Optimisation et compromis surface-vitesse dans le compilateur de silicium SYCO

Nourouddine Bekkara

THESE

présentée à

l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

pour obtenir

le titre de DOCTEUR-INGENIEUR en informatique

par

Nourouddine BEKKARA

Optimisation et compromis surface-vitesse dans le compilateur de silicium SYCO

Soutenue le : 19 octobre 1987 devant la commission d'examen.

JURY

Mr. J. Della Dora	Président
Mr. B. Courtois	Examinateurs
Mr. A. Jerraya	
Mr. J.P. Moreau	
Mr. G. Sagnes	

Thèse préparée au sein du laboratoire IMAG/TIM3

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : Georges LESPINARD **Année 1987**

Vice-Présidents : Jean-Marie PIERRARD
Jean-Pierre VERJUS

Professeurs des Universités

BARIBAUD	Michel	ENSERG
BARRAUD	Alain	ENSIEG
BAUDELET	Bernard	ENSPG
BEAUFILS	Jean-Pierre	ENSEEG
BESSON	Jean	ENSEEG
BLIMAN	Samuel	ENSERG
BLOCH	Daniel	ENSPG
BOIS	Philippe	ENSHMG
BONNETAIN	Lucien	ENSEEG
BOUVARD	Maurice	ENSHMG
BRISSONEAU	Pierre	ENSIEG
BRUNET	Yves	IUFA
BUYLE-BODIN	Maurice	ENSERG
CAILLERIE	Denis	ENSHMG
CAVAIGNAC	Jean-François	ENSPG
CHARTIER	Germain	ENSPG
CHENEVIER	Pierre	ENSERG
CHERADAME	Hervé	UFR PGP
CHERUY	Arlette	ENSIEG
CHIAVERINA	Jean	UFR PGP
CHOVET	Alain	ENSERG
COHEN	Joseph	ENSERG
COUMES	André	ENSERG
DARVE	Félix	ENSHMG
DELLA-DORA	Jean-François	ENSIMAG
DEPORTES	Jacques	ENSPG
DOLMAZON	Jean-Marc	ENSERG
DURAND	Francis	ENSEEG
DURAND	Jean-Louis	ENSIEG
FONLUPT	Jean	ENSIMAG
FOULARD	Claude	ENSIEG
GANDINI	Alessandro	UFR PGP
GAUBERT	Claude	ENSPG
GENTIL	Pierre	ENSERG
GREVEN	Hélène	IUFA
GUERIN	Bernard	ENSERG
GUYOT	Pierre	ENSEEG
IVANES	Marcel	ENSIEG
JAUSSAUD	Pierre	ENSIEG
JOUBERT	Jean-Claude	ENSPG
JOURDAIN	Geneviève	ENSIEG
LACOUME	Jean-Louis	ENSIEG
LESIEUR	Marcel	ENSHMG
LESPINARD	Georges	ENSHMG
LONGEQUEUE	Jean-Pierre	ENSPG
LOUCHET	François	ENSEEG
MASSE	Philippe	ENSIEG
MASSELOT	Christian	ENSIEG
MAZARE	Guy	ENSIMAG
MOREAU	René	ENSHMG
MORET	Roger	ENSIEG
MOSSIERE	Jacques	ENSIMAG
OBLED	Charles	ENSHMG
OZIL	Patrick	ENSEEG
PARIAUD	Jean-Charles	ENSEEG
PAUTHENET	René	ENSIEG
PERRET	René	ENSIEG
PERRET	Robert	ENSIEG
PIAU	Jean-Michel	ENSHMG
POUPOT	Christian	ENSERG
RAMEAU	Jean-Jacques	ENSEEG
RENAUD	Maurice	UFR PGP
ROBERT	André	UFR PGP
ROBERT	François	ENSIMAG
SABONNADIERE	Jean-Claude	ENSIEG
SAUCIER	Gabrielle	ENSIMAG
SCHLENKER	Claire	ENSPG
SCHLENKER	Michel	ENSPG
SERMET	Pierre	ENSERG
SILVY	Jacques	UFR PGP
SIRIEYS	Pierre	ENSHMG
SOHM	Jean-Claude	ENSEEG
SOLER	Jean-Louis	ENSIMAG
SOUQUET	Jean-Louis	ENSEEG
TROMPETTE	Philippe	ENSHMG
VEILLON	Gérard	ENSIMAG
ZADWORNY	François	ENSERG

Professeur Université des Sciences Sociales (Grenoble II)

BOLLIET Louis

Personnes ayant obtenu le diplôme

d'HABILITATION A DIRIGER DES RECHERCHES

BECKER	Monique	DANES	Florin	GHIBAUDO	Gérard
BINDER	Zdenek	DEROO	Daniel		
CHASSERY	Jean-Marc	DIARD	Jean-Paul	LADET	Pierre
COEY	John	DION	Jean-Michel	LATOMBE	Claudine
COLINET	Catherine	DUGARD	Luc	LE GORREC	Bernard
COMMAULT	Christian	DURAND	Robert	MADAR	Roland
CORNUEJOLS	Gérard	GALERIE	Alain	MULLER	Jean
DALARD	Francis	GAUTHIER	jean-Paul	NGUYEN TRONG	Bernadette
		GENTIL	Sylviane	TCHUENTE	Maurice
		PLA	Fernand	VINCENT	Henri

Chercheurs du C.N.R.S

Directeurs de recherche 1ère Classe

CAILLET	Marcel	JORRAND	Philippe
CARRE	René	LANDAU	Ioan
FRUCHART	Robert	MARTIN	

Directeurs de recherche 2ème Classe

ALEMANY	Antoine	EUSTATHOPOULOS	Nicolas
ALLIBERT	Colette	JOUD	Jean-Charles
ALLIBERT	Michel	KAMARINOS	Georges
ANSARA	Ibrahim	KLEITZ	Michel
ARMAND	Michel	KOFMAN	Walter
BINDER	Gilbert	LEJEUNE	Gérard
BONNET	Roland	MERMET	Jean
BORNARD	Guy	MUNIER	Jacques
CALMET	Jacques	SENATEUR	Jean-Pierre
DAVID	René	SUERY	Michel
DRIOLE	Jean	TEDOSIU	
ESCUDIER	Pierre	WACK	Bernard

Personnalités agréées à titre permanent à diriger des travaux de recherche (décision du conseil scientifique)

E.N.S.E.E.G

BERNARD	Claude	MALMEJAC	Yves
CHATILLON	Catherine	MARTIN GARIN	Régina
CHATILLON	Christian	SAINFORT	Paul
COULON	Michel	SARRAZIN	Pierre
		SIMON	Jean-Paul
FOSIER	Panayotis	TOUZAIN	Philippe
HAMMOU	Abdelkader	URBAIN	Georges

E.N.S.E.R.G

BOREL	Joseph

E.N.S.I.E.G

DESCHIZEAUX	Pierre	PERARD	Jacques
GLANGEAUD	François	REINISCH	Raymond

E.N.S.H.G

BOIS	Daniel	ROWE	Alain
MICHEL	Jean-Marie	VAUCLIN	Michel

E.N.S.I.M.A.G

BERT	Didier		
COURTIN	Jacques		
COURTOIS	Bernard	SIFAKIS	Joseph

E.F.P.G

CHARUEL	Robert

C.E.N.G

CADET	Jean	NIFENECKER	Hervé
COEURE	Philippe	PERROUD	Paul
DELHAYE	Jean-Marc	PEUZIN	Jean-Claude
DUPUY	Michel	TAIEB	Maurice
JOUVE	Hubert	VINCENDON	Marc
NICOLAU	Yvan		

<u>Laboratoires extérieurs</u>

C.N.E.T

DEMOULIN	Eric	MERCKEL	Gérard
DEVINE		PAULEAU	Yves
GERBER	Roland		

****

Remerciements

Je tiens à remercier :

Mr. Bernard Courtois, Directeur du laboratoire IMAG/TIM3 pour m'avoir accueilli dans son laboratoire et pour avoir bien voulu me faire l'honneur de participer au jury de cette thèse.

Mr. Ahmed Jerraya, Chargé de recherches au CNRS pour l'intérêt constant qu'il a porté à mon travail.

Mr. Jean Della Dora, Profcsseur à l'ENSIMAG, de me faire l'honneur de présider le jury de cette thèse.

Mr. Georges Sagnes, Professeur à l'université des sciences et techniques du Languedoc, pour avoir accepté d'être rapporteur de ce travail.

Mr. Jean-Pierre Moreau, Chef du département aide à la conception de circuits intégrés de Thomson Efcis pour avoir accepté d'être membre de ce jury.

Tous mes amis et collègues de l'équipe d'architecture des ordinateurs pour toutes les discussions et les critiques qui m'ont permis de mener à bien cette thèse.

A mes parents

Résumé

Le but de cette thèse est l'élaboration d'une phase d'optimisation de haut niveau et de recherche d'un compromis surface-vitesse dans le compilateur de silicium SYCO.
Le système SYCO développé dans le groupe architecture des ordinateurs du laboratoire IMAG/TIM3 est un compilateur de silicium pour des circuits intégrés de type microprocesseur. SYCO génère les spécifications des masques d'un circuit constitué d'une partie opérative et d'une partie contrôle à partir de sa description comportementale.
L'optimisation consiste à détecter et à fusionner les instructions qui peuvent être exécutées en parallèle.
La recherche d'un compromis surface-vitesse consiste à échanger une perte de performance du circuit contre un gain de surface ou vice-versa. Un ensemble de transformations qui peuvent être appliquées à la description afin d'aboutir à une meilleure implantation du circuit sans changer son comportement original sont présentées.
Le résultat de cette étude se concrétise par la réalisation d'un ensemble de logiciels écrits en langage LISP, chacun de ces logiciels réalisant une transformation de la description. Pour évaluer l'utilité de ces transformations, deux exemples de microprocesseurs ont été choisis et examinés.

Mots clés : Compilateur de silicium, graphes, optimisation de haut niveau, compromis surface-vitesse, parallélisme, compactage de microcode, réduction du nombre d'états.

Abstract

This thesis deals with the high-level optimizations and area/performance trade-offs in the SYCO silicon compiler.
The SYCO design system is a silicon compiler for VLSI microprocessor-like circuits. It consists of a set of computer programs whose goals is to produce a complete design, given as input a behavioral description of the circuit to be designed.
Optimization step is done by detection and compaction of concurrently executable statements. Area/performances trade-offs are made by improving execution speed of the circuit while increasing its area and vice-versa. A set of transformations that could be performed on the description to make trade-offs without changing the original behavior of the circuit to be designed are presented.
A software package was implemented to perform this transformations at the behavioral level, and methods were determined to measure the changes caused by these transformations.
To evaluate the usefulness of the transformations developed in this thesis, two examples of microprocessors are selected and examined.

Key words : Silicon compilers, graphs, high-level optimizations, area/performance trade-off, parallelism, compaction of microcode, state reduction.

INTRODUCTION

Introduction

La conception assistée par ordinateur de circuits intégrés complexes a pris un essor magistral. Ceci n'est pas dû au hasard, mais correspond bien à la nécessité de rabaisser les coûts de production en concevant vite et bien, de façon sûre et systématique. L'augmentation de la fiabilité dans la conception de circuits intégrés passe forcément par la réduction de l'intervention humaine, et donc par l'utilisation d'outils logiciels offrant aux concepteurs une assistance efficace. Le facteur de rapidité est également primordial. Certains secteurs industriels font de plus en plus usage de circuits spécialisés pour des applications spécifiques, et il devient impératif de pouvoir concevoir des circuits rapidement au moindre coût.

Ceci a stimulé de nombreuses équipes de recherche et de développement à mettre au point des outils de CAO qui permettent d'automatiser la plupart des étapes de la conception. Leur première tentative fut de produire des outils de génération automatique de structures régulières (PLA, ROM, etc...), puis de structures plus complexes : partie contrôle ou partie opérative. De cette course à l'automatisation est née la notion de "compilateur de silicum".

Un compilateur de silicium est un outil logiciel qui permet de transformer de façon quasi automatique une description de haut niveau d'un circuit intégré en les spécifications des masques de fabrication du circuit.

Le compilateur de silicium doit d'abord analyser la description fonctionnelle du circuit afin d'informer rapidement l'utilisateur sur la surface qu'occupera le circuit, ainsi que sur sa vitesse d'exécution. Dans le cas où une de ces informations ne satisfait pas l'utilisateur, ce dernier peut arrêter le processus de compilation et recommencer à zéro en modifiant la description fonctionnelle du circuit pour parvenir à un meilleur compromis surface-vitesse. Pour décharger l'utilisateur de cette tâche qui est assez lourde, certains compilateurs de silicium sont conçus de façon à trouver de façon automatique ou avec l'aide du concepteur un meilleur compromis surface-vitesse. Le compilateur de silicium idéal est celui qui produit à partir d'une description comportementale le "meilleur circuit possible". Actuellement il est impossible de produire un tel logiciel dans le cas général. Les compilateurs de silicium qui produisent automatiquement la meilleure implantation possible sont en général restreints à un domaine d'application spécifique, tel que le système FIRST spécialisé dans les applications de traitement du signal [BERG 84].

Le système SYCO, développé dans le groupe architecture des ordinateurs du laboratoire IMAG/TIM3 est un compilateur de silicium pour des circuits intégrés à très haute densité de type microprocesseur. SYCO génère les spécifications des masques d'un circuit constitué d'une partie contrôle et d'une partie opérative à partir de sa description comportementale. La description comportementale est un programme écrit dans un langage algorithmique (comme PASCAL) appelé LDS et

qui décrit de manière non ambiguë ce que doit faire le circuit.

L'objet de cette thèse, effectuée dans le cadre du projet de compilation de silicium SYCO, est l'étude des optimisations et des transformations possibles de la description comportementale d'un circuit pour la recherche d'un compromis surface-vitesse. Cette étude a abouti à la réalisation d'un ensemble de logiciels indépendants qui feront partie de l'environnement du système SYCO. Ces logiciels sont interactifs et devront permettre au concepteur de considérer un ensemble de transformations pour améliorer les performances du circuit final.

Dans le premier chapitre, une classification des compilateurs de silicium est établie. Ensuite, le compilateur de silicium SYCO est présenté. Le compilateur SYCO est décrit de manière générale, en partant de la description algorithmique jusqu'à la réalisation des masques.

Le chapitre 2 est consacré à l'évaluation de la surface d'un circuit généré par SYCO. Tout d'abord, les formules d'évaluation de la largeur de la partie opérative sont déterminées. Ensuite, les formules d'évaluation de la surface d'un étage de contrôle sont présentées. Le problème de l'évaluation de la vitesse d'un circuit est abordé à la fin de ce chapitre.

Le chapitre 3 est consacré à l'étude des techniques d'optimisation de haut niveau utilisées dans le compilateur SYCO. Le but de la phase d'optimisation est de réduire le nombre d'états et de transitions de l'algorithme d'interprétation des instructions du microprocesseur. Cette réduction entraîne une diminution de la surface du circuit (moins d'états mémorisés) et une augmentation de sa vitesse (moins de transitions). L'optimisation du nombre de registres et des opérateurs de la partie opérative est présentée à la fin de ce chapitre.

Le chapitre 4 est consacré au problème de la recherche d'un compromis surface-vitesse. Les transformations des descriptions comportementales de la partie opérative et de la partie contrôle sont présentées. Pour chaque transformation, nous présentons ses conséquences sur la surface et la vitesse du circuit à réaliser. Les formules qui déterminent le nombre d'états de l'algorithme d'interprétation des instructions après chaque transformation sont établies, ainsi que les conditions nécessaires pour qu'une transformation puisse conduire à un meilleur compromis surface-vitesse.

CHAPITRE I

LE COMPILATEUR SYCO

1. Introduction

Les hautes performances des ordinateurs actuels sont en grande partie dues aux progrès de la technologie des circuits intégrés. Aujourd'hui, ce sont les ordinateurs qui, à leur tour, permettent de créer des circuits plus performants. A l'origine de ce bel exemple d'interaction entre les calculateurs et les circuits intégrés : les compilateurs de silicium.

Le temps de conception d'un circuit peut être réduit par l'utilisation d'outils informatiques appropriés appelés "compilateurs de silicium", qui doivent :
- produire des circuits corrects par construction
- générer la description des masques de circuits de plus en plus complexes en un temps réduit (10 millions de transistors en un mois, avant 1990, d'après [WERN 82].

Wallich définit un compilateur de silicium comme étant un système (ensemble de programmes) qui permet de transformer une description de haut niveau d'un circuit (ou d'une partie d'un circuit) en un ensemble d'images géométriques (appelées "layout" ou dessin des masques du circuit) [WALL 83]. L'ensemble de ces images géométriques peuvent être directement utilisées pour générer les masques du circuit.

2. Classification des compilateurs de silicium

Si tous les compilateurs de silicium existant sur le marché ou dans les laboratoires de recherche génèrent des spécifications de même niveau (dessin des masques), ils ne partent pas du même niveau ni ne génèrent le même type de circuits.

On trouve actuellement dans la littérature ainsi que sur le marché, une grande variété de compilateurs de silicium. Ces derniers peuvent être classés selon le type des circuits générés ou selon le niveau du langage d'entrée du compilateur.

2.1. Type de circuit généré

Les circuits intégrés peuvent être classés en trois groupes selon le style de conception :
- les circuits prédiffusés ("Gate Array")
- les circuits précaractérisés ("Standard Cell")
- les circuits organisés à la demande ("Full Custom")

Ces trois styles de conception correspondent à trois catégories de compilateurs. Le problème est partiellement résolu pour les deux premiers groupes. Par contre, on commence juste à trouver des compilateurs qui génèrent des circuits du troisième groupe. De nombreux travaux sur la compilation de silicium de circuits organisés à la demande ont été publiés [GROS 83] [GAJS 86].

2.2. Description d'entrée

Les descriptions d'entrée de ces compilateurs peuvent être aussi classées en trois catégories :

2.2.1. La description comportementale

C'est le niveau de représentation le plus élevé. Elle représente l'algorithme d'interprétation des instructions du circuit [BLAC 85], [SISK 82], [JERR 86].
Dans ce cas, le processus de compilation de silicium se décompose en deux phases:
-génération d'une description structurelle correspondant à la description comportementale donnée,
-génération de la description géométrique (dessin des masques).

La description du comportement de l'algorithme à implanter est la première étape que le concepteur doit envisager. L'utilisation de langages de programmation classiques tels que Pascal ou PL/1 envisagée initialement pour la description du comportement de circuits intégrés s'est très rapidement avérée inadéquate. Ces langages sont par nature trop généraux pour les types de problèmes qui doivent être résolus ici et les étapes de compilation ou d'interprétation trop lentes.

Dans le cas présent, l'objectif à atteindre est l'implantation physique d'algorithmes respectant des contraintes temporelles (vitesse) et spatiales (faible surface, grande densité). Il a donc été important, pour des raisons de performance, de développer des langages - ou sous-langages de langages de programmation existants - prenant en compte ces contraintes, permettant ainsi de produire des descriptions claires et facilement simulables.

2.2.2. La description structurelle

C'est le niveau de représentation intermédiaire entre la description comportementale et la description géométrique. La description structurelle spécifie comment le système doit être construit. Avant d'utiliser le compilateur de silicium, le concepteur partitionne son circuit en blocs (PLAs, ROMs, RAMS, etc..) qui sont interconnectés (chaque bloc peut être décrit par un langage adapté au type du bloc). Il peut exister plusieurs descriptions structurelles pour un même comportement désiré. Par conséquent, la créativité et l'expérience du concepteur en architecture des ordinateurs joue un rôle important dans la partition du circuit. Dans ce cas, la tâche du compilateur de silicium se réduit à la génération du dessin des masques associés à chaque bloc [BRAY 85], [JOHA 79], [SHRO 82], [BERG 84]. La plupart des compilateurs qui partent de ce type de description permettent de générer des circuits organisés à la demande.

2.2.3. La description logique

Le circuit est décrit au niveau des portes (avec possibilité de hiérarchie). La plupart des compilateurs qui partent de ce type de description génèrent du prédiffusé ou du précaractérisé.

2.3. Modèle architectural et topologique

Il existe plusieurs façons de transformer une description comportementale en une description structurelle; de même, il existe plusieurs manières de transformer une description structurelle en une description géométrique. Pour réduire le nombre d'alternatives possibles pour passer d'une description à l'autre, la plupart des compilateurs de silicium développés actuellement sont basés sur un modèle. Ce modèle peut être :

- architectural (type de circuit - organisation fonctionnelle de la partie contrôle et de la partie opérative) [DEMA 86], [BLAC 85], [SISK 82], [JERR 86].

-topologique (organisation topologique du circuit) [MARS 86].

Ce choix restreint la portée du compilateur mais il permet en contrepartie d'utiliser des algorithmes simples et efficaces pour la génération des masques. Les

compilateurs de silicium sont en général conçus pour une certaine classe d'applications avec un modèle architectural spécifique, souvent appelé "architecture cible". L'architecture cible peut être très restreinte comme dans le cas des compilateurs de silicium pour le traitement du signal (elle est constituée essentiellement d'une partie opérative) ou peut être très générale. La plupart des compilateurs de silicium utilisent un des modèles architecturaux suivants [GAJS 86]:

1) Partie opérative
2) Partie contrôle (ou machine d'états finis)
3) Partie contrôle et une partie opérative
4) Partie contrôle, partie opérative et une mémoire.

Dans chaque modèle, on peut aussi identifier plusieurs styles d'implantation basés sur l'organisation de la partie opérative, de la partie contrôle, du protocole de communication, de l'horloge et du pipeline.

1) Partie opérative : Ce modèle est utilisé pour générer la partie opérative d'un circuit. La partie opérative exécute essentiellement des opérations ou des transferts de registres, elle n'exécute pas des fonctions de contrôle telles que la gestion des données, le choix d'une unité fonctionnelle ou la détermination de l'état suivant. Ce modèle est caractérisé par les types des unités de base utilisées (additionneurs, UALs, etc...), les types de mémorisation (registres, mémoires) et le style de connection (bus uni-directionnel, bus bidirectionnel, etc...). Apollon [JAMI 86] est un compilateur de parties opératives qui utilise une architecture dérivée de celle de la partie opérative du MC68000 : la partie opérative est constituée d'un ensemble de sous-parties opératives alignées à 2 bus.

2) Partie contrôle : Ce modèle est utilisé pour générer la partie contrôle d'une machine. La machine d'états finis implantée peut être un automate de type Moore où les sorties ne dépendent seulement que de l'état courant , ou de type Mealy où les sorties sont fonction de l'état courant et des entrées.
Si le nombre d'états est faible, la machine d'états finis peut être implantée à l'aide d'un simple compteur dont le contenu représente l'état courant, et d'une ROM qui génère les signaux de contrôle pour cet état. Dans le cas général, les composantes d'une machine d'états finis sont un registre d'état et la logique associée qui déterminent le nouvel état en fonction de l'état courant et des entrées. Cette logique peut être implantée à l'aide d'une structure régulière telle qu'un PLA ou une ROM, ou directement comme une logique anarchique.

3) Partie contrôle + partie opérative : Ce modèle possède une partie opérative qui exécute des opérations et des transferts. Les ordres d'activation sont générés par la partie contrôle. La partie opérative retourne des comptes rendus d'exécution. SYCO a une architecture basée sur une seule partie opérative (générée par Apollon) et une structure de contrôle hiérarchique constituée de "tranches" : l'utilisateur

décrit le comportement du circuit sous forme d'une hiérarchie de procédures, chaque niveau de cette hiérarchie est compilé dans une tranche de contrôle. La tranche de plus bas niveau contient la partie opérative.

4) Partie contrôle + partie opérative + mémoire : Ce modèle est une extension du modèle précédent où le type de mémorisation est étendu pour inclure des mémoires. le système CMU-DA (Carnegie-Mellon-University Design Automation) [WALK 83] possède un tel modèle architectural : la partie opérative est constituée d'UALs, d'éléments de mémorisation tels que les registres, mémoires, et le style de connection est distribué avec un nombre limité de bus.

Les compilateurs dont le point de départ est une description comportementale utilisent en général un modèle architectural. Les compilateurs qui partent d'une description structurelle peuvent eux aussi être basés sur un modèle mais sur un modèle topologique, par exemple la structure bit-slice [SHIRO 82].
Une nouvelle génération de compilateurs de silicium semble émerger [BREW 86], [KNAP 86]. Ces compilateurs sont constitués d'un grand nombre d'outils de type divers (synthèse, évaluation, simulation, optimisation, etc...) qui sont pilotés par un système expert. C'est ce système qui devra prendre des décisions pour la recherche d'un meilleur compromis surface-vitesse, en utilisant des connaissances exprimées sous forme de règles obtenues à partir d'experts en conception de circuits intégrés. Cette nouvelle approche a plusieurs avantages :
-un système expert à base de connaissances permet d'éviter une analyse combinatoire qui coûte généralement cher en temps CPU,
-l'organisation d'un compilateur de silicium sous forme d'un système expert, le rend facilement modifiable. On peut ainsi ajouter, modifier et tester de nouvelles connaissances. Actuellement, ce genre de compilateur de silicium est encore en cours de développement et connaîtra certainement un grand succès dans le futur.

3. Présentation du compilateur SYCO

3.1. Caractéristiques générales

Le système SYCO [JERR 86] développé dans le groupe architecture des ordinateurs du laboratoire IMAG/TIM3 est un compilateur de silicium, permettant de générer le dessin des masques d'un circuit intégré à très haute densité de type microprocesseur. Le système SYCO prend comme base de départ une description algorithmique comportementale et produit le dessin des masques du circuit qui réalise cet algorithme. La description comportementale est écrite dans un langage de description de circuits intégrés appelé LDS [LAUR 85]. Cette description est ensuite transformée en une hiérarchie de procédures telle qu'une procédure de niveau i ne peut appeler qu'une procédure du niveau directement inférieur i-1. Le principe du système SYCO est donc de convertir cette hiérarchie de procédures en une architecture de type microprocesseur telle que :
-la partie contrôle est constituée d'une hiérarchie de parties contrôle (ou étages de contrôle) où toutes les procédures d'un même niveau sont interprétées par un même étage de contrôle. Le programme principal est compilé dans la tranche de plus haut niveau.
-la partie opérative est générée à partir des actions opératives (additions, soustractions, transferts, etc...) spécifiées dans la description comportementale; elle constitue l'étage le plus bas du circuit.

La figure 1.1 montre un exemple de compilation d'un circuit à partir d'une description algorithmique :

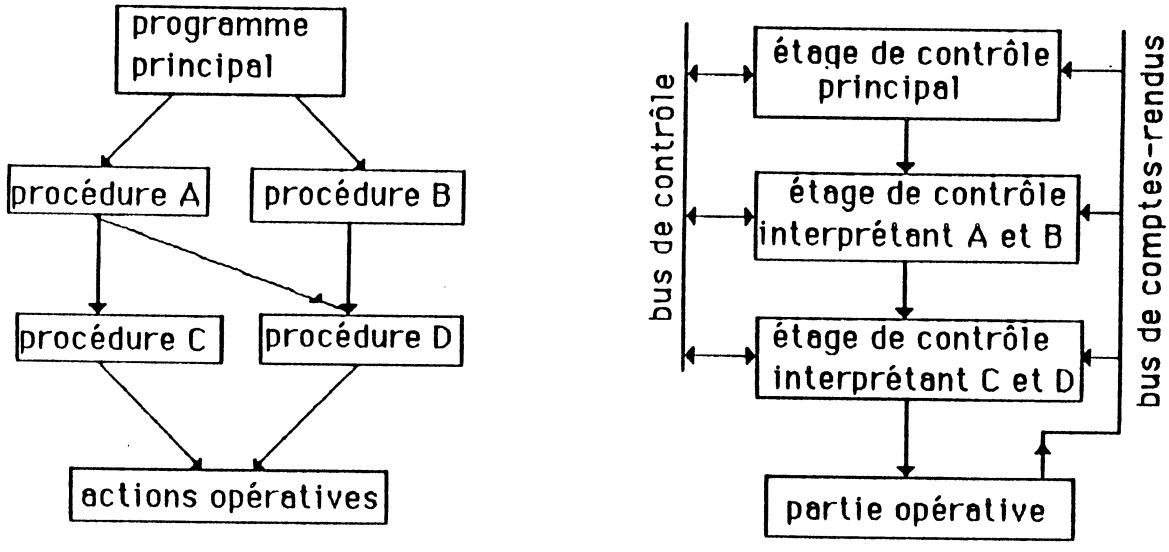


fig. 1.1 : compilation d'une hiérarchie de procédures

Dans ce qui suit nous présentons en détail le compilateur de silicium SYCO.

3.2. Architecture cible du compilateur SYCO

L'architecture cible du compilateur SYCO est représentée par la figure 1.2; elle consiste en une hiérarchie d'interpréteurs. L'interpréteur de niveau le plus haut interprète les instructions du microprocesseur et génère un langage de microinstructions pour le niveau directement inférieur. Ce dernier génère à son tour le langage des nanoinstructions pour son niveau inférieur, et ainsi de suite jusqu'au dernier niveau d'interprétation. Le dernier interpréteur est constitué des organes opératifs; il exécute directement les commandes qu'il reçoit à ses entrées : il constitue la partie opérative du circuit.

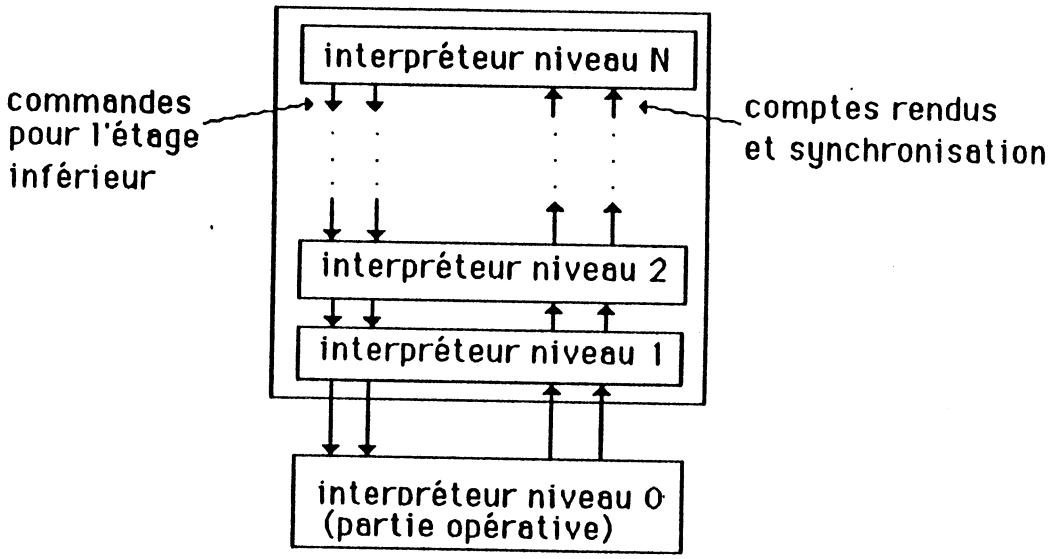


fig. 1.2 : architecture d'un circuit généré par SYCO

Le choix d'un modèle d'interprétation à plusieurs niveaux n'est pas un fait nouveau en conception de circuits VLSI. Plusieurs circuits ont mis en oeuvre ce principe. Des exemples en sont le 8085 de INTEL et le Z80 de ZILOG. Sagnes et al [SAGN 86] préconisent aussi l'utilisation de parties contrôle à plusieurs niveaux d'interprétation pour la conception de circuits VLSI complexes. Sagnes et al décrivent la réalisation d'un processeur 16 bits spécialisé (appelé POLLUX) en utilisant cette méthodologie. La partie contrôle de ce circuit est constituée de 3 étages de contrôle synchronisés par un générateur de temps.

3.2.1. Architecture de la partie contrôle

3.2.1.1. Architecture globale

La partie contrôle d'un circuit généré par SYCO est constituée d'une hiérarchie

d'étages de contrôle (fig. 1.3). Chaque étage de contrôle (sauf celui de niveau le plus haut) est constitué d'un ensemble d'automates d'états finis indépendants, tels que:

-à chaque procédure de niveau i dans la description comportementale, correspond un automate d'états finis de même niveau i,

-à un moment donné un seul automate est actif,

-un automate de niveau i ne peut être activé que par les automates de l'étage de contrôle supérieur i+1.
-les automates de niveau i reçoivent en entrée les sorties d'automates de niveau i+1;

-les sorties des automates de niveau i constituent les entrées des automates de niveau inférieur i-1.

Le premier et le dernier étage de contrôle jouent un rôle spécial. Le premier étage n'est constitué que d'un seul automate qui assure le séquencement global du circuit. Le dernier étage génère les commandes de la partie opérative.

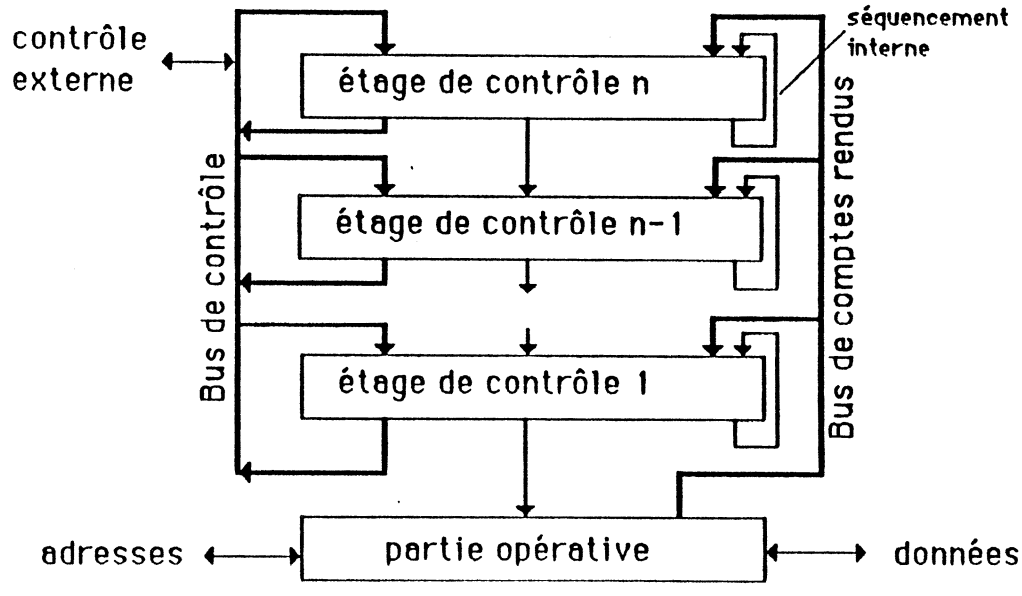


fig. 1.3 : architecture de la partie contrôle

3.2.1.2. Architecture d'un étage de contrôle

Le schéma d'un étage de contrôle est donné par la figure ci-dessous :

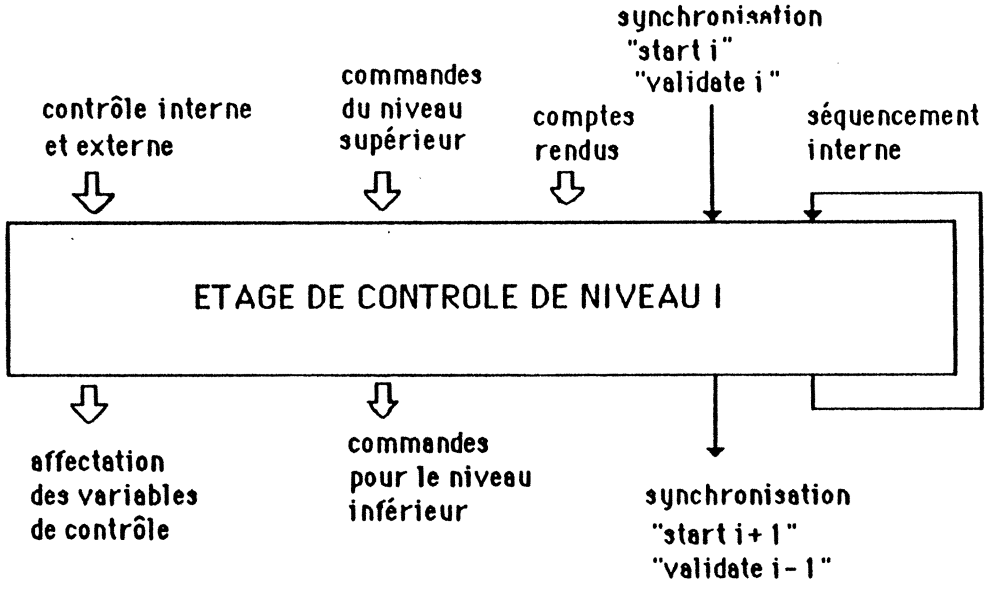


fig. 1.4 : entrées et sorties d'un étage de contrôle

Les entrées d'un étage de contrôle sont constituées :

- des commandes de niveau supérieur,
- des commandes de séquencement interne,
- des signaux de synchronisation des tranches de contrôle,
- des comptes rendus de la partie opérative,
- de requêtes de contrôle externes,
- et enfin de variables de contrôle internes.

Les sorties d'un étage de contrôle sont constituées:

- du nouvel état de la procédure en cours d'exécution (séquencement interne),
- des commandes pour l'étage de contrôle immédiatement inférieur,
- des signaux de synchronisation (signal "start i+1" pour rendre le contrôle à l'étage de contrôle supérieur et signal "validate i-1" pour activer l'étage inférieur),
- ainsi que des nouvelles valeurs et des commandes d'écriture pour toutes les variables de contrôle modifiées à ce niveau.

Les variables de contrôle ont la particularité qu'elles peuvent être directement affectées par une tranche de contrôle alors que les commandes de transfert dans la partie opérative doivent traverser toutes les tranches de contrôle. Les variables de contrôle internes sont utilisées pour mémoriser les paramètres des procédures.

En effet, SYCO permet l'emploi de procédures paramétrées, ce qui est d'ailleurs très pratique car cela permet d'éviter d'écrire plusieurs versions légèrement

différentes d'une même procédure. A chaque paramètre, on associe une variable de contrôle. Ces variables sont réalisées à l'aide de bascules ou de registres selon que les paramètres sont booléens ou non, et sont placées dans la partie contrôle. Elles peuvent éventuellement être intégrées dans un chemin de données de type bit-slice à l'intérieur de la partie contrôle au dessus des étages de contrôle.
Les registres contenant les valeurs des paramètres des procédures sont placés dans la partie contrôle pour 2 raisons :

- les paramètres des procédures ont souvent des valeurs booléennes ou des valeurs qui ne nécessitent qu'un faible nombre de bits. Le placement des registres de paramètres dans la partie opérative qui traite des mots de longueur très supérieure n'est pas souhaitable car cela conduirait à l'apparition de trous dans la structure bit-slice et donc à une mauvaise utilisation de la surface de silicium.

- d'autre part le placement de ces registres dans la partie contrôle permet d'affecter directement les paramètres. En effet, dans une partie contrôle multiniveau comme celle utilisée dans SYCO, une commande de transfert dans la partie opérative doit traverser tous les étages de la hiérarchie avant que la commande suivante de même niveau ne puisse être exécutée. Par contre un étage de contrôle peut agir directement sur les variables de contrôle.

Architecture actuelle d'un étage de contrôle :

Il existe de nombreuses architectures possibles pour implanter un étage de contrôle. La solution qui a été programmée dans SYCO est l'architecture mono-PLA. D'autres architectures sont également proposées dans [JERR 86] et [VARI 86] : partie contrôle microprogrammée et partie contrôle à générateur de temps.

La solution retenue actuellement a le mérite d'être simple et de pouvoir être implantée très facilement. La matrice ET décode l'instruction, les comptes rendus de la partie opérative, les requêtes de contrôle externes (reset, interruptions), les paramètres des procédures et l'état de l'automate. La matrice OU génère les sorties et en particulier les commandes de la partie opérative ou les commandes de la partie contrôle de niveau inférieur et le nouvel état de l'automate simultanément. L'architecture mono-PLA permet donc de réaliser naturellement des automates de Mealy.

3.2.2. Architecture de la partie opérative

L'architecture de la partie opérative est basée sur le principe du "bit slice", c'est à dire que l'on construit une tranche de partie opérative de un bit, que l'on assemble ensuite les unes au-dessus des autres pour former la partie opérative de n bits. Par conséquent, le dessin des masques d'une partie opérative est aisé à construire, par simple juxtaposition de cellules prises dans une bibliothèque.

Le modèle global de la partie opérative dans SYCO est dérivé de l'architecture de la partie opérative du MC68000. Le choix d'un modèle à 2 bus pour la compilation des parties opératives est préconisé dans [ANCE 83] et [ANCE 84]. Mead et Conway [MEAD 80] préconisent aussi l'utilisation de parties opératives à deux bus.

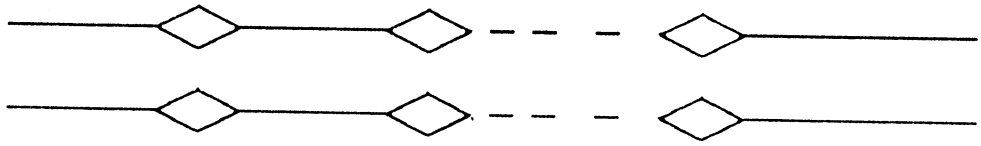


fig. 1.5 : modèle architectural de la partie opérative

La partie opérative est donc basée sur une architecture à deux bus. Ces deux bus peuvent être morcelés, ce qui permet de définir des sous-parties opératives (fig.1.5). La partie opérative est constituée d'un ensemble de sous parties opératives à deux bus placées côte à côte et pouvant fonctionner en parallèle. Chaque sous-partie opérative peut fonctionner de manière indépendante (exemple: une sous partie opérative pour les adresses et une autre pour les données) ou être connectée à une de ses voisines par l'intermédiaire d'un ou des deux bus.
Chaque sous-partie opérative est constituée d'un ensemble d'éléments de mémorisation:

- registres à simple ou double accès
- constantes à simple accès
- tampons des plots d'entrée-sortie
- d'un ensemble d'opérateurs
- et des tampons de sortie de ces opérateurs connectés à un ou aux deux bus.

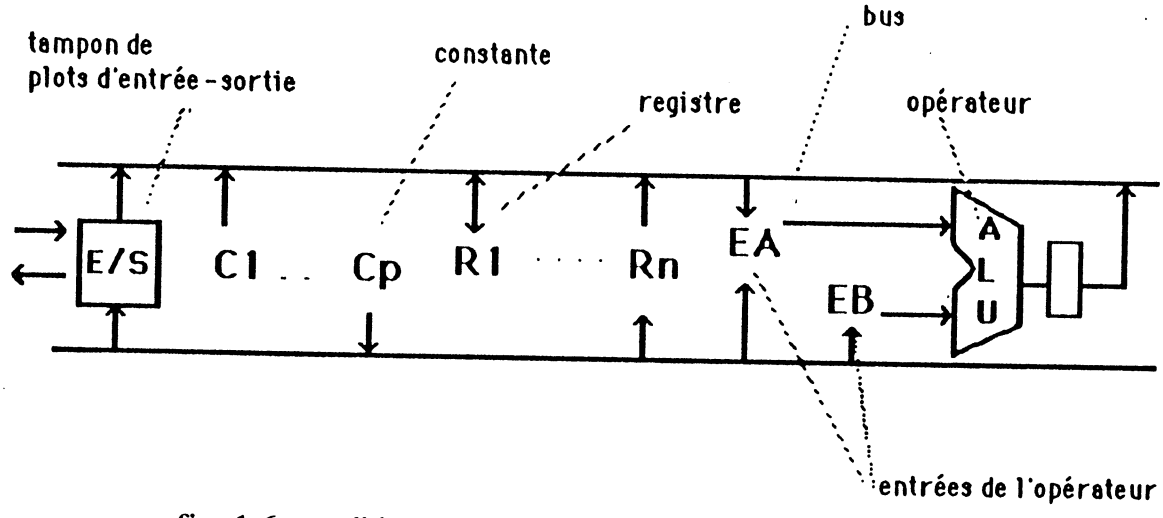


fig. 1.6 modèle architectural d'une sous-partie opérative

Chaque sous-partie opérative peut exécuter en un seul cycle machine (constitué de deux phases d'horloge notées "phi1" et "phi2") un des sous-ensembles d'actions suivants :

- une opération binaire et un transfert de registre : en phi1 transfert des opérandes dans les tampons d'entrées de l'UAL; en phi2 récupération de la sortie de l'UAL (résultat de l'opération) et exécution du transfert.

- une opération unaire et 2 transferts de registre : en phi1 transfert de l'opérande vers une entrée de l'UAL et exécution d'un transfert de registre; en phi2 récupération de la sortie de l'UAL et exécution du deuxième transfert.

- 4 transferts de registre de sources distinctes : à chaque phase 2 transferts sont exécutés, un sur chaque bus de la sous-partie opérative.

Il est possible d'exécuter en parallèle plusieurs transferts de source commune; par exemple, l'instruction suivante s'exécute en un seul cycle :

PC := AD := (RG := BR) + F

Pendant la phase phi1, le registre BR est transféré en même temps vers le registre RG et vers l'entrée de l'UAL où doit s'effectuer l'opération "BR + F". Pendant la phase phi2, la sortie de l'UAL est transférée vers les registres AD et PC.

3.3. Modèle topologique

Le modèle topologique est tel que les différentes tranches composant le circuit, tranches de partie contrôle, ou tranche de partie opérative, sont topologiquement assemblées l'une au-dessus de l'autre. Ce modèle est directement déduit du modèle architectural (hiérarchie d'interpréteurs).

Un bus situé à gauche des étages va propager les différents signaux de communication :
- les fils de contrôle (requêtes externes, variables de contrôle interne),
- les fils de compte-rendus de la partie opérative,
- les commandes pour l'étage de contrôle inférieur et le séquencement interne,
- les horloges et les fils de synchronisation (activation des étages).

La masse est située à gauche de la partie contrôle, alors que l'alimentation Vcc est à droite [GERO 87]. Autour du coeur du circuit, sont assemblés les plots de contrôle, les plots d'horloge, les plots de données et d'adresse. Les plots occupent ainsi le pourtour du circuit. Nous obtenons le schéma topologique suivant :

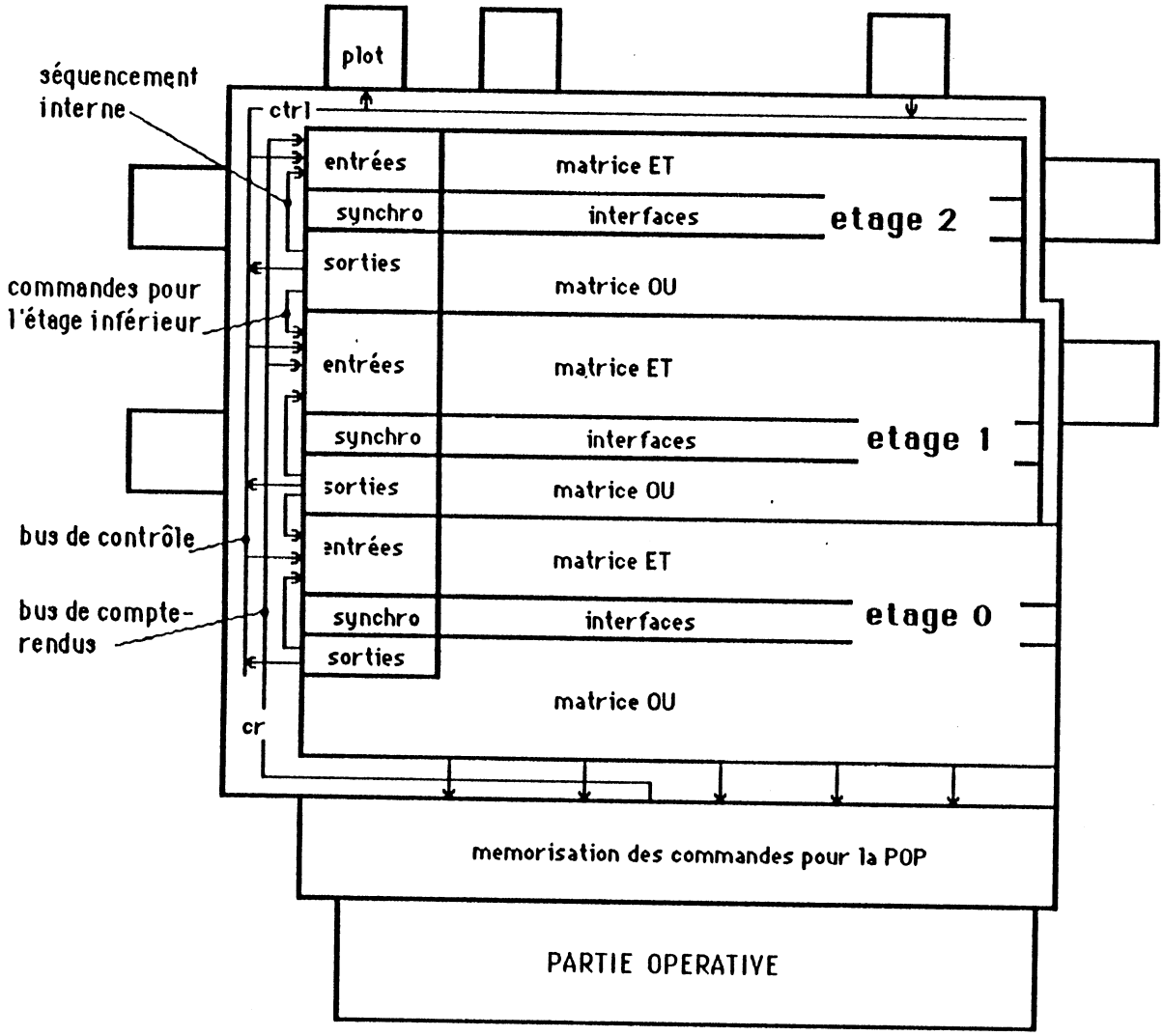


fig.1.7 : topologie globale du circuit

3.4. Modèle temporel

Le modèle temporel est de type synchrone, c'est à dire que l'horloge est la même pour tout le circuit. L'horloge de base du circuit est décomposée en 4 phases désignées respectivement par T1, T2, T3 et T4. Ces phases correspondent aux phases d'exécution d'un transfert dans la partie opérative :
T1 : précharge des bus
T2 : lecture des registres sources
T3 : amplification
T4 : écriture dans les registres destinations

Le temps d'exécution d'une tranche de contrôle est d'un cycle. A chaque cycle, une seule tranche est active. Lorsqu'une tranche de contrôle lance l'exécution d'une procédure, elle doit attendre que l'exécution de cette procédure par la tranche

inférieure soit terminée avant de lancer l'exécution d'une autre procédure ou de repasser le contrôle à la tranche supérieure.

Pour mettre en œuvre ce mécanisme de synchronisation dans le cas d'un circuit à n étages, il suffit de disposer de n-1 signaux d'activation de l'étage supérieur ("start"), de n-1 signaux d'activation de l'étage inférieur ("validate") et d'un reset.

Exemple : cas d'une machine à 3 étages

Décrivons le fonctionnement d'une machine à 3 étages de contrôle (fig. 1.8). Nous supposons que lors d'une phase d'initialisation (Reset), les entrées et les sorties des différentes tranches ont été remises à zéro sauf "start1" qui est remis à 1. La notation suivante a été choisie :
"validatei" (resp. "starti") = signal validate (resp. start) pour l'étage de contrôle i.

cycle 1 : la tranche de plus haut niveau est activée. En T4 le nouvel état et les sorties de l'étage 3 sont disponibles. "validate2" est mis à 1 pendant ce cycle.

cycle 2 : pendant la phase T1 de ce cycle, l'entrée de l'étage 2 prend la valeur de la sortie de l'étage 3. En T4 le nouvel état et les sorties sont disponibles. "validate2" est mis à 0 et "validate1" est mis à 1.

cycle 3 : c'est l'étage 1 qui travaille. A la fin de ce cycle, les commandes de la partie opérative sont disponibles en sortie de l'étage 1.

cycle 4 : ce n'est qu'au cycle 4 que la partie opérative commence à travailler.

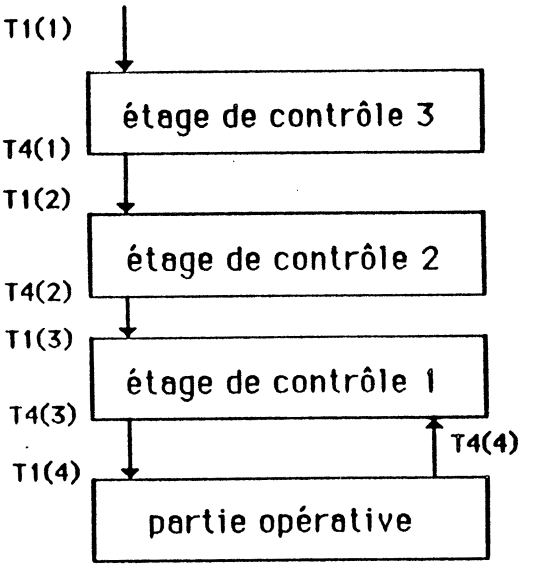


fig. 1.8 : modèle temporel pour un circuit à 3 étages de contrôle
(notation : Ti(j) = phase Ti du cycle j)

Nous n'avons décrit que la transmission des ordres de haut en bas. Décrivons le passage de contrôle de l'étage 1 à l'étage 2 après l'exécution d'une procédure de 2 instructions par l'étage 1 :

cycle 4 (suite) : en T1 du quatrième cycle, le nouvel état interne de l'étage 1 est transféré en entrée de cet étage. La deuxième instruction de la procédure est exécutée. En T4 les résultats sont disponibles en sortie de l'étage 1 et "start2" est mis à 1. Les tampons d'entrée de l'étage 1 sont alors remis à 0.

cycle 5 : La deuxième instruction de la procédure de l'étage 1 est exécutée par la partie opérative et en même temps la deuxième instruction de la procédure de l'étage 2 est activée.

3.5. Langage d'entrée du compilateur SYCO

SYCO utilise actuellement le langage de description de circuit appelé LDS. La première version de SYCO était basée sur le langage IRENE [MARI 85]. Le langage LDS permet de spécifier l'algorithme d'interprétation des instructions du microprocesseur. La description est de niveau transfert de registres, c'est à dire qu'elle fait intervenir des composants de base de type registres ou mémoires et non pas des portes ou des bascules.

Le langage LDS doit ainsi permettre de décrire les spécifications fonctionnelles du circuit intégré à réaliser, et d'en simuler le comportement par l'utilisation d'un simulateur fonctionnel afin de vérifier la cohérence de la description du concepteur.

3.5.1. Description d'un circuit en LDS

Un circuit est considéré comme un ensemble de blocs (ou modules) interconnectés entre-eux. Ces blocs possèdent une description structurelle et comportementale. Afin de distinguer la description structurelle de la description comportementale, LDS définit deux types de modules : les modules structurels (SMODULE) et les modules comportementaux (CMODULE). A chaque module structurel est associé un module comportemental qui permet de définir son comportement.

Le compilateur SYCO utilise un sous-ensemble de LDS :

- un seul module structurel est déclaré,
- à ce module est associé un module comportemental qui peut appeler d'autres modules comportementaux.

Tous les CMODULEs de même niveau seront implantés dans une même tranche de contrôle.

3.5.1.1. Description d'un SMODULE

Un SMODULE est caractérisé essentiellement par ses bornes (ou variables d'entrée-sortie) et les variables qu'il utilise.

Parmi les variables utilisées, on distingue 2 types de variables :

1) les variables de type données : ce sont les registres de la partie opérative. Ces registres sont de type point mémoire avec mémorisation sur un état et non sur un front.

2) les variables de type contrôle : Ces variables sont situées dans la partie contrôle et servent à mémoriser les requêtes de contrôle externes et les paramètres des procédures.

Cette distinction entre variables de contrôle et variables de données a été introduite pour les besoins du compilateur de silicium. En effet, il est peu intéressant d'intégrer les variables de contrôle booléennes ou qui portent sur un faible nombre de bits dans une partie opérative dont le nombre de bits est bien supérieur.

3.5.1.2. Description d'un CMODULE

Un CMODULE est constitué d'une suite de "séquences". Une séquence est une suite d'instructions étiquetée par le nom de la séquence. Les instructions dans une séquence sont exécutées séquentiellement sauf si une action de branchement est exécutée.
L'appel d'un CMODULE (par l'instruction EXECUTE) provoque le déroulement des séquences constituant ce CMODULE. A la fin de l'exécution du CMODULE appelé, l'exécution de la séquence appelante reprend. Le nombre de séquences d'un CMODULE est égal au nombre de points de branchement différents augmenté de 1; en effet, le point d'entrée d'un CMODULE doit être étiqueté.

Une instruction (appelée "PINST" en LDS) est constituée d'un ensemble d'actions qui s'exécutent en parallèle. Les actions d'une PINST peuvent être conditionnelles ou inconditionnelles. Parmi les actions inconditionnelles, on distinguera :

-les actions opératives : transferts de registres, opérations,
-l'appel à un CMODULE de l'étage inférieur : instruction EXECUTE,
-les actions de contrôle : les instructions SET et RESET permettent d'affecter des valeurs aux variables de contrôle,
-le branchement à l'instruction suivante : instruction NEXT,
-et le retour de contrôle à l'étage supérieur : instruction EXIT.

Exemple de description d'un Cmodule :

```
CMODULE  MICROP;

<rest>      IF (restart = 0)  NEXT rest;  ELSE  NEXT start; END;

<start>     (pc :=0;  NEXT newinstr;)

<newinstr>  (r := 0;  EXECUTE fetch;)
            IF (ir 4:2 = '00') NEXT decode; ELSE (r:=3; EXECUTE fetch;) END;
            (CASE (ir 4:2)

            WHEN ('01')  EXECUTE fetch;  ad :=ir;
            WHEN ('10')  EXECUTE fetch;  ad :=pc+ir;
            WHEN ('11')  EXECUTE fetch;  ad :=pc-ir;
            END;
            NEXT decode;)

<decode>    (CASE (ir 0:4)
             WHEN ('00 00')  EXECUTE ada;
             WHEN ('00 01')  EXECUTE lda;
               ...
            END;
            IF (restart = 0)  NEXT rest;  ELSE  NEXT newinstr;  END;)

END;
```

"rest", "start", "newinstr" sont des noms de séquences ("rest" est le point d'entrée du CMODULE "MICROP").

"fetch", "ada" et "lda" sont des noms d'autres CMODULEs interprétés par l'étage de contrôle situé au-dessous de celui interprétant le CMODULE "MICROP".

3.5.2. Influence de la description comportementale sur la forme du circuit

La description comportementale a une forte influence sur les caractéristiques de la partie opérative et de la partie contrôle du circuit. Deux éléments de base peuvent les faire varier : le parallélisme entre les actions opératives et le séquencement des instructions.

En effet, augmenter le nombre d'actions en parallèle correspond en général à augmenter la largeur de la partie opérative. Par exemple, l'action :
"A := A + B"
nécessite une UAL et deux registres A et B, alors que les deux actions suivantes

exécutées en parallèle :
"A := A + B ; C := C - D"
utilisent 2 UALs.

Mais en diminuant le nombre d'actions en parallèle on augmente le nombre d'étapes nécessaires à l'exécution de l'instruction. On est alors amené soit à faire croître le nombre d'états de l'algorithme et diminuer le parallélisme, soit à faire croître le parallélisme et diminuer le nombre d'états de l'algorithme. Ces choix peuvent être exprimés facilement par le concepteur en langage LDS. Ceci est réalisé par :

1) Le choix du nombre des niveaux d'interprétation
2) La complexité de chaque interpréteur, c'est à dire le nombre des états du séquencement interne.
3) Le choix de la complexité des opérateurs
4) Le nombre d'actions parallèles que doit réaliser la machine pendant un cycle.

Un exemple va nous permettre d'éclairer ce point; supposons que l'on veuille réaliser un circuit qui exécute l'instruction suivante :

A := A + B ; C := C - D ; E := E and F

Il existe plusieurs façons de décrire ce circuit dans le langage LDS :

Description 1 :

```
CMODULE prog;
<e1> : A := A + B ; C := C - D ; E := E and F ; EXIT;
END;
```

Dans ce cas, on a un niveau d'interprétation et une partie opérative de 3 UALs. Il s'agit de la description qui donne le circuit le plus rapide (un seul état).

Description 2 :

```
CMODULE prog;
<e1> : A := A + B ; NEXT e2;
<e2> : C := C - D ; E := E and F ; EXIT;
END;
```

Dans ce cas, la partie opérative ne contient que 2 UALs. Le circuit obtenu sera moins rapide, mais plus cohérent du point de vue surface que le premier.

Le langage LDS permet donc de contrôler le résultat obtenu par le compilateur de silicium et éventuellement de pouvoir modifier dans le bon sens sa description si le

résultat de l'évaluation est jugé insatisfaisant par le concepteur. Bien que le modèle d'architecture cible du compilateur soit celui d'une hiérarchie d'interpréteurs, dont le nombre est déterminé par celui des niveaux d'interprétations dans la description, le concepteur doit être conscient que ce concept doit être manié avec circonspection. Il est conseillé au concepteur de n'introduire un nouvel interpréteur que si cela est vraiment nécessaire. En effet, un nombre excessif de niveaux génère une machine plus lente. Le chapitre 4 est consacré à l'étude des transformations possibles de la description comportementale pour résoudre le problème du compromis surface-vitesse.

3.6. Les étapes de la compilation

Le compilateur SYCO est basé sur une architecture cible, ce qui facilite grandement l'algorithme de traduction. Le modèle adopté fournit à la fois un modèle architectural et un modèle topologique. Les 2 phases de la compilation : traduction d'une description comportementale en une description structurelle et traduction d'une description structurelle en une description géométrique sont donc toutes les 2 simplifiées. La simplicité de l'algorithme est cependant essentiellement basée d'une part sur la correspondance entre la hiérarchie des appels de procédures et la hiérarchie des étages de contrôle et d'autre part sur la correspondance entre le nombre maximum d'opérations en parallèle et le nombre de sous parties opératives. Cette correspondance permet bien sûr au concepteur de contrôler le fonctionnement du compilateur puisque le mécanisme de traduction est évident. Toutefois, une phase d'optimisation peut être déclenchée afin d'aboutir automatiquement à un meilleur compromis surface-vitesse.

Les différentes étapes de la compilation sont les suivantes :

1) validation de l'algorithme : cette phase permet de s'assurer de la validité des spécifications formelles par une simulation fonctionnelle.
2) transformation de la description.
3) extraction des descriptions comportementales de la partie contrôle et de la partie opérative.
4) optimisation.
5) évaluation de la surface du circuit.
6) selon le résultat de l'évaluation, une phase de recherche d'un compromis surface vitesse peut être déclenchée.
7) compilation de la partie opérative.
8) compilation de la partie contrôle.
9) assemblage du circuit.

Un organigramme du système SYCO est présenté ci-dessous.

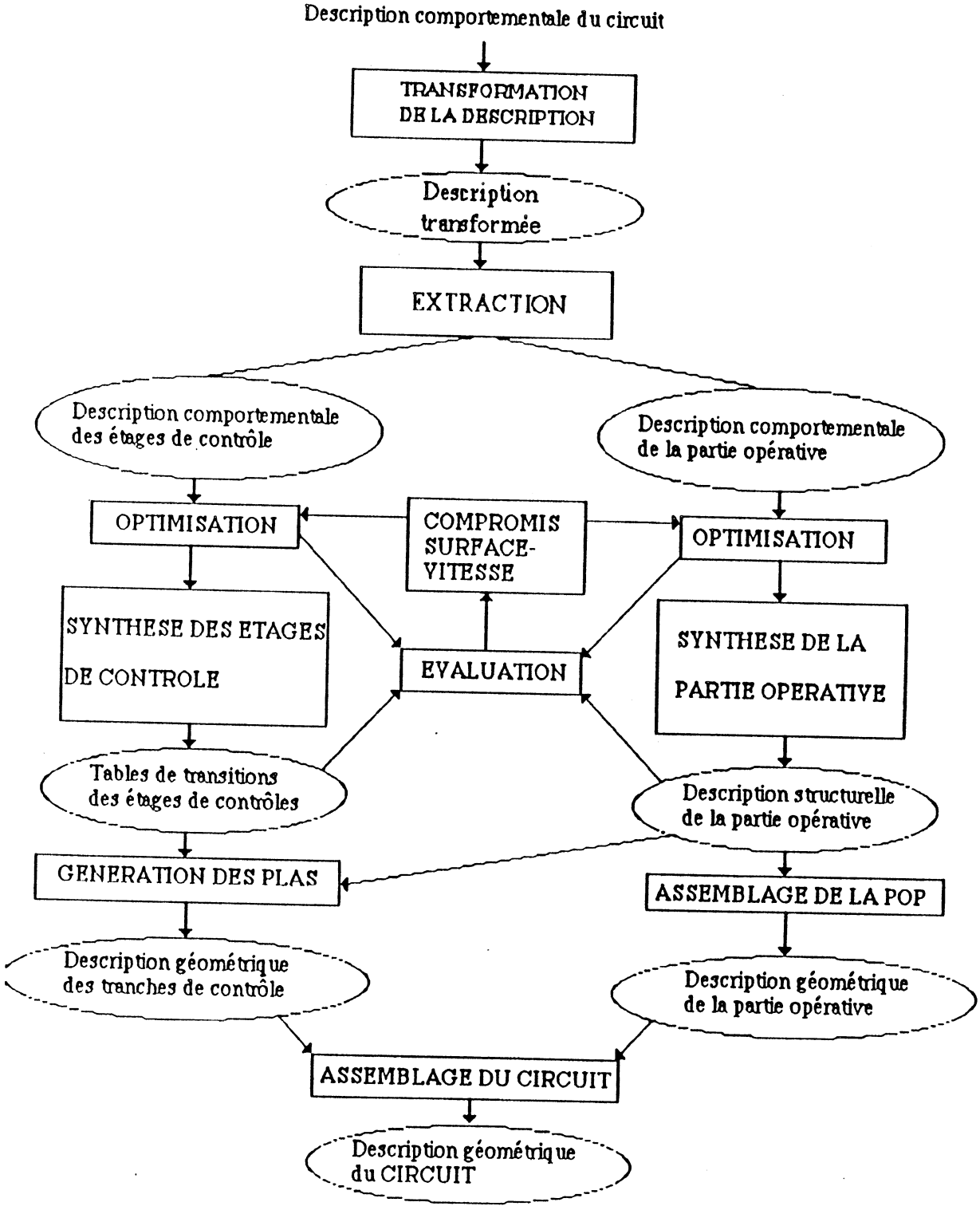


fig. 1.9 : organigramme du système SYCO

3.6.1. Transformation de la description comportementale

Avant d'extraire les spécifications de la partie contrôle, il faut adapter la description comportementale au modèle architectural du compilateur SYCO. En effet, nous rappelons que ce modèle architectural est tel qu'un étage de contrôle de niveau i ne peut activer directement que l'étage de contrôle inférieur i-1. Supposons par exemple, que dans la description comportementale originale introduite par le concepteur, il existe un CMODULE de niveau i qui appelle un CMODULE de niveau i-2 (fig. 1.10). Pour résoudre ce problème, on rajoute un CMODULE intermédiaire, dans le niveau d'interprétation i-1; ce CMODULE contient une seule instruction "EXECUTE"; on crée ainsi une chaîne d'appels de CMODULEs afin de pouvoir activer l'étage i-2 à partir de l'étage i.

exemple : Soit l'arbre d'appels des CMODULEs suivant :

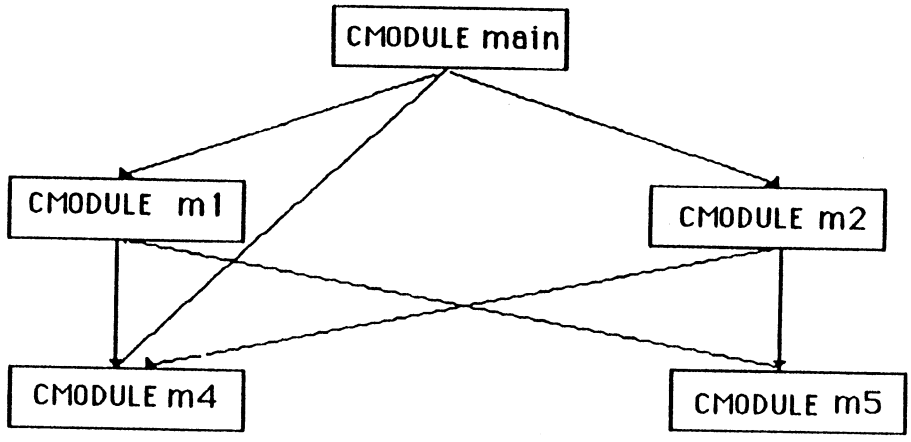


fig. 1.10 : arbre d'appel des Cmodules initial

Le CMODULE "main" de niveau 3 appelle le CMODULE "m4" qui est de niveau 1. Comme il n'existe pas actuellement de transparence d'un étage de contrôle, on rajoute un CMODULE intermédiaire (noté "int") qui est constitué d'une seule instruction "EXECUTE m4" (fig.1.11). On obtient le nouvel arbre d'appel des CMODULEs suivant :

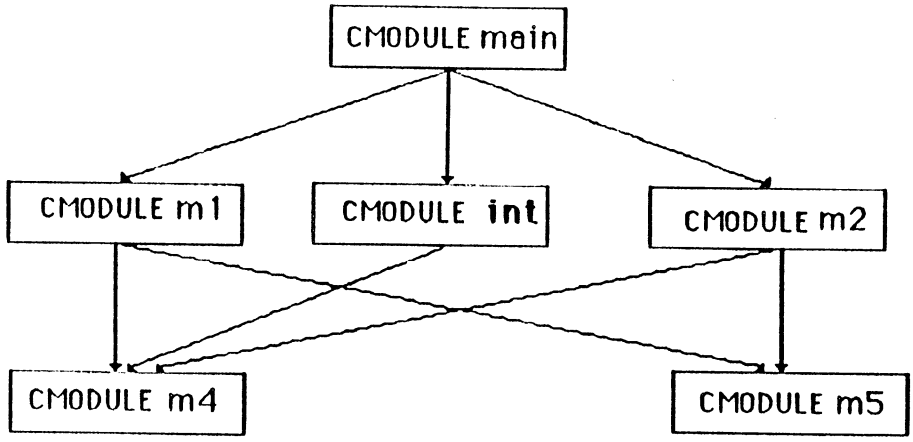


fig.1.11 : arbre d'appel de Cmodules final

Après cette phase, tous les CMODULES de même niveau peuvent être implantés dans une même tranche de contrôle.

3.6.2. Extraction des descriptions comportementales des tranches de contrôle et de la partie opérative

Le but de l'extraction est de séparer les instructions de contrôle des instructions opératives [MART 85], [BOUR 86]. La procédure la plus simple consiste à effectuer un traitement récursif. On extrait d'abord la description de la partie opérative, puis la description de l'étage de contrôle de plus bas niveau, puis celle de niveau immédiatement supérieur et on continue ainsi jusqu'à ce que l'on ait obtenu la description de tous les étages de contrôle.

3.6.2.1. Génération de la description comportementale de la partie opérative

La description comportementale de la partie opérative est constituée d'un ensemble non ordonné d'instructions opératives. Une instruction opérative (ou microinstruction) est constituée d'un ensemble d'actions opératives se déroulant en parallèle. On distingue 3 types d'actions opératives :

-transfert de registre à registre (ex : A := B)
-opérations binaires (ex : A := B + C)
-opérations unaires (ex : A := Not B)

Ces instructions sont stockées dans un fichier qui sera utilisé par le compilateur de parties opératives Apollon [JAMI 85], [JAMI 86] pour générer les masques de la partie opérative.

L'extracteur associe un numéro à chaque instruction opérative. A une instruction

conditionnelle à n alternatives correspondent n instructions opératives. D'une part, l'extracteur génère un tableau des instructions opératives avec leurs numéros et d'autre part, il remplace dans la description originale les instructions opératives par leurs numéros dans le but de générer la description des tranches de contrôle.

3.6.2.2. Génération de la description comportementale des différentes tranches de contrôle

les spécifications de la partie contrôle sont fournies par la structure de contrôle de l'algorithme donné par :

-le séquencement de l'algorithme, qui spécifie en particulier l'enchaînement (conditionnel ou inconditionnel) des instructions,

-les conditions qui déterminent à chaque étape de l'algorithme, les actions qui doivent être effectuées.

Génération des tables de transitions associées à chaque tranche de contrôle :

Il y a un grand nombre de façons de représenter les états, les transitions d'états, et les sorties de machines séquentielles correspondant à des séquences d'entrées données. Les diagrammes d'états, les tables de transitions, les équations booléennes, etc... sont des représentations possibles.

Dans le compilateur SYCO le comportement de chaque tranche de contrôle est représenté par une table de transitions. Chaque ligne de cette table est constituée de 5 champs (fig. 1.12):

1) "entrée" : ce champ contient le nom d'un CMODULE interprété par l'étage de contrôle.

2) "état" : contient le nom de l'instruction en cours d'exécution.

3) "condition" : contient la condition associée à la transition de l'état courant vers l'état suivant; ce champ est vide si l'instruction est inconditionnelle.

4) "action" : contient l'action générée au moment de la transition d'un état vers un autre. Une action ne peut contenir qu'un appel de CMODULE; en effet, deux CMODULEs ne peuvent être exécutés simultanément. Par contre, elle peut comporter plusieurs affectations de variables de contrôle, ou plusieurs actions opératives (opérations, transferts de registres) si la table de transitions décrit le comportement du dernier étage de contrôle.

5) "suivant" : contient le nom de l'état suivant ou le mot clé EXIT qui indique le

retour de contrôle à l'étage supérieur (étage appelant).
La figure 1.12 donne un exemple de table de transitions associée à un étage de contrôle. L'étage de contrôle représenté par cette table de transitions interprète deux CMODULEs appelés "DIV" et "MUL"; dans le CMODULE DIV, il existe une instruction conditionnelle étiquettée par "e1" et dont la syntaxe est :

<e1> : IF A=0 THEN EXECUTE XX; NEXT e4 ;

ENTREE	ETAT	CONDITION	ACTION	SUIVANT
DIV	e1 e4	A = 0 . . .	execute XX . . .	e4 . . .
MUL				

fig. 1.12 : exemple de table de transitions utilisée dans SYCO

Avant de générer la table de transitions associée à un étage de contrôle, il faut bien sûr, que dans chaque CMODULE interprété par cet étage, toutes les instructions comportant des actions conditionnelles soient mises sous forme canonique. Pour cela, toute instruction conditionnelle, c'est à dire comportant une ou plusieurs actions conditionnelles if ou case (qui sont alors en parallèle), ces actions conditionnelles pouvant elles mêmes comporter des actions conditionnelles imbriquées, est transformée en une suite de couples (condition, instruction inconditionnelle) où les différentes conditions sont exclusives et où les instructions sont constituées d'une suite d'actions inconditionnelles à exécuter en parallèle. Toute instruction conditionnelle est donc transformée en une instruction de type COND (de l'instruction lisp COND) où toutes les alternatives sont exclusives :

(COND
(cond 1 action 11 action 12... action $1p_1$)
(cond 2 action 21 ..... action $2p_2$)
(cond n action n1..... action np_n))

Cette décomposition peut être faite en 2 temps :

1°) on met d'abord tous les if et case à exécuter en parallèle sous la forme canonique. Cette transformation ne consiste qu'en une simple réécriture s'il n'y a pas d'imbrication.

2°) on génère ensuite toutes les alternatives correspondant aux différentes actions conditionnelles en parallèle. Il y en a :

$n_1 * n_2 * \ldots * n_p$

avec : n_i le nombre d'alternatives du ième IF ou CASE selon le cas,
p le nombre de if ou de case en parallèle.

exemple : l'instruction conditionnelle suivante :

IF c1 THEN action1 ELSE IF c2 action2 ELSE action3

est décomposée en :

IF c1 THEN action1
IF (non c1 et c2) THEN action2
IF (non c1 et non c2) THEN action3

3.6.3. Optimisation de la description comportementale de la partie contrôle et de la partie opérative

L'étape d'optimisation consiste à appliquer des transformations algorithmiques à la description comportementale des étages de contrôle et de la partie opérative afin de réduire la surface du circuit sans diminuer ses performances ou d'augmenter ses performances mais sans provoquer une augmentation de sa surface. Certaines de ces transformations sont analogues à celles utilisées par les compilateurs optimiseurs de langages de programmation évolués [AHO 77] telles que l'augmentation du parallélisme, l'expansion de procédures appelées une seule fois, l'élimination des expressions redondantes, etc...; d'autres d'optimisations sont propres aux contexte du compilateur SYCO. Cette étape de la compilation est étudiée en détail dans le chapitre 3.

3.6.4. Evaluation de la surface du circuit

La tâche de l'évaluateur consiste à déterminer si la description du concepteur conduira à un bon compromis surface-vitesse. L'évaluation doit être effectuée avant que les spécifications des masques ne soient générées, la génération du dessin des masques étant une des étapes les plus coûteuses en temps de calcul. L'évaluation peut porter sur 2 types de descriptions :

- la description comportementale des tranches de contrôle et de la partie opérative après la phase d'extraction; il s'agit alors d'une évaluation grossière qui permet de déterminer le nombre d'étages de contrôle et une estimation du nombre de sous-parties opératives,

- la description structurelle des PLAs de contrôle et de la partie opérative après la phase de synthèse. Il s'agit d'une évaluation plus fine qui permet de déterminer le nombre et la taille des différents PLAs de contrôle ainsi que le nombre de sous-parties opératives et la taille de la partie opérative.

L'évaluation de la surface d'un circuit généré par le compilateur SYCO est étudiée dans le chapitre 2.

3.6.5. Recherche d'un compromis surface-vitesse

La recherche d'un compromis surface-vitesse consiste à échanger une perte de performance contre un gain de surface ou vice versa. Supposons qu'on obtienne comme résultat de l'évaluation de surface, un circuit plus haut que large. En augmentant sa largeur, donc ses performances, on est à peu près certain de diminuer sa hauteur. Si, par contre, le circuit est trop large, il faut sacrifier des performances, en gagnant une meilleure forme. On devra donc choisir entre performance et surface. Des transformations algorithmiques de la description du circuit, telles que l'augmentation/réduction de parallélisme, insertion/expansion de procédures sont utilisées à ce niveau de la compilation. Le chapitre 4 est consacré à l'étude de ces transformations.

3.6.6. Compilation de la partie opérative

Le compilateur Apollon [JAMI 86] génère les spécifications des masques de la partie opérative et toutes les informations nécessaires à la génération de la partie contrôle à partir des instructions opératives extraites de la description comportementale. A chaque variable ainsi qu'à chaque constante le système Apollon alloue un registre dans la partie opérative. A chaque opération est alloué un opérateur (UAL, incrémenteur, décaleur); plusieurs opérations peuvent être affectées à un même opérateur si elles n'appartiennent pas à une même instruction opérative.

L'algorithme d'Apollon est basé sur une construction progressive de la partie opérative. Le compilateur crée des sous-parties opératives au fur et à mesure des besoins et y place les registres, constantes et opérateurs. Ce placement doit être fait tel qu'il n'y ait aucun conflit de partage de bus lors de l'exécution de chaque instruction opérative spécifiée dans l'algorithme d'interprétation du micro-processeur. Pour limiter la combinatoire et pour éviter de remettre en cause le placement des registres déjà placés, le compilateur Apollon utilise des heuristiques.

Parmi ces heuristiques on trouve celle qui consiste à placer les registres qui apparaissent souvent ensemble dans des transferts ou des opérations, à l'intérieur d'une même sous-partie opérative dans la mesure du possible. Pour plus de détails voir [JAMI 86].

3.6.7. Compilation de la partie contrôle

La compilation de la partie contrôle vient après celle de la partie opérative. Elle se décompose en deux phases :

-synthèse des étages de contrôle
-génération des PLAs

3.6.7.1. Synthèse des étages de contrôle

La synthèse des étages de contrôle est effectuée à partir des tables de transitions générées lors de l'extraction des spécifications de la partie contrôle [MHAY 86]. Le but de la synthèse est de générer les spécifications des matrices ET et OU des différents PLAs de contrôle. Les colonnes de la matrice ET sont constituées des entrées de l'étage de contrôle. Ce sont :

-les comptes rendus de la partie opérative,
-les requêtes de contrôle externes et internes,
-l'état constitué de 2 champs :

-le séquencement "externe" : l'appel d'un CMODULE par le niveau supérieur,
-le séquencement interne nécessaire au déroulement du CMODULE appelé.

Les lignes de la matrice ET sont les différents monômes et correspondent aux différentes transitions de l'étage de contrôle. Les colonnes de la matrice OU sont constituées des sorties de l'étage de contrôle. Ce sont :

-les ordres d'affectation des variables de contrôle internes et externes,
-les appels aux CMODULEs de niveau directement inférieur ou les commandes de la partie opérative s'il s'agit du dernier étage de contrôle,
-le nouvel état du CMODULE en cours d'exécution,
-et des signaux de synchronisation qui déclenchent le passage à un nouvel état de l'automate de contrôle supérieur si l'exécution du CMODULE est terminée ou qui passent le contrôle à l'étage inférieur si un nouveau CMODULE doit être exécuté.

Les lignes de la matrice OU correspondent bien sûr aux lignes de la matrice ET.

Les différentes étapes de la synthèse :

La synthèse commence par l'étage de contrôle immédiatement supérieur à la partie opérative. Avant de pouvoir générer les spécifications du PLA de génération des commandes de la partie opérative, le compilateur doit attendre le résultat de la compilation de cette dernière. L'architecture de la partie opérative ainsi que les ressources utilisées pour l'exécution de chaque instruction opérative doivent être connues. Les principales étapes de la synthèse de parties contrôle sont les suivantes [MIIAY 86] :

3.6.7.1.1. Mise sous forme canonique des expressions conditionnelles

1°) Les opérateurs complexes (par exemple exor,>,<) sont exprimés à l'aide des opérateurs de base suivants (and, or, not, ≠, =).

2°) Les opérateurs NOT sont éliminés, puis les opérateurs ≠ . Par exemple, l'expression NOT (A=1) est d'abord transformée en (A≠1), puis si A est un nombre à 2 bits A_0 et A_1 en $(A_0 = 0)$ OR $(A_1 = 1)$.

3°) Les instructions conditionnelles dont les conditions comportent l'opérateur OR sont éclatées.

Le but de ces 3 étapes est de se ramener à la forme canonique d'un monôme d'un PLA : E1*E2*...*Ep où Ei est de la forme (Ai = 0) ou (Ai = 1) ou 1 si l'on désigne par A1.... Ap les p entrées de la matrice ET.

3.6.7.1.2. Recensement des entrées et sorties du PLA

Cette étape consiste à recenser d'une part le nombre et la nature des entrées (comptes rendus de la partie opérative, commandes du niveau supérieur, requêtes de contrôle internes et externes) et des sorties (ordres d'affectation des variables de contrôle internes et externes, signaux de synchronisation des tranches de contrôle, appels de CMODULEs de niveau inférieur ou commandes de la partie opérative) que l'on ne peut coder librement et d'autre part les entrées et les sorties que l'on peut coder librement.

Dans le cas de SYCO, il faut tenir compte du fait que l'état est constitué de 2 champs :
- le séquencement interne (état courant du CMODULE en cours d'exécution)
- et le nom du CMODULE en cours d'exécution. Ce champ est bien sûr le même pour tous les états du CMODULE.

En conséquence, on ne peut coder librement que le champ de séquencement interne, puisque c'est l'étage supérieur qui détermine le nouveau CMODULE à exécuter. D'autre part, le premier état d'un Cmodule devra toujours avoir son champ de

séquencement interne codé par la valeur 0 puisque le tampon de séquencement interne est remis à 0 au début de chaque Cmodule.

Le codage des sorties destinées à l'étage inférieur est possible s'il ne s'agit pas de l'étage de génération des commandes de la partie opérative mais il remet en cause le codage des entrées de l'automate de contrôle inférieur.

3.6.7.1.3. Génération des PLAs

Pour générer les masques des PLAs, un générateur de partie contrôle (appelé GENPLA) a été réalisé [GERO 87]. Les PLAs utilisés ont une topologie de type classique. Ces sont des PLAs NOR-NOR en CMOS. Pour des raisons de vitesse et de consommation, on a choisi d'implémenter des PLAs à précharge avec des entrées en métal supérieur (poly doublé en alu2).
En effet, les matrices de PLAs que l'on extrait des descriptions comportementales des différentes tranches de contrôle, ont en général beaucoup de monômes et comparativement peu d'entrées/sorties (car il y a un codage). La longueur des fils d'entrée est alors importante, et on utilise du métal supérieur pour limiter le temps de propagation.

La matrice OU du dernier étage de contrôle est particulière car elle possède des sorties verticales (latérales) pour activer la partie opérative. Ces sorties sont réalisées en se connectant sur les lignes de sorties horizontales en métal inférieur avec une ligne verticale de métal supérieur. Cette ligne doit se trouver aussi près que possible (décalage à cause des rappels de masse) de la connexion à l'ampli de commande pour la partie opérative.
GENPLA assemble les matrices "ET-OU" du PLA avec les charges des monômes, les charges des sorties, les amplificateurs puis les registres d'entrées/sorties pour constituer un étage de contrôle. Actuellement, ce programme ne permet de générer que des PLAs classiques disposés verticalement. La figure suivante donne le schéma électrique d'un PLA :

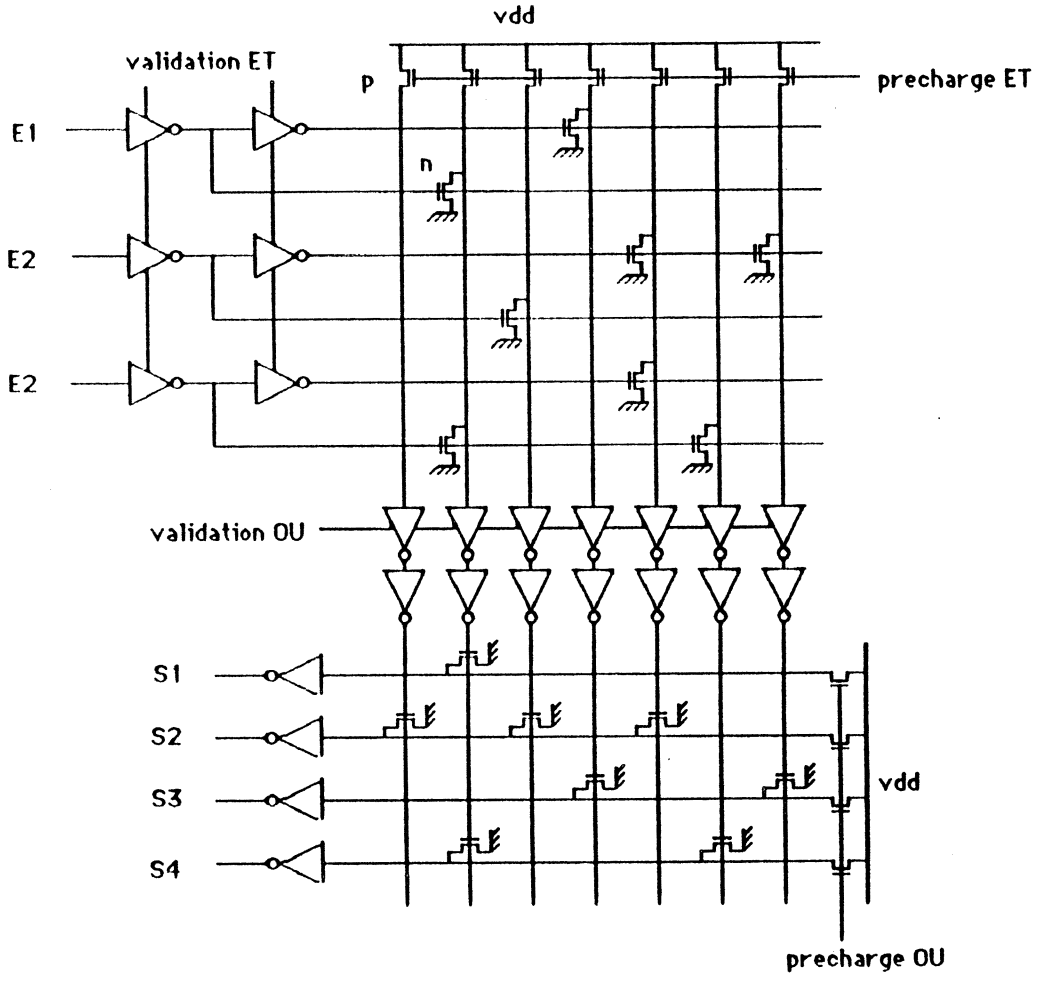


fig 1.13 : schéma d'un PLA CMOS à précharge

3.6.8. Assemblage global du circuit

L'ensemble des étages de contrôle et de la partie opérative une fois terminé, l'assemblage du circuit est réalisé. Cet assemblage consiste à :
-connecter les entrées/sorties des différents PLAs aux bus de contrôle, de compte-rendus et de synchronisation.
-connecter le dernier étage de contrôle à la partie opérative.

Autour du coeur du circuit ainsi formé, seront assemblés les plots de contrôle, les plots d'horloge, les plots de données et d'adresses. Les plots occupent ainsi le pourtour du circuit.

Un programme d'assemblage prototype a été réalisé [GERO 87]. Il permet d'évaluer les différentes cellules du circuit, et réalise le bus de connexion entre les étages. Il obtient les caractéristiques des différentes cellules à partir :
-du fichier lubrick de la cellule (fichier représentant la taille, la forme et les connexions),

-des descriptions des PLAs. Les nombres d'entrées, de sorties et de monômes permettent de calculer la taille et la connectique des différentes matrices (ET/OU).

Ce programme utilise le langage LISP et le système d'assemblage Nautile [BCHJ 87]. Le dessin des masques est généré en Lucie [PAILL 85], mais l'assemblage est fait à l'aide de Nautile. Dans la version actuelle, les cellules élémentaires sont décrites en Lucie-Lubrick. Ce programme accepte en entrée un fichier de description de la partie contrôle et les fichiers de description des différents PLAs générés par le compilateur de partie contrôle. Dans un futur proche, cette interface sera supprimée en utilisant directement la structure de données LDS générée par le compilateur de partie contrôle. On pourrait ainsi réaliser un logiciel qui permettrait de compiler un circuit de bout en bout (de la description au plan de masse) en restant dans l'environnement LDS.

4. Conclusion

L'originalité du système SYCO réside dans le fait qu'il est basé sur un modèle à la fois architectural et topologique. Ce choix restreint la portée du compilateur puisque l'organisation fonctionnelle et topologique du circuit est prédéfinie mais il permet en échange d'utiliser des algorithmes simples et efficaces pour la génération des masques. Le modèle architectural et le modèle topologique permettent aussi une évaluation rapide de la surface occupée par telle ou telle description, ce qui permet une comparaison rapide de différents résultats de la compilation.

D'autre part, le langage d'entrée de SYCO offre au concepteur toutes les possibilités offertes par les langages algorithmiques tels que PASCAL, ALGOL, etc... Le comportement d'un circuit VLSI complexe peut donc être spécifié par un programme écrit dans un langage tel que PASCAL, sans tenir compte des contraintes technologiques. Seule une partie déclaration oblige l'utilisateur à spécifier son système en terme de registres, bus, etc... Le langage d'entrée de SYCO permet aussi de décrire facilement des circuits complexes grâce à la hiérarchie définie dans le modèle architectural et à l'utilisation des concepts de blocs structurés et de procédures.

CHAPITRE II

EVALUATION DE SURFACE

1. Introduction

Un circuit généré par le compilateur SYCO peut être vu comme un rectangle dans lequel se trouvent deux autres rectangles de même largeur :
-en bas la partie opérative et ses amplificateurs de commande, notée PO,
-en haut la partie contrôle et son interface, notée PC.

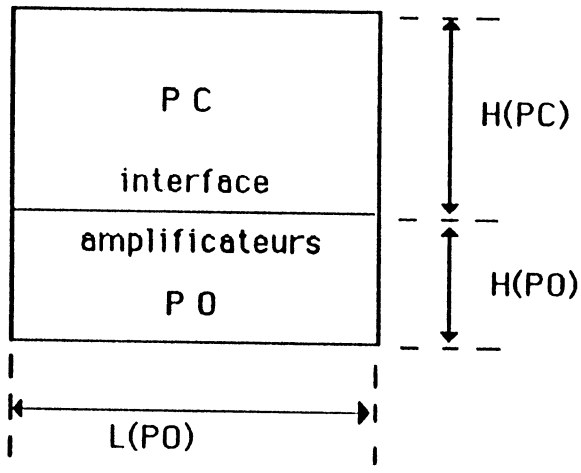


fig. 2.1 : forme globale d'un circuit généré par SYCO

Par conséquent, la surface d'un circuit est égale au produit de la largeur de la partie opérative par la hauteur du circuit. La hauteur du circuit est égale à la somme des hauteurs de la partie opérative et de la partie contrôle :

surface (circuit) = L(PO) * (H(PO) + H(PC))

2. Evaluation de la surface de la partie opérative

La partie opérative désigne ici à la fois les n tranches d'un bit du chemin de données du microprocesseur et la ou les tranches des amplificateurs des commandes de la partie contrôle.
La surface de la partie opérative dépend :
1) du parallélisme spécifié au niveau de l'algorithme de description du microprocesseur.
2) de la librairie de cellules utilisée pour générer le fichier des masques.

Dans toute la suite, on conviendra que le flux de données est parallèle à l'axe des x et que le flux de contrôle est parallèle à l'axe des y. On parlera de largeur dans le sens horizontal et de hauteur dans le sens vertical.

2.1. Evaluation de la hauteur de la partie opérative

La partie opérative résulte de l'empilement de n tranches fonctionnelles d'un bit. Chaque tranche a une hauteur fixe pour une technologie donnée. Par conséquent, le nombre de bits du chemin de données donne la hauteur H(PO) qui est égale à :

$$H(PO) = H(tranche) * NBIT + H(amplification)$$

où :

H(tranche) est la hauteur connue d'une tranche de 1 bit (BIT-SLICE)

NBIT est le nombre de bits,

et H(amplification) la hauteur de la tranche d'amplification

2.1.1. Hauteur d'une tranche

La hauteur d'une tranche est fonction :

1) du nombre de bus fonctionnels : les bus du modèle d'Apollon et les bus de services.

2) du nombre de bus d'alimentation

3) du fonctionnement électrique des bus : bus complémentés ou non.

4) de la largeur de chaque bus : largeur minimun imposée par la technologie ou largeur adaptée à la charge électrique des bus.

5) de l'espace entre chaque paire de bus voisins : distance minimum entre deux fils de métal ou distance majorée pour tenir compte des contacts.
la figure suivante donne un exemple de la tranche utilisée actuellement par le compilateur SYCO, règles Nmos du CMP; la hauteur est de 54 lambdas :

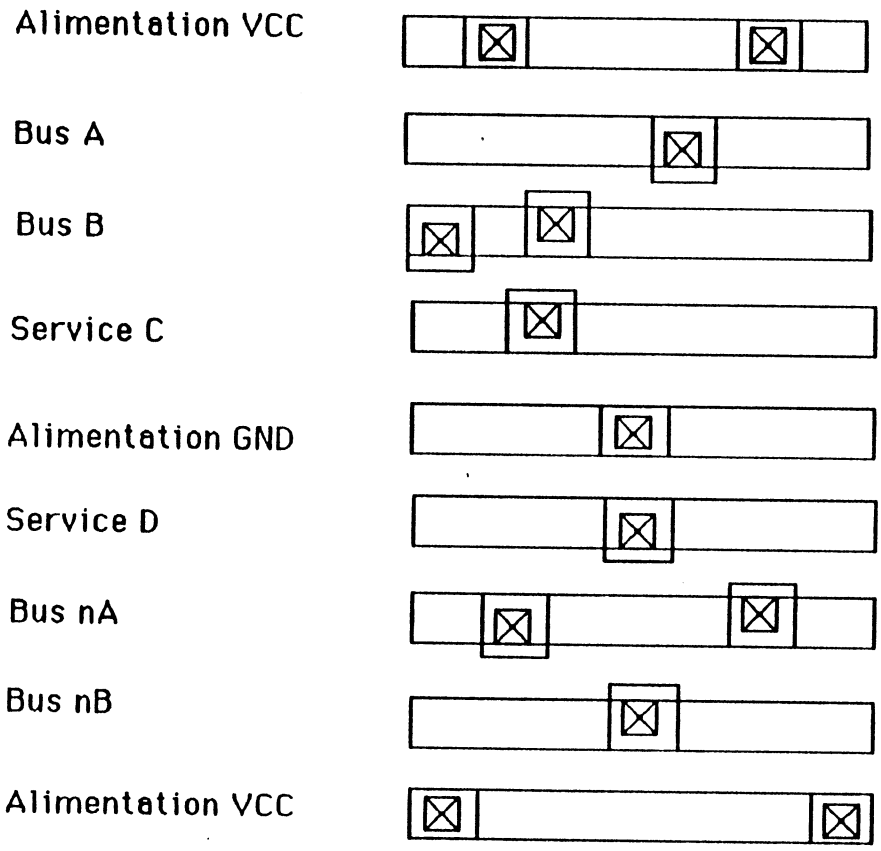


fig. 2.2 : dessin des masques d'une tranche de un bit

2.1.2. Hauteur de la tranche d'amplification

On suppose que la largeur de l'interface électrique est la même que celle du chemin de données.
La hauteur dépend :
- de la hauteur du ou des canaux de routages,
- de la hauteur d'une tranche; comme pour le chemin de données, on utilise des cellules de même hauteur que l'on assemble en tranches.
- du nombre de tranches.

Si les amplificateurs sont assemblés côte à côte, la largeur de la tranche d'amplification est en général supérieure à la largeur du chemin de données. Si on répartit les amplificateurs sur plusieurs étages de façon à ne pas dépasser la largeur

du chemin de données, on obtient une forme rectangulaire. Par contre, la hauteur de l'interface électrique entre partie opérative et partie contrôle est augmentée. La surface de cette interface est augmentée mais on obtient un bloc plus régulier qui sera connecté plus facilement au reste du circuit.

La hauteur d'une tranche est fixe et dépend de la technologie et de la stratégie d'implémentation. Le nombre de tranches est dans la pratique égal à 2.
Il ne reste alors qu'à estimer la hauteur des canaux de routage. Si l'on utilise un routage monocouche, la hauteur est égale au produit du pas de la couche choisie par le nombre maximum de lignes horizontales (le pas est égal à la somme de la largeur minimum d'un fil et de la distance minimum entre 2 fils - on se place dans le cas où tous les fils ont la même largeur). Ce nombre dépend de la position relative des connecteurs nord et sud.

2.2. Evaluation de la largeur de la partie opérative

Indépendamment de la technologie et de la stratégie d'implémentation des portes logiques, la largeur du chemin de données dépend :

1) du nombre de registres et de constantes déclarés par le concepteur.

2) du nombre de constantes dupliquées par le compilateur Apollon pendant la synthèse de la partie opérative.

3) du nombre d'opérateurs alloués aux opérations dans la description comportementale.

4) de la nature des opérations effectuées sur chaque sous-partie opérative.

5) du nombre de fils remontant vers la partie contrôle : fils de compte rendu d'un test (flags) ou d'une opération arithmétique ou fils des bits d'un registre accessible à la partie contrôle.

6) du nombre de fils descendant vers les plots. Ce nombre dépend :
-du nombre de bus d'entrée-sortie.
-du nombre de bits de chacun de ces bus.
-de la position des tampons d'entrée-sortie à l'intérieur de la partie opérative. S'ils sont placés sur les côtés, les sorties des tampons peuvent être horizontales et dans ce cas, aucune place n'a besoin d'être réservée pour faire passer des fils vers le bas.

La largeur est aussi fonction du nombre de connexions des différents éléments aux 2 bus du chemin de données et des connexions entre sous-parties opératives. Un registre peut être à simple accès ou à double accès. Les tampons d'entrée et de sortie des opérateurs ainsi que les tampons d'entrée-sortie des plots peuvent être eux aussi

connectés à un ou deux bus. Il n'existe pas de constantes à double accès. Il faut les dupliquer et en mettre une en connexion sur un bus et l'autre en connexion sur le second bus si l'on veut avoir l'équivalent d'une constante double accès.

Calculer la largeur de la partie opérative de manière exacte est très difficile. En effet, cette largeur dépend d'un grand nombre de paramètres qui ne seront finalement connus que lorsque l'assemblage de la partie opérative sera achevé. En particulier, il faut tenir compte non seulement de la taille des cellules fonctionnelles mais aussi de l'espacement entre cellules voisines, imposé par les règles technologiques. Toutefois, il est possible de fournir une estimation de la largeur de la partie opérative à partir d'une part :

-du nombre et de la nature des différents éléments fonctionnels
-de leurs connexions aux bus
-des connexions entre bus
-et d'autre part de la décomposition des éléments fonctionnels en cellules fonctionnelles et de la taille de ces cellules. Par exemple, une UAL est composée de plusieurs blocs fonctionnels :

-entrées de l'UAL
-2 demi-additionneurs
-et la sortie de l'UAL

Les entrées et la sortie de l'UAL sont elles-mêmes constituées de plusieurs cellules fonctionnelles.

Les gardes entre cellules seront approximées. En effet, les gardes ne représentent qu'une faible part de la largeur de la partie opérative; cette part est inférieure à 10 pour cent. De plus, il faut un grand nombre d'informations pour calculer ces gardes (voir la notion de frontière multi-segment et multi-niveau en Lubrick [SCHO 83]).

En Nmos les gardes peuvent prendre les valeurs 1, 2 ou 3. La garde n'est égale à 3 que si les fils en vis à vis sont en métal. On peut penser que cette situation n'est pas la plus fréquente d'autant plus que le niveau métallique n'est utilisé en principe dans la partie opérative que pour réaliser les bus. Or les bus n'interviennent pas dans le calcul de la garde lorsque l'on utilise le principe d'implémentation classique en VLSI : les connexions en métal au dessus de la logique. Il faut cependant tenir compte de la garde entre contacts métalliques qui ne sont pas sur le même bus. On prendra donc en moyenne une valeur de 2 pour la garde; cette valeur devrait conduire à une approximation par excès. Même si les gardes ne sont pas calculées de manière exacte, on peut cependant obtenir une très bonne évaluation de la surface à partir de la structure fonctionnelle de la partie opérative. On distinguera donc deux modes d'évaluation.

2.2.1. Evaluation fine

La partie opérative est décrite sous la forme d'un ensemble d'éléments fonctionnels interconnectés. La largeur de la partie opérative est la somme des largeurs des cellules dont chaque élément fonctionnel est constitué et des gardes intercellulaires.

2.2.2. Evaluation grossière

On ne dispose que d'une description comportementale : l'ensemble des actions à exécuter par la partie opérative. Pour obtenir une approximation par excès, on considèrera :

-que chaque registre est connecté aux deux bus.

-qu'il y a 2*n constantes; n est le nombre de constantes déclarées par l'utilisateur dans la description. En effet, comme on ne dispose pas de constantes double accès, on duplique les constantes : on en connecte une sur un bus et l'autre sur le second bus.

-que toutes les sous-parties opératives voisines sont connectées entre-elles par les deux bus.

-on prendra pour largeur d'une boîte à opérations une valeur moyenne.

-que tous les tampons d'entrée-sortie ont des sorties verticales.

Dans ce cas on obtient la formule suivante qui permet d'estimer la largeur de la partie opérative à partir de sa description comportementale :

$$L(PO) = Nreg \cdot Lreg + Nbop \cdot Lbop + 2.(Ncons \cdot Lcons) + (Npop - 1) \cdot Lcnex$$

où :

Nreg = nombre de registres déclarés par l'utilisateur
Lreg = largeur d'un registre à double accès
Ncons = nombre de constantes déclarées par l'utilisateur
Lcons = largeur d'une constante
Nbop = nombre de boîtes à opérations
Lbop = largeur moyenne d'une boîte à opérations
Npop = nombre de sous-parties opératives

calcul du nombre d'opérateurs (Nbop) :

Ce paramètre est égal au nombre maximal d'opérations unaires ou binaires

exécutées en parallèle.

Calcul du nombre de sous-parties opératives (Npop) :

Une sous-partie opérative peut exécuter en un seul cycle machine :
-4 transferts de sources distinctes (deux en phi1 et deux en phi2),
-ou une opération binaire et un transfert de registre,
-ou une opération unaire et 2 transferts de registres.
On peut donc calculer d'une manière exacte le nombre de sous-parties opératives nécessaires pour l'exécution de chaque instruction opérative.
Soit une instruction i contenant n opérations binaires, m opérations unaires et p transferts dont les sources ne figurent pas comme opérandes dans les opérations (les transferts dont les sources sont aussi opérandes d'une ou plusieurs opérations ne sont pas pris en compte puisqu'ils peuvent être exécutés en même temps que le transfert des opérandes vers les entrées des opérateurs où doivent s'exécuter ces opérations). Le nombre de sous-parties opératives nécessaires pour l'exécution de cette instruction est calculé de la façon suivante :

Si p-n-2.m est strictement positif :

-si le reste de la division entière de (p-n-2.m) par 4 est non nul alors le nombre de sous-parties opératives (noté $Npop_i$) nécessaire pour implanter les registres de l'instruction i est égal à :

$$Npop_i = n + m + (p-n-2.m)/4 + 1$$

-sinon :

$$Npop_i = n + m + (p-n-2.m)/4$$

Si p-n-2.m est négatif ou nul :

$$Npop_i = n + m$$

On calcule ainsi pour chaque instruction i le nombre de sous-parties opératives nécessaires pour son exécution. le nombre de sous-parties opératives utilisé pour le calcul de L(PO) est égal au nombre maximum rencontré :

$$Npop = \underset{i = 1, net}{MAX} (Npop_i)$$

avec net = nombre d'instructions opératives.

L'écart entre le nombre réel de sous-parties opératives (c.a.d celui généré par Apollon) et celui généré par l'algorithme ci-dessus dépend de la nature des opérations exécutées dans chaque instruction et des relations qui peuvent exister entre les différentes instructions opératives. Dans cet algorithme les instructions opératives sont analysées indépendamment les unes des autres. Or, on remarque que plus le nombre d'instructions opératives ayant plus de 2 registres opérandes en commun est grand, plus le nombre de sous-parties opératives tend à augmenter. Ceci est lié aux limites du modèle architectural de la partie opérative. L'exemple suivant permet de mieux clarifier ce problème :

exemple : Soient les 2 instructions opératives suivantes :
<e1> : A := B + C ; D := E - F
<e2> : G := B - E ; H := I - F

D'après l'algorithme décrit précédemment, il faut 2 sous-parties opératives pour implanter ces deux instructions, alors qu'en réalité il en faut 3. Ces instructions possèdent des registres-opérandes communs : B, E, et F. Pour que les conditions nécessaires de placement des opérandes des opérations binaires soient vérifiées, le compilateur Apollon rajoute une troisième sous-partie opérative (au milieu de la partie opérative et sans opérateur) (fig 2.3). De cette manière, le problème de partage de bus est évité au moment du chargement des opérandes pendant la phase phi1; la figure suivante donne une implantation possible des registres de ces 2 instructions :

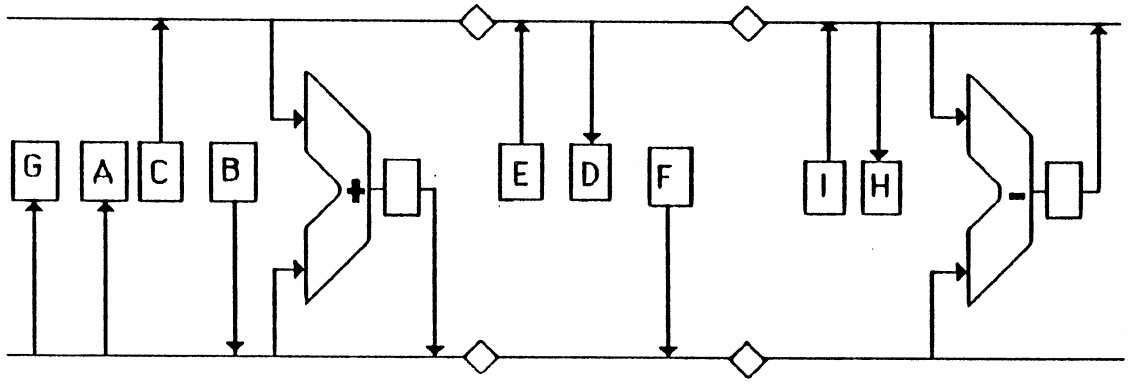


fig. 2.3 : exemple de partie opérative exécutant <e1> et <e2>

2.2.3. Largeur des cellules de la bibliothèque Nmos

Le choix de la librairie de cellules a une grande influence sur la taille de la partie opérative. Actuellement, le compilateur SYCO utilise une bibliothèque de cellules précaractérisées [SUZI 81] qui ont été dimensionnées et testées pour fonctionner à une fréquence donnée. L'utilisation d'une bibliothèque standard pour la compilation de silicium ne permet pas d'adapter les cellules de base aux contraintes spécifiées par

l'utilisateur (surface - rapidité - consommation). Les cellules ont en effet été conçues pour fonctionner à une fréquence donnée et leurs tailles ont été optimisées pour cette utilisation. Si l'on veut fabriquer un circuit moins rapide, on se retrouve avec un circuit surdimensionné qui consomme plus que nécessaire. On ne peut évidemment obtenir un circuit fonctionnant à une fréquence plus élevée, le seul moyen de pallier à cet inconvénient est d'augmenter le parallélisme entre les actions opératives. Nous donnons dans ce qui suit les largeurs des cellules de la bibliothèque Nmos utilisée actuellement par le compilateur SYCO; les largeurs sont données en lambdas :

registre simple accès : 20
registre double accès : 24
constante : 9
connecteur simple bus : 12
connecteur double bus : 20
précharge : 14
peignes d'alimentation du chemin de données : 4
tampon d'entrée-sortie : 86
tampon de sortie : 56
pas d'un fil de sortie vertical : 4
pas d'un fil de compte-rendu : 4

Boîte à opérations : La boîte à opérations se compose de 3 parties :

1) les entrées : il y en a une ou deux; une entrée peut être multiplexée : la première s'il y en a deux; la seconde entrée peut incorporer les constantes 0 et 1 : on utilise un interrupteur pour sélectionner l'entrée ou une des deux constantes.
-entrée : 24
-multiplexeur : 16
-interrupteur : 10
-constante : 10

2) le corps de la boîte à opérations : cela peut être un incrémenteur, un additionneur ou une UAL. Chacun de ces opérateurs comprend 2 parties :
une partie qui lui est propre :
-incrémenteur : 12
-additionneur : 41
-UAL : 56
et une partie commune calculant la retenue : 54 (64 si on ajoute l'indicateur arithmétique zéro : c'est un nor distribué).

De plus on peut ajouter un décaleur :
- qui décale à droite : 23
- qui décale à gauche : 19
- qui décale dans les deux sens : 32

3) et de la sortie :
-simple : 22
-double : 36

La taille d'une boîte à opérations "maximum" est de l'ordre de 280. Toutes les cellules du chemin de données ont une hauteur de 54 lambdas.
Toutefois la première tranche correspondant au premier bit a une taille de 64 lambdas : il faut donc rajouter 10 au résultat de la multiplication de la hauteur par le nombre de bits.

Interface électrique :

amplificateur standard : 17
amplificateur avec multiplexeur incorporé : 53
cellule de passage : 2
contact : 4

Toutes les cellules de la tranche des amplificateurs ont une hauteur de 79 lambdas.

3. Evaluation de la surface d'un étage de contrôle

La figure suivante représente un étage de contrôle avec ses connexions d'entrées/sorties aux bus de contrôle, de comptes rendus et de synchronisation.

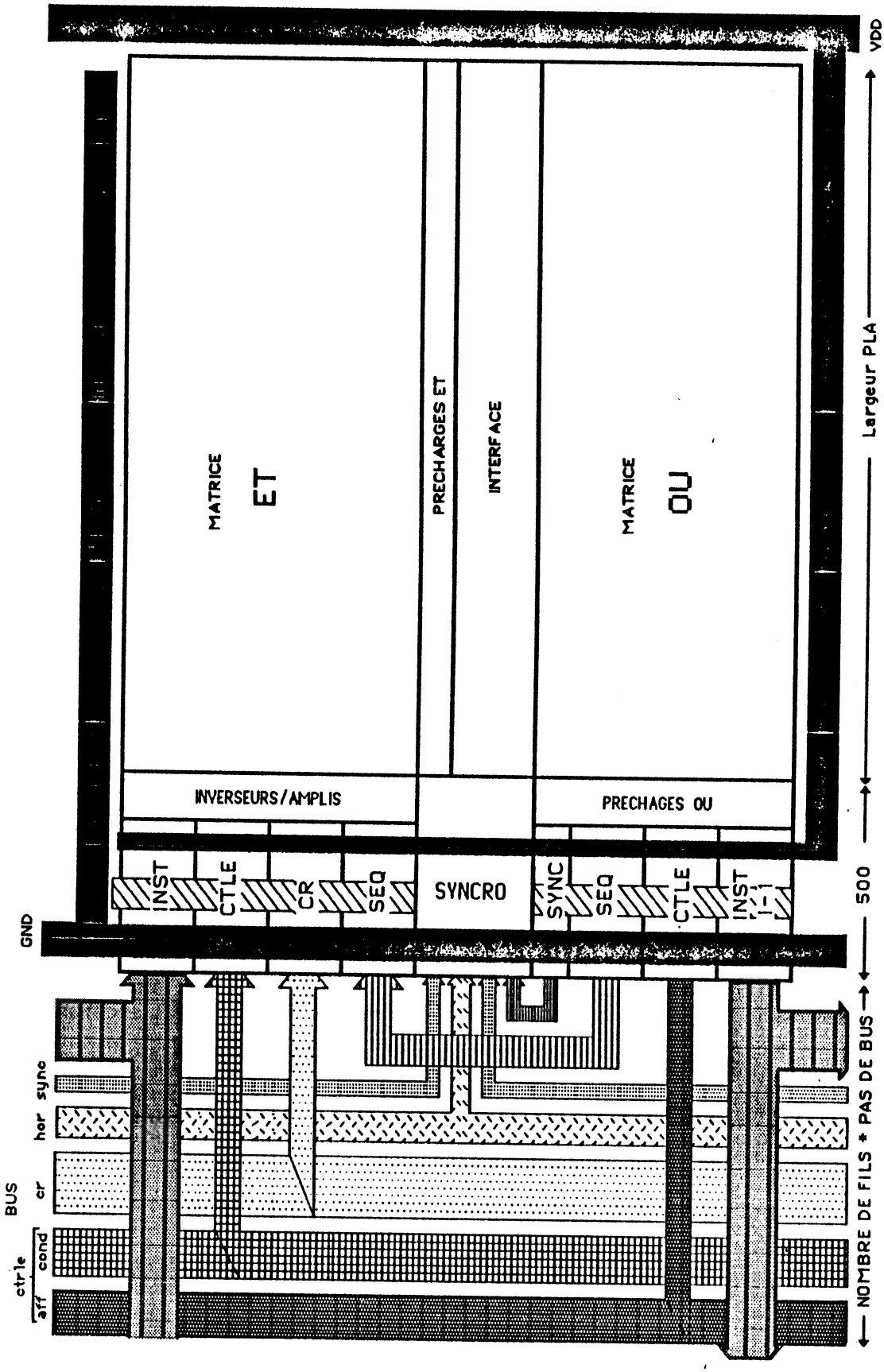


fig. 2.4 : topologie d'un étage de contrôle

Le PLA est de type classique. La matrice ET décode l'instruction, les comptes rendus de la partie opérative, les requêtes de contrôle externes et internes et l'état de l'automate. La matrice OU génère les sorties et en particulier les commandes de la partie opérative ou les commandes de la partie contrôle de niveau inférieur et le nouvel état de l'automate simultanément.
La matrice ET est topologiquement au-dessus de la matrice OU : chaque entrée (resp. sortie) occupe une ligne de la matrice ET (resp. OU). Les termes produits ou monômes en occupent chacun une colonne. Par conséquent, la largeur d'un étage de contrôle dépendra essentiellement du nombre de monômes, par contre sa hauteur dépendra du nombre d'entrées/sorties.

Les formules d'évaluation de la surface d'un PLA classique sont établies dans [REIS 83]. Soient :

NME : nombre de monômes du PLA,
NE : nombre d'entrées,
NS : nombre de sorties,
PE : pas d'entrée (11 lambdas),
PS : pas de sortie (11 lambdas),
PME : pas de monômes (on prendra 4 monômes pour 49 lambdas),

alors :

Hauteur (ET) = 2 . (NE . PE)
Hauteur (OU) = NS . PS

Hauteur (PLA) = Hauteur (ET) + hauteur (OU) + Hauteur (int)

avec :

Hauteur(int) : hauteur de l'interface entre les matrices ET, OU. L'estimation par excès de cette hauteur est de l'odre de 199 lambdas, pour tous les étages de contrôle (fig. 2.4).

Largeur(PLA) = NME . PME

Le nombre d'entrées NE de la matrice ET est égal à la somme de :
-nombre de fils de contrôle entrants (noté CTLE sur la figure 2.4)
-nombre de fils de comptes rendus (noté CR)
-nombre de fils de séquencement interne (noté SEQ)
-nombre de commandes provenant de l'étage de contrôle supérieur (noté INST).

Le nombre de sorties NS de la matrice OU est égal à la somme de :
-nombre de fils de contrôle sortants,

-nombre de fils de séquencement interne,
-nombre de commandes pour l'étage de contrôle inférieur,
-nombre de fils de synchronisation (noté SYNC sur la figure 2.4). Ce nombre est égal à 2 pour les étages de contrôle extrêmes et à 4 pour les autres étages.

Détermination du nombre de fils d'entrées/sorties à partir de la description comportementale :

Dans le cas général, les informations concernant un étage de contrôle sont calculées de la manière suivante :

1) le nombre de fils de séquencement interne = $Log_2(N)$ où N est la longeur maximale d'une séquence d'instructions (ou procédure) interprétée par l'étage de contrôle.

2) le nombre de fils de contrôle entrants = nombre de variables de contrôle utilisées dans des tests.

3) le nombre de fils de contrôle sortants = (n * p) où p est le nombre de variables de contrôle modifiées par l'étage de contrôle et n le nombre de bits de ces variables

4) le nombre de fils de comptes rendus = somme des nombres de bits des registres de la partie opérative testés par l'étage de contrôle.

5) le nombre de commandes pour le niveau inférieur = $Log_2(E)$ où E est le nombre d'instructions "EXECUTE" distinctes ou d'instructions opératives distinctes (si c'est le dernier étage de contrôle) présentes dans la description comportementale de l'étage inférieur.

6) le nombre de commandes du niveau supérieur = Log_2 (P) où P est le nombre de procédures interprétées par l'étage de contrôle.

Pour obtenir la surface totale d'un étage de contrôle, il faut rajouter à la surface du PLA la surface des amplificateurs d'entrée, des inverseurs de sortie et des blocs de mémorisation. La surface des inverseurs de sortie dépend de la charge qui se trouve en sortie de cet inverseur ainsi que des performances que l'on désire attribuer au PLA. Plus la charge en sortie est importante (capacité élevée, résistance importante), plus l'inverseur de sortie sera important si l'on désire des performances élevées. La surface des amplificateurs d'entrée est liée à la taille du PLA et de sa matrice ET [DAND 83]. Une estimation par excès des largeurs des amplificateurs d'entrées et des inverseurs de sortie a donné comme résultat 500 lambdas (fig. 2.4).

Remarque :
Lorsqu'on considère le modèle classique d'implantation d'un PLA pour lequel les entrées/sorties ne sont disponibles que perpendiculairement aux monômes du PLA, on est confronté aux problèmes de définir une (ou des) solution(s) topologique(s) convenable(s) vis-à-vis du voisinage du bloc constitué par le PLA classique. Dans ce cas, nous pouvons définir 2 positionnements du PLA :
-soit la matrice ET est au-dessus de la matrice OU (cas actuel du générateur de PLA)
-soit la matrice ET est à gauche de la matrice OU. Le PLA possède le même nombre d'entrées/sorties que la solution précédente. Les fils d'entrée entrent par le nord du PLA, et les fils de sorties sortent par le sud du PLA.
Le choix entre l'une ou l'autre de ces 2 solutions est lié aux nombres de commandes venant du niveau supérieur et allant vers le niveau inférieur, et au nombre de fils de contrôle et de comptes rendus. Par exemple, si le nombre de commandes est très important, comparativement au nombre de monômes, on choisira plutôt la deuxième solution.

4. Evaluation de la vitesse d'exécution

L'évaluation de la vitesse consiste à trouver le nombre de cycles nécessaires pour l'exécution de chaque instruction du microprocesseur. Pour une fonction F, le temps t(F) nécessaire à son exécution est égal à :

$$t(F) = N(F) \cdot t(cycle)$$

avec : N(F) = nombre de cycles nécessaires à l'exécution de la fonction F

t (cycle) = temps de cycle.

détermination du nombre de cycles :

Le nombre de cycles N(F) dépend du nombre de procédures nécessaires pour l'exécution d'une fonction F et de la fréquence d'exécution de ces procédures.
Soit une fonction F nécessitant pour son exécution k procédures (notées Pi); alors le nombre de cycles nécessaires pour l'exécution de cette fonction est donné par l'expression suivante :

$$N(F) = \sum_{i=1}^{k} freq(Pi) \cdot N(Pi)$$

avec :
N(Pi) = nombre de cycles nécessaires pour l'exécution de la procédure Pi.
freq (Pi) = nombre d'appels à la procédure Pi pendant l'exécution de la fonction F. Ce nombre n'est pas obligatoirement un nombre entier, mais il représente une

moyenne calculée à partir d'un certain nombre d'exécutions typiques de la fonction F.

Calcul de N(Pi) : deux cas sont à considérer :

-Si la procédure Pi ne contient pas de boucle (c.a.d d'instructions itératives) ce nombre est égal à la longueur du chemin le plus long dans le diagramme de transitions (ou graphe de contrôle) correspondant à la procédure Pi. Ce graphe est construit en affectant à chaque instruction un sommet de ce graphe, et à chaque branchement (conditionnel ou non) un arc.
-Si la procédure Pi contient des boucles :
- si on connait exactement le nombre de passages dans chaque boucle, on peut déterminer la longeur du chemin le plus long de manière précise.
- s'il existe au moins une boucle dont le nombre d'itérations dépend d'un certain paramètre on ne peut déterminer le nombre de cycles nécessaires pour l'exécution de la procédure. Dans ce cas, si l'on dispose d'un simulateur fonctionnel on peut choisir comme facteur N(Pi) le nombre de cycles moyen :

$$N(Pi) = 1/n \sum_{i=1}^{n} N_i(Pi)$$

avec :
n = un certain nombre d'observations (c.a.d d'exécutions de la procédure Pi)
$N_i(Pi)$ =nombre de cycles pour l'exécution de la procédure Pi pendant l'observation numéro i.

détermination du temps de cycle :

Pour déterminer le temps de cycle de la partie contrôle, il faut connaître les temps de propagation des signaux à travers tous les étages qui constituent la partie contrôle. Le temps de cycle doit être supérieur ou égal au temps de propagation maximal du signal à travers un étage de contrôle.

Le temps de réponse d'un étage de contrôle peut être analysé en utilisant un simulateur tel que MSYNC, SPICE, MOTIS, DIANA, SPLICE, etc... Ces systèmes permettent de décrire le circuit à étudier sous forme d'un réseau de transistors, parfois prenant en compte aussi l'environnement, simulé à un niveau moins fin. L'inconvénient majeur de tous ces systèmes est leur faible capacité : quelques centaines de transistors peuvent être décrits au maximum. Cela n'est pas suffisant pour étudier le cas des éléments tels qu'une ROM ou un PLA, qui contiennent des milliers de transistors.

Divers études ont été faites sur le calcul du temps de propagation des signaux dans les PLAs [BONN 81], [DAND 83]. Cagnola et al [CAGN 85] proposent une méthode simple pour l'estimation du temps de réponse d'un PLA. Elle consiste à chercher le chemin critique dans le PLA. Une fois que ce chemin est identifié, une simulation électrique et temporelle est faite sur ce chemin à l'aide du simulateur SPICE. Le programme de recherche du temps de réponse d'un PLA utilise les informations contenues dans la matrice de personnalité du PLA, ainsi que d'autres informations telles que les dimensions des transistors et l'architecture utilisée pour implémenter le PLA.

Un chemin est défini comme un triplet (i,p,o) où i est une colonne d'entrée, p un terme produit et o une colonne de sortie (fig.2.5) :

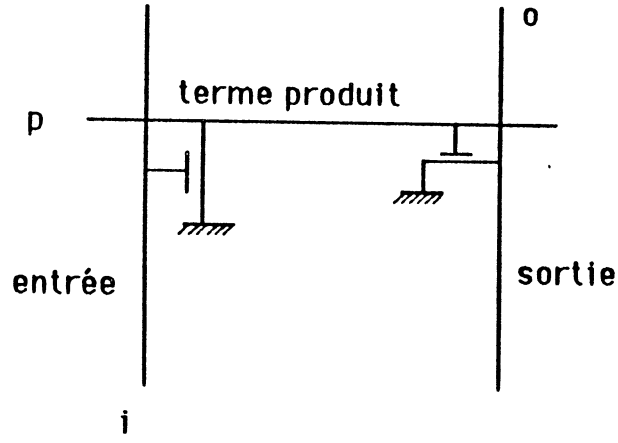


fig. 2.5 : chemin dans un PLA

Pour qu'un chemin existe, il faut deux transistors : un à l'intersection de la colonne d'entrée i et de la ligne de monômes p et l'autre à l'intersection de la ligne p et de la colonne de sortie o. (fig. 2.6).

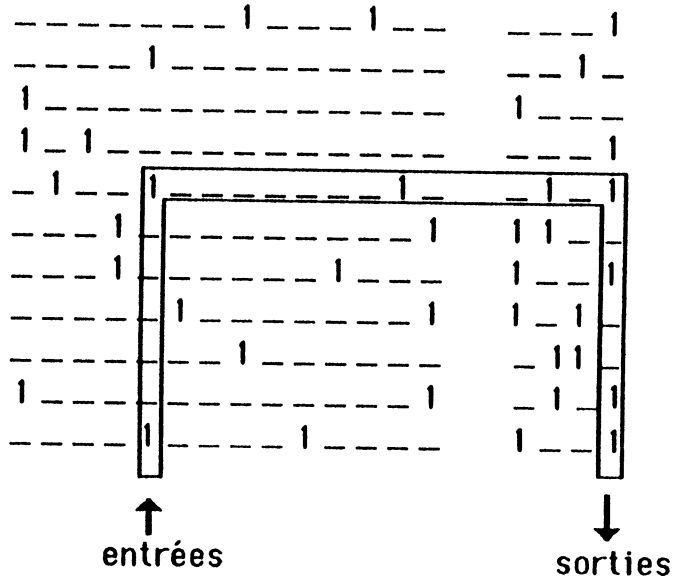


fig. 2.6 : exemple de chemin dans un PLA
décrit par sa matrice de personnalité

Les caractéristiques électriques d'un chemin sont exprimées par quatre constantes. Chacune de ces constantes est un couple qui contient la résistance et la capacité d'une partie du chemin : la ligne d'entrée, la ligne des termes produits, la ligne de sortie et les capacités externes. Le programme analyse la matrice de personnalité du PLA et pour chaque chemin existant, il calcule les valeurs de ces quatre constantes. Les temps de réponse des différents chemins sont ensuite calculés à partir de ces valeurs. L'évaluation du temps de propagation du signal dans un chemin est calculé dans les deux cas de front montant et de front descendant appliqués au tampon d'entrée. Le chemin critique est celui dont le temps de réponse est le plus élevé.

5. Conclusion

L'utilisateur d'un compilateur de silicium est principalement intéressé par deux caractéristiques de son circuit : sa vitesse d'exécution et sa surface. Il est donc souhaitable qu'un compilateur de silicium dispose d'un programme qui détermine en un temps relativement court si la description d'entrée introduite par le concepteur conduira à un bon compromis surface-vitesse. L'évaluation doit bien sûr être effectuée avant que le dessin des masques ne soient généré, la génération des masques étant une des étapes les plus coûteuses en temps CPU.

Dans le compilateur SYCO, l'évaluation peut porter sur deux types de descriptions :

-la description comportementale des tranches de contrôle et de la partie opérative après la phase d'extraction; il s'agit alors d'une évaluation grossière qui permet de

déterminer le nombre et la taille approchée des étages de contrôle et une estimation du nombre de sous-parties opératives.

-la description structurelle des PLAs de contrôle et de la partie opérative après la phase de synthèse. Il s'agit d'une évaluation plus fine qui permet de déterminer la taille des différents PLAs de contrôle ainsi que celle de la partie opérative.

Suivant le résultat de l'évaluation, deux types de modifications de la descriprition comportementale peuvent être effectuées :

-des optimisations : on entend par optimisations un ensemble d'actions dont le but est d'améliorer à la fois la surface du circuit et ses performances, ou du moins qui améliorent l'un sans dégrader l'autre,
-des transformations visant à la recherche d'un meilleur compromis surface performance qui consiste par contre à échanger une perte de performance contre un gain de surface ou vice versa.

Dans le cadre de cette thèse, trois outils d'évaluation ont été réalisés :
-un programme d'évaluation rapide de la largeur de la partie opérative. Ce programme analyse les instructions opératives indépendamment les unes des autres. Par conséquent l'évaluation est rapide mais peu précise. Ce programme est utilisé lorsque le nombre d'instructions opératives possédant plus de 3 registres opérandes en commun est nul ou peu élevé.
-un programme plus sophistiqué que le premier a été mis en oeuvre pour une évaluation plus fine de la largeur de la partie opérative. Ce programme tient compte de certaines combinaisons d'opérandes entre des opérations binaires exécutées en parallèles.
-un programme d'évaluation de la taille de la partie contrôle; ce programme utilise des formules classiques d'évaluation de la taille d'un PLA constitué de 2 matrices "ET-OU".

CHAPITRE III

OPTIMISATION

1. Introduction

La plupart des compilateurs de silicium font une optimisation de haut niveau lorsque la description comportementale du circuit est encore à un niveau très abstrait, ce qui rend le problème plus simple en réduisant le nombre d'éléments intervenant dans le processus d'optimisation. On entend ici par optimisation toute transformation algorithmique de la description comportementale qui a pour effet de réduire la surface du circuit sans diminuer ses performances ou d'augmenter ses performances mais sans provoquer une augmentation de sa surface.
L'étape d'optimisation de haut niveau dans les compilateurs de silicium dont le langage d'entrée est un langage algorithmique consiste à :
1) augmenter la vitesse du circuit en réduisant le nombre de transitions entre les états de l'algorithme.
2) réduire la surface du circuit en :
- réduisant le nombre d'états de l'algorithme d'entrée (optimisation de la surface de la partie contrôle)
- réduisant le nombre de ressources matérielles (tels que les registres, les opérateurs, les multiplexeurs, etc...) allouées aux instructions opératives pendant la génération de la partie opérative. Cette phase de l'optimisation correspond à l'optimisation de la surface de la partie opérative.

Pour réduire le nombre d'états et de transitions de l'algorithme d'interprétation des instructions ainsi que le nombre de ressources de la partie opéartive, les compilateurs de silicium utilisent des techniques d'optimisation similaires à celles employées par les compilateurs optimiseurs de langages de programmation évolués [AHO 77]. Parmi ces techniques on trouve :
-l'expansion de procédures appelées une seule fois
-la propagation des constantes
-l'élimination des expressions redondantes
-le déplacement des actions invariantes à l'extérieur des boucles.
Si le langage d'entrée du compilateur de silicium permet de spécifier le parallélisme entre les actions opératives, ou si l'architecture de la partie opérative permet d'exécuter plusieurs opérations simultanément, le compilateur de silicium peut aussi utiliser les techniques de compactage de microcode [TOKO 81], [DAVI 81], [AGER 76]. Le compactage de microcode consiste à combiner les microopérations en le plus petit nombre de microinstructions afin de réduire la taille et le temps d'exécution de l'algorithme d'interprétation des instructions du microprocesseur.

Nous présentons dans ce chapitre comment certaines de ces techniques d'optimisation classiques sont exploitées au niveau du compilateur SYCO, ainsi que leurs conséquences sur les performances et la surface du circuit final. D'autres transformations qui visent à réduire le nombre de composants de la partie opérative tout en maintenant les performances du circuit seront décrites à la fin de ce chapitre.

2. Réduction du nombre d'états de l'algorithme d'interprétation des instructions

La réduction du nombre d'états d'un algorithme est un problème bien connu des concepteurs de compilateurs de langages de programmation [AHO 77]. Elle consiste essentiellement à réduire le nombre de dépendances qui peuvent exister entre ces états. En général, les programmeurs attachent peu d'importance au nombre de dépendances qu'ils peuvent introduire quand ils expriment leur algorithme dans un langage de programmation. Néammoins, si un programme doit être exécuté sur une machine avec un système d'exploitation permettant le parallélisme, la réduction du nombre de dépendances entre les états de l'algorithme peut augmenter considérablement la vitesse d'exécution du programme.
La réduction du nombre d'états d'un algorithme peut être effectuée à différents niveaux :
- quand un langage est choisi pour écrire un programme
- quand un algorithme est exprimé comme un programme dans ce langage
- quand le programme est compilé
- et enfin, quand il est exécuté.

Les compilateurs optimiseurs de langages de programmation évolués réduisent automatiquement le nombre d'états de l'algorithme. Pour cela ils génèrent d'abord une structure intermédiaire appelée "graphe de précédence" [AHO 77], [KUCK 81]. Des transformations sont ensuite appliquées sur ce graphe pour réduire le nombre d'arcs et de sommets afin d'aboutir à un code objet optimal. Divers études ont été faites sur les techniques de réduction de graphes de précédences. Par exemple, le "déplacement de code" (code motion), la "duplication de variables" (variable renaming), et autres, sont des techniques traditionnelles qui conduisent à un graphe de précédence optimal [AHO 77].

Comme les compilateurs de langages de programmation évolués, les compilateurs de silicium dont le point de départ est une description algorithmique, génèrent eux aussi une structure intermédiaire (en général un graphe) qui leur permet d'effectuer des optimisations locales ou globales sur la description comportementale [ORAI 86]. Cette structure permet au compilateur de silicium d'avoir une vue globale sur le flot de données (décrit par les instructions d'affectation dans la description) et sur le flot de contrôle (décrit par les boucles, les instructions conditionnelles et les procédures).

Nous donnons dans ce paragraphe, quelques exemples de compilateurs de silicium utilisant le concept de graphe de précédence pour optimiser la description comportementale du circuit.

Dans le compilateur SYCO, la structure intermédiaire utilisée pour optimiser la description comportementale correspond aux différentes tables de transitions extraites de la description comportementale pour la synthèse de la partie contrôle.

Chaque table de transitions décrivant le comportement d'un étage de contrôle est optimisée indépendamment des autres, en partant de l'étage de contrôle juste au-dessus de la partie opérative, et ainsi de suite jusqu'au premier étage. La table de transitions décrivant le comportement de l'étage de contrôle juste au-dessus de la partie opérative est particulière; en effet, cet étage permet d'actionner directement la partie opérative, c'est pourquoi nous étudierons séparément dans la suite de ce chapitre, les transformations d'optimisations appliquées à cette table de transitions.

Le compilateur de silicium SILI [KAHR 85] génère à partir de la description comportementale, deux graphes orientés :
- un graphe de flot de données ("data flow graph" en anglais) où chaque noeud représente une variable ou un opérateur. Un arc relie deux noeuds ei et ej si des données sont transmises du noeud "source" ei vers le noeud "destination" ej.
- un graphe de flot de contrôle ("control flow graph") où chaque noeud correspond à une instruction et un arc (ei ej) existe si l'instruction ej est exécutée directement après l'instruction ei.

exemple :

Soit la description comportementale suivante :

```
F := 1
While (n <> 1)
 begin
      F := F * N
      N := N - 1
end
```

Cette description est transformée par le compilateur SILI en 2 graphes :

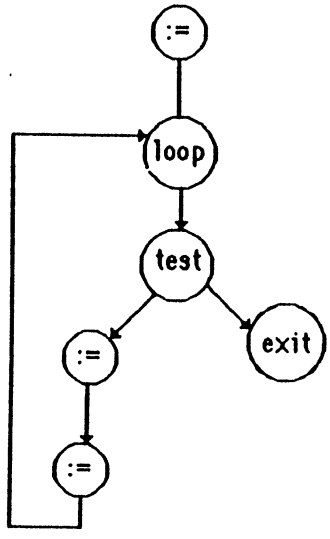


fig.3.1 : graphe de flot de contrôle

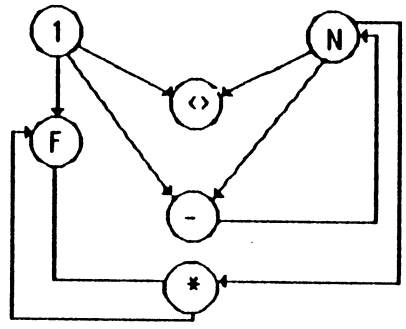


fig.3.2 : graphe de flot de données

Le CMU-DA est un autre exemple de compilateur de silicium qui optimise la description comportementale en utilisant les graphes de précédence. Il génère une structure intermédiaire (appelée "value trace" en anglais) représentant à la fois le flot de données et le flot de contrôle [SNOW 78]. Pour cela, la description comportementale est d'abord divisée en segments ne comportant qu'un seul point d'entrée situé au début du segment. Un segment (appelé "VT-body") peut être soit une procédure, une boucle ou une séquence d'instructions dont seule la première est étiquetée. La structure générée est donc un graphe acyclique où chaque noeud correspond à un segment. Un arc (Si Sj) existe si le segment Si contient une instruction de branchement vers le point d'entrée du segment Sj.
Chaque segment est lui-même représenté par un graphe orienté tel qu'un noeud correspond à une variable ou à un opérateur, et un arc (ei ej) signifie le transfert d'une donnée de ei vers ej.

exemple :

soit la description comportementale suivante, écrite dans le langage ISPS [BARB 81] utilisé comme langage de description de circuits par le CMU-DA :

```
e1 : A <- B + C next
     B <- C - D next
     C <- A * B
     IF C > D next e2 else next e3
```

Cette description est transformée par le CMU-DA en une structure représentée par la figure suivante :

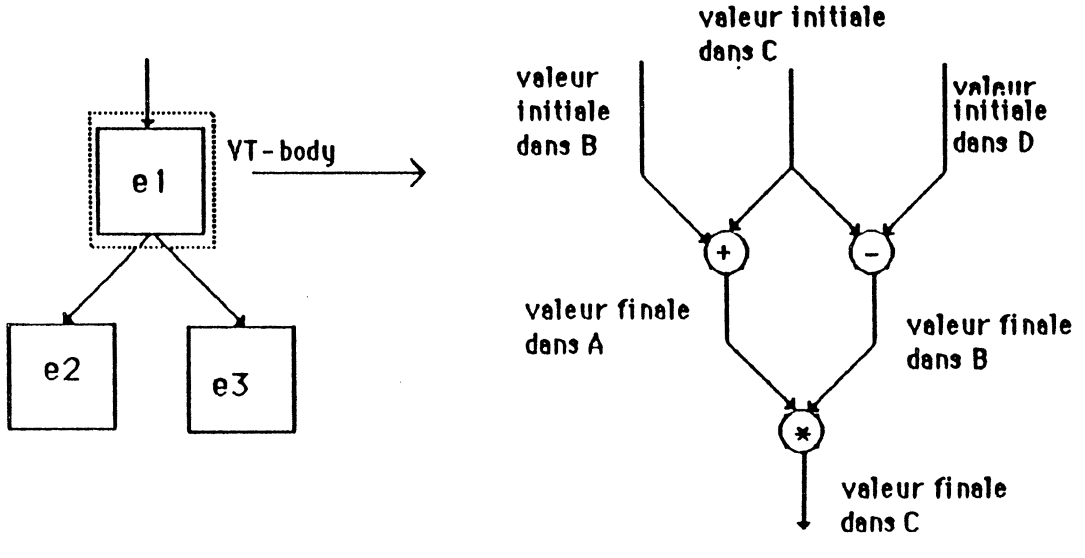


fig. 3.3 : graphe de flot de données et de contrôle utilisé dans le CMU-DA

Sagnes et al [SAGN 86] utilisent le concept de GRAFCET pour décrire le comportement de la partie contrôle d'un circuit (voir chapitre1, section 3.2). Le comportement d'une machine d'états finis complexe est donc défini sous forme d'une hiérarchie de graphes de type GRAFCET. Chaque niveau de cette hiérarchie décrit le comportement d'une machine d'états finis plus simple que l'originale et qui est destinée à être implémentée dans un étage de la partie contrôle finale. Cette hiérarchie est obtenue de la façon suivante : l'étage de contrôle de plus haut niveau est d'abord décrit par un graphe unique. Les noeuds de ce graphe sont ensuite décrits à un niveau plus fin par d'autres graphes qui constitueront le niveau inférieur de la hiérarchie et ainsi de suite jusqu'au niveau le plus bas qui actionne la partie opérative. Ces graphes sont ensuite transformés en tables de vérité pour être implémentés sous forme de PLAs ou de ROMs. Dans [SAGN 86], Sagnes et al montrent l'efficacité d'une telle partition hiérarchique de la partie contrôle car l'optimisation de la partie contrôle est grandement simplifiée.

Les systèmes ELF [GIRC 84] et MIMOLA [ZIMM 80] génèrent un seul graphe qui décrit en même temps le flot de données et le flot de contrôle. Les noeuds de ce graphe sont de deux types : ceux qui représentent les opérations sur les données et ceux qui représentent les opérations de contrôle; les noeuds de contrôle sont utilisés pour marquer le début et la fin des blocs conditionnels tels que les boucles et les instructions conditionnelles. Les arcs représentent le flot d'information qui circule d'un noeud vers l'autre.

exemple :

Soit la description algorithmique suivante d'une instruction d'un microprocesseur hypothétique; cette description est écrite dans le langage de description de circuits (qui ressemble au langage PASCAL) utilisé par le système MIMOLA :

```
routine meaningless ()
 begin
    a = 1000;
    b = 2;
    n = 10;
    while (n>0)
      begin
            a = a + 10;
            temp = b + n;
            temp = temp - 3;
            a = a - temp;
            n = n - 1;
       end; /* while */
  end. /* routine */
```

Le graphe représentant à la fois le flot de données et de contrôle ("control/data flow graph" en anglais) correspondant à cette procédure est représenté par la figure suivante (les noeuds "SW" marquent la fin de la boucle):

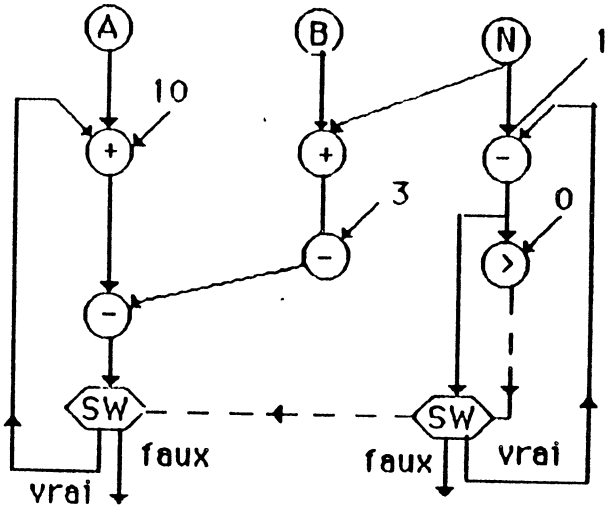


fig. 3.4 : graphe de flot de contrôle et de données

2.1. Théorie des automates et réduction d'états (approche de Glushkov)

L'approche de Glushkov [GLUS 65] est que tout algorithme peut être implanté sous forme de deux machines d'états finis interactives : une partie opérative (notée PO), et une partie contrôle (notée PC) décrites par des tables de transitions d'états [DAVI 80]. La PO reçoit de la PC des signaux imposant l'exécution d'actions telles que des transferts de registres. A son tour, la PO informe la PC des résultats des tests. Par conséquent, les signaux d'entrée de la PC correspondent aux signaux de sortie de la PO et les signaux de sortie de la PC aux signaux d'entrée de la PO (fig.3.5).
La partie opérative est généralement un automate de Moore (les sorties ne dépendent que de l'état courant). La partie contrôle qui travaille en collaboration avec la partie opérative peut être un automate d'états fini de Moore ou de Mealy.

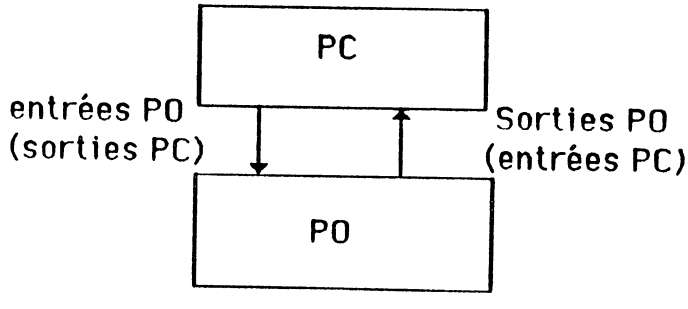


fig. 3.5 : implantation d'un algorithme sous forme de 2 automates

De ce fait, la réduction du nombre d'états de la PC devient possible en utilisant les techniques classiques d'optimisation d'automates d'états finis. Supposons que la PC est un automate de Mealy. Le problème est donc de réduire le nombre d'états de cet automate. Pour un automate complètement spécifié la première étape serait de minimiser le nombre d'états en les partitionnant en classes d'équivalence [BARA 79]. Baranov [BARA 79] définit 2 états équivalents de la manière suivante :

Définition : Deux états a_m et a_s appartiennent à une même classe d'équivalence si l'application d'une même séquence d'entrées produira toujours l'apparition d'une même séquence de sorties, que l'automate soit dans a_m ou dans a_s, et ceci pour toute séquence d'entrée.

Par conséquent, l'un de ces deux états, par exemple a_m, peut être supprimé de l'automate. Si une ou plusieurs classes d'équivalence contiennent plus d'un élément, on est en droit d'exclure de l'automate tous les éléments sauf un par classe : on obtient ainsi un automate ayant un nombre minimal d'états.

Définition : Deux états a_m, a_s sont k-équivalents si pour toute séquence d'entrées de longueur k appliquée à ces deux états, on obtient la même séquence de sortie.

Dans [BARA 79] une méthode de recherche de la partition de l'ensemble d'états d'un automate complétement spécifié en classes d'équivalences est proposée. Les auteurs de cette méthode sont D.D.Aufenkamp et F.E.Hohn :

L'algorithme de recherche de la partition P de l'ensemble des états est le suivant. On recherche par itérations successives les partitions P1, P2, ..., Pk, Pk+1 de l'ensemble des états de l'automate en classes d'états 1, 2, ..., k+1 équivalents entre eux, jusqu'à ce que, à une k+1 iéme itération, l'on constate que Pk+1 = Pk. Les auteurs de l'algorithme montrent très facilement qu'à cette itération la partition Pk = P, c'est à dire que les états k-équivalents sont équivalents, et que le nombre d'itérations k conduisant à Pk = P n'est jamais supérieur à M-1, où M est nombre d'états de l'automate.

2.2. Application de l'algorithme sur un exemple

L'approche de Glushkov est illustrée sur un exemple (donné dans [BARA 79]) pour montrer l'effort impliqué en utilisant cette approche pour minimiser le nombre d'états de la partie contrôle. Pour des raisons pratiques la partie contrôle est définie par deux tables : une table de transitions proprement dite (fig. 3.6a) et une table des sorties (fig. 3.6.b). Chaque sortie "si" représente un ensemble de microopérations de la partie opérative qui doivent être exécutées en parallèle.

état / entrées	a1	a2	a3	a4	a5	a6
z1	a3	a4	a3	a4	a5	a6
z2	a5	a6	a5	a6	a1	a2

fig. 3.6a : table des transitions d'états

état / entrées	a1	a2	a3	a4	a5	a6
z1	s1	s1	s1	s1	s1	s1
z2	s1	s1	s2	s2	s1	s1

fig. 3.6b : table des sorties

fig. 3.6 : tables décrivant le comportement d'une partie contrôle

D'aprés la table des sorties (fig. 3.6b) nous obtenons immédiatement la partition P1 en classe d'états 1-équivalents en regroupant les états qui ont des colonnes identiques:

$$P1 = [B1, B2] = \overline{1.2.5.6},\ \overline{3.4.}$$

En effet, deux états sont 1-équivalents si, placé dans ces états et attaqué par un mot d'entrée de longueur 1 de son alphabet, l'automate produit la même sortie quelque soit le mot d'entrée.

Construisons la table de la partition P1 en remplaçant les états de la table représentée dans la figure 3.6a par les classes de 1-équivalence correspondantes :

	B1	B2
	a1 a2 a5 a6	a3 a4
z1	B2 B2 B1 B1	B2 B2
z2	B1 B1 B1 B1	B1 B1

fig. 3.7 : partition P1 des états de l'automate

Il est évident que deux états 1-quivalents a_m et a_s sont 2-équivalents s'ils sont 1-équivalents pour tout signal d'entrée comme précédemment. D'aprés la table ci-dessus (fig.3.7), nous obtenons la partition P2 en classes d'états 2-équivalents :

$P2 = [C1, C2, C3] = \overline{1.2}, \overline{5.6}, \overline{3.4}.$

	C1	C2	C3
	a1 a2	a5 a6	a3 a4
z1	C3 C3	C2 C2	C3 C3
z2	C2 C2	C1 C1	C2 C2

fig. 3.8 : partition P2 des états de l'automate

La partition P3 se construit de la même façon :

$P3 = [D1, D2, D3] = \overline{1.2}, \overline{5.6}, \overline{3.4};$

Elle se confond avec P2. La procédure est terminée. La partition P3 = P2 = P est la partition de l'ensemble des états de l'automate de Mealy spécifié par la table initiale (fig. 3.6a).

Reconstitution de l'automate minimal :

La procédure de reconstitution de l'automate minimal équivalent à l'automate donné d'aprés sa partition P est la suivante [BARA 79] :

1- Choisir dans chaque classe d'équivalence de la partition P, un élément; former avec ces éléments l'ensemble E' des états de l'automate minimal.

2- Définir les fonctions de transitions et de sortie de l'automate minimal : A cet effet, supprimer dans les tables de transitions et des sorties les colonnes qui correspondent aux états non compris dans E' et remplacer tous les états dans les colonnes restantes de la table de transitions par leurs équivalents de l'ensemble E'.

3- Comme état initial choisir dans E' un état équivalent à l'état initial de l'automate optimisé.

Pour l'exemple traité ci-dessus, l'automate minimal est décrit par les tables suivantes :

entrée \ état	a1	a4	a5
z1	a4	a4	a5
z2	a5	a5	a1

fig. 3.9 : table de transitions de l'automate minimal

entrée \ état	a1	a4	a5
z1	s1	s1	s1
z2	s1	s2	s1

fig. 3.10 : table des sorties de l'automate minimal

La partie contrôle considérée dans cet exemple est très simple et par conséquent la procédure pour obtenir la partie contrôle minimale est très rapide. Malheureusement, la partie contrôle d'un circuit complexe peut contenir des milliers d'états. De plus, l'algorithme décrit ici est celui d'un automate de Mealy complètement spécifié, or dans la pratique la partie contrôle peut être incomplètement spécifiée (pour certaines combinaisons d'entrées et d'états les sorties sont indéfinies). Par conséquent, la théorie des automates d'états finis ne peut être utilisée que pour les machines d'états finis ne comportant qu'un petit nombre d'états et pour lesquelles nous pouvons utiliser des techniques d'énumération, c'est à dire décrire le comportement de telles machines en considérant de manière explicite tous les états possibles. Elle peut aussi être utilisée pour la synthèse de petits systèmes ou pour des optimisations locales.
Pour les machines d'états finis comportant un très grand nombre d'états, il est donc préférable de décrire leur comportement au moyen d'une méthode algorithmique.

3. Réduction du nombre d'états de l'algorithme d'interprétation dans SYCO

L'étape d'optimisation de haut niveau vient après la génération des tables de transitions décrivant le comportement de chaque tranche de contrôle; elle consiste essentiellement à appliquer certaines transformations sur ces tables de transitions pour réduire le nombre d'états et de transitions de l'algorithme d'entrée. Parmi ces optimisations certaines sont classiques puisque utilisées depuis longtemps dans les compilateurs optimiseurs de langages de programmation évolués [AHO 77], [HARB 82], [AUSL 82]. Ces transformations doivent préserver le comportement du système. Autrement dit, une optimisation ne doit pas modifier la sortie produite par l'algorithme pour une certaine entrée, ou provoquer une erreur, telle que la division par zéro. [AHO 77] adopte l'approche qui consiste à rejeter toute transformation complexe qui risque de changer l'algorithme original, et de ne considérer que des transformations simples qui garantissent un bon résultat en un temps de calcul relativement court. C'est l'approche que nous avons adopté dans le compilateur SYCO.

Pour réduire le nombre d'états de l'algorithme d'interprétation des instructions, les transformations suivantes sont appliquées :

1) Elimination des procédures inaccessibles dans l'arbre d'appel de la partie contrôle.
2) Compactage des instructions de contrôle
3) Compactage des instructions opératives.

3.1. Elimination des procédures inaccessibles

La première chose à faire dans l'optimisation de la partie contrôle est l'élimination des procédures inaccessibles, c'est à dire les procédures non référencées dans la description comportementale. Pour cela, on recherche dans l'arbre d'appels des procédures tous les chemins qui mènent de la racine de cet arbre (étage de contrôle le plus haut) aux feuilles (étage de contrôle le plus bas). Les sommets qui ne font partie d'aucun de ces chemins sont éliminés : ces sommets correspondent aux procédures inaccessibles.

exemple :

La figure suivante montre un exemple d'optimisation de l'arbre d'appel des procédures :

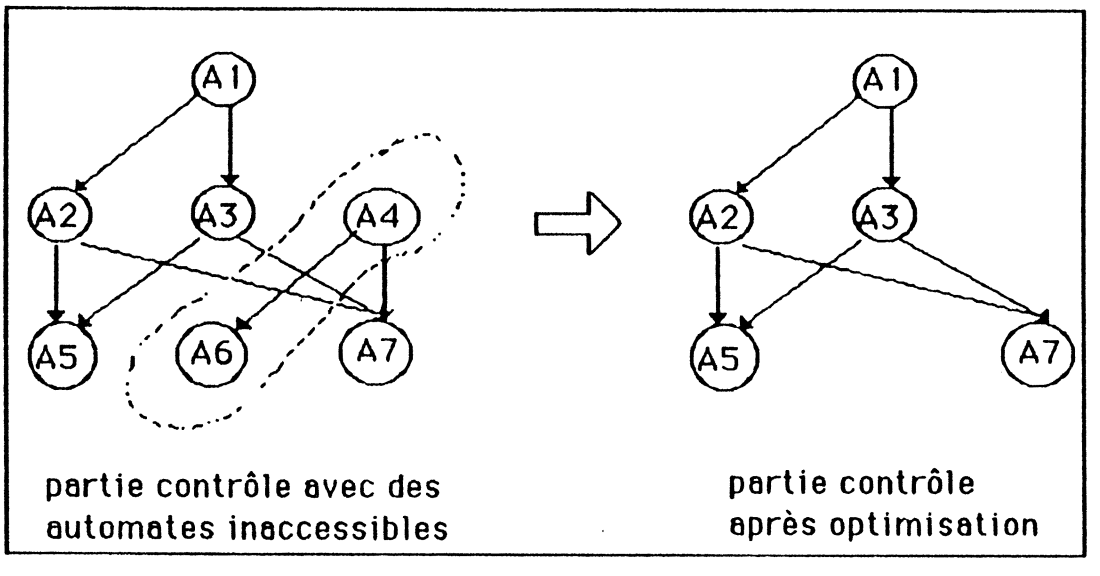


fig. 3.11 : optimisation de l'arbre d'appel des automates (CMODULES)
(Ai = automate, A1= automate principal)

Dans la figure ci-dessus, les automates A4 et A6 sont inaccessibles puisqu'il n'existe aucun chemin qui mène de A1 vers A4 ou A6. Par conséquent, ils peuvent être

enlevés de l'arbre d'appel de la partie contrôle sans changer l'algorithme de contrôle.

Remarque : Les procédures inaccessibles peuvent aussi résulter de l'application de certaines transformations de la description comportementale pendant la recherche d'un compromis surface-vitesse (voir chapitre4). L'élimination des procédures inaccessibles permet de réduire la surface de la partie contrôle et éventuellement de la partie opérative.

3.2. Réduction du nombre d'instructions dans une procédure

A chaque étage de contrôle correspond une table de transitions. Cette dernière est en fait composée de plusieurs sous-tables de transitions. Chaque sous-table représente une procédure interprétée par l'étage de contrôle. Les procédures d'un même étage de contrôle sont indépendantes puisqu'une procédure ne peut activer une autre de même niveau. Pour optimiser un étage de contrôle, il faut donc analyser chaque procédure interprétée par ce dernier pour essayer de réduire son nombre d'instructions.

Les étapes prises en compte pour réduire le nombre d'instructions dans une procédure sont les suivantes :

-élimination des instructions inaccessibles,
-élimination des instructions de branchements inutiles,
-fusion des instructions compatibles,
-élimination des instructions redondantes (ou équivalentes).

3.2.1. Elimination des instructions inaccessibles

Définition : Une instruction est dite inaccessible si elle n'est référencée par aucune autre, ou si elle n'est référencée que par des instructions elles-même inaccessibles.

D'après cette définition, on déduit l'algorithme suivant qui permet de détecter les instructions inaccessibles :
1- Marquer "inaccessible" les instructions qui ne sont pas référencées.
2- Réitérer l'étape 3 jusqu'à ce qu'il n'existe plus d'instructions inaccessibles.
3- Marquer "inaccessible" les instructions non encore marquées, et qui sont référencées par des instructions inaccessibles.

3.2.2. Elimination des instructions inutiles

Définition : une instruction est dite "inutile" si elle est inconditionnelle et si aucune action n'est générée au moment de l'exécution de cette instruction. On entend par action une activation de l'étage de contrôle inférieur ou une affectation de variables

de contrôle ou une affectation de registres de la partie opérative.

Cette optimisation consiste à éliminer les instructions inconditionnelles qui ne comportent qu'un ordre de branchement NEXT (ou EXIT si c'est la dernière instruction exécutée). Chaque fois qu'une instruction "NEXT ei" est éliminée, toutes les références à cette instruction sont ensuite remplacées par des branchements à l'étiquette ei.

exemple :

Soit la séquence d'instructions LDS suivante:

<e1>: EXECUTE ek; NEXT e2;
<e2>: NEXT e3;

L'instruction e2 est inutile; on peut la supprimer et remplacer dans la description toutes les instructions de branchement vers e2 par des branchements directs à e3. On obtient :

<e1>: EXECUTE ek; NEXT e3;

3.2.3. Compactage des instructions de contrôle

Le but de cette étape est de combiner les instructions de contrôle en le plus petit nombre possible afin de réduire la taille et le temps d'exécution de l'algorithme d'interprétation des instructions du microprocesseur.
L'architecture d'un étage de contrôle, permet d'effectuer en parallèle les actions suivantes :

-générer les commandes pour l'étage inférieur : activation d'un seul automate de l'étage inférieur par l'instruction EXECUTE.
-modifier le contenu des fils de contrôle : affectation d'une ou plusieurs variables de contrôle par l'instruction SET ou RESET.
-modifier le contenu des fils de séquencement interne ; branchement à l'état suivant par l'instruction NEXT.

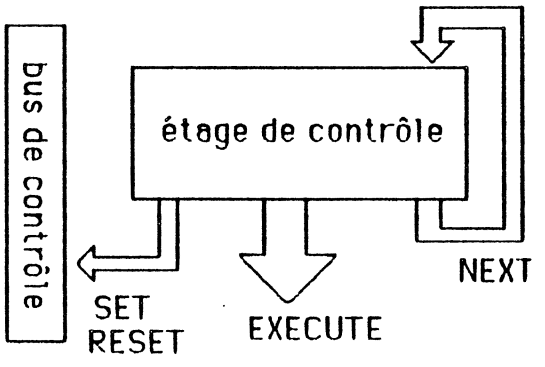


fig. 3.12 : commandes générées en parallèle

Ce parallélisme permis par l'architecture d'un étage de contrôle, est exploité au niveau de l'étape d'optimisation pour réduire le nombre d'états de l'algorithme.
Les instructions de contrôle peuvent donc être constituées à la fois d'un appel de CMODULE : EXECUTE, de plusieurs affectations de variables de contrôle : SET et RESET et d'un ordre de branchement : NEXT ou EXIT. Le but est donc de fusionner certaines instructions de contrôle de telle façon que chaque instruction comporte au moins une affectation de variables de contrôle ou un ordre EXECUTE.

3.2.3.1. Contraintes de compatibilité de 2 instructions de contrôle

Définition : deux instructions ei et ej appartenant à une même procédure sont dites compatibles si :
-ej est exécutée directement (ou sous une certaine condition) après ei,
-ei et ej peuvent être exécutées en parallèle sans modifier le comportement du système.

Avant de donner les règles de fusion (ou compactage) de deux instructions de contrôle, considérons d'abord les couples d'instructions simples incompatibles :

1) Une instruction de type EXECUTE ne peut être exécutée simultanément avec une autre instruction EXECUTE. En effet, l'architecture actuelle d'un étage de contrôle ne permet pas d'activer simultanément plusieurs automates d'un même étage de contrôle.

2) L'instruction "RESET var1" n'est pas compatible avec l'instruction "SET var1".

Cette condition permet d'éviter la mise à l'état indéfini d'une variable de contrôle au cours de la fusion de deux instructions.

exemple :

<ei> : RESET adbus ; NEXT ej;
<ej>: SET adbus ; NEXT ek;

ne peut être transformé en :

<ei> : RESET adbus ; SET adbus ; NEXT ek;

3) Une instruction "EXECUTE Pi" n'est pas compatible avec une instruction "RESET var1" (resp. "SET var1") si les conditions suivantes sont vérifiées :
- "RESET var1" (resp. "SET var1") est exécutée après "EXECUTE Pi"
- la procédure Pi ou un de ses descendants (dans l'arbre d'appel des procédures), contient l'instruction "SET var1" (resp. "RESET var1") ou un test utilisant la variable de contrôle var1.

exemple1 :

<ek> : SET dtbus ; NEXT ei;
<ei> : EXECUTE etiqn ; NEXT ej;
<ej> : RESET dtbus ; NEXT el;
.....
<etiqn> : IF dtbus = 1 EXECUTE etiqk;
.....

Si on fusionne ei et ej, on obtient :

<ei> : EXECUTE etiqn ; RESET dtbus ; NEXT el;

Dans ce cas, le test "dtbus = 1" devient faux (on suppose que RESET met à zéro une variable de contrôle) puisque à l'entrée de la procédure "etiqn" la variable dtbus aura la valeur 0, or avant la fusion elle avait la valeur 1. Le comportement du système est modifié. Par conséquent, ei et ej ne peuvent être fusionnés.

Remarque :
Dans cet exemple, ek et ei peuvent être fusionnés puique l'instruction SET est exécutée avant l'instruction EXECUTE. Cette possibilité est due au fait que l'affectation des variables de contrôle est "instantanée". Le comportement du système est donc bien préservé puisque la procédure "etiqn" ainsi que tous ses descendants dans l'arbre d'appel des procédures utiliseront la nouvelle valeur de "dtbus" (c.a.d la valeur 1).

exemple 2 :

<ei> : EXECUTE etiqn ; NEXT ej;
<cj> : RESET dtbus ; NEXT el;
.....
<etiqn> : SET dtbus;
.....

Dans ce cas aussi, la fusion de ei et ej est impossible : si on fusionne ei et ej la variable de contrôle dtbus aura la valeur 1 (affectée par la procédure etiqn) à l'entrée de l'état el; or, avant la fusion dtbus avait la valeur 0 (affectée par ej).

Nous présentons dans ce qui suit les différents cas possibles de couples d'instructions de contrôle qui peuvent être fusionnées ainsi que les conditions nécessaires et suffisantes que doivent respecter ces instructions pour préserver le comportement du système.

3.2.3.2. Fusion de 2 instructions inconditionnelles

Dans ce cas, il faut que l'instruction résultant de la fusion ne contienne pas d'instructions incompatibles.

exemple :

<sta-1> : read-write 0:1 := 1 ; EXECUTE Niv-0-sta-1 ; NEXT sta-3;

<sta-3> : adstrobe 0:1 := 1 ; dtstrobe 0:1 := 1 ; NEXT sta-5;

Ces 2 instructions peuvent être fusionnées en une seule instruction dont la syntaxe est la suivante :

<sta-1> : read-write 0:1 := 0 ; adstrobe 0:1 := 1 ; dtstrobe 0:1 := 1;
EXECUTE Niv-0-sta-1 ; Next sta-5;

Si l'instruction "sta-3" n'est référencée par aucune autre, elle est enlevée de la table de transitions de l'automate en cours d'optimisation. Dans ce cas, la fusion de "sta-1" et "sta-3" implique une augmentation de la vitesse du circuit (gain de 1 cycle d'horloge) et une réduction de la surface de la partie contrôle (gain de 1 monôme dans le PLA implémentant la table de transitions à optimiser).
Si sta-3 est référencée par d'autres instructions, elle est conservée dans la table de transitions; dans ce cas, la fusion n'entraîne que le gain de vitesse puisque le nombre d'états reste inchangé.

3.2.3.3 Fusion d'une instruction conditionnelle et d'une instruction inconditionnelle

Deux cas sont à distinguer :

1) Cas où l'instruction inconditionnelle est exécutée avant l'instruction conditionnelle :

Cas général :

<ei> : instruction inconditionnelle A ; NEXT ej;

<ej> : IF $cond_1$ action A_1 ; NEXT e1;
IF $cond_2$ action A_2 ; NEXT e2;
.....
IF $cond_n$ action A_n ; NEXT en;

Cette séquence est transformée en la séquence équivalente :

<ej> : IF $cond_1$ action A_1 ; A ; NEXT e1;
IF $cond_2$ action A_2 ; A ; NEXT e2;
.....
IF $cond_n$ action A_n ; A ; NEXT en;

si les conditions suivantes sont vérifiées :

-toutes les actions A_i (i= 1,...,n) sont compatibles avec l'instruction inconditionnelle A,
-l'instruction inconditionnelle A ne modifie pas (directement ou indirectement) un des tests "$cond_i$" (i = 1,...,n).

Après la fusion, l'instruction inconditionnelle est éliminée si elle n'est pas référencée par d'autres instructions.

exemple 1 :
<fetch-send-1> : adbus := pc ; SET adstrobe ; NEXT fetch-receive;

<fetch-receive> : IF dtack=0 adbus := pc ; NEXT fetch-receive;

IF dtack <> 0 pc := pc + 1 ; RESET adstrobe ; EXIT;

La fusion de ces 2 instructions est impossible; en effet, "SET adstrobe" et "RESET

adstrobe" étant incompatibles, si la condition "dtack <> 0" est vraie, ceci entrainerait la mise à l'état indéfini de la variable de contrôle "adstrobe".

exemple 2 :
<fetch-send-1> : adbus := pc ; SET adstrobe ; NEXT fetch-receive;

<fetch-receive> : IF dtack=0 adbus := pc ; NEXT fetch-receive;
IF dtack <> 0 pc := pc + 1 ; EXIT;

Toutes les actions dans "fetch-send-1" sont compatibles avec celles dans "fetch-receive". Ces deux instructions peuvent être fusionnées; on obtient la nouvelle instruction :

<fetch-receive> : IF dtack=0 adbus := pc; SET adstrobe; NEXT fetch-receive;
IF dtack <> 0 pc := pc + 1; adbus := pc; SET adstrobe; EXIT;

Si l'instruction "fetch-send-1" n'est référencée par aucune autre, elle est éliminée de la table de transitions en cours d'optimisation.

exemple 3 :
<fetch-send-1> : dtack := dtack + 1 ; NEXT fetch-receive;

<fetch-receive> : IF dtack = 0 adbus := pc ; NEXT fetch-receive;
IF dtack <> 0 pc := pc + 1 ; EXIT;

La fusion de ces 2 instructions est impossible puisque l'instruction "fetch-send-1" modifie le test de la variable "dtack".

2) Cas où l'instruction inconditionnelle est exécutée après l'instruction conditionnelle :

Dans ce cas, il suffit que l'instruction inconditionnelle (notée ei) soit compatible seulement avec les actions contenues dans la (ou les) alternative(s) de l'instruction conditionnelle dont l'état suivant est ei (c.a.d qui contiennent un "NEXT ei").

Cas général :
<ej> : IF $cond_1$ action A_1 ; NEXT e1;

.....

IF $cond_i$ action A_i ; NEXT ei;

.....

IF $cond_n$ action A_n ; NEXT en;

<ei> : instruction inconditionnelle A ; NEXT ek;

Si l'action A_i est compatible avec l'action A, <ei> et <ej> peuvent être fusionnées en une seule instruction suivante :

<ej> : IF $cond_1$ action A_1 ; NEXT e1;

.....

IF $cond_i$ action A_i ; A ; NEXT ek;

.....

IF $cond_n$ action A_n ; NEXT en;

exemple :

<e1> : IF dtack = 0 SET adstrobe ; NEXT e2;
IF dtack <> 0 EXECUTE P2 ; NEXT e3;

<e2> : SET read-write; NEXT e4;
<e3> : EXECUTE P3; NEXT e5;

-La fusion de e1 et e2 est possible puisque "SET read-write" peut être exécuté en parallèle avec "SET adstrobe".
-La fusion de e1 et e3 est impossible puisque "EXECUTE P2" est incompatible avec"EXECUTE P3".

Après la fusion de e1 et e2, on obtient :

<c1> : IF dtack = 0 SET adstrobe; SET read-write; NEXT e4;
IF dtack <> 0 EXECUTE P2; NEXT e3;

3.2.3.4. Fusion de 2 instructions conditionnelles

Cas général :
<ej>: IF $cond_1$ action A_1 ; NEXT e1;

.....

IF $cond_i$ action A_i ; NEXT ei;

.....

IF $cond_n$ action A_n ; NEXT en;

<ei> : IF $cond'_1$ action A'_1; NEXT e'1;
IF $cond'_2$ action A'_2; NEXT e'2;

.....

IF $cond'_m$ action A'_m; NEXT e'm;

Si les conditions suivantes sont vérifiées :
- l'action A_i ne modifie aucun test $cond'_k$ (k= 1,...,m).
- l'action A_i est compatible avec toutes les actions A'_k (k = 1,...,m).

alors, ei et ej peuvent être fusionnées; on obtient :

<ej> : IF $cond_1$ action A_1 ; NEXT e1;

.....

IF ($cond_i$ AND $cond'_1$) action A_i ; action A'_1; NEXT e'1;
IF ($cond_i$ AND $cond'_2$) action A_i ; action A'_2; NEXT e'2;

.....

IF ($cond_i$ AND $cond'_m$) action A_i ; action A'_m; NEXT e'm;

.....

IF $cond_n$ action A_n; NEXT en;

Cette transformation peut être classée dans la rubrique "recherche d'un compromis surface-vitesse" ou "optimisation" selon que l'instruction ei est référencée par d'autres instructions ou non (ou par d'autres alternatives de ej). Dans le premier cas, la fusion de ei avec ej provoque une augmentation de la vitesse du circuit mais aussi de sa surface; en effet, l'instruction ei est dupliquée dans la table de transitions à optimiser, ce qui provoque une augmentation du nombre de monômes du PLA implémentant cette table.
Par contre, si ei n'est pas référencée par d'autres instructions, après sa fusion avec ej elle est éliminée de la table de transitions; dans ce cas la fusion de ei et ej implique une réduction de la surface du circuit et une augmentation de sa vitesse d'exécution.

3.2.4. Compactage des instructions opératives

Dans le cas général, cette transformation consiste à détecter les actions opératives qui peuvent être exécutées en parallèle pour les combiner en une séquence de microinstructions proche de l'optimale [TOKO 81]. Yau et al [YAU 74] montre que la production d'une séquence optimale de microinstructions à partir d'une séquence de microopérations est un problème NP-complet.
Avant de présenter le compactage des instructions opératives dans le compilateur SYCO, nous étudions dans ce paragraphe les différents types de compactage.
On distingue 2 types de compactage :
-le compactage global
-le compactage local.

3.2.4.1. Le compactage global

Le compactage global optimise la description dans son ensemble. La description comportementale est d'abord divisée en blocs de base. Un bloc de base est défini

comme une séquence linéaire d'instructions n'ayant qu'un seul point d'entrée (la première instruction exécutée) et qu'un seul point de sortie (la dernière exécutée). Des techniques de compactage local sont ensuite appliquées à chacun de ces blocs [TOKO 81]. Les blocs successeurs et prédécesseurs de chaque bloc de base doivent être aussi déterminés afin de pouvoir effectuer une optimisation globale. Deux blocs connectés directement par un arc, quand ils sont considérés indépendamment des autres blocs, peuvent être traités comme un seul bloc et des techniques d'optimisation locale peuvent être appliquées entre ces 2 blocs. Autrement dit, une microopération peut être transférée d'un bloc vers son successeur ou d'un bloc vers son prédécesseur. Dans la figure ci-dessous, la microopération A dans le bloc S1 peut être exécutée en parallèle avec la microopération B dans le bloc S2 : ces deux microopérations peuvent être compactées soit dans S1, soit dans S2.

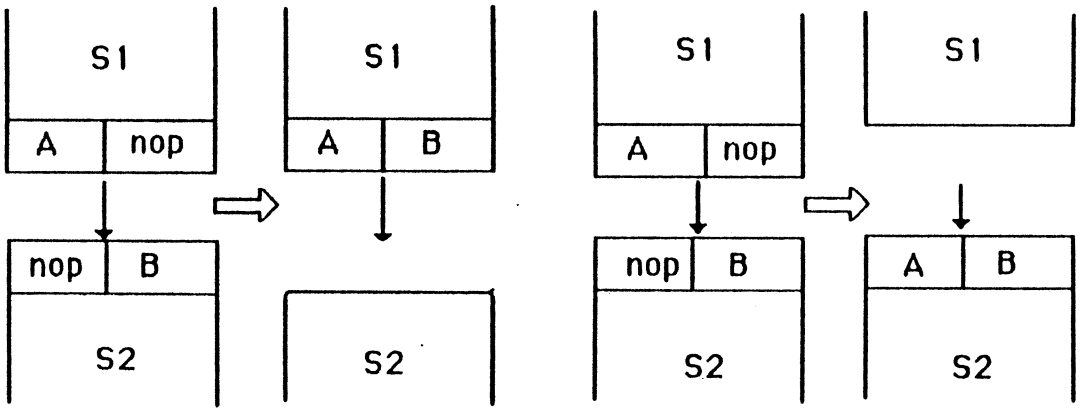


fig. 3.13 : transferts de microopérations entre blocs adjacents

Si plusieurs blocs prédécesseurs ou successeurs existent, des microopérations peuvent aussi être transférées entre ces blocs. La figure ci-dessous montre un tel cas:

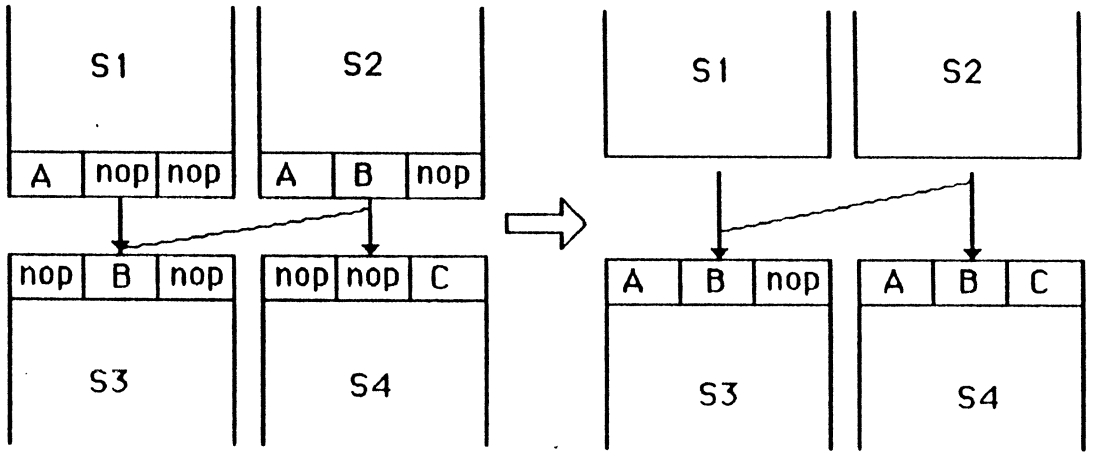


fig. 3.14 : transferts de microopérations entre blocs adjacents avec plusieurs successeurs.

3.2.4.2. Le compactage local

Le compactage local optimise la description à l'intérieur d'un bloc de base. Peu de méthodes concernant l'optimisation globale de microprogrammes ont étés définies. Par contre, de nombreuses techniques de compactage local de microcode ont été recensées [DAVI 81], [AGER 76], [FISH 81], [ISOD 83], [TOKO 81]. Ces techniques sont actuellement applicables pour les circuits microprogrammés ou non. Davidson et al comparent 4 méthodes de compactage [DAVI 81]. Si certaines microinstructions contiennent déjà des microopérations en parallèle, il est nécessaire de décomposer ces microinstructions en microopérations afin d'obtenir le minimum absolu du nombre des microinstructions.

1) la méthode du premier venu, premier servi : on parcourt séquentiellement la liste des opérations et on place une opération dans la première instruction après celle où tous ses opérandes sont définis.

exemple :

Soit la séquence d'opérations suivante :

```
V3 := V1 + V2
V5 := V3 - V4
V7 := V3 * V6
V8 := V3 + V5
V9 := V1 + V7
V11 := V10 / V5
V12 := 100
V13 := V3
V12 := V1
V14 := V11 and V8
V15 := V12 or V9
V1 := V14
V2 := V15
```

Cette séquence est transformée en la séquence équivalente de microinstructions suivante :

```
V3 := V1 + V2  ;  (V12 := 100)  ;  V12 := V1
V5 := V3 - V4  ;  V7 := V3 * V6  ;  V13 := V3
V8 := V3 + V5  ;  V9 := V1 + V7  ;  V11 := V10 / V5
V14 := V11 and V8  ;  V15 := V12 or V9
V1 := V14  ;  V2 := V15
```

On remarque que l'instruction "V12 := 100" est une instruction redondante et peut

donc être éliminée.

2) la méthode du chemin critique : c'est la méthode inverse de la précédente. Il s'agit de déterminer le nombre maximum de microinstructions que l'on peut exécuter avant une microopération sans provoquer d'augmentation du nombre de microinstructions.

3) la méthode du "branch and bound" : on construit un arbre dont les nœuds sont les microinstructions. On part des microopérations qui n'ont pas de prédécesseur dans le graphe de précédence. Des branches sont créées chaque fois qu'il y a plus d'une microinstruction qui peut être constituée à ce niveau. Toutes les branches de l'arbre qui peuvent conduire à une solution optimale sont parcourues. Seul cet algorithme dont le temps d'exécution est d'ailleurs une fonction exponentielle du nombre de microopérations garantit l'obtention de la solution optimale.

4) la méthode "list scheduling" : seule la meilleure branche de l'arbre de la méthode branch and bound est formée à chaque niveau. La meilleure peut être par exemple celle dont les microopérations qui constituent la microinstruction ont le plus de descendants (directs ou indirects) dans le graphe de précédence.

3.2.4.3. Compactage des instructions opératives dans SYCO

Le compilateur SYCO permet au concepteur de spécifier le parallélisme entre les actions opératives. Après la génération de la partie opérative il peut être utile de fusionner entre elles deux ou plusieurs microinstructions afin d'exploiter au maximum les ressources de la partie opérative. Les microinstructions ne sont pas décomposer en microopérations pour être ensuite fusionnées (comme dans le cas du CMU-DA par exemple, qui utilise la méthode du premier venu premier servi), mais elles sont directement fusionnées telles que spécifiées par le concepteur. Ceci ne garantit pas une solution optimale mais permet d'optimiser la description comportementale de la partie opérative pendant un temps de calcul relativement court. En effet, les techniques de compactage local décrites précédemment sont efficaces [LAND 80] mais nécessitent malheureusement un temps de calcul qui est souvent considérable.

Cette étape de l'optimisation correspond à la réduction du nombre d'états du dernier étage de contrôle (étage de génération de commandes de la partie opérative). Seul cet étage peut activer directement la partie opérative; par conséquent, la table de transitions associée à cet étage donne la relation de précédence qui existe entre les instructions opératives.

3.2.4.3.1. Contraintes de compatibilité de 2 instructions opératives

Soient 2 instructions opératives 'ei' et 'ej' appartenant à une même procédure et telles que 'ej' est exécutée directement après ei. Pour que ces deux instructions puissent être fusionnées, tout en préservant le comportement du système, il faut qu'elles vérifient les contraintes suivantes :

1) contrainte de précédence : Il n'existe aucun registre source dans ei qui figure comme registre destination dans ej, et réciproquement.

exemple : "A := B + C" et "D := X + Y" peuvent être fusionnées si :
$A \neq X$ et $A \neq Y$
$B \neq D$ et $C \neq D$

une opération ne pourra donc être effectuée avant une opération qui définit un de ses opérandes.

2) contrainte matérielle : il faut que la nouvelle microinstruction (notée eij) résultant de la fusion de ei et ej, puisse être exécutée par la partie opérative générée par le compilateur Apollon.

3.2.4.3.2. Algorithme de compactage des instructions opératives

Les instructions opératives sont parcourues séquentiellement :

1) essayer de fusionner une instruction avec celle qui la suit si les contraintes de compatibilité sont satisfaites, sinon passer à l'instruction suivante et refaire le même test.

2) réitérer l'étape1 jusqu'à ce qu'il n'existe aucun couple d'instructions qui vérifient les contraintes de compatibilité.

3) éliminer les instructions qui ne sont plus référencées.

Si les fréquences d'exécution des instructions opératives sont spécifiées par le concepteur, les instructions fusionnées en premier sont celles dont la fréquence est maximale.

3.3. Autres techniques de réductions du nombre d'états

3.3.1. Déplacement des actions invariantes à l'extérieur des boucles

Une opération qui est invariable dans une boucle peut être déplacée à l'extérieur de cette boucle afin d'être exécutée moins de fois. Une opération est dite invariable si

elle produit toujours les mêmes valeurs à chaque passage dans la boucle.

exemple :

```
While ( i < n )                                    x := y + z ;
        x := y + z ;              ----->           While ( i < n)
        . .....                                      .....
        . ensemble d'instructions
        . qui ne modifient pas y et z
```

[AHO 77] donne un algorithme simple qui permet de détecter des opérations invariables à l'intérieur d'une boucle :

1) Marquer "invariable" toute opération dont les opérandes sont soit des constantes, soit définis à l'extérieur de la boucle.
2) Répéter l'étape 3 jusqu'à ce qu'il n'existe plus d'opérations invariables
3) Marquer "invariable" toutes les opérations non précédemment marquées, dont les opérandes sont définis une seule fois à l'intérieur de la boucle par une opération marquée "invariable".

exemple :

```
While  (i < n)                                     x := y + z ;
        x := y + z ;              ----->           v := x + w ;
        .....                                      While  (i < n)
        . instructions qui ne                        .....
        . modifient pas y,z et w.
        .....
        v := x + w ;
```

L'opération "y + z" est invariable puisque y et z sont définis à l'extérieur de la boucle. "x + w" est aussi invariable puisque w est défini à l'extérieur de la boucle et x est défini à l'intérieur de la boucle par l'opération invariable " x := y + z".

3.3.2. Transformation d'une instruction CASE en une suite d'instructions IFs

Cette transformation permet de ne mémoriser que 2n microinstructions au lieu de 2^n si n est le nombre d'alternatives dans l'instruction CASE.

exemple :

CASE e1 :
When (000) (b;d;f)
(001) (a;d;f)
(010) (b;c;f)
(011) (a;c;f)
(100) (b;d;e)
(101) (a;d;e)
(110) (b;c;e)
(111) (a;c;e)

est transformée en

if e1 a; else b; end
if e2 c; else d; end
if e3 c; else f; end

4. Optimisation du nombre de ressources de la partie opérative

4.1. Fusion des constantes (Constant folding)

La fusion des constantes est un exemple classique d'optimisation et une des plus simples. Comme le montre la figure suivante, cette action consiste à remplacer un opérateur dont les entrées sont des constantes par une nouvelle constante.

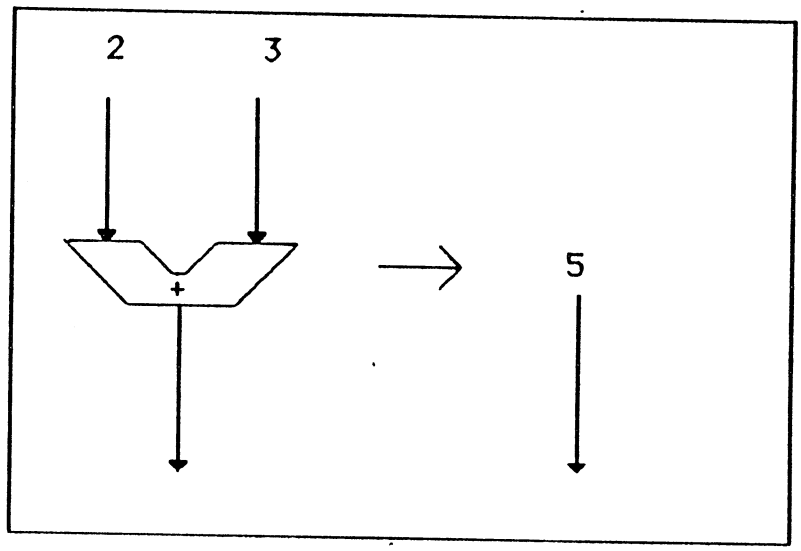


fig.3.15 : fusion des constantes 2 et 3

Cette transformation est toujours bénéfique et peut conduire à une élimination d'un opérateur dans la partie opérative. Un autre avantage possible de cette transformation est une réduction du nombre d'états de l'algorithme de contrôle :

exemple :

<e1> : C := 2 + 3 ; NEXT e2 ;
<e2> : IF C > 4 NEXT e4 ;

----->

<e1> : C := 5 NEXT e4 ;

Dans cet exemple, le test e2 est toujours vrai est peut donc être éliminé s'il n'est pas référencé par d'autres instructions que e1.

4.2. Propagation des constantes

Les références à une variable peuvent être remplacées par des références à la constante correspondante à cette variable, jusqu'à ce que la variable soit redéfinie [SCHE 77]. Cette transformation peut conduire à une réduction du nombre d'opérateurs de la partie opérative et du nombre d'états de l'algorithme.

exemple :

Soient les 2 instructions opératives suivantes :

<e1> : A := 1 ; C := H - P ; NEXT e2 ;

<e2> : G := A + 2 ; J := K - P ; EXIT ;

Si on ne modifient pas ces 2 instructions, le compilateur de parties opératives Apollon affectera 2 UALs à la partie opérative pour pouvoir implanter la deuxième instruction; par contre, si on propage la constante 1, l'opération "G <-- A + 2" dans "e2" devient "G <-- 1 + 2"; cette opération est ensuite remplacée par "G <-- 3" (fusion des constantes 1 et 2). Après ces deux transformations, il ne faut plus qu'une seule UAL pour implanter ces deux instructions.

La propagation des constantes permet aussi dans certains cas de réduire le nombre d'états de la partie contrôle :

exemple :

<e1> : A := 5 ; NEXT e2 ;
<e2> : IF A > 2 NEXT ei; -----> <e1> : A :=5 ; NEXT ei ;

Le test "A > 2" est toujours vrai et peut donc être enlevé de la description s'il n'est pas référencé par d'autres instructions que e1.

Remarque :
Le seul inconvénient de cette optimisation c'est qu'elle peut aboutir à une augmentation sensible du nombre de constantes.

4.3. Elimination des opérateurs redondants

L'élimination des opérateurs redondants est une transformation classique, similaire à l'élimination des sous-expressions redondantes dans les compilateurs optimiseurs

de langages de programmation évolués qui consiste à sauvegarder le résultat du calcul d'une expression complexe dans une variable temporaire pour des utilisations ultérieures.

Cette transformation peut être utilisée lorsque deux ou plusieurs opérateurs de même type ont les mêmes entrées; elle consiste à éliminer un des opérateurs et à remplacer toutes les références à ses sorties par des références aux sorties de l'opérateur retenu.

1) Cas où l'opérateur redondant appartient à une même instruction :

exemple :

C := A + B ; D := A + B

est transformée en :

(C , D) := A + B

Comme la "fusion des constantes", cette transformation est toujours bénéfique; elle conduit à une réduction du nombre d'opérateurs de la partie opérative (fig. 3.16) et éventuellement du nombre d'états de l'algorithme d'entrée du compilateur de silicium.

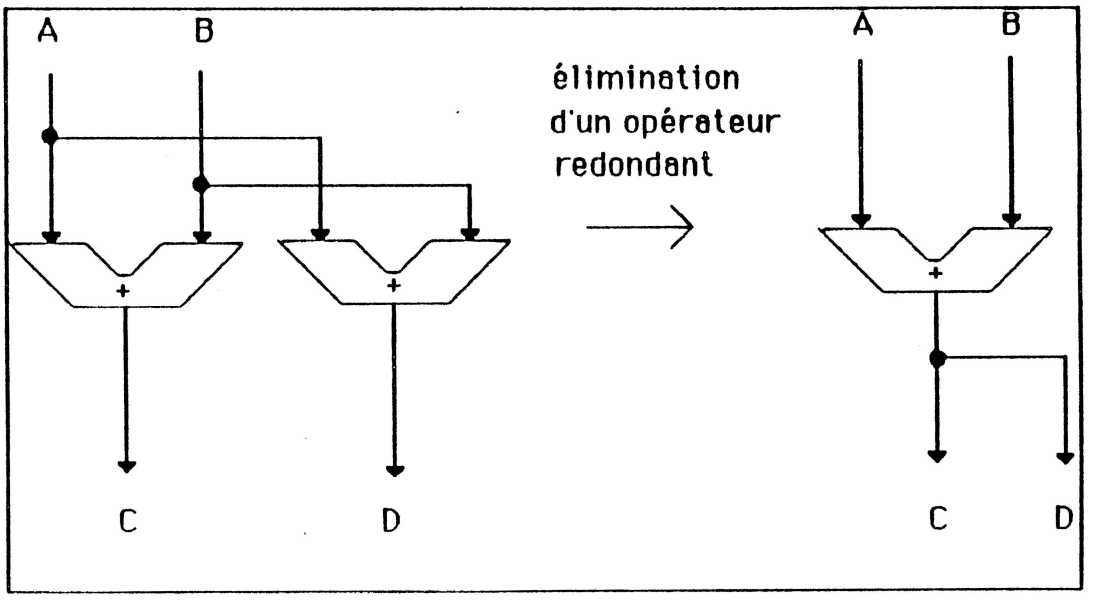


fig. 3.16 : élimination d'un opérateur redondant

2) Cas où l'opérateur redondant est commun à deux instructions :

exemple 1:

<e1> : A := B + C ; H := G - I ; NEXT e2 ;
<e2> : K := B + C ; U := NOT J ; L := G - I ;

La figure suivante donne une structure possible de partie opérative (avec 3 UALs) pouvant exécuter ces deux instructions :

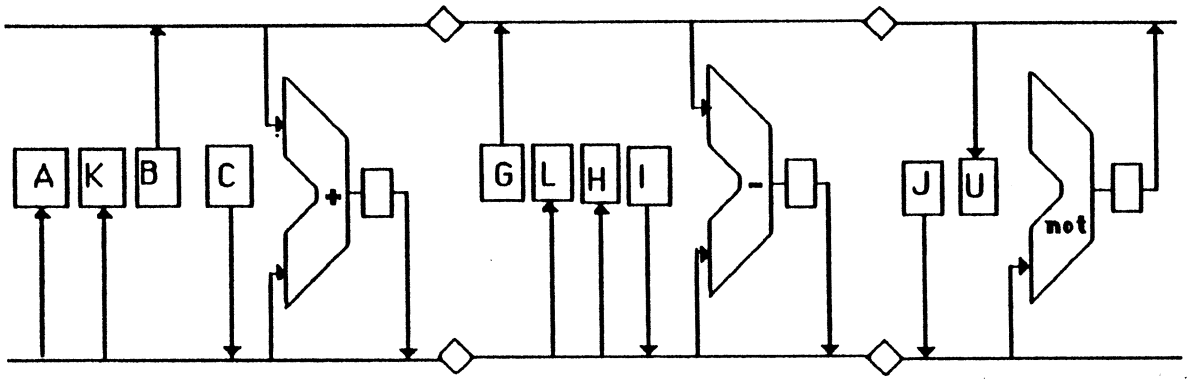


fig. 3.17 : partie opérative avant optimisation

Si dans <e2>, on remplace les expressions "B + C" et "G - I" respectivement par "A" et "H", alors il ne faut plus qu'une seule UAL pour implanter cette instruction. On obtient la nouvelle description :

<e1> : A := B + C ; H := G - I ; NEXT e2 ;
<e2> : K := A ; U := NOT j ; L := H

La figure suivante donne une structure de partie opérative (après optimisation), pouvant exécuter ces deux nouvelles instructions :

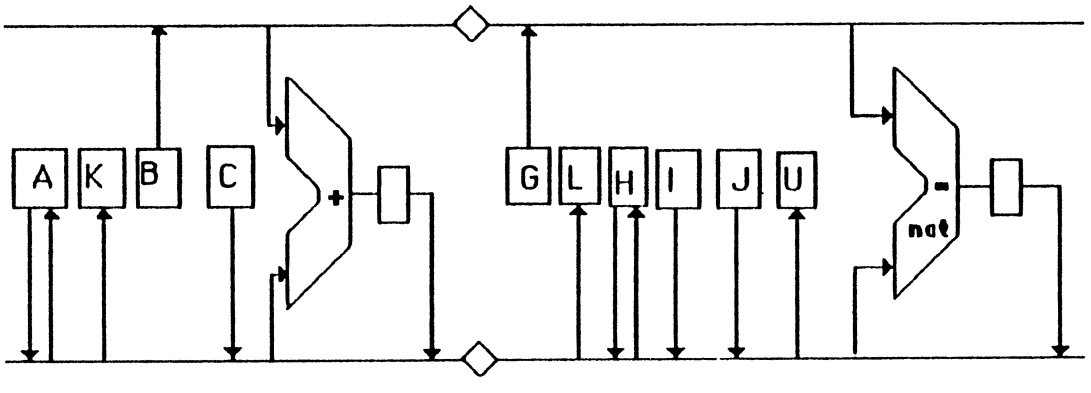


fig.3.18 : partie opérative après optimisation

exemple 2 :

<e1> : A := B + C ; NEXT e2;
<e2> : B := A - D ; NEXT e3;
<e3>: C := B + C ; NEXT e4;
<e4> : D := A - D ;

La deuxième et la quatrième instructions calculent la même expression, c'est à dire "B + C -D", et par conséquent cette séquence d'instructions peut être transformée en la séquence équivalente suivante :

<e1> : A := B + C ; NEXT e2;
<e2> : (B, D) := A - D ; NEXT e3;
<e3> : C := B + C ;

Remarquons que l'opération "B + C" apparait dans la première et la troisième instruction , mais la deuxième instruction <e2> redéfinit B. Par conséquent, la valeur de B dans <e3> est différente de celle de B dans <e1>; "B + C" n'est donc pas une opération redondante.

4.4. Mise sous forme canonique des opérations

Les opérandes des opérations commutatives sont réordonnés de façon à se ramener à un minimum d'opérations différentes et à mettre les opérandes toujours à la même place. Cette transformation permet d'appliquer d'autres optimisations telles que l'élimination des opérateurs redondants; dans certains cas, elle permet d'économiser des multiplexeurs [RAJA 85]. Actuellement dans le compilateur Apollon, la commutativité des opérations est prise en compte lors de la phase d'allocation.

4.5. Transformations des opérateurs binaires en opérateurs unaires

On remplace les opérations binaires faisant intervenir des constantes par des opérations unaires dans la mesure où l'on dispose d'opérateurs correspondants.

A+1	devient	incr A : incrémentation
A-1	devient	decr A : décrémentation
Ax2	devient	shfl A : décalage à gauche
A div 2	devient	shfr A : décalage à droite

Cette transformation permet par exemple d'utiliser un incrémenteur plutôt qu'un additionneur pour faire des incrémentations, ce qui est intéressant car un incrémenteur occupe une surface moins importante qu'un additionneur.

4.6. Optimisation du nombre de registres

Le problème est le suivant : Il est en général intéressant d'assigner plus d'une variable au même élément physique de mémorisation. Cette transformation permet de réduire en premier le nombre de registres de la partie opérative ainsi que le nombre d'autres composants. Soient A et B, 2 variables combinables :

-A et B ont une source commune. La fusion de A et B peut entraîner la suppression d'un démultiplexeur

-A et B ont une destination commune. La fusion de A et B peut entraîner la suppression d'un multiplexeur.

Peu de systèmes de synthèse décrits dans la littérature minimisent le nombre de registres de la partie opérative. Seul "Facet" [TSEN 83] donne un algorithme de minimisation du nombre des éléments de mémorisation de la partie opérative.

4.6.1. Conditions suffisantes pour combiner 2 variables

Etant donné un ensemble de variables, le problème est de combiner les variables qui peuvent partager un même élément de mémorisation. Pour spécifier les conditions suffisantes pour pouvoir combiner 2 variables, il est nécessaire de rappeler la définition de la durée de vie d'une variable.
- Une variable est dite "active" (ou "vivante") entre l'instant où elle est définie et l'instant où elle est utilisée pour la dernière fois.
- Une variable est dite "inactive" (ou "morte") entre l'instant de sa dernière utilisation et l'instant de sa nouvelle définition.

Deux variables peuvent être combinées si elles ne sont jamais actives en même temps.

4.6.2. Algorithme de compactage des variables

On construit le graphe de compatibilité de durée de vie des variables, où les nœuds sont les variables et les arcs relient 2 variables qui ne sont jamais actives en même temps. Les variables qui peuvent être combinées forment une clique :

Définition d'une clique :
Soit G un graphe non orienté constitué d'un ensemble fini de noeuds.
-un sous-ensemble C de noeuds de G forme un graphe complet si chaque noeud dans C est connecté à tous les autres noeuds de C.
-un graphe complet C est appelé une clique si C n'est pas contenu dans un autre graphe complet de G.

Dans [TSEN 83] Tseng et Siewiorek formalisent le problème de minimisation du nombre d'éléments de mémorisation en un problème de minimisation du nombre des cliques disjointes d'un graphe. La recherche des cliques d'un graphe est un problème NP-complet. [TSEN 83] décrit une procédure qui décompose un graphe en cliques disjointes en un temps qui est une fonction polynômiale du nombre de nœuds et d'arcs du graphe; le nombre de cliques obtenues est en général proche de l'optimal. L'algorithme est le suivant :

Algorithme de décomposition d'un graphe en cliques disjointes [TSEN 83] :

Soit G un graphe non orienté, tel que ses noeuds sont représentés par des nombres entiers; supposons que les noeuds i et j sont connectés et que i est inférieur à j. alors l'arc qui relie les noeuds i et j est représenté par le couple (i,j). Chacun de ces noeuds est le "voisin" de l'autre. Si un troisième noeud est connecté à i et j, il est considéré comme un "voisin commun" à ces 2 noeuds. On établit la liste des arcs du graphe, c'est-à-dire la liste des couples (i,j). Cette liste est ensuite ordonnée par ordre croissant sur le noeud gauche; par exemple, (i,j) est placé avant (k,l) si i<k.
Etant donné une liste ordonnée d'arcs d'un graphe, l'algorithme suivant partitionne les noeuds de ce graphe en un nombre (proche de l'optimal) de cliques disjointes.

1. Pour chaque arc (i,j) dans la liste ordonnée des arcs, calculer le nombre de voisins communs à i et j, et le nombre d'arcs qui seront exclus de G si i et j sont groupés. Un noeud k qui est connecté à seulement un des 2 noeuds i et j ne sera pas connecté au noeud résultant de la fusion de i et j. Par conséquent, l'arc (i, k) ou (j,k) doit être éliminé. Un noeud k qui est connecté aux deux noeuds i et j, sera connecté au noeud résultant de la fusion de i et j. Seulement un des arcs (i,k) et (j,k) doit être éliminé. Dans ce cas, et pour des raisons pratiques, on éliminera (j,k).

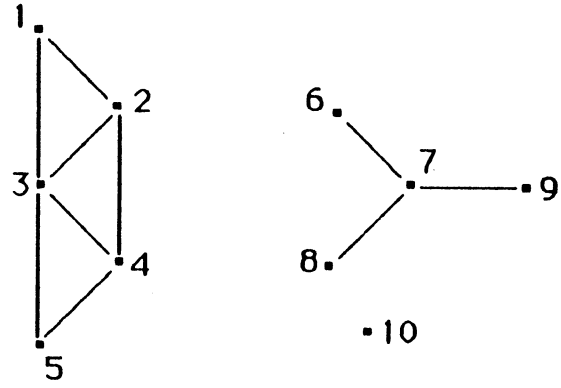
2. Choisir l'arc (p,q) dont le nombre de voisins communs est maximal. Combiner les noeuds p et q. Mettre à jour la liste des arcs de G en éliminant les arcs qui doivent être éliminés.

S'il existe plusieurs couples (i,j) ayant le nombre maximum de voisins, alors on choisit le couple qui exclut le minimum d'arcs. S'il existe encore plusieurs couples de ce type, alors on choisit un au hasard.

-Répéter l'étape 2 jusqu'à ce que la liste des arcs devient vide : la décomposition du graphe G en cliques disjointes est terminée.

exemple :
Cet exemple est donné par Tseng et Siewiorek dans [TSEN 83]; il montre la décomposition d'un graphe en un nombre mimimal de cliques disjointes en utilisant l'algorithme décrit précédemment.

Soit le graphe suivant :



La liste des arcs, le nombre de voisins communs et le nombre d'arcs à éliminer pour chaque couple de noeuds sont représentés ci-dessous :

	nombre de voisins communs ↓		
(1,2)	1	-3	← nombre d'arcs à éliminer
(1,3)	1	-3	
(2,3)	2	-3	
(2,4)	1	-4	
(3,4)	2	-3	
(3,5)	1	-3	
(4,5)	1	-3	
(6,7)	0	-3	
(7,8)	0	-3	
(7,9)	0	-3	

Par exemple, le nombre de voisins communs et le nombre d'arcs à éliminer pour (1,2) peuvent être calculés de la façon suivante : le noeud 3 est le seul noeud qui est connecté à la fois aux noeuds1 et 2. Le nombre de voisins communs pour (1,2) est donc 1. Si les noeuds 1 et 2 sont groupés ensemble, les arcs (1,2), (2,3) et (2,4) doivent être éliminés.

- Les arcs (2,3) et (3,4) ont le nombre maximum de voisins communs et excluent le même nombre d'arcs. Soit (2,3) le couple choisit pour être fusionné. Les noeuds 1 et 4 sont connectés aux noeuds 2 et 3 : (1,3) et (3,4) sont éliminés. Le noeud 5 est connecté seulement à un seul de ces deux noeuds : (3,5) est donc éliminé.
Après cette phase on obtient le graphe réduit suivant :

(1,2)	1	-3
~~(1,3)~~	~~1~~	~~-3~~
~~(2,3)~~	~~2~~	~~-3~~
(2,4)	1	-4
~~(3,4)~~	~~2~~	~~-3~~
~~(3,5)~~	~~1~~	~~-3~~
(4,5)	1	-3
(6,7)	0	-3
(7,8)	0	-3
(7,9)	0	-3

- Les arcs (1,2) et (2,4) ont le même nombre de voisins communs. (1,2) exclut moins d'arcs que (2,4); par conséquent, on fusionne les noeuds 1 et 2 : les arcs éliminés sont (1,2) et (2,4). A ce niveau {1,2,3} forme une clique puisque aucun autre arc du graphe réduit n'est connecté à un de ces noeuds.

En réitérant ainsi l'algorithme jusqu'à ce que tous les arcs soient éliminés, on obtient la partition suivante des noeuds du graphe original en 6 cliques disjointes : {1,2,3}, {4,5}, {6,7}, {8}, {9}, et {10}.

Mise à jour des instructions après le compactage des variables :

Une fois que les variables ont été compactées, les instructions opératives sont mises à jour. Pour cela, on affecte un même nom aux variables qui sont groupées ensemble; les transferts d'une variable vers elle-même sont ensuite éliminés [TSEN 83]. L'exemple suivant (donné dans [TSEN 83]) permet de mieux illustrer ce problème.

exemple :
soit la séquence de microinstructions suivante :

```
V3 := V1 + V2   ;  V12 := V1
V5 := V3 - V4   ;  V7 := V3 * V6 ;  V13 := V3
V8 := V3 + V5   ;  V9 := V1 + V7 ;  V11 := V10 / V5
V14 := V11 and V8 ;  V15 := V12 or V9
```

Supposons que les variables sont groupées de la façon suivante :
{1,14}, {2,7,9,15}, {3,8,13}, {4}, {5,11}, {6}, {10}, et {12}.

Les variables dans chacun de ces groupes peuvent être mémorisées dans un même registre. Après avoir renommé les variables, la séquence d'instructions précédente devient :

```
V3 := V1 + V2   ;  V12 := V1
```

```
V5 := V3 - V4   ;   V2  :=  V3 * V6
V3 := V3 + V5   ;   V2  :=  V1 + V2  ;  V5 := V10 / V5
V1 := V5 and V3 ;  V2 := V12 or V2
```

On remarque que le transfert "V13 := V3" dans la deuxième instruction a été éliminé, car ce dernier devient "V3 := V3".

Remarques :

L'algorithme décrit précédemment permet aussi de grouper en priorité les nœuds d'un sous graphe quelconque. On peut ainsi grouper en priorité des nœuds pour lesquels on sait que le gain résultant de ce groupement est plus important que le gain résultant du groupement d'autres ensembles de nœuds du graphe.
En particulier, si on combine 2 variables entre lesquelles existent des transferts simples, on pourra supprimer ces transferts de la description et ainsi diminuer le nombre de microinstructions ou diminuer le nombre des composants de la partie opérative. Pour grouper en priorité ces variables, il suffit en utilisant la procédure de [TSEN 83] de définir un sous graphe de compatibilité, où 2 variables sont reliées par un arc si elles ont des durées de vie compatibles et s'il existe au moins un transfert entre ces 2 variables.
Il est possible que l'on puisse diminuer davantage le nombre des microinstructions si on réapplique la procédure de compactage des microinstructions après avoir renommé les variables et supprimé les transferts d'une variable dans elle même.

Cas particulier des parties opératives à bus parallèles segmentés (cas de SYCO) :

Le principal effet de l'optimisation du nombre d'éléments de mémorisation est bien sûr de diminuer le nombre de composants de la partie opérative. Il y a cependant un effet secondaire qui peut ne pas être bénéfique dans le cas des parties opératives à bus parallèles segmentés (cas de SYCO) : les connexions sont modifiées.

Le problème d'allocation des registres d'une partie opérative à bus parallèles segmentés est tout à fait différent. Une partie opérative à bus parallèles segmentés ne permet pas toujours de connecter directement ses différents composants : il peut être nécessaire de passer par plusieurs segments de bus pour aller d'un composant à un autre. L'allocateur doit donc déterminer les segments des bus auxquels doivent être connectés les registres et l'ordre de ces segments, ce qui revient à ordonner ces registres les uns par rapport aux autres et à couper les bus en un certain nombre d'endroits pour que la description puisse être exécutable dans les limites de temps imparties à l'aide d'un nombre donné de bus parallèles. On peut simplifier le problème en supposant que les n bus sont coupés au même endroit. Le problème d'allocation revient alors à trouver une partition des registres en sous ensembles disjoints et à ordonner ces sous ensembles le long d'une droite. Ces sous-ensembles constituent des sous parties opératives.

Le principal problème d'Apollon est que l'on ne peut pas toujours connecter directement (c'est à dire au moyen d'un seul segment) 2 composants de la partie opérative et donc que l'on peut rencontrer des situations de verrou. Ces situations de verrou (non présentes dans la description originale) peuvent être provoquées par un compactage des variables algorithmiques.

exemple :

Supposons que après compactage et renommage des variables on obtient une instruction de la forme suivante :

? := A+B; ? := B+C; ? := C+A;

peu importent les destinations.

Les opérandes ne peuvent être chargés en un cycle sur une partie opérative à 2 bus segmentés [JAMI 86]. Plusieurs solutions peuvent être adoptées pour contourner ce problème :

- soit on augmente le nombre de bus,
- soit on augmente le nombre de cycles imparti à l'exécution de la ou des instructions créant le verrou, ce qui revient à ralentir l'exécution,
- soit on procède à l'opération inverse de la minimisation du nombre des registres : on les duplique de façon à diminuer le nombre des contraintes d'utilisation des bus.

Dupliquons un des registres, mettons A : pour cela, on insère dans la description comportementale avant l'instruction qui provoque le verrou une instruction qui recopie le contenu du registre source dans D.

<e1> D := A; NEXT e2 ;
<e2> ? := A+B; ? := B+C; ? := C+D;

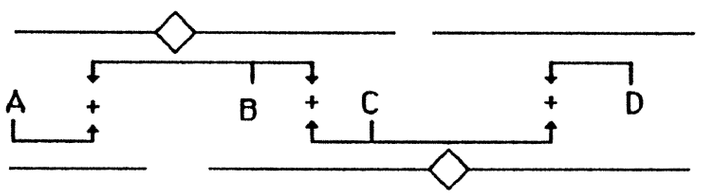


fig. 3.19 : chargement des opérandes en un cycle

Si l'on veut dupliquer un registre destination, on insèrera après l'instruction responsable du verrou une instruction qui recopie le contenu du registre destination que l'on a introduit dans le registre que l'on a "dupliqué".

Ces modifications peuvent d'ailleurs être gratuites. S'il existe une variable qui est morte (du point de vue de la durée de vie d'une variable), il suffit de recopier le registre à dupliquer dans ce registre, il n'est pas nécessaire de créer un nouveau registre.
D'autre part il peut être possible d'insérer l'opération de recopie dans une instruction existante, ce qui permet de ne pas rajouter d'instructions.

Par conséquent, dans le cas des parties opératives à bus parallèles segmentés, il faut compacter les variables de telle façon à ne pas créer une instruction opérative (ou un sous-ensemble d'instructions) impossible à exécuter. Ceci suppose que toutes les conditions nécessaires et suffisantes pour implanter une instruction (ou plusieurs) sont connues du compilateur de parties opératives, ce qui n'est pas facile dans le cas général.

4.7. Optimisation du nombre des opérateurs

Le but de cette optimisation est d'affecter un minimum d'opérateurs physiques (UAL, incrémenteur, etc...) aux opérations présentes dans la description comportementale. Pour cela, il faut grouper les opérations en sous-ensembles disjoints tels que :
-les opérations d'un même sous-ensemble peuvent être affectées à un même opérateur physique, sans créer de situation de verrou
-le nombre de sous-ensembles (donc d'opérateurs) est minimal ou proche de l'optimum.

Les outils de synthèse automatique génèrent autant d'opérateurs qu'il y a d'opérations simultanées dans la description comportementale. Différentes occurrences (non simultanées) d'une même opération peuvent donner lieu à la création d'autant d'opérateurs identiques qu'il y a d'opérations identiques, ou à la création d'un seul opérateur. Différentes occurrences de différentes opérations non simultanées peuvent donner lieu à la création d'opérateurs spécialisés ou d'opérateurs universels (UALs).

Un programme de minimisation du nombre des opérateurs se doit de tenir compte des points suivants :

1) est-ce que les opérations sont simultanées ?
2) est-ce que l'on cherche à combiner plusieurs opérateurs et à introduire des opérateurs universels ? Si oui : est ce que l'on a intérêt à regrouper les opérations qui ne portent pas sur le même nombre de bits ?
3) quelles sont les opérations que l'on peut effectuer sur un opérateur universel, comme une UAL ?
4) Et finalement est-ce qu'il existe des contraintes :

* matérielles : le nombre et la nature des opérateurs disponibles sont-ils fixés ?
*temporelles : la durée des opérations est-elle limitée ?
*architecturales : quels sont les éléments d'interconnexion utilisés ? Le nombre de ces éléments est-il limité ?

Nous distinguerons 2 cas principaux selon que la puissance du modèle architectural adopté est limitée ou non :

1) Si la puissance du modèle est limitée : la minimisation du nombre des opérateurs n'est pas le problème majeur. D'ailleurs certains opérateurs peuvent être dupliqués.

exemple :

1) Soient les 3 instructions opératives suivantes :

? := A + B ; ? := A or C;
? := B + C ; ? := B or A;
? := C + A ; ? := C or B;

La partie opérative de plus faible coût qui permet d'exécuter séquentiellement ces 3 instructions, une instruction devant s'exécuter en 2 cycles, possède 2 UALs pouvant exécuter chacune les 2 opérations "+" et "or" (fig. 3.20) :

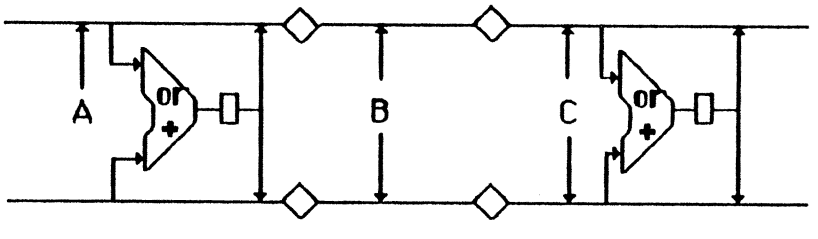


fig. 3.20 : duplication des opérateurs + et or

2) Si la puissance du modèle architectural n'est pas limitée : c'est-à-dire si on peut toujours connecter 2 composants de la partie opérative, le problème de minimisation du nombre des opérateurs est essentiel car ce sont les opérateurs qui sont en général les composants les plus coûteux en surface. Différentes approches sont possibles.

[TSEN 83] formalise à nouveau le problème de minimisation en un problème de décomposition d'un graphe en un nombre minimal de cliques disjointes : on crée un graphe de compatibilité des opérations où chaque noeud représente une opération et un arc (ei ej) existe si les opérations ei et ej ne sont jamais exécutées simultanément. L'algorithme décrit précédemment pour la décomposition d'un graphe en cliques disjointes est ensuite utilisé.

[TSEN 83] tient compte de la structure d'interconnexion des opérateurs. Par exemple si on regroupe 2 opérateurs dont les sources et les destinations sont différentes, on doit créer 2 multiplexeurs et un démultiplexeur alors que si on regroupe 2 opérateurs qui ont mêmes sources et même destination, on économise 2 démultiplexeurs et un multiplexeur.
8 types de relation de groupement sont ainsi définis, selon qu'une paire, les 2 paires d'opérandes, les destinations et les opérateurs sont identiques ou non.
Tseng cherche à grouper en priorité les opérateurs des opérations identiques, puis les opérateurs des opérations dont un seul des éléments (opérande 1, destination, opérande 2, opérateur) est différent.

5. Conclusion

Il est très difficile d'optimiser d'une manière efficace la description comportementale pour parvenir à une implémentation optimale [AGER 76]. C'est pourquoi chaque compilateur de silicium utilise des heuristiques de réduction du nombre d'états de l'algorithme d'interprétation des instructions. Certaines de ces heuristiques sont propres au compilateur de silicium (possibilité de parallélisme de la partie opérative et de la partie contrôle). D'autres sont similaires à celles utilisées par les compilateurs optimiseurs de langages de programmation évolués. Ces techniques ne garantissent pas une solution optimale, mais assurent une solution acceptable après un temps de calcul raisonnable.

Toutes les règles d'optimisation concernant la partie contrôle décrites dans ce chapitre ont été implémentées sous forme d'un programme écrit dans le langage LISP. Ce programme fait actuellement partie de l'environnement du système SYCO et permet d'optimiser chaque étage de contrôle indépendamment des autres. Il accepte en entrée une structure de données LDS représentant la table de transitions de l'étage de contrôle à optimiser et applique les transformations suivantes :
-élimination des instructions inaccessibles
-élimination des instructions de branchement inutiles
-fusion des instructions de contrôle compatibles.

L'optimisation est réalisée de manière ascendante, en partant de l'étage juste au-dessus de la partie opérative. Pendant l'optimisation d'un étage, le programme collecte certaines informations nécessaires à l'optimisation des étages supérieurs, telles que la liste des variables de contrôle affectées par chaque procédure, la liste des variables de contrôle utilisées dans des tests, etc...

La réalisation de ce programme d'optimisation a permis de mettre en évidence l'importance d'un tel logiciel dans le compilateur SYCO. Nous donnons en annexe le résultat de l'optimisation de la description comportementale du 6502, ainsi que celle d'un microprocesseur simplifié dont la description a été établie à partir d'un exemple donné dans [ANCE 82].

L'optimisation de la description comportementale de la partie opérative, qui consiste à compacter les instructions opératives compatibles n'a pu être implémentée dû à des contraintes de temps. Ce compactage doit tenir compte du taux de parallélisme maximal à ne pas dépasser de telle sorte que la largeur de la partie opérative ne dépasse pas une certaine limite spécifiée par le concepteur. Une autre optimisation intéressante à réaliser est la détection des instructions opératives identiques dans la table de transitions du dernier étage de contrôle. Elle consiste à garder une seule instruction de chaque ensemble d'instructions identiques et à remplacer toutes les références à celles éliminées par des références à celle retenue. L'analyse des tables de transitions des étages de contrôle de certains circuits pris comme exemple (Voir AnnexeI) a permis de constater l'existence de plusieurs instructions opératives identiques. Cet redondance est due au fait que pendant la phase d'extraction des niveaux d'interprétation, une instruction opérative est rajoutée autant de fois (dans la table de transitions du dernier étage de contrôle), que le nombre d'occurences de cette instruction dans la description comportementale.

CHAPITRE IV

COMPROMIS SURFACE-VITESSE

1. Introduction

La recherche d'un meilleur compromis surface-vitesse consiste à échanger une perte de performance du circuit contre un gain de surface ou vice versa. Le problème de la recherche d'un compromis surface-vitesse dans la conception automatisée de circuits intégrés n'est pas nouveau et a reçu beaucoup d'attention dans le passé. La nature de ce problème dépend du niveau de la description fonctionnelle du circuit et des critères qui rendent une implantation matérielle efficace. La recherche d'un compromis surface-vitesse dépend aussi de la stratégie de conception utilisée. Il existe actuellement deux grandes stratégies de conception automatisée de circuits intégrés :

1) Le compilateur de silicium dispose d'une architecture cible; il peut modifier la description comportementale pour trouver le compromis surface-vitesse qui satisfait au mieux les contraintes spécifiées par l'utilisateur. Des transformations dans la description comportementale telles que l'augmentation ou la réduction du parallélisme, l'expansion ou la formation de procédures peuvent être envisagées. Le compilateur SYCO fait partie de cette catégorie de compilateurs de silicium, ainsi que le système MacPitts des laboratoires Lincoln du MIT [SOUT 83], le système Bristle-Blocks [JOHA 79] et le système FIRST [BERG 84].

2) Le compilateur de silicium ne dispose pas d'une architecture cible. Dans ce cas, il doit d'abord générer une description structurelle dù circuit sous forme d'un ensemble de blocs interconnectés. Pendant la génération de cette structure, le compilateur de silicium doit déjà considérer les contraintes de surface et de vitesse spécifiées par le concepteur. Par exemple, il peut utiliser plusieurs types d'UALs pour augmenter la vitesse du circuit, les entrées d'une unité fonctionnelle peuvent être implantées à l'aide d'un bus ou d'un multiplexeur, l'horloge peut être à deux phases, ou trois phases, la partie contrôle peut être implantée à l'aide d'un PLA ou d'une ROM, etc.. Une fois la structure du circuit générée, le compilateur de silicium doit appliquer éventuellement des transformations à la description comportementale (comme celles décrites en (1)) pour améliorer l'implantation dans le style de conception choisi. Un exemple de système qui utilise cette technique de compilation est le CMU-DA [PARK 79] développé à l'université de Carnegie-Mellon.

2. Hypothèses topologiques globales

On choisit comme hypothèse de départ de restreindre la topologie globale du coeur d'un circuit intégré à un rectangle formé de deux autres rectangles de même largeur:

- en bas la Partie Opérative et ses amplificateurs de commande, notée PO,
- en haut la Partie Contrôle, notée PC.

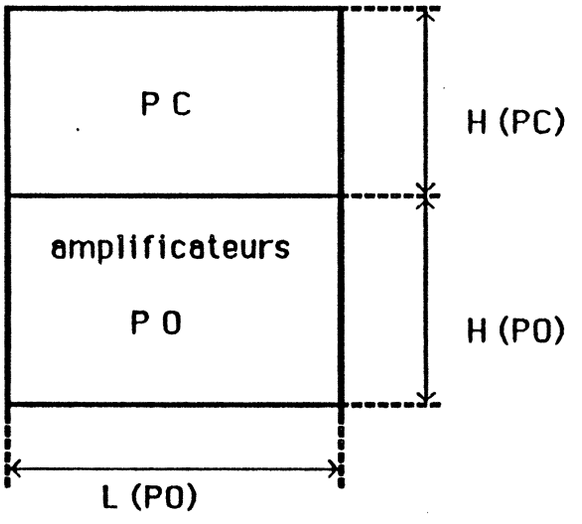


fig. 4.1 : topologie globale d'un circuit généré par SYCO

3. Relations entre performances et surface

Dans le compilateur SYCO les performances du circuit final dépendront essentiellement de la taille de la partie opérative, qui est fonction du :

- nombre de bits du chemin de données qui fixe la hauteur H(PO) qui est égale à :

H(PO) = H(tranche) * NBIT + H(amplificateur)

où H(tranche) est la hauteur connue d'une "tranche" de 1 bit (bit-slice), NBIT le nombre de bits, et H(amplificateur) la hauteur d'un amplificateur.

- du nombre de composants (éléments de mémorisation et opérateurs), de la puissance de ces opérateurs et du degré de parallélisme (sous parties opératives fonctionnant en parallèle).

Pour simplifier encore nos hypothèses, nous raisonnerons avec un nombre de bits NBIT fixé, ce qui introduit une constante topologique "H(PO)". Les performances

ne dépendront donc plus que de L(PO), et la Partie Contrôle PC devra bien sûr être capable de commander la PO, en exploitant le parallélisme fourni par ses ressources matérielles, et en fonctionnant à la même fréquence que cette dernière : les "temps de cycle" des 2 parties PC et PO doivent être égaux [SCHO 85].

La contrainte globale au niveau de la forme du circuit ayant été énoncée (L(PC) = L(PO), H(PO) fixe), le problème à résoudre se résume donc à compiler la description comportementale du circuit pour l'implanter dans un rectangle de largeur L(PO) et de hauteur minimum (H(PC)+H(PO) min.).

4. Position du problème

Dans le compilateur de silicium SYCO, le parallélisme entre les actions opératives est spécifié par le concepteur ainsi que le nombre d'étages de la partie contrôle. Cette grande liberté offerte à l'utilisateur dans l'élaboration de la description comportementale du circuit, pose parfois des problèmes d'implantation. Ces problèmes se résument de la façon suivante :

1) Le nombre d'étages de contrôle est élevé parce que la description comportementale est trop modulaire.

2) Si le degré de parallélisme entre les actions opératives est faible, cela conduit à un circuit dont la partie opérative est très étroite et la partie contrôle très haute :

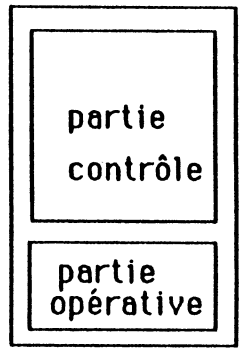


fig. 4.2 : circuit lent

3) Inversement, si le degré de parallélisme entre les actions opératives est élevé, cela conduirait certainement à un circuit dont la partie opérative est trop large (fig.4.3); ce circuit aura donc de bonnes performances, mais sa surface risque d'être trop élevée.

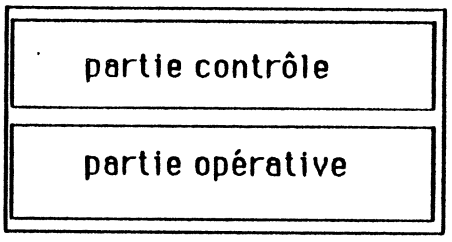


fig. 4.3 : circuit rapide mais trop large

4) Les étages constituant le circuit ont des largeurs très différentes (fig. 4.4). En effet, un des gros problèmes de la génération des spécifications des masques de la partie contrôle a pour origine l'irrégularité (d'un point de vue géométrique). Le découpage d'une partie contrôle en tranches permet d'obtenir des masques uniquement par empilement des différentes tranches de la même façon que l'on obtient les masques de la partie opérative en empilant les différentes tranches d'un bit. Il y a cependant une différence importante : les tranches de la partie opérative ont toutes la même largeur, ce qui n'est pas le cas des tranches de contrôle en général. Cette irrégularité est due au fait que dans la description comportementale, le nombre total d'instructions varie considérablement d'un niveau de la hiérarchie à l'autre. Lorsque les étages constituant le circuit ont des largeurs très différentes, la surface des connexions entre les étages de contrôle augmente, et entraîne par la suite une grande perte de surface (fig. 4.4) :

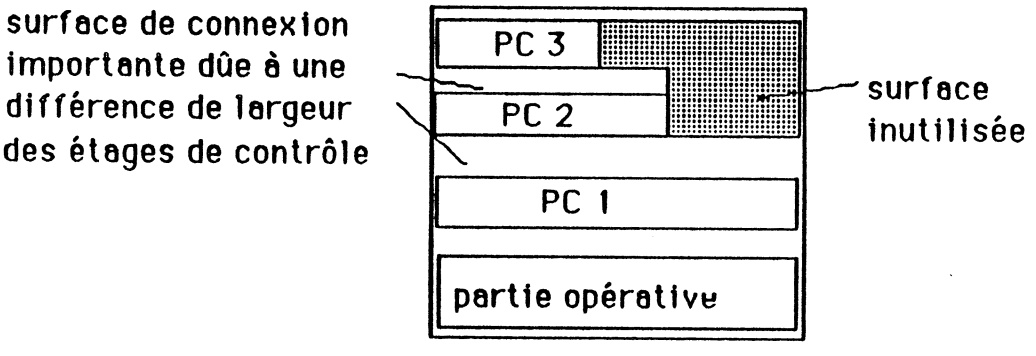


fig. 4.4 : perte de surface due à des étages irréguliers

5) "Effet pyramide" : ce cas se présente lorsque le circuit à la forme d'une pyramide; plus la tranche de contrôle est proche de la partie opérative, plus elle a tendance à s'élargir [GERO 85], [MHAY 86] :

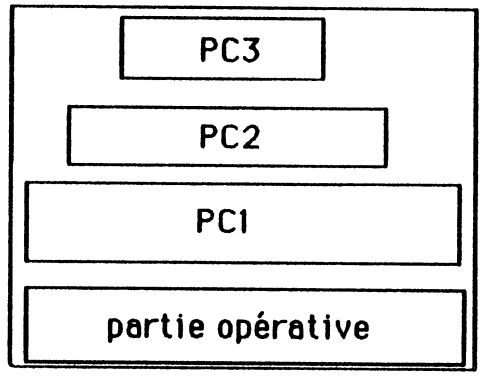


fig. 4.5 : circuit pyramidal

Plusieurs causes peuvent être à l'origine de ce problème :

(1) Dans la description comportementale, le nombre d'états de l'algorithme augmente d'un niveau d'interprétation au niveau d'interprétation inférieur et ainsi de suite jusqu'à la partie opérative.

(2) Dans la description comportementale, le nombre de procédures augmente d'un niveau d'interprétation au niveau inférieur et ainsi jusqu'au dernier niveau qui est la partie opérative. En effet, si chaque procédure appelle plusieurs autres procédures et si les procédures de niveau i-1 appelées par les procédures de niveau i sont toutes différentes entre elles, alors le nombre de branches à chaque niveau de l'arbre des appels de procédures ne peut qu'augmenter. En effet, une commande de niveau i ne peut actionner que la machine de niveau i-1, les différentes tranches traversées n'étant pas transparentes aux commandes.

la figure suivante montre un exemple d'arbre d'appel où le nombre de procédures augmente avec la profondeur des couches d'interprétation :

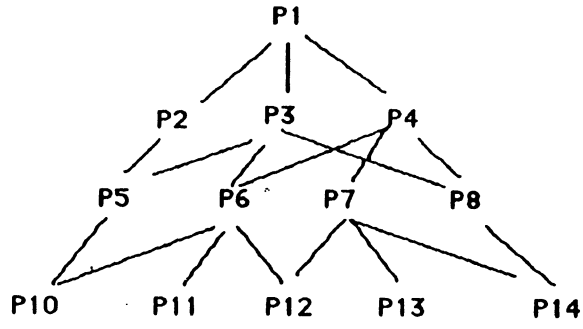


fig. 4.6 : arbre d'appel de procédures en forme de pyramide

Le nombre de branches de l'arbre des appels des procédures n'augmente pas avec la profondeur si aucune procédure ne fait appel à plus d'une procédure de niveau

inférieur ou alors si les mêmes procédures de niveau inférieur sont appelées plusieurs fois.

(3) Le problème de "l'effet pyramide" est aussi dû au fait que lorsqu'on écrit un programme dans un langage évolué tel que LDS, on a tendance à décharger les procédures de niveau supérieur et à mettre tous les traitements dans les procédures de niveau inférieur, ceci pour rendre le programme plus structuré et plus lisible.

(4) Enfin, ce problème est aussi accentué par le modèle architectural de partie contrôle qui a été adopté; actuellement ce modèle ne permet pas la transparence d'un étage de contrôle; cette limite du modèle de partie contrôle actuel, entraîne le concepteur à faire "descendre" les appels à la partie opérative vers les niveaux d'interprétation inférieurs.

Pour faire face à tous ces problèmes, il faut modifier la description comportementale pour trouver un meilleur compromis surface-vitesse. Les transformations appliquées à la description comportementale doivent bien sûr préserver le comportement du circuit.

5. Processus de recherche d'un compromis surface-vitesse

La recherche d'un compromis surface-vitesse consiste à transformer progressivement la description comportementale pour parvenir à une meilleure implantation du circuit. Le processus de recherche d'un meilleur compromis est un processus itératif qui prend fin lorsque toutes les contraintes de surface et de vitesse spécifiées par le concepteur au début de la compilation sont satisfaites (fig . 4.7).

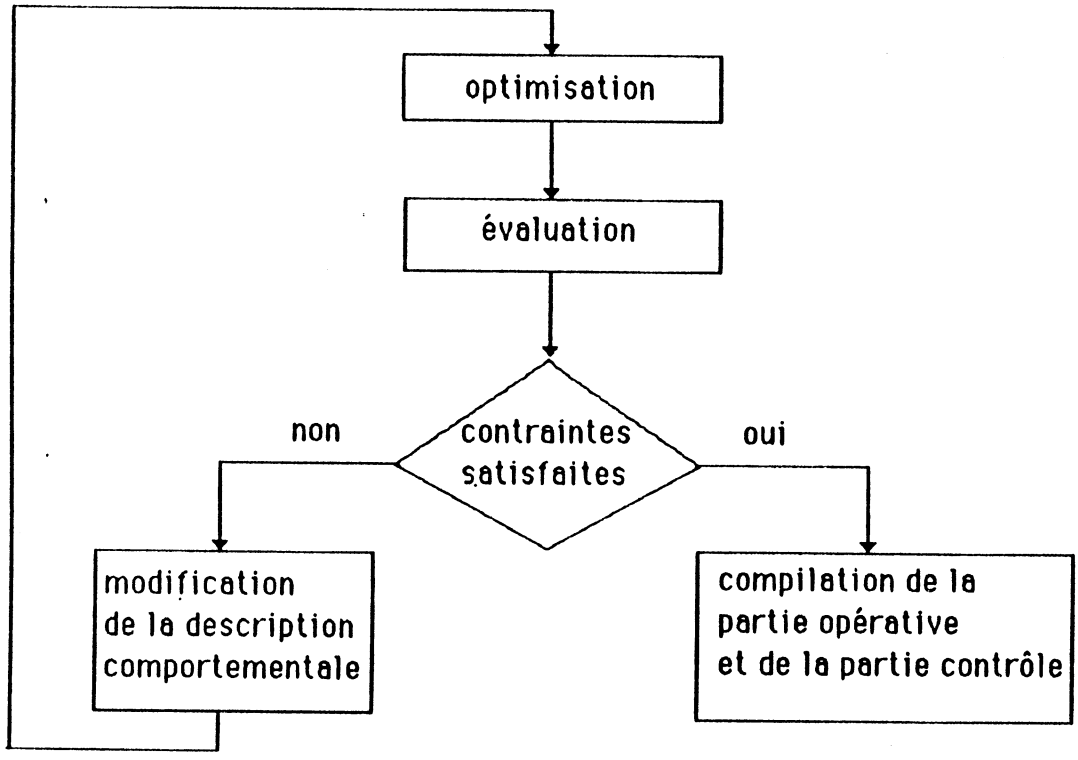


fig. 4.7 : boucle itérative de recherche d'un bon compromis surface-vitesse

Le module de transformation de la description comportementale coopère avec un évaluateur de surface et de vitesse. Le compilateur de silicium compare les résultats de l'évaluation avant et après chaque transformation de la description comportementale pour décider si une modification doit être effectuée ou non. Après chaque modification de la description comportementale le module d'optimisation est appelé car certaines transformations de la description comportementale telles que l'expansion de procédures peuvent conduire à l'apparition de certaines instructions inutiles ou inaccessibles. Il faut donc éliminer ces instructions ("dead code élimination") avant d'appeler le module d'évaluation afin d'obtenir une évaluation plus fine des nouvelles caractéristiques du circuit.

6. Critères de choix d'une transformation

- Si la vitesse est un paramètre important :

Si la vitesse est importante, il faut remplacer les appels de certaines procédures par leurs codes. Une description comportementale modulaire d'un circuit peut contenir plusieurs procédures qui ne sont appelées qu'une seule fois. Le concepteur peut avoir écrit la description de cette façon afin qu'elle soit plus facile à comprendre. Une implantation matérielle directe d'une telle description peut conduire à un

circuit très lent. La description peut aussi contenir des procédures qui sont appelées plusieurs fois. Si la procédure est courte, on peut augmenter la vitesse du circuit en remplaçant l'appel à cette procédure par son code sans pour autant augmenter la taille de la partie contrôle. Dans le compilateur SYCO, l'expansion de procédures se traduit au niveau matériel par une fusion de deux étages de contrôle voisins.

- Si la surface est un paramètre important :

Dans ce cas, il faut réduire les performances du circuit pour diminuer la surface de silicium occupée par ce dernier. Dans le compilateur SYCO, la surface d'un circuit est égale au produit de la largeur de la partie opérative (notée PO) par la hauteur du circuit :

Surface (circuit) = largeur (PO) * (hauteur (PC) + hauteur (POP))

La hauteur de la partie opérative étant invariable (nombre de bits fixe) pour réduire la surface d'un circuit il faut donc réduire soit la largeur de la partie opérative, soit la hauteur de la partie contrôle. Nous montrerons dans la suite de ce chapitre que ces deux paramètres sont très liés : la réduction (resp. augmentation) de la largeur de la partie opérative entraîne souvent une augmentation (resp. diminution) de la hauteur de la partie contrôle

7. Transformations de la description comportementale

Ces transformations peuvent être résumées dans la table suivante :

	Transformation	But
Actions sur la PC	Fusion de 2 étages de contrôle	Réduire le nombre d'étages de contrôle pour augmenter la vitesse du circuit
	Eclatement d'un étage de contrôle	Réduire la surface occupée par un étage de contrôle
	Fusion partielle de 2 étages de contrôle	Ajuster les largeurs de 2 étages de contrôle pour réduire la surface des interconnexions
Actions sur la PO	Augmentation du parallélisme	Augmentation de la vitesse du circuit
	Réduction du parallélisme	Réduction de la largeur du circuit

fig. 4.8 : transformations de la description comportementale

Pour chacune de ces transformations nous allons étudier ses conséquences sur les performances et la surface du circuit. Nous donnerons aussi les conditions nécessaires pour qu'une transformation puisse améliorer la vitesse ou la surface du circuit. Par conséquent, toute transformation qui provoque une augmentation de la surface du circuit et de sa vitesse sera rejetée par le système.

7.1. Transformations de la description comportementale de la partie contrôle

7.1.1. Fusion de deux étages de contrôle (expansion de procédures)

Dans le compilateur SYCO, le fait de partitionner la partie contrôle en plusieurs parties, réduit le nombre d'états, d'entrées et de sorties de la tranche principale (c.a.d l'étage de contrôle le plus haut), qui dans le cas contraire aurait une grande surface et un temps de réponse très grand. Mais d'un autre côté, la profondeur séquentielle du système augmente : un étage de PC appelle un autre étage de PC qui appelle à son tour un autre, et ainsi de suite jusqu'à la partie opérative. De ce fait, beaucoup de temps est perdu jusqu'à ce qu'une opération soit exécutée par la partie opérative. Pour éviter cet inconvénient, un algorithme de répartition sur l'ensemble des tables de transitions d'états est appliqué. Cet algorithme estime la taille d'un étage de PC à partir du nombre de ses entrées, de sorties et transitions d'états et essaye d'enlever les petits étages de contrôle en incluant leurs états dans l'étage de

contrôle juste au dessus (fig. 4.9).

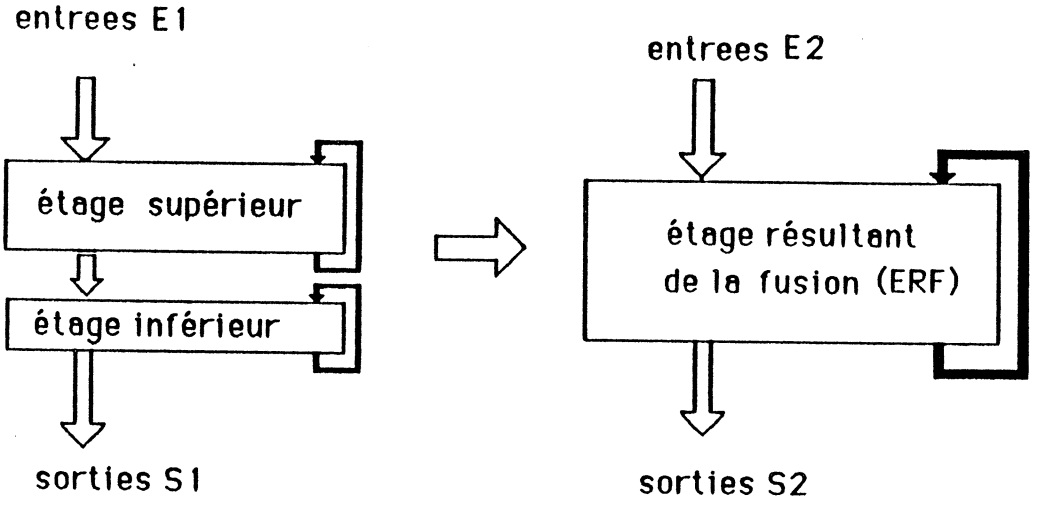


fig. 4.9 : fusion de deux étages de contrôle

La fusion de deux étages de contrôle adjacents est réalisée au niveau comportemental, en remplaçant les appels de procédures dans la table de transitions associée à l'étage supérieur, par les codes de ces procédures qui se trouvent dans la table de transitions correspondant à l'étage inférieur. Cette transformation est connue généralement sous le nom d'"expansion de procédures".
La nouvelle table de transition obtenue par cette transformation décrit le comportement de l'étage résultant de la fusion. Cette table est ensuite optimisée en éliminant les états inutiles ou inaccessibles résultant de l'expansion des procédures.
A partir du nombre d'états, d'entrées, de sorties et de transitions d'états de l'étage résultant de la fusion (ERF), l'évaluateur calcule pour chaque type d'implantation (PLA, ROM, etc...) la surface totale occupée par l'ERF ainsi que les nouvelles performances du circuit. Le résultat de l'évaluation est comparé à celui fait avant la fusion, ceci pour décider si cette transformation doit être réellement effectuée (un retour en arrière est donc possible).

Les compilateurs-optimiseurs de langages de programmation évolués utilisent l'expansion de procédures pour réduire le nombre d'appels de procédures ainsi que le traitement préliminaire à chaque appel (sauvegarde de l'adresse de retour, passage des paramètres). L'expansion de procédures est une bonne technique d'optimisation bien qu'elle puisse augmenter la taille d'un programme. Elle réduit la fréquence d'appel des procédures, diminue le temps de passage des paramètres, et permet de meilleures optimisations globales [ALLE 80], [SCHE 77].

L'expansion de procédures est aussi utilisée par les compilateurs de silicium pour la recherche d'un compromis surface-vitesse. Si le langage de description de circuits utilisé pour spécifier l'algorithme de fonctionnement permet d'utiliser des

procédures, il peut être utile de remplacer les appels de certaines procédures par des copies de ces procédures appelées. Cette transformation est d'ailleurs effectuée systématiquement dans le cas où il n'y a pas de mécanisme spécifique pour exécuter des procédures. Mais dans le cas où les appels de procédures sont interprétés de manière spécifique par la machine (empilement des niveaux d'interprétation [JERR 86], saut à des microsubroutines dans la "mémoire" microprogrammée lié à une pile d'adresses de retour et d'un pointeur de pile [RAJA 85], il peut être néanmoins utile de remplacer l'appel d'une procédure par sa copie, spécialement dans le cas où une procédure n'est appelée qu'une seule fois ou dans le cas où une procédure est très courte car son expansion ne conduira pas à une augmentation significative de la taille de la partie de contrôle.
L'expansion de procédures est actuellement utilisée dans plusieurs compilateurs de silicium dont le langage de description de circuits utilisé permet l'utilisation des procédures. Par exemple, elle est utilisée dans le système CMU-DA pour l'augmentation de la vitesse d'un circuit [RAJA 85]. La figure 4.10 montre un exemple d'insertion de procédure effectuée automatiquement par le CMU-DA dans le bloc d'interprétation de l'instruction RTI du microprocesseur MCS6502. Cette instruction est écrite dans le langage ISPS [BARB 81] utilisé comme langage d'entrée par le CMU-DA. L'instruction RTI appelle 3 fois la procédure POP qui retourne comme résultat le contenu du sommet d'une pile. Le temps d'exécution de chaque microinstruction est indiqué en commentaire après le point d'exclamation "!"; les instructions effectuées en parallèle sont séparées par ";" et les instructions exécutées séquentiellement sont séparées par "next". La figure 4.10a montre que le transfert de chaque sommet de pile prend 3 cycles d'horloge (1 pour l'appel de la procédure POP, et 2 pour exécuter cette procédure). Il faut donc 9 cycles pour exécuter l'instruction RTI. La figure 4.10b montre que l'instruction RTI peut être exécutée en 4 cycles seulement après l'expansion des 3 appels de la procédure POP.

```
RTI() :=
BEGIN ! (9)
   P = POP()  next ! (3)
   PCL = POP() next ! (3)
   PCH = POP() ! (3)
END

POP() :=
BEGIN ! (2)
   S = S + 1 next ! (1)
   POP = M[1'@S] ! (1)
END
```

4.10 a - Avant expansion

```
RTI() :=
BEGIN ! (4)
   S = S + 1 next ! (1)
   (P = M[1'@S]; S = S + 1) next ! (1)
   (PCL = M[1'@S]; S = S + 1) next ! (1)
   PCH = M[1'@S] ! (1)
END
```

4.10 b - Après expansion

fig 4.10 : expansion de procédure dans le 6502

7.1.1.1. Algorithme de la fusion de deux étages de contrôle

Pour fusionner deux étages de contrôle voisins, tout en préservant le comportement du système, les instructions de type "execute Pi" ne sont pas toutes remplacées par le corps de la procédure Pi; ceci dépend du contexte de l'instruction "execute Pi". Pour cela, les cas suivants ont été recensés :

1) La procédure appelante est constituée seulement d'une instruction de type "EXECUTE" suivie d'une instruction "EXIT" :

Dans ce cas, l'instruction "EXECUTE Pi" est remplacée par le corps de la procédure Pi.

exemple :

```
<P1> : execute P2; exit;
. . . . . .
<P2> : A := B + C;
       E := 1; exit;
```

⇨

```
<P1> : A := B + C;
       E := 1; exit
```

2) La procédure appelante est constituée seulement d'une instruction de type "EXECUTE Pi" suivie d'une instruction "NEXT Pj" :

Dans ce cas, l'instruction "EXECUTE Pi" est remplacée par le corps de la procédure Pi; toutes les instructions de type "EXIT" à l'intérieur de Pi sont remplacées par des "NEXT Pj".

exemple :

```
<P1> : execute P2; next P3;
. . . . . .
<P2> : A := B + C;
       E := 1; exit;
```

⇨

```
<P1> : A := B + C;
       E := 1;
       next P3;
```

Si la procédure appelée Pi contient une instruction "NEXT Pi" (boucle itérative), après la fusion, cette instruction sera remplacée par "NEXT Pk", Pk étant le nom de la procédure appelante.

exemple :

```
<P1> : execute P2; next P3;
. . . . . .
<P2> : if C>0  then C := C + 1;
                    exit;
       if C<0  then next P2;
```

⇨

```
<P1> : if C>0 then C := C+1;
                    next P3;
       if C<0 then next P1;
```

3) La procédure appelante est constituée de plusieurs instructions :

Dans ce cas, on remplace l'intruction "EXECUTE Pi" par "NEXT Pi".

exemple :

```
<P1> : r:= 1;
       execute P2;
       next P3;
. . . . . .
<P2> : A := B + C;
       E := 1;
       exit;
```

⇨

```
<P1> : r := 1;
       next P2;

<P2> : A := B + C;
       E := 1;
       next P3;
```

Le programme d'optimisation va ensuite fusionner les instructions P1 et P2 puisque l'action "r:=1" est compatible avec les actions "A:=B + C" et "E:= 1"; finalement, l'instruction P2 est éliminée et l'instruction P1 devient :

```
<P1> : r := 1;
       A := B + C;
       E := 1;
       next P3;
```

exemple :

```
<P1> : if C1 then r:= 1;
                  execute P2;
                  exit;
       if C2 then r:= 2;
                  execute P3;
                  exit;
       if C3 then r:= 3;
                  execute P2;
                  exit;
```

⇨

```
<P1> : if C1 then r:= 1;
                  next P2;
       if C2 then r:= 2;
                  next P3
       if C3 then r:= 3;
                  next P2.
```

Dans cet exemple, le module d'optimisation va essayer de fusionner P2 et P3 avec P1, en testant si les instructions de P2 sont compatibles avec r:= 1 et r:= 3, et si les instructions de P3 sont compatibles avec r:=2.

Remarques :

1) Si une procédure est dupliquée plusieurs fois, différents noms sont attribués aux étiquettes de cette procédure à chaque fois que celle-ci est insérée dans la table de

transitions de l'étage de contrôle appelant, afin d'éviter toute ambigüté dans le séquencement interne de cet étage.

2) Si la suite d'instructions "EXECUTE Pi; NEXT Pj" (ou "EXECUTE Pi ; EXIT") apparaît plusieurs fois dans le corps de la procédure appelante, la procédure Pi n'est insérée qu'une seule fois.

exemple :

```
<P1> : if C1 then r:= 1;
                    execute P2;
                    next P3;
       if C2 then r:= 2;
                    execute P2;
                    next P3;
       if C3 then r:= 3;
                    execute P2;
                    next P4;
. . . . . .

<P2> : A := B + C;
       E:= 1;
       exit;
```

⇨

```
<P1> : if C1 then r:= 1;next P2;
       if C2 then r:= 2; next P2;
       if C3 then r:=3; next P'2;

<P2> : A := B + C;
       E:= 1;
       next P3;

<P'2> : A := B + C;
       E:= 1;
       next P4;
```

7.1.1.2. Caractéristiques de l'étage de contrôle résultant de la fusion

Nombre d'états de l'ERF :

Soient deux étages de contrôle voisins E_i et E_{i-1} tels que l'étage supérieur E_i soit constitué de m automates notés Pi et l'étage inférieur E_{i-1} de n automates notés Hj. Soient A(Hj) le nombre d'appels total à l'automate Hj par l'étage supérieur E_i, et N(Hj) (resp. N(Pi)) le nombre d'états dans l'automate Hj (resp. Pi). Après fusion de ces deux étages de contrôle, on obtient un ERF composé de m automates et dont le nombre d'états maximum (c.a.d non optimisé) est donné par l'expression suivante :

$$(1)\ \text{Nombre d'états de l'ERF} = \sum_{i=1}^{m} N(Pi) + \sum_{j=1}^{n} A(Hj) \cdot N(Hj)$$

Remarque :

Dans l'expression (1), si A(Hj) est égal à 1 quelque soit j, alors le nombre d'états de l'étage résultant de la fusion est égal à la somme des nombres d'états des étages

fusionnés. Par conséquent, si chaque automate d'un même étage de contrôle est appelé une seule fois par l'étage supérieur, il est préférable de fusionner ces deux étages de contrôle surtout dans le cas où ces étages ne contiennent pas beaucoup d'états.

7.1.1.3. Influences sur les performances et la surface du circuit

La fusion de deux étages de contrôle implique les conséquences suivantes sur les caractéristiques du circuit :

-Réduction du nombre d'étages de contrôle, ce qui entraîne une réduction du temps de traversée global de la partie contrôle et une réduction de la surface totale occupée par les connexions.
-La fusion implique aussi un gain de vitesse dû à une réduction du nombre d'appels et de retours de procédures. Le gain de vitesse calculé en nombre de cycles d'horloge peut facilement être estimé; il est égal au nombre d'instructions "EXECUTE Pi" remplacées directement par le corps de la procédure Pi; en effet, à chaque élimination d'un appel de procédure, on diminue d'un cycle d'horloge le temps d'exécution de l'algorithme d'interprétation des instructions du microprocesseur. Le gain maximum est lorsque toutes les instructions "EXECUTE Pi" sont remplacées par les procédures Pi :

$$(2)\quad \text{Gain de performances maximum} = \sum_{j=1}^{n} A(Hj)$$

Les expressions (1) et (2) montrent bien le compromis "surface-vitesse" : plus les termes A(Hj) sont grands, plus le gain de performances est élevé ainsi que l'augmentation de la surface de la partie contrôle.

-La fusion peut conduire à modifier la durée du cycle d'horloge si le temps de réponse du PLA implémentant l'étage résultant de la fusion est plus grand que le temps critique de réponse d'un étage de contrôle avant la fusion.

7.1.2. Fusion partielle de deux étages de contrôle

Cette transformation est un cas particulier de la fusion totale et consiste à fusionner seulement quelques procédures d'un étage de contrôle avec l'étage supérieur ou inférieur. Cette transformation permet d'ajuster la taille de deux étages de contrôle de telle manière qu'ils possèdent à peu près la même largeur ou la même hauteur en fonction du type d'implantation choisi pour chaque étage de contôle. Par exemple, si l'implantation choisie pour réaliser les deux tranches de contrôle est un PLA classique disposé horizontalement (c'est à dire que sa matrice ET est topologiquement au-dessus de sa matrice OU), alors la variation du nombre d'états (donc du nombre de monômes du PLA) implique une variation de la largeur de la

tranche de contrôle. Dans ce cas, la fusion partielle permet d'équilibrer les largeurs de ces deux tranches de contrôle. Nous distinguerons deux types de fusion partielle :

- fusion partielle avec l'étage supérieur
- fusion partielle avec l'étage inférieur.

7.1.2.1. Fusion partielle d'un étage de contrôle avec l'étage supérieur

La fusion partielle avec l'étage supérieur permet d'équilibrer la taille de deux étages en "transférant" une partie du code interprété par l'étage inférieur dans la table de transitions de l'étage supérieur :

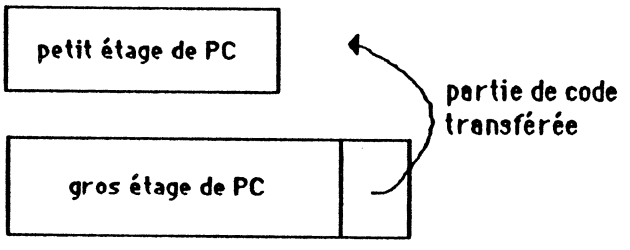


fig. 4.11 : fusion partielle avec l'étage supérieur

Cette transformation est effectuée en remplaçant dans la table de transitions de l'étage de contrôle supérieur les instructions d'appel aux procédures selectionnées pour être fusionnées par les codes de ces dernières qui se trouvent dans la table de transitions de l'étage de contrôle inférieur. Pour préserver le comportement du système et par manque de transparence d'un étage de contrôle, il faut rajouter autant d'états intermédiaires dans l'étage inférieur que d'automates activés par les procédures fusionnées. Ceci permettra d'activer ces automates à partir de l'étage supérieur après la fusion partielle.

exemple :

La figure 4.12 montre un exemple de fusion partielle avec l'étage supérieur. Dans ce cas, on suppose que seulement la procédure P4 de l'étage E_i est fusionnée avec l'étage supérieur E_{i+1}. La procédure P4 appelle les procédures P7 et P8 de E_{i-1}. Par conséquent, une fois que cette procédure est fusionnée avec E_{i+1}, il faut rajouter deux états intermédiaires e7 et e8 dans E_i pour pouvoir appeler les procédures P7 et P8 à partir de l'étage E_{i+1}. Les instructions correpondant à ces états ont les syntaxes suivantes :

<e7> : execute P7; exit;

<e8> : execute P8; exit;

Après la fusion de P4 avec E_{i+1} toutes les instructions "execute P7" et "execute P8" dans le corps de la procédure P4 sont remplacées respectivement par "execute e7" et "execute e8". L'appel de l'état e7 (resp. e8) dans E_i provoque à son tour l'activation de la procédure P7 (resp. P8).

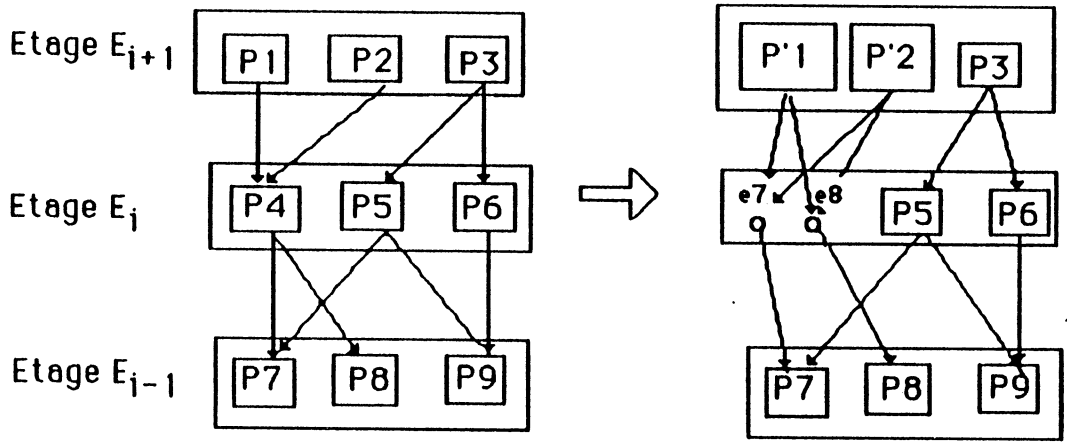


fig. 4.12 : fusion de la procédure P4 avec l'étage E_{i+1}

Conséquences sur la partie contrôle :

- Le nombre d'états de l'étage de contrôle supérieur E_{i+1} augmente, donc sa surface augmente aussi.

- Le nombre d'états de l'étage de contrôle E_i diminue si le nombre d'instructions dans la (ou les) procédure(s) fusionnée(s) avec l'étage supérieur est supérieur au nombre d'états intermédiaires rajoutés dans E_i. Cette condition est d'ailleurs une condition nécessaire pour que la fusion partielle puisse être envisagée par le système.

-La fusion partielle implique aussi une augmentation de la vitesse puisque le nombre d'appels des procédures est réduit.

7.1.2.2. Fusion partielle d'un étage de contrôle avec l'étage inférieur

Cette transformation est l'inverse de la précédente; elle consiste à transférer une partie du code interprété par l'étage de contrôle supérieur vers l'étage inférieur.

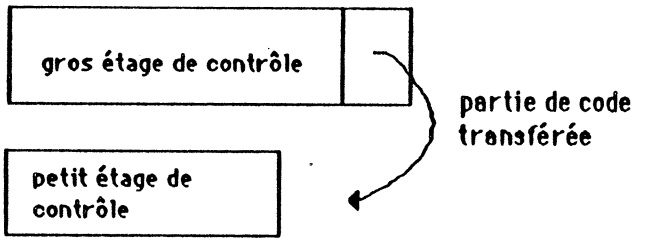


fig. 4.13 : fusion partielle avec l'étage inférieur

Pour chaque procédure Pi de l'étage supérieur E_i fusionnée avec l'étage inférieur E_{i-1}, on ne rajoute qu'un seul état intermédiaire ei dans E_i, dont la syntaxe est la suivante :
<ei> : execute Pi; exit;

Ceci permet d'activer la procédure Pi à partir de l'étage E_{i+1}.

Reprenons l'exemple précédent (fig. 4.12), et supposons que cette fois la procédure P4 doit être fusionnée avec l'étage inférieur E_{i-1}. P4 appelle les procédures P7 et P8 de l'étage E_{i-1}. Pour fusionner P4 avec E_{i-1}, il faut remplaçer dans P4 toutes les instructions d'appel à P7 et P8 par les codes de ces procédures. Après la fusion de P4 avec E_{i-1}, il faut éliminer de E_{i-1} les procédures qui ne sont plus référencées ("dead code elimination"). C'est le cas de la procédure P8 de E_{i-1} qui n'est appelée que par P4. Pour pouvoir appeler la procédure P4 à partir de l'étage E_{i+1}, on rajoute un état intermédiaire e4 dans E_i et dont la syntaxe est la suivante :

<e4> : execute P4; exit ;

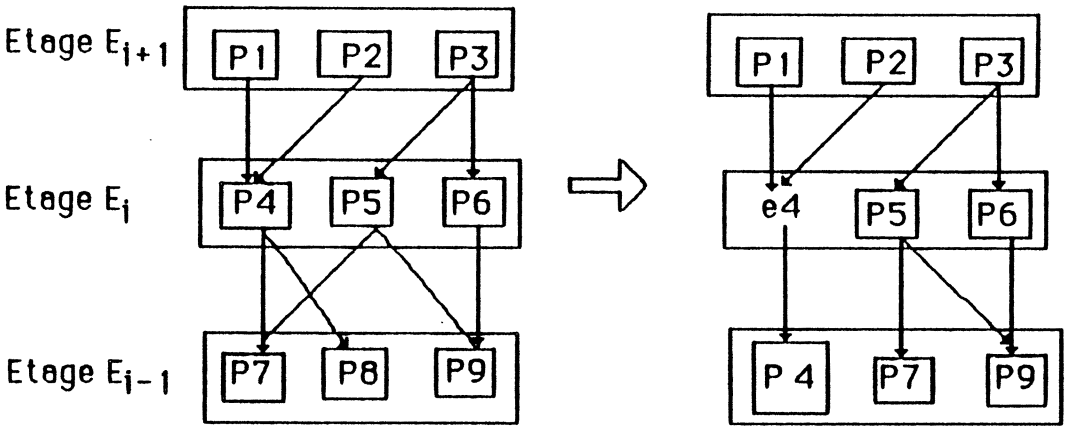


fig. 4.14 : fusion de la procédure P4 avec l'étage inférieur E_{i-1}.

Généralisation de la fusion partielle "inférieure" :

Soient { $P_1, P_2, \ldots, P_k$ } l'ensemble des procédures de l'étage E_i choisies pour être fusionnées avec l'étage inférieur E_{i-1} et $N(P_i)$ le nombre d'états de la procédure P_i. On obtient alors l'expression suivante pour le calcul du nombre d'états (noté $N(E'_i)$) de l'étage E_i après la fusion :

$$(3) \quad N(E'_i) = N(E_i) + k - \sum_{i=1}^{k} N(P_i)$$

($N(E_i)$ représente le nombre d'états de E_i avant la fusion partielle)

Remarque : Une condition nécessaire pour que la fusion partielle avec l'étage inférieure puisse être appliquée par le système est que $N(E'_i) < N(E_i)$. Autrement dit, il faut que le nombre d'états de l'étage supérieur E_i diminue après la fusion partielle. D'après l'expression (3) cette condition est vérifiée si :

$$k < \sum_{i=1}^{k} N(P_i)$$

Il faut donc que le nombre d'états intermédiaires rajoutés dans E_i soit inférieur au nombre d'états total dans les procédures fusionnées avec l'étage inférieur E_{i-1}.

7.1.3. Décomposition d'un étage de contrôle

Cette transformation est l'action inverse de la fusion. Elle permet de décomposer un gros étage de contrôle en deux étages de contrôle plus petits (fig. 4.15).

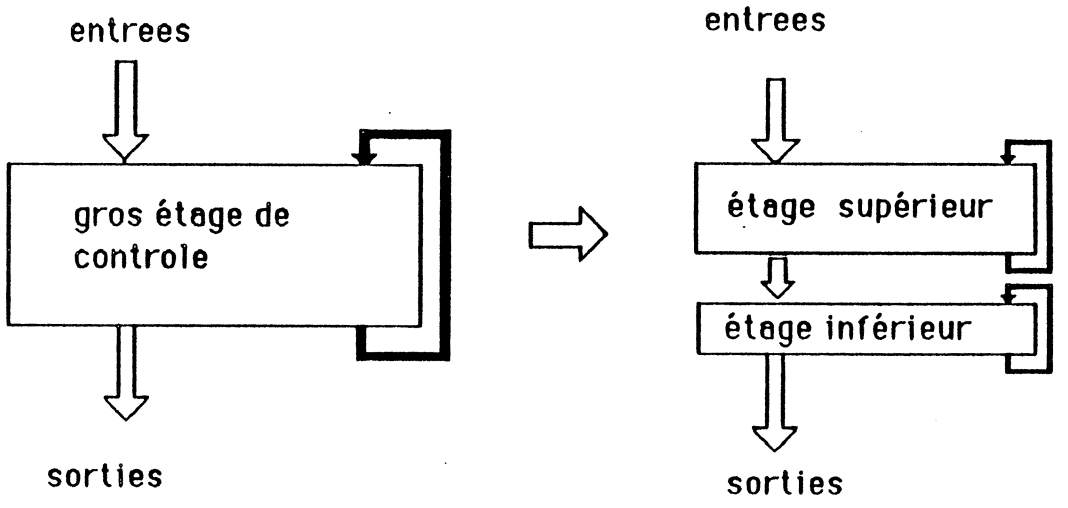


fig. 4.15 : éclatement d'un étage de contrôle

La surface totale occupée par les deux étages de contrôle après la décomposition est déterminée par l'évaluateur de surface en fonction du nombre d'états, d'entrées, de sorties et de l'implantation choisie pour chaque étage de contrôle (les étages résultant de la décomposition peuvent être implantés de manières différentes). La surface occupée par les connexions est prise en compte. Les résultats de l'évaluation de surface sont ensuite comparés à ceux de l'évaluation effectuée juste avant la décomposition pour décider si cette transformation doit être exécutée.

La décomposition d'un étage de contrôle peut être réalisée de deux manières différentes :

-une première solution consiste à détecter des séquences d'instructions identiques dans les procédures interprétées par l'étage de contrôle à décomposer. Cette transformation réalisée par certains compilateurs optimiseurs de langages de programmation évolués est connue généralement sous le nom d'"insertion de procédures".

-la deuxième solution, beaucoup plus simple que la première mais moins avantageuse du point de vue surface, consiste à répartir les procédures de l'étage à décomposer en deux niveaux d'interprétation.

7.1.3.1. Insertion de procédures

L'insertion de procédures consiste à enlever d'un étage de contrôle des parties de code (de procédures) identiques pour créer un nouvel étage de contrôle constitué de ces codes; des instructions d'appel à ces nouvelles procédures sont ensuite rajoutées dans les procédures originales. Chaque partie de code choisie pour être remplacée par une procédure doit avoir un seul point d'entrée et un seul point de sortie. En effet, cette partie de code est destinée à constituer un automate d'un nouvel étage de

contrôle; par conséquent elle ne peut être activée que par un seul point d'entrée. D'autre part, elle ne doit avoir qu'une seule sortie afin de ne pas créer d'ambigüité dans le séquencement interne de l'étage appelant lors du retour de contrôle à cet étage.
La figure 4.16 représente un étage de contrôle constitué de 3 automates et contenant des parties de code identiques (représentées par des carrés). Dans cet exemple, on suppose que chaque arc (ei ej) est étiqueté avec un seul numéro représentant à la fois la condition (si l'état est conditionnel) et les actions (activation de l'étage inférieur + affectations de variables de contrôle) liées à la transition de l'état ei vers l'état ej; deux arcs portent le même numéro si les transitions correspondantes utilisent le même test (si ce sont des transitions conditionnelles) et génèrent les mêmes actions.

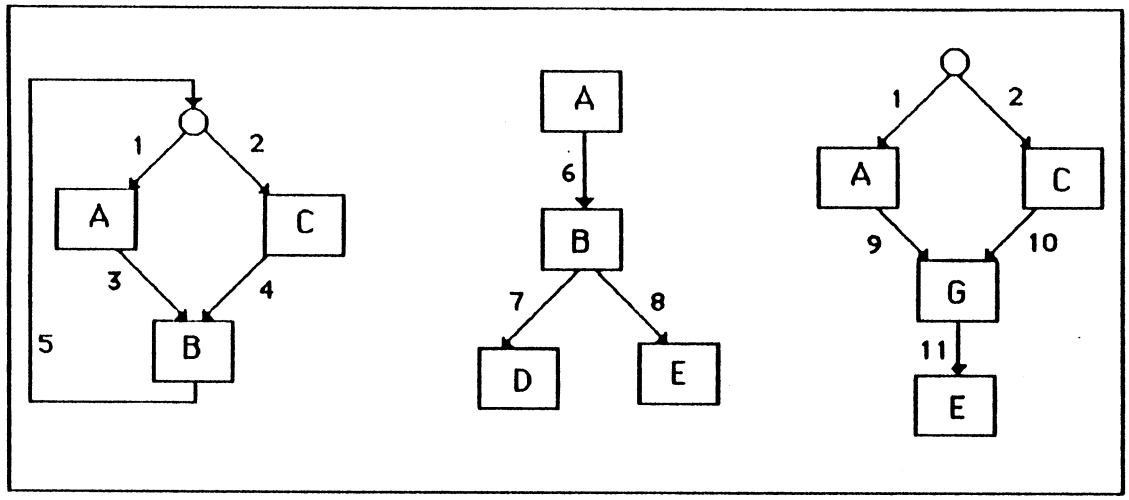


fig. 4.16 : étage de contrôle avec des parties d'automates identiques

Les parties d'automates identiques possèdant une seule entrée et une seule sortie sont A, C et E; par conséquent, l'insertion de procédure est donc possible et cet étage de contrôle peut être décomposé en deux étages de contrôle représentés dans la figure suivante :

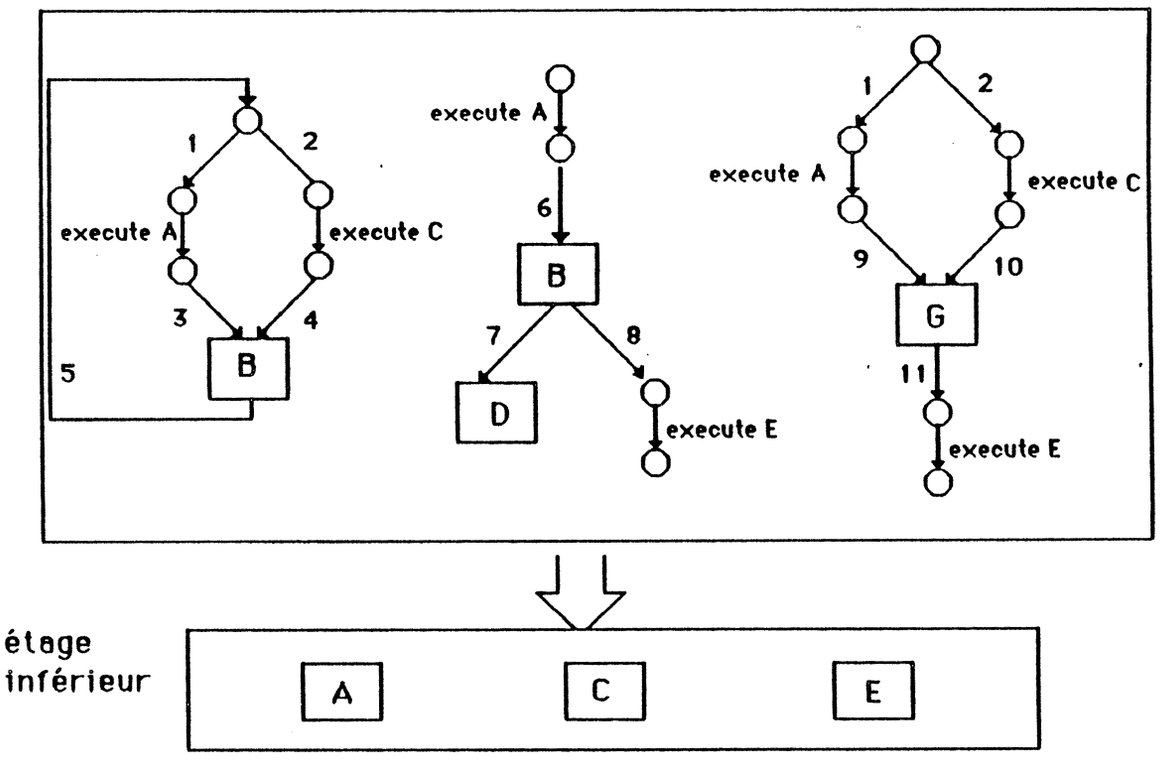


fig. 4.17 : insertion de procédures

Le problème est donc de détecter à l'intérieur des différentes procédures interprétées par un étage de contrôle, toutes les séquences d'instructions identiques (s'il en existe) et qui possèdent un seul point d'entrée et un seul point de sortie. La seule solution efficace pour résoudre ce problème est de faire une analyse combinatoire. Si le nombre d'états de l'étage de contrôle est élevé, cette solution nécessiterait un temps de calcul trop élevé.

Pour simplifier le problème, nous considèrerons seulement les séquences constituées d'instructions non-conditionnelles et dont seule la première est étiquetée. Ces séquences possèdent bien un seul point d'entrée (la première instruction) et un seul point de sortie (la dernière instruction).

Nous donnons dans ce qui suit, un algorithme qui permet de rechercher des séquences d'instructions identiques dans les procédures interprétées par un même étage de contrôle. Cet algorithme consiste à rechercher dans les diagrammes de transitions des différents automates d'un étage de contrôle, des chemins "identiques". Deux chemins notés $(a_1,...,a_n)$ et $(b_1,...,b_n)$, sont dits identiques s'ils ont le même nombre d'arcs et si chaque arc a_i porte la même étiquette que b_i (on suppose que 2 arcs portent la même étiquette s'ils correspondent à 2 instructions identiques ou équivalentes). Tous les noeuds de ces chemins doivent avoir un seul prédecesseur et un seul successeur; seul le premier noeud (point d'entrée) peut avoir plusieurs prédecesseurs.

7.1.3.1.1. Algorithme de recherche de séquences identiques

1) Rechercher les chemins de longueur 2 et qui apparaissent au moins 2 fois dans un même automate ou dans des automates distincts. Soit P_2 l'ensemble constitué de ces chemins. Si P_2 est vide alors arrêter, sinon i=2.

2) Constituer à partir de P_i l'ensemble P_{i+1} des chemins de longueur i+1 et dont le nombre d'occurences est supérieur ou égal à 2.

3) Réitérer en (2) jusqu'à ce que à une certaine itération n, l'ensemble P_{n+1} est vide. A cette phase toutes les séquences d'instructions dans P_2, P_3,...,P_n sont candidates pour être transformées en procédures.

4) Soit OC(Si) le nombre d'occurences d'une séquence d'instructions notée Si et L(Si) sa longueur. Transformer en procédures toutes les séquences dont le produit "OC(Si) . L(Si)" est maximal.

5) éliminer de P_2, P_3,...,P_n toutes les séquences qui ont au moins une instruction commune avec celles choisies pendant la phase (4).

6) Réitérer en (4) jusqu'à ce que tous les ensembles P_2, P_3,...,P_n soient vides.

7.1.3.1.2 Caractéristiques des étages de contrôle résultant de l'insertion de procédures

Soit un étage de contrôle E_i contenant m parties de code (notées PC_1,..., PC_m) pouvant être remplacées par des procédures. Supposons que E_i doit être décomposé en 2 étages E_{i1} et E_{i2} tels que E_{i2} soit constitué de PC_1,...,PC_m :

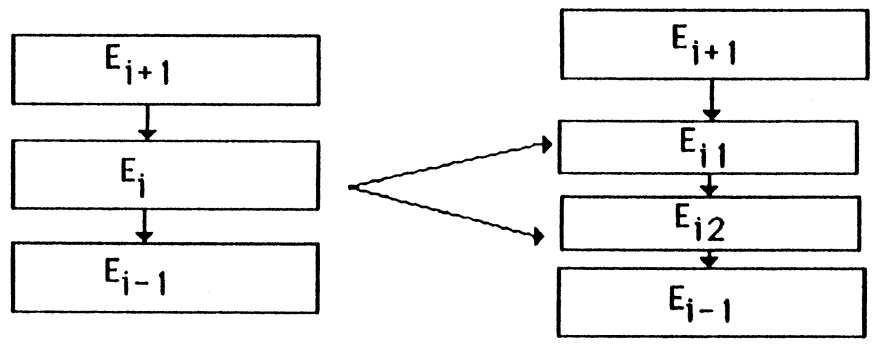


fig. 4.18 : décomposition de l'étage E_i en E_{i1} et E_{i2}

Pour préserver le comportement du système, il faut rajouter dans E_{i2} autant d'états

intermédiaires que de procédures de E_{i-1} activées par E_{i1}. Soit $N(PC_i)$ le nombre d'instructions dans la séquence PC_i. Alors le nombre d'états dans E_{i2} (noté $N(E_{i2})$) est égal à :

$$N(E_{i2}) = \sum_{i=1}^{m} N(PC_i) + h$$

avec : h = nombre de procédures de E_{i-1} activées par E_{i1}.

Dans l'étage E_{i1}, il faut rajouter des instructions d'appels aux séquences d'instructions remplacées par des procédures. Soit $OC(PC_i)$ le nombre d'occurences de la séquence d'instructions PC_i (avant la décomposition). Alors le nombre d'états de l'étage E_{i1} (noté $N(E_{i1})$) est égal à :

$$N(E_{i1}) = N(E_i) - \sum_{i=1}^{m} OC(PC_i) * N(PC_i) + \sum_{i=1}^{m} OC(PC_i)$$

($N(E_i)$ représente le nombre d'états de l'étage décomposé E_i).

Remarque : pour que l'insertion de procédure puisse être effectuée par le système, il faut que la condition suivante soit vérifiée :

$$N(E_{i1}) + N(E_{i2}) < N(E_i)$$

Si cette condition est vraie, le nombre d'états de l'algorithme d'interprétation des instructions diminue; il y donc moins d'états à mémoriser : ceci peut conduire à une diminution de la surface de la partie contrôle, si la surface totale occupée par les deux étages de contrôle et par leur interconnexion est inférieure à celle de l'étage décomposé. A ce niveau, une évaluation de surface est nécessaire pour calculer et comparer les surfaces des étages résultant de la décomposition avec celle de l'étage décomposé.

7.1.3.2. Répartition des procédures d'un même niveau en deux niveaux d'interprétation

Une autre alternative possible pour décomposer un étage de contrôle serait de répartir les procédures de cet étage en deux niveaux d'interprétation. Nous donnons l'algorithme de cette décomposition sur un exemple simple.

Soit à décomposer un étage E_i décrit par sa table de transitions (fig. 4.19) en deux

étages E_1 et E_2 tels que E_1 est constitué de l'automate d'état initial e1 et E_2 de l'automate d'état initial e2 :

Etat	Cond	Action	Next
e1		execute e'1	e1.1
e1.1		execute e'2	e1.2
e1.2	c1		e1
	non c1		exit
e2	c2	execute e'3	e2.1
	non c2		exit
e2.1		execute e'2	e2

fig. 4.19 : table de transitions correspondant à E_i (2 automates)

Après décomposition de cette table, on obtient les deux tables de transitions suivantes

Etat	Cond	Action	Next
e1		execute e'1i	e1.1
e1.1		execute e'2i	e1.2
e1.2	c1		e1
	non c1		exit
ei2		execute e2	exit

fig. 4.20 Table de transitions T1 correspondant à E_1

Etat	Cond	Action	Next
e2	c2 non c2	execute e'3	e2.1 exit
e2.1		execute e'2	e2
e'1i		execute e'1	exit
e'2i		execute e'2	exit

fig. 4.21 : table de transitions T2 correspondant à E_2

Les états suivants ont été rajoutés pour maintenir le comportement de l'étage décomposé :

- un état intermédiaire ei2 dans la table T1 pour propager l'appel à l'état e2,

- deux états intermédiaires e'1i et e'2i dans la table T2 pour propager l'appel à e'1 et e'2 .

7.1.3.2.1. Caractéristiques des étages de contrôle résultant de la répartition des procédures

Soit à décomposer un étage de contrôle E_i composé de n procédures P1, ..., Pn en deux étages E_{i1} et E_{i2} tels que :
- E_{i1} soit topologiquement au-dessus de E_{i2}
- $E_{i1} = \{ P_{11}, \ldots, P_{m1} \}$
- $E_{i2} = \{ P_{12}, \ldots, P_{k2} \}$

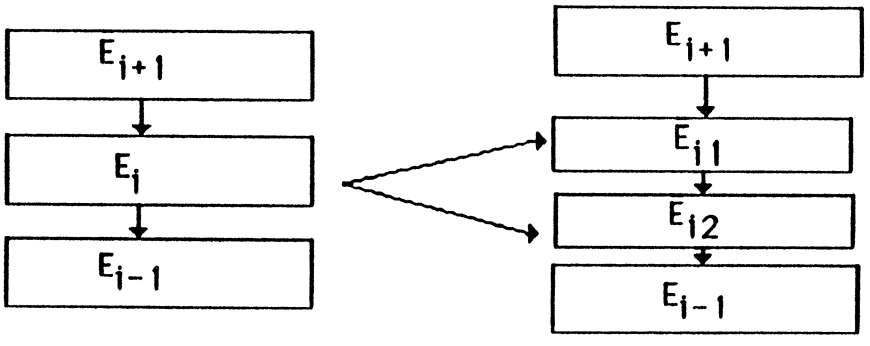


fig. 4.22 : répartition des procédures de E_i en 2 niveaux d'interprétation

Pour préserver le comportement du système il faut rajouter dans E_{i1} autant d'états intermédiaires que de procédures dans E_{i2}, afin de pouvoir propager à travers E_{i1} les appels aux procédures dans E_{i2}. De même, il faut rajouter autant d'états dans E_{i2} que de procédures de E_{i-1} activées par E_{i1}. On obtient donc les formules suivantes qui donnent le nombre d'états de E_{i1} et E_{i2}, notés respectivement $N(E_{i1})$ et $N(E_{i2})$:

$$(2) \quad N(E_{i1}) = \sum_{j=1}^{m} N(P_{j1}) + k$$

$$(3) \quad N(E_{i2}) = \sum_{j=1}^{k} N(P_{j2}) + h$$

avec : k = nombre de procédures dans E_{i2} ,
h = nombre de procédures de E_{i-1} activées par E_{i1}.

Remarques :

1) Si le nombre de procédures de l'étage à décomposer n'est pas trop élevé, une analyse combinatoire peut être faite pour trouver les sous-ensembles de procédures E_{i1} et E_{i2} qui donnent le meilleur compromis surface-vitesse. Les formules (2) et (3) sont utilisées pour calculer le nombre d'états des étages de contrôle E_{i1} et E_{i2} dans chaque cas de décomposition.

2) Si le nombre de procédures de l'étage à décomposer est élevé, les procédures maintenues dans l'étage supérieur E_{i1} seront celles qui activent le moins de procédures distinctes de l'étage inférieur E_{i-1}; ceci permet de réduire le nombre d'états intermédiaires rajoutés dans l'étage E_{i2} après la décomposition.

7.1.3.3.Influences sur les performances et la surface du circuit

Les conséquences de la décomposition d'un étage de contrôle sur la partie contrôle sont tout à fait l'inverse de celles de la fusion :

- une augmentation du nombre d'étages de contrôle et de la surface occupée par les connexions. Par conséquent, le temps de traversée de la partie contrôle est augmenté. Cette perte de performances pourrait être compensée par une diminution du temps de cycle d'horloge si l'étage décomposé fixait le temps de cycle critique.

- augmentation du nombre d'appels de procédures.

-l'insertion de procédures provoque une réduction du nombre d'états de l'algorithme d'interprétation des instructions, ce qui contribue à une réduction de la surface de la partie contrôle.
Par contre, la répartition des procédures en deux niveaux d'interprétation provoque une augmentation du nombre d'états de l'algorithme; en effet, d'après les formules (2) et (3), on obtient :

$$N(E_{i1}) + N(E_{i2}) > N(E_i)$$

Cette augmentation implique une augmentation de la surface de la partie contrôle.

La répartition des procédures d'un étage de contrôle en 2 niveaux d'interprétation n'a donc pas un grand intérêt au contraire de l'insertion de procédures. Par contre, elle peut être effectuée lorsque l'insertion de procédures ne donne aucun résultat satisfaisant; elle peut être utilisée, par exemple, dans le cas où le concepteur décrit l'algorithme d'interprétation du microprocesseur en un seul niveau, ce qui peut conduire à un gros PLA impossible à générer automatiquement (vues les limites des logiciels de génération automatique de dessin des masques des PLAs), ou dont les performances sont très faibles dû à un temps de propagation trop élevé.

7.1.4. Combinaisons d'actions de fusion et de décomposition

Parfois une combinaison d'actions de fusion et de décomposition d'un étage de contrôle peut conduire à un meilleur compromis surface-vitesse. Par exemple, si on ne peut pas fusionner directement deux étages de contrôle, il faudrait décomposer ces gros étages de contrôle pour obtenir des étages plus petits. Ces derniers peuvent ensuite être fusionnés plus facilement et conduire à une meilleure implantation :

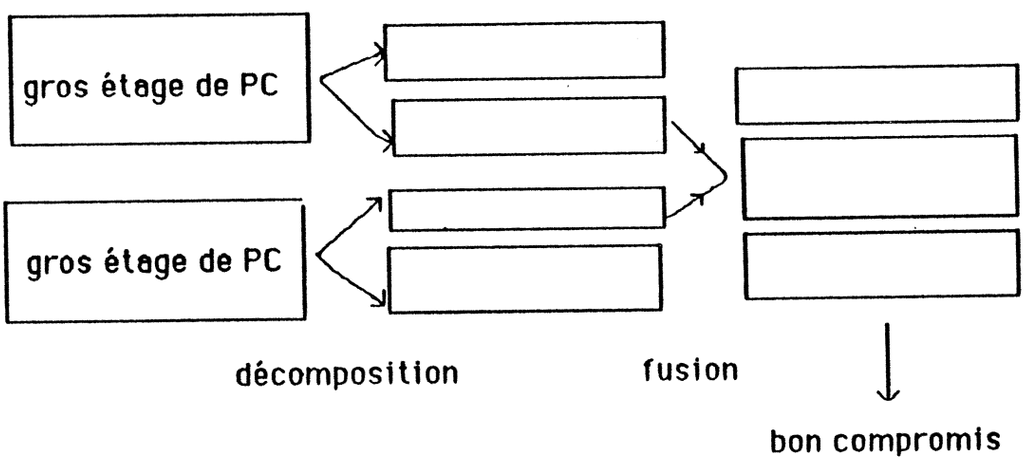


fig. 4.23 : combinaisons "fusion-décomposition"

7.2. Transformations de la description comportementale de la partie opérative

Pour illustrer les types de compromis qui doivent être considérés au niveau transfert de registre et la façon dont ceci interagit avec le niveau matériel, considérons l'exemple ci-dessous :

```
<e1> :  ADR := ADR + X ; next e2 ;
<e2> :   B  := M[ADR] ; next e3 ;
<e3> :   A  := A + B ; next e4 ;
<e4> :  ADR := ADR + 1 ; exit ;
```

Ces quatre lignes sont tirées d'une description comportementale d'un processeur hypothétique écrite dans le langage de description LDS. Les crochets dans la ligne (2) signifient que M est un tableau et que le mot dans M indexé par ADR est ajouté à A. Le "next e2" dans la ligne (1) indique que la ligne (2) suit la ligne (1) dans l'ordre d'exécution, par conséquent la valeur originale de ADR est ajoutée à la valeur de X avant d'être utilisée comme indice.

La figure 4.24 montre une implantation possible pour le comportement donné :

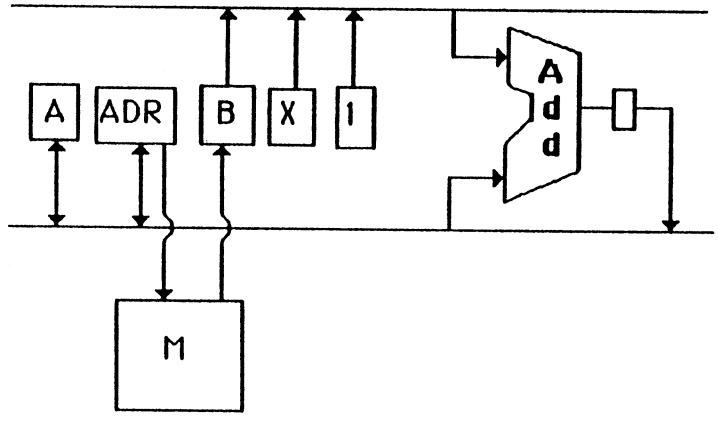


fig 4.24 : partie opérative avec un coût minimal

C'est la structure la plus simple possible, puisqu'il y a seulement une UAL, et deux bus. par conséquent, il n'y a pas de possibilité de parallélisme, et les opérations doivent être exécutées une après l'autre dans l'ordre spécifié.

Une autre configuration possible est donnée dans la figure 4.25. Dans ce cas, l'addition d'une autre sous-partie opérative avec un additionneur permet à ADR d'être incrémenté en parallèle avec le calcul de A. A ce niveau d'analyse, ceci semble être un simple compromis entre la taille et la vitesse du circuit. C'est ce type de décision que nous devons considérer à ce niveau de la conception.

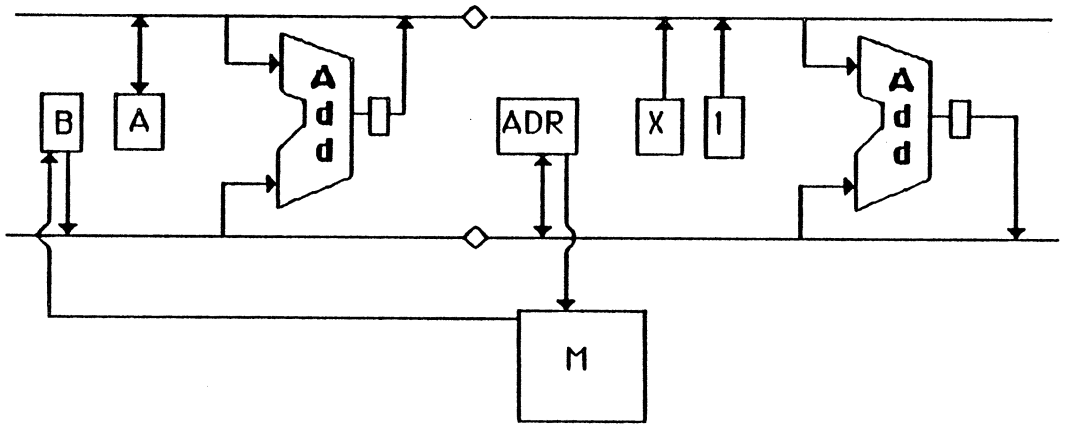


fig. 4.25 : partie opérative permettant le parallélisme

Si nous devons considérer quelques paramètres physiques, certaines complications peuvent surgir. Si par exemple, il est nécessaire de combiner ADR ou X avec A, le délai nécessaire pour transférer une de ces valeurs d'une sous-partie opérative vers l'autre peut conduire à augmenter la durée du cycle d'horloge. En effet, le temps de cycle machine minimum est limité par le temps nécessaire pour transférer une donnée d'un registre à l'autre dans la partie opérative [HENN 84].

Dans ce cas, la figure 4.24 serait la conception la plus simple puisqu'elle est la plus compacte. Ceci montre l'importance des détails physiques de conception sur la recherche d'un compromis "surface-vitesse".

La hauteur de la partie opérative étant supposée constante, les transformations de la description comportementale que nous allons décrire ne portent que sur la largeur de la partie opérative. Deux types de transformations sont à distinguer :
-celles qui conduisent à une réduction de la largeur de la partie opérative
-celles qui conduisent à une augmentation de la largeur de la partie opérative.

7.2.1. Réduction de la largeur de la partie opérative

7.2.1.1. Réduction par décomposition des instructions opératives

Si l'on ne veut pas dépasser une limite donnée pour la largeur de la partie opérative, il est nécessaire de réduire le parallélisme entre les actions opératives en décomposant certaines instructions opératives.
La largeur de la partie opérative dépend du nombre de registres, de constantes et d'UALs nécessaires pour exécuter toutes les instructions opératives dans la description comportementale. Dans SYCO, le nombre de registres et de constantes est fixé par le concepteur. Par conséquent, pour réduire la largeur de la partie opérative, il faut réduire le nombre d'UALs; au niveau comportemental, cette action revient à réduire le parallélisme entre les opérations binaires ou unaires.

Soit n le nombre d'UALs maximum que pourrait contenir la partie opérative de telle façon à ne pas dépasser une certaine largeur imposée par l'utilisateur. Il faut décomposer les instructions qui contiennent plus de n opérations (binaires ou unaires), en deux ou plusieurs instructions. Pour minimiser le nombre total d'instructions opératives distinctes, une instruction sera décomposée dans la mesure du possible en des instructions déjà existantes. En effet, le compilateur de parties opératives Apollon [JAMI 85] génère deux vecteurs de commandes (de la partie opérative) pour chaque instruction : un vecteur de commandes pour les transferts exécutés pendant la première phase phi1 du cycle d'horloge, et un deuxième vecteur pour les transferts exécutés pendant la deuxième phase phi2. Pour réduire la taille du PLA de génération de commandes de la partie opérative, il faut donc réduire le nombre d'instructions opératives distinctes.

Cet algorithme peut être exploité par le compilateur Apollon si ce dernier ne trouve aucune solution possible pour implanter les registres d'une instruction lors de la synthèse de la partie opérative. Ceci permet d'éviter une combinatoire très coûteuse en temps CPU.

Conséquences de la décomposition des instructions opératives sur le circuit :

La première conséquence, c'est une réduction de la largeur du circuit (on suppose que c'est la partie opérative qui fixe la largeur du circuit).

La décomposition des instructions opératives implique aussi une réduction de la vitesse du circuit. En effet, elle provoque une augmentation du nombre d'états de l'algorithme d'interprétation des instructions, d'où une augmentation de la surface de la partie contrôle, et notamment de l'étage de génération de commandes de la partie opérative. Si cet étage est implanté à l'aide d'un PLA, la décomposition implique une augmentation du nombre de monômes du PLA de génération de commandes de la partie opérative. Cette conséquence implique une augmentation de la hauteur ou de la largeur de la partie contrôle selon que le PLA de génération de commandes est un PLA classique disposé horizontalement ou verticalement. D'autre part, le nombre de fils de séquencement interne de ce PLA doit être augmenté; en effet, avant la décomposition toute instruction opérative consistait en deux ensembles de transferts : un exécuté en phi1, et l'autre en phi2; un seul fil de séquencement interne suffisait pour mettre les deux valeurs binaires 0 et 1 : le "0" par exemple, pour l'exécution des transferts pendant la phase phi1, et le "1" pour les transferts exécutés pendant phi2. Après la réduction de la largeur de la partie opérative, une instruction décomposée nécessitera au moins 4 phases. Il faut donc au moins deux fils de séquencement interne pour le PLA du dernier étage de contrôle.

Par conséquent, plus le nombre d'instructions à décomposer est grand, plus les pertes de performances seront élevées. Pour les autres étages de la partie contrôle tout est transparent, ils restent donc inchangés.

7.2.1.2. Réduction par compactage des instructions opératives

Si l'ordre d'exécution des instructions opératives est connu (graphe de précédence), au lieu de décomposer une instruction, il est possible de réduire la largeur de la partie opérative en "transférant" une (ou plusieurs) opération(s) de cette instruction vers l'instruction qui la suit ou vers celle qui la précède.

exemple :

Soit la description comportementale suivante écrite dans le langage LDS :

<e1> : A := B + C ; E := D - F ; H := not X ; Next e2 ;
<e2> : B := H ; exit;

Soit n = 2 le nombre maximum d'UALs que peut contenir la partie opérative pour ne pas dépasser une certaine largeur imposée par l'utilisateur.

L'instruction e1 contient 3 opérations : elle nécessite 3 UALs pour son implantation par le compilateur Apollon. Au lieu de décomposer cette instruction, il est préférable de transférer une opération de cette dernière vers l'instruction e2; par exemple, l'opération "E := D - F" peut être exécutée en parallèle avec le transfert "B := H". On obtient :

<e1'> : A := B + C ; H := not X ; Next e2' ;
<e2'> : B := H ; E := D - F ; exit ;

Remarque : dans cet exemple, on suppose que l'instruction e2 n'est référencée par aucune autre instruction; elle peut donc être modifiée sans altérer le comportement du système.

L'avantage de cette solution par rapport à la décomposition, c'est qu'elle ne provoque pas une augmentation du nombre d'instructions opératives; par conséquent, la largeur de la partie opérative est réduite sans augmenter la taille de la partie contrôle.

7.2.2. Augmentation de la largeur de la partie opérative

Il existe en gros 2 types de descriptions comportementales :

- soit le concepteur spécifie le parallélisme entre les actions opératives ainsi que leur séquencement (cas des systèmes SYCO, Mimola [ZIMM 80], de SILC [BLAC 85])

- soit c'est le compilateur de silicium qui établit et optimise le séquencement des actions opératives à partir du séquencement établi par le concepteur (cas du CMU-DA [SNOW 78]) [DIRE 81], [PARK 85], [PARK 86]).

Dans ce dernier cas, le concepteur est alors libre de décrire des expressions complexes. C'est le compilateur de silicium qui décompose ces expressions complexes en une séquence d'opérations tout en assignant les résultats intermédiaires à des registres. Une fois cette décomposition faite, le compilateur localise automatiquement les sous-ensembles d'actions opératives pouvant être exécutées en parallèle et les rassemble sous forme de microinstructions.

Dans le compilateur SYCO, nous considérons le cas où le parallélisme spécifier par le concepteur est insuffisant et conduirait à un circuit dont les performances sont faibles. Dans ce cas, il faut fusionner entre elles deux ou plusieurs instructions opératives pour augmenter le parallélisme. Par conséquent, l'ordre d'exécution des instructions doit être connu. Deux instructions peuvent être fusionnées si elles vérifient les contraintes suivantes :
-contrainte de précédence : aucun registre source dans l'une ne doit figurer comme destination dans l'autre, et réciproquement.
-contrainte de surface : la nouvelle instruction résultant de la fusion doit vérifier les contraintes de surface imposées par le concepteur.

exemple : soit les 2 instructions suivantes :
<e1> : A := B + C ; H := I ; NEXT e2 ;
<c2> : G := J - K ; NEXT ek ;

Si les contraintes de surface sont telles que la partie opérative peut au maximum contenir 2 UALs, alors e1 et e2 ne peuvent être fusionnées puisque cela conduirait à une partie opérative avec 3 UALs (le compilateur Apollon génère autant d'opérateurs que le nombre maximal d'opérations simultanées) .

-enfin, la dernière contrainte est due aux limites du modèle architectural de la partie opérative. Il faut donc que l'instruction résultant de la fusion puisse être exécutée par ce modèle.

exemple : supposons que après fusion de 2 instructions, on obtient une autre de la forme suivante :

? := A + B ; ? := B + C ; ? := C + A

peu importe les destinations; les opérandes de cette instruction ne peuvent être chargés simultanément sur une partie opérative à 2 bus segmentés [JAMI 86].

Remarque : si les fréquences d'exécution des différentes instructions opératives sont spécifiées par le concepteur, les instructions fusionnées en premier seront celles dont la fréquence est maximale.

Cette transformation de la description peut être classée dans la rubrique "optimisation" ou dans la rubrique "compromis surface-vitesse" selon qu'elle est

appliquée après la construction de la partie opérative ou dès le début de la compilation. Dans le premier cas, il s'agit d'augmenter le parallélisme afin d'exploiter au maximum les ressources de la partie opérative, dans le second cas il s'agit d'augmenter le parallélisme afin de trouver un compromis surface-vitesse qui satisfait les contraintes de l'utilisateur.

Conséquences sur les performances du circuit :

La fusion des instructions opérative implique une réduction du nombre d'états de l'algorithme. En effet, lorsqu'on augmente la largeur de la partie opérative, on augmente le nombre des commandes qu'elle reçoit à chaque cycle, et alors le nombre total de cycles nécessaire à l'exécution de l'algorithme diminue. Sur le plan matériel cette réduction se traduit par une réduction du nombre de monômes du PLA de génération de commandes de la partie opérative d'où une réduction de la hauteur ou de la largeur de la partie contrôle selon que le PLA de génération de commandes est un PLA classique vertical ou horizontal. Pour les autres étages de la partie contrôle cette action implique une réduction éventuelle du nombre d'états.

8. Conclusion

Le compromis surface-vitesse se résume de la manière suivante :

Lorsqu'on augmente les performances de la partie opérative (c.a.d sa largeur), on augmente le nombre de commandes qu'elle reçoit à chaque cycle, et alors le nombre total de cycles nécessaire à l'exécution de l'algorithme diminue : la performance globale augmente. La partie contrôle a donc une complexité horizontale (nombre de commandes) qui augmente, et une complexité verticale (nombre de cycles) qui diminue. Si, par contre, le circuit est trop large, il faut sacrifier des performances en gagnant une meilleure forme.

[SCHO 85] fait remarquer que dans le cas général, un circuit plus large que haut a forcément :
-soit une complexité intrinsèque réduite et un nombre de bits élevé (c'est le cas de la puce RISC-II de UC Berkeley),
-soit une complexité intrinsèque importante et un nombre de bits réduit; la même forme de puce peut accueillir deux ordinateurs très différents (un 16 bits très complexe ou un 32 bit très simple)

En ce qui concerne le nombre de niveaux d'interprétation, le compromis surface-vitesse s'exprime selon deux ordres :
- plus le nombre de niveaux d'interprétation est important, plus le temps d'exécution d'une instruction de niveau le plus élevé augmente.
- plus le nombre de niveaux est faible, plus la surface respective des différents niveaux de contrôle augmente, ayant pour effet d'augmenter le temps de cycle d'horloge. Par conséquent, si l'on veut un circuit complexe rapide et de surface

faible, il est nécessaire de disposer d'un nombre d'étages réduit. Par exemple, dans les processeurs existants sur le marché, la limitation en nombre de niveaux de contrôle est de deux à trois niveaux (le Z80 de chez ZILOG ou le 8085).

Dans le cadre de cette thèse, les transformations décrites dans ce chapitre ont été implémentées sous forme d'un ensemble de programmes écrits dans le langage LISP. Ces programmes constituent un logiciel faisant partie de l'environnement SYCO et qui s'appelle "ARTS" (Architectural Trade-off System). Le système ARTS est constitué de plusieurs modules, chaque module réalisant une transformation de la descripion comportementale. Ce système est implanté actuellement sur SM-90 sous système UNIX, et utilise l'environnement LDS. L'appel de ce système provoque l'affichage sur l'écran d'un menu contenant toutes les transformations décrites dans ce chapitre. Le concepteur peut ainsi choisir la transformation qu'il désire effectuée; il doit fournir préalablement au système certaines informations. Par exemple, dans le cas de la fusion de deux étages de contrôle, il doit donner les noms de ces étages ainsi que le nom du circuit.

Le système ARTS opère sur la (ou les) table(s) de transitions spécifiée(s) en entrée et génère en sortie une ou plusieurs tables de transitions selon que la transformation effectuée est une fusion ou une décomposition.

Les tables de transitions générées peuvent ensuite être utilisées par les autres outils du compilateur SYCO, tel que le programme d'optimisation (décrit dans le chapitre 3); ce programme doit d'ailleurs être systématiquement appelé après chaque transformation pour éliminer les instructions inaccessibles ou redondantes résultant de l'application de cette transformation.

La figure suivante donne un aperçu global de l'environnement du système ARTS :

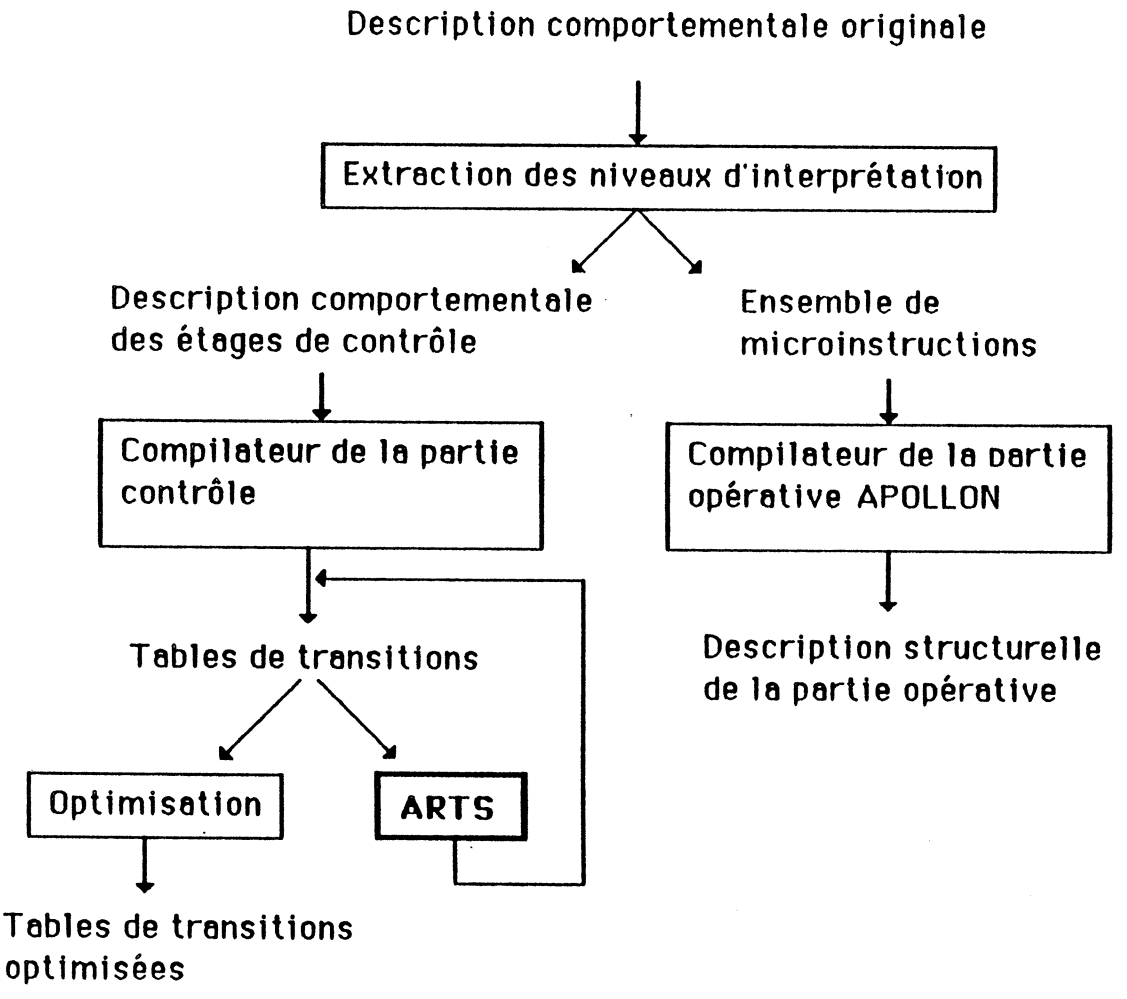


fig. 4.26 : environnement du système ARTS

Pour mesurer le changement provoqué par une transformation et pour aider l'utilisateur à choisir la transformation à appliquer, le système ARTS génère aussi certaines informations. Par exemple, dans le cas de la fusion de deux étages de contrôle, il calcule le nombre de monômes du PLA implémentant l'étage de contrôle résultant de la fusion, ainsi que le gain de vitesse. Le gain de vitesse est estimé en calculant pour chaque procédure de l'étage de contrôle supérieur le nombre maximal d'instructions EXECUTE consécutives qui ont été éliminées après la fusion de cet étage avec l'étage inférieur.

Pour explorer la faisabilité de guider automatiquement le processus de transformation, une commande a été implantée pour exécuter automatiquement l'expansion de procédures. Cette commande permet de fusionner deux étages de contrôle dans le cas où toutes les procédures de l'étage inférieur sont appelées une seule fois par l'étage supérieur.

EVOLUTION FUTURE

Evolution future

De cette étude, nous avons déduit les propositions suivantes afin d'améliorer le compilateur SYCO.

1) Une première évolution du système SYCO serait de permettre à un CMODULE d'appeler n'importe quel CMODULE de niveau inférieur (actuellement, un CMODULE ne peut appeler que les CMODULEs du niveau directement inférieur). Au niveau matériel, cette amélioration consiste à permettre le passage d'une ou plusieurs lignes d'informations à travers un PLA sans modification de sa fonctionnalité (transparence d'un étage de contrôle). L'introduction de la transparence d'un étage de contrôle a plusieurs avantages :

- n'importe quelle tranche de contrôle pourrait activer directement la partie opérative, en particulier l'étage de contrôle de plus haut niveau. Par conséquent, la vitesse du circuit est augmentée puisque le temps de traversée de la partie contrôle est diminué.
- la recherche d'un compromis surface-vitesse sera facilité. En effet, pendant la décomposition d'un étage de contrôle aucun état intermédiaire ne sera rajouté dans les étages de contrôle résultant de la décomposition d'un gros étage de contrôle. Le nombre d'états de l'algorithme reste donc inchangé après la décomposition ou après la fusion partielle de 2 étages de contrôle. Cette évolution pourrait conduire aussi à une automatisation complète du processus de recherche d'un compromis surface-vitesse.
- enfin, l'algorithme de traduction de SYCO sera simplifié puisque la phase de la compilation qui consiste à rajouter des CMODULEs intermédiaires pour propager les appels à travers les étages de contrôle n'est plus nécessaire. Le nombre d'états de l'algorithme d'interprétation des instructions n'est donc pas augmenté.

2) Actuellement les parties contrôles générées sont de type monoPLA. Pour accroître les possibilités du compilateur SYCO et lui fournir beaucoup plus d'alternatives pour la recherche d'un compromis surface-vitesse, il faudrait permettre d'autres choix dans l'implantation d'un étage de contrôle, tels que ROM, ROM+PLA, etc...

3) Au niveau synchronisation, une évolution possible du sytème SYCO serait de maintenir plusieurs tranches de contrôle actives au même cycle. Le but est de "pipeliner" l'exécution de façon à ce que la partie opérative soit utilisée au maximum. Actuellement, le modèle de séquencement est tel que à chaque cycle une seule tranche est active. Lorsqu'une tranche de contrôle lance l'exécution d'une procédure, elle doit attendre que l'exécution de cette procédure par la tranche inférieure soit terminée avant de lancer l'exécution d'une autre procédure ou de repasser le contrôle à la tranche supérieure. Ce mécanisme de synchronisation est simple, mais n'est pas très efficace car plus le nombre d'étages de contrôle est élevé, plus la partie opérative reste inactive.

Pour accélérer le déroulement des instructions, il faut pouvoir générer le signal de fin d'exécution d'une procédure non pas à la fin de l'exécution mais un ou plusieurs cycles avant. Cette amélioration est en cours de réalisation [GERO 87].

La dernière amélioration possible du système SYCO concerne le compilateur de parties opératives Apollon. Actuellement, ce compilateur ne permet pas d'effectuer des opérations sur un nombre restreint de bits d'un registre. C'est pourquoi les opérations de ce type sont effectuées comme des opérations logiques dont les opérandes sont un registre et une constante de masquage. C'est pour cette raison qu'un nombre assez important de constantes sont dupliquées pendant la génération de la partie opérative [JAMI 86]. L'introduction d'opérations portant sur un nombre quelconque de bits permettra d'une part de diminuer le nombre des constantes utilisées dans la partie opérative et d'autre part d'augmenter la vitesse d'exécution de ces opérations.

CONCLUSION

Dans cette thèse, une étude des transformations possibles de la description comportementale pour la recherche automatique d'un compromis surface-vitesse, a été menée. Le besoin d'une phase d'optimisation dans le compilateur SYCO a été montré. Cette phase constite essentiellement à :

- réduire le nombre d'états et de transitions de l'algorithme d'interprétation des instructions,
- optimiser le nombre de ressources de la partie opérative.

La réduction du nombre d'états de l'algorithme d'interprétation des intructions est réalisée en détectant et en fusionnant les instructions qui peuvent être exécutées en parallèle. Ces intructions peuvent être soit des actions de contrôle, soit des actions opératives. Par conséquent, la fusion de ces 2 types d'instructions a été présentée séparément. La réduction du nombre d'états a une grande influence sur la vitesse du circuit, en particulier la réduction du nombre d'instructions opératives est très importante; en effet, dans une partie contrôle multiniveau comme celle utilisée dans SYCO, une commande de transfert dans la partie opérative doit traverser tous les étages de la hiérarchie avant que la commande suivante de même niveau ne puisse être exécutée.

L'optimisation du nombre de ressources de la partie opérative est effectuée en utilisant des techniques d'optimisations similaires à celles employées par les compilateurs optimiseurs de langages de programmation évolués, telles que l'allocation des registres aux variables algorithmiques, la propagation des constantes, l'élimination des opérateurs redondants, etc...

La recherche d'un meilleur compromis surface-vitesse consiste par contre à échanger une perte de performance contre un gain de surface ou vice-versa. Pour augmenter la vitesse du circuit il faut augmenter le parallélisme entre les actions opératives et réduire le nombre de tranches de contrôle en remplaçant les appels de certaines procédures par leurs codes. Pour réduire la surface du circuit, il faut réduire le parallélisme entre les actions opératives (si la partie opérative est trop large), et essayer de décomposer les tranches de contrôle qui occupent une grande surface en des tranches de contrôle plus petites.

La recherche d'un compromis surface-vitesse est un problème plus difficile que celui de l'optimisation, et nécessite en général l'intervention du concepteur si le compilateur ne trouve aucune alternative possible pour résoudre ce compromis; dans ce cas, le concepteur peut soit réécrire la description comportementale sous une autre forme, soit modifier les contraintes de surface (largeur et hauteur du circuit) imposées au compilateur de silicium.

La façon dont l'utilisateur écrit la description comportementale du circuit a donc une grande influence sur la surface et la vitesse du circuit. SYCO permet déjà au concepteur de considérer le problème du compromis surface-vitesse; en effet, le

concepteur peut spécifier explicitement le parallélisme entre les actions opératives ainsi que leur séquencement. Par exemple, si la description contient une instruction avec 2 additions exécutées en parallèle, la partie opérative du circuit final contiendra 2 additionneurs. D'autre part, le nombre d'étages de contrôle est égal au nombre de niveaux d'interprétation dans la description comportementale. L'avantage de ces spécifications explicites (parallélisme + nombre de tranches de contrôle) est que l'utilisateur a un grand contrôle sur le système et l'algorithme de traduction du compilateur SYCO devient simple.

L'étude des transformations possibles de la description comportementale s'est concrétisée par la réalisation d'un logiciel appelé ARTS (Architectural Trade-off System), constitué de plusieurs modules intégrés autour d'une structure de données LDS. Chaque module réalise une des transformations suivantes :
-fusion de 2 étages,
-décomposition d'un étage,
-fusion partielle avec l'étage inférieur,
-fusion partielle avec l'étage supérieur.

Le concepteur peut tester les résultats d'une ou plusieurs transformations successives avec des retours en arrière possible.

ANNEXE I

APPLICATIONS SUR DES EXEMPLES

1. Exemples de recherche d'un compromis surface-vitesse

Pour évaluer l'utilité des transformations développées dans cette thèse, deux exemples ont été choisis et analysés. Le premier exemple est celui d'un microprocesseur simplifié (appelé "MICROP") exécutant un jeu d'instructions réduit. Le deuxième exemple est celui du 6502.

1.1. Cas d'un microprocesseur simplifié

La description de ce microprocesseur a été établie à partir d'un exemple donné dans [ANCE 82] : Le programme LDS ci-dessous est constitué de 2 parties :
-d'une part, d'une partie déclaration de ressources, incorporée dans un module structurel SMODULE,
-et d'autre part d'un ensemble de modules comportementaux CMODULEs inclus dans un CMODULE global.

```
SMODULE MICROP(adbus,dtbus,dtack,restart,adstrobe,dtstrobe,read_write);

VARIABLE;
r 0:2, CTRL;
END;

SIGNAL;
 adbus 0:8, OUT;          ? address bus
 dtbus 0:8, INOUT;        ? data bus
 dtack, IN, CTRL;         ? data acknowledgement
 restart, IN, CTRL;
 adstrobe, OUT, CTRL;     ? address strobe
 dtstrobe, OUT, CTRL;     ? data strobe
 read_write, OUT, CTRL;   ? selection read=1 / write=0
END;

REGISTER;
 a 0:8;                   ? accumulator
 ad 0:8;                  ? address register
 b 0:8;                   ? special purpose register
 c 0:8;                   ? special purpose register
 d 0:8;                   ? special purpose register
 ir 0:8;                  ? instruction register
 pc 0:8;                  ? program counter
 streg 4:4;               ? status register
 car, LIKE streg 1        ? carry bit
 neg, LIKE streg 2        ? negative bit
 ovf, LIKE streg 3        ? overflow bit
 zer, LIKE streg 4        ? zero bit
END;

LINK MICROP;

END;

?*********************************************
```

```
CMODULE MICROP;

<rest> IF (restart = 0) NEXT rest; ELSE NEXT start; END;

<start> (pc := 0; NEXT newinstr;)

<newinstr>
 (r := 0; EXECUTE fetch;)
 IF (ir 4:2 = '00') NEXT decode; ELSE (r :=3; EXECUTE fetch;) END;
 (CASE (ir 4:2
          WHEN ('01') ;
          WHEN ('10') ad := pc + ad;
          WHEN ('11') ad := pc - ad;
 END;
 NEXT decode;)

<decode>
 (CASE (ir 0:4)
          WHEN ('0000') EXECUTE ada;          ? a <- a + dtbus
          WHEN ('0001') EXECUTE lda;          ? a <- dtbus
          WHEN ('0010') EXECUTE sta;          ? memory <- a
          WHEN ('0011') EXECUTE cma;          ? compare dtbus to a
          WHEN ('0100') EXECUTE beq;          ? jump if dtbus=A
          WHEN ('0101') EXECUTE bgt;          ? jump if dtbus>A
          WHEN ('0110') EXECUTE bra;          ? unconditional jump
          WHEN ('0111') EXECUTE div;          ? division of "a" by dtbus
          WHEN ('1xxx') EXECUTE mva;          ? a <- b
 END;
 IF (restart = 0) NEXT rest; ELSE NEXT newinstr; END;)

END;

CMODULE ada;
<e1>
 (r := 2; EXECUTE fetch;)
 a := a + b;
END;

CMODULE lda;
<e1> (r := 1; EXECUTE fetch;)
END;

CMODULE sta;
<e1>
 (adbus := ad; RESET read_write; dtbus := a;)
 (adbus := ad; SET adstrobe; dtbus := a; SET dtstrobe; NEXT e2;)
<e2>
 IF (dtack = 0)
 (adbus := ad; dtbus := a; NEXT e2;)
 ELSE (RESET adstrobe; RESET dtstrobe;) END;
END;

CMODULE cma;
<e1>
 (r := 2; EXECUTE fetch;)
 BOP(, b, -, a, streg 4:4);
END;
```

```
CMODULE beq;
<e1> IF (zer = 1) pc := ad; END;
END;

CMODULE bgt;
<e1> IF ((neg = 0) AND (zer = 0)) pc := ad; END;
END;

CMODULE bra;
<e1> pc := ad;
END;

CMODULE div;
<e1>
 (r := 2; EXECUTE fetch;)
 (c := b; NEXT e2;)
<e2>
 BOP(, c, >, a, streg 4:4);
 IF ((neg=0) AND (zer=0)) NEXT e3; END;
 (c := c * 2; NEXT e2;)
<e3>
 (d := 0; NEXT e4;)
<e4>
 BOP(, c, <=, b, streg 4:4);
 IF ((neg=1) OR (zer=1)) NEXT e5; END;
  (d := d * 2; c := c / 2;)
 BOP(, a, <, c, streg 4:4);
 IF (neg=1) NEXT e4; END;
 (d := d + 1; a := a - c;)
 NEXT e4;
<e5>
 b := d;
 ? quotient in b, remainder in a
END;

CMODULE mva;
<e1> a := b;
END;

CMODULE fetch;
<send>
 (adbus := pc; SET read_write;)
 (adbus := pc; SET adstrobe; NEXT receive;)
<receive>
 IF (dtack = 0) (adbus := pc; NEXT receive;)
 ELSE
 (CASE (r)
        WHEN (0) ir := dtbus;
        WHEN (1) a := dtbus;
        WHEN (2) b := dtbus;
        WHEN (3) ad := dtbus;
 END;
 pc := pc + 1;
 RESET adstrobe;)
 END;
END;
?*********************************************
```

Le CMODULE fetch est une procédure de lecture d'un mot mémoire. Le CMODULE sta décrit le séquencement des actions opératives et des actions de contrôle nécessaires à l'écriture du contenu d'un registre en mémoire externe. Le signal "read-write" indique à la mémoire que l'on désire effectuer une lecture ou une écriture. Le signal "adstrobe" (resp. "dtstrobe") indique à la mémoire qu'une adresse (resp. donnée) est disponible sur le bus d'adresses (resp. de données) . Enfin, le signal "dtack" indique au microprocesseur que des données sont disponibles sur le bus de données (lecture d'un mot mémoire) ou que les données envoyées par le microprocesseur ont été écrites en mémoire.

Avant d'extraire les descriptions comportementales des différents étages (POP + PC) constituant le circuit, le compilateur SYCO transforme la description originale du circuit de telle façon qu'un CMODULE de niveau i ne puisse appeler qu'un autre de niveau directement inférieur i-1. La figure ci-dessous représente l'arbre d'appels des CMODULEs après la transformation de la description. Chaque niveau de cet arbre sera compilé dans une tranche de contrôle. On remarque que des CMODULEs intermédiaires ont été rajoutés dans le niveau 2 afin de pouvoir appeler la partie opérative à partir du CMODULE principal "MICROP". Ces CMODULES sont notés "et1", "et2" et "et3". Le CMODULE "FETCH2" a été rajouté uniquement pour pouvoir appeler le CMODULE "FETCH" (déjà présent dans la description originale) à partir de l'étage 3.

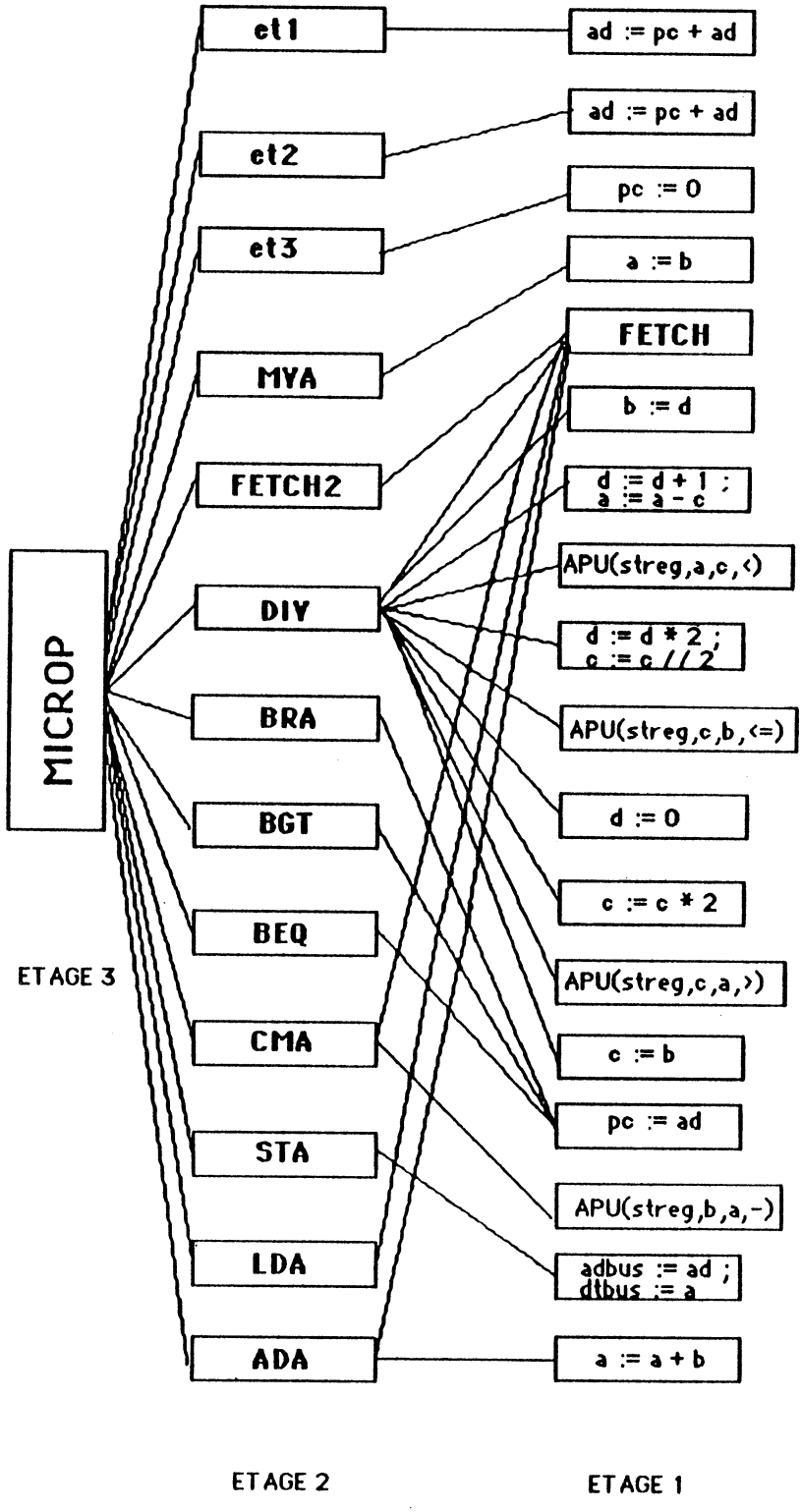


fig.1 : arbre d'appels des CMODULEs

état	condition	suivant	action
rest	restart = 0	rest	
rest	restart = 1	start	
start		newinstr	et1
newinstr		newinstr.1	FETCH2; r:=0
newinstr1	ir4:2 = 00	decode	
newinstr1	ir4:2 = x1	newinstr2	FETCH2; r:=3
newinstr1	ir4:2 = 1x	newinstr2	FETCH2; r:=3
newinstr2	ir4:2 = 01	decode	
newinstr2	ir4:2 = 10	decode	et2
newinstr2	ir4:2 = 11	decode	et3
decode	ir4:2 = 0000	decode1	ADA
decode	ir4:2 = 0010	decode1	STA
decode	ir4:2 = 0001	decode1	LDA
decode	ir4:2 = 0011	decode1	CMA
decode	ir4:2 = 0100	decode1	BEQ
decode	ir4:2 = 0101	decode1	BFT
decode	ir4:2 = 0110	decode1	BRA
decode	ir4:2 = 0111	decode1	DIV
decode	ir4:2 = 1xxx	decode1	MVA
decode1	restart = 0	rest	
decode1	restart = 1	newinstr	

fig. 2 : table des micro-opérations de l'étage 3

Avant de présenter les résultats de l'expansion de procédure, nous donnerons d'abord les résultats de l'optimisation de la description comportementale originale du microprocesseur MICROP.

Optimisation :

Descriptions des niveaux de parties contrôle :

A partir des tables de transitions (non optimisées) générées par le compilateur de partie contrôle [MHAY 86] on déduit les informations suivantes pour les 3 étages constituant la partie contrôle :

Le troisième étage (étage le plus haut) est constitué de :

22 états (chaque état correspond à une ligne de la table de transitions)
Nombre de fils de séquencement interne := Log_2 (3) = 2
Nombre de fils de contrôles entrants :=1 (variable de contrôle "restart")
Nombre de fils de contrôles sortants:= 2 (variable de contrôle "r");
Nombre de fils de compte-rendu := 6 (registre "IR 0:6")
Nombre de commandes pour le niveau inférieur := Log_2 (13) := 4
Nombre de commandes du niveau supérieur := 1 (cet étage est constitué d'un seul automate).

Le deuxième étage est constitué de :

40 états
Nombre de fils de séquencement interne := Log_2 (15) := 4
Nombre de fils de contrôles entrants :=1 (variable de contrôle "dtack") ;
Nombre de fils de contrôles sortants:= 10 (variables de contrôle read-write, r(0), r(1), adstrobe, dtstrobe);
Nombre de fils de compte rendu := 2
Nombre de commandes pour le niveau inférieur := Log_2 (20):= 5
Nombre de commandes du niveau supérieur := Log_2 (14) := 4 (14 est le nombre d'automates interprétés par cet étage de contrôle).

Le premier étage (étage de génération de commandes de la partie opérative) est constitué de :

25 états
Nombre de fils de séquencement interne := Log_2(3) := 2;
Nombre de fils de contrôles entrants :=3 (dtack, r(0), r(1));
Nombre de fils de contrôles sortants:= 4 (read-write, adstrobe);
Nombre de fils de compte rendu := 0
Nombre de commandes pour le niveau inférieur := Log_2(23) :=5
Nombre de commandes du niveau supérieur := Log_2 (18) := 5 (18 est le nombre d'automates de cet étage de contrôle)

Le programme d'optimisation a permis de réduire le nombre d'états de ces étages de contrôle en éliminant essentiellement des instructions de branchement inutiles présentes dans la descriprion originale du microprocesseur. La table suivante donne le résultat de l'optimisation des 3 étages de contrôle :

	Nombre d'états avant	Nombre d'états après	Séquences éliminées	Temps CPU
étage 3	22	21	"start"	2.5
étage 2	40	34	"e3", "e5"	5
étage 1	25	24		3.5

fig. 3 : résultats de l'optimisation dans le microprocesseur MICROP

Fusions des étages de contrôle :

Les étages définis ci dessus sont très petits, et leur nombre important provoque une chute des performances préjudiciable. C'est pourquoi une opération de fusion de deux étages de contrôle est indispensable.

Fusion de l'étage 1 avec l'étage 2 :

En analysant la description comportementale du microprocesseur MICROP, on remarque que tous les CMODULEs de l'étage 1 (sauf la procédure FETCH) ne contiennent qu'une seule instruction opérative et ne sont appelés qu'une seule fois par l'étage supérieur. La procédure FETCH qui contient 7 instructions après la phase d'optimisation n'est appelée que 5 fois par l'étage supérieur. Par conséquent, l'étage 1 peut être fusionné avec l'étage 2, sans risque d'augmenter la surface de la partie contrôle. Cette fusion a donc été réalisée automatiquement par le système ARTS à partir des tables de transitions optimisées et a donné les résultats suivants :

	Entrées	Sorties	Etats
Etage 3	11	11	21
Etage 2	11	19	34
Etage 1	10	11	24

Fusion + optimisation →

	Entrées	Sorties	Etats
Etage 3	11	11	21
Etage12	22	16	64

fig. 4 : expansion de procédures dans le microprocesseur MICROP

Le nombre d'étages de contrôle a été réduit, mais le nombre d'états des 2 étages de contrôle après la fusion n'est pas équilibré. Il est hors de question d'arrêter la

recherche d'un compromis surface-vitesse à ce niveau puisque cela conduirait à un circuit pyramidal. Une deuxième fusion a donc été réalisée entre l'étage 3 et l'étage12 (étage résultant de la fusion précédente) :

Fusion de l'étage 3 avec l'étage 12 :

-tous les CMODULEs de l'étage12, sauf le CMODULE "fetch2" sont appelés une seule fois par l'étage 3. D'autre part, 4 CMODULEs de l'étage12 ne contiennent qu'une seule instruction opérative. Une expansion de procédures entre l'étage3 et l'étage 12 a donc été réalisée par le système ARTS et a donné le résultat suivant :

	Entrées	Sorties	Etats
Etage 3	11	11	21
Etage12	22	16	64

Fusion + optimisation →

	Entrées	Sorties	Etats
Etage123	30	18	87

fig. 5 : fusion des étages 3 et 12

Le nombre d'états de l'étage de contrôle résultant de la fusion n'est pas très élevé; par conséquent, l'algorithme d'interprétation du microproceseur MICROP peut être implanté dans une seule tranche de contrôle.
En réalité, le nombre d'états de cet étage est calculé en rajoutant aux nombre de lignes de la table de transitions associée à cet étage (c.a.d 87), le nombre d'instructions opératives contenues dans cette table et comportant au moins une opération (il y en a 11). En effet, une opération nécessite 2 phases (phi1 et phi2) pour son exécution : en phi1 chargement des opérandes vers les entrées de l'UAL , et récupération du résultat de l'opération en phi2. Pour chaque instruction opérative comportant au moins une opération, il faut donc 2 monômes dans le PLA de génération de commandes de la partie opérative pour mémoriser cette instruction. Par conséquent, si la partie contrôle est réalisée à l'aide d'un PLA classique, il faudrait 98 monômes pour mémoriser tous les états de l'algorithme d'interprétation des instructions du microprocesseur MICROP.

1.2. Le 6502 :

Dans ce paragraphe, nous présentons la recherche d'un compromis surface-vitesse lors de la compilation du 6502. La compilation du 6502 est présentée dans [REIS 86]. La description a été effectuée dans le langage de description de circuit LDS [LAUR 85] à partir des spécifications du W65C02 [WDC 83]. Le W65C02 est un microprocesseur 8 bits fabriqué en CMOS totalement compatible avec le 6502 NMOS.

Le 6502 a été choisi parce que d'une part c'est un exemple de circuit ayant la complexité d'un microprocesseur et d'autre part parce qu'il a été adopté comme référence pour comparer plusieurs compilateurs de silicium [EVAN 85]. [EVAN 85] donne les résultats de la compilation du 6502 par plusieurs compilateurs de silicium.

L'évaluation de la surface du circuit, à partir de la première version de la description comportementale, donne comme résultat une partie opérative avec 3 UALs et une partie contrôle constituée de 3 étages. Les tailles de ces étages de contrôle ont été estimées (en lambdas) dans le cas d'une technologie NMOS :

- étage 3 : largeur : 746 ; hauteur : 910
- étage 2 : largeur : 2689 ; hauteur : 1097
- étage 1 : largeur : 3741 ; hauteur : 2119
-partie opérative : largeur : 3600 ; hauteur :1200

La figure 6 montre une représentation du plan de masse du 6502 à 3 étages.

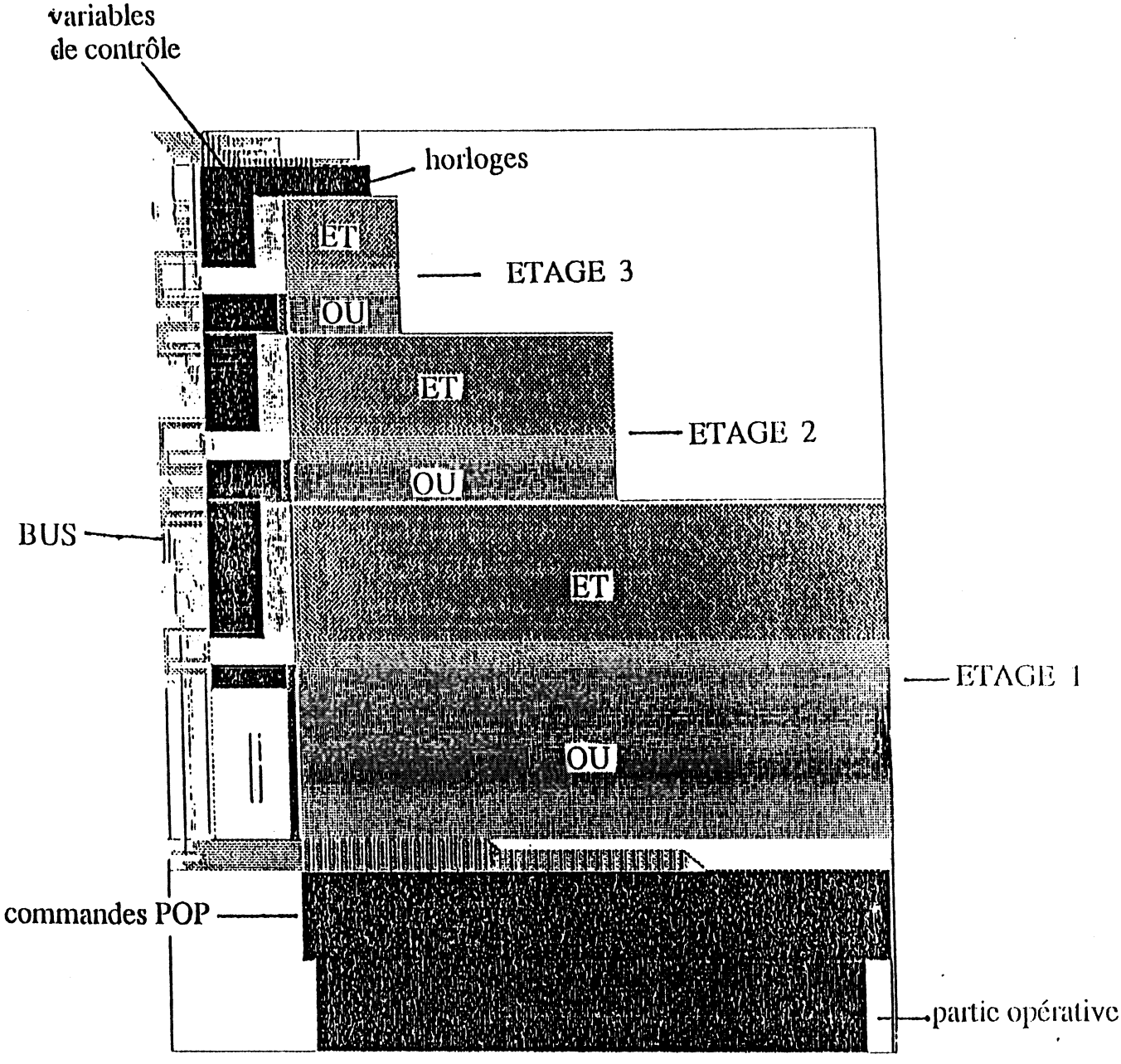


fig. 6 : plan de masse du 6502 avec 3 étages de contrôle

On remarque que plus une tranche de contrôle est proche de la partie opérative, plus elle a tendance à s'élargir : beaucoup de surface est donc perdue. Pour résoudre ce problème, l'expansion de procédures a été utilisée entre les étages 2 et 3 : les appels de CMODULEs dans l'étage 3 ont été remplacés par des copies des CMODULEs de l'étage 2. En effet, Les CMODULEs de l'étage 2 ne contiennent pas beaucoup d'instructions (en moyenne 3). Un grand nombre de ces CMODULEs ne sont appelés qu'une seule fois par l'étage supérieur. L'expansion de procédure est donc possible, et ne risque pas de conduire à une augmentation significative de la taille de la partie contrôle :

	Entrées	Sorties	Etats
Etage 3	36	28	57
Etage 2	52	24	160
Etage 1	76	28	246

fusion →

	Entrées	Sorties	Etats
Etage 32	54	31	224
Etage 1	76	28	246

fig. 7 : expansion de procédures dans le 6502

Le temps CPU nécessaire pour réaliser cette fusion est environ 6.60 secondes. Le gain de vitesse est de l'ordre de 16 cycles pendant l'exécution de la procédure (unique) constituant l'étage le plus haut.
On remarque que la fusion des étages 3 et 2 conduit à une partie contrôle avec des étages plus équilibrés. Ceci est dû au fait qu'un grand nombre de procédures interprétées par l'étage 1 ne sont appelées qu'une seule fois par l'étage 2 et ne contiennent qu'une seule instruction.
La fusion des étages 1 et 23 a été rejetée puisque le nombre d'états de l'étage résultant de la fusion était très élevé (environ 1246). Ceci est dû essentiellement au fait que la procédure "chop-start-1" interprétée par l'étage 1, est appelée 24 fois par l'étage 23; d'autre part, cette procédure contient un nombre considérable d'instructions (34 instructions). Si on fusionne les étages 1 et 23, le corps de cette procédure serait dupliqué 24 fois, ce qui provoque une augmentation considérable de la taille de partie contrôle.

Décomposition des microinstructions : Il fut aussi nécessaire de réduire le nombre d'UALs dans la partie opérative en décomposant les instructions opératives contenant au moins 3 opérations exécutées en parallèle. Le nombre de ce type d'instructions étant faible, la compilation du 6502 à partir de la description originelle, conduirait à un circuit avec une partie opérative dont les 3 UALs seraient très peu utilisées simultanément.

Finalement, ces transformations de la description comportementale ont conduit à un circuit dont la partie contrôle est constituée seulement de deux étages de contrôle :

- étage2 : largeur : 3705; hauteur : 1103
- étage1 : largeur : 3741; hauteur : 2299

La partie opérative finale du SYCO6502 possède 2 UALs; ses dimensions sont :
- largeur : 3600
- hauteur : 1200

Par partie opérative, on entend le chemin de données de 8 bits et les amplificateurs de validation répartis sur deux étages sans les plots d'entrée-sortie.

La figure 8 montre le plan de masse du 6502 avec 2 étages de contrôle.

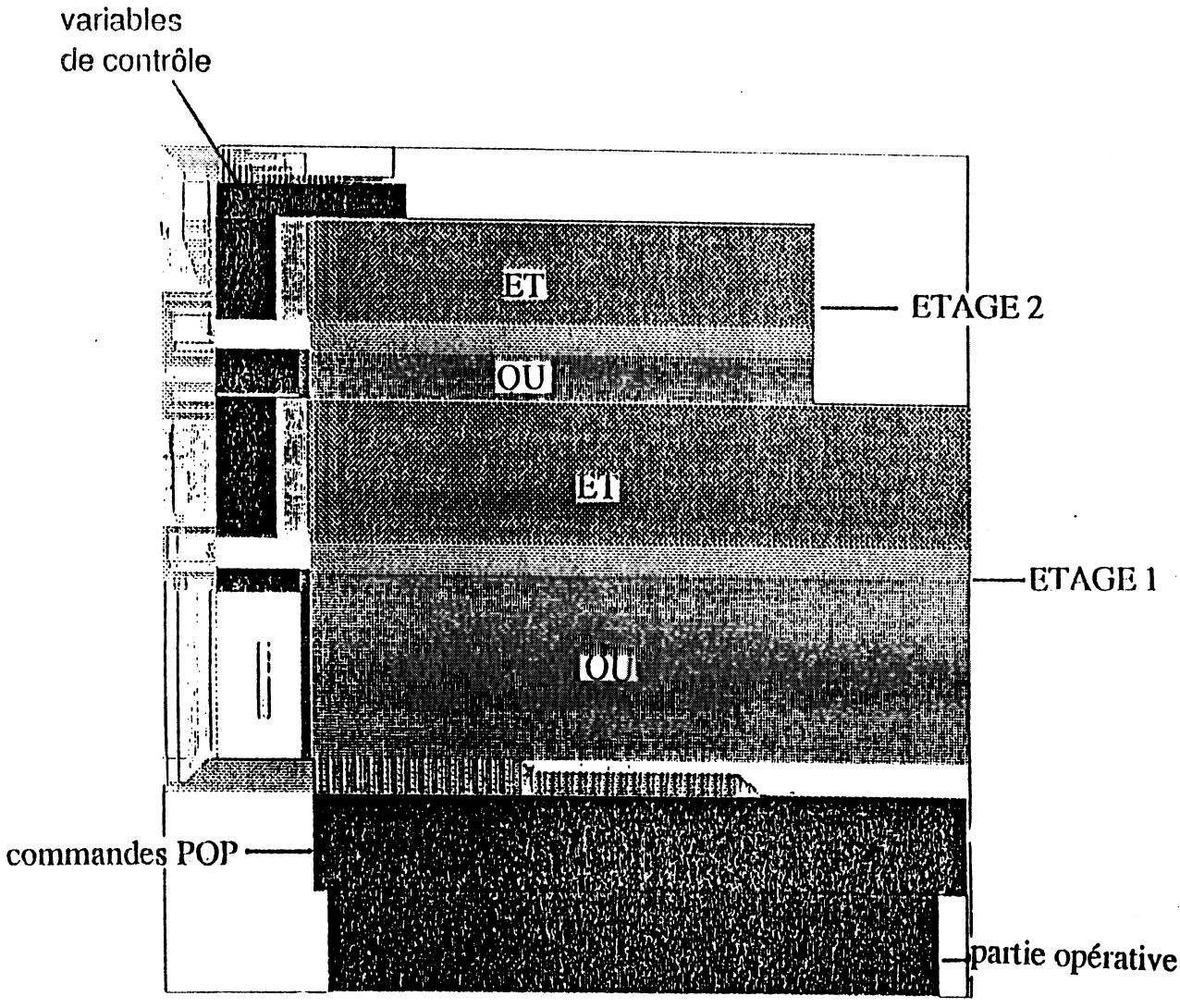


fig. 8 : plan de masse du 6502 avec 2 étages de contrôle

Optimisation du 6502 :

Après cette phase de recherche d'un compromis surface-vitesse, les performances du circuit ont encore été améliorées, en éliminant les états inutiles ou inaccessibles résultant de l'expansion de procédure. Le tableau suivant donne le nombre d'états de chaque étage de contrôle avant, et après la phase d'optimisation :

	nombre d'états avant optimisation	nombre d'états après optimisation	temps CPU (en seconde)
étage 2	224	196	83
étage 1	246	240	30

fig. 9 : réduction du nombre d'états après l'expansion de procédure dans le 6502

ANNEXE II

LE SYSTEME "ARTS"
(ARchitectural Trade-off System)

Le système ARTS est un logiciel interactif écrit en langage LISP et implanté sur machine SM-90 sous système UNIX. Il est constitué d'un ensemble de modules, chaque module réalisant une transformation de la description. Le programme principal permet d'afficher sur l'écran du terminal, un menu constitué des différentes fonctions actuelles du système ARTS. Pour aider le concepteur à choisir une transformation, des fonctions utiles sont présentes dans le menu, telles que l'affichage des tables de transitions des procédures d'un étage de contrôle, le nombre d'états de chacune de ces procédures, le nombre de monômes du PLA implémentant l'étage de contrôle, etc...

Pour lancer le système ARTS, il faut être dans un environnement approprié. Pour cela, il faut :
- charger le compilateur LDS en tapant la commande "lds",
- le compilateur retourne le message "LEDS". A ce niveau, il faut préciser le directory dans lequel se trouvent les fichiers contenant les structures de données LDS correspondant aux tables de transitions des étages de contrôle du circuit. Chacune de ces structures est une liste constituée de plusieurs sous-listes. Chaque sous-liste contient une information concernant l'étage de contrôle. Parmi ces informations on trouve:
- le nom de l'étage,
- le nom de l'étage inférieur,
- la liste des procédures interprétées par cet étage,
- et enfin les corps de ces procédures.

Après avoir précisé le directory, il faut passer dans l'environnement LISP et charger le système ARTS.

Nous donnons dans ce qui suit, un exemple d'utilisation du système ARTS; pour cela, nous avons choisi le microprocesseur MICROP (décrit en Annexe 1) avec 3 étages de contrôle.

```
[Ti]lds
[LEMS] LEMS SYSTEM Logged on Tue Sep 1 09:32:47 EET 1987
[LEMS] . Device Identifier (dku/vt100/tek/...) : tty11
[LEMS] . User Name : Bekkara
[LEDS] [LEDS Version V02]
[LEDS] Session Tue Sep. 1 09:32:42 EET 1987
LEDS : library dirlds/lib.cpc;
-
LEDS : load;
-
LEDS : view memory;

[LEDS] Module(s) in Memory :

[LEDS] --- Smodule ---
     NANOP

[LEDS] --- Cmodule ---
    NANOP_NIV_2   NANOP_NIV_1   NANOP_NIV_0

[LEDS] --- Pmodule ---

[LEDS] --- Fmodule ---

[LEDS] --- Xmodule ---

[LEDS] --- Vmodule ---

-
LEDS:!!;
[LEDS] Lisp Environment : Do It Carefully !! (exit by END)
LISP:^LARTS

-------------------------------------------------------------------------------
          BIENVENUE SUR LE SYSTEME ARTS (Architectural Trade-off System)
-------------------------------------------------------------------------------

LISP:(Arts)

  Donner le nom du circuit sur lequel vous allez travaillez

LISP: NANOP

-------------------------------------------------------------------------------
                              MENU PRINCIPAL :
-------------------------------------------------------------------------------
1: FUSION TOTALE DE DEUX ETAGES DE CONTROLE

2: FUSION PARTIELLE AVEC L'ETAGE SUPERIEUR

3: FUSION PARTIELLE AVEC L'ETAGE INFERIEUR

4: DECOMPOSITION D'UN ETAGE DE CONTROLE
-------------------------------------------------------------------------------
5:*  VOIR UN CMODULE (table de microopérations)
6:*  VOIR UNE PROCEDURE (table de microopérations)
```

```
7:*  LISTE DES PROCEDURES D'UN ETAGE
8:*  NOMBRE DE MONOMES D'UN ETAGE DE CONTROLE
9:*  NOMBRE DE MONOMES D'UNE PROCEDURE
0:*  ARRET

 tapez le numéro de votre choix

LISP:5

 voulez-vous voir le résultat sur écran? (y/n)

LISP:y

 Donnez le nom du CMODULE

LISP: NANOP_NIV_2

*****************************************************************************
*                Table de microopérations du CMODULE NANOP_NIV_2                *
*****************************************************************************

? ---------- Module NANOP_NIV_2
CMODULE NANOP_NIV_2 ;
< MICROP_rest_1 >
( IF  ( restart 0:1 = ( 0 ) ) NEXT MICROP_rest_1  ;
 ENDIF ;
 IF  ( restart 0:1 = ( BIN'1' ) ) NEXT MICROP_start_1  ;
 ENDIF ;
 )
< MICROP_start_1 >
( EXECUTE NANOP_NIV_1 (MICROP_start_1) ;
NEXT MICROP_newinstr_1  ;
 )
< MICROP_newinstr_1 >
( r 0:2 := 0 ;
EXECUTE NANOP_NIV_1 (MICROP_newinstr_1) ;
NEXT MICROP_decode_1  ;
 )
< MICROP_decode_1 >
( IF  ( ir 0:4 = ( BIN'0110' ) ) EXECUTE NANOP_NIV_1 (bru_g169_1) ;
EXIT ;
 ENDIF ;
 IF  ( ir 0:4 = ( BIN'0101' ) ) EXECUTE NANOP_NIV_1 (bneg_g170_1) ;
EXIT ;
 ENDIF ;
 IF  ( ir 0:4 = ( BIN'0010' ) ) EXECUTE NANOP_NIV_1 (sta_g168_1) ;
EXIT ;
 ENDIF ;
 IF  ( ir 0:4 = ( BIN'0001' ) ) EXECUTE NANOP_NIV_1 (lda_g167_1) ;
EXIT ;
 ENDIF ;
 IF  ( ir 0:4 = ( BIN'0000' ) ) EXECUTE NANOP_NIV_1 (ada_g166_1) ;
EXIT ;
 ENDIF ;
 IF  ( restart 0:1 = ( 0 ) ) NEXT MICROP_rest_1  ;
 ENDIF ;
```

```
IF  ( restart 0:1 = ( BIN'1' ) ) NEXT MICROP_newinstr_1  ;
ENDIF ;
)
END ;
? ---------- End of Module NANOP_NIV_2
```

Voulez-vous continuer? (y/n)

LISP:y

MENU PRINCIPAL :

1: FUSION TOTALE DE DEUX ETAGES DE CONTROLE

2: FUSION PARTIELLE AVEC L'ETAGE SUPERIEUR

3: FUSION PARTIELLE AVEC L'ETAGE INFERIEUR

4: DECOMPOSITION D'UN ETAGE DE CONTROLE

5:* VOIR UN CMODULE (table de microoperations)
6:* VOIR UNE PROCEDURE (table de microoperations)
7:* LISTE DES PROCEDURES D'UN ETAGE
8:* NOMBRE DE MONOMES D'UN ETAGE DE CONTROLE
9:* NOMBRE DE MONOMES D'UNE PROCEDURE
0:* ARRET

tapez le numéro de votre choix

LISP:5

voulez-vous voir le résultat sur écran ? (y/n)

LISP:y

Donnez le nom du CMODULE

LISP:NANOP_NIV_1

```
**********************************************************************
*                Table de microopérations du CMODULE NANOP_NIV_1     *
**********************************************************************

? ---------- Module NANOP_NIV_1
CMODULE NANOP_NIV_1 ;
< MICROP_start_1 >
( EXECUTE NANOP_NIV_0 (MICROP_start_1) ;
EXIT ;
)
< MICROP_newinstr_1 >
( EXECUTE NANOP_NIV_0 (fetch_send_1) ;
EXIT ;
)
```

```
< bru_g169_1 >
( r 0:2 := 2 ;
EXECUTE NANOP_NIV_0 (fetch_send_1) ;
NEXT bru_g169_3 ;
)
< bru_g169_3 >
( EXECUTE NANOP_NIV_0 (bru_g169_3) ;
EXIT ;
)
< bneg_g170_1 >
( r 0:2 := 2 ;
EXECUTE NANOP_NIV_0 (fetch_send_1) ;
NEXT bneg_g170_3 ;
)
< bneg_g170_3 >
( EXECUTE NANOP_NIV_0 (bneg_g170_3) ;
EXIT ;
)
< sta_g168_1 >
( r 0:2 := 2 ;
EXECUTE NANOP_NIV_0 (fetch_send_1) ;
NEXT sta_g168_3 ;
)
< sta_g168_3 >
( read_write 0:1 := 0 ;
EXECUTE NANOP_NIV_0 (sta_g168_3) ;
NEXT sta_g168_5 ;
)
< sta_g168_5 >
( adstrobe 0:1 := 1 ;
dtstrobe 0:1 := 1 ;
EXECUTE NANOP_NIV_0 (sta_g168_5) ;
NEXT sta_ack_1 ;
)
< sta_ack_1 >
( IF ( dtack 0:1 = ( BIN'1' ) ) adstrobe 0:1 := 0 ;
dtstrobe 0:1 := 0 ;
EXIT ;
ENDIF ;
IF ( dtack 0:1 = ( 0 ) ) NEXT sta_ack_1 ;
ENDIF ;
EXECUTE NANOP_NIV_0 (sta_ack_1) ;
)
< lda_g167_1 >
( r 0:2 := 1 ;
EXECUTE NANOP_NIV_0 (fetch_send_1) ;
EXIT ;
)
< ada_g166_1 >
( r 0:2 := 2 ;
EXECUTE NANOP_NIV_0 (fetch_send_1) ;
NEXT ada_g166_3 ;
)
```

```
< ada_g166_3 >
( EXECUTE NANOP_NIV_0 (ada_g166_3) ;
EXIT ;
)
END ;
? ---------- End of Module NANOP_NIV_1
```

Voulez-vous continuer ? (y/n)

LISP:y

MENU PRINCIPAL :

1: FUSION TOTALE DE DEUX ETAGES DE CONTROLE

2: FUSION PARTIELLE AVEC L'ETAGE SUPERIEUR

3: FUSION PARTIELLE AVEC L'ETAGE INFERIEUR

4: DECOMPOSITION D'UN ETAGE DE CONTROLE

5:* VOIR UN CMODULE (table de microoperations)
6:* VOIR UNE PROCEDURE (table de microoperations)
7:* LISTE DES PROCEDURES D'UN ETAGE
8:* NOMBRE DE MONOMES D'UN ETAGE DE CONTROLE
9:* NOMBRE DE MONOMES D'UNE PROCEDURE
0:* ARRET

tapez le numéro de votre choix

LISP:1

étage de contrôle : nombre de monômes
NANOP_NIV_2 : 11
NANOP_NIV_1 : 15
NANOP_NIV_0 : 16

Donnez le nom du CMODULE supérieur
LISP:NANOP_NIV_2
Donnez le nom du CMODULE inférieur
LISP:NANOP_NIV_1

```
**********************************************************************
*          Fusion des étages de contrôle NANOP_NIV_2  et NANOP_NIV_1          *
**********************************************************************
```

Voulez-vous voir le résultat sur écran? (y/n)

LISP:y

---------- RESULTAT DE LA FUSION TOTALE ----------
NOMBRE DE MONOMES DE L'ETAGE DE CONTROLE APPELANT : 11

NOMBRE DE MONOMES DE L'ETAGE DE CONTROLE APPELE : 15

NOMBRE DE MONOMES DE L'ETAGE DE CONTROLE RESULTANT DE LA FUSION (E.R.F) : 24

---------- TEMPS CPU : .6399994 ----------

Voulez-vous voir le gain de vitesse pour chaque procédure de l'étage supérieur? (y/n)

LISP:y
procédure MICROP_rest_1 : 2 cycles

Voulez-vous réellement réaliser la fusion ? (y/n)

LISP:y

* ATTENTION * : Vous ne pourrez plus utiliser le CMODULE NANOP_NIV_1.

```
**********************************************************************
*              Table de microopérations du CMODULE résultant de la fusion            *
**********************************************************************

? ---------- Module NANOP_NIV_2
CMODULE NANOP_NIV_2 ;
< MICROP_rest_1 >
( IF  ( restart 0:1 = ( 0 ) ) NEXT MICROP_rest_1  ;
 ENDIF ;
 IF  ( restart 0:1 = ( BIN'1' ) ) NEXT MICROP_start_1  ;
 ENDIF ;
 )
< MICROP_start_1 >
( EXECUTE NANOP_NIV_0 (MICROP_start_1) ;
NEXT MICROP_newinstr_1  ;
 )
< MICROP_newinstr_1 >
( r 0:2 := 0 ;,
EXECUTE NANOP_NIV_0 (fetch_send_1) ;
NEXT MICROP_decode_1  ;
 )
< MICROP_decode_1 >
( IF  ( ir 0:4 = ( BIN'0110' ) ) NEXT bru_g169_13  ;
 ENDIF ;
 IF  ( ir 0:4 = ( BIN'0101' ) ) NEXT bneg_g170_14  ;
 ENDIF ;
 IF  ( ir 0:4 = ( BIN'0010' ) ) NEXT sta_g168_15  ;
 ENDIF ;
 IF  ( ir 0:4 = ( BIN'0001' ) ) NEXT lda_g167_16  ;
 ENDIF ;
 IF  ( ir 0:4 = ( BIN'0000' ) ) NEXT ada_g166_17  ;
 ENDIF ;
 IF  ( restart 0:1 = ( 0 ) ) NEXT MICROP_rest_1  ;
 ENDIF ;
 IF  ( restart 0:1 = ( BIN'1' ) ) NEXT MICROP_newinstr_1  ;
 ENDIF ;
 )
< bru_g169_13 >
( r 0:2 := 2 ;
```

```
EXECUTE NANOP_NIV_0 (fetch_send_1) ;
NEXT bru_g169_33 ;
)
< bru_g169_33 >
( EXECUTE NANOP_NIV_0 (bru_g169_3) ;
EXIT ;
)
< bneg_g170_14 >
( r 0:2 := 2 ;
EXECUTE NANOP_NIV_0 (fetch_send_1) ;
NEXT bneg_g170_34 ;
)
< bneg_g170_34 >
( EXECUTE NANOP_NIV_0 (bneg_g170_3) ;
EXIT ;
)
< sta_g168_15 >
( r 0:2 := 2 ;
EXECUTE NANOP_NIV_0 (fetch_send_1) ;
NEXT sta_g168_35 ;
)
< sta_g168_35 >
( read_write 0:1 := 0 ;
EXECUTE NANOP_NIV_0 (sta_g168_3) ;
NEXT sta_g168_55 ;
)
< sta_g168_55 >
( adstrobe 0:1 := 1 ;
dtstrobe 0:1 := 1 ;
EXECUTE NANOP_NIV_0 (sta_g168_5) ;
NEXT sta_ack_15 ;
)
< sta_ack_15 >
( IF ( dtack 0:1 = ( BIN'1' ) ) adstrobe 0:1 := 0 ;
dtstrobe 0:1 := 0 ;
EXIT ;
ENDIF ;
IF ( dtack 0:1 = ( 0 ) ) NEXT sta_ack_15 ;
ENDIF ;
EXECUTE NANOP_NIV_0 (sta_ack_1) ;
)
< lda_g167_16 >
( r 0:2 := 1 ;
EXECUTE NANOP_NIV_0 (fetch_send_1) ;
EXIT ;
)
< ada_g166_17 >
( r 0:2 := 2 ;
EXECUTE NANOP_NIV_0 (fetch_send_1) ;
NEXT ada_g166_37 ;
)
```

```
< ada_g166_37 >
( EXECUTE NANOP_NIV_0 (ada_g166_3) ;
EXIT ;
)
END ;

? ---------- End of Module NANOP_NIV_2
```

Voulez-vous continuer? (y/n)

LISP: n
Au revoir et à bientôt
LISP:

REFERENCES

[AGER 76] **T. Agerwala,** "Microprogram optimization : A survey", IEEE, Trans. on Comp. Vol C-25, No 10 Oct. 1976.

[AHO 77] **A. V. Aho, and J. D. Ullmann,** "Principles of compiler design" Addison Wesley, 1977.

[ALLE 80] **F.E. Allen, J.L. Carter, J. Fabri,** "The experimental Compiling System", IBM J. Res. Develop. 24, 695-715, 1980.

[ALLE 81] **F.E Allen,** "The history of langage processor technology", IBM J. Res, Develop. Vol 25 Sept. 1981.

[ANCE 82] **F. Anceau,** "Architecture des ordinateurs : exemple de conception d'un petit microprocesseur", cours ENSIMAG, Juin 1982.

[ANCE 83] **F. Anceau,** "Capri : a design methodology and a silicon compiler for VLSI circuits specified by algorithms", 3rd Caltech Conference on VLSI, March 1983.

[ANCE 84] **F. Anceau,** "Silicon compilation for microprocessor-like VLSI", NATO advanced study institute on micro-architecture of VLSI Computers, Urbino, Italy, July 1984.

[AUSL 82] **M. Auslender and M. Hopkins,** "An overview of the PL.8 compiler", Proceedings of the SIGPLAN' 82 Symposium on Compiler Construction.

[BARA 79] **S. Baranov,** "Synthèse des automates microprogrammés" , Editions de Moscou, 1979

[BARB 81] **M.R. Barbacci,** "Syntax and semantics of CHDLs", Computer hardware description languages and their applications, IFIP, 1981.

[BCHJ 87] **Ph. Bondono, J.P. Caisso, A. Hornik, A.A. Jerraya,** "NAUTILE : New Automatic and Techno Independant Layout Environment. Rapport interne, TIM3-46 avenue Félix Viallet-Grenoble, Juin 87.

[BERG 84] **N. Bergman,** "A case study of the FIRST silicon compiler", Dep of Computer Science, U. of Edinburgh, Internal report, February 1984.

[BLAC 85] **T. Blackman, J. Fox, C. Rosebrugh**, "The Silc silicon compiler language and features", 22nd DAC, 1985.

[BONN 81] **M. J. Bonnet** "Calcul des temps de propagation dans les PLAs", Rapport de DEA, ENSERG, Septembre 1981.

[BOUR 86] **E. Bourcier**, "LDS : langage d'entrée du compilateur de silicium SYCO", rapport interne, IMAG/TIM3, 46 av. Félix Viallet, 38031 Grenoble Cédex, avril 1986.

[BRAY 85a] **R.K. Brayton et al.**, "The Yorktown silicon compiler", ISCAS'85, Kyoto, Japan, June 1985.

[BRAY 85b] **R.K. Brayton et al.**, "A microprocessor design using the Yorktown silicon compiler", ICCD'85.

[BREW 86] **F.D. Brewer, D.D. Gajski**, "An expert system paradigm for design", 23rd DAC, 1986.

[CAGN 85] **A. Cagnola, M. Corti, G. Vignati**, "LAPLACE : Another second generation PLA design tool" Microprocessing and Microprogramming 16 (1985) pp. 61-66.

[CHUQ 84] **S. Chuquillanqui Bernaola**, "Une nouvelle approche pour l'optimisation topologique et l'automatisation du dessin des masques de PLA complexes", thèse de docteur ingénieur INPG, octobre 1984.

[DAND 83] **A. Dandache**, "Evaluations électriques et temporelles des PLAs complexes" thèse de docteur de 3è cycle (INPG -1983).

[DAVI 80] **M. Davio and A. Thayse**, "Implementation and transformation of algorithms based on automata", Philips J. Res. 35, 122-144, 1980.

[DAVI 81] **S. Davidson**, "Some experiments in local microcode compaction for horizontal machines", IEEE transactions on computers, Vol C-30, n°7, July 1981.

[DEMA 86] **H. De Man**, "A synthesis and module generation system for multiprocessor systems on a chip", Nato advanced study institute on logic synthesis and silicon compilation for VLSI design, L'Aquila, Italy, July 7-18, 1986.

[DIRE 81] **S.W. Director**, "A design methodology and computer aids for digital VLSI systems", IEEE transactions on circuits and systems, VOL. CAS-28, N°7, July 1981.

[EGAN 84] **J.R. Egan and C.L. Liu**, "Bipartite folding and partitionning of a PLA", IEEE. Transactions on Computer aided design, vol. CAD-3, N°3 Juillet 84.

[EVAN 85] **S. Evanczuk**, "Results of a silicon compiler design challenge", VLSI design, July 1985.

[FISH 81] **J.A. Fisher**, "Trace scheduling : a technique for global microcode compaction", IEEE transactions on computers, Vol C-30, n°7, July 1981.

[GAJS 86] **D. D. Gajski et al**, "Silicon cell compilers", Nato advanced study institute on logic synthesis and silicon compilation for VLSI design, L'Aquila, Italy, July 7-18, 1986.

[GERO 85] **J.P. Geronimi, S. Ringot, D. Savart**, "Conception d'un microprocesseur à l'aide d'outils de CAO de haut niveau", rapport de stage ENSERG, IMAG/TIM3, 46 Av. Félix Viallet, 38031 Grenoble Cédex, 1985.

[GERO 87] **J.P. Geronimi, A.A. Jerraya**, "Architecture interne des parties contrôles générées par SYCO", rapport interne, IMAG/TIM3, 46 Av. Félix Viallet, 38031 Grenoble Cedex, Aout 1987.

[GIO 83] **G. De Micheli and M. Santomauro** "Topological partitionning of programmable logic array", ICCAD 83.

[GIRC 84] **E.F. Girczyc, J.P. Knight**, "An ada to standard cell hardware compiler based on graph grammars and scheduling", ICCD'84.

[GLUS 65] **V.M. Glushkov**, "Automata theory and structural design problems of digital machines", Kibernetika, Vol 1 n°1 pp 3-11, 1965.

[GROS 83] **R.R. Gross**, "Silicon compilers : a critical survey", Dep. of Computer Science, University of North Carolina at Chapel Hill, May 1983. DAC, 1981.

[HARB 82] **S. P. harbison,** "An architectural alternative to optimizing compilers" ACM, Mars 1982.

[HENN 83] **J. L. Hennessy,** "Partitionning programmable logic array summary" ICCAD 83.

[HENN 84] **J. L. Hennessy,** "VLSI Processor Architecture" IEEE Trans. on Comp., Vol.C-33, N°12, Decembre 1984.

[ISOD 83] **S. Isoda,** "Global compaction of horizontal microprograms based on the generalized data dependency graph", IEEE transactions on computers, Vol C-32, n°10, October 1983.

[JAMI 85] **R. Jamier, A. Jerraya,** "Apollon, a datapath silicon compiler", ICCD'85.

[JAMI 86] **R. Jamier, N. Bekkara, A. Jerraya,** "The automatic synthesis of data processing sections", ICCD'86.

[JERR 86] **A. Jerraya, P. Varinot, R. Jamier, B. Courtois** "Principles of the SYCO compiler" 23rd DAC. June1986.

[JOHA 79] **D. Johannsen,** "Bristle blocks : a silicon compiler", 16th DAC, 1979.

[KAHR 85] **M. Kahrs,** "An overview of SILI (a SILIcon compiler)" 22rd DAC, 1985.

[KNAP 86] **D. W. Knapp, A.C. Parker,** "A design utility manager : the ADAM planning engine", 23rd DAC, 1986.

[KRIS 82] **P. Krishna , D. S. Fussel and A. J. Welch,** "High-level optimization in a silicon compiler", TR-215 Nov. 1982 Departement of computer sciences, The university of Texas at Austin. Austin, TX 78 712.

[KUCK 81] **D. J. Kuck, R. H. Kuhn, D. A. Padua,** "Dependence graphs and compiler optimizations" Oct. 1981, J. of the ACM.

[LAND 80] **D. Landskov et al,** "Local microcode compaction techniques", ACM Comput. Surveys, Vol.12, pp. 261-294, September 1980.

[LAUR 85] **D. Laurent, H.N. Nguyen,** "LDS", rapport Bull, Juin 1985.

[MALA 85] **Y. K. Malaiya**, "Options in control implementation", ICCD'85.

[MARI 86] **S. Marine**, "Irène : un langage de description, simulation et synthèse automatique du matériel VLSI", thèse de docteur INPG, février 1986.

[MARS 86] **T. Marshburn et al.**, "Datapath : a CMOS Data Path silicon assembler", 23rd DAC, 1986.

[MART 85] **F. Martinez**, "Compilateur du langage Irène", thèse CNAM, Novembre 1985.

[MEAD 80] **C. Mead, L. Conway**, "Introduction to VLSI systems", Addison-Wesley, 1980.

[MHAY 86] **N. Mhaya**, "Compilation de parties contrôle", rapport de DEA, IMAG/TIM3, 46 av. Félix Viallet, 38031 Grenoble Cédex, juillet 1986.

[NAGL 82] **A. Nagle et al**, "Synthesis of hardware for the control of digital systems", IEEE transactions on CAD, Vol CAD-1, N°4, october 1982.

[NGUY 85] **M.N. Nguyen, D. Laurent**, "LDS : un langage de description de système", Juin 1985, document interne , projet SYCOMORE, IMAG/TIM3, 46 Av. Félix Viallet, 38031 Grenoble Cedex.

[NIC 84] **A. Nicolau, and J. A. Fisher**, "Measuring the parallelism available for very long instructions word architectures", IEEE, Trans. on Computers, Vol. C-33, No11, November 84.

[OBRE 82] **M. Obrebska**, "Etude comparative de différentes méthodes de conception des parties contrôle de microprocesseurs", Thèse INPG, Juin 1982.

[ORAI 86] **A. Orailoglu and D. Gajski**, "Flow graph representation" 23nd Design Automation Conference, juin 1986.

[PAILL 85] **J.F. Paillotin**, "Le système LUCIE", rapport interne, IMAG/TIM3, 46,av. Félix Viallet, 38031 Grenoble Cédex, Juillet 1985.

[PARK 79] **A. Parker, D. Thomas, D. Siewiorek,** The CMU-DA Design Automation System : An example of Automated Data Path Design" Proceedings of The 16th Design Automation Conference, ACM/IEEE June 1979.

[PARK 85] **N. Park, A. Parker,** "Synthesis of optimal clocking schemes", 22nd DAC, 1985.

[PARK 86] **N. Park, A. Parker,** "Sehwa : a program for synthesis of pipelines", 23rd DAC, 1986.

[RAJA 85] **J.V. Rajan, D.E. Thomas,** "Synthesis by delayed binding of decisions", 22nd DAC, 1985.

[REIS 83] **R.A. Da Luz Reis,** "TESS: Evaluateur Topologique Prédictif pour la Génération Automatique des Plans de Masse de Circuit VLSI", Thèse de docteur-ingénieur INP de Grenoble option Informatique, 1983.

[REIS 86] **R.Reis, A.Jerraya, R. Jamier,** "Design of the SYCO 6502 using the Syco compiler", ICCD'86.

[SAGN 86] **G. Sagnes, T. Ezzedine, J. Lassale,** "Hierarchical algorithmic description for complex control parts design", ICTC (Integrated Circuits Technology Conference), Sept. 86, Limerick, Ireland.

[SCHE 77] **R. W. Scheiffer,** "An analysis of inline substitution for a structured programming language". CACM, 20, 9, Sept. 77, p. 647-654.

[SCHO 83] **J.P. Schoellkopf,** "Lubrick : a silicon assembler and its application to data-path design for FISC", VLSI'83.

[SCHO 85] **J.P. Schoellkopf,** "Contributions à l'architecture des circuits intégrés et à la compilation de silicium" Thèse d'Etat INPG, Mars 1985.

[SHRO 82] **H.E. Shrobe,** " The Data Path Generator", 1982 Conference on advanced research in VLSI, MIT.

[SISK 82] **J.M. Siskind, J.R. Southard, K.W. Crouch,** "Generating custom high performance VLSI designs from succinct algorithmic descriptions", Conference on advanced research in VLSI, M.I.T., 1982.

[SNOW 78] **E. A. Snow, D. P. Siewiorek, and D. E. Thomas,** "A technology relative computer-aided design system : abstract representation, transformation and design tradeoffs", Proc. 15th Design Automation Conf., pp. 220-226, Juin 1978.

[SOUT 83] **J.R. Southard,** "Macpitts : an approach to silicon compilation", IEEE computer, December 1983.

[SUZI 81] **A.A.Suzim,** "Etude des parties opératives à éléments modulaires pour processeurs monolithiques", Thèse Docteur Ingénieur, INPG, novembre 1981.

[TADJ 70] **G. S. Tadjen, and M. J. Flyinn,** "Detection and parallel execution of independent instructions" IEEE, Trans. on Computers, Vol C-19, No 10 Oct. 1970.

[TOKO 81] **M. Tokoro,** "Optimization of microprograms", IEEE transactions on computers, Vol C-30, n°7, July 1981.

[TSEN 83] **C.J. Tseng, D.P. Siewiorek,** "Facet : a procedure for the automated synthesis of digital systems", 20th DAC, 1983.

[VARI 86] **P. Varinot,** "Compilation de silicium : application à la génération automatique de parties contrôle", thèse INPG, Decembre 1986.

[WALK 83] **R. A. Walker, D. E. Thomas,** "Behavioral level transformation in the CMU-DA system", 20th DAC, 1983.

[WALL 83] **P. Wallich,** 'Technology 83: Automation, Design Manufacturing', IEEE Spectrum 20, 1983.

[WDC 83] **Western Design Center,** "The W65C02 8-bit microprocessor data sheet", november 1983.

[WERN 82] **J. Werner,** "A silicon compiler : Panorama, Wishful thinking, or old fat", VLSI Design, 1982.

[YASH 85] **K. M. Yashwani,** "Options in control implementation", IEEE International Conference on Computer Design : VLSI in computer, ICCD 85 Oct.7-10, pp. 267-272.

[YAU 74] **S.S. Yau et al.,** "On storage optimization of horizontal microprograms", in Proc. 7th Annu. Workshop Microprogramming, 1974, pp. 98-106.

[ZIMM 80] **G. Zimmerman,** "MDS : The Mimola Design System", journal of digital system, Vol. 4, N°3 , pp. 337-369, 1980.

TABLE DES MATIERES

Liste des figures

Chapitre 1 : nº de page :

Chapitre 2 :

Chapitre 3 :

Chapitre 4 :

Annexe 1 :

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 3 de l'arrêté du 16 avril 1974

VU les rapports de présentation de Messieurs

. B. COURTOIS, Directeur de recherche
. G. SAGNES, Professeur

Monsieur BEKKARA Nouroudine Ahmed

est autorisé à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR-INGENIEUR, spécialité "Informatique".

Fait à Grenoble, le 6 octobre 1987

Georges LESPINARD
Président
de l'Institut National Polytechnique
de Grenoble

P.O. le Vice-Président,

Résumé

Le but de cette thèse est l'élaboration d'une phase d'optimisation de haut niveau et de recherche d'un compromis surface-vitesse dans le compilateur de silicium SYCO.
Le système SYCO développé dans le groupe architecture des ordinateurs du laboratoire IMAG/TIM3 est un compilateur de silicium pour des circuits intégrés de type microprocesseur. SYCO génère les spécifications des masques d'un circuit constitué d'une partie opérative et d'une partie contrôle à partir de sa description comportementale.
L'optimisation consiste à détecter et à fusionner les instructions qui peuvent être exécutées en parallèle.
La recherche d'un compromis surface-vitesse consiste à échanger une perte de performance du circuit contre un gain de surface ou vice-versa. Un ensemble de transformations qui peuvent être appliquées à la description afin d'aboutir à une meilleure implantation du circuit sans changer son comportement original sont présentées.
Le résultat de cette étude se concrétise par la réalisation d'un ensemble de logiciels écrits en langage LISP, chacun de ces logiciels réalisant une transformation de la description. Pour évaluer l'utilité de ces transformations, deux exemples de microprocesseurs ont été choisis et éxaminés.

Mots clés : Compilateur de silicium, graphes, optimisation de haut niveau, compromis surface-vitesse, parallélisme, compactage de microcode, réduction du nombre d'états.