) + "0001");
      ctlnbout_sig <= (ctlnbout(15 downto 0) + "0000000000000001");
```

```
    dec_ieq0_sig <= '1';
    dec_cbp_sig <= (dec_cbp(5 downto 0) & '0');

  ------------------------------------------------
  WHEN test_at_tblock_ughc_200 =>
    IF (to_stdulogicvector(dec_value(18 downto 12)) = "1000001")
    THEN NEXT_STATE <= asg_buf_out_sz_at_tblock_ughc_202;
    ELSE NEXT_STATE <= asg_ugh_main_tmp_wired_8_at_tblock_ughc_224;
    END IF;

  ------------------------------------------------
  WHEN asg_buf_r_th_at_tblock_ughc_136 =>
    NEXT_STATE <= while_test_at_tblock_ughc_246;

    if ( ( (buf_flushbits_tmp = '1') ) )  then
        buf_r_th_sig <= (buf_r_th(5 downto 0)
               + buf_r_tl(5 downto 0)) - buf_out_sz(5 downto 0);
    end if;
    if ( ( not ((buf_flushbits_tmp = '1') ) ) )  then
        buf_r_th_sig <=
                 buf_r_th(5 downto 0) - buf_out_sz(5 downto 0);
    end if;
    if ( ( (buf_flushbits_tmp = '1') ) )  then
        buf_r_h_sig <= buf_r_l(31 downto 0);
    end if;
    if ( ( (buf_flushbits_tmp = '1') ) )  then
        buf_r_tl_sig <= "000000";
    end if;

  ------------------------------------------------
  WHEN wait_at_tblock_ughc_260 =>
    IF ( (NOT(start) = '1') )
    THEN NEXT_STATE <= wait_at_tblock_ughc_260;
    ELSE NEXT_STATE <= asg_buf_r_tl_at_tblock_ughc_115;
    END IF;

  ------------------------------------------------
  WHEN while_test_at_tblock_ughc_246 =>
    IF ( NOT((to_stdulogicvector(dec_comp(3 downto 0)) /= "0110")
       AND (NOT(dec_fault) = '1')) )
    THEN NEXT_STATE <= asg_ugh_main_tmp_at_tblock_ughc_247;
    ELSE NEXT_STATE <= asg_buf_out_sz_at_tblock_ughc_140;
    END IF;

  ------------------------------------------------
  WHEN asg_lc_write_code5_at_tblock_ughc_153 =>
    ugh_tmp_unsigned_83(31 downto 0) :=
       (buf_r_l(31 downto 27) & "---------------------------");
    ugh_tmp_unsigned_84(31 downto 0) :=
       catshr( ugh_tmp_unsigned_83(31 downto 0),
                buf_r_th(5 downto 0),buf_r_h(31 downto 0) );

    NEXT_STATE <= asg_lc_write_code7_at_tblock_ughc_153;

    lc_write_code5_sig <= ugh_tmp_unsigned_84(31 downto 27);
    lc_state_sig <= '0';

  ------------------------------------------------
  WHEN call_ugh_read_at_tblock_ughc_136 =>
    IF ( (NOT(data_in_rok) = '1') )
    THEN NEXT_STATE <= call_ugh_read_at_tblock_ughc_136;
    ELSE NEXT_STATE <= asg_ctlnbdatawordread_at_tblock_ughc_136;
    END IF;

    buf_r_l_sig <= data_in(31 downto 0);
    data_in_read <= '1';

  ------------------------------------------------
  WHEN call_ugh_write_at_tblock_ughc_198 =>
    IF ( (NOT(data_out_wok) = '1') )
    THEN NEXT_STATE <= call_ugh_write_at_tblock_ughc_198;
    ELSE NEXT_STATE <= asg_ctlnbout_at_tblock_ughc_198;
    END IF;

    data_out <= tmp(31 downto 0);
    data_out_write <= '1';

  ------------------------------------------------
  WHEN call_ugh_write_at_tblock_ughc_164 =>
    IF ( (NOT(data_out_wok) = '1') )
    THEN NEXT_STATE <= call_ugh_write_at_tblock_ughc_164;
    ELSE NEXT_STATE <= asg_dec_comp_at_tblock_ughc_165;
    END IF;

    data_out <= lc_write_val(31 downto 0);
    data_out_write <= '1';
                  .
                  .
                  .
                  .
  END CASE;


  IF ( ((reset = '1') AND (abort = '1')) )
  THEN NEXT_STATE <= wait_at_tblock_ughc_260;
  ELSIF ( (reset = '0') )
  THEN NEXT_STATE <= beg_func_at_tblock_ughc_260;
  END IF;

  END PROCESS;


--------------------------------------------------------
--------------------------------------------------------
  PROCESS( ck )
  BEGIN
    IF ( ((ck = '1') AND (ck'EVENT)) ) THEN
      CURRENT_STATE <= NEXT_STATE;

      buf_r_th <= buf_r_th_sig;
      buf_r_tl <= buf_r_tl_sig;
      ctlnbin <= ctlnbin_sig;
      ctlnbdatawordread <= ctlnbdatawordread_sig;
      ctlnbout <= ctlnbout_sig;
      dec_comp <= dec_comp_sig;
      dec_fault <= dec_fault_sig;
      control <= control_sig;
      dec_cbp <= dec_cbp_sig;
      skipcount <= skipcount_sig;
      buf_out_sz <= buf_out_sz_sig;
      buf_valid <= buf_valid_sig;
      buf_r_h <= buf_r_h_sig;
      buf_r_l <= buf_r_l_sig;
      dec_ieq0 <= dec_ieq0_sig;
      lc_state <= lc_state_sig;
      dec_value <= dec_value_sig;
      buf_flushbits_tmp <= buf_flushbits_tmp_sig;
      lc_write_code5 <= lc_write_code5_sig;
      lc_write_code7 <= lc_write_code7_sig;
      lc_write_size <= lc_write_size_sig;
      lc_write_val <= lc_write_val_sig;
      dec_calcaddr_error <= dec_calcaddr_error_sig;
      dec_calcaddr_addr <= dec_calcaddr_addr_sig;
      dec_calcaddr_dec_in <= dec_calcaddr_dec_in_sig;
      tmp <= tmp_sig;
      tmp_wired_8 <= tmp_wired_8_sig;
      val <= val_sig;
      tblock_skip_tmp <= tblock_skip_tmp_sig;
    END IF;
  END PROCESS;
END;
```

C.2 La description VHDL RTL $n^o 2$

Le chemin de données est une description comportementale distincte de l'automate (cf. figure C.2).

Les états sont déjà encodés via le signal *sig_state*. L'automate possède la *fonction de transition*. L'automate est synchrone car la mise à jour du registre d'états se fait dans un processus sensible uniquement à l'horloge.

L'automate contrôle aussi l'écriture dans les registres du chemin de données pour permettre un

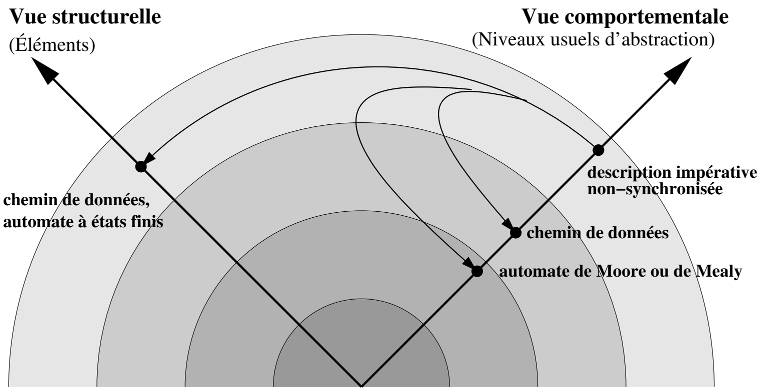


FIG. C.2 Chemin de données comportemental séparé de l'automate

éventuel *réordonnancement* de *ugh*.
En effet, si l'on fait durer sur plusieurs cycles un état pour satisfaire les contraintes temporelles, c'est seulement au dernier cycle que l'écriture dans le registre doit être effective (cf. chapitre 4.4.1). La commande d'écriture des registres du chemin de données doit donc être contrôlée par l'automate réordonnancé.

Description RTL de l'automate de contrôle

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY tblock_fsm IS
PORT
(
  cmd_up_out_ctl: OUT STD_LOGIC;
  cmd_dn_in_ctl: OUT STD_LOGIC;
  data_up_out_ctl: OUT STD_LOGIC;
  data_dn_in_ctl: OUT STD_LOGIC;
  buf_flushbits_tmp_ctl: OUT STD_LOGIC;
  tmp_ctl: OUT STD_LOGIC;
  ctlnbout_ctl: OUT STD_LOGIC;
  ctlnbin_ctl: OUT STD_LOGIC;
  dec_fault_ctl: OUT STD_LOGIC;
  dec_calcaddr_addr_ctl: OUT STD_LOGIC;
  ugh_test_52: IN STD_LOGIC;
  ugh_test_51: IN STD_LOGIC;
  ugh_test_50: IN STD_LOGIC;
  cmd_dn_in_rok: IN STD_LOGIC;
  data_up_out_wok: IN STD_LOGIC;
  data_dn_in_rok: IN STD_LOGIC;
  cmd_up_out_wok: IN STD_LOGIC;
  ck: IN STD_LOGIC;
  start: IN STD_LOGIC;
  reset: IN STD_LOGIC;
  abort: IN STD_LOGIC;
  sig_state : OUT STD_LOGIC_VECTOR(6 DOWNTO 0)
            .
            .
            .
            .
);
END tblock_fsm;

ARCHITECTURE FSM OF tblock_fsm IS

TYPE STATE_TYPE IS
(
  beg_func_at_tblock_ughc_260, wait_at_tblock_ughc_260,
            .
            .
            .
            .
);

SIGNAL CURRENT_STATE, NEXT_STATE: STATE_TYPE;

BEGIN
```

```
  PROCESS( CURRENT_STATE, abort, reset, start,
 ugh_test_48, ugh_test_49,
           .
           .
           .
           .
)
  BEGIN

  CASE CURRENT_STATE IS

    WHEN asg_dec_comp_at_tblock_ughc_154 =>
      NEXT_STATE <= while_test_at_tblock_ughc_244;
      sig_state <= "0100100";
      cmd_up_out_ctl <= '0';
      cmd_dn_in_ctl <= '0';
      data_up_out_ctl <= '0';
      data_dn_in_ctl <= '0';
      buf_flushbits_tmp_ctl <= '0';
      tmp_ctl <= '0';
      ctlnbin_ctl <= '0';
      dec_fault_ctl <= '0';
      dec_calcaddr_addr_ctl <= '0';
      dec_calcaddr_error_ctl <= '0';
      dec_calcaddr_dec_in_ctl <= '0';
      ctlnbdatawordread_ctl <= '0';
      buf_r_tl_ctl <= '0';
      buf_r_th_ctl <= '0';
      buf_r_l_ctl <= '0';
      buf_r_h_ctl <= '0';
      buf_valid_ctl <= '0';
      buf_out_sz_ctl <= '0';
      dec_value_ctl <= '0';
      val_ctl <= '0';
      control_ctl <= '0';
      lc_write_code5_ctl <= '0';
      lc_write_code7_ctl <= '0';
      lc_state_ctl <= '0';
      lc_write_size_ctl <= '0';
      lc_write_val_ctl <= '0';
      tblock_skip_tmp_ctl <= '0';
      skipcount_ctl <= '0';
      tmp_wired_8_ctl <= '0';
      ctlnbout_ctl <= '1';
      dec_ieq0_ctl <= '1';
      dec_cbp_ctl <= '1';
      dec_comp_ctl <= '1';

    WHEN while_test_at_tblock_ughc_244 =>
```

```
      IF    ( (ugh_test_49 = '1') )
      THEN NEXT_STATE <= while_test_at_tblock_ughc_246;
      ELSE NEXT_STATE <= asg_buf_out_sz_at_tblock_ughc_183;
      END IF;
      sig_state <= "0110010";
      cmd_up_out_ctl <= '0';
      cmd_dn_in_ctl <= '0';
      data_up_out_ctl <= '0';
      data_dn_in_ctl <= '0';
      buf_flushbits_tmp_ctl <= '0';
      tmp_ctl <= '0';
      ctlnbout_ctl <= '0';
      ctlnbin_ctl <= '0';
      dec_fault_ctl <= '0';
      dec_calcaddr_addr_ctl <= '0';
      dec_calcaddr_error_ctl <= '0';
      dec_calcaddr_dec_in_ctl <= '0';
      ctlnbdatawordread_ctl <= '0';
      buf_r_tl_ctl <= '0';
      buf_r_th_ctl <= '0';
      buf_r_l_ctl <= '0';
      buf_r_h_ctl <= '0';
      buf_valid_ctl <= '0';
      buf_out_sz_ctl <= '0';
      dec_value_ctl <= '0';
      val_ctl <= '0';
      dec_ieq0_ctl <= '0';
      control_ctl <= '0';
      dec_cbp_ctl <= '0';
      dec_comp_ctl <= '0';
      lc_write_code5_ctl <= '0';
      lc_write_code7_ctl <= '0';
      lc_state_ctl <= '0';
      lc_write_size_ctl <= '0';
      lc_write_val_ctl <= '0';
      tblock_skip_tmp_ctl <= '0';
      skipcount_ctl <= '0';
      tmp_wired_8_ctl <= '0';

    WHEN asg_buf_out_sz_at_tblock_ughc_183 =>
      IF    ( (NOT(ugh_test_36) = '1') )
      THEN NEXT_STATE <= while_test_at_tblock_ughc_246;
      ELSE NEXT_STATE <= test_at_tblock_ughc_187;
      END IF;
      sig_state <= "0110011";
      cmd_up_out_ctl <= '0';
      cmd_dn_in_ctl <= '0';
      data_up_out_ctl <= '0';
      data_dn_in_ctl <= '0';
      buf_flushbits_tmp_ctl <= '0';
      tmp_ctl <= '0';
      ctlnbout_ctl <= '0';
      ctlnbin_ctl <= '0';
      dec_fault_ctl <= '0';
      dec_calcaddr_addr_ctl <= '0';
      dec_calcaddr_error_ctl <= '0';
      dec_calcaddr_dec_in_ctl <= '0';
      ctlnbdatawordread_ctl <= '0';
      buf_r_tl_ctl <= '0';
      buf_r_th_ctl <= '0';
      buf_r_l_ctl <= '0';
      buf_r_h_ctl <= '0';
      buf_valid_ctl <= '0';
      dec_value_ctl <= '0';
      val_ctl <= '0';
      dec_ieq0_ctl <= '0';
      control_ctl <= '0';
      dec_cbp_ctl <= '0';
      dec_comp_ctl <= '0';
      lc_write_code5_ctl <= '0';
      lc_write_code7_ctl <= '0';
      lc_state_ctl <= '0';
      lc_write_size_ctl <= '0';
      lc_write_val_ctl <= '0';
      tblock_skip_tmp_ctl <= '0';
      skipcount_ctl <= '0';
      tmp_wired_8_ctl <= '0';
      buf_out_sz_ctl <= '1';

    WHEN call_ugh_write_at_tblock_ughc_256 =>
      IF    ( (cmd_up_out_wok = '1') )
      THEN NEXT_STATE <= asg_ctlnbout_at_tblock_ughc_125;
      ELSE NEXT_STATE <= call_ugh_write_at_tblock_ughc_256;
      END IF;
      sig_state <= "1011000";
      cmd_dn_in_ctl <= '0';
      data_up_out_ctl <= '0';
      data_dn_in_ctl <= '0';
      buf_flushbits_tmp_ctl <= '0';
      tmp_ctl <= '0';
      ctlnbout_ctl <= '0';
      ctlnbin_ctl <= '0';
      dec_fault_ctl <= '0';
      dec_calcaddr_addr_ctl <= '0';
      dec_calcaddr_error_ctl <= '0';
      dec_calcaddr_dec_in_ctl <= '0';
      ctlnbdatawordread_ctl <= '0';
      buf_r_tl_ctl <= '0';
      buf_r_th_ctl <= '0';
      buf_r_l_ctl <= '0';
      buf_r_h_ctl <= '0';
      buf_valid_ctl <= '0';
      buf_out_sz_ctl <= '0';
      dec_value_ctl <= '0';
      val_ctl <= '0';
      dec_ieq0_ctl <= '0';
      control_ctl <= '0';
      dec_cbp_ctl <= '0';
      dec_comp_ctl <= '0';
      lc_write_code5_ctl <= '0';
      lc_write_code7_ctl <= '0';
      lc_state_ctl <= '0';
      lc_write_size_ctl <= '0';
      lc_write_val_ctl <= '0';
      tblock_skip_tmp_ctl <= '0';
      skipcount_ctl <= '0';
      tmp_wired_8_ctl <= '0';
      cmd_up_out_ctl <= '1';
            .
            .
            .
            .
  END CASE;

  IF    ( ((reset = '1') AND (abort = '1')) )
  THEN NEXT_STATE <= wait_at_tblock_ughc_260;
  ELSIF ( (reset = '0') )
  THEN NEXT_STATE <= beg_func_at_tblock_ughc_260;
  END IF;

  END PROCESS;


  PROCESS( ck )
  BEGIN
    IF ( ((ck = '1') AND (ck'EVENT)) )  THEN
        CURRENT_STATE <= NEXT_STATE;
    END IF;
  END PROCESS;
END;
```

Le chemin de données est composé d'actions mutuellement exclusives dont les conditions d'exécution dépendent du signal *sig_state*. C'est la synthèse RTL qui décide de l'allocation et de l'affectation de ces opérations.

Les registres du chemin de données sont des bascules car leur écriture est dans un processus VHDL sensible uniquement à l'horloge. Leur écriture est aussi conditionnée par l'autorisation d'écriture venant de l'automate.

Description RTL des états comportementaux

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY tblock_dp IS
PORT
(
  sig_state : OUT STD_LOGIC_VECTOR(6 DOWNTO 0);
  cmd_up_out : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
  cmd_dn_in : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
  data_up_out : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
  data_dn_in : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
  ck: IN STD_LOGIC;
  cmd_up_out_wok: IN STD_LOGIC;
  cmd_up_out_write: OUT STD_LOGIC;
  ugh_test_cmd_up_out_wok: OUT STD_LOGIC;
  data_dn_in_rok: IN STD_LOGIC;
  data_dn_in_ctl: IN STD_LOGIC;
  ctlnbout_ctl : IN STD_LOGIC;
  dec_fault_ctl : IN STD_LOGIC;
  buf_r_l_ctl : IN STD_LOGIC;
  dec_ieq0_ctl : IN STD_LOGIC;
  lc_state_ctl : IN STD_LOGIC;
  ugh_test_51: OUT STD_LOGIC;
  ugh_test_52: INOUT STD_LOGIC;
                    .
                    .
                    .
                    .
);
END tblock_dp;

ARCHITECTURE tblock_dp OF tblock_dp IS

SIGNAL  buf_r_th, buf_r_th_sig :  STD_LOGIC_VECTOR(5 DOWNTO 0) ;
SIGNAL  buf_r_tl, buf_r_tl_sig :  STD_LOGIC_VECTOR(5 DOWNTO 0) ;
SIGNAL  tblock_skip_tmp, tblock_skip_tmp_sig :
                                  STD_LOGIC_VECTOR(24 DOWNTO 0) ;
                    .
                    .
                    .
                    .
BEGIN

  PROCESS( sig_state )
VARIABLE  ugh_tmp_unsigned_122 :  STD_LOGIC_VECTOR(31 DOWNTO 0) ;
VARIABLE  ugh_tmp_unsigned_0 :  STD_LOGIC_VECTOR(6 DOWNTO 0) ;
TYPE dec_comp_rom_type IS ARRAY (0 TO 15) OF STD_LOGIC;
CONSTANT  dec_comp_rom :  dec_comp_rom_type := ( '1', '1', '1', '1',
'0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0' );
                    .
                    .
                    .
                    .

  BEGIN

cmd_up_out_write <= '0';
cmd_up_out <= "00000000000000000000000000000000";
data_up_out_write <= '0';
data_up_out <= "00000000000000000000000000000000";
data_dn_in_read <= '0';
cmd_dn_in_read <= '0';
ugh_test_52 <= '0';
ugh_test_51 <= '0';
ugh_test_50 <= '0';
ugh_test_48 <= '0';
ugh_test_49 <= '0';
ugh_test_46 <= '0';
ugh_test_47 <= '0';
ugh_test_42 <= '0';
ugh_test_43 <= '0';
ugh_test_44 <= '0';
ugh_test_45 <= '0';
ugh_test_38 <= '0';
ugh_test_39 <= '0';
ugh_test_41 <= '0';
ugh_test_36 <= '0';
ugh_test_34 <= '0';
ugh_test_33 <= '0';
ugh_test_32 <= '0';
ugh_test_30 <= '0';

buf_r_th_sig <= buf_r_th;
buf_r_tl_sig <= buf_r_tl;
ctlnbin_sig <= ctlnbin;
ctlnbdatawordread_sig <= ctlnbdatawordread;
ctlnbout_sig <= ctlnbout;
dec_comp_sig <= dec_comp;
dec_fault_sig <= dec_fault;
control_sig <= control;
dec_cbp_sig <= dec_cbp;
skipcount_sig <= skipcount;
buf_out_sz_sig <= buf_out_sz;
buf_valid_sig <= buf_valid;
buf_r_h_sig <= buf_r_h;
buf_r_l_sig <= buf_r_l;
dec_ieq0_sig <= dec_ieq0;
lc_state_sig <= lc_state;
dec_value_sig <= dec_value;
buf_flushbits_tmp_sig <= buf_flushbits_tmp;
lc_write_code5_sig <= lc_write_code5;
lc_write_code7_sig <= lc_write_code7;
lc_write_size_sig <= lc_write_size;
lc_write_val_sig <= lc_write_val;
dec_calcaddr_error_sig <= dec_calcaddr_error;
dec_calcaddr_addr_sig <= dec_calcaddr_addr;
dec_calcaddr_dec_in_sig <= dec_calcaddr_dec_in;
tmp_sig <= tmp;
tmp_wired_8_sig <= tmp_wired_8;
val_sig <= val;
tblock_skip_tmp_sig <= tblock_skip_tmp;


    -- beg_func_at_tblock_ughc_260
    IF ( sig_state = "0000000" ) THEN

    -- asg_buf_flushbits_tmp_at_tblock_ughc_136
    ELSIF ( sig_state = "0000110" ) THEN
      buf_flushbits_tmp_sig <= to_stdulogic(
        (buf_r_th(5 downto 0)) <= (buf_out_sz(5 downto 0)) );
      ctlnbin_sig <= (ctlnbin(15 downto 0)
                  + ("0000000000" & buf_out_sz(5 downto 0)));

    -- asg_ctlnbout_at_tblock_ughc_221
    ELSIF ( sig_state = "1001010" ) THEN
      ctlnbout_sig <= (ctlnbout(15 downto 0) + "0000000000000001");
                    .
                    .
                    .
                    .
    END IF;

  END PROCESS;


  PROCESS( ck )
  BEGIN
    IF ( ((ck = '1') AND (ck'EVENT)) ) THEN
      IF ( buf_r_th_ctl = '1' ) THEN buf_r_th <= buf_r_th_sig; END IF;
      IF ( buf_r_tl_ctl = '1' ) THEN buf_r_tl <= buf_r_tl_sig; END IF;
      IF ( ctlnbin_ctl = '1' ) THEN ctlnbin <= ctlnbin_sig; END IF;
      IF ( ctlnbout_ctl = '1' ) THEN ctlnbout <= ctlnbout_sig; END IF;
      IF ( dec_comp_ctl = '1' ) THEN dec_comp <= dec_comp_sig; END IF;
      IF ( control_ctl = '1' ) THEN control <= control_sig; END IF;
      IF ( dec_cbp_ctl = '1' ) THEN dec_cbp <= dec_cbp_sig; END IF;
      IF ( buf_r_h_ctl = '1' ) THEN buf_r_h <= buf_r_h_sig; END IF;
      IF ( buf_r_l_ctl = '1' ) THEN buf_r_l <= buf_r_l_sig; END IF;
      IF ( dec_ieq0_ctl = '1' ) THEN dec_ieq0 <= dec_ieq0_sig; END IF;
      IF ( lc_state_ctl = '1' ) THEN lc_state <= lc_state_sig; END IF;
                    .
                    .
                    .
                    .
    END IF;
  END PROCESS;
END;
```

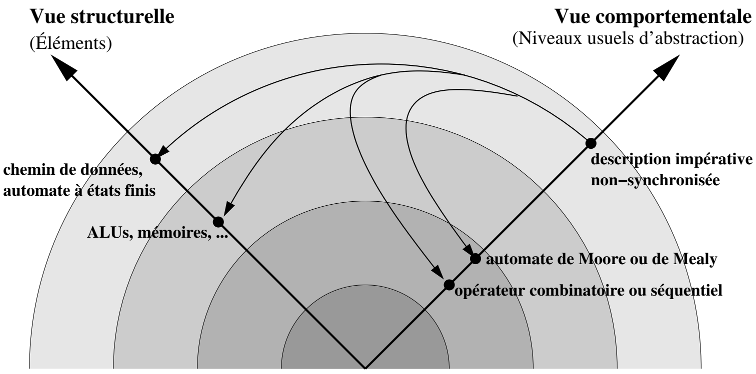


FIG. C.3 Chemin de données structurel séparé de l'automate

C.3 La description VHDL RTL $n^o 3$

Le chemin de données est une description structurelle distincte de l'automate (cf. figure C.3). L'*affectation des opérateurs combinatoires* est faite.

L'automate est un automate de contrôle dirigeant les commandes d'écritures des registres mais aussi les commandes des multiplexeurs et des opérateurs arithmétiques.

Description RTL de l'automate de contrôle

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY vld_fsm IS
PORT
(
  cmd_up_out_ctl: OUT STD_LOGIC;
  cmd_dn_in_ctl: OUT STD_LOGIC;
  data_up_out_ctl: OUT STD_LOGIC;
  data_dn_in_ctl: OUT STD_LOGIC;
  tmp_ctl: OUT STD_LOGIC;
  ctlnbout_ctl: OUT STD_LOGIC;
  ctlnbin_ctl: OUT STD_LOGIC;
  dec_fault_ctl: OUT STD_LOGIC;
  dec_calcaddr_addr_ctl: OUT STD_LOGIC;
  tblock_skip_tmp_ctl: OUT STD_LOGIC;
  skipcount_ctl: OUT STD_LOGIC;
  alu1_op : OUT STD_LOGIC_VECTOR(2 DOWNTO 0) ;
  ugh_mux_cat_shr1_sin_s : OUT STD_LOGIC_VECTOR(8 DOWNTO 0) ;
  ugh_mux_dec_ieq0_d_s : OUT STD_LOGIC_VECTOR(1 DOWNTO 0) ;
  ugh_mux_buf_out_sz_d_s : OUT STD_LOGIC_VECTOR(7 DOWNTO 0) ;
  ugh_mux_buf_r_th_d_s : OUT STD_LOGIC_VECTOR(1 DOWNTO 0) ;
  ugh_mux_buf_r_tl_d_s : OUT STD_LOGIC_VECTOR(1 DOWNTO 0) ;
  ugh_mux_tmp_d_s : OUT STD_LOGIC_VECTOR(6 DOWNTO 0) ;
  ugh_mux_data_up_out_cosy_i_s : OUT STD_LOGIC_VECTOR(1 DOWNTO 0) ;
  ugh_test_52: IN STD_LOGIC;
  ugh_test_51: IN STD_LOGIC;
  ugh_test_49: IN STD_LOGIC;
  ugh_test_48: IN STD_LOGIC;
  ugh_test_47: IN STD_LOGIC;
  ugh_test_46: IN STD_LOGIC;
  ugh_test_45: IN STD_LOGIC;
  ugh_test_12: IN STD_LOGIC;
  ugh_test_11: IN STD_LOGIC;
  ugh_test_10: IN STD_LOGIC;
  cmd_dn_in_rok: IN STD_LOGIC;
  data_up_out_wok: IN STD_LOGIC;
  data_dn_in_rok: IN STD_LOGIC;
  cmd_up_out_wok: IN STD_LOGIC;
  ck: IN STD_LOGIC;
  start: IN STD_LOGIC;
  reset: IN STD_LOGIC;
  abort: IN STD_LOGIC;
                    .
                    .
                    .
                    .
);
```

```
END vld_fsm;

ARCHITECTURE FSM OF vld_fsm IS

TYPE STATE_TYPE IS
(
  beg_func_at_tblock_ughc_260, wait_at_tblock_ughc_260,
  call_ugh_read_at_tblock_ughc_128, asg_control_at_tblock_ughc_132,
                    .
                    .
                    .
                    .
);

SIGNAL CURRENT_STATE, NEXT_STATE: STATE_TYPE;

BEGIN
  PROCESS( CURRENT_STATE, abort, reset, start, ck, cmd_up_out_wok,
           data_dn_in_rok, data_up_out_wok, cmd_dn_in_rok,
            ugh_test_2, ugh_test_3, ugh_test_6,
                    .
                    .
                    .
                    .
  )
  BEGIN

  CASE CURRENT_STATE IS

    WHEN wait_at_tblock_ughc_260 =>
      IF    ( (NOT(start) = '1') )
      THEN NEXT_STATE <= wait_at_tblock_ughc_260;
      ELSE NEXT_STATE <= asg_buf_r_tl_at_tblock_ughc_115;
      END IF;
      cmd_up_out_ctl <= '0';
      cmd_dn_in_ctl <= '0';
      data_up_out_ctl <= '0';
      data_dn_in_ctl <= '0';
      tmp_ctl <= '0';
      ctlnbout_ctl <= '0';
      ctlnbin_ctl <= '0';
      dec_fault_ctl <= '0';
      dec_calcaddr_addr_ctl <= '0';
      dec_calcaddr_error_ctl <= '0';
      dec_calcaddr_dec_in_ctl <= '0';
      ctlnbdatawordread_ctl <= '0';
      buf_r_tl_ctl <= '0';
```

```
      buf_r_th_ctl <= '0';
      buf_r_l_ctl <= '0';
      buf_r_h_ctl <= '0';
      buf_valid_ctl <= '0';
      buf_out_sz_ctl <= '0';
      dec_value_ctl <= '0';
      val_ctl <= '0';
      dec_ieq0_ctl <= '0';
      control_ctl <= '0';
      dec_cbp_ctl <= '0';
      dec_comp_ctl <= '0';
      lc_write_code5_ctl <= '0';
      lc_write_code7_ctl <= '0';
      lc_state_ctl <= '0';
      lc_write_size_ctl <= '0';
      lc_write_val_ctl <= '0';
      tblock_skip_tmp_ctl <= '0';
      skipcount_ctl <= '0';
      alu1_op <= "---";
      ugh_mux_cat_shr1_sin_s <= "---------";
      ugh_mux_buf_sub_wired_a_s <= "--";
      ugh_mux_alu1_a_s <= "----";
      ugh_mux_alu1_b_s <= "-----";
      ugh_mux_skipcount_d_s <= "--";
      ugh_mux_lc_write_val_d_s <= "---";
      ugh_mux_lc_state_d_s <= "--";
      ugh_mux_lc_write_code7_d_s <= "---";
      ugh_mux_dec_comp_d_s <= "--";
      ugh_mux_dec_cbp_d_s <= "--";
      ugh_mux_dec_ieq0_d_s <= "--";
      ugh_mux_buf_out_sz_d_s <= "--------";
      ugh_mux_buf_r_th_d_s <= "--";
      ugh_mux_buf_r_tl_d_s <= "--";
      ugh_mux_ctlnbdatawordread_d_s <= "--";
      ugh_mux_dec_calcaddr_error_d_s <= "--";
      ugh_mux_dec_calcaddr_addr_d_s <= "-----------";
      ugh_mux_dec_fault_d_s <= "---";
      ugh_mux_ctlnbin_d_s <= "--";
      ugh_mux_ctlnbout_d_s <= "--";
      ugh_mux_tmp_d_s <= "-------";
      ugh_mux_data_up_out_cosy_i_s <= "--";

    WHEN call_ugh_write_at_tblock_ughc_198 =>
      IF    ( (NOT(data_up_out_wok) = '1') )
      THEN NEXT_STATE <= call_ugh_write_at_tblock_ughc_198;
      ELSE NEXT_STATE <= asg_ctlnbout_at_tblock_ughc_198;
      END IF;
      cmd_up_out_ctl <= '0';
      cmd_dn_in_ctl <= '0';
      data_dn_in_ctl <= '0';
      tmp_ctl <= '0';
      ctlnbout_ctl <= '0';
      ctlnbin_ctl <= '0';
      dec_fault_ctl <= '0';
      dec_calcaddr_addr_ctl <= '0';
      dec_calcaddr_error_ctl <= '0';
      dec_calcaddr_dec_in_ctl <= '0';
      ctlnbdatawordread_ctl <= '0';
      buf_r_tl_ctl <= '0';
      buf_r_th_ctl <= '0';
      buf_r_l_ctl <= '0';
      buf_r_h_ctl <= '0';
      buf_valid_ctl <= '0';
      buf_out_sz_ctl <= '0';
      dec_value_ctl <= '0';
      val_ctl <= '0';
      dec_ieq0_ctl <= '0';
      control_ctl <= '0';
      dec_cbp_ctl <= '0';
      dec_comp_ctl <= '0';
      lc_write_code5_ctl <= '0';
      lc_write_code7_ctl <= '0';
      lc_state_ctl <= '0';
      lc_write_size_ctl <= '0';
      lc_write_val_ctl <= '0';
      tblock_skip_tmp_ctl <= '0';
      skipcount_ctl <= '0';
      alu1_op <= "---";
      ugh_mux_cat_shr1_sin_s <= "---------";
      ugh_mux_buf_sub_wired_a_s <= "--";
      ugh_mux_alu1_a_s <= "----";
      ugh_mux_alu1_b_s <= "-----";
      ugh_mux_skipcount_d_s <= "--";
      ugh_mux_lc_write_val_d_s <= "---";
      ugh_mux_lc_state_d_s <= "--";
      ugh_mux_lc_write_code7_d_s <= "---";
      ugh_mux_dec_comp_d_s <= "--";
      ugh_mux_dec_cbp_d_s <= "--";
      ugh_mux_dec_ieq0_d_s <= "--";
      ugh_mux_buf_out_sz_d_s <= "--------";
      ugh_mux_buf_r_th_d_s <= "--";
      ugh_mux_buf_r_tl_d_s <= "--";
      ugh_mux_ctlnbdatawordread_d_s <= "--";
      ugh_mux_dec_calcaddr_error_d_s <= "--";
      ugh_mux_dec_calcaddr_addr_d_s <= "-----------";
      ugh_mux_dec_fault_d_s <= "---";
      ugh_mux_ctlnbin_d_s <= "--";
      ugh_mux_ctlnbout_d_s <= "--";
      ugh_mux_tmp_d_s <= "-------";
      data_up_out_ctl <= '1';
      ugh_mux_data_up_out_cosy_i_s <= "10";

    WHEN test_at_tblock_ughc_200 =>
      IF    ( (NOT(ugh_test_6) = '1') )
      THEN NEXT_STATE <= asg_buf_out_sz_at_tblock_ughc_202;
      ELSE NEXT_STATE <= test_at_tblock_ughc_227;
      END IF;
      cmd_up_out_ctl <= '0';
      cmd_dn_in_ctl <= '0';
      data_up_out_ctl <= '0';
      data_dn_in_ctl <= '0';
      tmp_ctl <= '0';
      ctlnbout_ctl <= '0';
      ctlnbin_ctl <= '0';
      dec_fault_ctl <= '0';
      dec_calcaddr_addr_ctl <= '0';
      dec_calcaddr_error_ctl <= '0';
      dec_calcaddr_dec_in_ctl <= '0';
      ctlnbdatawordread_ctl <= '0';
      buf_r_tl_ctl <= '0';
      buf_r_th_ctl <= '0';
      buf_r_l_ctl <= '0';
      buf_r_h_ctl <= '0';
      buf_valid_ctl <= '0';
      buf_out_sz_ctl <= '0';
      dec_value_ctl <= '0';
      val_ctl <= '0';
      dec_ieq0_ctl <= '0';
      control_ctl <= '0';
      dec_cbp_ctl <= '0';
      dec_comp_ctl <= '0';
      lc_write_code5_ctl <= '0';
      lc_write_code7_ctl <= '0';
      lc_state_ctl <= '0';
      lc_write_size_ctl <= '0';
      lc_write_val_ctl <= '0';
      tblock_skip_tmp_ctl <= '0';
      skipcount_ctl <= '0';
      alu1_op <= "---";
      ugh_mux_cat_shr1_sin_s <= "---------";
      ugh_mux_buf_sub_wired_a_s <= "--";
      ugh_mux_alu1_a_s <= "----";
      ugh_mux_alu1_b_s <= "-----";
      ugh_mux_skipcount_d_s <= "--";
      ugh_mux_lc_write_val_d_s <= "---";
      ugh_mux_lc_state_d_s <= "--";
      ugh_mux_lc_write_code7_d_s <= "---";
      ugh_mux_dec_comp_d_s <= "--";
      ugh_mux_dec_cbp_d_s <= "--";
      ugh_mux_dec_ieq0_d_s <= "--";
      ugh_mux_buf_out_sz_d_s <= "--------";
      ugh_mux_buf_r_th_d_s <= "--";
      ugh_mux_buf_r_tl_d_s <= "--";
      ugh_mux_ctlnbdatawordread_d_s <= "--";
      ugh_mux_dec_calcaddr_error_d_s <= "--";
      ugh_mux_dec_calcaddr_addr_d_s <= "-----------";
      ugh_mux_dec_fault_d_s <= "---";
      ugh_mux_ctlnbin_d_s <= "--";
      ugh_mux_ctlnbout_d_s <= "--";
      ugh_mux_tmp_d_s <= "-------";
      ugh_mux_data_up_out_cosy_i_s <= "--";
                     .
                     .
                     .
                     .

  END CASE;

  IF    ( ((reset = '1') AND (abort = '1')) )
  THEN NEXT_STATE <= wait_at_tblock_ughc_260;
  ELSIF ( (reset = '0') )
  THEN NEXT_STATE <= beg_func_at_tblock_ughc_260;
  END IF;

  END PROCESS;


  PROCESS( ck )
  BEGIN
    IF ( ((ck = '1') AND (ck'EVENT)) )  THEN
        CURRENT_STATE <= NEXT_STATE;
      END IF;
  END PROCESS;
END;
```

Pour chaque instance d'opérateur, un modèle est généré. La généricité n'est pas utilisée.

Description RTL d'un modèle d'opérateur du chemin de données

```
Library IEEE;
Use     IEEE.STD_LOGIC_1164.All;
Use     IEEE.STD_LOGIC_ARITH.All;
Library DWARE;
Use     DWARE.DWpackages.All;
Library DW06;
Use     DW06.DW06_components.All;

ENTITY model_sub3 IS
    PORT (
    A          : in    STD_LOGIC_VECTOR (13 downto 0)       ;
    B          : in    STD_LOGIC_VECTOR (13 downto 0)       ;
    S          : out   STD_LOGIC_VECTOR (13 downto 0)       ;
    NS         : out   STD_LOGIC_VECTOR (13 downto 0)       ;
    NG         : out   STD_LOGIC                        ;
    CO         : out   STD_LOGIC
    );
END model_sub3;

Architecture SynoVhdl of model_sub3 is
    -- definition of signals (store the operation results)
    signal A_val   : STD_LOGIC_VECTOR (14 downto 0)       ;
    signal B_val   : STD_LOGIC_VECTOR (14 downto 0)       ;
    signal S_val   : STD_LOGIC_VECTOR (14 downto 0)       ;

-- function purpose: sub
function my_sub (a,b : std_logic_vector;ci : std_logic)
           return std_logic_vector is
variable S : std_logic_vector(a'length downto 0);
 begin  -- my_sub
 S := SIGNED(a&'0')-SIGNED(b&ci);
 return S(a'length downto 1);
end my_sub;

begin
    S <= S_val(S_val'length-2 downto 0);
    A_val <= '0'&A;
    B_val <= '0'&B;
    NS <= NOT S_val(S_val'length-2 downto 0);
    CO <= S_val(14);
    NG <= S_val(13);
    S_val <= my_sub(A_val,B_val,'0');
end SynoVhdl;
```

Le chemin de données est une vue structurelle de modèles. Un extrait du chemin de données du vld est présenté ci-dessous.

Description structurelle du chemin de données

```
library ieee;
use ieee.std_logic_1164.all;
entity vld_dp is
   port (
      ck                        : in    std_logic;
      control_out_ctl           : in    std_logic;
      control_in_ctl            : in    std_logic;
      data_out_ctl              : in    std_logic;
      data_in_ctl               : in    std_logic;
      tmp_ctl                   : in    std_logic;
      ctlnbout_ctl              : in    std_logic;
      lc_state_ctl              : in    std_logic;
      lc_write_size_ctl         : in    std_logic;
      lc_write_val_ctl          : in    std_logic;
      tblock_skip_tmp_ctl       : in    std_logic;
      skipcount_ctl             : in    std_logic;
      alu1_op                   : in    std_logic_vector(2 downto 0);
      ugh_mux_cat_shr1_sin_s    : in    std_logic_vector(8 downto 0);
      ugh_mux_ctlnbout_d_s      : in    std_logic_vector(1 downto 0);
      ugh_mux_tmp_d_s           : in    std_logic_vector(6 downto 0);
      ugh_test_52               : out   std_logic;
      ugh_test_49               : out   std_logic;
      ugh_test_48               : out   std_logic;
      ugh_test_9                : inout std_logic;
      ugh_test_7                : out   std_logic;
      ugh_test_6                : out   std_logic;
      ugh_test_3                : out   std_logic;
      ugh_test_2                : inout std_logic;
      ugh_test_control_in_rok   : out   std_logic;
      control_in_read           : out   std_logic;
      control_in_rok            : in    std_logic;
      ugh_test_data_out_wok     : out   std_logic;
      data_out_write            : out   std_logic;
      data_out_wok              : in    std_logic;
      ugh_test_data_in_rok      : out   std_logic;
      data_in_read              : out   std_logic;
      data_in_rok               : in    std_logic;
      ugh_test_control_out_wok  : out   std_logic;
      control_out_write         : out   std_logic;
      control_out_wok           : in    std_logic;
      data_in                   : in    std_logic_vector(31 downto 0);
      data_out                  : out   std_logic_vector(31 downto 0);
      control_in                : in    std_logic_vector(31 downto 0);
      control_out               : out   std_logic_vector(31 downto 0);
                     .
                     .
                     .
                     .
 );
end vld_dp;

architecture structural of vld_dp is

Component model_ugh_mux_buf_r_tl_d
   port (
      i0 : in    std_logic;
      i1 : in    std_logic;
      s  : in    std_logic_vector(1 downto 0);
      z  : out   std_logic
 );
end component;

Component model_cat_shr1
   port (
      i    : in    std_logic_vector(31 downto 0);
      sin  : in    std_logic_vector(31 downto 0);
      sh   : in    std_logic_vector(5 downto 0);
      z    : out   std_logic_vector(31 downto 0);
      nz   : out   std_logic_vector(31 downto 0)
 );
end component;

Component model_sub3
   port (
      a  : in    std_logic_vector(13 downto 0);
      b  : in    std_logic_vector(13 downto 0);
      s  : out   std_logic_vector(13 downto 0);
      ns : out   std_logic_vector(13 downto 0);
      ng : out   std_logic;
      co : out   std_logic
 );
end component;

Component model_buf_r_tl
   port (
      d    : in    std_logic_vector(5 downto 0);
      q    : out   std_logic_vector(5 downto 0);
      nq   : out   std_logic_vector(5 downto 0);
      ctl  : in    std_logic;
      ck   : in    std_logic
 );
end component;

Component model_fifo_data_out
   port (
      din     : in    std_logic_vector(31 downto 0);
      dout    : out   std_logic_vector(31 downto 0);
      full    : out   std_logic;
      extfull : in    std_logic;
      ctl     : in    std_logic;
      extctl  : out   std_logic
 );
```

```
end component;

Component model_fifo_control_out
   port (
      din     : in    std_logic_vector(31 downto 0);
      dout    : out   std_logic_vector(31 downto 0);
      full    : out   std_logic;
      extfull : in    std_logic;
      ctl     : in    std_logic;
      extctl  : out   std_logic
 );
end component;
                     .
                     .
                     .
                     .

signal fifo_control_in_cosy_o : std_logic_vector( 31 downto 0);
signal ctlnbout_q             : std_logic_vector( 15 downto 0);
signal dec_fault_nq           : std_logic_vector( 0 downto 0);
signal dec_calcaddr_addr_q    : std_logic_vector( 8 downto 0);
                     .
                     .
                     .
                     .

begin


ugh_mux_dec_fault_d : model_ugh_mux_dec_fault_d
   port map (
      i0 => ugh_test_9,
      i1 => ugh_cst_1_z(0),
      i2 => ugh_cst_0_z(0),
      s => ugh_mux_dec_fault_d_s(2 downto 0),
      z => ugh_mux_dec_fault_d_z(0)
   );

ugh_mux_dec_comp_d : model_ugh_mux_dec_comp_d
   port map (
      i0 => add1_s,
      i1 => ugh_cst_0000_z,
      s => ugh_mux_dec_comp_d_s(1 downto 0),
      z => ugh_mux_dec_comp_d_z
   );

ugh_cmp_13 : model_ugh_cmp_13
   port map (
      z => ugh_test_29,
      a => lc_write_size_q,
      b => ugh_cst_00000000_z
   );

cat_shr1 : model_cat_shr1
   port map (
      i(31 downto 14) => buf_r_l_q(31 downto 14),
      i(13 downto 0) => ugh_cst_00000000000000_z(13 downto 0),
      sin(31 downto 20) => ugh_mux_cat_shr1_sin_z(11 downto 0),
      sin(19 downto 0) => buf_r_h_q(19 downto 0),
      sh => buf_r_th_q,
      z(31) => ugh_test_2,
      z(30 downto 0) => cat_shr1_z(30 downto 0),
      nz => cat_shr1_nz
   );

sub3 : model_sub3
   port map (
      a => ugh_cst_00000000011111_z,
      b => skipcount_q,
      s => sub3_s,
      ns => sub3_ns,
      ng => sub3_ng(0),
      co => ugh_test_48
   );

add1 : model_add1
   port map (
      a => ugh_cst_0001_z,
      b => dec_comp_q,
      s => add1_s,
      ns => add1_ns,
      ng => add1_ng(0),
      co => add1_co(0)
   );

dec_rom : model_dec_rom
   port map (
      z => dec_rom_z,
      nz => dec_rom_nz,
      i => dec_calcaddr_addr_q
   );

dec_value : model_dec_value
   port map (
      d => dec_rom_z,
      q => dec_value_q,
      nq => dec_value_nq,
      ctl => dec_value_ctl,
      ck => ck
   );

buf_r_tl : model_buf_r_tl
   port map (
      d(5) => ugh_mux_buf_r_tl_d_z(0),
      d(4 downto 0) => ugh_cst_00000_z(4 downto 0),
      q => buf_r_tl_q,
      nq => buf_r_tl_nq,
      ctl => buf_r_tl_ctl,
      ck => ck
   );

fifo_data_in : model_fifo_data_in
   port map (
      din => data_in(31 downto 0),
      dout => fifo_data_in_cosy_o,
      empty => ugh_test_data_in_rok,
      extempty => data_in_rok,
      ctl => data_in_ctl,
      extctl => data_in_read
   );

fifo_control_out : model_fifo_control_out
   port map (
      din => tmp_q,
      dout => control_out(31 downto 0),
      full => ugh_test_control_out_wok,
      extfull => control_out_wok,
      ctl => control_out_ctl,
      extctl => control_out_write
   );

                     .
                     .
                     .
                     .

end structural;
```